Horváth Péter

Digitális rendszerek modellezése és szintézise

Gyakorlati RTL-tervezés

L'Harmattan

Ez a mű a Creative Commons Nevezd meg! - Ne add el! - Így add tovább! 4.0 Nemzetközi Licenc feltételeinek megfelelően felhasználható.
https://creativecommons.org/licenses/by-nc-sa/4.0/

Digitális rendszerek modellezése és szintézise

Modelling and Synthesis of Digital Systems

Horváth Péter

https://orcid.org/0009-0006-0097-259X

Villamosmérnöki és informatikai tudományok / Electrical engineering, Electronic engineering, Information engineering (10880), Beágyazott rendszerek / Embedded systems (10124)

digitális rendszertervezés, RTL-tervezés, HDL-modellezés, funkcionális verifikáció

digital system design, RTL design, HDL modelling, functional verification

ISBN 978-963-646-445-5 (pdf)

DOI: https://doi.org/10.56037/978-963-646-445-5

https://openaccess.hu/

Horváth Péter

DIGITÁLIS RENDSZEREK MODELLEZÉSE ÉS SZINTÉZISE

Gyakorlati RTL-tervezés

Horváth Péter

DIGITÁLIS RENDSZEREK MODELLEZÉSE ÉS SZINTÉZISE

Gyakorlati RTL-tervezés

L'Harmattan, 2025

A kötet megjelenését támogatták:

Ericsson Magyarország Kft.

BME VIK, Elektronikus Eszközök Tanszéke

Pro Progressio Alapítvány

C3S Elektronikai Fejlesztő Kft.

© Horváth Péter, 2025
© L'Harmattan, 2025

ISBN 978-963-646-444-8
ISBN 978-963-646-445-5 (pdf)
DOI: https://doi.org/10.56037/978-963-646-445-5

TARTALOM

Ajánló ix
A szerző előszava xi
Köszönetnyilvánítás xiii

PROBLÉMAFELVETÉS ÉS RENDSZERTERV 15

1.1. Követelmények 16
1.2. Rendszerszintű modell, rendszerterv 18
1.2.1. Almodulok közötti adatcsere 20
1.2.1.1. Pont–pont kapcsolatok 21
1.2.1.2. Az MMIO-buszrendszer 24
1.2.1.3. Órajeltartományok 28
1.2.2. CPU – Mikroarchitektúra és utasításkészlet 29
1.2.3. A funkcióbővítő rendszer toplevel architektúrája 32
1.2.3.1. A programtárak inicializálása 32
1.2.3.2. Megszakításvezérlő 35
1.2.3.3. Memóriarendszer 36
1.2.3.4. Lokális adatgyűjtés, időzítők 37
1.2.3.5. Az üzemeltetői interfész áramkörei 39
1.2.3.6. Hibakeresést segítő modulok 41
1.2.3.7. A bővítőmodulok integrációja 42

AZ ALAPRENDSZER RTL-TERVEZÉSE 45

2.1. Az órajel-szinkronizáló fokozatok elhelyezése 46
2.2. A CPU részletes tervezése 53
2.2.1. Viselkedési modell 53
2.2.2. Műveletvégző egység 59
2.2.3. Vezérlőegység 73
2.2.4. A CPU órajel-szinkronizáló fokozatai 86
2.3. A programtárak inicializálása 91
2.4. Az MMIO periféria-áramkörök szerkezete 111
2.5. Megszakításvezérlő 116
2.6. Memóriarendszer 120
2.6.1. On-chip statikus RAM mint operatív tár 120
2.6.2. Külső szinkron dinamikus RAM mint operatív tár 123
2.7. Lokális adatgyűjtés, időzítők 132
2.7.1. Adatgyűjtő modul 132
2.7.2. Időzítők 141

2.8. Az üzemeltetői interfész áramkörei 144
2.8.1. UART-vezérlő 144
2.8.2. GPIO-vezérlő 148
2.8.3. SPI-FIFO útválasztó 154
2.9. Hibakeresést segítő modulok 165

A BŐVÍTŐMODULOK RTL-TERVEZÉSE 169

3.1. Hullámforma-generátor 170
3.2. DSP-bővítőmodul – szűrő és spektrumanalizátor 178
3.2.1. Követelmények és rendszerterv 178
3.2.2. RTL-modell 181
3.3. Lebegőpontos aritmetikai bővítőmodul 200

IP-KATALÓGUS 203

4.1. Segédmodulok 203
4.1.1. Órajel-előosztó 203
4.1.2. Prioritásos hozzáférés-vezérlő 207
4.2. On-chip adattárolás 214
4.2.1. Véletlen elérésű tárolómodulok (RAM-ok) 215
4.2.2. FIFO-tároló 218
4.2.3. Memória IP-k kézfogásos burkolói 219
4.2.4. CDC a tároló IP-magok interfészein 221
4.2.5. CDC dual-clock blokk-RAM modulokkal 222
4.2.6. A memóriatartalom inicializálása 225
4.3. A soros kommunikáció alapáramkörei 229
4.3.1. UART-adó és -vevő 229
4.3.1.1. Órajel-szinkronizálás 236
4.3.2. SPI master és slave 241
4.3.2.1. Órajel-szinkronizálás 249
4.3.3. I²C master 253
4.3.3.1. Órajel-szinkronizálás 264
4.3.4. Nagysebességű UART-vevő 267
4.3.5. Nagysebességű SPI slave 273
4.4. Katalógus-áramkörök interfészadapterei 286
4.4.1. SPI EEPROM 288
4.4.2. I²C EEPROM 296
4.4.3. SDRAM 302
4.4.3.1. Időzítési peremfeltételek 314
4.4.4. I²C A/D-átalakító 316
4.4.4. I²C hőmérséklet-érzékelő 324
4.4.5. I²C RTCC áramkör 327

4.4.6. SPI A/D-átalakító ... 332
4.4.8. SPI D/A-átalakító ... 334
4.4.9. Órajel-szinkronizálás ... 336
4.5. Aritmetika és jelfeldolgozás ... 339
4.5.1. Fixpontos aritmetikai egység ... 340
4.5.1.1. Komplex aritmetika ... 350
4.5.2. Iteratív fixpontos szorzó ... 353
4.5.2.1. Viselkedési modell ... 355
4.5.2.2. RTL-modell ... 356
4.5.2.3. Órajel-szinkronizálás ... 360
4.5.3. Lebegőpontos aritmetikai egység ... 360
4.5.3.1. Viselkedési modell ... 361
4.5.3.2. Műveletvégző egység ... 368
4.5.3.3. Vezérlőegység ... 381
4.5.3.4. Órajel-szinkronizálás ... 394
4.5.4. CRC-kódgenerálás visszacsatolt léptetőregiszterrel ... 394
4.5.4.1. Órajel-szinkronizálás ... 399
4.5.5. FFT ... 399
4.5.5.1. Viselkedési modell ... 399
4.5.5.2. Műveletvégző egység ... 404
4.5.5.3. Vezérlőegység ... 408
4.5.5.4. Órajel-szinkronizálás ... 417
4.5.6. FIR-szűrő ... 417
4.5.6.1. Műveletvégző egység ... 420
4.5.6.2. Vezérlőegység ... 422
4.5.6.3. Órajel-szinkronizálás ... 424
4.5.7. RC aluláteresztő szűrő ... 425
4.5.7.1. Órajel-szinkronizálás ... 428

FÜGGELÉK 431

5.1. Kellékek a CPU assembly nyelvű programozásához ... 431
5.1.1. Utasításkészlet ... 431
5.1.1.1. Natív (hardver) utasítások ... 431
5.1.1.2. Pszeudoutasítások ... 441
5.1.2. Javasolt regiszterhasználati konvenció ... 444
5.1.3. Központi megszakításkiszolgáló rutin ... 445
5.2. Egyszerű soros kommunikációs interfészek ... 447
5.2.1. UART ... 448
5.2.2. SPI ... 450
5.2.3. I²C ... 452
5.3. Bináris számábrázolás ... 456
5.3.1. Fixpontos számok ... 456

5.3.1.1. A fixpontos számábrázolás gyakorlati kérdései 459
5.3.2. Lebegőpontos számok 465
5.3.2.1. Az *flp_exception_codec* modul VHDL-modellje 468
5.4. A ciklikus redundanciaellenőrzés elve 471
5.5. Gray-enkóder és -dekóder 475
5.6. Az FFT-algoritmus 476
5.7. Az elsőfokú RC aluláteresztő szűrő diszkretizálása 481
5.8. Feladatok 483
5.8.1. A bemutatott részegységek továbbfejlesztése 483
5.8.2. Új bővítőmodulok tervezése 484

Rövidítésjegyzék 488

Ajánló

A minket körülvevő világban aligha találunk olyan területet, ahol valamilyen elektronikus eszközt ne használnánk. A legegyszerűbb érzékelőktől a háztartási gépekig, az okostelefonoktól a járműveken át a szerverfarmokig számtalan integrált áramkör teszi a dolgát, többnyire – és remélhetőleg – a felhasználónak láthatatlan módon. Ki készíti ezeket? Mit tanulnak azok a mérnökök, akik ebben jártasak? Milyen praktikák gyűltek össze a logikai tervezés évtizedeket tekintve nem túlságosan hosszú múltra visszatekintő, de évenként elképesztő fejlődési pályát leíró történetében?

Hiánypótló művet tart a kezében az olvasó, ugyanis bármely fent említett területre készül egy eszköz, a funkcionális tervezés alapjai ugyanazokhoz a gyökerekhez nyúlnak vissza. A megkülönböztetés csak később történik, attól függően, hogy nagyobb számítási teljesítményre, alacsonyabb fogyasztásra, vagy éppen a felhasznált chipméretre optimalizálunk a gyártási költséget leszorítandó. A szerző fantasztikus munkát végzett, bárcsak rendelkezésre állt volna hasonlóan logikusan felépített, teljesnek mondható magyar nyelvű szakirodalom a korábbi mérnökgenerációk számára! Egyetemi jegyzetek gyakran kizárólag az elméletre szorítkoznak, a hallgatókra hagyva a gyakorlati megértést és az elmélet alkalmazásának feltalálását. Ez a könyv más: igényes diagramok, hullám-formák és mintakódok, mindezek szöveges értelmezése váltják egymást. A gondolkodásmódot csiszolja, alakítja, ezáltal túlmutat egy egyetemi jegyzeten. Haszonnal forgathatja hallgató, oktató, gyakorló mérnök, sőt bárki, aki a téma iránt érdeklődik. Ezt a könyvet végigtanulmányozva biztos alapot kap. Olyat, amire építve azon kevesek közé fog tartozni, akik mindennapjaikban a jövő megteremtésén dolgoznak.

Kiss Gergely
okleveles villamosmérnök
Senior Director, Engineering, Arm
Budapest, 2025. április

A szerző előszava

A műszaki témájú oktatási anyagok jellegzetessége, hogy a gyakorlatban felmerülő problémáknak és az azokra adott megoldásoknak csak a jelentősen leegyszerűsített, a megértést esetleg megzavaró tényezőktől megtisztított, „kitenyésztett" verzióit tárják a hallgatóság elé, hiszen a bemutatandó alapelvek így tehetők jól megfigyelhetővé. E méltányolható egyszerűsítő szándék következtében azonban a bemutatott példák annyira eltávolodhatnak a valóságtól, hogy már nem képesek igazán betölteni alátámasztó, szemléltető szerepüket. Az ily módon magára hagyott, elvont elmélet alkalmazása valós problémák megoldására komoly nehézséget jelenthet. Ennek szélsőséges esete, amikor egy bonyolult, egyedi problémára adott, már létező megoldás kínkeserves megértése útján „fejtjük vissza", hogy az általánosan érvényes elméleti anyagot hogyan is kell – vagy kellett volna – értelmezni.

Ez a gondolatmenet lényegében azt a dilemmát rejti magában, hogy vajon úgy vagyunk-e képesek megtanulni valamit, hogy a sok-sok konkrét esetből általánosítunk, tanulságokat vonunk le, amelyeket a következő probléma esetén már tapasztalatként beépítünk az újonnan előállított megoldásainkba (indukció), vagy úgy, hogy készen kapunk egy általánosított képet, amelynek ismeretében minden probléma esetén rögtön a megfelelő megoldással állunk elő (dedukció). Meggyőződésem szerint ez utóbbi kép csak igen egyszerű problémák esetén működőképes. Megtanulhatjuk előre a másodfokú egyenlet megoldóképletét, amivel bármely másodfokú egyenletet azonnal meg tudunk oldani, de a mérnöki gyakorlat számos módszere intuitív, kevéssé formalizálható ilyen egyértelműen. Ebből adódóan a tanulási folyamat inkább iteratívnak mondható. Tanulmányozhatjuk az elméleti anyagot, de egy idő után csak úgy fejlődhetünk tovább, ha konkrét problémákat oldunk meg, önállóan. Egyiket a másik után… sokáig… Ez egy nem megspórolható, idő- és energiaigényes folyamat, aminek eredményeképpen felsejlenek azok a jellemző sémák, amelyeket az a bizonyos elméleti forrásanyag igyekezett bemutatni. Ekkor újra áttanulmányozva ezt az elméleti jellegű anyagot a megértés egy újabb szintjére juthatunk. A fenti folyamat egyszeri „végigjátszása" adhat ugyan alkalmazható tudást, ennek ellenére, ha újból és újból visszanyúlunk az alapfogalmakhoz, a legalapvetőbb tételekhez, amelyeket a tanulás adott pontján – nem is feltétlenül tudatosan – csak megjegyeztünk, elfogadtunk, egyre mélyebb és mélyebb megértését érhetjük el az adott szakterületnek.

Ez a kötet a *Digitális rendszerek modellezése és szintézise* (a továbbiakban DRMSz) című tankönyvet kiegészítő, az informálisan rögzített követelmények alapján végzett rendszertervezés és az azzal szorosan összefüggő, a hardverleíró nyelvekre épülő RTL-tervezés praktikus aspektusait szemléltető példatár. A ma már kissé talán elavultnak ható „példatár" kifejezés láttán az idősebb olvasó lelki szemei előtt alighanem felsejlenek a tanulmányait végigkísérő, többszáz vagy akár többezer oldalas kötetek, amelyek egy szűk szakterület elméletét egymástól független példák garmadájának felvonultatásával szemléltették és támasztották alá. A gyakorlásnak és tapasztalatszerzésnek a tanulásban betöltött szerepéről alkotott véleményem alapján valószínűleg nem meglepő, hogy lelkes híve vagyok e műfajnak. Azonban az olvasmányosság érdekében az egyenrangú példák ömlesztett közlése helyett én inkább egy jól körülhatárolt céllal tervezett, összefüggő funkcionális egység – ha úgy tetszik, egyetlen nagy áramköri terv – bemutatásával igyekszem szemléltetni

az RTL-tervezés gyakorlati részleteit. Ez a példarendszer – demonstrációs jellege ellenére – egy, a gyakorlatban is értelmezhető feladatkört lát el, és telis-tele van áramkörtervezési szempontból érdekes és didaktikailag is releváns részegységekkel. Ugyancsak fontos jellegzetesség, hogy a rendszer természetéből adódóan bővíthető, ami reményeim szerint saját, a nagy egésszel kompatibilis áramkörök tervezésére sarkallja majd az olvasót.

A könyv első fejezetében megfogalmazunk egy releváns, való életből vett rendszertechnikai problémát, életre hívva példaáramkörünk, a teljes kötet háttértörténetét jelentő „funkcióbővítő rendszer" alapötletét. Erre az elképzelésre támaszkodva rögzítünk néhány kulcsfontosságú technikai követelményt, majd kidolgozunk egy, az elvárásoknak eleget tevő részletes rendszertervet, megnevezve a főbb szereplőket és azok viszonyrendszerét. Az RTL-modellezést három fejezetre bontva tárgyaljuk: a második és harmadik fejezetben sorra vesszük a rendszerterv részegységeit az első fejezetbeli kifejtésük sorrendjében. Ezek a részegységek változatos és egyedi szerepkörű funkcionális egységek, amelyek kellően összetettek ahhoz, hogy bennük egyszerűbb, újra és újra felbukkanó, általános részfunkciókat (pl. memóriák, kommunikációs és aritmetikai áramkörök) azonosítsunk. Az ilyen részfunkciók megvalósítására a különféle áramköri környezetekbe változtatás nélkül beépíthető RTL-terveket, ún. IP (**I**ntellectual **P**roperty) magokat tervezünk, amelyek részleteit a negyedik fejezetben fejtjük ki. Ily módon az RTL-tervezés tanulmányozása „szélességi és mélységi bejárással" is lehetővé válik; ha például az olvasó a rendszer központi processzorának 2.2.2. alfejezetbeli tárgyalásakor hivatkozott, általános célú dual-port memóriamodul RTL-tervére kíváncsi, akkor a 4.2. alfejezethez lapozva azonnal beleáshatja magát a részletekbe, de ha előbb a teljes képet „madártávlatból" szeretné látni, akkor haladhat tovább a második fejezetben, a megértéshez szükséges legfontosabb információk a beépülő almodulokról ott is megtalálhatóak.

A téma elméletét feldolgozó könyvben bemutatott számtalan – talán túlságosan is – egyszerű példa tanúsága szerint az RTL-modellek formális rögzítésére használt VHDL és SystemVerilog nyelvek gyakorlatilag egyenértékűnek tekinthetők. Ez annak köszönhető, hogy az automatizált áramkörszintézis bemenetéül szolgáló modellek belső struktúrái nem túl változatosak, az azokat leíró VHDL és SystemVerilog nyelvi szerkezetek egymásnak könnyen megfeleltethetők. Az itt bemutatott áramköri modellek esetén – a terjedelmi korlátok kétes megalapozottságú álcája mögé bújva – csak az egyik hardverleíró nyelvű modellt elemzem részletesen, meghagyva a lehetőséget az olvasónak az azonos tartalmú másik modell elkészítésére, és az efelett érzett elégedettség maradéktalan átélésére.

A regiszter-transzfer szintű tervezésben egy adott problémára rengeteg jó megoldás adható, így az itt bemutatott példák is csak egy-egy lehetséges megoldásnak tekintendők. E lehetséges megoldások összessége valójában áramköri modelleknek egy konzisztens gyűjteménye, amely lehetővé teszi az ismétlődő elemeknek, a könyv lényegi mondanivalóját alkotó tervezési mintázatoknak a felismerését. Ezekre a tervezési mintázatokra az olvasó bizalommal építhet saját áramköreinek megtervezése során.

Dr. Horváth Péter
Budapest, 2024. november

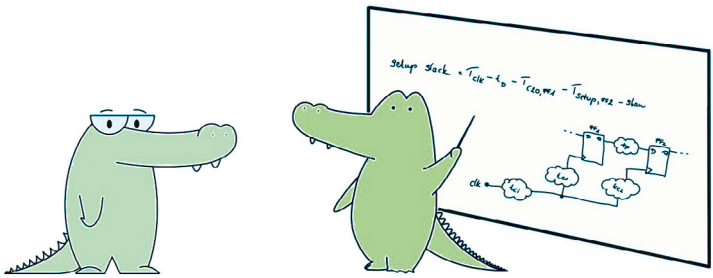
Köszönetnyilvánítás

A kézirat hibáinak felderítésében, a szöveg olvashatóbbá és érthetőbbé tételében egykori és jelenlegi hallgatóim és munkatársaim működtek közre, időt és energiát nem sajnálva. Kiss Dóra, Atiya Ofek Penina, Gembicki Bálint, Rozsvai Kristóf, Hepp Márton, segítségeteket hálásan köszönöm!

Ugyancsak köszönet illeti a BME Elektronikus Eszközök Tanszékének munkatársait – legfőképpen Dr. Jani Lázárt –, akik feladataim átvállalásával egy teljes éven át biztosították a zavartalan, elmélyült munka lehetőségét.

Köszönetemet fejezem ki továbbá Kiss Gergelynek a könyv témájának relevanciáját alátámasztó gondolataiért, valamint Mayer Adriennek a tankönyvszerű száraz egyhangúságot megtörő szellemes illusztrációkért.

A legnagyobb hálával Arnóczki Ildikónak tartozom, aki – amellett, hogy alaposan áttanulmányozta a kézirat minden betűjét, tökéletesen betöltve a témával először találkozó, roppant érdeklődő olvasó szerepét – a több mint három évig tartó munka során nemcsak tolerálta megszállottságomat, hanem minden eszközzel támogatott a fókuszálásban, lehetővé téve, hogy a feladat végeztével fellélegezve vághassak bele minden eddiginél izgalmasabb, közös kalandunkba.

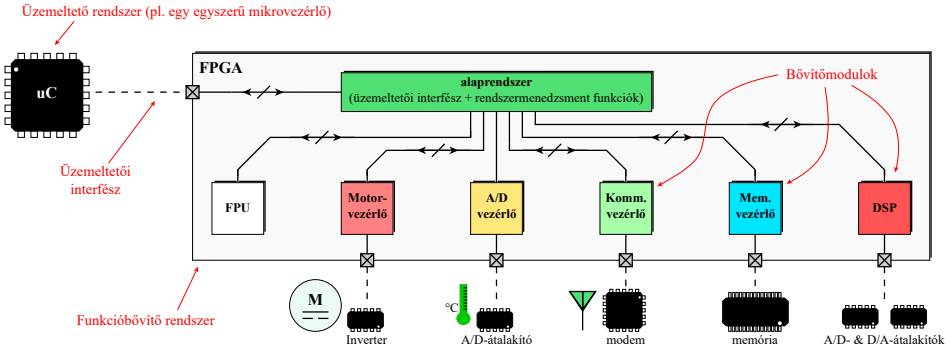


1

Problémafelvetés és rendszerterv

A TÁROLT PROGRAMOT VÉGREHAJTÓ, általános célú mikroáramkörök, vagyis a mikrovezérlők kifejlesztése során a gyártó leginkább arra törekszik, hogy minél többféle probléma megoldására tegye alkalmassá a termékét, természetesen a gyártási költséget is figyelembe vevő, észszerű határokon belül. Az ebből adódó kompromisszumok a felhasználó szempontjából korlátozásokkal járnak. Az általános felhasználhatóságot a tárolt programú működés biztosítja, ti. az eszköz konkrét funkcióját egy adott alkalmazásban a felhasználó által létrehozott program határozza meg. Az utasítássorozat végrehajtására alkalmas szerkezet és az azon futó szoftver együttesének számítási teljesítménye azonban elmarad egy céláramkörétől. A másik figyelembe veendő sajátosság az, hogy bár a perifériakészlet tekintetében ezek az eszközök nagyon változatosak, mégis nehéz olyan típust találni, ami tartalmazza mindazokat a funkciókat, amelyekre szükségünk van, annál azonban nem többet. Az adott alkalmazásban ki nem használt többletképességek és áramkörök feleslegesen növelik általános értelemben vett költségeinket, a fogyasztást, a méretet és az árat.

A fenti problémákra a programozható logikai eszközök technológiáját segítségül hívva kínálunk megoldást. Példarendszerünk egy FPGA-ban[1] való megvalósításra szánt soft-IP keretrendszer[2] (a továbbiakban „funkcióbővítő rendszer”), amely lehetővé teszi üzemeltetője (pl. egy egyszerű mikrovezérlő) számára, hogy általános célú vagy egyedi funkciójú adatfeldolgozó



1-1. ábra. *A funkcióbővítő rendszer alapelve*
*DSP: **D**igital **S**ignal **P**rocessor, FPU: **F**loating-**P**oint **U**nit*

[1] **F**ield-**P**rogrammable **G**ate **A**rray, lásd DRMSz: 7.2. alfejezet.

[2] Regiszter-transzfer szintű (**R**egister-**T**ransfer **L**evel, RTL), hardverleíró nyelvű (**H**ardware **D**escription **L**anguage, HDL) modellekkel reprezentált részegységek koherens rendszere. Az RTL HDL-modellekre az „IP-mag” (**I**ntellectual **P**roperty) kifejezéssel hivatkozunk.

részegységek (bővítőmodulok) adott alkalmazás igényei szerint összeállított gyűjteményéhez kapcsolódjon egy közös interfészen, az ún. üzemeltetői interfészen keresztül (lásd 1-1. ábra).

Ez a funkcióbővítő rendszer egy, az üzemeltetői interfésszel kapcsolatos feladatokat és az általános rendszermenedzsment funkciókat – futásidejű paraméterek beállítása, hibainformációk gyűjtése stb. – megvalósító alaprendszerből és egy bővítőmodul-katalógusból áll. Ez utóbbi „áramköriterv-gyűjtemény" nagy számítási teljesítményű, szabványos jelfeldolgozó algoritmusokat megvalósító gyorsító áramköröket, kommunikációs és adattároló periféria-áramköröket vagy akár teljesen egyedi funkciójú egyéb részegységeket is tartalmazhat. E könyvtárból válogatva összeállítható az adott alkalmazás igényeinek pontosan megfelelő toplevel architektúra. A funkcióbővítő rendszer egy keretrendszer abban az értelemben, hogy maga a bővítőmodul-könyvtár továbbfejleszthető, kiegészíthető, újabb és újabb, specifikus feladatokat ellátó IP-magok adhatók hozzá. Ezek tervezése során természetesen szem előtt kell tartanunk néhány alapvető, leginkább az újonnan kifejlesztett modul interfészeire vonatkozó előírást, amelyek lehetővé teszik az alaprendszerhez való illesztést.[3]

1.1. Követelmények

A funkcióbővítő rendszer megtervezését – a digitális rendszertervezés elméletével összhangban – az azzal szemben támasztott legalapvetőbb elvárások megfogalmazásával kezdjük. Tulajdonképpen a „funkcióbővítő rendszer" kifejezés mögött megbúvó, korábban vázolt elképzelés tekinthető egy magas szintű követelménynek, ezzel azonban nem elégszünk meg, pár apróságot még rögzítünk a tervezési tér szűkítése céljából.

Fontos, hogy a funkcióbővítő rendszer ne várjon el túl sokat az üzemeltető mikrovezérlőtől, ezáltal minél többféle eszközzel használható legyen. A széles körű alkalmazhatóság érdekében tehát megfogalmazzuk a következő követelményt:

> ***1**. Az üzemeltetői interfész csak a legelterjedtebb, bármilyen gyártó kis- és közepes teljesítményű mikrovezérlőiben is rendelkezésre álló megoldásokra támaszkodhat.*

Az 1-1. ábrán felvázolt rendszert úgy szeretnénk megtervezni, hogy az üzemeltetői interfész kialakításában ne okozzon gondot, ha a bővítőáramköröket tartalmazó FPGA az azokat üzemeltető mikrovezérlőtől távol helyezkedik el. Ennek érdekében előírjuk a következőt:

> ***2**. Az üzemeltetői interfész megvalósítására csak kevés vezetékezést igénylő, soros adatátviteli technikákat alkalmazzunk.*

Az üzemeltetői interfész meghatározza, hogy a mikrokontroller és az FPGA-ban megvalósított funkcióbővítő rendszer hogyan cserélnek információt. Ez az információ lehet a mikrovezérlő által küldött parancs vagy az arra adott válasz, de adatfeldolgozási problémák esetén idetartoznak

[3] A mikrovezérlők és programozható logikai eszközök efféle kombinációja ma már egyetlen chipre integrált formában is elérhető. Ezeket az eszközöket **S**ystem-**o**n-**C**hip (SoC) FPGA-knak hívjuk (pl. Intel Agilex, Xilinx ZYNQ, Mircochip PolarFire SoC stb.). Az SoC FPGA-kban a nagyteljesítményű mikroprocesszor mag és a programozható logikai erőforrások közötti interfészek kialakítása erősen gyártófüggő, ezért funkcióbővítő rendszerünkben nem támaszkodunk erre a technológiára. Az üzemeltetői interfész kialakítása során pedig arra törekszünk, hogy a mikrovezérlők minél szélesebb körével kompatibilis rendszert alkossunk.

a mikrovezérlőtől származó forrásadatok és a visszaküldött, feldolgozott adatok, egyszerű adatgyűjtési feladatok esetén pedig maguk a nyers mérési adatok is. Az ilyen kommunikációs interfészek hardver/szoftver elemeit az általuk nyújtott szolgáltatások és az azokat megvalósító protokollok (logikai szabályrendszerek) szerint rétegekbe szervezzük. Ezt a szervezési elvet különféle hivatkozási modellek írják le, amelyek közül az egyik leggyakrabban idézett az ún. OSI (**O**pen **S**ystem **I**nterconnection) modell. Az OSI-modellben az ún. fizikai réteg definiálja azt, hogy az információ elvont alapegysége, a bit miképpen képződik le egy fizikai mennyiségre, például egy vezetéken mérhető feszültségre. Ennek a rétegnek a szolgáltatására, vagyis a nullák és egyesek egyértelmű megkülönböztetésének képességére építkezik az eggyel magasabb, ún. adatkapcsolati réteg, amelynek feladata az összetartozó bitek, az adatkeretek, legjellemzőbb esetben a bájtok (8 bit) fogalmának rögzítése és azok egyik pontból a másikba való átviteli módjának meghatározása. Az ennél magasabb rétegek azután már használhatják az adatkeret fogalmát további adatszerkezetek, pl. üzenetek, csomagok definiálására, és azok átviteli módszerének, protokolljának meghatározására.

Az OSI-modell nem rögzíti, hogy hol van a határ a hardver és a szoftver között. Azt, hogy melyik réteg szolgáltatásait valósítjuk meg „hardverben", tehát céláramkörök segítségével, és melyiket egy processzoron futó szoftver segítségével, ezek szerint csak döntés kérdése. A fizikai réteg esetén ez a döntés túl sok gondolkodást nem igényel, a fizikai mennyiségek bitekké alakításához megfelelő hardverelemekre van szükségünk. Az RS-485 szabvány például a következőképpen definiálja a „bit" fogalmát:

- A bit értékét két vezeték (A és B) potenciálkülönbsége határozza meg. Ha az A vezeték potenciálja legalább 200 mV-tal nagyobb, mint a B vezetéké, akkor a bit értéke 0, ha a B vezeték potenciálja legalább 200 mV-tal nagyobb, mint az A vezetéké, akkor a bit értéke 1. Ha a két vezeték potenciálkülönbségének abszolútértéke kisebb, mint 200 mV, akkor a bit értéke nem értelmezhető.

Elvileg nincs akadálya, hogy az adatkapcsolati réteg összes szolgáltatását szoftveresen valósítsuk meg, hiszen a mikroprocesszorok képesek arra, hogy beolvassák és megváltoztassák digitális kivezetéseik értékét. A gyakorlatban azonban – mivel az 1 és 0 értékek váltakozása egy valóságos kommunikációs interfészen igen gyors lehet – ez a megoldás inkább csak elvi jelentőségű, mert a processzorok részéről túl nagy, adott esetben megvalósíthatatlanul nagy erőfeszítést igényelne. Az adatkapcsolati réteget, amely tehát a biteknek a nagyobb logikai egységekbe, adatkeretekbe szervezését végzi, tipikusan ugyancsak céláramkörökkel valósítjuk meg. Gondoljunk például egy UART (**U**niversal **A**synchronous **R**eceiver/**T**ransmitter) átvitelre (lásd 5.2.1. Függelék), amelyben a 8-bites bájt egyetlen vezetéken, megfelelő időzítési szabályoknak megfelelő módon való előállítását egy UART periféria-áramkör, vagyis egy párhuzamos-soros átalakító végzi.

A lehető legnagyobb rugalmasság érdekében a funkcióbővítő rendszert úgy tervezzük meg, hogy az üzemeltetői interfész protokollrétegei közül a lehető legkevesebbet vessük kőbe, így a rendszer és az azt használó mikrovezérlő közötti kommunikáció maga is az adott alkalmazás igényei szerint változtatható, testreszabható legyen. Megfogalmazható tehát a következő követelmény:

***3**. A funkcióbővítő rendszernek lehetővé kell tennie, hogy az üzemeltetői interfész protokolljainak az adatkapcsolati rétegénél magasabb szintű szolgáltatásait a felhasználó maga tervezze és valósítsa meg az alaprendszer HDL-modelljének módosítása nélkül.*

A következő alfejezetben felvázoljuk a funkcióbővítő rendszernek egy magas szintű modelljét, az ún. rendszertervét (lásd DRMSz: 2. fejezet), amely az alapvető architekturális elemekkel, valamint a külső/belső kommunikációs interfészekkel kapcsolatos tervezési döntéseinket rögzíti a fenti követelményekkel összhangban.

1.2. Rendszerszintű modell, rendszerterv

A funkcióbővítő keretrendszer dolga az, hogy specifikus feladatot ellátó céláramkörök szolgáltatásait egy egységes felületen keresztül tegye elérhetővé egy mikrovezérlő számára. A céláramkörök pontos szolgáltatásait egyelőre ismeretlennek kell tekintenünk, hiszen nem szeretnénk, hogy maga a rendszerterv bármilyen módon is függjön ezektől, ugyanakkor azt nagy biztonsággal kijelenthetjük, hogy e szolgáltatások a mikrovezérlő és az FPGA közötti kétféle kölcsönhatáson alapulnak majd.

- **Vezérlés és állapotlekérdezés**. A bővítőáramkörök részletes tervezése során az újrafelhasználhatóságra törekszünk, ami azt jelenti, hogy igyekszünk minél több funkciót parametrikusan megvalósítani (lásd DRMSz: 4.6. alfejezet). A paraméterek egy része ún. szintézis-paraméter lesz, amelyek a HDL-modell szintjén biztosítanak ugyan némi testreszabhatóságot, de az áramkörszintézist követően értékük már nem változtatható meg.[4] A paraméterek másik típusát futásidejű paramétereknek hívjuk. Ezek a céláramköröknek azon „beállításai", amelyeket használat közben is bármikor módosíthatunk, jellemzően valamilyen konfigurációs regiszter értékének átírásával. A konfigurációs regisztereknek az adott alkalmazás igényei szerinti beállítása az üzemeltetést végző mikrovezérlő feladata. A céláramkörök nemcsak konfigurációs, hanem állapotregisztereket is tartalmaznak, amelyek segítségével a mikrovezérlő nyomon követheti belső állapotuk, vagy egy általuk mért külső mennyiség értékének alakulását (pl. egy A/D-átalakító által mért feszültségszint digitalizált értékét).
- **Adatforgalmazás**. A konfigurációs és állapotregiszterek jellemzően csak legfeljebb néhányszor tíz bitesek, vagyis ezek írása és olvasása nem jelent nagy adatforgalmat a mikrovezérlő és az FPGA között. Bizonyos bővítőmodulok esetén viszont szükség lehet arra, hogy nagy mennyiségű információt rövid idő alatt mozgassunk a két eszköz között (pl. egy külső, nagysebességű memóriamodulhoz való hozzáférés esetén).

A kétféle adatcseretípus eltérő szerepe (vezérlés vs. adat) és jellege (kis adatmennyiség vs. nagy adatmennyiség) okán célszerű a kettőt egymástól függetleníteni, lehetővé téve a mikrokontroller számára, hogy azokat akár egymással párhuzamosan is használja. Az üzemeltetői interfészt tehát logikailag és fizikailag is két részre bontjuk, egy vezérlő- és egy adatinterfészre. Mindezeket figyelembe véve a rendszertervezés részeként a következő döntéseket hozzuk az üzemeltetői interfész kialakítására vonatkozóan:

[4] Sőt, valójában az automatizált áramkörszintézisnek egy pontján, az ún. generic-folding eljárás következtében megszűnnek paraméterként létezni (DRMSz: 8.2.1.1 pont).

- *A vezérlőinterfész adatkapcsolati rétegének megvalósítására hardver-átvitelvezérlés nélküli UART-ot használunk (lásd 5.2.1. Függelék). Ez a megoldás mindössze két vezetéket használ a kétirányú kommunikáció létrehozásához.*
- *Az adatinterfész adatkapcsolati rétegének megvalósítására egyszerű, négyvezetékes SPI-t (**S**erial **P**eripheral **I**nterface, lásd 5.2.2. Függelék) használunk. Az SPI bitsebessége már a közepes teljesítményű mikrovezérlőkben is elérheti a néhányszor 10 Mbit/s-ot, így nagymennyiségű adat forgalmazására is alkalmas. Mivel ezen az interfészen igyekszünk a lehető legnagyobb átviteli sebességet elérni, az SPI négy vezetéke mellé további két, általános célú I/O vezetéket (**G**eneral-**P**urpose **I**nput/**O**utput, GPIO) is bevezetünk, amelyeket gyors forgalomszabályozásra, lényegében hardveres átvitelvezérlésre használunk majd (a részleteket lásd 1.2.3.5. pont).*

Az UART- és SPI-interfészek a legáltalánosabban használt soros átviteli megoldások a mikrovezérlőkben, így az 1. és 2. számú követelményeket teljesítjük. Az UART és SPI szabványok – adatkapcsolati szintű protokollok lévén – a bitek fizikai megvalósításával nem foglalkoznak, ezt a kérdést a széles körű alkalmazhatóság érdekében nyitva hagyjuk. Ha rendszerünk konkrét fizikai kialakításában az üzemeltető mikrovezérlő és a bővítőmodulokat tartalmazó FPGA egyazon nyomtatott huzalozású lemezen (**P**rinted **C**ircuit **B**oard, PCB) helyezkedik el, akkor a kommunikáció fizikai rétegét megoldhatjuk egyszerűen a megfelelő kivezetések összekötésével (természetesen figyelve a helyes jelszintillesztésre). Ha a két eszköz két külön PCB-n, adott esetben egymástól nagy távolságra van, akkor az átvitel fizikai rétegét megvalósíthatjuk valamilyen differenciális jelvezetési módszerrel, akár RS-485/422 vagy LVDS (**L**ow **V**oltage **D**ifferential **S**ignaling) meghajtóáramkörökkel illesztett kábelezéssel is, de ha épp a huzalozás mennyiségét kell minimalizálnunk, akkor akár valamilyen PLC (**P**ower **L**ine **C**ommunication) modem áramkört is használhatunk.

A 3. számú követelmény, mely szerint az üzemeltetői interfész magasabb rétegbeli protokolljainak megtervezését és megvalósítását a felhasználóra kell bíznunk, igen fontos, a funkcióbővítő rendszer alapvető architektúráját érintő döntésre kényszerít bennünket. A magasabb szintű protokollrétegek fogalmazzák meg azt, hogy az üzemeltető mikrovezérlő milyen módon fér hozzá a bővítőáramkörök szolgáltatásaihoz. Ilyen szolgáltatás lehet például a konfigurációs és állapotregiszterek beállítása és lekérdezése, egy memóriaillesztő áramkör esetén a külső memória írása és olvasása vagy egy kommunikációs interfészáramkör esetén az adott csatornán érkező információk kiolvasása is. Az üzemeltetői interfész adatkapcsolati rétegbeli protokolljai (UART és SPI) csak annyit fogalmaznak meg, hogy a két eszköz között adatkeretek (bájtok) jönnek-mennek, de ezeknek az értelmezése nem feladata a keretek szintjén működő áramköröknek. Egy adott bővítőmodul konfigurációjának megváltoztatásához például azonosítanunk kell magát a bővítőmodult és azon belül a szóban forgó regisztert, továbbá definiálnunk kell az abba beleírandó adat értékét is. A bővítőmodulok száma, a regiszterek száma és azok mérete határozza meg, hogy mindennek az információnak a kódolására hány bájtra van szükségünk. Az egyszerűség kedvéért tegyük fel, hogy összesen 256 bővítőmodult szeretnénk megkülönböztetni, és azokon belül legfeljebb 256 darab, 1 bájt méretű konfigurációs regiszter lehet. Ekkor az üzemeltető mikrovezérlőnek három bájtot kell elküldenie ahhoz, hogy a rendszer egy konkrét beállítását módosítsa. A protokollnak továbbá – sok egyéb mellett – rögzítenie kell azt is, hogy ezek a bájtok milyen sorrendben kövessék egymást. Ezt a szabályrendszert a kommunikációban részt vevő mindkét fél ismeri, így tudnak egymással „szót érteni".

Ha a protokoll ismert, akkor nincs elvi akadálya az adatkereteket értelmező és generáló céláramkörök, jellemzően összetett állapotgépek kialakításának. A 3. számú követelményünk szerint azonban ezt nem tehetjük meg. Biztosítanunk kell a felhasználónak a lehetőséget, hogy maga találja ki és valósítsa meg a két eszköz közötti kommunikáció szabályrendszerét, ami tulajdonképpen azt jelenti, hogy a funkcióbővítő rendszer viselkedését felhasználói szinten módosíthatóvá kell tennünk. Mivel ebben a kontextusban a „felhasználó" egy olyan személy, akinek a HDL-modellhez nincs hozzáférése, csak olyan technológia jöhet szóba, ahol a rendszer viselkedését egy, a felhasználó számára is elérhető médiumban tárolt információ határozza meg. Ez a gondolatsor elvezet bennünket a tárolt programot végrehajtó gépek, a mikroprocesszorok világába. Megfogalmazható tehát rendszertervünk egyik legfontosabb tétele:

Funkcióbővítő rendszerünk – az azt üzemeltető mikrovezérlőhöz hasonlóan – egy ***tárolt programot végrehajtó mikroáramkör*** *lesz, amelynek periféria-áramkörei töltik majd be az általános célú és/vagy alkalmazás-specifikus adatfeldolgozó feladatokat ellátó bővítőmodulok szerepét.*

Az ilyen, programozható logikai eszközökben megvalósított, tárolt programot végrehajtó mikroáramkörök és a hozzájuk tartozó periféria-áramkörök egységét soft-processzoros rendszernek nevezzük.[5]

Az eddig meghozott tervezői döntéseink lehetővé teszik a funkcióbővítő rendszer modelljének további finomítását, amely így már kellően részletgazdag lesz ahhoz, hogy az RTL-tervezés közvetlen specifikációjául szolgáljon. Szándékunk kifejezésére ezen a szinten formális nyelvet még nem használunk, a rendszerszintű modellt tehát természetes nyelvű leírások alkotják majd, amelyeket a leendő részáramköröktől elvárt viselkedést szemléltető időzítési diagramokkal (hullámformákkal), illetve a rendszer szerkezetét és almoduljainak összeköttetéseit szemléltető blokkvázlatokkal egészítünk ki.

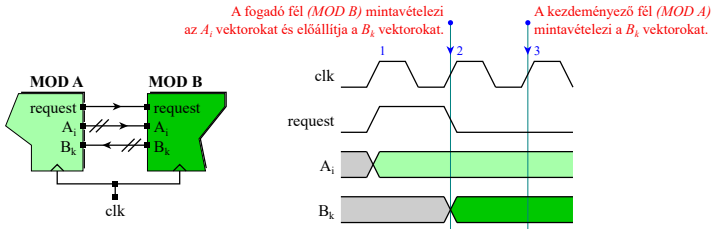
1.2.1. Almodulok közötti adatcsere

A digitális áramkörök funkcióinak változatosságából adódóan azok interfészeinek kialakítására is rengetegféle megoldás kínálkozik. Ha egy összefüggő, sok részáramkörből álló, komplex rendszert, vagy akár csak egy több, egymástól független, újrafelhasználható IP-magból álló áramkörkatalógust készítünk, célszerű az egyes IP-k interfészeire vonatkozóan bizonyos szabályokat bevezetni a konzisztencia érdekében. Az egységesítés mind a tervezést, mind a felhasználást megkönnyíti, és egyéb rendszertechnikai előnyökkel is jár. Ha a hasonló funkciót ellátó részáramkörök egyazon interfész-specifikációt valósítják meg (pl. egyazon utasításkészletet implementáló két processzor két eltérő teljesítményű és erőforrás-felhasználású mikroarchitektúrával), akkor egymással könnyen kicserélhetők, növelve a rugalmasságot, szélesítve a lehetséges felhasználások körét. Funkcióbővítő rendszerünk egésze egy összefüggő, egyedi feladatot ellátó áramköregyüttes, amelynek részegységeit ráadásul újrafelhasználható IP-ként tervezzük elkészíteni, így az egységesített interfészek használata feltétlenül követendő tervezési elv számunkra is.

[5] Az üzemeltető rendszer és a funkcióbővítő rendszer központi egysége tehát egyaránt egy-egy tárolt programú mikroáramkör, amire a magyar nyelvű szakirodalom gyakran használja a „processzor" megnevezést. Annak érdekében, hogy a szövegben egyértelmű legyen, hogy épp melyikre hivatkozunk, a továbbiakban az előbbire következetesen az „üzemeltető mikrovezérlő/rendszer", utóbbira pedig a processzor vagy CPU (**C**entral **P**rocessing **U**nit) kifejezést használjuk.

1.2.1.1. Pont–pont kapcsolatok

A legegyszerűbb ütemezett adatcsere (tranzakció) két almodul között az, amikor adott pillanatban a kezdeményező fél tetszőleges számú logikai vektor értékét közli a fogadó féllel, amely erre ugyancsak tetszőleges számú logikai vektorral válaszol.[6] Ha biztosak vagyunk abban, hogy a fogadó fél bármely pillanatban készen áll a tranzakció kezelésére és azonnal képes előállítani a válaszként küldendő logikai vektorokat, akkor használhatjuk az 1-2. ábrán látható hullámforma szerint működő interfészt.



1-2. ábra. *Két almodul közötti egyszerű adatcsere (tranzakció) időzítési modellje*

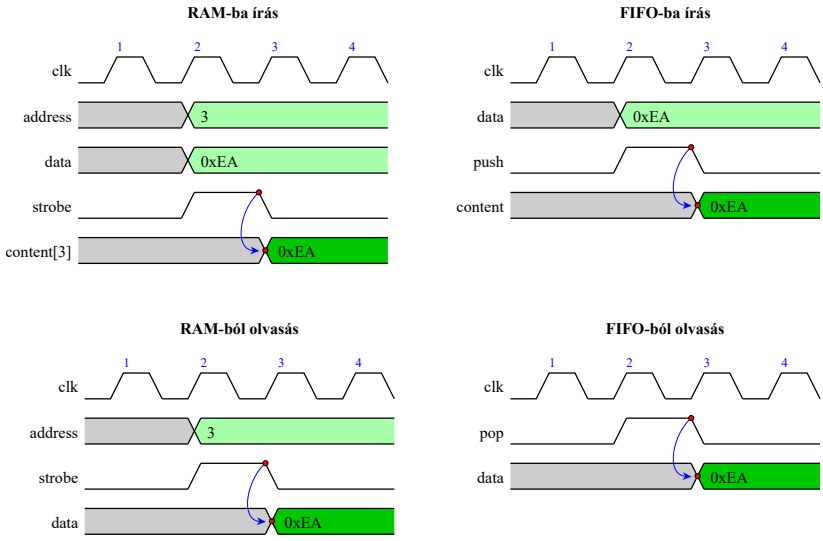
A tranzakció értelmezéséhez fontos hangsúlyozni, hogy az almodulok szinkron működésűek, vagyis a bemeneteiken bekövetkező változásokra csak az órajel következő felfutóélénél képesek reagálni, illetve kimeneteiket is csak az órajel felfutóélénél képesek változtatni. A kezdeményező fél egyetlen pozitív impulzussal jelzi, hogy az A_i vektorok bitjeit a kívánt értékre állította, a *request* jelet tehát a fogadó fél használhatja azok mintavételezésére.[7] A kezdeményező fél feltételezheti továbbá azt, hogy a B_k vektorok értéke a *request* jel lefutásával egyidőben változik meg, vagyis azok az órajel következő felfutóélénél már mintavételezhetők.[8] Ennek a tranzakciónak egy speciális esete, ha az A_i és/vagy a B_k vektorok száma nulla, vagyis a tranzakció tulajdonképpen csupán egy esemény jelzését szolgálja a két almodul között.

Ezt az interfésztípust alkalmazzuk például azoknál az on-chip memóriamoduloknál, amelyek az FPGA belső blokk-RAM (**R**andom **A**ccess **M**emory, véletlen elérésű memória) moduljaiban tárolják az információt, és sem az adat beírásához, sem annak kiolvasásához nincs szükségük több órajelciklusra. Az 1-2. ábra általánosan megfogalmazott interfész-specifikációját véletlen elérésű memóriamodulokra és FIFO (**F**irst **I**n **F**irst **O**ut) tárolókra alkalmazva a konkrét hullámformák az 1-3. ábrán látható alakot öltik.

[6] Természetesen az is almodulok közötti adatcserének nevezhető, ha az egyik almodul által meghajtott vezetéket a másikba statikusan bekötjük. Némi izgalmat jelenthet, ha a két almodult két független, aszinkron órajelről járatjuk, ebben az esetben ui. órajel-szinkronizáló fokozatokat kell használnunk (lásd DRMSz: 6.3.1. alfejezet). Ez azonban RTL-tervezési probléma, nem igényel rendszerszintű megfejtéseket és iránymutatást.

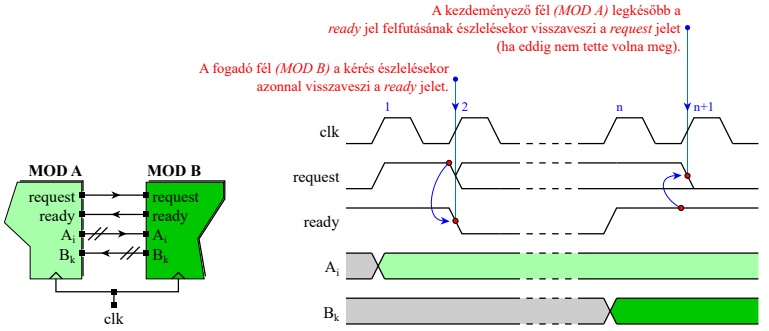
[7] A rendszerterv – és később az RTL-modellek – konkrét elemeire (részegységek, jelek, be- és kimenetek stb.) a folyószövegben *kiemelt betűtípussal* hivatkozunk.

[8] Az időzítési diagramok a kombinációs logikai hálózatok és vezetékek késleltetését nem érzékeltetik. Ennek az ábrázolási módszernek a célja, hogy ezek az ábrák minél jobban hasonlítsanak az időzítések nélküli, tisztán logikai szimulációval nyert hullámformákhoz, amelyekkel az olvasó a logikai szimulációs szoftverekben találkozhat.



1-3. ábra. *Egyszerű adatcsere-interfész használata RAM- és FIFO-memóriák esetén*

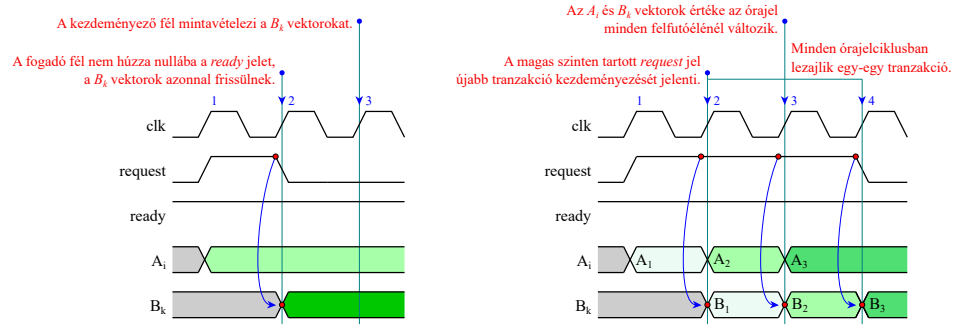
Ha a fogadó fél bármely pillanatban készen áll ugyan üzenet fogadására, de nem garantálja, hogy az üzenet hatására végbemenő folyamatok azonnal lezajlanak benne, akkor az egyszerű *request* impulzust egy visszajelző vonallal kell kiegészítenünk, ahogy azt az 1-4. ábra szemlélteti. Az ilyen visszacsatolást tartalmazó interfészeket kézfogásos (hand-shake) interfészeknek nevezzük.



1-4. ábra. *Visszajelző vonallal kiegészített, kézfogásos (hand-shake) adatcsere-interfész időzítési modellje*

A *request* jel felfutásának észlelésekor, vagyis az órajel második felfutóélénél a fogadó fél a *ready* jelet alacsony szintre húzza, jelezve, hogy megkezdte az üzenet feldolgozását. Ekkor a kezdeményező fél a *request* jelet magas szinten tarthatja vagy vissza is veheti alacsony szintre, a fogadó félnek erre érzéketlennek kell lennie. A fogadó fél az órajel n-edik felfutóélénél a *ready* jel magas szintre állításával jelzi, hogy az üzenet feldolgozása megtörtént, a B_k vektorok értéke mintavételezhető. Ezt az órajel (n+1)-edik felfutóélénél érzékelve a kezdeményező félnek – ha eddig nem tette volna meg – logikai alacsony szintre kell állítania a *request* jelet. Ezzel a tranzakció befejeződött.

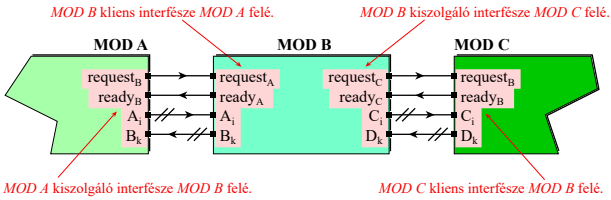
Elképzelhető olyan adatcsere-interfész is, amelyben a tranzakciók késleltetése, vagyis a kezdeményező fél *request* jelének felfutása és a fogadó fél *ready* jelének felfutása között eltelt órajelciklusok száma nem állandó, hanem szintézis- vagy futásidejű paraméter (pl. függ az A_i vektorok értékétől). Ilyenkor a nulla késleltetésű tranzakció egy külön figyelmet érdemlő, szélsőséges eset, ami alatt azt értjük, hogy a fogadó fél a *ready* jelzését a kezdeményező fél *request* jelének felfutása hatására nem húzza nullába (lásd 1-5. ábra).



1-5. ábra. *Nulla késleltetésű adatcsere-tranzakciók a kézfogásos interfészen*

Ebben az esetben a kezdeményező fél már az órajel harmadik felfutóélénél logikai magas szintet tapasztal a fogadó fél *ready* vonalán, tehát a B_k vektorokat azonnal mintavételezi. Ez az eset időzítési szempontból tehát tulajdonképpen a kézfogás nélküli adatcserével egyenértékű. Ha ebben az esetben a kezdeményező fél az órajel második felfutóélénél magas szinten tartja a *request* jelzését, továbbá frissíti az A_i vektorok értékét, az egy újabb tranzakció kezdeményezését jelenti, vagyis a B_k vektorok az órajel harmadik felfutóélénél ismét megváltozhatnak. Ezt a mintát folytatva a tranzakciók szorosan egymás után fűzhetőek, vagyis minden órajelciklusban lezajlik egy-egy tranzakció.

Az almodulok közötti viszonyok szemléletessé tétele érdekében bevezetjük a „kliens/felhasználó” (client) és a „kiszolgáló” (server) fogalmakat. Az eddig vázolt adatcsere-mechanizmusokban mindkét irányba történhet adatáramlás, ebben az értelemben tehát a tranzakció szimmetrikus. Ugyanakkor a két fél szerepe nyilvánvalóan eltér, hiszen csak az egyik fél képes tranzakció kezdeményezésére, a másik fél csak reagálhat arra. Úgy is felfoghatjuk, hogy a kezdeményező fél mint kliens vagy felhasználó egy, a fogadó félnek, vagyis a kiszolgálónak szóló kérést fogalmaz meg, mely szerint annak egy szolgáltatását igénybe szeretné venni, a kiszolgáló pedig ennek a kérésnek eleget tesz. Maga a szolgáltatás gyakorlatilag bármi lehet. Egy egyszerű memóriamodul esetén szolgáltatásként foghatjuk fel az adat eltárolását adott címre, vagy annak visszaolvasását adott címről. Egy komplex aritmetikai almodul esetén pedig akár egy bonyolult jelfeldolgozási algoritmus hosszadalmas végrehajtását is tekinthetjük szolgáltatásnak. Figyeljük meg, hogy a „kliens” és „kiszolgáló” jelzők szigorúan véve nem az IP-maghoz, hanem annak interfészeihez tartoznak, hiszen egy adott IP-mag egy másik almodulhoz kapcsolódva lehet „kliens szerepű”, míg egy harmadik almodullal való kölcsönhatásának kontextusában „kiszolgáló szerepű”. Az egységesség és egyértelműség érdekében rögzítjük, hogy egy adott IP-magnak a tranzakciók kezdeményezésére képes interfészeire a „kiszolgáló interfész”, a tranzakciókat fogadó interfészeire pedig a „kliens interfész” kifejezéssel hivatkozunk (lásd 1-6. ábra).



1-6. ábra. *IP-magok kiszolgáló és kliens interfészei*

Az elnevezések az adott IP-mag szempontjából értelmezendők. Ha egy IP-maghoz az adott interfészen keresztül kéréssel fordulhatunk, akkor az az adott IP-mag szempontjából egy „kliens felé néző” interfész, tehát kliens interfész. Ha egy IP-mag egy adott interfészén kéréseket tud megfogalmazni, akkor az az IP-mag szempontjából egy „kiszolgáló felé néző” interfész, tehát kiszolgáló interfész.

1.2.1.2. Az MMIO-buszrendszer

Az előző pontban tárgyalt interfész-specifikációk belső almodulok közötti pont–pont kapcsolatok egyszerű, egységes megvalósítására alkalmasak. A processzoros rendszer központi feldolgozó egysége *(Central Processing Unit, CPU)* és a periféria-áramkörök között egy hasonló interfészt alakítunk ki. Sem a *CPU*-val, sem a periféria-áramkörökkel kapcsolatban nem rögzítettünk erre vonatkozó konkrét követeléseket, a tervezés kiindulópontja tehát csak az lehet, hogy a rendszerben egy *CPU* kommunikál több periféria-áramkörrel olyan tranzakciókon keresztül, amelyek során a *CPU* mint kliens igénybe veszi a periféria-áramkörök mint kiszolgálók szolgáltatásait. Fogalmazhatunk tehát úgy is, hogy feladatunk egy általános célú, csillag topológiájú, azon belül is többkiszolgálós (multi-slave) kialakítású hálózat, és az azon folyó kommunikáció protokolljának megalkotása.

A processzoros rendszer architektúrájának egyszerűsítése, ezáltal az erőforrásigény minimalizálása érdekében ún. MMIO (**M**emory-**M**apped **I**nput/**O**utput) perifériakezelést valósítunk meg. A kifejezés arra utal, hogy a központi egység szempontjából az adatmemória és az egyedi funkciójú periféria-áramkörök (esetünkben ideértve magukat a bővítőmodulokat is) egyenértékűek. A *CPU* az adatmemóriát és a periféria-áramköröket egyazon buszrendszeren keresztül éri el, amely mindössze kétféle műveletet, az MMIO-írás és az MMIO-olvasás tranzakciókat definiálja.[9] Az egyes tranzakciókhoz rendelt, a *CPU* által meghatározott címinformáció alapján dönthető el, hogy a *CPU* éppen melyik periféria-áramkört „szólította meg”. A különböző periféria-áramkörök egyedi módon értelmezhetik a nekik szóló írás- és olvasáskéréseket,[10] általánosan igaz azonban, hogy írás esetén a *CPU* küld egy 16-bites adatot a periféria-áramkör felé, olvasás esetén pedig fordítva. Az MMIO-buszrendszert alkotó jeleket az 1-1. táblázat foglalja össze.

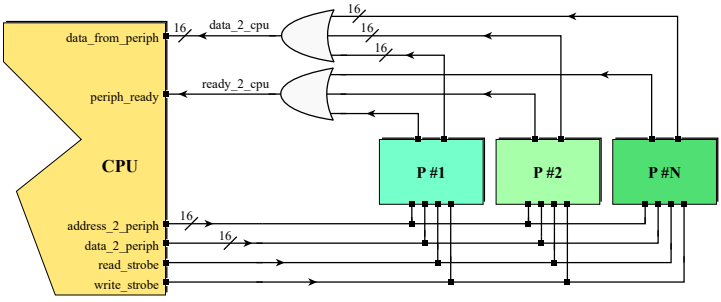
[9] A *CPU* MMIO-írást a *store*, MMIO-olvasást pedig a *load* gépi utasítás végrehajtásával kezdeményez (lásd 1-2. táblázat).

[10] Fogalmazhatunk úgy is, hogy az adatmemória csupán egy korlátolt periféria-áramkör, amelynek képességei nem mutatnak túl a neki küldött adatok eltárolásán és visszaolvasásán.

Jel/busz neve	Bitek száma	Jel/busz funkciója	Jel/busz iránya		
read_strobe	1	A *CPU* ennek a jelnek a logikai magas szintre állításával kezdeményez olvasástranzakciót.	*CPU*	→	Periféria
write_strobe	1	A *CPU* ennek a jelnek a logikai magas szintre állításával kezdeményez írástranzakciót.	*CPU*	→	Periféria
periph_ready	1	A periféria-áramkörök ennek a jelnek a logikai alacsony szintre állításával a *CPU* által kezdeményezett írás/olvasás-tranzakció megkezdését, majd magas szintre állításával a művelet befejezését jelzik.	*CPU*	←	Periféria
address_2_periph	16	Az írás/olvasástranzakcióhoz tartozó cím.	*CPU*	→	Periféria
data_2_periph	16	Az írástranzakcióban a *CPU*-tól a periféria-áramkörnek küldött adat.	*CPU*	→	Periféria
data_from_periph	16	Az olvasástranzakcióban a periféria-áramkörtől a *CPU*-nak küldött adat.	*CPU*	←	Periféria

1-1. táblázat. *Az MMIO-buszrendszert alkotó jelek*

A rendszert FPGA-ban szeretnénk megvalósítani úgy, hogy az a lehető legtöbbféle konkrét eszköztípusban létrehozható legyen. Mivel az FPGA áramkörök csak a külvilághoz kapcsolódó I/O-blokkjaikban tartalmaznak háromállapotú (tri-state) meghajtókat, ezért a belső buszrendszer kialakításakor erre a megoldása nem támaszkodhatunk. A processzornak szóló információt hordozó buszok esetén azonban – a többkiszolgálós topológiából adódóan – a periféria-áramkörök egyazon jeleket kell hogy meghajtsák, ami a tranzakcióhoz tartozó, a processzor által kiadott memóriacím alapján végzett multiplexálással megvalósítható lenne. Ez a megoldás a periféria-áramkörök számának növekedésével gyorsan növekvő méretű központi multiplexereket igényelne, amelyek – amellett, hogy sok logikai erőforrást emésztenek fel – FPGA technológián igen kedvezőtlen időzítési tulajdonságokkal valósíthatók csak meg. Ennek elkerülése érdekében ún. elosztott címdekódolást alkalmazunk, ahol az egyes periféria-áramkörök kimeneti jelei nem multiplexerekbe, hanem *OR* kapukba futnak be (lásd 1-7. ábra).

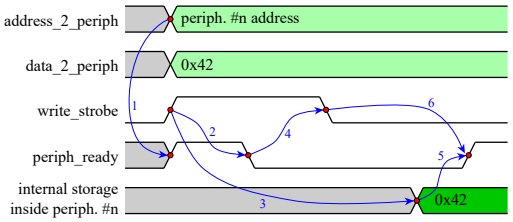


1-7. ábra. *Elosztott címdekódolás*

Az *OR* kapu lényegesen kisebb erőforrásigényű megoldás, de ez a címdekódolási séma a periféria-áramkörökkel szemben újabb követelményeket támaszt. Minden periféria-áramkörnek ismernie kell a saját címtartományát és tartalmaznia kell egy belső címdekódert, amely gondoskodik arról, hogy ha a processzor címbuszán nem a saját címtartományába tartozó cím van, akkor

a kimeneti jeleinek mindegyike szigorúan logikai alacsony szintre legyen húzva. Így biztosítható, hogy az *OR* kapuk kimenete mindig az éppen megcímzett periféria-áramkör kimeneti adatbuszának az értékét hordozza.

Az MMIO-buszon a *CPU* írás- vagy olvasáskéréseket küld a periféria-áramköröknek. A periféria-áramkörök közül minden ilyen kérésre csak annak az egynek van „joga" válaszolni, amelyiknek a saját címtartományába beleesik a processzor által a kérésben elküldött konkrét cím. Az ilyen kérés-válasz párosokat MMIO-tranzakcióknak nevezzük. Az MMIO-interfész specifikációját – az azt megvalósító jelek szerepének és fizikai kialakításának meghatározása mellett – a tranzakciókat leíró szabályrendszer (protokoll) teszi teljessé. Az 1-8. ábra az MMIO-írás, az 1-9. ábra pedig az MMIO-olvasás folyamatát szemlélteti. Az egyszerűség kedvéért a magyarázó ábrákon memória jellegű periféria-áramkört feltételezünk, tehát a processzor által küldött adatot a periféria-áramkör egyszerűen eltárolja. A tárolás elvégzése után a periféria-áramkör azonnal értesíti a processzort, hogy az adatátvitel megtörtént.

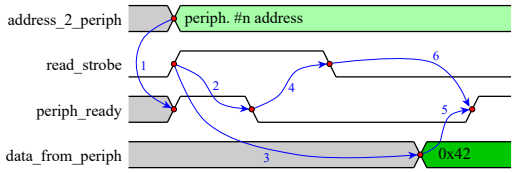


1-8. ábra. *Az MMIO-írás tranzakció időzítési modellje*

Az MMIO-írás tranzakció a következő lépésekből áll:

1. A *CPU* az *address_2_periph* buszra helyezi a megszólítandó periféria-áramkör címét. Ennek hatására a megcímzett periféria-áramkör lokális címdekódere azonnal (a címdekóder kombinációs hálózatának késleltetése után) magas szintre húzza a *periph_ready* vonalat. A *periph_ready* jel magas szintje a *CPU* számára azt jelzi, hogy a kiadott címet egy periféria-áramkör „felismerte" sajátjaként, és készen áll a tranzakció megkezdésére.
2. A *CPU* magas szintre húzza a *write_strobe* jelet. Ezt érzékelve a megcímzett periféria-áramkör a *periph_ready* jel alacsony szintre állításával jelzi a *CPU* felé, hogy a műveletvégzést megkezdte.
3. A *write_strobe* jel magas szintjét érzékelve a megcímzett periféria-áramkör megkezdi működését a kapott cím és adat ismeretében. Egy memória jellegű periféria-áramkör esetén ez a kapott adat eltárolását jelenti a kapott logikai címből származtatott fizikai címre.
4. A periféria-áramkör a *periph_ready* jel alacsony szintre állításával jelezte a *CPU* felé, hogy az írástranzakciót megkezdte. A *CPU* ezt érzékelve alacsony szintre állítja a *write_strobe* jelet.
5. Az írásművelet végrehajtása után a periféria-áramkör a *periph_ready* jelet újra magas szintre húzza, jelezve a *CPU* felé, hogy a művelet befejeződött, a *CPU* folytathatja a műveletvégzést a következő utasítás feldolgozásával.

6. A periféria-áramkör a *periph_ready* jel magas szintre állítása előtt „bevárja” a *write_strobe* jel visszavételét.[11]



1-9. ábra. *Az MMIO-olvasás tranzakció időzítési modellje*

Az MMIO-olvasás tranzakció a következő lépésekből áll:

1. A *CPU* az *address_2_periph* buszra helyezi a periféria-áramkör címét. Ennek hatására a megcímzett periféria-áramkör lokális címdekódere azonnal (a címdekóder kombinációs hálózatának késleltetése után) magas szintre húzza a *periph_ready* vonalat. A *periph_ready* jel magas szintje a *CPU* számára azt jelzi, hogy a kiadott címet egy periféria-áramkör „felismerte” sajátjaként, és készen áll a tranzakció megkezdésére.
2. A *CPU* magas szintre húzza a *read_strobe* jelet. Ezt érzékelve a megcímzett periféria-áramkör a *periph_ready* jel alacsony szintre állításával jelzi a *CPU* felé, hogy a műveletvégzést megkezdte.
3. A *read_strobe* jel magas szintjét érzékelve a megcímzett periféria-áramkör megkezdi működését a kapott cím ismeretében. Egy memória jellegű periféria-áramkör esetén ez egy memóriarekesz tartalmának visszaolvasását és annak a *data_from_periph* buszra kapcsolását jelenti.
4. A periféria-áramkör a *periph_ready* jel alacsony szintre állításával jelezte a *CPU* felé, hogy az olvasástranzakciót megkezdte. A *CPU* ezt érzékelve alacsony szintre állítja a *read_strobe* jelet.
5. Az olvasás művelet végrehajtása után a periféria-áramkör a *periph_ready* jelet újra magas szintre húzza, jelezve a *CPU* felé, hogy a művelet befejeződött, a *CPU* folytathatja a műveletvégzést a következő utasítás feldolgozásával.
6. A periféria-áramkör a *periph_ready* jel magas szintre állítása előtt „bevárja” a *read_strobe* jel visszavételét.

*Az 1.2.1.1. pontban bemutatott, pont-pont kapcsolatok esetén használható interfésztípusra a továbbiakban a **„szinkron kézfogásos interfész”**, a CPU és a periféria-áramkörök közötti kapcsolatra pedig az **„MMIO-interfész”** kifejezésekkel hivatkozunk. Bár a két interfésztípus nagyon hasonló, fontos eltérés, hogy míg a szinkron kézfogásos interfész egy közös órajelhez viszonyítva definiálja a kommunikációban részt vevő jelek elvárt viselkedését (erre utal a szinkron jelző), addig az MMIO-interfész – időzítési szempontból – semmilyen formában nem támaszkodik az egyes almodulok órajeleire. Ennek az eltérésnek a hátterét és következményeit a következő pontban részletesen tárgyaljuk.*

[11] Erre azért van szükség, mert ellenkező esetben egy gyors (a *CPU* órajelénél gyorsabb órajelről járó) perifériavezérlő esetén előfordulhatna, hogy az írásművelet még azelőtt véget ér, hogy a *CPU* „észrevette volna”, hogy a perifériavezérlő megkezdte az írásművelet végrehajtását, és így a 2. lépésben ragadna.

1.2.1.3. Órajeltartományok

A sok részegységből álló rendszerekben az egyes almodulok feladatuktól függően más-más frekvenciájú, különálló órajelforrásokról járhatnak. Ez a széles körben alkalmazott rendszertechnikai megoldás elsősorban a fogyasztás optimalizálását szolgálja (lásd DRMSz: 6.3. alfejezet). Minden almodult csak olyan magas frekvenciájú órajelről érdemes járatni, ami feladatának teljesítéséhez feltétlenül szükséges, hiszen – CMOS kapcsolástechnikát feltételezve – a szinkron áramkörök fogyasztása órajelük frekvenciájával egyenesen arányos. Egy 9600 bps bitsebességű UART áramkört például felesleges lenne 100 MHz frekvenciájú órajellel meghajtani, hiszen az áramkör az órajel felfutóéleinek túlnyomó részét csak arra használja, hogy segítségükkel egy számláló értékét növelje, várva, hogy elteljen egy bit átviteléhez szükséges idő. Az időmérés magas frekvenciájú órajellel persze pontosabb, de az alkalmazások többségében elfogadható pontosság sokkal alacsonyabb frekvenciájú órajellel is biztosítható.[12] Ugyanakkor egy processzor esetén a műveletvégzés sebessége arányos az órajel frekvenciájával, azt tehát érdemes magasabbra állítani.

A részletes tervezés szempontjából fontos lesz az, hogy pontosan mit is értünk „különálló órajelforrások" alatt. Az FPGA áramkörök tipikusan több fáziszárt hurkot (**P**hase-**L**ocked **L**oop, PLL) is tartalmaznak. Ezek az áramkörök alkalmasak több, egymáshoz képest szinkron órajel előállítására, amelyek frekvenciája eltérő lehet, fázisviszonyuk viszont ismert és időben állandó. Feltételezhetnénk, hogy rendszerünk almoduljait ilyen, egyazon PLL által előállított, tehát egymáshoz képest szinkron órajelekről hajtjuk meg. Ebben az esetben az almodulok közötti adatátvitel (**C**lock **D**omain **C**rossing, CDC) időzítési problémáit a statikus időzítésvizsgálat (**S**tatic **T**iming **A**nalysis, STA) segítségével kezelhetjük (lásd DRMSz: 6.2. és 8.2.3. alfejezet), RTL-tervezési szempontból tehát ezzel nem kellene foglalkoznunk. Ugyanakkor vegyük észre, hogy ebben az esetben hallgatólagosan azt is állítjuk, hogy az adott FPGA-ban legalább annyi szinkron órajelet tudunk előállítani, ahány féle különböző órajelforrást az adott alkalmazási környezetben összeállított toplevel modul igényel. Ezt a függőséget a konkrét FPGA áramkör és az eszközfüggetlennek tervezett keretrendszerünk között nem engedhetjük meg. A megoldás az, hogy az egyes almodulokat meghajtó órajelforrásokat aszinkronnak tekintjük, vagyis sem frekvenciájukkal, sem fázisviszonyaikkal kapcsolatban nem feltételezünk semmit. Ez RTL-tervezési szempontból nagyon fontos részlet, hiszen így bármely két almodul közötti kommunikációs interfész esetén órajel-szinkronizáló fokozatokat kell beiktatnunk (lásd DRMSz: 6.3. alfejezet).

Az órajel-szinkronizáló fokozatok beépítése általában a toplevel integrátor feladata, hiszen az almodulok tervezésekor nem tudhatjuk, hogy mely almodulok milyen órajelforrásról járnak majd. Ugyanakkor az almodulok újrafelhasználható RTL-modelljének tervezőjeként megkönnyíthetjük a toplevel integrátor feladatát, ha ezeket a szinkronizáló fokozatokat az almodulhoz tartozó „extraként" előre elkészítjük. Technikailag ezt többféle módon is kivitelezhetjük, ezeket a megoldásokat a részletes tervezésről szóló fejezetben mutatjuk be (lásd 2.1. pont). Rendszerszinten egyelőre csak a következő alapelvet rögzítjük:

[12] A nagyenergiájú részecskék által előidézett tranziens jelenségeknek (**S**ingle-**E**vent **T**ransient, SET) kitett rendszerek órajel-frekvenciáját egy másik érv mentén is érdemes a lehető legalacsonyabbra beállítani. Az ilyen tranziensek az egyes flip-flopokat meghajtó kombinációs hálózatok kimenetén helytelen (a nominális funkció szerintivel ellentétes) vagy még rosszabb esetben érvénytelen (sem nem alacsony, sem nem magas) logikai szintet idézhetnek elő. Minél nagyobb az adott flip-flop órajelének frekvenciája, annál nagyobb a valószínűsége, hogy a flip-flop egy ilyen, ideiglenesen helytelen vagy érvénytelen értéket mintavételez, ami funkcionális hibához vagy akár metastabil állapot kialakulásához is vezethet. Az alacsony órajel-frekvencia tehát ezekben az alkalmazásokban a fogyasztás csökkentése mellett a megbízhatóságot is növeli.

*A **szinkron kézfogásos interfésszel** rendelkező, újrafelhasználható IP-magok alapfunkcióit megvalósító RTL-modellek elkészítésekor feltételezzük, hogy az IP-mag beágyazó környezetének órajele megegyezik magának az IP-magnak az órajelével. Ezeket az alapmodelleket különálló, **az alapmodellről leválasztható órajelszinkronizáló fokozatokkal egészítjük ki**, amelyek lehetővé teszik az IP-mag integrálását egy, a sajátjához képest aszinkron órajelről járó áramköri környezetbe annak módosítása nélkül.*

Figyeljük meg, hogy az MMIO-busz működését leíró 1-8. és 1-9. ábrák időzítési diagramjai nem utalnak a rendszerórajelre, csupán események sorrendjét definiálják, ami azt fejezi ki, hogy a *CPU* és az MMIO periféria-áramkörök közötti kommunikáció szabályrendszere az egyes események bekövetkezését nem köti konkrét órajelélhez, sőt, azt sem feltételezi, hogy a kommunikáló felek órajelforrásai azonosak. Az MMIO-busz specifikációjának tehát inherens része az az alapelv, hogy a központi egység és a periféria-áramkörök különálló, aszinkron órajelforrásokról járnak. Ennek megfelelően:

*Az **MMIO-interfésszel** rendelkező IP-magok (CPU és periféria-áramkörök) esetén az **órajel-szinkronizáló fokozatokat** nem opcionális extrákként, hanem az **IP-magok szerves részeként** kezeljük és építjük be azok RTL-modelljeibe.*

1.2.2. CPU – Mikroarchitektúra és utasításkészlet

Erőforrás-felhasználás ◆ Fontos szem előtt tartanunk funkcióbővítő rendszerünk központi feldolgozó egységének szerepét az 1-1. ábrán szemléltetett elképzelés megvalósításában. A teljes rendszer, amelynek segítségével a felhasználó problémákat akar majd megoldani, valójában az üzemeltetést végző mikrovezérlő és az FPGA-ban létrehozott funkcióbővítő rendszer együttese. Az utóbbi belsejében lévő, a teljes rendszer szempontjából másodlagosnak tekintendő *CPU* az üzemeltetői interfész rugalmas megvalósításának eszköze. Ez azt jelenti, hogy nem várunk tőle világmegváltó számítási teljesítményt, számunkra a programozhatóságból eredő rugalmasság a lényeg. A funkcióbővítő rendszer egészét ugyan valóban azért hozzuk létre, hogy többlet számítási teljesítményt nyújtsunk az üzemeltető mikrovezérlőnek, de ezt nem magával a *CPU*-val, hanem céláramkörökkel, a bővítőmodulokkal tervezzük megoldani. A *CPU* szerepe csupán annyi, hogy lehetővé tegye e céláramkörök használatát az üzemeltetői interfészen keresztül. Megállapítható tehát, hogy a *CPU* tervezése során nem nagy számítási teljesítményre, hanem kis méretre célszerű törekednünk, hogy minél több FPGA erőforrás (flip-flopok, logikai kapuk stb.) maradjon a bővítőmodulok számára.

Utasításátlapolás és regiszterkészlet ◆ A tárolt programot végrehajtó áramkörök mikroarchitektúrájának szintjén az erőforrásigény kordában tartásának egyik legjobb eszköze az utasításátlapolás (pipeline) teljes elhagyása. Az utasításátlapolást lehetővé tévő mikroarchitektúra ti. bizonyos belső funkcionális egységek (pl. utasításmutató, utasításregiszter) többszörözését igényelné, ami esetünkben feleslegesen növelné a leendő áramkör méretét, hiszen az általa nyújtott többlet számítási teljesítményre nincs szükségünk. Nem szabad azonban túlzásba esnünk az erőforrásigény csökkentéséért folyó küzdelemben, inkább az egyéb szempontokat is szem előtt tartó, észszerű kompromisszumokra kell törekednünk. Figyelembe kell vennünk például azt, hogy a *CPU*-n futó

programot a felhasználónak kell elkészítenie, ami nem könnyű feladat, ha nem állnak rendelkezésre a magas szintű programozási nyelvek (pl. C, C++) használatát lehetővé tévő fordítóprogramok.[13] Példarendszerünk egyszerűsége okán azzal a feltételezéssel élünk, hogy a processzort a felhasználó assembly nyelven, tehát meglehetősen alacsony szinten programozza majd. A könnyű programozhatóságot segíti elő, hogy viszonylag sok, 16 darab, 16-bites belső regisztert hozunk létre, amelyeknek semmiféle megkülönböztetett szerepe nem lesz a processzor működésében, ezeket a programozó bármire használhatja az assembly program elkészítésekor.[14]

Utasításkészlet ◆ Az assembly nyelvű forráskód elkészítését ugyancsak megkönnyíthetjük az utasításkészlet-architektúra megfelelő megválasztásával. Háromcímes, 16-bites utasításkészletet dolgozunk ki, így az assembly nyelvű programozás aránylag kényelmes, ennek az árát viszont pontosan az erőforrásigény oldalán fizetjük meg. Egy 8-bites, akkumulátoros utasításkészlet például kevesebb alkatrészből megoldható volna, az ilyen processzorok assembly-programozásának horrorisztikus kalandjaitól azonban feltétlenül meg szeretnénk óvni a felhasználót. A gépi utasítások funkcióját az 1-2. táblázat foglalja össze, részletes leírásukat pedig az 5.1.1.1. Függelék tartalmazza.

Utasítás	Utasítás funkciója	Módosított állapotjelzők
i2rfl	RF[IR[11:8]] [15:8] = 0 RF[IR[11:8]] [7:0] = IR[7:0]	Z, N
load	RF[IR[11:8]] = PERIPH[RF[IR[7:4]] + RF[IR[3:0]]]	Z, N
store	PERIPH[RF[IR[7:4]] + RF[IR[3:0]]] = RF[IR[11:8]]	-
add	RF[IR[11:8]] = (RF[IR[7:4]] + RF[IR[3:0]]) % 2^{16}	Z, N
sub	RF[IR[11:8]] = (RF[IR[7:4]] - RF[IR[3:0]]) % 2^{16}	Z, N
and	RF[IR[11:8]] = RF[IR[7:4]] **AND** RF[IR[3:0]]	Z, N
or	RF[IR[11:8]] = RF[IR[7:4]] **OR** RF[IR[3:0]]	Z, N
xor	RF[IR[11:8]] = RF[IR[7:4]] **XOR** RF[IR[3:0]]	Z, N
shl	RF[IR[11:8]] = RF[IR[7:4]] **SHL** unsigned(RF[IR[3:0]])	Z, N
shr	RF[IR[11:8]] = RF[IR[7:4]] **SHR** unsigned(RF[IR[3:0]])	Z, N
bnz	if (Z = 0) then IP = IP + signed(IR[11:0]) else IP = IP+1	-
bnn	if (N = 0) then IP = IP + signed(IR[11:0]) else IP = IP+1	-
jmp	IP = IP + signed(IR[11:0])	-
call	STACK[SP] = EI, Z, N, IP+1 SP = SP+1 EI = 0 IP = IP + signed(IR[11:0])	EI
ret	SP = SP-1 EI, Z, N, IP = STACK[SP]	EI, Z, N
ei	EI = IR[0]	EI

1-2. táblázat. *A CPU utasításkészletének összefoglalása*

[13] Tekintve, hogy egyedi utasításkészletet megvalósító *CPU*-t tervezünk, ilyen fordítóprogramok valóban nem állnak még rendelkezésre.

[14] Egy lehetséges regiszterhasználati konvenciót mutat be az 5.1.2. Függelék. Az ilyen konvenciók következetes használata jelentősen felgyorsíthatja az assembly programok elkészítését.

Az 1-2. táblázat jelölései a következők:

- ***RF** (**R**egister **F**ile): 16 darab 16-bites bejegyzés tárolására alkalmas regisztertömb.*
- ***PERIPH** (Peripheral): A periféria-áramkörök 16 biten címezhető, 16-bites konfigurációs és állapotregisztereket reprezentáló címtartománya.*
- ***IR** (**I**nstruction **R**egister): A végrehajtás alatt álló gépi utasítás kódja.*
- ***IP** (**I**nstruction **P**ointer): A végrehajtás alatt álló gépi utasítás programtárbeli, 16-bites címe.*
- ***Z** (**Z**ero): Állapotjelző, értéke 1, ha a legutóbbi utasítás eredménye 0x0000, egyébként 0.*
- ***N** (**N**egative): Állapotjelző, értéke 1, ha a legutóbbi utasítás eredménye kettes-komplemens kódolású, előjeles egészként értelmezve negatív, egyébként 0.*
- ***EI** (**E**nable **I**nterrupt): Állapotjelző, amelynek 1 értéke esetén a processzor reagál a megszakítási vonalon érkező kérésekre, ellenkező esetben azokat figyelmen kívül hagyja.*
- ***STACK**: 16 darab 19-bites bejegyzés tárolására alkalmas belső veremtár. Megszakításkérés kiszolgálásakor és szubrutinhíváskor (kontextusváltáskor) a processzor itt tárolja el a visszaérési címet (IP) és az állapotjelzőket (EI, Z, N). Részletesen lásd később: „autonóm kontextusmentés”.*
- ***SP** (**S**tack **P**ointer): A belső veremtár címzésére használt, -4bites regiszter.*
- ***signed(…)**: Előjel-kiterjesztés*
- ***[…]**: Indexelés, pl. RF[A]: a regisztertömb A. eleme, IR[7:4]: az utasításregiszter (7…4). helyiértékű bitjei által képezett bináris szám.*

Az 1-2. táblázatban látható, hogy a *CPU* mindössze 16 darab utasítást valósít meg, ami ugyancsak a kis erőforrásigényre való törekvésünk jele, hiszen a kisszámú gépi utasítás egyúttal kisebb méretű dekóderáramköröket is jelent majd. Az ilyen típusú utasításkészleteket megvalósító gépeket minimális utasításkészletű processzoroknak (**M**inimal **I**nstruction-**S**et **P**rocessor, MISP) hívjuk. Megjegyzendő, hogy a kevés utasítás megnehezíti ugyan az eszköz programozását, ez azonban könnyen kezelhető ún. pszeudoutasítások segítségével. Az 5.1.1.2. Függelék ilyen, a hardverben megvalósított gépi utasítások rövidebb-hosszabb sorozataival létrehozható másodrendű gépi utasításokat mutat be. Az assembly nyelvű forráskód előállítása után a pszeudoutasítások helyére a megfelelő gépiutasítás-sorozatokat az assembler, tehát egy, a felhasználói programok írását hatékonyabbá tévő segédszoftver illeszti be a bájtkódba, a processzor maga tehát nem kell hogy képes legyen ezek önálló értelmezésére, ő már csak a natív utasításokból álló programot látja. A pszeudoutasítások használata tehát a processzor erőforrásigényének növelése nélkül könnyíti meg a programozást.

Harvard-architektúra külső boot-memóriával ◆ A processzorunk memóriaszervezési szempontból Harvard-architektúrájú lesz, amelynek sajátossága, hogy a futtatott programot tároló, ún. programtárat és az adatmemóriát fizikailag két különálló memória-áramkör valósítja majd meg (szemben a Von Neumann-architektúrával, amely a programkódot és az adatokat egyazon fizikai memóriában tárolja). Ily módon az alkalmazásról alkalmazásra változó perifériakészletet reprezentáló címtérkép konstrukciója során nem kell foglalkoznunk a futtatott program méretével és a memórián belüli elhelyezkedésével, hiszen az egy külön memóriamodulban foglal helyet. A regularitás jegyében az utasításmutató regiszter *(Instruction Pointer, IP)* is 16-bites lesz, ami azt jelenti, hogy a programtár legfeljebb 65.536 darab gépi utasítást (128 kB) tartalmazhat, ebbe kell belefférnie a felhasználó által készített programkódnak. Ez a méret nem indokolja külső memória-áramkör használatát, az FPGA-kban található belső statikus RAM-modulok, az ún. blokk-RAM-ok jellemző kapacitása a programtár belső kialakítását is lehetővé teszi, ráadásul ilyen

módon a processzor és a programtár közötti interfész a lehető leggyorsabb lehet, ami kedvező lesz a programfuttatás sebessége szempontjából.[15] Figyelembe kell vennünk azonban, hogy a blokk-RAM-ok illékony memóriák, vagyis tápfeszültség nélkül tartalmukat nem képesek megőrizni. A programtár inicializálásáról tehát minden bekapcsoláskor gondoskodnunk kell (lásd 1.2.3.1. pont: *BOOTSTRAP* modul).

Autonóm kontextusmentés ◆ Szubrutinhíváskor (lásd 5.1.1.1. pont: *call* utasítás) a processzor a programtárban az ún. ugrási címre, a szubrutin első utasítására (belépési pontjára) ugrik. A szubrutin végrehajtása után egyrészt visszatér a hívás helye utáni utasításra (az ún. visszatérési címre), másrészt visszaállítja belső állapotjelzőinek *(EI, Z, N)* értékét közvetlenül a hívás előtti állapotukra. Ehhez szükség van egy memóriára, ahol a szubrutin végrehajtása alatt tárolhatjuk mindezeket az információkat, amelyek együttesét kontextusnak hívjuk. Az adatmemória kézenfekvő megoldásnak tűnhet, viszont az MMIO perifériakezelés miatt az adatmemória nem szükségképpen van jelen a rendszerben, hiszen azt az adott alkalmazás és az ahhoz létrehozott toplevel architektúra nem feltétlenül igényli.[16] Annak érdekében, hogy a szubrutinhívások – és az ezekkel egyenértékű megszakítási rutinok (lásd később) – az adatmemória meglététől függetlenül megvalósíthatók legyenek, a processzor műveletvégző egységében kifejezetten erre a célra létrehozunk egy külön memóriamodult, amit a vezérlőegység által automatikusan kezelt *SP (Stack Pointer)* regiszterrel címezve veremtárként *(STACK)* használunk. A veremtár 16 bejegyzés tárolására lesz képes, ami akár 16 darab egymásba ágyazott szubrutinhívást is lehetővé tesz. A programozó szempontjából szintén fontos képesség, hogy a processzorunknak lesz megszakítási rendszere, vagyis képes lesz egy külső jel hatására is szubrutinhívást (ebben az esetben ez az ún. megszakításkiszolgáló rutin: **I**nterrupt **S**ervice **R**outine, ISR) végrehajtani. Ez a szubrutinhívás abban különbözik a *call* utasítás végrehajtásától, hogy míg utóbbi esetben az ugrási cím magába a *call* utasításba van belekódolva, addig megszakítás esetén ez a cím rögzített, a programtárnak egy kitüntetett címe, amelyet szintézis-paraméterként valósítunk meg.

1.2.3. A funkcióbővítő rendszer toplevel architektúrája

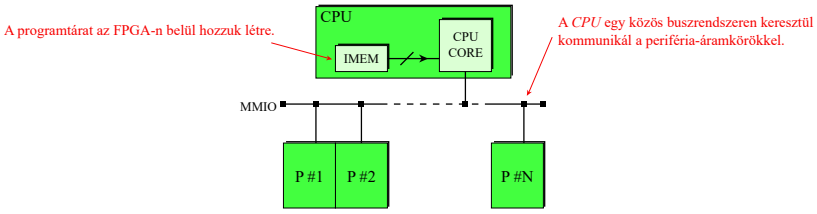
1.2.3.1. A programtárak inicializálása

Elhatároztuk tehát, hogy funkcióbővítő rendszerünk egy processzorból *(CPU)* és ahhoz egy buszrendszeren keresztül kapcsolódó periféria-áramkörökből áll majd (lásd 1-10. ábra).

Azt is rögzítettük, hogy a processzor belső programtárát *(IMEM)* az FPGA-n belül, a belső blokk-RAM-modulok segítségével valósítjuk meg, mert a kapacitásigény (max. 128 kB) önmagában nem indokolja külső memória használatát, továbbá ilyen módon tudjuk maximalizálni az utasítás-végrehajtás sebességét. További tisztázandó rendszertechnikai kérdés, hogy maga a programkód hogyan kerül bele a programtárba. A programtár inicializálására kétféle lehetőséget biztosítunk:

[15] Ugyan a rendszert szándékosan nem a műveletvégzés sebességére optimalizáljuk, de a külső memóriából való közvetlen programfuttatás irreálisan lelassítaná a műveletvégzést. Egy nagysebességűnek mondható, párhuzamos interfésszel ellátott EEPROM hozzáférési ideje kb. 100 ns, ami egy nagyságrenddel nagyobb, mint ami blokk-RAM-mal elérhető egy közepesnek tekinthető, 100 MHz-es belső rendszerórajelet feltételezve.

[16] A funkcióbővítő rendszerben mindenképpen lesz adatmemória, de magát a *CPU*-t is az újrafelhasználhatóságot figyelembe véve tervezzük meg, így számítanunk kell rá, hogy más alkalmazási körülmények között nincs adatmemória.

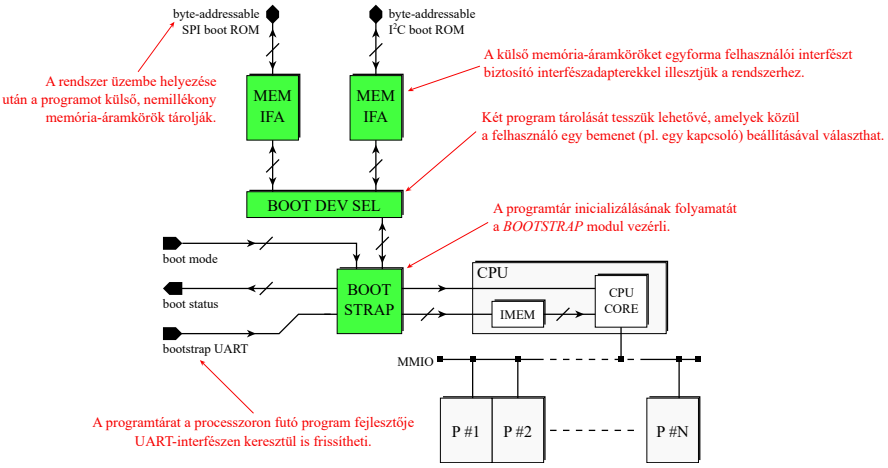


1-10. ábra. *A CPU, a programtár és a periféria-áramkörök összekapcsolása*

1. Lehetővé tesszük a felhasználó számára, hogy a processzoron futó program kidolgozása során a lehető legegyszerűbb módon tudja a programtár tartalmát frissíteni, hiszen erre a műveletre a fejlesztés ezen szakaszában tipikusan elég gyakran lesz szüksége. Ha feltesszük, hogy ez a fejlesztés egy asztali számítógépen történik, akkor kézenfekvő itt is az UART-kommunikációhoz nyúlnunk, ui. ez a protokoll az alapja az asztali számítógépek soros portjának is, számos segédszoftver (ún. terminálprogram) rendelkezésre áll – operációs rendszertől függetlenül – ennek kezelésére. A mai modern asztali számítógépek a soros portok használatát már jellemzően USB-n (**U**niversal **S**erial **B**us) keresztül, ún. virtuális COM-portok létrehozásával oldják meg, mert így nem szükséges különálló, nagyméretű csatlakozót beépíteni a soros portoknak. A funkcióbővítő rendszert magában foglaló FPGA szempontjából ez azt jelenti, hogy szükség lesz egy külső USB/UART-átalakítóra, mert az FPGA maga ilyen eszközt nem tartalmaz.
2. A programtárat megvalósító blokk-RAM-ok illékony memóriák, vagyis a kész rendszer üzembe helyezése után valamiképp gondoskodni kell a programtár autonóm inicializálásáról. Ehhez egyrészt tárolni kell valahol magát a programot, illetve szükség lesz egy mechanizmusra, ami ezt a programot a tápfeszültség bekapcsolásakor betölti a programtárba. A program nemillékony tárolására bizonyos FPGA-k nyújtanak ugyan megoldást, de azt semmiképp sem állíthatjuk, hogy erre minden FPGA képes. Ezért az eszközspecifikus belső nemillékony memóriák használata helyett a kész programok tárolására külső memória-áramköröket, ún. boot-memóriákat használunk.[17]

A programtár – és egyúttal a boot-memóriák – tartalmának inicializálására egy különálló áramkört fejlesztünk ki, amit *BOOTSTRAP* modulnak hívunk (lásd 1-11. ábra).

[17] Érdemes megjegyezni, hogy ha a fejlesztési fázis során használt UART-vonalat az üzemeltetői feladatot ellátó mikrovezérlőbe kötjük be, akkor a külső, nemillékony programmemória is feleslegessé válik. Ekkor persze minden bekapcsoláskor az üzemeltető mikrovezérlőnek kell feltöltenie a programtárat UART-on keresztül, ami nagyméretű program esetén kényelmetlenül sokáig tarthat. 115.2 kbps jelzési sebesség (baud rate) esetén egy maximális méretű programtár inicializálása protokolltól függően akár 12-18 másodpercet is igénybe vehet, de adott alkalmazásban a külső memória megspórolása kárpótolhat ezért a kellemetlenségért. Az itt vázolt rendszerterv lehetővé teszi, hogy ezt a döntést a felhasználó hozza meg.



1-11. ábra. *A belső programtár inicializálásáért felelős BOOTSTRAP modul és a boot-memóriák interfészadapterei*

A *BOOTSTRAP* modulnak több feladata és üzemmódja lesz, amelyek közül a modul – *reset* jelének elengedésekor (praktikusan a bekapcsolás pillanatában) – két bemenet (az FPGA két kivezetése) pillanatnyi értéke alapján választ:

- **A *CPU* belső programtárának közvetlen inicializálása UART-on keresztül**: Ebben az üzemmódban a *BOOTSTRAP* modul reset-ben tartja a processzort, és várakozik az UART-on érkező, a programot tartalmazó bájtfolyamra. A bejövő utasítások bájtjait a modul közvetlenül a processzor belső programtárába írja be. Ha a bájtfolyam hibátlanul megérkezett és az összes utasítás a helyére került, a modul elengedi a processzor reset bemenetét, elindítva a program futtatását.
- **A külső boot-memória inicializálása UART-on keresztül**: Ez az üzemmód lehetővé teszi programok elhelyezését a külső boot-memóriákban. Az előző üzemmódhoz képest a különbség csupán annyi, hogy a *BOOTSTRAP* modul a teljes folyamat során és annak végén is reset-ben tartja a processzort, az utasítások bájtjait pedig nem a processzor belső programtárába, hanem a külső boot-memóriák valamelyikébe írja az annak fizikai és protokolláris sajátosságait „eltakaró”, ún. interfészadapter *(MEM IFA, **M**emory **I**nterface **A**dapter)* áramkörön keresztül (lásd 4.4.1. és 4.4.2. pont).
- **A *CPU* belső programtárának inicializálása boot-memóriából**: Ebben az üzemmódban az UART-interfésznek nincs szerepe. A *BOOTSTRAP* modul bekapcsolás után reset-ben tartja a processzort mindaddig, amíg a boot-memóriában tárolt programot át nem másolta a processzor belső programtárába. Ekkor a processzor reset bemenetét elengedi, elindítva a program futtatását.

Az 1-11. ábrán látható, hogy a *BOOTSTRAP* modulnak csupán egy boot-memória interfésze van, amelyhez egy kiválasztó logikán *(BOOT DEV SEL)* keresztül a korábban említett interfészadapter áramkörökből két példány csatlakozik. Így lehetőség van két különböző program tárolására is, amelyek közül egy bemenet beállításával választhatunk. Az interfészadapterek szerepe az, hogy a külső eszközök, ebben az esetben például a boot-memóriák eszközfüggő sajátosságait eltakarják

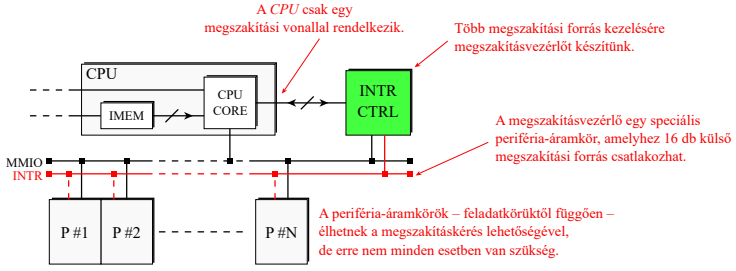
az azokat használó egyéb almodulok elől. Az interfészadapterek kliensinterfészei egyformák (lásd 1.2.1.1. pont), és pontosan illeszkednek a *BOOTSTRAP* modul boot-memória interfészéhez. A különböző interfészadapterek belső működése természetesen eltér, hiszen különböző eszközökkel kell együttműködniük. Esetünkben ez két, jelentősen eltérő viselkedésű EEPROM (**E**lectrically **E**rasable **P**rogrammable **R**ead-**O**nly **M**emory) lesz, amelyek közül az egyik SPI, a másik pedig I^2C (**I**nter-**I**ntegrated **C**ircuit) interfészen írható és olvasható. Figyeljük meg, hogy ebben a struktúrában a *BOOTSTRAP* modulnak nem kell tudnia azt, hogy pontosan milyen típusú külső eszközben tároljuk a programot, mindössze az interfészadapterek egységes kliensinterfészének használatával kell tisztában lennie. Ily módon a *BOOTSTRAP* modul belső működése teljesen független lesz a boot-memóriák konkrét típusától, ami újrafelhasználhatóság szempontjából nagyon kényelmes. Ha a rendszerbe egy harmadik típusú, például valamilyen egyedi párhuzamos interfésszel ellátott külső memóriát szeretnénk beilleszteni, akkor csak annyi a dolgunk, hogy elkészítsük a hozzá tartozó interfészadaptert, amelynek kliensinterfésze pontosan megegyezik a másik két adapterével. Ez a forgatókönyv jól szemlélteti az 1.2.1.1. pontban említett, egységesített interfész-specifikációkra való törekvésünk létjogosultságát.

1.2.3.2. Megszakításvezérlő

A megszakítás mint programozástechnikai eszköz nagyon hatékony megoldást kínál arra, hogy a processzor látszólag egyszerre több feladatot is ellásson úgy, hogy az egyes feladatok között akár dinamikusan változó prioritási sorrendet is felállíthassunk.

A megszakítási vonal aktiválásakor – ha a megszakítások engedélyezettek (lásd *ei* utasítás) –, a processzor befejezi az aktuális gépi ciklust, az állapotjelzőket *(EI, Z, N),* illetve az utasításmutató értékét (a kontextust) a belső veremtárba *(STACK)* menti és a végrehajtást a program egy rögzített pontján (megszakítási vektor), az ún. megszakításkiszolgáló szubrutin végrehajtásával folytatja. A megszakítást kérő modul a kiszolgálás megkezdéséről nyugtázást kap. A megszakításkiszolgáló szubrutin végén a processzor a veremtárból visszaállítja az utasításmutató és a belső állapotjelzők értékét a megszakításkérés előtti állapotukra, majd folytatja a főprogram végrehajtását ott, ahol a megszakításkérés miatt korábban abbahagyta.

Maga a processzor csak egy megszakítási vonallal rendelkezik, több megszakítási forrás egyidejű kezelését egy erre a célra tervezett periféria-áramkörrel, a megszakításvezérlővel tesszük lehetővé (lásd 1-12. ábra).



1-12. ábra. *A megszakításvezérlő kapcsolata a CPU-val és a többi periféria-áramkörrel*

Több megszakítási forrás esetén a megszakítás folyamata a következőképpen megy végbe:

1. A megszakításkérés forrása értesíti a megszakításvezérlőt.
2. A megszakításvezérlő érzékeli a megszakításkérést, azonosítja annak forrását, majd értesíti a processzort.
3. A processzor a megszakításkérés hatására elmenti állapotát és elkezdi végrehajtani a megszakításkiszolgáló rutint ugyanúgy, mintha csak egyetlen megszakítási forrás lenne a rendszerben.
4. A processzor megszakításkiszolgáló rutinjának feladata egyetlen megszakítási forrás esetén nem több, mint hogy végrehajtsa az ahhoz rendelt utasítássorozatot. Ha több forrás is van, akkor a megszakítási rutinnak először meg kell állapítania, hogy melyik forrás küldött megszakításkérést, majd meg kell hívnia az ehhez a megszakítási forráshoz tartozó kiszolgáló szubrutint. Ennek megfelelően definiálhatjuk a „központi megszakításkiszolgáló rutin" és a „perifériaspecifikus megszakításkiszolgáló rutin" fogalmakat.
5. Ha a perifériaspecifikus megszakításkiszolgáló rutin lefutott, akkor a műveletvégzés visszatér a központi megszakításkiszolgáló rutinba, amely a műveletvégzést visszairányítja a főprogramba.

A fent vázolt esetben a központi megszakításkiszolgáló rutin feladata eléggé kötött, ami lehetővé teszi, hogy előre kidolgozott sablon alapján valósítsuk meg, amely minimális módosítással az adott alkalmazás igényei szerint testreszabható. Az 5.1.3. Függelék egy konkrét példán keresztül mutatja be egy ilyen szubrutin-sablonnak a működését.

1.2.3.3. Memóriarendszer

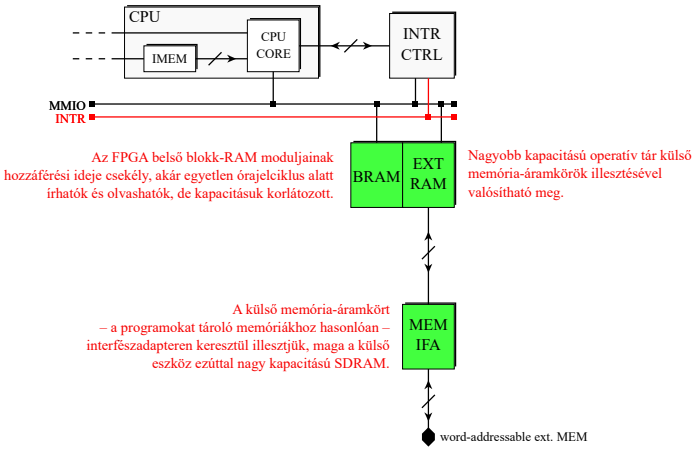
Bár nagyon egyszerű feladatok esetén a belső állapotveremtár lehetővé teszi, hogy akár adatmemória nélkül is üzemeltethessük a processzort, bonyolultabb programok futtatása gyakorlatilag lehetetlen volna operatív tár nélkül.[18] Mivel a processzoron futtatni kívánt algoritmusok összetettségéről nincs információnk – hiszen azok megírását a felhasználóra bíztuk –, ezért olyan memóriarendszert alakítunk ki, amely képességek és erőforrás-felhasználás szempontjából rugalmasan képes alkalmazkodni az igényekhez.

Az 1-13. ábra szemlélteti a kétféle fizikai tárolót tartalmazó memóriarendszer szerkezetét.

- **Blokk-RAM** *(BRAM):* Az FPGA belső blokk-RAM moduljainak óriási előnye, hogy akár egyetlen órajelciklus alatt is hozzáférhetők, függetlenül attól, hogy írásról vagy olvasásról van szó. Hátrányuk, hogy nincs belőlük túl sok; közepes méretű FPGA-k esetén néhányszor száz kbit, nagyobb eszközök esetén néhány Mbit nagyságrendű kapacitás áll rendelkezésre, amelyen az összes almodulnak osztoznia kell.
- **Külső memória-áramkör illesztése** *(EXTRAM)*: Ha nagyobb tárkapacitásra van szükségünk, akkor külső memória-áramkört kell használnunk. A külső eszköz sokféle lehet, használhatunk egyszerű SRAM (**S**tatic **R**andom **A**ccess **M**emory) chipeket, SDRAM-ot

[18] Erre jó példa az 5.1.3. Függelékben bemutatott központi megszakításkiszolgáló rutin. Ez a megoldás egy, az adatmemóriában kialakított veremtárat igényel.

(**S**ynchronous **D**ynamic **R**andom **A**ccess **M**emory), flash-memóriát vagy akár EEPROM-ot is, ahogy a programok tárolásánál tettük. Ezeknek az eszközöknek a tárkapacitása, hozzáférési ideje, fogyasztása és ára is széles skálán változhat, ezért optimális megoldás csak a konkrét alkalmazás ismeretében választható. Ahhoz, hogy a konkrét eszköz és technológia minél kisebb mértékben befolyásolja rendszerünk belső felépítését, a külső memória-áramköröket – a futtatandó programok nemillékony tárolására használt külső EEPROM-oknál látottakhoz hasonlóan – ezúttal is eszközspecifikus interfészadaptereken keresztül illesztjük. Demonstrációs rendszerünkben SDRAM-ot fogunk használni.



1-13. ábra. *A belső blokk-RAM-ot és külső memória-áramkört is tartalmazó memóriarendszer szerkezete*

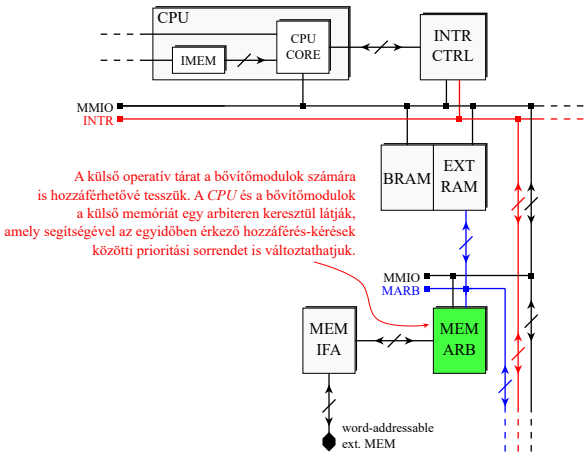
Mindkét memóriatípus esetén törekednünk kell arra, hogy a HDL-modell a rendelkezésre álló tárkapacitás mennyiségét paraméterként kezelje, lehetővé téve a felhasználó számára az erőforrások minél előnyösebb kihasználását a konkrét alkalmazási környezet ismeretében.

Az 1-13. ábrán vázolt rendszerben a külső memóriához közvetlenül csak a processzor férhet hozzá. Annak érdekében, hogy e nagy kapacitású tárhoz a bővítőmodulok is hozzáférjenek a processzor közreműködése nélkül, a rendszerbe beépítünk egy, a memóriát mint megosztott erőforrást kezelő, ún. arbitert (*MEMARB,* lásd 1-14. ábra).

Az arbiter nyolc független kliensinterfészt biztosít, amelyek mindegyikén keresztül kezdeményezhetünk írás- és olvasáskéréseket a külső memória felé. E kliensinterfészek között az arbiter prioritási sorrendet állít fel, amelynek jellege (pl. fix vagy round-robin) futásidőben, az MMIO-buszon keresztül módosítható.

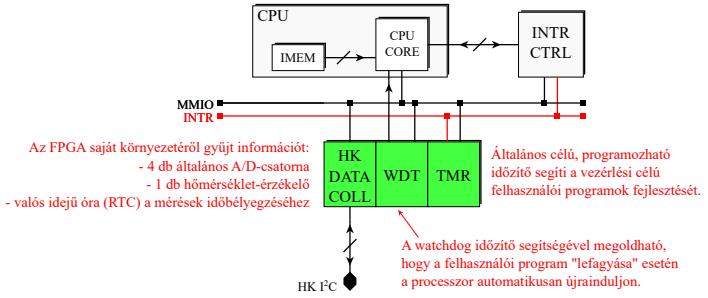
1.2.3.4. Lokális adatgyűjtés, időzítők

A rendszerterv tárgyalásának elején megemlítettük, hogy nincs elvi akadálya annak, hogy a funkcióbővítő rendszer az üzemeltető mikrovezérlőtől nagy távolságra, fizikailag egy teljesen eltérő közegben működjön. Ehhez csupán az üzemeltetői interfészen zajló kommunikáció fizikai rétegét kell megfelelően kialakítani. Ez esetben a funkcióbővítő rendszer az azt üzemeltető mikrovezérlő szempontjából egyfajta összetett, távoli perifériaegyüttesnek is tekinthető. Praktikus, ha egy ilyen



1-14. ábra. *A memória-arbiter elhelyezkedése és interfészei*

távoli alrendszer monitorozza saját környezetének állapotát, és méréseit elérhetővé teszi az üzemeltetője számára. Ezt a célt szolgálja a processzoros rendszerünkbe beépített adatgyűjtő modul (lásd 1-15. ábra: *HK DATA COLL, **H**ouse**k**eeping **Data Coll**ector*), amely egy I^2C-buszra felfűzött hőmérséklet-érzékelő, egy általános célú, négycsatornás A/D-átalakító és egy valósidejű óra (**R**eal-**T**ime **C**lock, RTC) adatait gyűjti össze és teszi elérhetővé az MMIO-buszon a *CPU*, ezáltal közvetetten az üzemeltető mikrokontroller számára. A hőmérséklet-érzékelő segítségével megvalósítható például autonóm lekapcsolás túlmelegedés esetén, az általános célú A/D-átalakító pedig lehetővé teszi, hogy a megfelelő analóg illesztő áramkörökkel bármilyen egyéb érzékelőt is a rendszerbe építhessünk. A valós idejű óra alkalmas mindezeknek a mérési adatoknak az időbélyegzésére, amivel akár a mérési adatok naplózása is lehetővé válik.



1-15. ábra. *Adatgyűjtő és időzítő modulok*

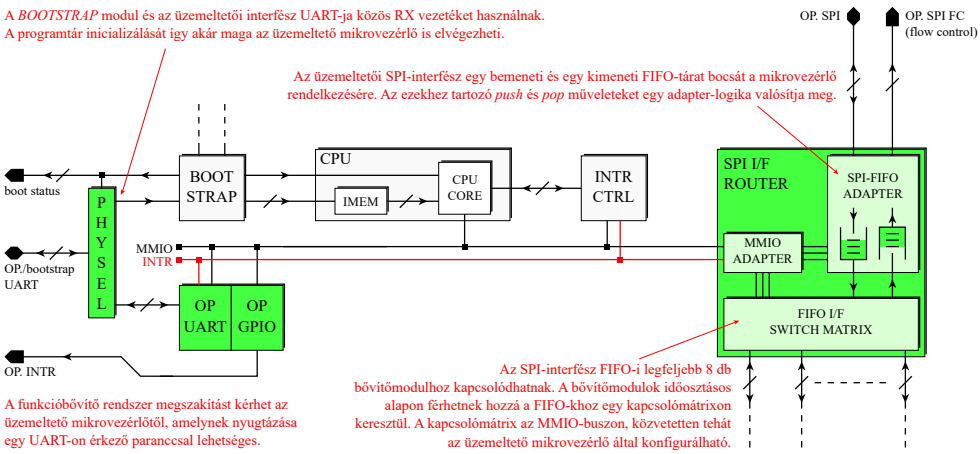
A processzoros rendszerek tipikus kelléke a programozható időzítő áramkör, amelynek feladata, hogy egy, a felhasználói program által meghatározott ciklusidő elteltével periodikusan megszakításkéréseket generáljon. Azáltal, hogy magát a megszakításkiszolgáló rutint a felhasználó írja meg, egy ilyen egyszerű számláló áramkör sokféle funkciót elláthat. Rendszerünkben például könnyen kivitelezhető vele a lokális adatgyűjtő modul másodpercenkénti aktiválása és az időbélyeggel

ellátott mérési adatok elmentése. Mindezt jól átlátható módon, egyetlen megszakításkiszolgáló rutinba szervezhetjük.

Hasonló felépítésű, de jellemző alkalmazását tekintve eltérő almodul az ún. watchdog időzítő *(WDT, **W**atch**d**og **T**imer)*. Ez tulajdonképpen ugyancsak egy egyszerű, MMIO-interfészen programozható időzítő áramkör, amely azonban nem a megszakításvezérlő rendszerhez kapcsolódik, hanem a processzor reset bemenetét hajtja meg. Ez azt jelenti, hogy a felhasználói program által meghatározott periódusidő elteltével a watchdog újraindítja a processzort, feltéve, hogy közben a felhasználói program nem nullázza le a benne található számlálót. Ezt a mechanizmust tipikusan arra használjuk, hogy ha a felhasználói program valami hiba folytán holtpontra (pl. végtelen ciklusba) kerül, „lefagy”, ezáltal nem látja el feladatát (jelesül nem nullázza a watchdog-ot sem), akkor a watchdog ciklusidejének leteltével a teljes rendszert újraindítsuk vele.

1.2.3.5. Az üzemeltetői interfész áramkörei

Az üzemeltetői interfész az 1.2. pontban tárgyaltaknak megfelelően egy aránylag lassú, elsősorban konfigurációs és státuszinformációk közlésére szolgáló UART-összeköttetésből, valamint egy gyors, nagy adatcsomagok hatékony, adatfolyam jellegű átvitelére szánt SPI-kapcsolatból áll. Az ezek megvalósításáért felelős áramköri részleteket az 1-16. ábra szemlélteti.



1-16. ábra. *Az üzemeltetői interfész áramkörei*

UART vezérlőinterfész ◆ A vezérlőinterfész csak akkor működőképes, ha a processzorunk programja már a helyén van, így készen áll parancsok fogadására. Ebből adódóan lehetőség van arra, hogy a *BOOTSTRAP* modul és az üzemeltetői UART közösített *RX* vezetéket használjanak. Amíg a programtár tartalma nem inicializálódott – UART-on keresztül vagy a külső boot-memóriák valamelyikéből –, addig a közösített *RX* vezetéket a *BOOTSTRAP* modulba kötjük. Amikor a programtár inicializálásának állapotát jelző vezetékek (boot status) jelzik a folyamat végét – és a program betöltésének sikerességét –, akkor egy multiplexer logika *(PHYSEL)* átkapcsolja az *RX* bemenetet az üzemeltetői UART-vevő moduljára *(OP UART)*. Így megoldható a programtár inicializálása akár

magával az üzemeltető mikrovezérlővel is anélkül, hogy erre a célra egy külön UART-interfészt alakítanánk ki a két eszköz között.

Az UART-interfész adatkapcsolati réteg feletti protokolljának tervezését ugyan a felhasználóra bíztuk, de feltételezésekkel itt is élhetünk a rendszer használatának megkönnyítése érdekében. Mivel a vezérlőinterfészen futásidejű paraméterek beállítása és státuszinformációk lekérdezése zajlik, feltehetjük, hogy a felhasználó ezt egy egyszerű kérdés-válasz protokoll segítségével oldja meg. Amikor az üzemeltető mikrovezérlő be akarja állítani valamelyik konfigurációs regiszter értékét, akkor elküld egy üzenetet, amire egy nyugtázó választ vár a művelet sikerességéről. Olvasás esetén ugyancsak egy kérdés-válasz tranzakcióról lehet szó, de itt a kiolvasott regiszter értékét is bele kell kódolni a válaszcsomagba. Egy ilyen protokoll programozástechnikailag mindkét fél számára könnyen kezelhető, gondot okozhatnak azonban azok a helyzetek, amikor a funkcióbővítő rendszer úgy szeretne közölni valamit, hogy az üzenete nem egy kérdésre adott válasz, hanem például egy azonnali beavatkozást igénylő hibajelenségre utaló figyelmeztető jelzés. Az UART ilyen esetben túl lassúnak bizonyulhat, ezért egy plusz vezetéket, egyetlen általános célú kimenetet hozunk létre az ilyen helyzetek kezelésére *(OP. INTR).* Ezt a kimenetet a felhasználó bekötheti az üzemeltető mikrovezérlő megszakítási rendszerébe, így a különleges helyzetekre a lehető leggyorsabban tud reagálni. A megszakítás nyugtázása már nem időzítéskritikus, így ez megoldható egyetlen UART-üzenet segítségével, amelynek hatására a megszakítási vonal nullázható.

SPI adatinterfész ◆ Az SPI tipikus bitsebessége egy-két nagyságrenddel meghaladja az UART-ét, így nagymennyiségű adat átvitelére alkalmas. Ennek természetesen ára van, ez az interfész a kétirányú kapcsolat megvalósításához legalább három vezetéket igényel *(OP. SPI),* ti. az átvitel sebessége azáltal növelhető ilyen mértékben, hogy az adatvonalak mintavételezéséhez használt órajelet (**S**erial **Cl**oc**k**, SCLK) is átvisszük a két eszköz között.[19]

Egy SPI-tranzakció során a master egység – ez esetben az üzemeltető mikrovezérlő – N × 8 órajelimpulzust generál, amellyel szinkronban változtatja a MOSI (**M**aster **O**utput **S**lave **I**nput) vonal szintjét, egyúttal mintavételezi a slave egység – az FPGA-ban helyet foglaló interfészáramkör – által meghajtott MISO (**M**aster **I**nput **S**lave **O**utput) vonalat. A művelet során N bájt információ kerül át a master egységből a slave-be és fordítva. Az FPGA-ban lévő soros-párhuzamos átalakító áramkör *(SPI-FIFO ADAPTER)* feladata, hogy a MOSI vonal mintavételezésével nyert bájtokat egy bemeneti FIFO-tárolóba mentse, illetve egy kimeneti FIFO-tárolóból kiolvassa a MISO vonalra bitenként kiírandó bájtokat.

Nagy átviteli sebességet úgy érhetünk el, ha ezek a tranzakciók minél sűrűbben követik egymást, illetve ha a folyamatot kordában tartó forgalomszabályozási mechanizmus – ami gondoskodik róla, hogy a teli FIFO-ba ne próbáljunk adatot írni, az üresből pedig ne olvassuk ki a nem létező adatot – ugyanolyan gyors, mint maga az adatátvitel. A FIFO-k telítettségi állapotát hozzáférhetővé tesszük ugyan az MMIO-buszon, közvetetten tehát az üzemeltetői UART-on keresztül is, de ennek a visszajelzésnek a sebessége több nagyságrenddel elmarad az SPI-interfészétől. Ezért bevezetünk két dedikált kimenetet az SPI-vonalak mellé *(OP. SPI FC, **F**low **C**ontrol)*, amelyek rendre a be- és kimeneti FIFO-k *full* és *empty* állapotjelzőit képviselik. Az üzemeltető mikrovezérlő tehát

[19] Az UART esetén – és általában az aszinkron adatátviteli megoldásoknál – az időmérés pontossága gátat szab a sebesség növelésének.

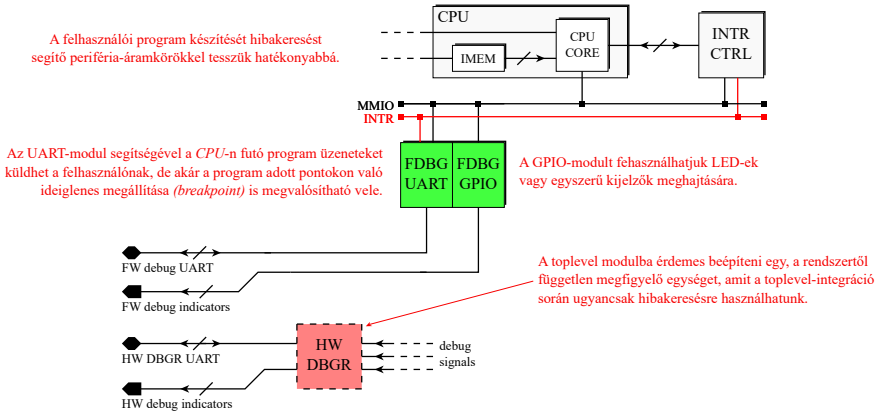
minden tranzakció elején csupán ezeknek a vezetékeknek az olvasásával meggyőződhet róla, hogy indítható-e a következő tranzakció.

Felhasználjuk továbbá a standard SPI-interfészhez tartozó CS (**C**hip **S**elect) jelet is, bár eredeti céljától eltérő módon. Míg egy általános SPI-interfészen ezt a jelet – ahogy arra elnevezése is utal – az egyazon adatvezetékeket használó eszközök közül az éppen megszólítottnak a kiválasztására használjuk, addig itt ennek a jelnek a lefutóéle fogja jelezni az üzemeltető mikrovezérlő tranzakciós szándékát. E lefutóél váltja majd ki a kimeneti FIFO legelső pop-olását annak érdekében, hogy az órajelimpulzusok megérkezésekor a MISO kimenetre juttatandó bitek már rendelkezésre álljanak.

Az SPI-interfészhez tartozó be- és kimeneti FIFO-kból csupán egy-egy lesz a rendszerben, amelyeket egyszerre csak egy, időosztásos alapon legfeljebb nyolc bővítőmodulnak bocsátunk a rendelkezésére. Az összeköttetés a FIFO-k és a bővítőmodulok között egy kapcsolómátrixon keresztül valósul meg, amelynek konfigurációját az MMIO-buszrendszeren keresztül közvetetten az üzemeltető mikrovezérlő határozza meg, ő dönt arról, hogy az adott pillanatban melyik bővítőmodulhoz rendeli hozzá a FIFO-kat.[20] A FIFO-k és az MMIO-interfészen konfigurálható kapcsolómátrix együtt alkotják az ún. SPI-FIFO útválasztót *(SPI I/F ROUTER)*.

1.2.3.6. Hibakeresést segítő modulok

Az assembly programok készítése során a felhasználónak a lehetőségei programja futásának ellenőrzésére korlátozottak. Tulajdonképpen mást nem tehet, mint hogy műszerekkel (pl. oszcilloszkóppal vagy logikai analizátorral) megfigyeli a külső eszközökhöz tartozó kommunikációs vonalakat, vagy azok viselkedését más módon ellenőrzi. Ez egy bizonyos összetettség felett borzasztó körülményes lehet, amin könnyen segíthetünk olyan almodulok beépítésével, amelyek kifejezetten a megfigyelhetőség javítása céljából kerülnek bele a rendszerbe.



1-17. ábra. *Hibakeresést segítő (debug) modulok*

[20] Ehhez természetesen feltételeznünk kell egy, a felhasználó által megírt, a processzoron futó programot, ami ezt lehetővé teszi.

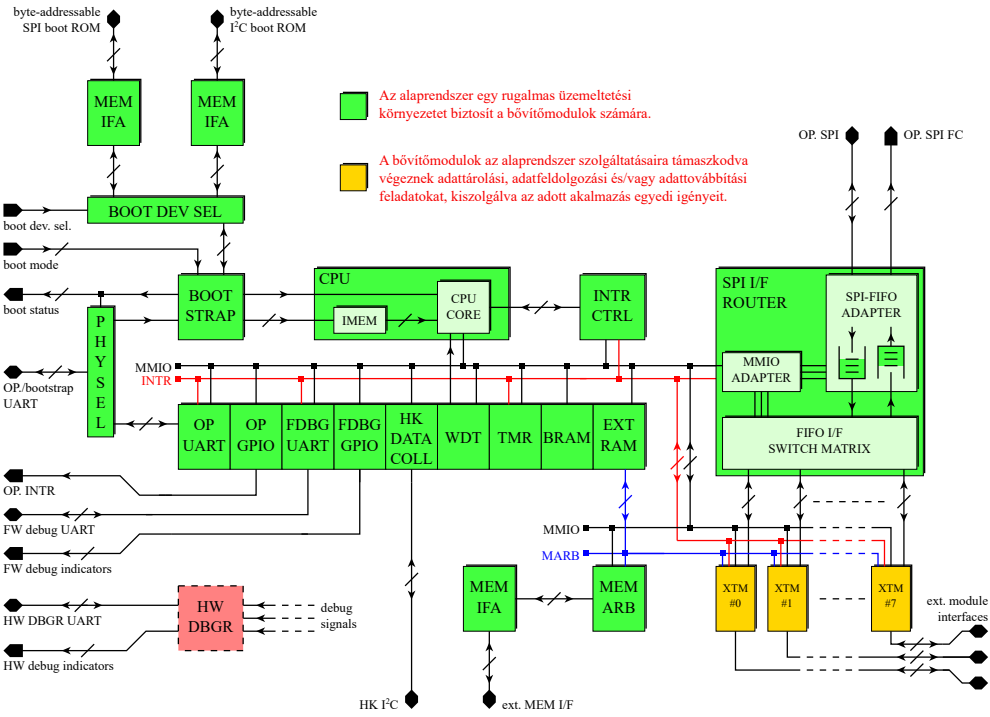
Az 1-17. ábrán látható két, az assembly programok készítését segítő almodul. Az *FDBG (**F**irmware **D**e**b**u**g**) GPIO* egy általános kimenetiport-vezérlő, amit egyszerűen világítódiódákra (**L**ight **E**mitting **D**iode, LED) vagy valamilyen kijelzőre kapcsolva hatékony, a felhasználói program futásáról hasznos információkkal szolgáló indikátorokat hozhatunk létre. Az *FDBG UART* egy soros kommunikációs csatornát hoz létre, amit a *BOOTSTRAP* modulnál tárgyaltakhoz hasonlóan kapcsolhatunk a felhasználó számítógépéhez. Ezen keresztül a *CPU*-n futó program az állapotát leíró üzeneteket küldhet a felhasználónak. Az UART-kapcsolat kétirányúságát kihasználva a program adott pontokon való ideiglenes megállításának (breakpoint) mechanizmusa is kidolgozható, ami ugyancsak nagy segítséget jelenthet a hibakeresés során. Mindkét modul az MMIO-buszra kapcsolódik, vagyis a felhasználói program akkor és arra használja ezeket, amire csak akarja.

Az 1-17. ábrán látható, szaggatott körvonallal jelölt modul *(**H**ard**w**are **D**e**b**u**g**ge**r**, HWDBGR)* szerepe hasonló, ez azonban nem egy MMIO periféria-áramkör, hanem a funkcióbővítő rendszertől független, annak megfigyelésére szánt almodul. Célja, hogy a toplevel-integráció során fokozatosan előálló rendszer hardver-prototípusainak tesztelésekor javítsa a megfigyelhetőséget. Az ábrán „debug signals" néven hivatkozott jelek szerepét bármely belső jel betöltheti, amit az adott pillanatban meg szeretnénk figyelni. Ennek a „debugger" logikának a pontos működését nem tervezzük meg előre, hanem az adott szituációban megfigyelni kívánt jelenség jellege szerint alakítjuk ki. Ha egy statikus, időben ritkán változó jel állapotára vagyunk kíváncsiak, akkor elegendő ezt a jelet egy külső LED-re kapcsolni. Ha azonban például az MMIO-busznak egy bizonyos periféria-áramkörre vonatkozó tranzakcióit szeretnénk megfigyelni, akkor olyan logikát hozunk létre, ami képes az MMIO-tranzakciók értelmezésére és adott paraméterek (pl. memóriacím) szerinti szűrésére. Természetesen mindig körültekintően kell eljárnunk, figyelembe véve az UART-kommunikáció és az MMIO-busz jellemző sebessége közötti eltérést. Összetett toplevel modulokban célszerű egy ilyen debugger logikát előkészíteni, mert nagyban megkönnyítheti a HDL-modell fejlesztését azokban a helyzetekben, amikor a funkció összetettsége a logikai szimuláció futtatását már nagyon időigényessé teszi.[21]

1.2.3.7. A bővítőmodulok integrációja

Rendszertervünk eddig bemutatott részei a funkcióbővítő infrastruktúra állandó kellékei. Ennek az alaprendszernek az architektúráját és periféria-áramköreit kifejezetten arra optimalizáljuk, hogy egy kényelmesen használható, rugalmas környezetet teremtsenek a legkülönfélébb bővítőmodulok számára. A „funkcióbővítés" lényegi részét tehát nem az eddig felsorolt almodulok, hanem az ebbe a környezetbe beépíthető, azzal kompatibilis, egyedi periféria-áramkörök képviselik majd. Az 1-18. ábra a teljes rendszer blokkvázlatát mutatja be.

[21] Megjegyzendő, hogy e technika ebben a formában kifejezetten FPGA-specifikus, hiszen a debugger logika a fejlesztés során folyamatosan változik, ami az eszköz sorozatos újrakonfigurálását igényli.



1-18. ábra. *A funkcióbővítő rendszer teljes blokkvázlata*

Kompatibilitáson azt értjük, hogy a bővítőmoduloknak meg kell felelniük bizonyos kritériumoknak, ha az alaprendszer szolgáltatásait ki akarják használni:

- Minden bővítőmodulnak képesnek kell lennie az MMIO-buszra slave egységként csatlakozni. Ezen az interfészen keresztül van lehetőség futásidejű paramétereik beállítására és állapotinformációk lekérdezésére (1-18. ábra: MMIO).
- A bővítőmodulnak – ha funkciója igényli – lehetősége van a *CPU* felé megszakításkérést küldeni, amihez meg kell felelnie a megszakítási rendszer interfészspecifikációjának (1-18. ábra: *INTR*).
- Ha a bővítőmodul nagy mennyiségű információ tárolását igényli, akkor ezt megteheti a külső memória kezelését végző arbiter modul kliensinterfészének specifikációjához igazodva (1-18. ábra: *MARB*).
- Ha a bővítőmodul jellegéből adódóan nagysebességű adatátvitelt igényel az üzemeltető mikrovezérlő felé, akkor kompatibilisnek kell lennie az üzemeltetői SPI-interfész bővítőmodulok közötti időosztásos használatát megvalósító kapcsolómátrix interfészével.

Az ebben a fejezetben rögzített rendszerterv kellően részletgazdag ahhoz, hogy az egyes almodulok RTL-tervezését ez alapján elkezdhessük. A 2. fejezetben az alaprendszer részegységeit (1-18. ábra: zöld blokkok), a 3. fejezetben pedig három, funkcióik és interfészeik szempontjából

tipikusnak tekinthető bővítőmodult mutatunk be (1-18. ábra: sárga blokkok). A 4. fejezet az alaprendszer és a bővítőmodulok tervezése során felbukkanó, újrafelhasználható IP-magként megtervezhető részegységek RTL-terveit tárgyalja.

2

Az alaprendszer RTL-tervezése

A hardverleíró nyelveken alapuló regiszter-transzfer szintű tervezés sajátossága, hogy az alapvető építőelemek egyszerűsége ellenére nagyon gyorsan hozhatunk létre elképesztő komplexitású rendszereket. Míg a regisztereknek (flip-flopoknak), a logikai kapuknak és a multiplexereknek, sőt akár az ezekből felépülő állapotgépeknek a működése is önmagában aránylag egyszerű, addig az ilyen alkatrészek összekapcsolásával kapott hálózatok már egészen bonyolult viselkedésmintákat produkálhatnak, például megvalósíthatnak egy kommunikációs interfészprotokollt vagy egy intelligens szabályozó vagy jelfeldolgozó algoritmust. Mindeközben a HDL-modellek és az azokból szintetizált áramköri struktúrák általános viszonyának ismeretében (lásd DRMSz: 8. fejezet) a tervezés során módunkban áll kézben tartani az erőforrás-felhasználást és -fogyasztást nagymértékben befolyásoló finom részleteket is. Röviden megfogalmazva tehát a hardverleíró nyelvű RTL-modellezés **összetett** rendszerek **részletes** tervezését teszi lehetővé.[22]
A regiszter-transzfer szint mint fogalmi keret és a HDL-ek mint leíró eszközök tehát kétségkívül hatékonyak, de ahhoz, hogy ez a komplexitás kézben tartható maradjon, tartanunk kell magunkat néhány további, fontos irányelvhez.[23]

- ***Újrafelhasználhatóság**: Egy összetett rendszer megtervezése során törekszünk a kontextustól függetlenül értelmezhető részfunkciók azonosítására és az azokat megvalósító almodulok parametrizált megvalósítására.*
- ***Modularitás**: Az egymástól függetlenül értelmezhető funkciókat (szolgáltatásokat) külön almodulok valósítják meg.*
- ***Lokalitás** (egységbe zárás): Minden almodul fekete doboznak tekinthető, egy adott szolgáltatás megvalósítási részletei nincsenek hatással a rendszer teljes egészének működési mechanizmusára, legfeljebb annak teljesítményére. Egy almodulon belüli apró módosítás nem teszi szükségessé a beágyazó rendszer megváltoztatását.*
- ***Regularitás**: A különböző almodulok ugyanazokra a problémákra ugyanazokat a megoldásokat alkalmazzák. Az egyazon szolgáltatásokat egységesített interfészeken keresztül elérhetővé tévő almodulok egymással könnyen kicserélhetők.*
- ***Technológiafüggetlenség**: A soft-IP katalógusok legfőbb jellegzetessége, hogy elvont, jellemzően hardverleíró nyelven megfogalmazott RTL-modellekből állnak, amelyek bármilyen programozható logikai eszközre, bármilyen szintézis-szoftverrel változtatás nélkül szintetizálhatók.*

[22] Óvatosnak kell lennünk azonban az „összetettség" fogalmával, mert ez különböző szövegösszefüggésekben nagyon eltérő dolgokat jelenthet. Nevezhetünk összetettnek egy iteratív osztó áramkört, hiszen például egy egyszerű összeadóhoz képest sokkal több alkatrészt tartalmaz és használata is nagyobb körültekintést igényel, de egy mai, modern okostelefon központi processzoraként használt rendszerchip (**S**ystem-**o**n-**C**hip, SoC) a maga több milliárd tranzisztorával és sokezer oldalnyi felhasználói dokumentációjával nyilván egy másik kategóriát képvisel.

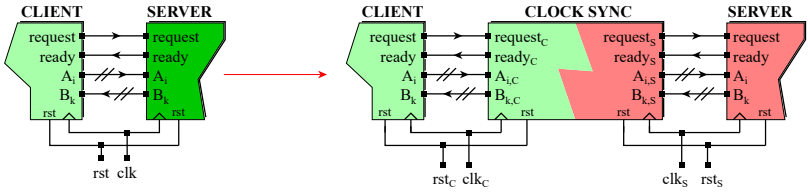
[23] Érdemes megjegyezni, hogy ezek az irányelvek nem az RTL-tervezés sajátjai, érvényességük tulajdonképpen bármilyen tervezési módszerre kiterjed.

Az Előszó néhány gondolatának elolvasása után alighanem az olvasóban is felmerül, hogy az általános elveknek effajta nyers, definíciószerű közzététele a tanulásban nem feltétlenül segít, amíg nincsenek az absztrakt fogalmakhoz köthető konkrétumok, példák. Ahhoz, hogy ezeknek az alapelveknek a lényege jobban átélhető legyen, példarendszerünk almoduljainak részletes RTL-tervezése során egyrészt – nyilván – tartjuk magunkat hozzájuk, másrészt hivatkozunk is rájuk az egyes tervezői döntések meghozatalakor.

2.1. Az órajel-szinkronizáló fokozatok elhelyezése

A rendszerszintű modellben meghatároztunk egységes kommunikációs sémákat (lásd 1.2.1. pont), amelyeket a modularitás és regularitás jegyében igyekszünk a lehető legtöbb konkrét IP-mag esetén alkalmazni. Ugyancsak rendszerszintű döntés, hogy az általános funkciójú (pl. memória), újrafelhasználható IP-magokhoz előre elkészítjük azokat az áramköri részleteket is, amelyek lehetővé teszik, hogy maga az IP-mag egy saját, a környezetéhez képest aszinkron órajelről járhasson. Az órajelforrások egymástól független kezelése a lokalitás alapelvének egy megnyilvánulása, ti. ily módon az IP-mag belső működésének sebessége, ezáltal az egész rendszer számítási teljesítménye egyetlen órajelforrás frekvenciájának megváltoztatásával befolyásolható anélkül, hogy a HDL-modelleken akár csak egy karaktert is módosítanánk. Az órajeltartományok közötti adatátvitel megvalósítására a funkcióbővítő rendszer részegységeinek RTL-modelljeiben háromféle alapsémát használunk:

CDC a szinkron kézfogásos interfészeken ◆ A 2-1. ábra azt az esetet szemlélteti, amikor két, szinkron kézfogásos interfészen összekapcsolt almodul a toplevel integráció során két különálló, aszinkron órajeltartományba kerül, így elkerülhetetlenné válik egy órajel-szinkronizáló fokozat beépítése.



2-1. ábra. *CDC a szinkron kézfogásos interfészeken*

A szinkronizáló fokozat feladata, hogy a két órajelforrást és a hozzájuk tartozó reset jeleket[24] felhasználva megelőzze az órajelek aszinkronitásából adódó metastabilitás okozta hibákat, miközben a két kommunikáló IP-mag elől elfedi, hogy a tranzakció valójában két aszinkron órajelű almodul között megy végbe. Ez utóbbi kritérium azt jelenti, hogy a szinkronizáló fokozat beépítése a két IP-mag HDL-modelljének módosítása nélkül is lehetséges kell hogy legyen (lokalitás), vagyis működés közben a saját nézőpontjából mindkét félnek ugyanazt kell tapasztalnia, mint szinkronizáló fokozat nélküli esetben. A 2-2. ábra egy, a fenti elvárásokat teljesítő szinkronizáló fokozat VHDL-leírását és RTL kapcsolási sémáját mutatja be.

[24] Feltételezzük, hogy minden órajelforráshoz tartozik egy külön reset jel, amelyet a megfelelő reset-szinkronizáló fokozat állít elő (lásd DRMSz: 6.3.2. pont).

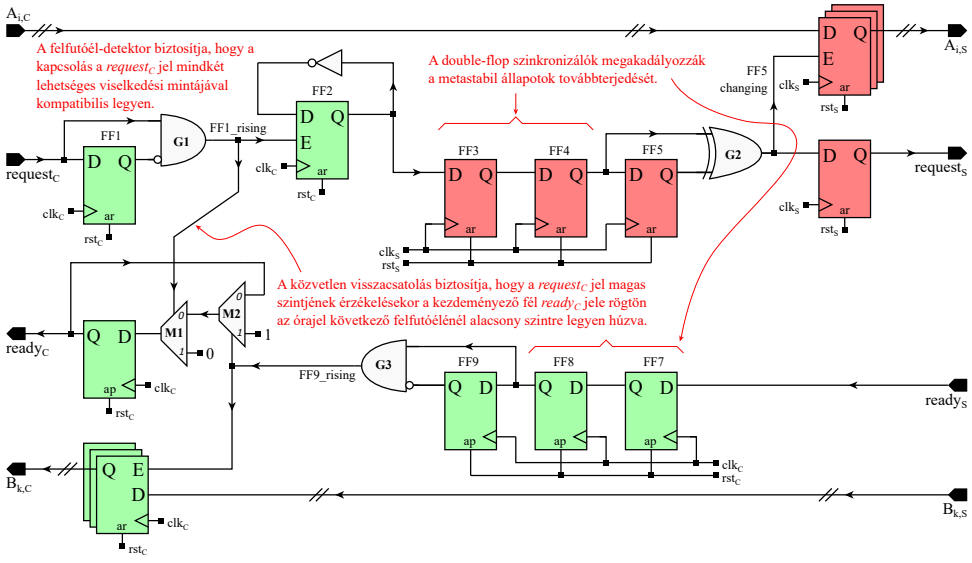
VHDL – a kliens órajeltartománya

```
1  L_CLIENT: process ( clk_c, rst_c )
2  begin
3    if ( rst_c = '1' ) then
4    |  FF1 <= '0'; FF2 <= '0';
5    |  FF7 <= '1'; FF8 <= '1'; FF9 <= '1';
6    |  ready_c <= '1';
7    |  << B_k_c <= ( others => '0' ); >>
8    elsif ( rising_edge(clk_c) ) then
9    |
10   |  FF1 <= request_c;
11   |
12   |  if ( FF1_rising = '1' ) then
13   |    FF2 <= not FF2;
14   |  end if;
15   |
16   |  FF7 <= ready_s;
17   |  FF8 <= FF7;
18   |  FF9 <= FF8;
19   |
20   |  if ( FF1_rising = '1' ) then
21   |    ready_c <= '0';
22   |  elsif ( FF9_rising = '1' ) then
23   |    ready_c <= '1';
24   |    << B_k_c <= B_k_s >>;
25   |  end if;
26   |
27   end if;
28
29 end process;
30
31 FF1_rising <= request_c and not FF1;
32 FF9_rising <= FF8 and not FF9;
```

VHDL – a kiszolgáló órajeltartománya

```
1  L_PROVIDER: process ( clk_s, rst_s )
2  begin
3    if ( rst_s = '1' ) then
4    |  FF3 <= '0'; FF4 <= '0'; FF5 <= '0';
5    |  request_s <= '0';
6    |  << A_i_s <= (others => '0'); >>
7    elsif ( rising_edge(clk_s) ) then
8    |
9    |  FF3 <= FF2;
10   |  FF4 <= FF3;
11   |  FF5 <= FF4;
12   |
13   |  request_s <= FF5_changing;
14   |
15   |  if ( FF5_changing = '1' ) then
16   |    << A_i_s <= A_i_c; >>
17   |  end if;
18   |
19   end if;
20
21 end process;
22
23 FF5_changing <= FF5 xor FF4;
```

Áramköri struktúra



2-2. ábra. *CDC-fokozat a szinkron kézfogásos interfészekhez*

Tranzakció kezdeményezésekor a kliens magas szintre húzza a $request_C$ jelet. Az *FF1* flip-flopból és a *G1 AND* kapuból álló felfutóél-detektor ezt a felfutóélt egyetlen órajel-periódusnyi pozitív impulzussá formálja, függetlenül attól, hogy a kliens a $request_C$ jelet magas szinten tartja, vagy egy órajelciklus elteltével alacsony szintre állítja-e – az interfész-specifikáció mindkét verziót megengedi.

A felfutóél-detektor által előállított pozitív impulzus az *M1* multiplexeren keresztül rögtön az órajel következő felfutóélénél alacsony szintre húzza a kliens $ready_C$ jelét, ezzel párhuzamosan negálja az *FF2* flip-flop értékét. Ez utóbbi változás az *FF3* és *FF4* flip-flopok alkotta double-flop szinkronizáló fokozaton keresztül jut át a kiszolgáló órajeltartományába, ahol az *FF5* flip-flopból és a *G2 XOR* kapuból álló éldetektor egy, immár a kiszolgáló órajeléhez szinkronizált, egyetlen periódusnyi pozitív impulzussá alakítja azt. Ezt a pozitív impulzust használjuk a folyamat során – a specifikációnak megfelelően – stabilan tartott $A_{i,C}$ vektorok mintavételezésére az $A_{i,S}$ regiszterekben. Annak érdekében, hogy a kiszolgáló $request_S$ jele és az $A_{i,S}$ kimenetek értéke – ugyancsak a specifikációnak megfelelően – egyszerre változzon meg, az *XOR* kapu kimenetére még egy késleltető flip-flopot iktatunk be. A kiszolgáló a $request_S$ jel felfutása hatására alacsony szintre állítja a $ready_S$ bemenetet, megkezdi a kérés feldolgozását, majd a művelet végeztével a $ready_S$ jelet újra magas szintre állítja. Ezt a felfutóélt az *FF7* és *FF8* flip-flopokból álló double-flop szinkronizáló fokozat a kliens órajeltartományába juttatja, majd az *FF9* flip-flopból és a *G3 AND* kapuból álló felfutóél-detektor egy, a kliens órajeléhez szinkronizált, egyetlen periódusnyi pozitív impulzust állít elő belőle, amelyet egyrészt a $ready_C$ jel *M2* multiplexeren keresztüli nullázására, másrészt – ezzel egyidőben – a $B_{k,S}$ vektorok mintavételezésére használunk fel a $B_{k,C}$ regiszterekben.

A 2-2. ábrán látható séma a szinkron kézfogásos interfészek többségénél remekül működik, de némi kiegészítésre szorul, ha nulla késleltetésű tranzakciókat is kezelni kell. Ahogy azt az 1-5. ábra is szemlélteti, a nulla késleltetésű tranzakcióknál a kiszolgáló $ready_S$ jelzése mindvégig logikai magas szintű marad, vagyis az *FF9* flip-flop és a *G3* kapu által képezett felfutóél-detektor sosem fog jelezni, így nincs, ami magas szintre húzza a kliens $ready_C$ jelét a tranzakció végén. Ezt a problémát a szinkronizáló áramkör 2-3. ábrán látható módosításával orvosoljuk.

A $request_S$ jelen megjelenő pozitív impulzus és az *M3* multiplexer kombinációjával „kikényszerítünk" egy negatív impulzust a visszacsatoló ágban. Ezután ennek a negatív impulzusnak a felfutóélét ugyanúgy visszük át a kliens órajeltartományába, mint ahogyan a $request_C$ jelét vittük át a kiszolgálóéba; az *FF10* flip-flop és a *G4* kapu egy felfutóél-detektort képeznek, aminek a kimeneti jelét az *FF11* negálására használjuk fel. Ezt az élváltást az *FF7* és *FF8* flip-flopok alkotta double-flop szinkronizáló átviszi a kliens órajeltartományába. A double-flop szinkronizáló kimenetén immár nemcsak felfutó-, hanem lefutóélekre is kell számítanunk, ezért a *G3 AND* kaput egy *XOR* kapura cseréljük. Így az *FF9* flip-flop és a *G3* által képezett, mindkét típusú élváltásra érzékeny éldetektor fogja a kliens $ready_C$ jelzését logikai magas szintre húzni a tranzakciók végén.

A 2-2. és 2-3. ábrákon bemutatott kapcsolások általános vektorszinkronizáló fokozatok, amelyek az A_i vektorokat a kliens órajeltartományából a kiszolgálóéba, a B_k vektorokat pedig a kiszolgálóéból a kliensébe viszik át. E két áramköri modellben ezek a vektorok általános értelműek, minden olyan, akár egy-, akár többites jelet képviselhetnek, amelyeket az órajeltartományok között át akarunk vinni (lásd pl. 4.2.4. pont: véletlen elérésű RAM IP-mag esetén az írási/olvasási cím és a beírandó/kiolvasott adat, FIFO-tároló esetén a beírandó/kiolvasott adat, valamint a *full* és *empty* állapotjelzők).

CDC dual-clock FIFO-val ◆ Az aszinkron órajeltartományok közötti adatátvitelt gyakran olyan illesztőáramkörök beépítésével oldjuk meg, amelyek nemcsak időzítését, hanem logikáját tekintve is egy teljesen más jellegű kölcsönhatást eredményeznek a két kommunikáló fél között, mint amit a szinkronizáló fokozatok nélküli IP-magok önmagukban megvalósítanak. Tegyük fel, hogy a két fél közötti adatáramlás egyirányú és „lazán csatolt" abban az értelemben, hogy a kliens számára

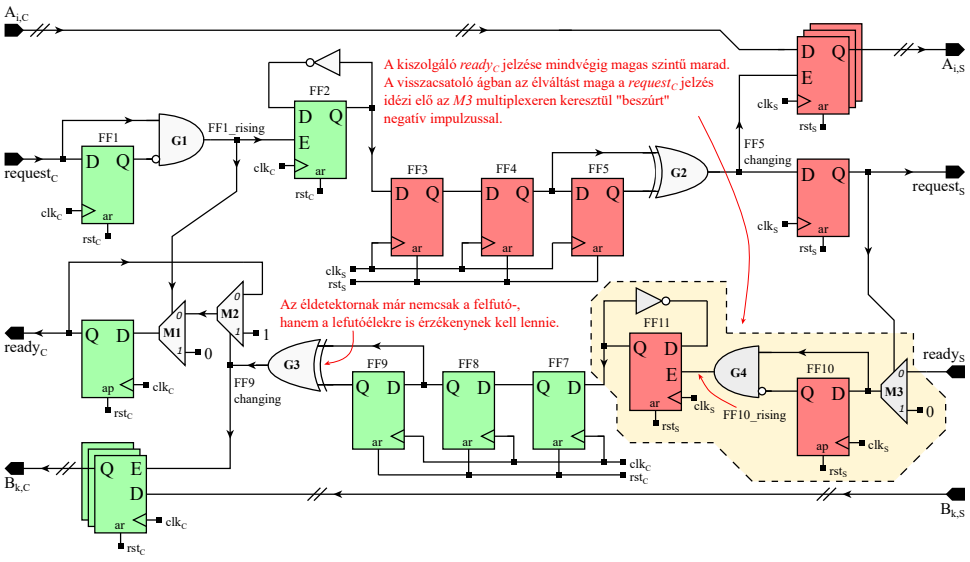
VHDL – a kliens órajeltartománya

```
1  always @ ( posedge clk_c, posedge rst_c )
2  begin
3    if ( rst_c == 1 ) begin
4      FF1 <= 0; FF2 <= 0;
5      FF7 <= 0; FF8 <= 0; FF9 <= 0;
6      ready_c <= 1;
7      << B_k_c <= 0; >>
8    end else begin
9
10     FF1 <= request_c;
11
12     if ( FF1_rising == 1 ) FF2 <= ~FF2;
13
14     FF7 <= FF11;
15     FF8 <= FF7;
16     FF9 <= FF8;
17
18     if ( F1_rising == 1 ) ready_c <= 0;
19     else if ( FF9_changing == 1 ) begin
20       ready_c <= 1;
21       << B_k_c <= B_k_s; >>
22     end
23
24   end
25
26 end
27
28 assign FF1_rising = request_c & ~FF1;
29 assign FF9_changing = FF9 ^ FF8;
```

VHDL – a kiszolgáló órajeltartománya

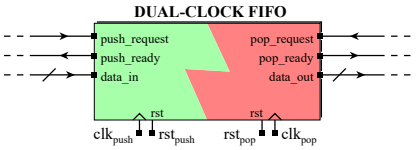
```
1  always @ ( posedge clk_s, posedge rst_s )
2  begin
3    if ( rst_s == 1 ) begin
4      FF3 <= 0; FF4 <= 0; FF5 <= 0;
5      request_s <= 0;
6      << A_i_s <= 0; >>
7      FF10 <= 1; FF11 <= 0;
8    end else begin
9
10     FF3 <= FF2;
11     FF4 <= FF3;
12     FF5 <= FF4;
13
14     request_s <= FF5_changing;
15
16     if ( FF5_changing == 1 ) << A_i_s <= A_i_c; >>
17
18     FF10 <= MX3;
19
20     if ( FF10_rising == 1 ) FF11 <= ~FF11;
21
22   end
23
24 end
25
26 assign FF5_changing = FF5 ^ FF4;
27 assign MX3 = ( request_s == 1 ) ? 0 : ready_s;
28 assign FF10_rising = MX3 & ~FF10;
```

Áramköri struktúra



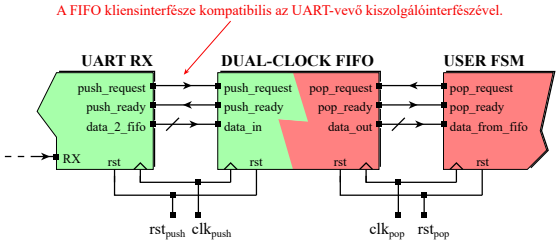
2-3. ábra. *A nulla késleltetésű szinkron kézfogásos tranzakciók kezelésére is alkalmas CDC-fokozat*

irreleváns, hogy a kiszolgálónak küldött adatot az mikor dolgozza fel, vagy – ellenkező irányú adatforgalom esetén – a kiszolgáló mikor állította elő a kiolvasott adatot. A kliens szempontjából csak az a fontos, hogy az egymást követő tranzakciók a lehető leggyorsabban követhessék egymást. Másképp megfogalmazva, azt szeretnénk elérni, hogy az adat átvitele és maga a feldolgozás időben ne legyen szorosan összekapcsolva. Ezt FIFO-tárolók közbeiktatásával tudjuk megoldani. Mivel az FPGA-kban lévő blokk-RAM modulok fizikailag tipikusan két külön interfészen hozzáférhetők,



2-4. ábra. *Két aszinkron órajelről működő FIFO-tároló*

amelyekhez két aszinkron órajelforrás is tartozhat, ezekből könnyen alakíthatunk ki olyan FIFO-tárolókat, amelyek *push* és *pop* logikái fizikailag két külön órajeltartományt képeznek (lásd 2-4. ábra). Az ilyen FIFO-tárolókat dual-clock FIFO-knak nevezzük.[25] A FIFO-tároló *push* és *pop* interfészei egyaránt kliens típusú interfészek, hiszen mind a FIFO-ba való beírás *(push),* mind az onnan való olvasás *(pop)* műveletét egy külső félnek kell kezdeményeznie. Ennek következtében az órajel-szinkronizálás céljából beépített FIFO-tároló egyéb kiegészítő elemek létrehozását is szükségessé teheti. Ennek megértéséhez tekintsük először egy UART soros-párhuzamos átalakító (vevő) modul FIFO-n keresztüli illesztésének 2-5. ábrán látható példáját.



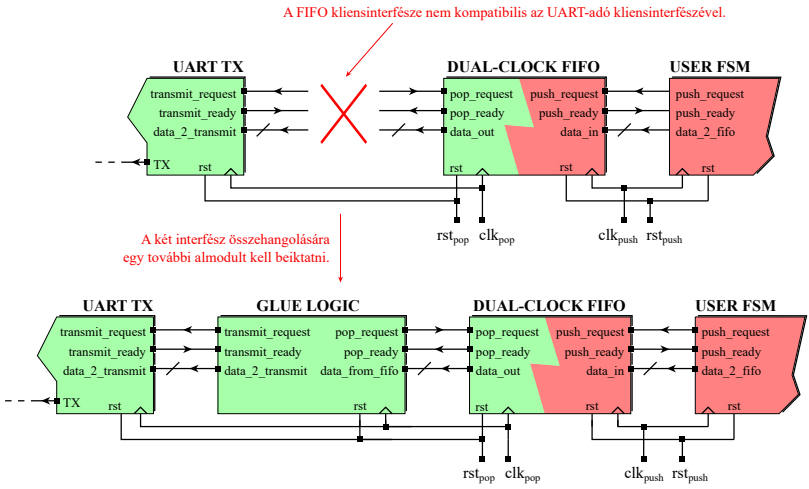
2-5. ábra. *UART vevő modul illesztése FIFO-tárolóval*

Az UART-vevő az adatkereteket azok megérkezésének pillanatában beírja a FIFO-ba. Ebben az esetben az UART-vevő kiszolgáló típusú interfésze áll szemben a FIFO kliens típusú *(push)* interfészével, vagyis az illesztés közvetlenül lehetséges.[26] Ha azonban egy hasonló struktúrát hozunk létre egy UART párhuzamos/soros átalakító (adó) modulhoz, akkor a helyzet némileg bonyolultabb (lásd 2-6. ábra).

Az UART adómodul egy kliens típusú interfészt bocsát a felhasználó rendelkezésére, amelyen keresztül az adatkeretek küldése kezdeményezhető. Ez az interfész egy ugyancsak kliens típusú interfésszel, a FIFO modul *pop* interfészével néz szembe, így a tranzakciókat tulajdonképpen nincs, aki kezdeményezze. Szükség van tehát egy köztes logika (glue logic) beiktatására, ami folyamatosan – vagy legalábbis periodikusan – ellenőrzi, hogy a FIFO-ban van-e kiküldendő adat, illetve hogy az UART adómodul készen áll-e újabb adat fogadására. Ha mindkét feltétel teljesül, akkor az illesztőlogika előbb a FIFO, majd az UART adómodul kliens interfészén kezdeményez egy-egy tranzakciót az adatkeret kiküldéséhez.

[25] Egy ilyen FIFO-modul RTL megvalósítását részletesen tárgyaljuk a 4.2.5. pontban.

[26] Ebben a kontextusban az UART-vevő mint kliens igénybe veszi a FIFO-nak mint kiszolgálónak az adatkeretek eltárolására vonatkozó szolgáltatását.



2-6. ábra. *UART adómodul illesztése FIFO-tárolóval*

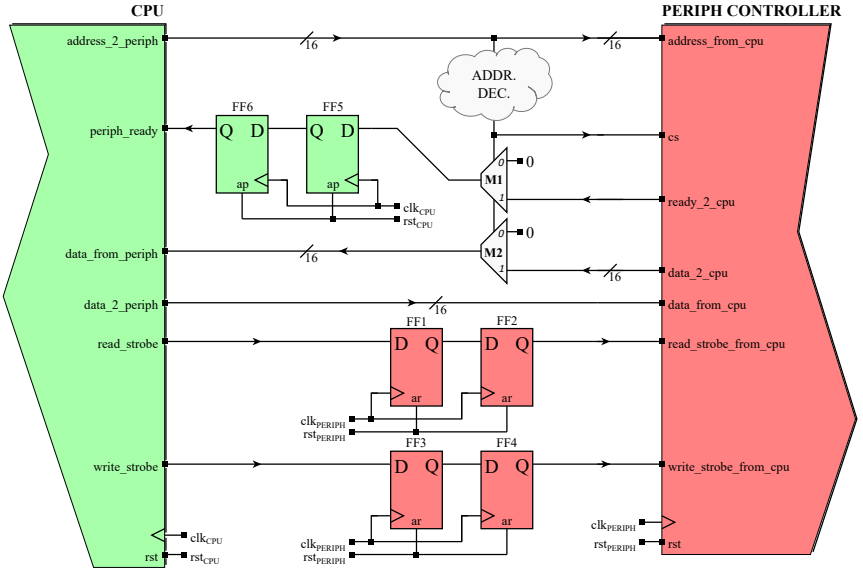
A dual-clock FIFO-k használatát lehetővé tevő köztes logikáknak a pontos szerkezete és az összetettsége is erősen függ a konkrét problémától, így érdemes azokat az újrafelhasználható IP-magokkal együtt tervezni, és a toplevel integrátor rendelkezésére bocsátani.[27]

CDC az MMIO-interfészen ◆ Az MMIO-interfész 1.2.1.2. pontban ismertetett specifikációja első ránézésre nagyon hasonlónak tűnhet a szinkron kézfogásos interfészéhez, azonban a két megoldás közötti apró eltérések okán az órajel-szinkronizálás itt egyszerűbben is megoldható. A két kulcsfontosságú különbség a következő:

- A *CPU* számára nem megengedett, hogy a periféria-áramkörökhöz intézett kérést a *read/write_strobe* jeleken előállított egyszerű, egy órajel-periódus ideig fennálló pozitív impulzussal jelezze. A *strobe* jel magas szintre állítása után annak visszavételével türelmesen ki kell várnia a megcímzett periféria-áramkör által meghajtott *ready* jel lefutását (lásd 1-8. és 1-9. ábra: 4. lépés).
- A megcímzett periféria-áramkör nem köteles a *ready* jelzést „azonnal” – rögtön az órajele következő felfutóélénél – alacsony szintre állítani. A *read/write_strobe* jel felfutóéle és a *ready* jel lefutóéle között eltelt idő tetszőleges lehet.

[27] Figyeljük meg, hogy a dual-clock FIFO és az azzal együtt járó illesztőlogika beépítésével tulajdonképpen rögtön két újrafelhasználható IP-magot adunk ki a kezünkből; egyet az eredeti kliensinterfésszel, egyet pedig FIFO-*push* jellegű kliensinterfésszel. A felhasználó maga dönthet, hogy adott szituációban melyiket szeretné használni, az újrafelhasználható IP-magunk lehetséges alkalmazási köre tehát bővül. Az UART-adó és -vevő áramkörök FIFO-alapú órajel-szinkronizáló burkolóit részletesen is tárgyaljuk a 4.3.1.1. pontban.

E két eltérés következtében a *read/write_strobe* és a *ready* jelek egyaránt statikus jelzések, vagyis, ha a két fél két külön órajeltartományban működik, e jelek átvitelére használhatunk egyszerű double-flop szinkronizálókat a 2-7. ábrán látható séma szerint.



2-7. ábra. *Órajel-szinkronizáló fokozatok az MMIO-interfészen*

Egy tranzakció kezdeményezésekor a *CPU* a címbuszra *(address_2_periph)* helyezi a megszólítandó periféria-áramkör kiválasztott regiszterének címét, illetve – írásművelet esetén – az adatbuszra *(data_2_periph)* az abba beírandó adatot, majd magas szintre állítja a *read_strobe* vagy a *write_strobe* jelek egyikét attól függően, hogy olvasást vagy írást akar végrehajtani. A *read_strobe* és *write_strobe* jeleket rendre az *FF1* és *FF2,* valamint az *FF3* és *FF4* flip-flopokból álló double-flop szinkronizáló fokozatok továbbítják a periféria-áramkör órajeltartományába. Mivel a *strobe* jeleket a *CPU* mindaddig nem állítja vissza alacsony szintre, amíg a periféria-áramkör *ready* jelének lefutóélét nem érzékeli, ezért a double-flop szinkronizálók mellett egyéb elemekre nincs szükség. A periféria-áramkörök saját címdekóderei kombinációs hálózatok, vagyis az órajeltől függetlenül „azonnal”[28] megállapítják, hogy a címbuszon olvasható új érték a saját címtartományukba tartozik-e. Ha nem, akkor az *M1* és *M2* multiplexerek gondoskodnak róla, hogy a *CPU* felé mutatott kimenetek – az elosztott címdekódolási séma (lásd 1-7. ábra) által támasztott követelménynek megfelelően – alacsony szintre legyenek húzva. Minden tranzakció esetén pontosan egy periféria-áramkör lokális címdekódere magas szintre állítja az adott periféria-áramkör kiválasztó *(chip select, cs)* jelét, átadva a *CPU* bemeneteit meghajtó jelek vezérlését a belső logikának. Ez a belső logika az MMIO-interfész specifikációjának megfelelően előbb-utóbb alacsony szintre

[28] Természetesen a kombinációs hálózatnak is van késleltetése, amiről az RTL-tervezésnek ezen a pontján nyugodtan feltételezhetjük, hogy a *CPU* órajel-periódusánál kisebb. Erről a tervezési folyamat egy későbbi pontján, a fizikai szintézis során az elhelyező/huzalozó algoritmusok igyekeznek majd gondoskodni. Ha az adott FPGA technológián, az előírt órajel-frekvenciák esetén ez a feltétel nem teljesül, arra a statikus időzítésvizsgálatkor derülhet fény.

állítja a *ready_2_cpu* jelet, amit ugyancsak egy egyszerű double-flop szinkronizálóval *(FF5, FF6)* csatolunk vissza a *CPU* órajeltartományába. E lefutóél hatására a *CPU* alacsony szintre állítja a *read/write_strobe* jelet. Ha a periféria-áramkör elvégezte a feladatát, illetve a *read/write_strobe* jel lefutását is megvárta, akkor a *ready_2_cpu* jelet ismét magas szintre húzza, amivel az MMIO-tranzakció befejezettnek tekinthető. A 2-8. ábra a *CPU* és egy periféria-áramkör közötti interfész Verilog-leírását mutatja be.

Verilog – a *CPU* órajeltartománya

```
always @ ( posedge clk_cpu,
           posedge rst_cpu )
begin
  if ( rst_cpu == 1 ) begin
    FF5 <= 1; FF6 <= 1;
  end else begin
    FF5 <= M1; FF6 <= FF5;
  end
end

assign periph_ready = FF6;
assign data_from_periph = M2;
```

Verilog – a periféria-áramkör órajeltartománya

```
always @ ( posedge clk_periph, posedge rst_periph )
begin
  if ( rst_periph == 1 ) begin
    FF1 <= 0; FF2 <= 0;
    FF3 <= 0; FF4 <= 0;
  end else begin
    FF1 <= read_strobe; FF2 <= FF1;
    FF3 <= write_strobe; FF4 <= FF3;
  end
end

assign data_from_cpu = data_2_periph;
assign read_strobe_from_cpu = FF2;
assign write_strobe_from_cpu = FF4;

assign cs = ( ( address_2_periph == << address_1 >> ) ||
              ( address_2_periph == << address_2 >> ) ||
              ( address_2_periph == << address_3 >> ) ||
              ( [...] )
            ) ? 1 : 0;

assign M1 = ( cs == 1 ) ? ready_2_cpu : 0;
assign M2 = ( cs == 1 ) ? data_2_cpu : 16'h0;
```

2-8. ábra. *CDC a CPU és a periféria-áramkörök között*

2.2. A CPU részletes tervezése

Áttanulmányozásra javasolt alfejezetek:

4.2. *On-chip adattárolás*

Funkcióbővítő rendszerünk központi feldolgozó egysége egy tárolt programot végrehajtó mikroáramkör, amelynek feladatköre kellően változatos és összetett ahhoz, hogy a részletes RTL-tervezést egy viselkedési modell elkészítésével alapozzuk meg. Az egyértelműség kedvéért az elvárt viselkedés megfogalmazására egy „majdnem formális” leírási módszert, pszeudokódot használunk.

2.2.1. Viselkedési modell

A viselkedési modell létrehozásakor a *CPU* utasításkészletének definíciójából indulunk ki (2-1. táblázat, részletesen lásd 5.1.1.1. pont).

Utasítás	Utasítás funkciója	Módosított állapotjelzők
i2rfl	RF[IR[11:8]] [15:8] = **0** RF[IR[11:8]] [7:0] = IR[7:0]	Z, N
load	RF[IR[11:8]] = PERIPH[RF[IR[7:4]] + RF[IR[3:0]]]	Z, N
store	PERIPH[RF[IR[7:4]] + RF[IR[3:0]]] = RF[IR[11:8]]	-
add	RF[IR[11:8]] = (RF[IR[7:4]] + RF[IR[3:0]]) % 2^{16}	Z, N
sub	RF[IR[11:8]] = (RF[IR[7:4]] - RF[IR[3:0]]) % 2^{16}	Z, N
and	RF[IR[11:8]] = RF[IR[7:4]] **AND** RF[IR[3:0]]	Z, N
or	RF[IR[11:8]] = RF[IR[7:4]] **OR** RF[IR[3:0]]	Z, N
xor	RF[IR[11:8]] = RF[IR[7:4]] **XOR** RF[IR[3:0]]	Z, N
shl	RF[IR[11:8]] = RF[IR[7:4]] **SHL** unsigned(RF[IR[3:0]])	Z, N
shr	RF[IR[11:8]] = RF[IR[7:4]] **SHR** unsigned(RF[IR[3:0]])	Z, N
bnz	if (Z = **0**) then IP = IP + signed(IR[11:0]) else IP = IP+1	-
bnn	if (N = **0**) then IP = IP + signed(IR[11:0]) else IP = IP+1	-
jmp	IP = IP + signed(IR[11:0])	-
call	STACK[SP] = EI, Z, N, IP+1 SP = SP+1 EI = **0** IP = IP + signed(IR[11:0])	EI
ret	SP = SP-1 EI, Z, N, IP = STACK[SP]	EI, Z, N
ei	EI = IR[0]	EI

2-1. táblázat. *A CPU utasításkészletének összefoglalása*

A gépi utasítások definícióját már adottnak tekintjük, de érdemes megjegyezni, hogy a mikroprocesszorok tervezésének egy igen fontos része ennek az utasításkészlet-definíciónak a megalkotása is. Ennek a specifikációnak az apró részletei komoly hatással lehetnek a leendő mikroarchitektúra szerkezetére, ezáltal annak számítási teljesítményére és erőforrás-felhasználására is. A csekély erőforrásigény esetünkben különösen fontos szerepet játszik. Az erre való törekvés az utasításkészlet-definícióban is tetten érhető, nemcsak abban, hogy a hardverben megvalósított gépi utasítások száma minimális. Ha figyelmesen átolvassuk az egyes gépi utasítások leírását, azt látjuk, hogy bár nyilván minden utasításnak más a feladata, de sok bennük a hasonlóság is (regularitás). Az aritmetikai/logikai utasítások *(add, sub, and, or, xor, shl, shr)* esetén például azt figyelhetjük meg, hogy ezek operandusait mindig az általános célú regisztertömbből *(**R**egister **F**ile, RF)* olvassuk ki, ráadásul mindig az utasításregiszternek ugyanazon helyiértékű bitjei által címzett rekeszekből. Nem árulunk el nagy titkot azzal, hogy ezeknek az utasításoknak a magját egytől-egyig egy közös aritmetikai/logikai egység *(**A**rithmetic/**L**ogic **U**nit, ALU)* hajtja majd végre. Ezt figyelembe véve akár úgy is fogalmazhatunk, hogy ez a hét utasítás lényegében ugyanaz, végrehajtásuk folyamata pusztán abban különbözik majd, hogy a megfelelő órajelciklusban az *ALU* funkciókiválasztó bemenetére milyen bitkombináció kerül. Azt is észrevehetjük, hogy az utasítások eredménye mindig az utasításregiszter egyazon bitjei által címzett regisztertömb-rekeszbe kerül. Az utasítások végrehajtásának szervezését végző részáramkörök szempontjából minden ilyen egyszerűsítés, általánosítás egyúttal erőforrás-spórolást is jelent.

Ha a gépi utasítások egymástól nagyon eltérnének, akkor a *CPU* viselkedésének megfogalmazása egy sokfelé ágazó feltételes szerkezettel *(case)* lenne kifejezhető, amelynek feltétel mezőjében az a kérdés szerepel, hogy melyik utasítást hajtjuk éppen végre. A reguláris, vagyis hasonló gépi utasítások azonban lehetővé teszik azt, hogy a végrehajtást olyan fázisokra bontsuk, amelyek a különböző gépi utasítások esetén nagyrészt megegyeznek. A funkcióbővítő rendszer *CPU*-ja esetén ezek a fázisok végrehajtásuk sorrendjében a következők lesznek:

Utasításfelhozás (fetch) ◆ Az utasításfelhozás fázis alapfeladata, hogy a programtárnak az utasításmutató *(IP)* által címzett rekeszét beolvassa és betöltse az utasításregiszterbe *(IR)*. A 2-9. ábra kódrészletében azonban látható, hogy a *CPU* műveletvégzését befolyásoló külső események miatt a helyzet kicsit bonyolultabb.

Pseudo – viselkedési modell

```
1  FETCH: request IMEM read
2         wait until IMEM data available
3         if ( firmware reset asserted ) {
4           IP <= reset vector
5           goto FETCH
6         } elsif ( interrupt request detected and EI = 1 ) {
7           clear interrupt request flag
8           STACK[SP] <= (EI, Z, N, IP)
9           SP++
10          IP <= interrupt vector
11          EI <= 0
12          goto FETCH
13        } else IR <= data from IMEM
```

2-9. ábra. *Az utasításfelhozás fázis folyamata*

Az egyik ilyen külső esemény a „firmware reset” (2-9. ábra: 3-5. sor), amely során a program futtatását újraindíthatjuk anélkül, hogy a *CPU* és az azt körülvevő periféria-áramkörök *reset* bemenetét meghúznánk.[29] Ezt a funkciót a *BOOTSTRAP* áramkör (lásd 1.2.3.1. pont) használja majd, ami a programtár inicializálása során a *firmware reset* bemenet magas szinten tartásával gondoskodik arról, hogy a végrehajtandó program maradéktalan betöltése előtt a végrehajtás semmiképp se induljon el.

A másik kivételes esemény a megszakításkérés, amikor a programvégrehajtást egy speciális címen, a központi megszakításkiszolgáló rutin kezdőcímén kell folytatni oly módon, hogy a megszakítási rutin végrehajtása után a *CPU* képes legyen a megszakítás beérkezése előtti állapotából folytatni a műveletvégzést (lásd 1.2.3.2. pont). Ilyenkor a *CPU* – miután nyugtázta a megszakításkérést (2-9. ábra: 7. sor) és a rendszer aktuális állapotát leíró regisztereket azok visszaállíthatósága céljából a belső állapotveremtárba elmentette (2-9. ábra: 8-9. sor)[30] – betölti az utasításmutatóba a központi megszakításkiszolgáló rutin programtárbeli címét (2-9. ábra: 10. sor), tiltja az összes megszakítást (2-9. ábra: 11. sor), majd ismételt olvasást hajt végre a programtáron.

Dekódolás ◆ A gépi utasítások többségének magjában az *ALU* végez el valamilyen műveletet, amelynek operandusai az általános célú regisztertömbben vannak eltárolva. A dekódolás fázis fő feladata, hogy kiolvassa ezeket az operandusokat a regisztertömbből, és eltárolja őket az *ALU* bemeneteit közvetlenül meghajtó *X* és *Y* regiszterekben.

[29] Az alacsony szintű reset jeleket amennyire lehet „érintetlenül hagyjuk” abban az értelemben, hogy csak kifejezetten a reset áramkörök kimeneteit használjuk azok meghajtására. A „firmware reset” egy logikailag magasabb rendű újraindítási funkció, amely csak a *CPU*-n futó programot indítja újra az utasításmutató alapállapotba állításával.

[30] A viselkedési modellhez használt pszeudokódban az *(...)* operátor összefűzést jelent.

Pseudo – viselkedési modell

```
1  DECODE: X <= RF[IR[7:4]]
2          if ( IR[15:12] = i2rfl ) Y <= unsigned(IR[7:0])
3          else Y <= RF[IR[3:0]]
```

2-10. ábra. *A dekódolás fázis folyamata*

A 2-10. ábrán jól látható, hogyan egyszerűsíti le a rendszer viselkedését, ezáltal az azt megvalósító áramkörök szerkezetét az utasítások regularitása. Az *X* operandust minden utasítás esetén az általános célú regisztertömbnek az utasításregiszter [7:4] bitjei által címzett rekeszéből olvassuk ki (2-10. ábra: 1. sor). Az *Y* operandusnál a helyzet bonyolultabb, mivel az *i2rfl* utasítás esetén ez nem a regisztertömbben van eltárolva, hanem magába az utasításba van belekódolva.[31] Itt tehát nem spórolható meg az aktuális gépi utasítás kódja alapján végzett feltételvizsgálat (2-10. ábra: 2-3. sor), ami áramköri szinten többlet dekódereket és multiplexálást igényel majd. A gépi utasítások egyformasága az ehhez hasonló többlet erőforrásokat minimalizálja.

Perifériaműveletek ◆ A *CPU* programjának írója a periféria-áramkörökkel – egyúttal az adatmemóriával – mindössze két gépi utasítás, a *store* és a *load* segítségével teremthet kapcsolatot. A perifériaműveletek fázis első lépéseként megvizsgájuk, hogy e két utasítás valamelyikét hajtjuk-e éppen végre, és ha nem, akkor az egész fázist átugorjuk (2-11. ábra: 1. sor).

Pseudo – viselkedési modell

```
1  PERIPH_ACCESS: if ( IR[15:12] != store and IR[15:12] != load ) goto STACK
2                 PDR <= RF[IR[11:8]]
3                 PAR <= X + Y
4                 if ( IR[15:12] = store ) request MMIO write
5                 if ( IR[15:12] = load ) request MMIO read
6                 wait until MMIO transaction is done
7                 if ( IR[15:12] = load ) Y <= data from peripheral
```

2-11. ábra. *A periféria-áramkörökkel való kölcsönhatás fázisának folyamata*

A periféria-áramköröket MMIO-tranzakciókkal érhetjük el, amelyek előkészítéseként az általános célú regisztertömbből kiolvassuk a periféria-áramkör felé kiküldendő adatot, majd betöltjük azt a *PDR (Peripheral Data Register)* regiszterbe (2-11. ábra: 2. sor). A tranzakcióhoz tartozó címet a *store* és *load* utasítások specifikációjának megfelelően a dekódolás fázisban már előkészített *X* és *Y* regiszterek értékének összegeként képezzük, és betöltjük a *PAR (Peripheral Address Register)* regiszterbe (2-11. ábra: 3. sor).[32] Ezután kezdeményezzük az MMIO-tranzakciót (2-11. ábra: 4-5. sor), majd, ha *load* utasítást hajtunk végre, vagyis egy periféria-áramkörtől származó adatot olvasunk be, akkor a tranzakció befejeződése után betöltjük azt az *Y* regiszterbe (2-11. ábra: 7. sor).

Állapotverem-műveletek ◆ Ebben a fázisban a *call* és a *ret* utasítások több lépést igénylő végrehajtásának előkészítése zajlik le. A *call* utasítás végrehajtásakor letiltjuk a megszakításokat (2-12. ábra: 2. sor). Ez a megoldás tulajdonképpen egy, a programozó dolgát megkönnyítő kényelmi funkció.

[31] A 2. sor jobb oldalán az utasításmutató alsó bájtjának előjel nélküli egészként való értelmezése gyakorlatilag azt jelenti, hogy az *Y* regiszter felső bájtját nullázzuk.

[32] Figyeljük meg a regularitásra való törekvés hatását ebben az esetben is. Az utasítások operandusainak szerepe a *store* és a *load* utasítások esetén egészen más, mint az aritmetikai/logikai utasítások esetén, de az *X* és *Y* regiszterek értéke mindkét utasítástípusnál az általános célú regisztertömbnek ugyanazokból a rekeszeiből származik annak érdekében, hogy a dekódolás fázisnak ne kelljen különbséget tennie a kétféle utasítástípus között.

Sok megszakítási forrást tartalmazó rendszerekben a processzoron futó program írása során nagy könnyebbséget jelenthet az események nyomon követésében, ha biztosak lehetünk benne, hogy a megszakítások csak a főprogramba való visszatérés után juthatnak érvényre.

Pseudo – viselkedési modell

```
1  STACK: if ( IR[15:12] = call ) {
2           EI <= 0
3           STACK[SP] <= (EI, Z, N, IP+1)
4           SP++
5         }
6         if ( IR[15:12] = ret ) SP--
```

2-12. ábra. *Az állapotverem-műveletek fázisának folyamata*

A szubrutinhívást végző *call* utasítás végrehajtásának következő lépése a *CPU* aktuális állapotának, vagyis az utasításmutatónak és az *EI, Z* és *N* állapotjelzőknek (flag)[33] az elmentése a belső állapotveremtárba (2-12. ábra: 3. sor). Ez a lépés csaknem teljesen azonos a megszakítások érvényre jutásakor végrehajtott állapotmentéssel (lásd 2-9. ábra: 8. sor). A különbség csupán az, hogy míg megszakításkor arra a programtár-címre kell visszatérnünk, ami a megszakított utasítást tartalmazza, tehát az állapotveremtárba magát az utasításmutatót töltjük, addig szubrutinhívás esetén az állapot elmentésének pillanatában az utasításmutató magának a *call* utasításnak a címét tartalmazza, az állapotveremtárba tehát annak inkrementált értékét kell tölteni. Így a *ret* utasítás végrehajtásakor a *call* utasítást közvetlenül követő utasításra ugrunk vissza. Az állapotveremtárat egy dedikált regiszterrel (***S**tack **P**ointer, SP*) címezzük. A verembe íráskor e mutató értékét eggyel növeljük (2-12. ábra: 4. sor). A szubrutinból vagy megszakításkiszolgáló rutinból való visszatéréskor, vagyis a *ret* utasítás végrehajtásakor ebben a fázisban csupán az *SP* regiszter értékének csökkentése történik meg (2-12. ábra: 6. sor), magának a veremtárnak az olvasása már a visszaírás fázis feladata lesz.

Visszaírás ◆ A visszaírás fázis neve arra utal, hogy fő feladata az aritmetikai utasítások eredményének az általános célú regisztertömbbe való visszaírása. Első lépésként az *ALU* funkcióválasztó bemenetén elhelyezzük az utasítás kódját (2-13. ábra: 1. sor), ez alapján dől majd el az *ALU*-n belül, hogy a két operandussal milyen műveletet kell végezni. Ezután az *ALU* bemeneteire kapcsoljuk magukat az operandusokat is (2-13. ábra: 2–3. sor).

Pseudo – viselkedési modell

```
1  WRITE_BACK: ALU.op <= IR[15:12]
2              ALU.x <= X
3              ALU.y <= Y
4
5              if ( IR[15:12] = i2rfl, load, add, sub, and, or, xor, shl, shr )
6                (RF[IR[11:8]], Z, N) <= results from ALU
7
8              if ((IR[15:12] = bnz and !Z) or
9                  (IR[15:12] = bnn and !N) or
10                 (IR[15:12] = jmp) or
11                 (IR[15:12] = call)
12             ) IP <= IP + signed(IR[11:0])
13             else IP++
14
15             if ( IR[15:12] = ret ) (IP, EI, Z, N) <= STACK[SP]
16             if ( IR[15:12] = ei ) EI <= IR[0]
```

2-13. ábra. *A visszaírás fázis folyamata*

[33] A „flag” kifejezés egybites állapotjelző regisztert jelent, ami az áramkörben egy-egy egyedi feltételrendszer bekövetkezését jelzi.

Az 5–6. sorokban az aritmetikai/logikai utasítások eredményét az általános célú regisztertömbbe írjuk vissza, illetve frissítjük az állapotjelzőket is, amelyeknek az új értékét ugyancsak az *ALU* állítja elő. Figyeljük meg, hogy az 5. sorban az *ALU*-t érintő utasítások listájában van két kakukktojás; az *i2rfl* és a *load* utasítások esetén is az *ALU*-tól kapjuk meg a regisztertömbbe visszaírandó eredményt, annak ellenére, hogy ezek az utasítások nem igényelnek sem aritmetikai, sem logikai műveletet. Ezeknél az *ALU* egyszerűen csak „átengedi magán" az *Y* operandust úgy, hogy közben megvizsgálja annak értékét és megfelelően beállítja a *Z* és *N* állapotjelzőket. Ha visszatekintünk a dekódolás fázis 2. és a perifériaműveleteket leíró fázis 7. sorára, láthatjuk, hogy valóban, ez a manőver már ott előkészíttetett azáltal, hogy az *Y* regiszterbe betöltöttük a megfelelő értékeket, így a visszaírás fázisban az *ALU y* bemenetére már minden esetben rákapcsolhatjuk az *Y* regisztert. Ezzel a megoldással azt értük el, hogy az utasítások különbözőségéből adódó, elkerülhetetlen multiplexálási funkciót az *Y* regiszter bemenetére korlátoztuk. Ez azt jelenti, hogy sem az *ALU y* bemenetére, sem a regisztertömb adatbemenetére nem kell majd multiplexert kapcsolnunk, ami áramköri szempontból kedvezőbb időzítési tulajdonságokhoz, közvetetten magasabb maximális órajel-frekvenciához vezethet. Ennek akkor lehet különösen nagy jelentősége, ha magát az *ALU*-t tisztán kombinációs hálózatként valósítjuk meg.[34]

A visszaírás fázis következő lépése az utasításmutató értékének beállítása úgy, hogy az a következő végrehajtandó utasításra mutasson (2-13. ábra: 8–13. sor). Alapesetben ez csak egy inkrementálást jelent, az elágazást megvalósító utasítások *(bnz, bnn, jmp, call)* esetén viszont az utasításmutató aktuális értékéhez egy, az utasításban megadott konstanst adunk hozzá.[35] Ezt a konstanst előjeles egészként értelmezzük, vagyis a programban előre és hátra is ugorhatunk. Az ugrás mértékét az utasításregiszter e célra szánt bitjeinek száma korlátozza. Esetünkben ez 12, így az ugrás abszolútértéke pozitív irányba 2047-nél, negatív irányba 2048-nál nem lehet nagyobb.

A visszaírás fázisban történik a *ret* utasítás végrehajtásának befejezése, vagyis a legutóbb az állapotveremtárba mentett utasításmutató és az állapotjelzők visszatöltése is (2-13. ábra: 15. sor), továbbá a megszakítások engedélyezettségét tároló állapotjelző *(EI)* beállítása az erre vonatkozó utasításban megadott értékre (2-13. ábra: 16. sor).

A program futtatása tulajdonképpen nem más, mint a 2-9–2-13. ábrákon látható fázisok periodikus ismétlése (lásd 2-14. ábra).

Pseudo – viselkedési modell

```
1 repeat {
2 | FETCH
3 | DECODE
4 | PERIPH_ACCESS
5 | STACK
6 | WRITE_BACK
7 }
```

2-14. ábra. *A CPU az egyes végrehajtási fázisok periodikus ismétlésével futtatja a programot*

[34] Felmerülhet a kérdés, hogy az ehhez hasonló, meglehetősen körmönfont okfejtések valóban a viselkedési modellezés tárgykörébe tartoznak-e. Kétségtelen, hogy az időzítési tulajdonságok optimalizálása a tervezésnek ezen a pontján kissé korai. Az összetett rendszerek tervezése iteratív folyamat, ahol az ilyen finomhangolások jellemzően már egy korábbi verzió részletes terveinek ismeretében valósulnak meg. Egy adatút optimalizálásának lehetősége például tipikusan sokkal könnyebben észrevehető egy már meglévő RTL kapcsolási rajzon, mint egy viselkedési modellen. Ugyanakkor – ahogy azt az olvasó is megtapasztalja majd – sok-sok gyakorlással, valós problémák megoldása során szerzett tapasztalattal egyre világosabban láthatjuk a magas szinten meghozott döntéseink áramköri szintű következményeit.

[35] Ez az ún. IP-relatív címzés, ami lehetővé teszi, hogy az egymásra hivatkozó, „egymást hívó" szubrutinokat a programban módosítás nélkül áthelyezhessük.

A *CPU* RTL-modelljét a „vezérlőegység + műveletvégző egység” séma szerinti bontásban valósítjuk meg (lásd még DRMSz: 4.8. alfejezet), ahol a vezérlőegység szerepét egy állapotgép tölti be, a műveletvégző egységet pedig az egyes gépi utasítások végrehajtásához szükséges aritmetikai, logikai és tároló elemek együttese alkotja. Mindkét részegység részletes tervezéséhez a viselkedési modellt vesszük alapul.

2.2.2. Műveletvégző egység

A *CPU* műveletvégző egységét a viselkedési modellből szisztematikus lépések sorozatával építjük fel. A műveletvégző egység tulajdonképpen a gépi utasítások összességének végrehajtásához szükséges alkatrészek megfelelően összekapcsolt halmaza. Létre kell tehát hoznunk egy „RTL-netlistát”[36], az erőforrások közötti kapcsolatoknak egy olyan listáját, ami lehetővé teszi a viselkedési modellben megfogalmazott, regiszterek közötti adatátviteli tranzakciók (register transfer) megvalósulását.

1. lépés – RTL-netlista létrehozása a viselkedési modellből ◆ A 2-15. ábra az utasításfelhozás fázis viselkedési modelljét és az abból következő RTL-netlistát mutatja be.

Pseudo – viselkedési modell

```
1  FETCH: request IMEM read
2         wait until IMEM data available
3         if ( firmware reset asserted ) {
4           IP <= reset vector
5           goto FETCH
6         } elsif ( interrupt request detected and EI = 1 ) {
7             clear interrupt request flag
8             STACK[SP] <= (EI, Z, N, IP)
9             SP++
10            IP <= interrupt vector
11            EI <= 0
12            goto FETCH
13        } else IR <= data from IMEM
```

RTL-netlista

```
1  IMEM.raddr <= IP
2  IR <= IMEM.dout
3  IP <= reset vector
4  IP <= interrupt vector
5  STACK.waddr <= SP
6  STACK.din[18] <= EI
7  STACK.din[17] <= Z
8  STACK.din[16] <= N
9  STACK.din[15:0] <= IP
10 SP_add.opa <= SP
11 SP_add.opb <= 1
12 SP <= SP_add.result
13 EI <= 0
```

2-15. ábra. *Az utasításfelhozás fázis és az annak végrehajtását lehetővé tévő RTL-netlista*

Az utasításfelhozás első lépése, hogy kiolvassuk az utasításmemória *(IMEM)* utasításmutató által címzett rekeszének tartalmát. Az utasításmemóriát a 4.2.3. pontban részletesen bemutatott, *singleclock_dualport_ram_hswrp* nevű IP-maggal valósítjuk meg. E memóriamodul mindkét interfésze az 1.2.1.1. pontban tárgyalt, szinkron kézfogásos interfész-specifikáció szerint működik. A műveletvégző egység konstrukciója szempontjából az interfész használatának pontos koreográfiája nem túl izgalmas, hiszen most csak azt szeretnénk meghatározni, hogy az egyes alkatrészek adatáramlás szempontjából lényeges ki/bemeneti portjait miként kapcsoljuk össze. Az utasításmemóriából való olvasás esetén ez azt jelenti, hogy a kiolvasó interfész címbuszára *(raddr)* rákapcsoljuk az utasításmutatót, majd a kimeneti adatbusz értékét *(dout)* betöltjük az utasításregiszterbe. Így tehát az RTL-netlista 1. és 2. sorai rögzítik a viselkedési modell 1. és 13. soraiban definiált működéshez szükséges adatkapcsolatokat.

[36] A „netlista” kifejezést – elsősorban az áramkörszimulációs szoftverek terminológiájában – áramköri szintű kapcsolások szöveges reprezentációira használjuk. A *CPU* regiszter-transzfer szintű részegységeinek hálózatát specifikáló szöveges formátumú leírást ennek mintájára nevezzük „RTL-netlistának”.

A viselkedési modell 3–5. soraiban definiált műveletekkel egyetlen dolgunk van; az utasításmutatóba betölthetővé kell tennünk a *CPU*-n futó program legelső utasításának programtárbeli címét *(reset vector)*. A viselkedési modell 4. sorának értékadása tehát változatlan formában átkerülhet az RTL-netlistába is (3. sor).

Az érvényre jutó megszakítások kiszolgálásának a viselkedési modell 6–12. soraiban leírt előkészítése több adatút létrehozását is szükségessé teszi. A 8. sor értékadása azt fejezi ki, hogy a belső állapotveremtár *SP* regiszter által címzett rekeszébe be kell írni az állapotjelzők *(EI, Z, N)*, illetve az utasításmutató aktuális értékét. Az állapotveremtárat a 4.2.1. pontban bemutatott *singleclock_dualport_ram* modullal valósítjuk meg, ami adat- és címbuszok szempontjából az utasításmemóriával egyenértékű.[37] Az *SP* regisztert tehát az állapotveremtár beírásért felelős interfészének címbuszára (RTL-netlista: 5. sor), az állapotjelzők és az utasításmutató értékéből összefűzéssel képezett vektort pedig annak adatbuszára kell kapcsolnunk (RTL-netlista: 6. sor). A viselkedési modell leírja, hogy az állapot elmentése után az állapotveremtárat címző *SP* regisztert inkrementálni kell, amihez egy dedikált összeadó áramkört *(SP_add)* használunk. *SP* értékének inkrementálásához az *SP_add* operátor egyik bemenetére magát az *SP* regisztert, másik bemenetére konstans 1-et kapcsolunk, kimenetét pedig az *SP* regiszter bemenetére kötjük (RTL-netlista: 7–9. sor). A megszakítások feldolgozása ezzel még nem ért véget, hiszen az utasításmutatóba be kell töltenünk a központi megszakításkiszolgáló rutin programtárbeli kezdőcímét (RTL-netlista: 4. sor), illetve az *EI* regiszter nullába állításával tiltanunk kell a további megszakítások érvényre jutását (RTL-netlista: 10. sor). Így lesz teljes az utasításfelhozás fázis viselkedési modelljében definiált műveletsor végrehajtásához szükséges adatkapcsolatok listája.

Pseudo – viselkedési modell

```
1  DECODE: X <= RF[IR[7:4]]
2          if ( IR[15:12] = i2rfl ) Y <= unsigned(IR[7:0])
3          else Y <= RF[IR[3:0]]
```

RTL-netlista

```
1  RF.raddr_a <= IR[7:4]
2  RF.raddr_b <= IR[3:0]
3  X <= RF.dout_a
4  Y <= unsigned(IR[7:0])
5  Y <= RF.dout_b
```

2-16. ábra. *A dekódolás fázis és az annak végrehajtását lehetővé tévő RTL-netlista*

[37] Az állapotveremtár (és az általános célú regisztertömb) esetén az utasításmemóriával ellentétben nem használunk kézfogásos interfészt. A műveletvégzés sebessége szempontjából előnyös volna az utasításmemóriát is kézfogás nélküli interfészekkel ellátni, hiszen a programfuttatás során minden egyes utasítás felhozásakor értékes órajelciklusokat nyerhetnénk ezáltal. Ugyanakkor ezzel korlátoznánk is magunkat annyiban, hogy így az utasításmemóriát az alkalmazási körülményektől függetlenül, mindig csak az FPGA belső blokk-RAM-jában vagy közvetlenül flip-flopokból tudnánk kialakítani, hiszen másképpen nem lehetne megoldani az 1-3. ábrán szemléltetett, kézfogás nélküli interfészre jellemző, egyetlen órajelcikluson belüli válaszidőt. A funkcióbővítő rendszerünk esetén valóban belső blokk-RAM-ban hozzuk majd létre az utasításmemóriát, de magát a *CPU* modellt úgy készítjük el, hogy ez a memória könnyen lecserélhető legyen egy olyanra, amelynek több órajelciklus ideig tart rendelkezésünkre bocsátani a megcímzett adatrekesz tartalmát. Ennek záloga pedig nem más, mint a kézfogásos interfész kialakítása. A könnyű lecserélhetőség alatt azt értjük, hogy az eleve kézfogásos interfésszel ellátott utasításmemóriára felkészített *CPU* mag (rendszertervben: *CPU core*) HDL-modelljét egyáltalán nem kell módosítanunk a cseréhez, hiszen az eredeti formájában is kompatibilis lesz az új, lassabb utasításmemóriával. A több órajelciklust igénylő programtár lehet például egy EDAC (**E**rror **D**etection **A**nd **C**orrection)-védelemmel ellátott modul, ami a hibakorrekciót végző belső logika miatt igényel több órajelciklust, de elképzelhető olyan modell is, amelyben a *CPU* mag melletti programtár valójában nem egy fizikai memória, hanem egy interfészadapter (vö. 1.2.3.1. pont: boot-memóriák illesztése), ami egy külső memóriamodult illeszt a rendszerhez. Ez utóbbi esetben a program futtatása „külső memóriából" történne, ami jócskán lelassítaná a műveletvégzést, de ha az alkalmazás megkívánja, RTL-modellünk a kézfogásos interfésszel ellátott utasításmemóriával akár ezt is lehetővé teszi. Ez a megoldás hasznos lehet például akkor, ha blokk-RAM nélküli programozható eszközben (pl. **C**omplex **P**rogrammable **L**ogic **D**evice, CPLD) kívánjuk megvalósítani a *CPU*-t. E gondolatmenetet vö. a fejezet elején említett alapelvekkel: újrafelhasználhatóság, modularitás, lokalitás, technológiafüggetlenség.

A 2-16. ábra a dekódolás fázis viselkedési modelljét és az annak végrehajtását lehetővé tévő adatkapcsolatok listáját mutatja. A *singleclock_tripleport_ram* IP-val (lásd 4.2.1. pont) megvalósított általános célú regisztertömbből a dekódolás során két különböző címről (*IR[7:4]* és *IR[3:0]*) olvasunk ki egy-egy bejegyzést, amihez az RTL-netlista 1. és 2. sorában látható módon az utasításregiszter megfelelő bitjeit a két független címbuszra *(raddr_a, raddr_b)* kapcsoljuk, majd a két kimeneti adatbuszt *(dout_a, dout_b)* rendre az *X* és *Y* regiszterekre kötjük. Az *i2rfl* utasítás esetén az *Y* regiszter nem a regisztertömb kimenetének, hanem az utasításregiszter alsó bájtjának értékét kapja.

A periféria-áramkörökbe való írás és onnan való olvasás lebonyolításáért felelős fázis több új erőforrást is igényel (2-17. ábra).

Pseudo – viselkedési modell

```
1  PERIPH_ACCESS:
2    if ( IR[15:12] != store and IR[15:12] != load ) goto STACK
3    PDR <= RF[IR[11:8]]
4    PAR <= X + Y
5    if ( IR[15:12] = store ) request MMIO write
6    if ( IR[15:12] = load ) request MMIO read
7    wait until MMIO transaction is done
8    if ( IR[15:12] = load ) Y <= data from peripheral
```

RTL-netlista

```
1  RF.raddr_a <= IR[11:8]
2  PDR <= RF.dout_a
3  PAR_add.opa <= X
4  PAR_add.opb <= Y
5  PAR <= PAR_add.result
6  address_2_periph <= PAR
7  data_2_periph <= PDR
8  Y <= data_from_periph
```

2-17. ábra. *A perifériaműveletek fázis és az annak végrehajtását lehetővé tévő RTL-netlista*

A *PDR* regiszterbe az *X* regiszterhez hasonlóan a regisztertömb *dout_a* kimenetét kötjük (RTL-netlista: 2. sor), de ezúttal az ehhez az adatkimenethez tartozó címbuszra az utasításregiszternek a [11:8] bitjeit kapcsoljuk. A *PAR* regiszter a viselkedési modell 4. sorában az *X* és *Y* regiszterek összegét kapja értékül. Ennek az összegnek a képzéséhez egy újabb dedikált aritmetikai elemet *(PAR_add)* használunk, amelynek két bemenetére az *X* és *Y* regisztereket kötjük (RTL-netlista: 3–4. sor), kimenetét pedig a *PAR* regiszterre kapcsoljuk (RTL-netlista: 5. sor). Az MMIO-tranzakció lebonyolításához a viselkedési modellben explicit módon nem szereplő adatkapcsolatokat is meg kell valósítanunk. Az RTL-netlista 6–7. sorában a *CPU address_2_periph* kimenetére a *PAR, data_2_periph* kimenetére pedig a *PDR* regisztert kapcsoljuk rá. Különbséget teszünk tehát az RTL-netlistában a modell belső regiszterei és portjai között. Hasonlóan, a viselkedési modell 8. sorában megfogalmazott adatátvitelt lehetővé tévő, az RTL-netlistának ugyancsak a 8. sorában látható adatkapcsolatban a *CPU*-nak a periféria-áramkörök által meghajtott bemenetére már egy portnévvel *(data_from_periph)* hivatkozunk, a port értékét pedig képesnek kell lennünk az *Y* regiszterbe betölteni.

A 2-18. ábrán látható, az állapotverem-műveletek fázishoz tartozó RTL-netlistában nem nevezünk meg az eddigiekhez képest új RTL-erőforrást. Ennek oka, hogy a *call* utasítás folyamata csaknem teljesen azonos az utasításfelhozás fázis által levezényelt megszakításkezeléssel.

Pseudo – viselkedési modell

```
1  STACK: if ( IR[15:12] = call ) {
2           | EI <= 0
3           | STACK[SP] <= (EI, Z, N, IP+1)
4           | SP++
5         }
6         if ( IR[15:12] = ret ) SP--
```

RTL-netlista

```
1  EI <= 0
2  STACK.waddr <= SP
3  IP_add.opa <= IP
4  IP_add.opb <= 1
5  STACK.din[18] <= EI
6  STACK.din[17] <= Z
7  STACK.din[16] <= N
8  STACK.din[15:0] <= IP_add.result
9  SP_add.opa <= SP
10 SP_add.opb <= 1
11 SP_add.opb <= -1
12 SP <= SP_add.result
```

2-18. ábra. *Az állapotverem-műveletek fázis és az annak végrehajtását lehetővé tévő RTL-netlista*

A különbség mindössze az, hogy az ezt a folyamatot kiváltó esemény ezúttal maga is egy gépi utasítás, nem pedig egy, az utasításoktól független külső esemény, ezért a visszatéréskor nem az éppen megszakított, de még végre nem hajtott utasítás címére, hanem az az utáni gépi utasításra kell visszatérni. Tehát, a viselkedési modell 3. sorában látható módon, az utasításmutató helyett annak eggyel megnövelt értékét kell az állapotveremtárba mentenünk az állapotjelzők mellett. Az utasításmutató megnövelt értékének előállításához az *IP_add* operátort használjuk, amelynek bemeneteire ehhez magát az utasításmutatót és konstans 1-et kell kötnünk (RTL-netlista: 3–4. sor), kimenetét pedig az állapotjelzőkkel összefűzve az állapotveremtár adatbemenetére kell kapcsolnunk (RTL-netlista: 8. sor).

Az állapotveremtárat most is az *SP* regiszterrel címezzük (RTL-netlista: 2. sor), amelynek értékét a *call* utasítás végrehajtásakor – a megszakítások érvényre jutásánál látottakhoz hasonlóan – eggyel növelnünk, a *ret* utasítás végrehajtásakor pedig eggyel csökkentenünk kell. Ezért a korábban is említett *SP_add* operátor *opb* bemenetére immár nemcsak az 1, hanem a -1 konstanst is rá kell kapcsolnunk (RTL-netlista: 10-11. sor).

Végül a visszaírás fázis viselkedését és az azt lehetővé tévő RTL-netlistát a 2-19. ábra mutatja be. A viselkedési modell 2–4. sorai azt fejezik ki, hogy az *ALU* funkciókiválasztó bemenetére az utasításregiszter felső bitjeit, vagyis a műveleti kódot kapcsoljuk, az *x* és *y* bemeneteire pedig rendre az *X* és *Y* regisztereket. Ezek az értékadások formailag is RTL-adatkapcsolatokat fejeznek ki, tehát közvetlenül másolhatóak az RTL-netlistába (1-3. sor).

Pseudo – viselkedési modell

```
1  WRITE_BACK:
2    ALU.op <= IR[15:12]
3    ALU.x <= X
4    ALU.y <= Y
5
6    if ( IR[15:12] = i2rfl, load, add, sub, and, or, xor, shl, shr )
7      (RF[IR[11:8]], Z, N) <= results from ALU
8
9    if ((IR[15:12] = bnz and !Z) or
10       (IR[15:12] = bnn and !N) or
11       (IR[15:12] = jmp) or
12       (IR[15:12] = call)
13     ) IP <= IP + signed(IR[11:0])
14   else IP++
15
16   if ( IR[15:12] = ret ) (IP, EI, Z, N) <= STACK[SP]
17   if ( IR[15:12] = ei ) EI <= IR[0]
```

RTL-netlista

```
1  ALU.op <= IR[15:12]
2  ALU.x <= X
3  ALU.y <= Y
4  RF.waddr <= IR[11:8]
5  RF.din <= ALU.result
6  Z <= ALU.z
7  N <= ALU.n
8  IP_add.opa <= IP
9  IP_add.opb <= signed(IR[11:0])
10 IP_add.opb <= 1
11 IP <= IP_add.result
12 STACK.raddr <= SP
13 IP <= STACK.dout[15:0]
14 EI <= STACK.dout[18]
15 Z <= STACK.dout[17]
16 N <= STACK.dout[16]
17 EI <= IR[0]
```

2-19. ábra. *A visszaírás fázis és az annak végrehajtását lehetővé tévő RTL-netlista*

A viselkedési modell 7. sorában az általános célú regisztertömbbe való visszaírás művelete látható, amihez a regisztertömb beíró interfészének címbuszára az utasításregiszter megfelelő bitjeit (RTL-netlista: 4. sor), adatbemenetére pedig az *ALU* kimenetét kapcsoljuk (RTL-netlista: 5. sor). Az *ALU*-műveletnek további termékei az *ALU z* és *n* kimenetein megjelenő értékek, ezeket rendre a *Z* és *N* regiszterekbe kötjük be (RTL-netlista: 6–7. sor).

A viselkedési modell 9–13. sorai azt a lépést fejezik ki, amikor egy feltételes utasítás ténylegesen ugrást valósít meg, vagyis az utasításmutató értékét az utasításregiszter alsó 12 bitjében tárolt eltolás szerint növelnünk vagy csökkentenünk kell. Az utasításmutató inkrementálására korábban már bevezetett *IP_add* operátor használható erre a célra, tehát annak *opb* bemenetére az utasításregiszter alsó 12 bitjének előjel-kiterjesztett értékét is rákapcsoljuk. Ha egy feltételes utasítás az állapotjelzők aktuális értéke miatt nem vált ki valódi ugrást, akkor az utasításmutatót inkrementálni

kell a viselkedési modell 14. sorának megfelelően. Mindezen műveletekhez az RTL-netlista 8–11. soraiban megfogalmazott adatkapcsolatokra lesz szükség.

A *ret* utasítás második lépését leíró értékadás (16. sor) tulajdonképpen az állapotveremtáron végrehajtott olvasás, amely során ezúttal is az *SP* regiszter adja a címet (RTL-netlista: 12. sor). Az RTL-netlista 13–16. sorai azt fejezik ki, hogy az állapotveremtár adatkimenetét jelentő logikai vektor egyes bitjeit „szétosztjuk" azok között a regiszterek között, amelyeknek értékét a beíráskor a verembe mentettünk. Végül a visszaírás fázishoz tartozó RTL-netlista utolsó eleme (17. sor) lehetővé teszi, hogy az *ei* utasítás végrehajtásakor az *EI* regiszter értékét az utasítás legalsó bitjében megadott értékre állítsuk be.

2. lépés – Multiplexerek beiktatása az RTL-netlistába ◆ A *CPU* műveletvégzési fázisainak elemzésével előállított RTL-netlista tehát azt fogalmazza meg, hogy mely komponensek (regiszterek, regisztertömbök, operátorok) mely portjai között kell összeköttetést teremtenünk ahhoz, hogy megfelelő vezérlés esetén lehetőség legyen a viselkedési modellben megfogalmazott műveletek végrehajtására. Az RTL-netlista ebben a formában azonban még nem alakítható át közvetlenül HDL-modellé. A megvalósítandó adatkapcsolatok listájában több olyan sort is találunk, amelyekben a balértékek egyazon regiszterre vagy portra hivatkoznak. A műveletvégző egység szisztematikus konstruálásának következő lépése ezeknek a többszörös hozzárendeléseknek a feloldása multiplexerek beiktatásával.

A 2-20. ábra az utasításmutatóra vonatkozó többszörös hozzárendelést feloldó, négybemenetű multiplexer beépítését szemlélteti.

RTL-netlista

```
1  IP <= IP_add.result
2  IP <= reset vector
3  IP <= interrupt vektor
4  IP <= STACK.dout[15:0]
```

RTL-netlista multiplexerekkel

```
1  mux_IP.din0 <= IP_add.result
2  mux_IP.din1 <= reset vector
3  mux_IP.din2 <= interrupt vector
4  mux_IP.din3 <= STACK.dout[15:0]
5  IP <= mux_IP.dout
```

2-20. ábra. *Multiplexer beiktatása az utasításmutató bemenetére*

Figyeljük meg, hogy a multiplexer beiktatásával nyert RTL-netlista jellegét tekintve hasonló, mint a multiplexereket nem tartalmazó változat, hiszen a multiplexer maga is a regiszterekkel, regisztertömbökkel és operátorokkal egyenrangú RTL-komponens.

A 2-21. ábrán az állapotveremtár *din* adatbemenetének [15:0] bitjeire és az utasításmutatóhoz tartozó *IP_add* összeadónak az operandusaira vonatkozó adatkapcsolatokat látjuk. A felső két hozzárendeléspár alapján azt állapíthatjuk meg, hogy mind az állapotveremtár *(STACK) din* adatbemenetére, mind az *IP_add* operátor *opb* adatbemenetére egy-egy kétbemenetű multiplexer beiktatása szükséges.

RTL-netlista

```
1  STACK.din[15:0] <= IP
2  STACK.din[15:0] <= IP_add.result
3
4  IP_add.opb <= signed(IR[11:0])
5  IP_add.opb <= 1
6
7  IP_add.opa <= IP
```

RTL-netlista multiplexerekkel

```
1  mux_STACK.din0 <= IP
2  mux_STACK.din1 <= IP_add.result
3  STACK.din[15:0] <= mux_STACK.dout
4
5  mux_IP_add.din0 <= signed(IR[11:0])
6  mux_IP_add.din1 <= 1
7  IP_add.opb <= mux_IP_add.dout
8
9  IP_add.opa <= IP
```

2-21. ábra. *Multiplexer beiktatása az állapotveremtár adatbemenetére és az IP_add operátor bemenetére (1)*

Ha figyelembe vesszük, hogy az *IP_add* operátor *opa* bemenetén mindig az utasításmutató szerepel, akkor egy apró trükkel az állapotveremtár *din* adatbemenetén lévő multiplexer megspórolható. Alakítsuk át az állapotveremtár *din* adatbemenetére és az *IP_add* operátor *opb* bemenetére vonatkozó hozzárendelések listáját a 2-22. ábra bal oldalán látható módon.

RTL-netlista

```
1  STACK.din[15:0] <= IP
2  STACK.din[15:0] <= IP_add.result
3
4  IP_add.opb <= signed(IR[11:0])
5  IP_add.opb <= 1
6  IP_add.opb <= 0
7
8  IP_add.opa <= IP
```

RTL-netlista multiplexerekkel

```
1  STACK.din[15:0] <= IP_add.result
2
3  mux_IP_add.din0 <= signed(IR[11:0])
4  mux_IP_add.din1 <= 1
5  mux_IP_add.din2 <= 0
6  IP_add.opb <= mux_IP_add.dout
7
8  IP_add.opa <= IP
```

2-22. ábra. *Multiplexer beiktatása az állapotveremtár adatbemenetére és az IP_add operátor bemenetére (2)*

A 6. sorban hozzáadott új adatkapcsolat lehetővé teszi azt, hogy az *IP* értékéhez az *IP_add* operátoron keresztül – valójában az *IP*-hez nullát adva – férjünk hozzá. Így az 1. sorban lévő hozzárendelés feleslegessé válik. A 2-22. ábra jobb oldalán láthatjuk, hogy így tulajdonképpen a két darab kétbemenetű multiplexert egy darab hárombemenetűre cseréltük. Áramköri szempontból ennek sok jelentősége nincs, hiszen a hárombemenetű multiplexer több erőforrást igényel, mint egy kétbemenetű, ugyanakkor a műveletvégző egység vezérlőjeleit (beleértve a multiplexerek kiválasztójeleit) előállító állapotgép tervezésekor a vezérlendő objektumok számának csökkentésével jelentősen egyszerűsíthetjük saját dolgunkat.

A 2-23. ábra az *SP* regiszter inkrementálását és dekrementálását végző operátor, az *SP_add* bemenetét meghajtó multiplexer beiktatását mutatja.

RTL-netlista

```
1  SP_add.opa <= SP
2
3  SP_add.opb <= -1
4  SP_add.opb <= 1
5
6  SP <= SP_add.result
```

RTL-netlista multiplexerekkel

```
1  SP_add.opa <= SP
2
3  mux_SP_add.din0 <= -1
4  mux_SP_add.din1 <= 1
5  SP_add.opb <= mux_SP_add.dout
6
7  SP <= SP_add.result
```

2-23. ábra. *Multiplexer beiktatása az SP_add operátor bemenetére*

A következő többszörös hozzárendelés a regisztertömb *raddr_a* bemenetére vonatkozik. A dekódolás fázisban erre a címbemenetre az utasításregiszter [7:4], a perifériaműveletek fázisában pedig a [11:8] bitjeit kell rákapcsolnunk (lásd 2-24. ábra).

RTL-netlista

```
1  RF.raddr_a <= IR[7:4]
2  RF.raddr_a <= IR[11:8]
3
4  RF.raddr_b <= IR[3:0]
5  RF.waddr <= IR[11:8]
```

RTL-netlista multiplexerekkel

```
1  mux_RF.din0 <= IR[7:4]
2  mux_RF.din1 <= IR[11:8]
3  RF.raddr_a <= mux_RF.dout
4
5  RF.raddr_b <= IR[3:0]
6  RF.waddr <= IR[11:8]
```

2-24. ábra. *Multiplexer beiktatása a regisztertömb raddr_a bemenetére (1)*

Figyeljük meg, hogy a regisztertömb beírásra szolgáló címbemenetének *(waddr)* meghajtására az utasításregiszternek ugyanazokat a bitjeit használjuk, mint amiket az *raddr_a* címbemenet esetén használunk egy multiplexeren keresztül. Mivel a regisztertömbből való olvasás és az abba való beírás sosem történhet egyszerre – ti. a kiolvasás a perifériaműveletek fázisban, a beírás pedig

a visszaírás fázisban történik –, ezért megtehetjük, hogy a *waddr* bemenet meghajtására nem közvetlenül az utasításregisztert, hanem a *mux_RF* multiplexernek a kimenetét használjuk (lásd 2-25. ábra).

RTL-netlista

```
1  RF.raddr_a <= IR[7:4]
2  RF.raddr_a <= IR[11:8]
3
4  RF.raddr_b <= IR[3:0]
5  RF.waddr <= IR[11:8]
```

RTL-netlista multiplexerekkel

```
1  mux_RF.din0 <= IR[7:4]
2  mux_RF.din1 <= IR[11:8]
3  RF.raddr_a <= mux_RF.dout
4
5  RF.raddr_b <= IR[3:0]
6  RF.waddr <= mux_RF.dout
```

2-25. ábra. *Multiplexer beiktatása a regisztertömb raddr_a bemenetére (2)*

Ezzel a lépéssel lehetőséget adunk az áramkörszintézis-szoftvernek némi erőforrás-optimalizálásra a következők szerint: A két független interfészen keresztül olvasható és egy ugyancsak független interfészen keresztül írható *singleclock_tripleport_ram* IP az egy-egy beíró/kiolvasó interfésszel rendelkező, egyszerű *singleclock_dualport_ram* IP-ből a belső tartalom megduplázásával származtatható (lásd részletesen: 4.2.1. pont). Abban az esetben, ha az egyik kiolvasó interfész címbemenetére ugyanazt a vektort kapcsoljuk, mint a beíró interfész címbemenetére, akkor lényegében egy olyan regisztertömböt írunk le, amelynek egy kiolvasó és egy kombinált beíró/kiolvasó interfésze van. Ezt a működést az FPGA-kban megtalálható blokk-RAM modulok a memóriatartalom többszörözése nélkül is képesek megvalósítani. A 2-25. ábrán látható megoldást alkalmazva tehát azt fogjuk tapasztalni, hogy bár az RTL-modellünk egy *singleclock_tripleport_ram* IP-t tartalmaz, amelytől azt várnánk, hogy fizikailag kétszer annyi tárolóelemet igényel, mint amennyit logikailag használunk, az áramkörszintézis után mégis azt látjuk majd, hogy a feladatot a szintézis-szoftver a memóriatartalom duplikálása nélkül is meg tudja oldani.

Ha szemügyre vesszük a dekódolás és a perifériaműveletek fázisához tartozó RTL-netlistákat, láthatjuk, hogy a végrehajtás különböző pillanataiban az *Y* regiszterbe is más-más értékeket kell tudnunk betölteni. A 2-26. ábra az ehhez szükséges multiplexer beiktatását szemlélteti.

RTL-netlista

```
1  Y <= unsigned(IR[7:0])
2  Y <= RF.dout_b
3  Y <= data_from_periph
```

RTL-netlista multiplexerekkel

```
1  mux_Y.din0 <= unsigned(IR[7:0])
2  mux_Y.din1 <= RF.dout_b
3  mux_Y.din2 <= data_from_periph
4  Y <= mux_Y.dout
```

2-26. ábra. *Multiplexer beiktatása az Y regiszter bemenetére*

A *Z*, *N* és *EI* állapotjelzőkbe ugyancsak többféle forrásból tölthetünk adatot (lásd 2-27. ábra). A *Z* és *N* regisztereket az aritmetikai/logikai utasítások végén az *ALU* megfelelő kimenetei alapján frissítjük, a *ret* utasítás végrehajtásakor pedig az állapotveremtár adatkimenetének megfelelő bitjeit használjuk értékük beállítására.

Az *EI* regiszterbe a megszakítások kiszolgálásakor, illetve a *call* utasítás végrehajtásakor automatikusan nullát, ez *ei* utasítás végrehajtásakor pedig az utasításregiszter legalsó bitjében meghatározott értéket kell betöltenünk. Ugyanakkor ez *EI* regiszter is egy állapotjelző, amelybe a szubrutinokból vagy megszakítási rutinokból való visszatéréskor a *Z* és *N* állapotjelzőkhöz hasonlóan az állapotveremtár adatkimenetének megfelelő bitjét kell beírnunk.

RTL-netlista

```
1  Z <= STACK.dout[17]
2  Z <= ALU.z
3
4  N <= STACK.dout[16]
5  N <= ALU.n
6
7  EI <= 0
8  EI <= IR[0]
9  EI <= STACK.dout[18]
```

RTL-netlista multiplexerekkel

```
1  mux_Z.din0 <= STACK.dout[17]
2  mux_Z.din1 <= ALU.z
3  Z <= mux_z.dout
4
5  mux_N.din0 <= STACK.dout[16]
6  mux_N.din1 <= ALU.n
7  N <= mux_N.dout
8
9  mux_EI.din0 <= 0
10 mux_EI.din1 <= IR[0]
11 mux_EI.din2 <= STACK.dout[18]
12 EI <= mux_EI.dout
```

2-27. ábra. *Multiplexer beiktatása a Z, N és EI állapotjelzők bemenetére*

3. lépés – Hardverleíró nyelvű modell létrehozása ◆ A multiplexerekkel kiegészített RTL-netlista már minden szükséges információt tartalmaz ahhoz, hogy belőle a műveletvégző egység hardverleíró nyelvű modelljét elkészítsük. A HDL-modellt és annak grafikus reprezentációját, az RTL-áramköri sémát a 2-28–2-33. ábrák szemléltetik.

A 2-28. ábra az utasításfelhozás fázishoz tartozó erőforrások RTL-modelljét mutatja be. Magát az utasításmemóriát a műveletvégző egység részének tekintjük. Erre a célra egy általános célú, újrafelhasználható IP-magot (*singleclock_dualport_ram_hswrp*: 4.2.3. pont) használunk, amelynek példányosítását a VHDL-modell 25-39. sora írja le. A modul beíró interfészének portjaira *(waddr, wstrobe, rdy2wr, din)* egyelőre nem kötünk jeleket. A *CPU* belső részegységei – beleértve magát a vezérlő állapotgépet is – feltételezik, hogy a programtárban a végrehajtás megkezdésekor már jelen van maga a program.[38]

Az utasításmemória példányosításakor hivatkozunk az *imem_addr_width* paraméterre, amely a programtár címbuszának méretére utal. A *CPU* a 16-bites utasításmutatóból adódóan akár egy 65.536 utasítást tartalmazó programtárat is képes lenne kezelni, erre azonban az alkalmazások többségében nincs szükség. Annak érdekében, hogy a *CPU* ne emésszen fel feleslegesen sok blokk-RAM-modult az FPGA-ban, a programtár méretét az *imem_addr_width* szintézis-paraméter segítségével korlátozzuk. Ennek értékét a felhasználó az általa írt program mérete alapján beállíthatja. Az utasításmutatót és az utasításregisztert egy közös szinkron *process*-ben írjuk le (2-28. ábra: 16–23. sor). Mindkét regiszter esetén – csakúgy, mint az összes többi regiszternél is – használjuk a *ce (clock enable)* funkciót, ami lehetővé teszi a vezérlő állapotgép számára, hogy csak adott órajelciklusban engedélyezze a regiszterek értékének átírását az aktuális bemenetük alapján. A vezérlő állapotgép az utasításmutatóba a *sel_IP* jel megfelelő beállításával képes betölteni a *reset_vector* és az *interrupt_vector* névvel hivatkozott értékeket. A *reset_vector* az a cím, ahonnan a *CPU* reset után az első gépi utasítást beolvassa, az *interrupt_vector* pedig az, ahol a futtatandó program írója a központi megszakításkiszolgáló rutint helyezi el. Mivel a különböző kódrészletek programtárbeli elhelyezése a felhasználó feladata, ezért e két speciális cím meghatározását ugyancsak a felhasználóra kell bíznunk. E két konstanst tehát az *imem_addr_width*-hez hasonlóan szintézis-paraméterként valósítjuk meg.

[38] A programtár tartalommal való feltöltését a *BOOTSTRAP* modul végzi (lásd 2.3. pont), közvetetten egy órajel-szinkronizáló blokkon keresztül (lásd 2.2.4. pont).

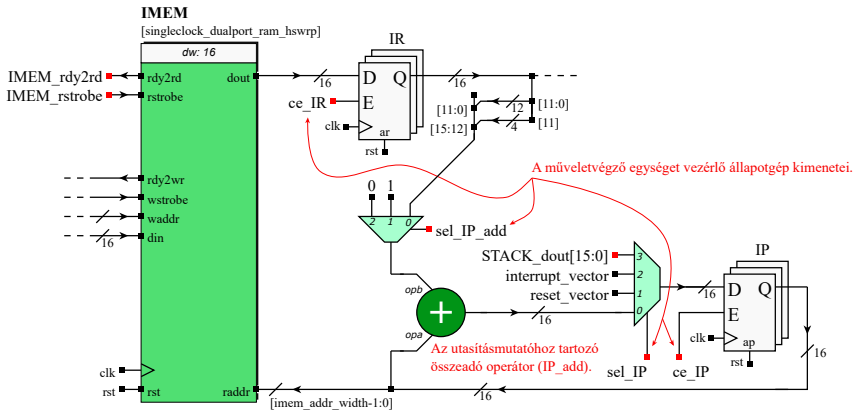
RTL-netlista

```
1  mux_IP_add.din0 <= signed(IR[11:0])
2  mux_IP_add.din1 <= 1
3  mux_IP_add.din2 <= 0
4  IP_add.opb <= mux_IP_add.dout
5
6  IP_add.opa <= IP
7
8  mux_IP.din0 <= IP_add.result
9  mux_IP.din1 <= reset vector
10 mux_IP.din2 <= interrupt vector
11 mux_IP.din3 <= STACK.dout[15:0]
12 IP <= mux_IP.dout
13
14 IMEM.raddr <= IP
15 IR <= IMEM.dout
```

VHDL – műveletvégző egység

```
1  L_SIGN_EXT: sign_ext(15 downto 12) <= (others => ir(11));
2              sign_ext(11 downto 0) <= IR(11 downto 0);
3
4  L_MUX_IP_ADD: mux_IP_add <= sign_ext when sel_IP_add = "00"
5                              else X"0001" when sel_IP_add = "01"
6                              else X"0000";
7
8  L_IP_ADD: IP_add_result <= std_logic_vector(unsigned(IP) +
9                             unsigned(mux_IP_add));
10
11 L_MUX_IP: mux_IP <= IP_add_result when sel_IP = "00" else
12                    reset_vector when sel_IP = "01" else
13                    interrupt_vector when sel_IP = "10" else
14                    STACK_dout(15 downto 0);
15
16 L_IP: process ( clk, rst ) begin
17       | if ( rst = '1' ) then
18       |   IP <= reset_vector; IR <= (others => '0');
19       | elsif ( rising_edge(clk) ) then
20       |   if ( ce_IP = '1' ) then IP <= mux_IP; end if;
21       |   if ( ce_IR = '1' ) then IR <= IMEM_dout; end if;
22       | end if;
23      end process;
24
25 L_IMEM: entity work.singleclock_dualport_ram_hswrp(rtl)
26          generic map (aw => imem_addr_width,
27                       dw => 16)
28          port map (
29           | clk     => clk,
30           | rst     => rst,
31           | waddr   => [...],
32           | raddr   => IP(imem_addr_width-1 downto 0),
33           | wstrobe => [...],
34           | rstrobe => IMEM_rstrobe,
35           | rdy2rd  => IMEM_rdy2rd,
36           | rdy2wr  => [...],
37           | din     => [...],
38           | dout    => IMEM_dout
39          );
```

Áramköri struktúra– műveletvégző egység



2-28. ábra. *Az utasításfelhozás fázis adatkapcsolatait megvalósító áramköri összetevők részletes RTL-modellje*[39]

[39] A flip-flopok és regiszterek aszinkron törlő bemenetét az RTL kapcsolási sémákon *„ar”* (***a**synchronous **r**eset)* névvel szerepeltetjük, ha a reset hatására felvett értékük nulla és *„ap”* (***a**synchronous **p**reset)* névvel, ha attól eltérő (pl. az *IP* regiszterbe reset hatására a *reset_vector* paraméter értéke íródik, ami általános esetben nullától eltérő). Az RTL kapcsolási rajzokon a példányosított újrafelhasználható részegységek szintézis-paramétereit az áramköri szimbólumok fejlécében csak akkor tüntetjük fel, ha azokat a példányosításkor egy fix értékre állítjuk be. A fel nem tüntetett paraméterek a beágyazó almodulnak is paraméterei lesznek (pl. az utasításmemória *dw* paramétere a *CPU* műveletvégző egységében fixen 16, de az *aw* paraméter a *CPU* HDL-modelljének is paramétere).

A 2-29. ábra az általános célú regisztertömb, valamint az *X* és *Y* regiszterek VHDL-modelljét és áramköri szerkezetét szemlélteti. Az általános célú regisztertömböt szintén egy újrafelhasználható IP-mag (*singleclock_tripleport_ram*: 4.2.1. pont) képviseli. A VHDL-modell 12. és 13. sorában azt láthatjuk, hogy ennek az általános memóriamodulnak az *rstrobe_a/b* bemeneteit a *CPU* modellje nem használja ki. E két bemenet konstans magas szinten tartásával azt érjük el, hogy a címbemenetek megváltozásának hatására a megfelelő adatkimenet az órajel következő felfutóélénél rögtön megváltozik (lásd 4-15. ábra).

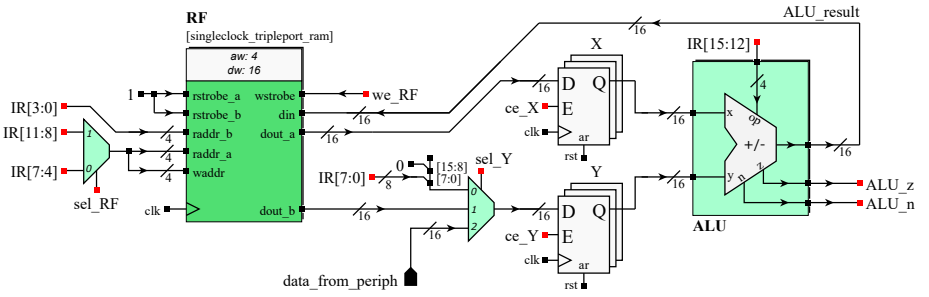
RTL-netlista

```
1  mux_RF.din0 <= IR[7:4]
2  mux_RF.din1 <= IR[11:8]
3  RF.raddr_a <= mux_RF.dout
4  RF.raddr_b <= IR[3:0]
5
6  RF.waddr <= mux_RF.dout
7  RF.din <= ALU.result
8
9  X <= RF.dout_a
10
11 mux_Y.din0 <= unsigned(IR[7:0])
12 mux_Y.din1 <= RF.dout_b
13 mux_Y.din2 <= data_from_periph
14 Y <= mux_Y.dout
15
16 ALU.op <= IR[15:12]
17 ALU.x <= X
18 ALU.y <= y
```

VHDL – műveletvégző egység

```
1  L_MUX_RF: mux_RF <= IR(7 downto 4) when sel_RF = '0'
2                      else IR(11 downto 8);
3
4  L_RF: entity work.singleclock_tripleport_ram(rtl)
5          generic map (aw => 4, dw => 16)
6          port map (
7            clk       => clk,
8            waddr     => mux_RF,
9            raddr_a   => mux_RF,
10           raddr_b   => IR(3 downto 0),
11           wstrobe   => we_RF,
12           rstrobe_a => '1',
13           rstrobe_b => '1',
14           din       => ALU_result,
15           dout_a    => RF_dout_a,
16           dout_b    => RF_dout_b
17         );
18
19 L_IMM_EXT: imm_ext(15 downto 8) <= (others => '0');
20            imm_ext(7 downto 0) <= IR(7 downto 0);
21
22 L_MUX_Y: mux_y <= imm_ext when sel_Y = "00" else
23                  RF_dout_b when sel_Y = "01" else
24                  data_from_periph;
25
26 L_X_Y: process ( clk, rst ) begin
27   if ( rst = '1' ) then
28     X <= (others => '0'); Y <= (others => '0');
29   elsif ( rising_edge(clk) ) then
30     if ( ce_X = '1' ) then X <= RF_dout_a; end if;
31     if ( ce_Y = '1' ) then Y <= mux_Y; end if;
32   end if;
33 end process;
34
35 L_ALU: [...]
```

Áramköri struktúra – műveletvégző egység



2-29. ábra. *A regisztertömb, valamint az X és Y operandusregiszterek adatkapcsolatai és részletes RTL-modelljük*

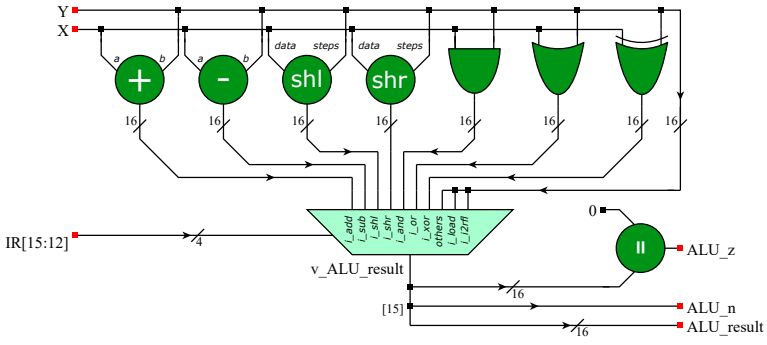
Az aritmetikai/logikai egység 2-30. ábrán bemutatott HDL-modelljében arra láthatunk példát, amikor egy viszonylag összetett feladatot ellátó kombinációs hálózat működését nem tudjuk, vagy legalábbis nem praktikus egyetlen konkurens értékadással leírni. Ilyen esetben bizonyul

hasznosnak a kombinációs folyamatok *(process)* és a változók *(variable)* használata. A kombinációs folyamat jellegzetessége, hogy érzékenységi listája mindazokat a külső jeleket tartalmazza, amelyek a leendő kombinációs hálózatnak bemenetei, vagyis minden olyan jelet, amely értékadások jobb oldalán vagy feltételes utasítások feltétel-kifejezéseiben szerepel. Így azt fejezzük ki, hogy bármely bemenet megváltozása esetén a kimeneteket ki kell értékelni, ami egy kombinációs hálózattól is elvárt viselkedés; bármely bemenet megváltozása a kimenet azonnali megváltozását idézheti elő.[40] Ennek megfelelően az *ALU* működését leíró folyamat érzékenységi listájában az utasításregiszter felső négy bitjét, valamint az *X* és *Y* regisztereket soroljuk fel.

VHDL – műveletvégző egység

```
1  L_ALU: process ( IR(15 downto 12), X, Y )
2    variable v_ALU_result: std_logic_vector (15 downto 0);
3  begin
4    case ( IR(15 downto 12) ) is
5      when i_i2rfl    =>  v_ALU_result := Y;
6      when i_load     =>  v_ALU_result := Y;
7      when i_add      =>  v_ALU_result := std_logic_vector(unsigned(X) + unsigned(Y));
8      when i_sub      =>  v_ALU_result := std_logic_vector(unsigned(X) - unsigned(Y));
9      when i_and      =>  v_ALU_result := X and Y;
10     when i_or       =>  v_ALU_result := X or Y;
11     when i_xor      =>  v_ALU_result := X xor Y;
12     when i_shl      =>  v_ALU_result := std_logic_vector(unsigned(X) sll to_integer(unsigned(Y)));
13     when i_shr      =>  v_ALU_result := std_logic_vector(unsigned(X) srl to_integer(unsigned(Y)));
14     when others     =>  v_ALU_result := Y;
15   end case;
16   ALU_result <= v_ALU_result;
17
18   if ( v_ALU_result = X"0000" ) then ALU_z <= '1'; else ALU_z <= '0'; end if;
19   ALU_n <= v_ALU_result(15);
20
21 end process;
```

Áramköri struktúra – műveletvégző egység



2-30. ábra. *Az aritmetikai/logikai egység (ALU) részletes RTL-modellje*

[40] Gyakori HDL-modellezési hiba, hogy egy kombinációs hálózat leírására szánt folyamat érzékenységi listája nem tartalmaz minden bemenő jelet. Az ilyen folyamatok viselkedését pontosan visszaadó áramkörök latch-ek segítségével szintetizálhatók ugyan, de ezek működése valószínűleg nem felel meg a tervező elvárásainak. A VHDL 2008-as verziója ezért bevezette a *process (all)* kifejezést, ami automatikusan minden, a folyamat törzsében jobbértékként és/vagy feltétel-kifejezésben használt jelet az érzékenységi lista részének tekint. Sajnos a kifejezést nem minden szintézis-szoftver támogatja. Újrafelhasználható IP-magok tervezésekor használata kerülendő (vö. technológiafüggetlenség). Verilog esetén kombinációs folyamatot az *always @ (*)* vagy az *always_comb* kifejezéssel hozhatunk létre (ez utóbbit a SystemVerilog vezette be, a nyelv korábbi verziói ezt a formát nem tartalmazzák). A VHDL-beli változókéval azonos viselkedést Verilogban a blokkoló értékadások használatával érhetjük el.

Az utasításregiszter felső négy bitje kódolja, hogy melyik gépi utasításról van szó, tehát hogy milyen műveletet kell elvégezni a két operanduson. A folyamat törzsében ezt egy *case* szerkezettel fejezzük ki. Az elvégzett műveleten felül – annak eredménye alapján – képeznünk kell a *z (zero)* és *n (negative)* állapotjelzőket is, ezért a *case* szerkezet belsejében megfogalmazott aritmetikai/logikai kifejezések eredményét egy változóban tároljuk. Ily módon a *case* szerkezet utáni kódrészletben a *v_ALU_result* kifejezéssel már az éppen aktuális eredményt érhetjük el, felhasználva azt további kimenetek előállítására úgy, hogy a teljes modul kívülről továbbra is egyetlen összefüggő kombinációs hálózatként viselkedik. A 2-30. ábra kapcsolási sémája azt szemlélteti, hogy a változó létrehozásával tulajdonképpen „nevén neveztünk” egy olyan buszt a leendő áramkörben, amely a változó használata nélkül magában a HDL-modellben hivatkozhatatlan lett volna, csak a szintézis során, a kapuszintű netlistában kapott volna konkrét azonosítót.

A 2-31. ábra a *Z, N* és *EI* állapotjelzőknek, a 2-32. ábra pedig a perifériaműveletek segédregisztereinek, a *PDR* és *PAR* regisztereknek a HDL-leírásait és áramköri szerkezetét mutatja be.

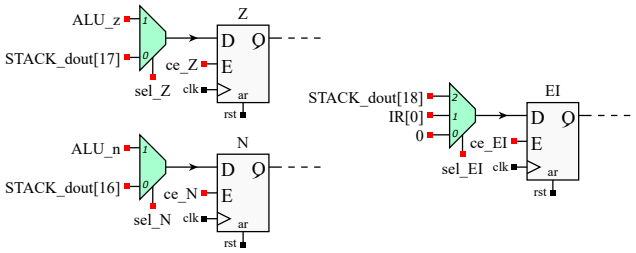
RTL-netlista

```
1  mux_Z.din0 <= STACK.dout[17]
2  mux_Z.din1 <= ALU.z
3  Z <= mux_Z.dout
4
5  mux_N.din0 <= STACK.dout[16]
6  mux_N.din1 <= ALU.n
7  N <= mux_N.dout
8
9  mux_EI.din0 <= 0
10 mux_EI.din1 <= IR[0]
11 mux_EI.din2 <= STACK.dout[18]
12 EI <= mux_EI.dout
```

VHDL – műveletvégző egység

```
1  L_MUX_Z: mux_Z <= STACK_dout(17) when sel_Z = '0'
2                    else ALU_z;
3
4  L_MUX_N: mux_N <= STACK_dout(16) when sel_N = '0'
5                    else ALU_n;
6
7  L_MUX_EI: mux_EI <= '0' when sel_EI = "00"
8                      else IR(0) when sel_EI = "01"
9                      else STACK_dout(18);
10
11 L_Z_N_EI: process ( clk, rst ) begin
12   if ( rst = '1' ) then
13     Z <= '0'; N <= '0'; EI <= '0';
14   elsif ( rising_edge(clk) ) then
15     if ( ce_Z = '1' ) then Z <= mux_Z; end if;
16     if ( ce_N = '1' ) then N <= mux_N; end if;
17     if ( ce_EI = '1' ) then EI <= mux_EI; end if;
18   end if;
19 end process;
```

Áramköri struktúra – műveletvégző egység



2-31. ábra. *Az állapotjelzők adatkapcsolatai és részletes RTL-modelljük*

Végül a 2-33. ábrán láthatjuk az állapotverem-műveletekhez tartozó erőforrások HDL-modelljét és RTL kapcsolási sémáját. Az állapotveremtár megvalósítására a *singleclock_dualport_ram* IP-magot használjuk. Az általános célú regisztertömbnél látottakhoz hasonlóan itt is konstans magas szintre állítjuk a kiolvasó interfészhez tartozó *rstrobe* jelet (21. sor), így az *SP* regiszter megváltozásának hatására a memóriamodul adatkimenete rögtön az órajel következő felfutóélénél frissül.

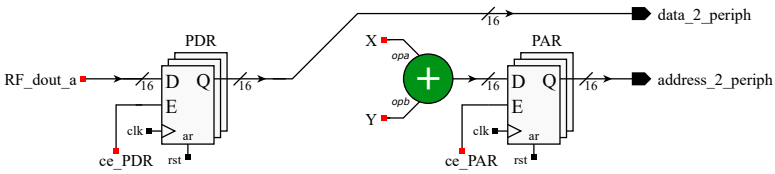
RTL-netlista

```
1  PDR <= RF.dout_a
2
3  PAR_add.opa <= X
4  PAR_add.opb <= Y
5  PAR <= PAR_add.result
6
7  address_2_periph <= PAR
8  data_2_periph <= PDR
```

VHDL – műveletvégző egység

```
1  L_PAR_ADD: PAR_add_result <= std_logic_vector(unsigned(X) + unsigned(Y));
2
3  L_PDR_PAR: process ( clk, rst ) begin
4    if ( rst = '1' ) then
5      PDR <= (others => '0'); PAR <= (others => '0');
6    elsif ( rising_edge(clk) ) then
7      if ( ce_PDR = '1' ) then PDR <= RF_dout_a; end if;
8      if ( ce_PAR = '1' ) then PAR <= PAR_add_result; end if;
9    end if;
10 end process;
11
12 data_2_periph <= PDR;
13 address_2_periph <= PAR;
```

Áramköri struktúra – műveletvégző egység



2-32. ábra. *A perifériaműveletek segédregisztereinek adatkapcsolatai és részletes RTL-modelljük*

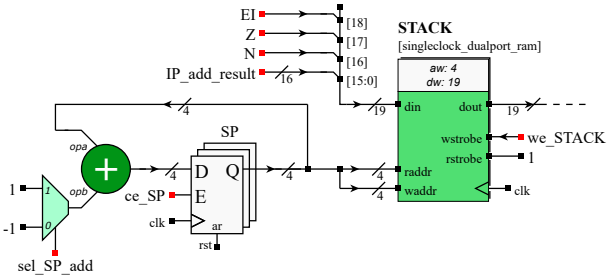
RTL-netlista

```
1  SP_add.opa <= SP
2
3  mux_SP_add.din0 <= -1
4  mux_SP_add.din1 <= 1
5  SP_add.opb <= mux_SP_add.dout
6
7  SP <= SP_add.result
8
9  STACK.din[18] <= EI
10 STACK.din[17] <= Z
11 STACK.din[16] <= N
12 STACK.din[15:0] <= IP_add.result
13
14 STACK.raddr <= SP
15 STACK.waddr <= SP
```

VHDL – műveletvégző egység

```
1  L_MUX_SP_ADD: mux_SP_add <= "1111" when sel_SP_add = '0'
2                              else "0001";
3
4  L_SP_ADD: SP_add_result <= std_logic_vector(unsigned(SP) +
5                                              unsigned(mux_SP_add));
6
7  L_SP: process ( clk, rst ) begin
8    if ( rst_core = '1' ) then SP <= "0000";
9    elsif ( rising_edge(clk) ) then
10     if ( ce_SP = '1' ) then SP <= SP_add_result; end if;
11   end if;
12 end process;
13
14 L_STACK: entity work.singleclock_dualport_ram(rtl)
15           generic map (aw => 4, dw => 19)
16           port map (
17             clk      => clk,
18             waddr    => SP,
19             raddr    => SP,
20             wstrobe  => we_STACK,
21             rstrobe  => '1',
22             din      => EI & Z & N & IP_add_result,
23             dout     => STACK_dout
24           );
```

Áramköri struktúra – műveletvégző egység



2-33. ábra. *Az állapotverem-műveletekhez tartozó részegységek adatkapcsolatai és részletes RTL-modelljük*

A műveletvégző egység HDL-modelljének néhány sajátosságát érdemes külön kiemelni:

- **Regularitás** – A viselkedési modell alapján szisztematikusan előállított RTL-netlista implicit módon létrehozott bizonyos alkatrész-kategóriákat; regiszter, regisztertömb, operátor és multiplexer. Az áttekinthetőség javítása céljából célszerű egységességre, regularitásra törekednünk a HDL-modell részleteinek kidolgozásakor is, ezért az egyes erőforrás-kategóriákba tartozó komponenseket ugyanazokkal a VHDL-utasítástípusokkal modellezzük. A multiplexereket és az operátorokat konkurens értékadások, a regisztereket pedig szinkron folyamatok írják le.[41]
- **Modularitás**, egységbe zárás – A regisztertömbök funkciója, hogy adott számú bináris vektort tároljanak. Ez egy jól körülhatárolható, a részegységet körülvevő áramköri részletektől függetleníthető funkció, amit újrafelhasználható IP-magként érdemes elkészíteni és a lehető legtöbb helyen használni (lásd utasításmemória: *IMEM*, általános célú regisztertömb: *RF*, állapotveremtár: *STACK*).
- A multiplexerek és aritmetikai/logikai operátorok a VHDL-modellben egy-egy konkurens értékadással reprezentáltak, amelyeket áramköri szinten kombinációs hálózatok valósítanak majd meg. Ennek a tulajdonságnak a vezérlő állapotgép VHDL-modelljének megszerkesztésekor lesz jelentősége. A konkurens értékadással leírt áramköri részletek bemenetein bekövetkező változások hatása azonnal megjelenik azok kimenetein, vagyis az új értékekre már a bemenetek változását előidéző órajelél utáni első felfutóél pillanatában számíthatunk.[42]

Pár gondolat a technológiafüggetlenségről…

Az RTL-tervezés követendő alapelvei között felsoroltuk a technológiafüggetlenséget is, vagyis azt, hogy az RTL-modelljeinket célszerű úgy megalkotnunk, hogy azok változtatás nélkül szintetizálhatók legyenek a lehető legtöbb FPGA gyártó különféle eszközeire. Ha különböző „technológiák" alatt csak a különböző programozható logikai eszközöket értjük, akkor arra kell törekednünk, hogy ne alkalmazzunk olyan megoldásokat, amelyek csak az egyik vagy csak a másik gyártó eszközei esetén működőképesek. A technológiafüggetlenség fogalmát némileg szigorúbban értelmezve kitűzhetjük célként azt is, hogy modelljeink ne csak FPGA-ra, hanem standard cellakönyvtárakra is leképezhetőek, szintetizálhatóak legyenek, ez azonban egy sokkal nehezebb feladat e két megvalósítás alapvető kapu- és tranzisztorszintű különbségeiből adódóan.
A CPU-modellünk műveletvégző egységének RTL-modelljében négy aritmetikai áramkört használunk: az ALU egy általános célú, többfunkciós aritmetikai/logikai egység, az IP_add, az SP_add és a PAR_add pedig egyedi funkciójú regiszterekhez rendelt, dedikált összeadó áramkörök. Felmerülhet a kérdés, hogy ez utóbbiak helyett a megfelelő vezérléssel miért nem használjuk az általános célú ALU-t ezeknek az összegeknek a képzésére, hiszen összeadni az ALU is tud. Általában az erőforrások vezérelt újrafelhasználása (lásd DRMSz: 4.1. pont) a párhuzamosítás igénye miatt nem jöhet szóba; nagy számítási teljesítményhez több, párhuzamosan működő erőforrásra van szükség. Esetünkben azonban nem erről van szó, nyugodtan megtehettük volna, hogy az ALU-t a végrehajtás különböző fázisaiban más-más célra használjuk, ily módon feleslegessé téve a három másik aritmetikai egységet. Azonban figyeljük meg, hogy a három dedikált

[41] A regiszterek leírhatók az itt láthatótól eltérő módon is, a konkurens értékadások pedig helyettesíthetők kombinációs folyamatokkal (*process* vagy *always-blokk*), a lényeg az egységesség. Azt, hogy ezt hogyan érjük el, tulajdonképpen már ízlés és legfőképp megszokás dolga.

[42] Ez azt jelenti, hogy az RTL-modell előállításakor a konkurens értékadásokkal leírt, a leendő áramkörünkben kombinációs hálózattal megvalósított részegységek késleltetését gyakorlatilag nullának tekintjük. Az automatizált áramkörszintézis, azon belül is a fizikai szintézis feladata lesz ennek a feltételnek a biztosítása (lásd DRMSz: 4.4.4. alfejezet).

összeadó bármelyikének kiváltása csak többlet multiplexálási képesség beépítésével volna lehetséges, hiszen ekkor a dedikált összeadók lehetséges bemeneteinek az ALU bemeneteire való rákapcsolását is lehetővé kellene tennünk. A PAR_add összeadó operandusai ugyan megegyeznek az ALU-éival, de a store és load gépi utasítások végrehajtásának különböző fázisaiban az ALU funkciókiválasztó bemenetére különböző értékeket kellene kapcsolni, ami ugyancsak egy többlet multiplexert igényelne. Az aritmetikai operátorok vezérelt újrafelhasználása tehát láthatóan a multiplexálást végző erőforrások mennyiségének növekedésével jár.

A programozható eszközökben nagy mennyiségben állnak rendelkezésre kedvező időzítési tulajdonságokkal bíró aritmetikai erőforrások, míg a multiplexálás újabb kapcsolóelemek beiktatásával jár, amelyek tovább rontják az amúgy is kapcsolóelemek sokaságával terhelt jelutak időzítési jellemzőit. Standard cellás esetben a helyzet fordított; az aritmetikai áramkörök tipikusan nagy méretűek és nagy fogyasztásúak, a multiplexálás viszont igen hatékonyan kezelhető transzferkapus cellák használatával. Összességében elmondható, hogy programozható logikai eszközök esetén inkább több aritmetikát, standard cellás áramkörök esetén inkább több multiplexálást célszerű alkalmazni. Így aztán kénytelenek vagyunk elfogadni, hogy bár készíthetünk olyan mikroarchitektúrát, amely mindkét technológia esetén szintetizálható, de olyat aligha, amely mindkét esetben optimális. Az e fejezetben bemutatott CPU műveletvégző egység például egészen másképp nézne ki, ha nem FPGA-ra, hanem standard cellás megvalósításra szánnánk.

A műveletvégző egység 2-28. és 2-33. ábrákon látható részáramköreinek megfelelően összekapcsolt hálózatát a viselkedési modell értékadásaiból képezett adatkapcsolatok listája alapján hoztuk létre. A műveletvégző egység azonban önmagában nem képes megvalósítani a viselkedési modellben megfogalmazott összetett működést. Ehhez egy olyan további részegységre lesz szükség, ami a viselkedési modell vezérlési szerkezeteinek (ciklus, elágazás) megfelelően a műveletvégző egység alkatrészeinek működését összehangolja azok vezérlő- és állapotjeleinek (regiszterek órajel-engedélyező bemenetei, multiplexerek kiválasztójelei stb.) segítségével. Ezt a szerepet funkcióbővítő rendszerünk központi egységében egy állapotgép tölti be.

2.2.3. Vezérlőegység

Egy *CPU* vezérlőegységét megvalósító állapotgép HDL-leírását többféleképpen is elkészíthetjük (lásd DRMSz: 4.5.5. alfejezet). A lehetséges megoldások közül ebben az alfejezetben az ún. one-shot HDL-kódolási módszert választjuk. A műveletvégző egységnél látottakhoz hasonlóan a vezérlőegység részletes tervezésénél is a viselkedési modellt hívjuk segítségül. Ezúttal azonban nem a pszeudokód értékadásaira, hanem annak vezérlési szerkezeteire koncentrálunk. A 2-34. ábra az állapotgép VHDL-modelljének vázlatát szemlélteti.

Az állapotgép állapotregiszterét egy felsorolt típusú jellel írjuk le *(state)*. Ennek bináris kódolásával a HDL-modellezés során nem foglalkozunk, azt a szintézis-szoftverre bízzuk. A one-shot leírási módszer az állapotgép egészét egyetlen szinkron folyamat (*process* vagy *always*-blokk) segítségével írja le, amelynek jellemzője, hogy a törzsében egyetlen *if-elsif* utasítás foglal helyet. Az *if* ágban (2. sor) fejezzük ki a leendő áramkörtől elvárt viselkedést aktív *reset* hatására, az *elsif* ágban (9. sor) pedig azt írjuk le, hogy inaktív *reset* esetén minek kell történnie az órajel felfutóélekor. Ennek megfelelően a VHDL-modell 3–8. soraiban magát az állapotregisztert és minden vezérlőjelet alapállapotba állítunk, a 9. sor *elsif* utasításának törzsében pedig létrehozunk egy *case* szerkezetet (16–24. sor), ami lehetővé teszi annak kifejezését, hogy az állapotgéptől annak különböző állapotaiban más-más működést várunk el. A *case* szerkezetet megelőző „alapértelmezett" értékadások (12–14. sor) a tranziens jellegű, impulzusszerű vezérlőjelek (pl. regiszterek *clock enable*

jelei) alapállapotba állítását fejezik ki. Ezek lehetővé teszik, hogy az adott vezérlőjelen az egyetlen órajel-periódus időtartamú, magas szintű impulzusokat tömören, mindössze egy értékadással hozzuk létre. Az órajel következő felfutóélénél a vezérlőjel nullába húzása automatikusan megtörténik (hacsak a következő vezérlési állapotban újra magas szintre nem húzzuk azt).[43]

VHDL – vezérlőegység

```
1  L_CONTROLLER: process ( clk, rst ) begin
2  | if ( rst = '1' ) then
3  | | state <= wait_for_imem_1;
4  | | IMEM_rstrobe <= '0'; clear_intr_rqst_flag <= '0'; write_strobe <= '0'; read_strobe <= '0';
5  | | ce_IP <= '0'; ce_IR <= '0'; ce_SP <= '0'; ce_PDR <= '0'; ce_PAR <= '0';
6  | | ce_X <= '0'; ce_Y <= '0'; ce_EI <= '0'; ce_Z <= '0'; ce_N <= '0'; we_RF <= '0'; we_STACK <= '0';
7  | | sel_IP <= "00"; sel_IP_add <= "00"; sel_SP_add <= '0'; sel_Y <= "00"; sel_RF <= '0';
8  | | sel_Z <= '0'; sel_N <= '0'; sel_EI <= "00";
9  | elsif ( rising_edge(clk) ) then
10 | |
11 | | -- default values of the pulse-like control signals...
12 | | IMEM_rstrobe <= '0'; clear_intr_rqst_flag <= '0';
13 | | ce_IP <= '0'; ce_IR <= '0'; ce_SP <= '0'; ce_PDR <= '0'; ce_PAR <= '0';
14 | | ce_X <= '0'; ce_Y <= '0'; ce_EI <= '0'; ce_Z <= '0'; ce_N <= '0'; we_RF <= '0'; we_STACK <= '0';
15 | |
16 | | case state is
17 | | |
18 | | | when fetch_1 => [...]
19 | | |
20 | | | [...]
21 | | |
22 | | | when others => null;
23 | | |
24 | | end case;
25 | |
26 | end if;
27 end process;
```

2-34. ábra. *A vezérlőegység VHDL-modelljének vázlata*

Ezt a vázat felhasználva feladatunk most ennek az állapotfüggő viselkedésnek a megfogalmazása, amely a korábban megtervezett műveletvégző egység részegységeit megfelelő vezérléssel látja el a viselkedési modellben megfogalmazott működés megvalósításához. A 2-35. ábra az utasításfelhozás fázis viselkedési modelljét és az ahhoz tartozó vezérlési állapotok VHDL-leírását mutatja be.

Annak érdekében, hogy a vezérlő állapotgép működése könnyebben nyomon követhető legyen, az ábra a műveletvégző egység vonatkozó részleteit is szemlélteti. A viselkedési modellben azt látjuk, hogy az utasításfelhozás fázis egy bejegyzés utasításmemóriából való kiolvasásával kezdődik. Az utasításmemória olvasásához a címet mindig az utasításmutató adja. A *reset* jel hatására az utasításmutatóba a *reset_vector* értéke töltődik be (lásd 2-28. ábra VHDL-modell: 18. sor), tehát a *CPU* indulásakor erről a címről történik meg az első beolvasás. Az utasításmemória egy, a szinkron kézfogásos interfész-specifikáció szerint működő *singleclock_dualport_ram_hswrp*

[43] Ez a megoldás kihasználja a sorrendi utasításblokkokban az egyazon jelre vonatkozó értékadások közötti prioritást; a később kiértékelt utasítás eredménye felülírja a korábbit. Gyakran találkozhatunk ezzel a módszerrel folyamatok (*process* vagy *always*-blokk) segítségével modellezett kombinációs hálózatok esetén, ahol az alapértelmezett értékadások biztosítják, hogy minden kimenet kapjon valamilyen értéket bármely bemeneti kombináció esetén. Kombinációs folyamatoknál ez a latch szintézisének elkerülése érdekében különösen fontos (lásd még DRMSz: 4-43. ábra, 4-45. ábra). A fenti példában azonban ez a módszer csak kényelmi funkciót szolgál; a szinkron *process*-ben meghajtott jelek – szándékunknak megfelelően – D flip-flopokként viselkednek, a latch szintézisének veszélye nem áll fenn (vö. 4-205. ábra lebegőpontos aritmetikai egység vezérlő állapotgépének kimeneti/következőállapot-logikáját leíró *always*-blokk). One-shot típusú állapotgép-leírás esetén ezek az alapértelmezett értékadások akár el is hagyhatók, ilyenkor az adott impulzusszerű vezérlőjel alapállapotba állításáról a magas szintre húzást előidéző állapotot követő állapotban gondoskodnunk kell (lásd pl. 2.3. pont: a *BOOTSTRAP* modul PPP-dekóder állapotgépe, 2-63. ábra jobb oldala).

IP-mag egy példánya, amelynek kiolvasása az 14. ábrán látható időzítési modell szerint lehetséges. Ezt az interfész-specifikációt szem előtt tartva fogalmazzuk meg az egyes vezérlési állapotokban végrehajtandó műveleteket (2-35. ábra VHDL-modell: 1–5. sor), így jutunk a program legelső utasításának kiolvasását szemléltető időzítési diagramhoz (lásd 2-36. ábra).

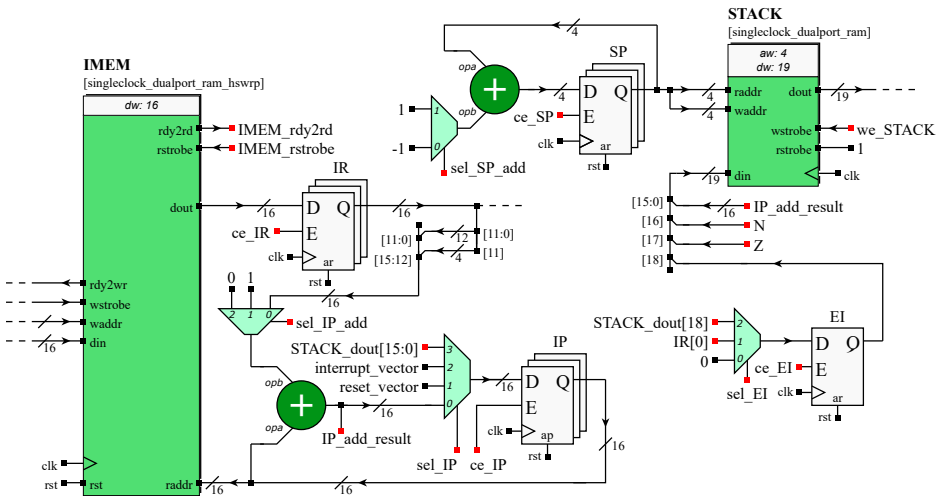
Pseudo – viselkedési modell

```
1  FETCH: request IMEM read
2    wait until IMEM data available
3    if ( firmware reset asserted ) {
4      IP <= reset vector
5      goto FETCH
6    } elsif ( interrupt request detected and EI = 1 ) {
7    | clear interrupt request flag
8    | STACK[SP] <= EI, Z, N, IP
9    | SP++
10   | IP <= interrupt vector
11   | EI <= 0
12   | goto FETCH
13   } else IR <= data from IMEM
```

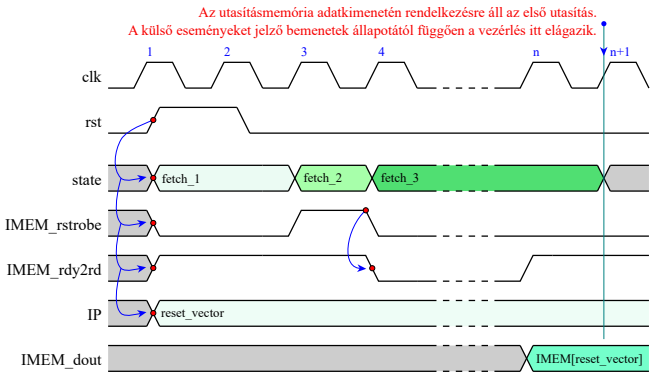
VHDL – vezérlőegység

```
1  when fetch_1 => IMEM_rstrobe <= '1'; state <= fetch_2;
2
3  when fetch_2 => state <= fetch_3;
4
5  when fetch_3 => if ( IMEM_rdy2rd = '1' ) then
6                  | if ( firmware_reset_df1 = '1' ) then
7                  |   sel_IP <= "01"; ce_IP <= '1'; state <= fetch_1;
8                  | elsif ( intr_rqst_flag = '1' and EI = '1' ) then
9                  | | clear_intr_rqst_flag <= '1';
10                 | | sel_SP_add <= '1'; sel_IP_add <= "10"; sel_IP <= "10"; sel_EI <= "00";
11                 | | we_STACK <= '1'; ce_SP <= '1'; ce_IP <= '1'; ce_EI <= '1';
12                 | | state <= fetch_1;
13                 | else
14                 |   ce_IR <= '1'; state <= fetch_4;
15                 | end if;
16                 end if;
17
18 when fetch_4 => state <= decode_1;
```

Áramköri struktúra – műveletvégző egység

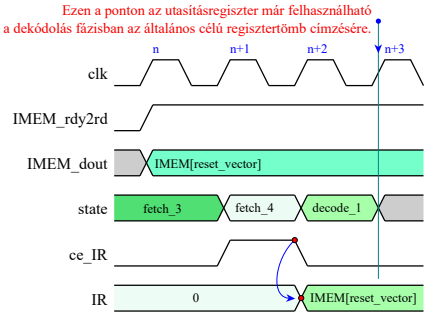


2-35. ábra. *Az utasításfelhozás fázis viselkedési modellje, a hozzá tartozó vezérlési állapotok és a műveletvégző egység vonatkozó részletei*



2-36. ábra. *A program legelső utasításának kiolvasása az utasításmemóriából*

Az órajel (n+1)-edik felfutóélénél a program első utasítása az utasításmemória adatkimenetén *(IMEM_dout)* már rendelkezésre áll. Ennél a felfutóélnél történik meg a külső eseményeket jelző bemenetek vizsgálata, a vezérlés tehát ezen a ponton elágazik. A legegyszerűbb eset, ha sem a program újraindításának szándékát jelző *firmware_reset_df1*, sem a megszakításkérést jelző *intr_rqst_flag* jel[44] nem aktív, ilyenkor az utasításmemória adatkimenetén lévő vektort betöltjük az utasításregiszterbe (2-35. ábra VHDL-modell: 14. sor). Ezt szemlélteti a 2-37. ábra.

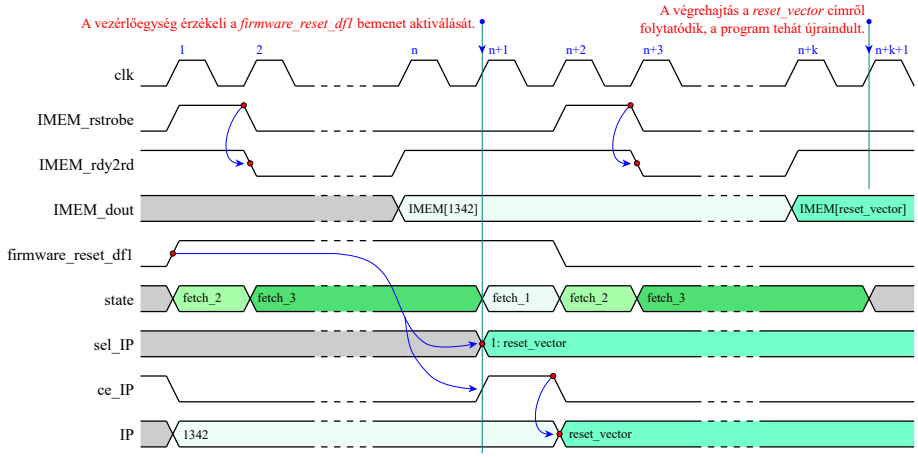


2-37. ábra. *Az utasítás betöltése az utasításregiszterbe*

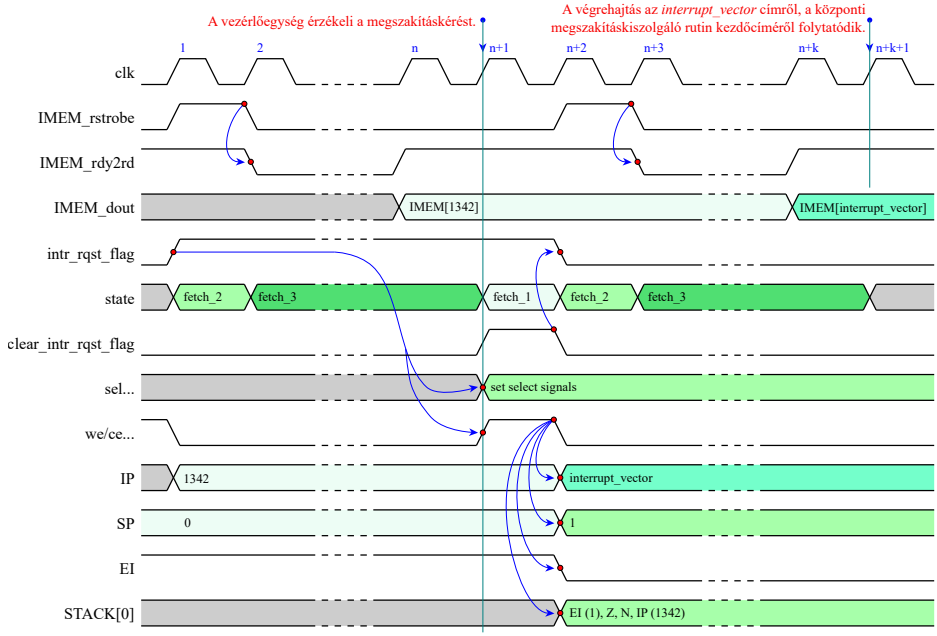
Vizsgáljuk meg azt az esetet, amikor a programvégrehajtást a *firmware_reset_df1* jel aktiválása szakítja meg. Ezt az eseményt a *CPU* vezérlőegysége a *fetch_3* állapotban képes detektálni, amikor az aktuális utasításmutatóval (az ábrán 1342) végrehajtott programtár-olvasási ciklus véget ér (lásd 2-38. ábra).

Ekkor a vezérlőegység a *sel_IP* és a *ce_IP* jelek megfelelő beállításával betölti az utasításmutatóba a *reset_vector* értékét, majd a *fetch_1* állapotba tér vissza, ahol ezen címen ismételt programtár-olvasást kezdeményez. Hasonlóképpen játszódik le a megszakításkérések kiszolgálása is. A vezérlőegység ezt a jelzést is csak az aktuális utasításmemória-olvasást követően érzékeli, a *fetch_3* állapotban (lásd 2-39. ábra).

[44] A *firmware_reset_df1* és az *intr_rqst_flag* vezérlő jelek a processzormag részét képezik, de mivel vezérléshez kapcsolódó feladatot látnak el, ezért nem a műveletvégző egység részeként tárgyaljuk őket. Mindkét flip-flop a *CPU* saját belső órajel-szinkronizáló fokozatainak része, amelyek felépítését a 2.2.4. pontban részletezzük.



2-38. ábra. *A program kényszerített újraindításának folyamata*



2-39. ábra. *A megszakításkérés kiszolgálásának folyamata*

A megszakításkérés észlelésekor a vezérlőegység törli a megszakításkérés beérkezésére utaló állapotjelzőt *(intr_rqst_flag)* és beállítja az *EI*, az *SP* és az *IP* regisztereket meghajtó adatutak multiplexereinek kiválasztó jeleit oly módon, hogy – a viselkedési modellben leírtaknak megfelelően – az *EI* regiszterbe nulla íródjon, az *SP* regiszter értéke eggyel növekedjen, az *IP* regiszterbe pedig a központi megszakításkiszolgáló rutin kezdőcíme *(interrupt_vector)* kerüljön. Ezután kerül sor a szóban forgó regiszterek értékének frissítésére azok *ce* vezérlőjelén előállított pozitív impulzus

segítségével. Ugyanezen vezérlési ciklusban a *we_STACK* jelen előállított impulzus a *CPU* aktuális állapotát leíró regiszterek *(EI, Z, N, IP)* régi értékét elmenti az állapotveremtárba. A 2-38. és a 2-39. ábrákon láthatjuk, hogy akár a program újraindításáról, akár egy megszakítás kiszolgálásáról van szó, a *CPU* a *fetch_3* állapotból folytatja a végrehajtást, ahol az utasításmemória adatkimenetén rendelkezésre áll a következő végrehajtandó utasítás. Ha ekkor sem a *firmware_reset_df1*, sem az *intr_rqst_flag* jel nem aktív, akkor a következő lépés a programtárból felhozott utasítás betöltése az utasításregiszterbe, ami a 2-37. ábrán már bemutatott módon megy végbe a *fetch_3-fetch_4-decode_1* állapotsorozaton keresztül. Így tehát a végrehajtás elérkezik a dekódolás fázishoz.

A dekódolás fázis vezérlési állapotainak működése a 2-40. ábrán követhető nyomon. Ebben a fázisban az *X* és *Y* regiszterek frissítése történik meg. Az *X* regiszterbe a végrehajtott utasítástól függetlenül mindig az általános célú regisztertömb egy bejegyzését töltjük, az *Y* regiszterbe pedig a regisztertömbből származó adatot vagy az utasításban megadott konstanst, az éppen végrehajtott utasítás típusától függően.

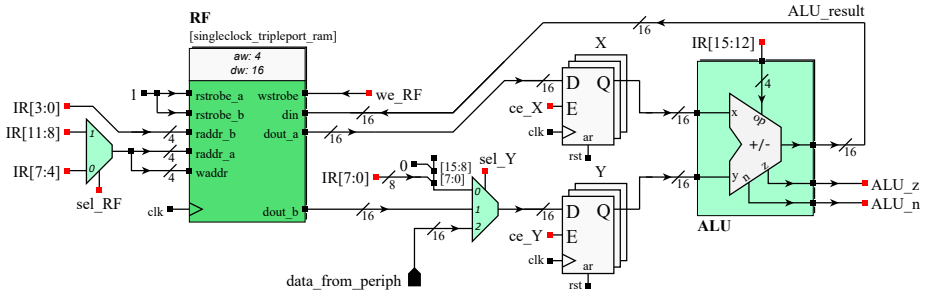
Pseudo – viselkedési modell

```
1  DECODE: X <= RF[IR[7:4]]
2
3          if ( IR[15:12] = i2rfl )
4            Y <= unsigned(IR[7:0])
5          else Y <= RF[IR[3:0]]
```

VHDL – vezérlőegység

```
1  when decode_1 => sel_RF <= '0';
2
3                   if ( IR(15 downto 12) = i_i2rfl ) then
4                     sel_Y <= "00";
5                   else sel_Y <= "01"; end if;
6
7                   state <= decode_2;
8
9  when decode_2 => ce_X <= '1'; ce_Y <= '1';
10                  state <= periph_access_0;
```

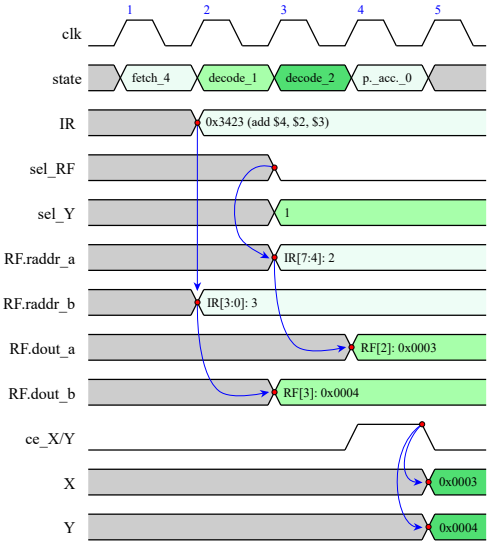
Áramköri struktúra – műveletvégző egység



2-40. ábra. *A dekódolás fázis viselkedési modellje, a hozzá tartozó vezérlési állapotok és a műveletvégző egység vonatkozó részletei*

A két operandusregiszter frissítésének folyamatát a 2-41. ábrán látható időzítési diagram szemlélteti egy összeadás utasítás végrehajtásának példáján (5-12. táblázat első példa). A vezérlőegység tervezése során itt is figyelembe kell vennünk a vezérelt részegységek interfészeinek jellegzetességeit. Az általános célú regisztertömb egy *singleclock_tripleport_ram* IP-mag példánya (részletesen lásd 4.2.1. pont), amelynek kiolvasó interfésze a 4-13. ábrán látható interfész-specifikáció szerint működik. Az almodul *rstrobe* bemeneteit folyamatosan logikai magas szinten tartjuk, aminek következtében az adott címbemenet megváltozásakor a megcímzett rekesz tartalma a megfelelő adatkimeneten az órajel következő felfutóélénél áll elő.

A 2-41. ábrán azt látjuk, hogy az utasításregiszter értéke az órajel második felfutóélénél megváltozik. Az utasításregiszter [3:0] mezői közvetlenül meghajtják a regisztertömb *raddr_b* bemenetét,



2-41. ábra. *Az X és Y regiszterek frissítésének folyamata a dekódolás fázisban*

aminek eredményeképpen a 3. rekesz tartalma (0x0004) az órajel harmadik felfutóélénél jelenik meg a *dout_b* kimeneten. Az utasításregiszter [7:4] bitjei a regisztertömb *raddr_a* bemenetére egy multiplexeren keresztül, a *sel_RF* kiválasztójel megváltozásának pillanatában kerülnek rá,[45] ennek megfelelően az e bitek által címzett rekesz tartalma (0x0003) csak az órajel negyedik felfutóélénél íródik a regisztertömb *dout_a* kimenetére. Ugyanebben a pillanatban már kezdeményezhetjük az *X* és *Y* regiszterek értékének frissítését azok *ce* bemenetének magas szintre állításával. Ezek az impulzusok az órajel ötödik felfutóélénél töltik be a regisztertömb megfelelő rekeszeinek tartalmát az operandusregiszterekbe, amivel a dekódolás fázis véget ér.

Miután az operandusregiszterek a dekódolás fázisban frissültek, kezdetét veszi a perifériaműveletek fázisa, amelynek során az MMIO-tranzakciókhoz tartozó *PAR* és *PDR* regiszterek betöltése és a periféria-áramkörökkel való adatcsere zajlik le. E folyamat viselkedési modelljét, vezérlési állapotainak VHDL-leírását és a vezérelt műveletvégző elemeket a 2-42., a folyamat időzítését pedig a 2-43. ábra mutatja be egy *load* utasítás példáján keresztül.

A VHDL-modell 1–3. sorában láthatjuk, hogy a perifériaműveletek csak akkor zajlanak le, ha *load* vagy *store* utasítást hajtunk végre, máskülönben az állapotgép rögtön a *stack* állapotra ugrik. A *store* és a *load* utasítások végrehajtása a perifériaműveletek fázisban nagyon hasonló; az általános célú regisztertömbből – annak *addr_a* és *waddr* bemenetét meghajtó multiplexer kiválasztó jelének beállításával – kiolvassuk a *store* utasítással a megcímzett periféria-áramkörnek kiküldendő adatot (lásd 2-43. ábra: 2–3. órajel-felfutóél). Ezt akkor is megtesszük, ha *load* utasításról van szó, így minimalizálva a vezérlési állapotokban megfogalmazott feltételvizsgálatok számát, közvetetten az e leírásból szintetizált áramkörök méretét (vö. erőforrás-csökkentés a regularitás alapelvén keresztül). A *PAR* regiszterbe betöltendő cím az *X* és *Y* regiszterek összegeként áll elő, amelyeknek a frissítése már a dekódolás fázis legvégén megtörtént (lásd 2-43. ábra: második órajel-felfutóél).

[45] A multiplexer az aritmetikai/logikai operátorokhoz hasonlóan egy kombinációs hálózatnak tekinthető aszinkron részegység, amelynek késleltetését a modellezés ezen szakaszában nullának tekintjük. Bármely bemenetének – legyen az adat- *(dinX)* vagy kiválasztó *(sel)* bemenet – megváltozása a kimenetének azonnali megváltozását idézi elő.

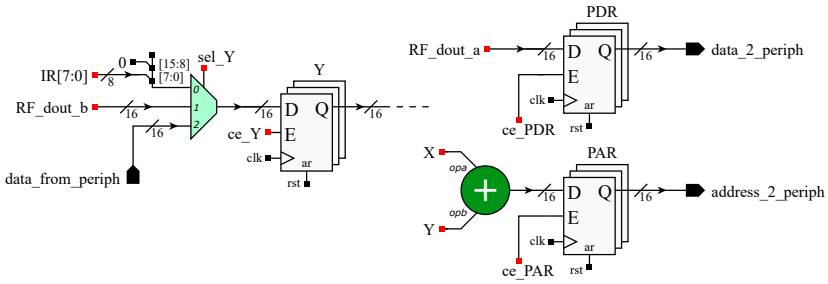
Pseudo – viselkedési modell

```
PERIPH_ACCESS: if ( IR[15:12] != store and IR[15:12] != load ) goto STACK
               PDR <= RF[IR[11:8]]
               PAR <= X + Y
               if ( IR[15:12] = store ) request MMIO write
               if ( IR[15:12] = load ) request MMIO read
               wait until MMIO transaction is done
               if ( IR[15:12] = load ) Y <= data from peripheral
```

VHDL – vezérlőegység

```
when periph_access_0 => if ( IR(15 downto 12) /= i_store and ir(15 downto 12) /= i_load ) then
                          state <= stack;
                        else sel_RF <= '1'; state <= periph_access_1; end if;

when periph_access_1 => ce_PDR <= '1'; ce_PAR <= '1';
                        state <= periph_access_2;

when periph_access_2 => if ( IR(15 downto 12) = i_store ) then write_strobe <= '1'; end if;
                        if ( IR(15 downto 12) = i_load ) then read_strobe <= '1'; end if;
                        state <= periph_access_3;

when periph_access_3 => if ( periph_ready_df1 = '0' ) then
                          write_strobe <= '0'; read_strobe <= '0';
                          state <= periph_access_4;
                        end if;

when periph_access_4 => if ( periph_ready_df1 = '1' ) then
                          if ( IR(15 downto 12) = i_load ) then sel_Y <= "10"; ce_Y <= '1'; end if;
                          state <= stack;
                        end if;
```

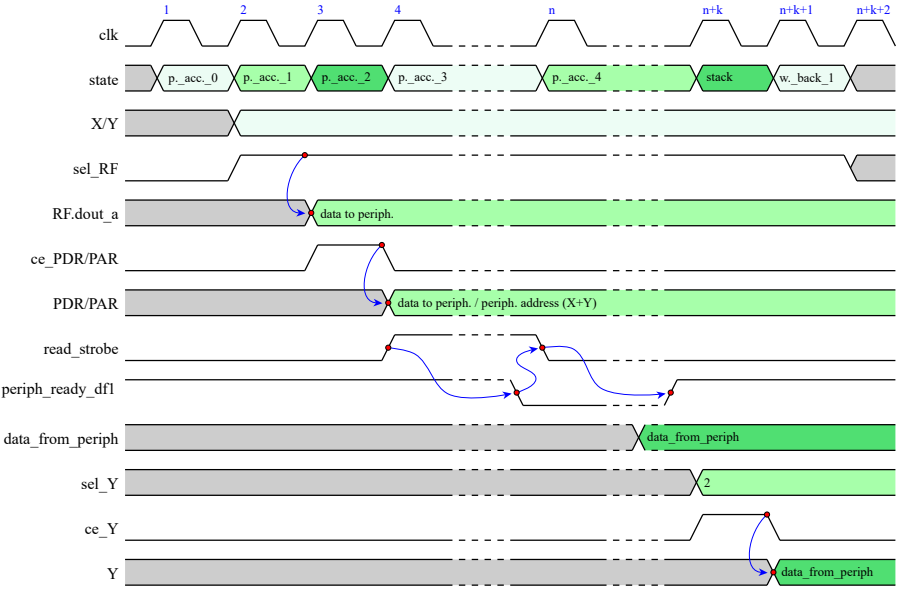
Áramköri struktúra – műveletvégző egység



2-42. ábra. *A perifériaműveletek fázis viselkedési modellje, a hozzá tartozó vezérlési állapotok és a műveletvégző egység vonatkozó részletei*

A *ce_PDR* és *ce_PAR* jeleken előállított pozitív impulzus segítségével frissítjük a *PDR* és *PAR* segédregisztereket, előkészítve magát az MMIO-tranzakciót.

A *PAR* és *PDR* regiszterek közvetlenül meghajtják a *CPU address_2_periph és data_2_periph* kimeneteit, értékük frissítése pillanatában tehát rögtön megkezdhető az MMIO-tranzakció az utasításnak megfelelő *strobe* jelen előállított felfutóéllel (vö. MMIO-tranzakciók időzítési modelljei: 1-8. és 1-9. ábra). Ezután a vezérlőegység két egymást követő várakozási ciklust valósít meg. A *periph_access_3* állapotban arra vár, hogy a megcímzett periféria-áramkör nullába húzza az MMIO-busz *ready* jelét, jelezve, hogy a *CPU* által küldött kérés kiszolgálása megkezdődött, majd a *periph_access_4* állapotban kivárja, míg a *ready* jel ismét magas szintre áll, amivel a periféria-áramkör a művelet eredményes végrehajtását jelzi. Ha *store* utasításról van szó, akkor a perifériaműveletek fázisa ezzel véget is ért, ha azonban *load* utasítást hajtunk éppen végre, akkor a periféria-áramkörből beolvasott adat elmentése még hátravan. A *CPU data_from_periph* adatbemenetén ez az adat immár rendelkezésre áll. A regisztertömb



2-43. ábra. *A perifériaműveletek fázisának időzítése egy load utasítás végrehajtása során*

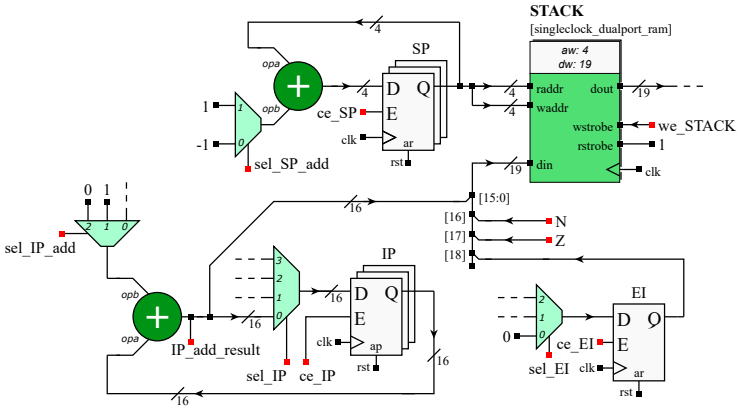
Pseudo – viselkedési modell

```
1  STACK: if ( IR[15:12] = call ) {
2           | EI <= 0
3           | STACK[SP] <= EI, Z, N, IP+1
4           | SP++
5         }
6         if ( IR[15:12] = ret ) SP--
```

VHDL – vezérlőegység

```
1  when stack => if ( IR(15 downto 12) = i_call ) then
2                  | sel_EI <= "00"; ce_EI <= '1';
3                  | sel_IP_add <= "01";
4                  | we_STACK <= '1';
5                  | sel_SP_add <= '1'; ce_SP <= '1';
6                end if;
7
8                if ( IR(15 downto 12) = i_ret ) then
9                  sel_SP_add <= '0'; ce_SP <= '1'; end if;
10
11               state <= write_back_1;
```

Áramköri struktúra – műveletvégző egység



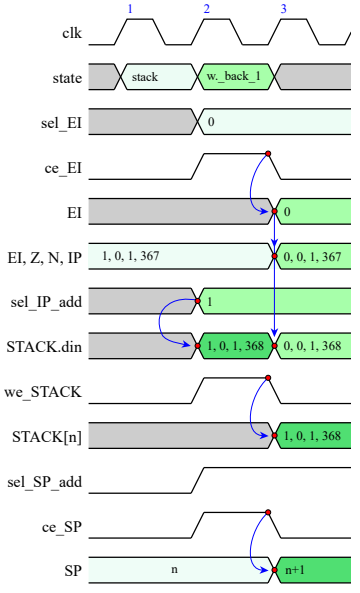
2-44. ábra. *Az állapotverem-műveletek fázis viselkedési modellje, a hozzá tartozó vezérlési állapotok és a műveletvégző egység vonatkozó részletei*

írására majd csak a visszaírás fázisban kerítünk sort, a periféria-áramkörtől beolvasott adatot egyelőre csak az *Y* regiszterbe mentjük el [lásd 2-43. ábra: (n+k)-adik és (n+k+1)-edik órajel-felfutóél].

A periféria-áramkörökkel való adatcserét levezénylő vezérlési állapotok után az állapotveremtárral kapcsolatos utasítások, vagyis a *call* szubrutinhívó utasítás és a szubrutinokból való visszatérést lehetővé tévő *ret* utasítás végrehajtásának előkészítése következik.

A 2-44. ábra VHDL-kódrészletén láthatjuk, hogy ez a fázis mindössze egyetlen órajelciklus alatt lezajlik. A *ret* utasítás esetén e fázis feladata csupán annyi, hogy az *SP* regiszter értékét eggyel csökkentse. A *call* utasítás esetén azonban a helyzet némileg bonyolultabb, mert ebben az egy órajelciklusban több műveletet hajtunk végre egymással párhuzamosan. A 2-45. ábrán egy, a programtár 367. címén lévő *call* utasítás végrehajtásának részletét látjuk. Az első feladat az *EI* regiszter értékének nullába állítása a viselkedési modell második sorának megfelelően. Ez egy egyszerű regiszterírás, amelyre korábban több példát is láthattunk; először beállítjuk a regiszter bemenetét hajtó kombinációs hálózat multiplexereinek kiválasztó jeleit – ebben az esetben a *sel_EI* jelet –, majd egy egyetlen órajel-periódusnyi pozitív impulzust állítunk elő a regiszter *ce* bemenetén. Ezzel párhuzamosan a *CPU* aktuális állapotát, vagyis az *EI, Z* és *N* állapotjelzők, valamint az utasításmutató eggyel megnövelt értékét az állapotveremtárba mentjük. Ennek érdekében az utasításmutatóhoz rendelt összeadó bemeneti multiplexerének megfelelő vezérlésével *(sel_IP_add)* elérjük, hogy az összeadó kimenetén – egyúttal a veremtár adatbemenetén – előálljon az *IP+1* érték, amit az állapotjelzőkkel összefűzve a *we_STACK* jelen előállított pozitív impulzussal beírunk a veremtárba. A harmadik végrehajtandó feladat az *SP* regiszter értékének növelése, ami ugyancsak egy egyszerű, a *sel_SP_add* és a *ce_SP* jelek segítségével végrehajtott regiszterírás.

Az utolsó végrehajtási fázis a visszaírás, amelynek viselkedési leírását, vezérlési állapotainak VHDL-modelljét, valamint a hozzá tartozó műveletvégző elemek RTL kapcsolási sémáját a 2-46. ábra szemlélteti. Ebben a fázisban érvényesül legkevésbé az utasítások regularitása, a végrehajtott



2-45. ábra. *Az állapotverem-műveletek fázis időzítése egy call utasítás végrehajtása során*

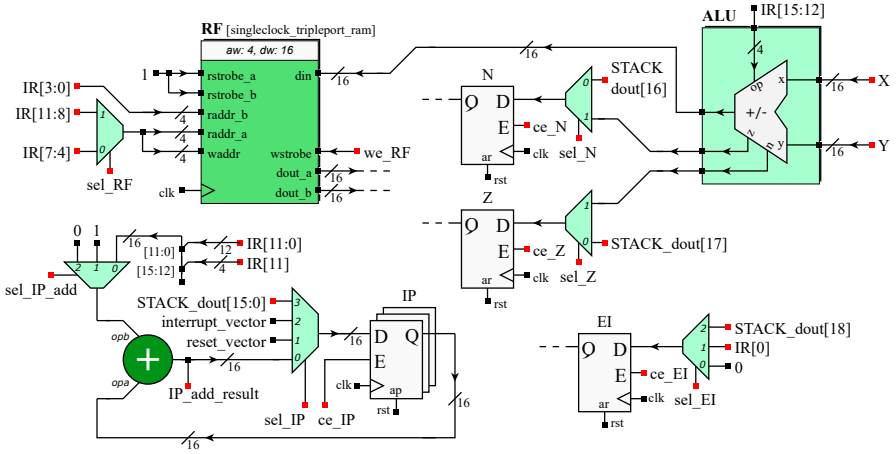
Pseudo – viselkedési modell

```
WRITE_BACK: ALU.op <= IR[15:12]; ALU.x <= X; ALU.y <= Y

            if ( IR[15:12] = i2rfl, load, add, sub, and, or, xor, shl, shr )
              RF[IR[11:8]], Z, N <= results from ALU

            if ((IR[15:12] = bnz and !Z) or
                (IR[15:12] = bnn and !N) or
                (IR[15:12] = jmp) or
                (IR[15:12] = call)
            ) IP <= IP + signed(IR[11:0])
            else IP++

            if ( IR[15:12] = ret ) IP, EI, Z, N <= STACK[SP]
            if ( IR[15:12] = ei ) EI <= IR[0]
```

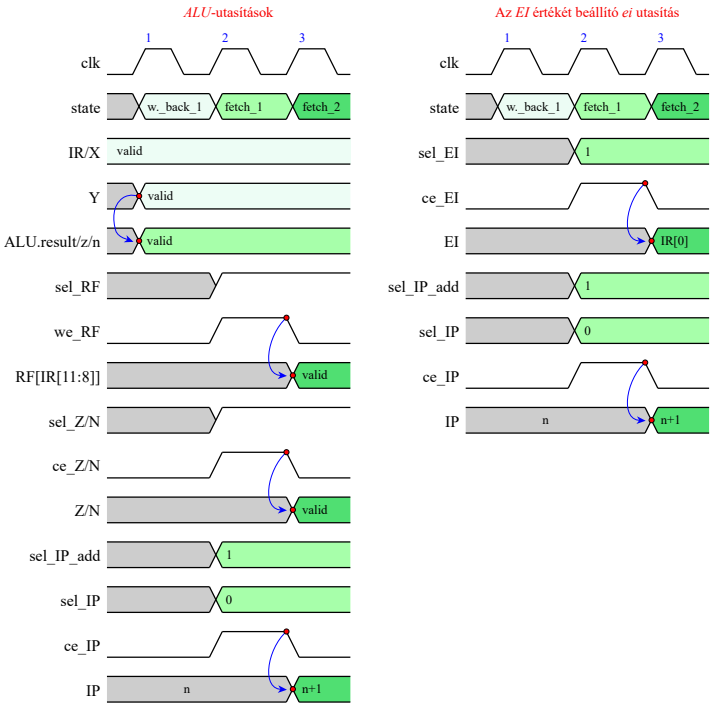
VHDL – vezérlőegység

```
when write_back_1 => if ( IR(15 downto 12) = i_i2rfl or IR(15 downto 12) = i_load or
                          IR(15 downto 12) = i_add or IR(15 downto 12) = i_sub or
                          IR(15 downto 12) = i_and or IR(15 downto 12) = i_or or
                          IR(15 downto 12) = i_xor or
                          IR(15 downto 12) = i_shl or IR(15 downto 12) = i_shr ) then
                       sel_RF <= '1'; we_RF <= '1';
                       sel_Z <= '1'; ce_Z <= '1';
                       sel_N <= '1'; ce_N <= '1';
                     end if;

                     if ( (IR(15 downto 12) = i_bnz and Z = '0') or
                          (IR(15 downto 12) = i_bnn and N = '0') or
                          (IR(15 downto 12) = i_jmp) or (IR(15 downto 12) = i_call) ) then
                       sel_IP <= "00"; sel_IP_add <= "00"; ce_IP <= '1';
                     else sel_IP <= "00"; sel_IP_add <= "01"; ce_IP <= '1';
                     end if;

                     if ( IR(15 downto 12) = i_ei ) then sel_EI <= "01"; ce_EI <= '1'; end if;

                     if ( IR(15 downto 12) = i_ret ) then state <= write_back_2;
                     else state <= fetch_1;
                     end if;

when write_back_2 => sel_IP <= "11"; ce_IP <= '1'; sel_Z <= '0'; ce_Z <= '1'; sel_N <= '0'; ce_N <= '1';
                     sel_EI <= "10"; ce_EI <= '1';
                     state <= fetch_1;
```

Áramköri struktúra – műveletvégző egység



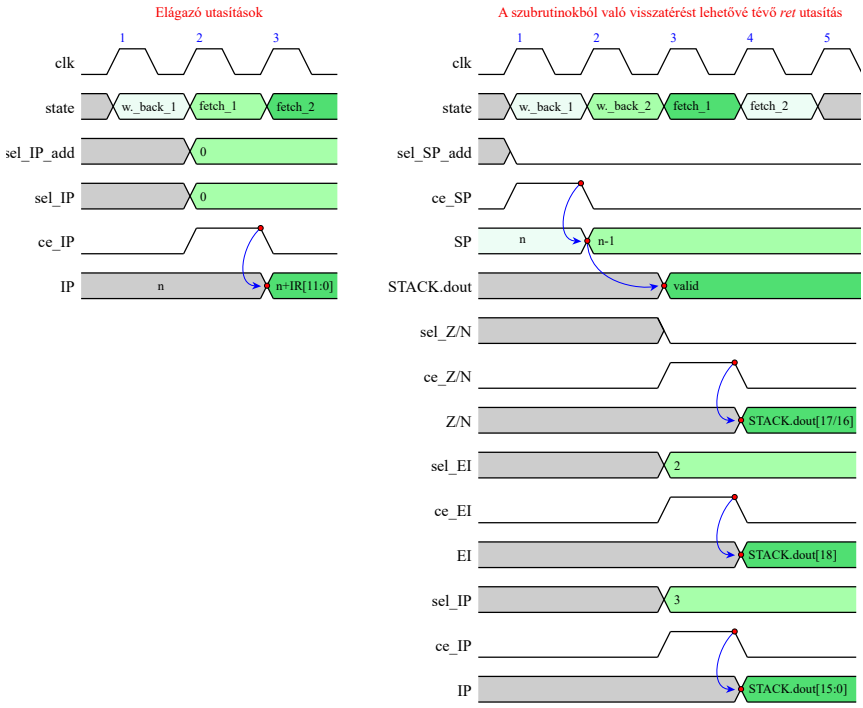
2-46. ábra. *A visszaírás fázis viselkedési modellje, a hozzá tartozó vezérlési állapotok és a műveletvégző egység vonatkozó részletei*



2-47. ábra. *A visszaírás fázis időzítése az ALU-utasítások és az ei utasítás végrehajtása során*

konkrét utasítástípustól függően más-más események zajlanak le. A 2-47. ábra bal oldalán a visszaírás fázis időzítésének azt a változatát látjuk, amikor az aritmetikai/logikai egység által előállított eredményvektor visszaírása történik meg az általános célú regisztertömbbe. Az utasításregiszter és az *X* operandusregiszter az órajel első felfutóélénél már mindenképpen stabil állapotban van, ezek betöltése a korábbi fázisokban már megtörtént; az utasításregiszter értéke az utasításfelhozás fázisban, az *X* regiszteré pedig a dekódolás során frissült. Az *Y* operandusregiszter legkésőbbi frissítésének időpontja a végrehajtott utasítás típusától függ; rendszerint a dekódolás során, az *X* regiszterrel együtt kap értéket, de a *load* utasítás végrehajtásakor az éppen megcímzett periféria-áramkörtől kapott adatot is ebbe a regiszterbe töltjük be a perifériaműveletek fázis legutolsó lépésében (a 2-47. ábra bal oldalán lévő hullámforma szerinti első órajel-felfutóélnél). Az *ALU*-t kombinációs hálózatként írtuk le, ezért annak kimenete az *Y* regiszter értékével együtt változik. E változás után az *ALU* által szolgáltatott eredményeket az órajel második és harmadik felfutóélénél a *we_RF*, *ce_Z* és *ce_N* jeleken előállított pozitív impulzusokkal az általános célú regisztertömbbe, valamint a *Z* és *N* állapotjelzőkbe töltjük.[46]

[46] Ha jobban szemügyre vesszük az 2-47. ábra bal oldali hullámformáját, észrevehetjük, hogy az általános célú regiszter-tömb, valamint a *Z* és *N* állapotjelzők frissítése valójában egy órajelciklussal korábban is megtörténhetne, hiszen az *ALU* kimenetén az eredmények már az órajel második felfutóélénél rendelkezésre állnak. Ezt az egy órajelciklusnyi tartalékot az esetleges későbbi időzítési optimalizáció megkönnyítése céljából hagyjuk a rendszerben. Ha a fizikai szintézis után a legnagyobb késleltetésű lokális adatút (kritikus út, lásd DRMSz: 6.2.1. alfejezet) az *ALU*-ban alakul ki, akkor módunkban áll olyan modellt készíteni az *ALU*-hoz, ami nem egy egyszerű kombinációs hálózatként viselkedik, hanem szinkron működésű áramkörként egy órajelciklus késleltetéssel állítja elő eredményeit. A szinkron *ALU* megfontolt tervezésével, az azt alkotó kombinációs részhálózatok és tároló elemek megfelelő elhelyezésével (register balancing) kisebb késleltetésű kritikus út, ezáltal magasabb maximális órajel-frekvencia érhető el, mint a tisztán kombinációs *ALU*-val.



2-48. ábra. *A visszaírás fázis időzítése az elágazó utasítások és a ret utasítás végrehajtása során*

Ezzel párhuzamosan az utasításmutató bemenetén az azt meghajtó kombinációs hálózat multiplexereinek kiválasztó jeleit *(sel_IP_add, sel_IP)* megfelelően meghajtva előállítjuk a következő utasítás programtárbeli címét, amit a *ce_IP* jel segítségével be is töltünk az utasításmutatóba, előkészítve ez által a következő gépi utasítás felhozását.

Az *ei* utasítás végrehajtásának időzítését a 2-47. ábra jobb oldalán láthatjuk. Ennek az utasításnak a végrehajtásakor csupán annyi történik, hogy az utasításregiszter legalsó bitjének értékét betöltjük az *EI* regiszterbe. Természetesen nem maradhat el az utasításmutató értékének frissítése sem, ami itt is pontosan ugyanúgy történik, ahogy az *ALU*-műveletek esetén láthattuk.

A 2-48. ábra bal oldali hullámformája mutatja be az elágazó utasítások végrehajtásának folyamatát. Ilyenkor a visszaírás egyetlen szerepe, hogy az utasításmutató értékét előkészítse a következő utasítás felhozásához. A programtárban való ugrás feltételeinek fennállása esetén az utasításmutatót az utasításregiszterben megadott konstans értékével kell növelni, ami a többi utasításnál látható inkrementálástól csak abban különbözik, hogy az utasításmutató dedikált összeadójának *opb* operandusbemenetét hajtó multiplexer kiválasztó bemenetét ezúttal 1 helyett 0 értékűre állítjuk.

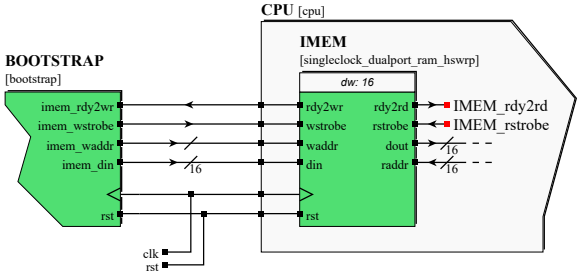
A 2-48. ábra jobb oldalán a szubrutinokból való visszatérést lehetővé tévő *ret* utasítás visszaírás fázisának ütemezését láthatjuk. Maga a visszaírás ebben az esetben az *EI, Z* és *N* állapotjelzőknek, valamint az utasításmutatónak az állapotveremtárból való visszaállítását jelenti. Ezt a lépést már az állapotverem-műveletek fázis előkészítette az *SP* regiszter dekrementálásával (lásd 2-44. ábra, viselkedési modell: 6. sor, VHDL-leírás: 8–9. sor). Mivel az állapotveremtárat a szinkron működésű *singleclock_dualport_ram* IP-maggal valósítottuk meg, ezért annak kimenetén a visszaállítandó adatvektor csak az órajel harmadik felfutóélénél áll elő. Ekkor a már megszokott módon

kezdeményezhetjük az állapotjelzők és az utasításmutató értékének frissítését a *ce_Z, ce_N, ce_EI* és *ce_IP* jeleken előállított pozitív impulzusokkal.

2.2.4. A CPU órajel-szinkronizáló fokozatai

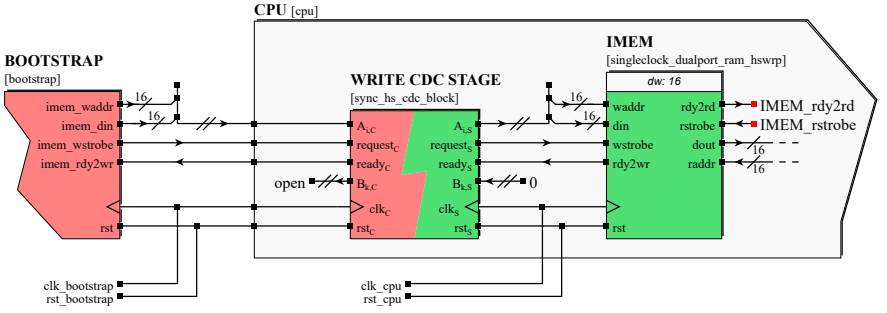
A rendszertervben megfogalmazott tervezési elveknek megfelelően a *CPU* RTL-modelljét úgy készítjük el, hogy ha szükséges, akkor egy önálló, a környezetében lévő egyéb almodulokhoz képest aszinkron órajeltartományt képezhessen. Ez gyakorlatilag azt jelenti, hogy a *CPU* minden külső interfészéhez terveznünk kell egy-egy órajel-szinkronizáló fokozatot, amelyek pontos felépítése az interfész működésének részleteitől függ.

Az utasításmemória beíró interfészének szinkronizálása ◆ A műveletvégző egység tervezése során a *CPU* belső utasításmemóriájának beíró interfészével nem foglalkoztunk, az ahhoz tartozó interfészjeleket szabadon hagytuk (lásd 2-28. ábra). Ez az interfész valójában nem más, mint a *singleclock_dualport_ram_hswrp* IP-mag (lásd 4.2.3. pont) beíró interfésze, amelynek jelei egyúttal a *CPU* külső interfészjelei is. Ezek segítségével a *BOOTSTRAP* modulnak lehetősége van az utasításokat elhelyezni a *CPU* belső utasításmemóriájában (lásd 2-49. ábra).



2-49. ábra. *A program betöltését végző BOOTSTRAP modul és a CPU kapcsolata*

Ez a megoldás csak akkor működőképes, ha a *CPU* és a *BOOTSTRAP* modul egyazon órajelforrásról jár, ami egyrészt ellentmond a rendszertervben lefektetett alapelveknek, másrészt nem is praktikus, hiszen a *CPU*-t célszerű minél magasabb frekvenciájú órajellel meghajtani a maximális műveletvégzési sebesség eléréséhez, a *BOOTSTRAP* modul viszont alapvetően nem sebességkritikus szerepeket ellátó almodul, amelynek órajel-frekvenciáját ebből adódóan célszerű alacsonyan tartani

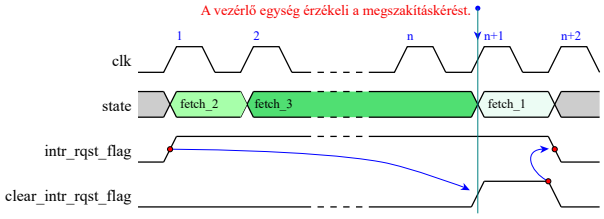


2-50. ábra. *A BOOTSTRAP modul és a CPU közötti órajel-szinkronizáló fokozat*

a fogyasztás mérséklése érdekében. A két almodul közé tehát egy órajel-szinkronizáló fokozatot helyezünk el. Mivel a *singleclock_dualport_ram_hswrp* modul az 1.2.1.1. pontban ismertetett szinkron kézfogásos interfész-specifikáció szerint működik, ezért használhatjuk a 2-3. ábrán látható általános vektorszinkronizáló fokozatot a 2-50. ábrán látható módon. A szinkronizáló fokozat általánosított A_j és B_k vektorai ebben az esetben az utasításmemória adat- és címbuszait rejtik.

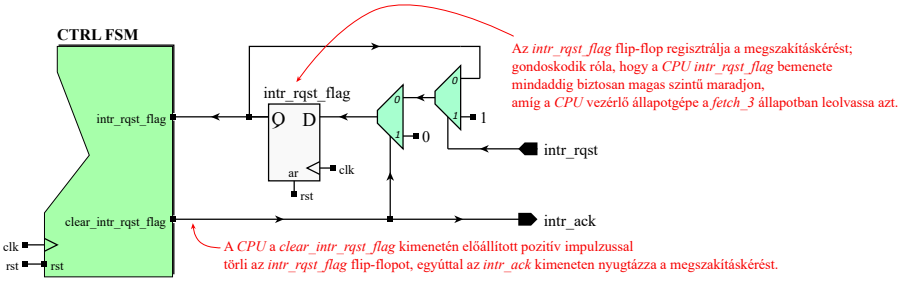
A *BOOTSTRAP* modul és a *CPU* órajel-szinkronizálása megoldható úgy is, hogy az utasításmemóriát eleve olyan IP-maggal valósítjuk meg, ami önmagában is két külön órajelbemenetet biztosít a beíró és a kiolvasó interfész számára (lád 4.2.5. pont). Ebben az esetben az órajel-szinkronizáció valójában a memóriamodul belsejében valósul meg, a *BOOTSTRAP* modul és az utasításmemória pedig közvetlenül összekapcsolható.[47]

Megszakítási rendszer szinkronizálása ◆ A megszakításkérések mechanizmusának tárgyalásakor abból indultunk ki, hogy megszakításkéréskor egy, a vezérlőegységen kívüli állapotjelző, az *intr_rqst_flag* magas szintet vesz fel, amire a *CPU* vezérlőegysége a *fetch_3* állapotban reagál (lásd 2-51. ábra).



2-51. ábra. *A megszakítások érvényre jutásának mechanizmusa*

Ilyenkor a *CPU* egy, a *clear_intr_rqst_flag* jelen előállított pozitív impulzussal törli az *intr_rqst_flag* értékét, ezzel nyugtázva a megszakításkérést. Ennek a viselkedésnek a megvalósításához elvileg elegendő lenne csupán magával az *intr_rqst_flag*-gel, tehát egyetlen flip-floppal kiegészíteni a *CPU* eddig megtervezett modelljét (lásd 2-52. ábra).



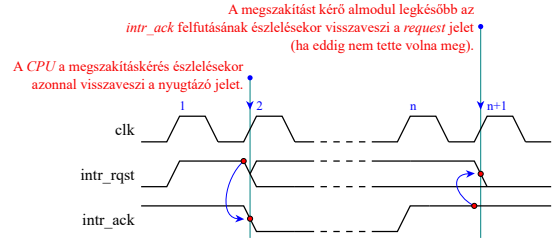
2-52. ábra. *A megszakításkérést regisztráló állapotjelző*

A 2-52. ábrán látható kapcsolás viselkedése funkcionálisan megfelelő ugyan – hallgatólagosan feltételezve, hogy a megszakításkérés forrása egyetlen pozitív impulzussal jelzi megszakítási igényét –,

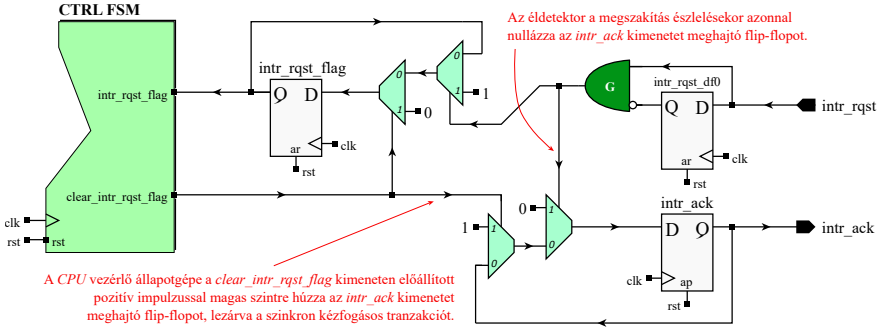
[47] A *CPU* esetén nem így járunk el, mert ezzel azt a követelményt támasztanánk az utasításmemóriát megvalósító IP-maggal szemben, hogy képes legyen két külön órajelet fogadni. Ez blokk-RAM alapú megvalósítás esetén ugyan nem probléma, de ha az IP-n belül valójában egy összetettebb működésű (pl. EDAC) memória vagy egy külső ROM-memóriát illesztő interfészadapter rejlik, akkor a két aszinkron órajelforrásról való működés képessége nem elvárható.

de a regularitás jegyében mégis egy másik megoldást választunk. Megszakításkéréskor egy külső almodul „megkéri” a *CPU*-t, hogy a műveletvégzést szakítsa meg, és folytassa az utasítások beolvasását és végrehajtását a központi megszakításkiszolgáló rutin kezdőcímétől. Fogalmazhatunk úgy is, hogy a megszakítást kérő külső almodul mint kliens, a *CPU*-nak mint kiszolgálónak a programvégrehajtás megszakítására vonatkozó szolgáltatását veszi igénybe. Ha a teljes funkcióbővítő rendszert tekintve egységességre törekszünk, akkor célszerű a *CPU* megszakításkérő interfészén is alkalmazni az 1.2.1.1. pontban tárgyalt, kifejezetten erre a kliens-kiszolgáló viszonyra tervezett szinkron kézfogásos interfész-specifikációt.[48] A 2-53. ábra szemlélteti, hogy ezek alapján hogyan kellene működnie a megszakításkéréshez tartozó jeleknek. Figyeljük meg, hogy a 2-52. ábra egyszerű kapcsolása az *intr_ack* jelen egészen más viselkedést valósít meg, mint amit a szinkron kézfogásos interfész-specifikáció előír; egyrészt az *intr_rqst* jel felfutásának hatására nem nullázza azonnal – az órajel következő felfutóélénél – a *clear_intr_rqst_flag* jellel azonos nyugtázó jelet, mert az állapotgép csak a *fetch_3* állapotba visszatérve foglalkozik a megszakítást jelző flip-flop állapotával, másrészt a megszakítás nyugtázását egyetlen pozitív impulzussal jelzi, ahelyett, hogy a kimenetet logikai magas szinten tartaná.

A 2-53. ábrán látható működést a 2-54. ábrán látható áramköri sémával érhetjük el. Ebben a megoldásban a *CPU* állapotgépe által generált *clear_intr_rqst_flag* jelet nem közvetlenül használjuk a megszakítás nyugtázására, hanem erre a célra bevezetünk egy újabb állapotjelzőt *(intr_ack)*, amit



2-53. ábra. *A szinkron kézfogásos interfész-specifikáció szerint működő megszakítási rendszer viselkedése*



2-54. ábra. *A szinkron kézfogásos interfész-specifikációt megvalósító megszakítási rendszer szerkezete*

[48] A szinkron kézfogásos interfész-specifikációnak itt azt a speciális esetét látjuk, amikor az A_i és B_k vektorok száma egyaránt nulla, tehát a két almodul közötti tranzakcióban egyik fél sem küld adatot a másiknak, csupán egy esemény jelzéséről és az arra adott válaszról van szó.

a *clear_intr_rqst_flag* impulzus állít logikai magas szintre. A külső almodultól érkező *intr_rqst* jelet az *intr_rqst_df0* flip-flop és a *G AND* kapu által képezett felfutóél-detektorral fogadjuk, amelynek kimenetét az *intr_rqst_flag* flip-flop magas szintre húzására, valamint a nyugtázó állapotjelző törlésére használunk. Az újonnan bevezetett éldetektor garantálja a szinkron kézfogásos interfész azon előírásának való megfelelést, hogy a külső almodul a megszakítási igényét egyetlen pozitív impulzussal vagy a megszakításkérő vonal magas szinten tartásával is jelezheti. A nyugtázó vonalra beépített állapotjelző pedig – ugyancsak a szinkron kézfogásos interfész-specifikációnak megfelelően – biztosítja, hogy a külső almodulnak szánt nyugtázó vonal a megszakításkérés észlelésekor, rögtön az órajel következő felfutóélénél nullázódjon.

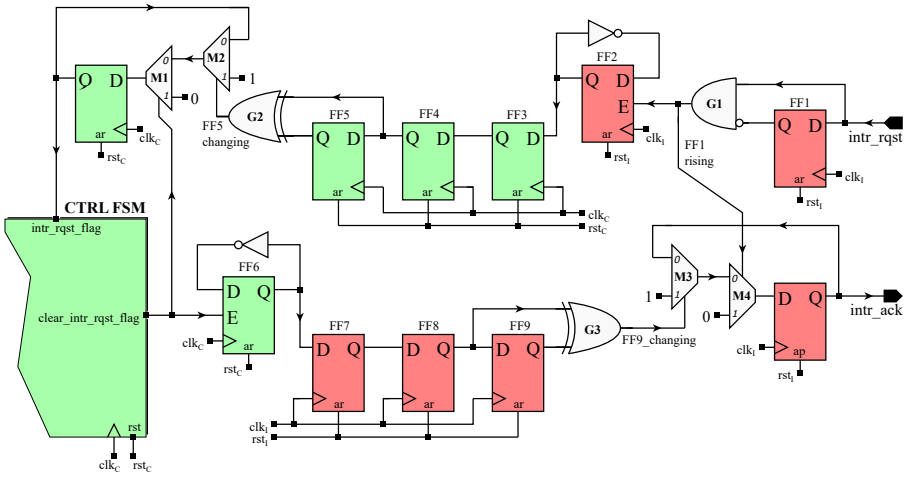
VHDL – a megszakítási forrás órajeltartománya

```
1  L_INTR_SOURCE: process ( clk_i, rst_i ) begin
2    if ( rst_i = '1' ) then
3      FF1 <= '0'; FF2 <= '0';
4      FF7 <= '0'; FF8 <= '0'; FF9 <= '0';
5      intr_ack <= '1';
6    elsif ( rising_edge(clk_i) ) then
7
8      FF1 <= intr_rqst;
9
10     if ( FF1_rising = '1' ) then
11       FF2 <= not FF2;
12     end if;
13
14     FF7 <= FF6;
15     FF8 <= FF7;
16     FF9 <= FF8;
17
18     if ( FF1_rising = '1' ) then
19       intr_ack <= '0';
20     elsif ( FF9_changing = '1' ) then
21       intr_ack <= '1';
22     end if;
23
24   end if;
25 end process;
26
27 FF1_rising <= intr_rqst and not FF1;
28 FF9_changing <= FF9 xor FF8;
```

VHDL – a *CPU* órajeltartománya

```
1  L_CPU: process ( clk_c, rst_c ) begin
2    if ( rst_c = '1' ) then
3      FF3 <= '0'; FF4 <= '0'; FF5 <= '0';
4      intr_rqst_flag <= '0';
5      FF6 <= '0';
6    elsif ( rising_edge(clk_c) ) then
7
8      FF3 <= FF2;
9      FF4 <= FF3;
10     FF5 <= FF4;
11
12     if ( clear_intr_rqst_flag = '1' ) then
13       intr_rqst_flag <= '0';
14     elsif ( FF5_changing = '1' ) then
15       intr_rqst_flag <= '1';
16     end if;
17
18     if ( clear_intr_rqst_flag = '1' ) then
19       FF6 <= not FF6;
20     end if;
21
22   end if;
23 end process;
24
25 FF5_changing <= FF5 xor FF4;
```

Áramköri struktúra



2-55. ábra. *A CPU megszakítási interfészének VHDL-modellje és kapcsolási sémája*

Ez a kapcsolás tehát alkalmas a szinkron kézfogásos interfész-specifikáció megvalósítására, de fő feladatunkat ezzel még nem végeztük el; gondoskodnunk kell róla, hogy a megszakítási interfész akkor is működjön, ha a megszakítás forrása és a *CPU* két, egymáshoz képest aszinkron órajeltartomány részei. Az interfész-áramkörünket tehát újabb elemekkel egészítjük ki a 2-55. ábrán látható módon.

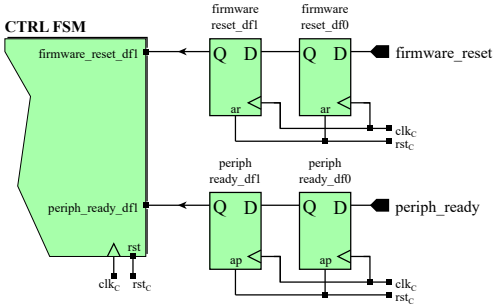
A megszakításkérést fogadó éldetektort *(FF1, G1)*, valamint a külső almodul számára a megszakításkérés nyugtázását jelző flip-flopot *(intr_ack)* ezúttal a megszakítást kérő almodul órajelével hajtjuk meg *(clk)*. Az éldetektor kimenetét az *intr_ack* nyugtázójel nullázására, illetve *FF2* negálására használjuk. Ez utóbbi művelettel a megszakításkérés eseményét egy statikus jelváltozássá transzformáljuk. Ezt a változást az *FF3* és *FF4* flip-flopok által képezett double-flop szinkronizáló fokozattal visszük át a *CPU* órajeltartományába. A double-flop szinkronizáló kimenetét egy újabb, mindkét jelváltozási irányra érzékeny éldetektorba kötjük be *(FF5, G2)*, az ez által előállított kimeneti impulzust pedig a *CPU* vezérlő állapotgépének *intr_rqst_flag* bemenetét meghajtó flip-flop *(intr_rqst_flag)* magas szintre állítására használjuk. A *CPU* ez által értesül a megszakításkérés beérkezéséről. A megszakításkérés kiszolgálása után a *CPU* vezérlő állapotgépe a *clear_intr_rqst_flag* jelen egy pozitív impulzust állít elő (lásd 2-35. ábra, VHDL-modell: 9. sor). Ezzel az impulzussal az *FF6* flip-flopot negáljuk, kimenetét pedig ezúttal is egy double-flop szinkronizáló fokozattal *(FF7, FF8)* és egy éldetektorral *(FF9, G3)* viszünk át a megszakítási forrás órajeltartományába. A megszakítás nyugtázását képviselő *intr_ack* állapotjelzőt az éldetektor kimenetével állítjuk magas szintre, lezárva a szinkron kézfogásos interfész-specifikációnak megfelelő megszakításkérési műveletet.[49]

Statikus vezérlő bemenetek szinkronizálása ◆ A *CPU* és a periféria-áramkörök közötti MMIO-interfész 2-7. ábrán látható kialakítása alapján jól látható, hogy a *CPU*-ba beépítendő CDC-áramkör – az MMIO-protokoll sajátosságainak következtében – egy egyszerű, a *periph_ready* bemenetre kötött double-flop szinkronizálóra egyszerűsödik. A *periph_ready* jelzéshez hasonlóan a *firmware_reset* jelzés is statikus bemenetnek tekinthető, így a double-flop szinkronizálás itt is megfelelő. A 2-56. ábra e két szinkronizáló fokozat kapcsolási sémáját és VHDL-modelljét mutatja be.

VHDL

```
1  L_DOUBLE_FLOPS: process ( clk_c, rst_c )
2  begin
3    if ( rst_c = '1' ) then
4      firmware_reset_df0 <= '0';
5      firmware_reset_df1 <= '0';
6      periph_ready_df0 <= '1';
7      periph_ready_df1 <= '1';
8    elsif ( rising_edge(clk_c) ) then
9      firmware_reset_df0 <= firmware_reset;
10     firmware_reset_df1 <= firmware_reset_df0;
11     periph_ready_df0 <= periph_ready;
12     periph_ready_df1 <= periph_ready_df0;
13   end if;
14 end process;
```

Áramköri struktúra



2-56. ábra. *A statikus vezérlő bemenetek szinkronizálása a CPU órajeléhez*

[49] A megszakítási interfészhez tartozó órajel-szinkronizáló fokozat kísértetiesen hasonlít a 2.1. pontban bemutatott, általános vektorszinkronizálási sémára, de teljesen nem egyezik meg azzal. Ennek magyarázata az, hogy ez a megoldás a *CPU* oldalán nem a szinkron kézfogásos interfész-specifikáció támasztotta követelmények maradéktalan teljesítését célozza, hanem a *CPU intr_rqst_flag* és *clear_intr_rqst_flag* jeleinek viselkedéséhez igazodik. Tulajdonképpen tehát ez az áramkör nemcsak órajeltartományok közötti adatátvitelt, hanem némi protokolláris illesztést is végez.

2.3. A programtárak inicializálása

Áttanulmányozásra javasolt alfejezetek:

4.2. *On-chip adattárolás*
4.3.1. *A soros kommunikáció alapáramkörei – UART-adó és -vevő (+* ***4.3.4****. Nagysebességű UART-vevő)*
4.4.1. *Katalógus-áramkörök interfészadapterei – SPI EEPROM*
4.4.2. *Katalógus-áramkörök interfészadapterei – I²C EEPROM*
4.5.4. *Aritmetika és jelfeldolgozás – CRC-kódgenerálás visszacsatolt léptetőregiszterrel*

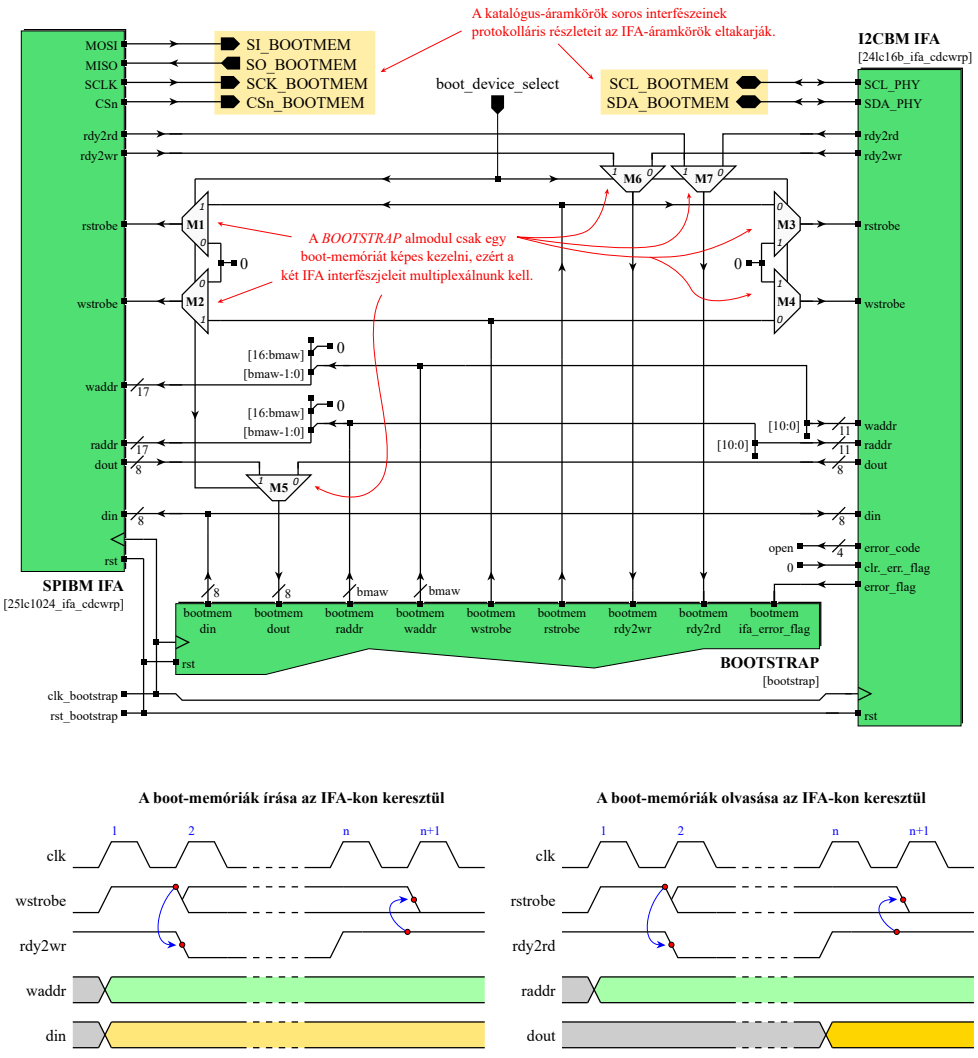
A rendszerterv 1-11. ábrán látható részletének megfelelően a *CPU* külső, nemillékony programtárainak (boot-memóriák) tartalommal való feltöltését a *BOOTSTRAP* almodulon keresztül kell megoldanunk. Kiválasztott üzemmódjától függően ennek az IP-magnak képesnek kell lennie egy UART-linken érkező bájtfolyammal feltölteni a boot-memóriákat vagy közvetlenül a *CPU* belső utasításmemóriáját *(IMEM)*, illetve a boot-memóriák tartalmát áttölteni az *IMEM* utasításmemóriába. Mindezen funkciók megvalósításához egy viszonylag összetett interfészrendszert kell kialakítanunk a *BOOTSTRAP* almodul és környezete között. A *BOOTSTRAP* almodul belső felépítésének és működésének tárgyalása előtt tekintsük át ennek az interfészrendszernek a részleteit.

A boot-memóriák nemillékony (EEPROM), a funkcióbővítő rendszert tartalmazó FPGA-hoz soros kommunikációs interfészeken (SPI és I²C) kapcsolódó katalógus-áramkörök. Mivel funkciójuk tisztán memória jellegű, ezért interfészadaptereiket úgy célszerű megtervezni, hogy klienseik szempontjából használatuk megegyezzen egy általános memóriáéval. Ennek megfelelően a funkcióbővítő rendszer toplevel moduljában példányosított – és a 4.4.1. és 4.4.2. alfejezetekben részletesen bemutatott – interfészadapterek kliensinterfészei megegyeznek egy egyszerű szinkron kézfogásos interfészű on-chip memóriamoduléval (lásd 2-57. ábra).

A *BOOTSTRAP* modul tervezésekor tehát a konkrét boot-memóriák soros interfészének protokolljával nem kell foglalkoznunk, ezeket a részleteket az interfészadapterek eltakarják előlünk. Az I²C boot-memória interfészadaptere az általános, minden memóriamodulra jellemző jeleken felül rendelkezésünkre bocsát egy hibainterfészt is, amin az esetleges I²C-buszhibákat képes jelezni (lásd 4-103. ábra: *error_flag*, *clear_error_flag*, *error_code*). A *BOOTSTRAP* modul részleteinek megtervezésekor e buszhibák kezelésére egy nagyon egyszerű megoldást alkalmazunk, ezért a hibainterfész jelei közül csak az *error_flag* állapotjelzőre lesz szükségünk.

A két boot-memóriát sosem szeretnénk egyidőben használni, ezért a *BOOTSTRAP* modult sem készítjük fel több boot-memória egyidejű kezelésére. A két interfészadapter összekapcsolása egyazon *BOOTSTRAP* almodullal némi illesztőlogika beépítését igényli, ami – abból adódóan, hogy a két boot-memória közül egy statikus (a működés teljes időtartama alatt fix értékű) bemenet *(boot_device_select)* állapota alapján választunk – valójában csupán egy néhány multiplexerből álló kombinációs hálózat lesz.[50] A *BOOTSTRAP* modul *strobe* jeleit az *M1...M4* multiplexereken keresztül kapcsoljuk az interfészadapterek megfelelő bemeneteire. E multiplexerek gondoskodnak arról, hogy a *boot_device_select* bemenet állapotától függően vagy csak az egyik, vagy csak a másik IFA-hoz jussanak el a *BOOTSTRAP* modul *strobe* kimeneteinek az írás vagy olvasás kezdeményezését jelző változásai, a nem kiválasztott IFA tehát a működés teljes időtartama alatt inaktív marad. Az *M5...*

[50] A *boot_device_select* jelet akár az üzemeltetői interfész részének is tekinthetjük, és meghajtását az üzemeltető mikrovezérlőre bízhatjuk. A *BOOTSTRAP* modul megvalósítása szempontjából csak az a fontos, hogy a funkcióbővítő rendszer *reset* bemenetének elengedése után ez a bemenet ne változzon.



2-57. ábra. *A BOOTSTRAP modul és a boot-memóriák interfészadaptereinek összeköttetései*

M7 multiplexerek pedig biztosítják azt, hogy a *BOOTSTRAP* modul bootmemória-interfészének bemeneteit mindig az éppen aktív interfészadapter kimenetei hajtsák meg. Hibakimenetet *(error_flag)* csak az I²C EEPROM interfészadaptere állít elő. Ezt közvetlenül beköthetjük a *BOOTSTRAP* modul *bootmem_ifa_error_flag* bemenetébe, hiszen ha a *boot_device_select* bemenet nem ezt az interfészadaptert választja ki, akkor – mivel az IFA *strobe* jelei így sosem aktiválódnak – annak *error_flag* kimenete mindvégig alacsony szintű marad.

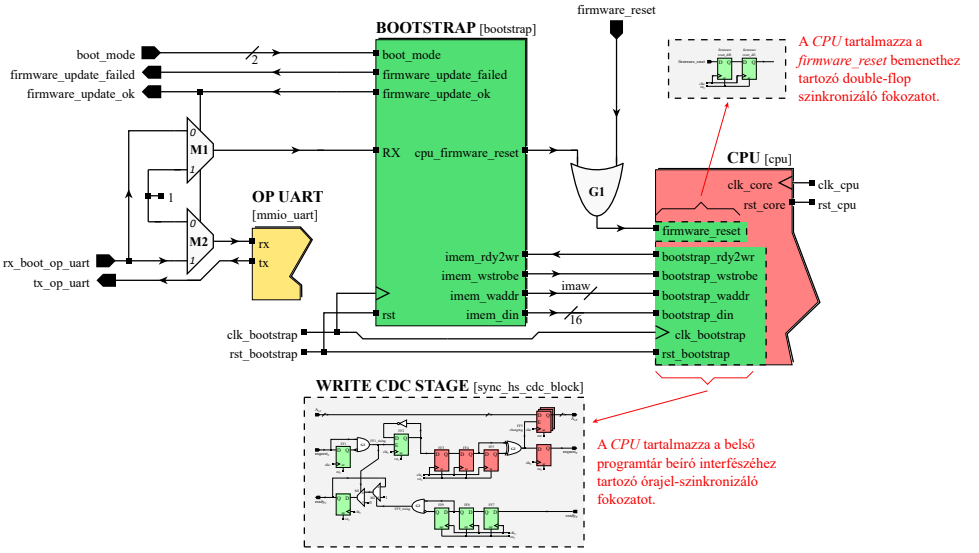
A boot-memóriák kapacitását a *BOOTSTRAP* modul tervezése során paraméternek tekintjük, de feltételezzük, hogy az 2 kB és 256 kB között van. Ez azt jelenti, hogy a *BOOTSTRAP* modulhoz kapcsolt interfészadapterek címbuszainak mérete legalább 11, legfeljebb 17 bit. A *BOOTSTRAP* modul példányosításakor meghatározandó konkrét értéket a *bmaw (**b**oot**m**em **a**ddress **w**idth)* szintézis-paraméter

rögzíti. Funkcióbővítő rendszerünkben a boot-memóriaként használt 25LC1024 és 24LC16B EEPROM chipek rendre 128 kB és 2 kB kapacitásúak, interfészeik címbuszai tehát 17, illetve 11 bitesek. A 24LC16B chiphez tartozó interfészadapter címbuszait a *BOOTSTRAP* modul címbuszainak alsó 11 bitjével hajtjuk meg, azok felsőbb bitjeit figyelmen kívül hagyjuk. A 25LC1024 chiphez tartozó IFA címbuszaival kissé eltérő módon kell eljárnunk, hiszen a *bmaw* paraméter értéke 17-nél kisebb is lehet. Itt a *BOOTSTRAP* címbuszainak alsó *bmaw* db bitjét kapcsoljuk az interfészadapter címbemeneteire, a felső biteket pedig alacsony szintre húzzuk. Ez azt jelenti, hogy ha a toplevel integrátor a *bmaw* paraméter értékét 17-nél kisebbre állítja be, akkor az EEPROM-nak a $(2^{bmaw}-1)$-nél magasabb címen lévő rekeszeit nem használjuk ki.

Az EEPROM interfészadapterekhez – az újrafelhasználható IP-magokkal szemben megfogalmazott követelményekkel összhangban – rendelkezésünkre állnak azok az órajel-szinkronizáló fokozatok, amelyek segítségével az IFA-knak és beágyazó környezetüknek órajelei függetleníthetők egymástól (lásd 4.4.9. pont). Ahogy az a 2-57. ábrán is látszik, ebben a konkrét szituációban ezeket nem használjuk, az IFA-k és a *BOOTSTRAP* modul mind egyazon órajelről járnak. Ennek oka kettős; egyrészt a bootstrap-folyamatok időigényét alapvetően az aránylag lassú UART, SPI és I^2C-linkek sebessége határozza meg, így az ezektől függő *BOOTSTRAP* modul órajelét felesleges lenne magasabbra állítani, hiszen az idő nagy részében tétlenül várja, hogy feldolgozható információhoz jusson. Másrészt a teljes boot-mechanizmust rendszerszinten sem tekintjük sebességkritikusnak, így a *BOOTSTRAP* modul magasabb frekvenciájú órajeltartományba helyezése semmiképp sem indokolt.

A *BOOTSTRAP* modulnak – kétbites *boot_mode* bemenete által kiválasztott üzemmódjától függően – képesnek kell lennie a boot-memóriákat és a *CPU* belső utasításmemóriáját tartalommal feltölteni. E folyamatok állapotát a felhasználó – vagy akár az üzemeltető mikrovezérlő – a *BOOTSTRAP* modul két egybites, magas-aktív kimenete alapján ellenőrizheti; a *firmware_update_ok* kimenet magas szintje az aktuálisan zajló folyamat sikeres befejezését jelzi, a *firmware_update_failed* kimenet magas szintje pedig arra utal, hogy a folyamat közben valami hiba lépett fel, ezért a rendszer teljes újraindítására van szükség. Függetlenül attól, hogy melyik memóriamodult szeretnénk tartalommal feltölteni, maga a beírandó információ – ami nem más, mint a futtatandó program bájtkódja – eredendően egy UART-linken keresztül érkezik. Ennek protokollja (lásd később) nem igényel visszirányú kommunikációt, ezért ez az UART-link valójában csupán egyetlen vezetéket jelent, amit – tekintve, hogy a program letöltése közben a *CPU* garantáltan inaktív, így parancsok fogadására alkalmatlan – az üzemeltetői UART-interfész soros-párhuzamos átalakítója és a *BOOTSTRAP* modul megosztva használ (lásd 2-58. ábra: *rx_boot_op_uart*). Ha a program letöltése sikerrel zárul, akkor a *BOOTSTRAP* modul magas szintre állítja *firmware_update_ok* kimenetét, ezzel „átirányítva" az *rx_boot_op_uart* bemenetet a *BOOTSTRAP* modul *rx* bemenetéről az üzemeltetői UART-interfész soros-párhuzamos átalakító funkcióját betöltő *OP_UART* perifériavezérlő *RX* bemenetére.

A *BOOTSTRAP* modul közvetlenül kapcsolódik a *CPU* belső utasításmemóriájának beíró interfészéhez. Ez a belső utasításmemória *(IMEM)* a 4-21. ábrán bemutatott *singleclock_dualport_ram_hswrp* IP-mag egy példánya, így a *BOOTSTRAP* modul által kezelt beíró interfész működése ugyancsak megfelel a szinkron kézfogásos interfész-specifikációnak. Korábbi megállapításunk alapján a *BOOTSTRAP* modult nem érdemes magas frekvenciájú órajelről járatni, ugyanakkor a *CPU* magórajele nagymértékben befolyásolja az utasításvégrehajtás sebességét, ezért annak frekvenciáját érdemes aránylag magasra állítani. A két modul közötti órajel-szinkronizáló fokozatokat a *CPU* már



2-58. ábra. *A BOOTSTRAP modul egyéb interfészei*

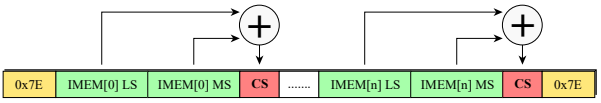
tartalmazza (vö. 2-50. ábra), ezért a funkcióbővítő rendszer toplevel moduljában a *BOOTSTRAP* és a *CPU* interfészjeleit közvetlenül összekapcsolhatjuk.

A *cpu_firmware_reset* kimenetének magas szinten tartásával a *BOOTSTRAP* modul a boot-folyamatok alatt a *CPU*-t *reset* állapotban tartja. Ha az adott folyamat a *CPU* programjának futtatásával ér véget, akkor ezt a kimenetet a *BOOTSTRAP* modul nullába húzza, elindítva a *CPU* működését.[51] A *CPU firmware_reset* bemenetéhez tartozó double-flop szinkronizáló fokozatot ugyancsak a *CPU* maga tartalmazza (lásd 2-56. ábra).

A *BOOTSTRAP* modul interfészeinek áttekintése után még két dolgot kell tisztáznunk, mielőtt nekilátnánk az RTL-modell elkészítésének. A rendszerterv előírja ugyan, hogy a programtárak tartalmát UART-interfészen kell fogadnunk, de arról nem nyilatkozik, hogy a futtatandó programot pontosan milyen módon kódoljuk bele ebbe az UART bájtfolyamba, használjunk-e bármiféle forgalomszabályozást, hogyan védekezzünk az esetleges átviteli hibák ellen stb. Rögzítenünk kell tehát a *BOOTSTRAP* modul UART-interfészének magasabb szintű protokollját. A rendszerterv hasonlóan elnagyolt a boot-memóriákkal kapcsolatban; világos, hogy ezeknek a memóriáknak képesnek kell lenniük egy-egy – akár különböző – utasítássorozat eltárolására, de a tárolás módjának részletei, mint a bájtsorrend, a hibavédelmi mechanizmusok stb., egyelőre szintén tisztázatlanok. Az ezekkel a részletekkel kapcsolatos döntések alapvetően meghatározzák a *BOOTSTRAP* modul RTL-modelljének felépítését és működését, ezért a tervezést ezek rögzítésével folytatjuk.

Az UART-interfész protokollja ◆ A *BOOTSTRAP* modul UART-interfészének adatkapcsolati réteg feletti protokollja a probléma egyszerűségéből adódóan nem lesz túl bonyolult. Egyetlen adatcsomagot definiálunk, amely a programtár teljes tartalmát magában foglalja. A csomag szerkezetét a 2-59. ábra szemlélteti.

[51] A *CPU*-t a külső *firmware_reset* jel is képes újraindítani a *G1 OR* kapun keresztül, ennek azonban a *BOOTSTRAP* modul működése szempontjából nincs jelentősége.



2-59. ábra. *A BOOTSTRAP modul UART feletti protokolljának csomagszerkezete*
*LS: **L**east **S**ignificant (bájt), MS: **M**ost **S**ignificant (bájt), CS: **c**heck**s**um*

A csomag a 0x7E jelzőkarakterrel kezdődik, amelyet programtárbejegyzések (utasítások) követnek. Az utasítások 16-bitesek, ezért minden utasítás két karakter (két bájt) átvitelét igényli. Az egyazon utasításhoz tartozó karakterpárok átvitelét egy-egy ellenőrzőösszeg (checksum) követi, ami az őt megelőző két utasítás-bájt összege.[52] A programtár tartalmát meghatározó bájtfolyam végét a 0x7E karakter újbóli elküldése jelzi. A jelzőkarakter speciális szerepe miatt az utasításokat tároló bájtfolyamban a 0x7E érték nem szerepelhet. A probléma megoldására az ún. PPP (**P**oint-to-**P**oint **P**rotocol) keretezési módszert alkalmazzuk. Ha a hasznos adat (az utasítások és az ellenőrzőösszegek) a 0x7E karaktert tartalmazzák, akkor az adó egy ún. escape-karaktert (0x7D) iktat be az átvitelbe a 0x7E karakter elé, majd a 0x7E karakter helyett a 0x5E (0x20 XOR 0x7E) karaktert továbbítja. Továbbá, ha a hasznos adat magát a 0x7D escape-karaktert tartalmazza, akkor az adó hasonló módon eljárva újabb escape-karaktert iktat be a hasznos adatban lévő 0x7D karakter elé, majd ahelyett a 0x5D (0x20 XOR 0x7D) karaktert küldi el. A *BOOTSTRAP* modul ennek megfelelően dekódolja a bejövő bájtfolyamot, amelyben a 0x7E karakter immár csak a csomag elején és végén fordulhat elő.

A boot-memória tartalma ◆ A *CPU* belső utasításmemóriájának 16-bites címbusza akár 65.536 utasításból álló programok létrehozását is lehetővé teszi, de mivel az alkalmazások többségében ez feleslegesen sok blokk-RAM ,modult emésztene fel az FPGA-n belül, ezért a *CPU* és a *BOOTSTRAP* modul HDL-modelljében is szintézis-paraméterként kezeljük a programtár címbuszának tényleges méretét *(imem_addr_width)*, ami értelemszerűen a boot-memóriákban és a *CPU* belső utasításmemóriájában eltárolandó bájtkód maximális méretét is meghatározza. A funkcióbővítő rendszernek egy szintetizált példányában tehát a maximális programméret rögzített, de ez nem jelenti azt, hogy ennél kisebb programot a rendszer ne tudna kezelni. Tehát amikor a *BOOTSTRAP* modul a boot-memóriákból tölti át a programot a *CPU* belső utasításmemóriájába, akkor szüksége van az éppen áttöltött program tényleges méretére. A boot-memóriákban ezért a programkódon kívül a program méretét is el kell tárolnunk. A 2-2. táblázat a boot-memóriák tartalmát mutatja be egy N db utasításból álló programkód esetén.

A *BOOTSTRAP* modul UART protokollja egy PPP-kódolt, változó méretű csomagba tömöríti a teljes programkódot. A fogadott bájtsorozatot már a soros átvitel közben elkezdjük betölteni a boot-memóriába vagy közvetlenül a *CPU* utasításmemóriájába, a célterület írása tehát elkezdődik, mielőtt még tudnánk, hogy hány bájtot kell majd eltárolnunk. Ennek a boot-memóriák címtérképe szempontjából azért van jelentősége, mert így az eltárolt programhoz tartozó metaadatokat (adatokat leíró adatok) csak a programot alkotó bájtok elé, fix címekre írhatjuk. A programot alkotó utasítások számát a 0...2. címeken tároljuk, magát a programot pedig az 5. címtől kezdődően kezdjük betölteni. A boot-memóriák bájt-szervezésűek, tehát minden 16-bites utasítás két címet foglal el.

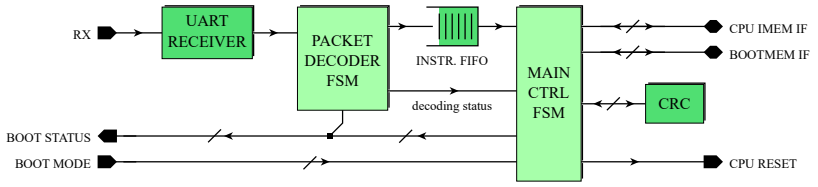
[52] Pontosabban fogalmazva a két bájt összegét 256-tal leosztva kapott maradék, hiszen csak 8 bit áll rendelkezésre az összeg tárolására.

Memóriacím (i)	Rekesz tartalma (BM[i])
0	Utasítások száma – bájt #0 [7:0]
1	Utasítások száma – bájt #1 [15:8]
2	Utasítások száma – bájt #2 [23:16]
3	Ellenőrzőösszeg: utasítások száma $\left(\sum_{i=0}^{2} BM[i]\right) \% 256$
4	Ellenőrzőösszeg: utasítások CRC (BM[5] … BM[2 × (N-1)+6])
5	Utasítás #0 – bájt #0 [7:0]
6	Utasítás #0 – bájt #1 [15:8]
7	Utasítás #1 – bájt #0 [7:0]
8	Utasítás #1 – bájt #1 [15:8]
…	…
2 × (N-1) + 5	Utasítás #(N-1) – bájt #0 [7:0]
2 × (N-1) + 6	Utasítás #(N-1) – bájt #1 [15:8]

2-2. táblázat. *A boot-memóriák tartalma N utasításból álló programkód esetén*

A boot-memóriákban tárolt információk kritikusak a rendszer helyes működése szempontjából; akár a program méretét tartalmazó bejegyzések, akár maga a programkód sérül a tárolás során, a *BOOTSTRAP* modul működését fel kell függeszteni, és a hibát jelezni kell az üzemeltető mikrovezérlő felé. A kétféle adathalmaz sértetlenségét egy-egy ellenőrzőösszeg segítségével tesszük igazolhatóvá; a hárombájtos programkódméret esetén megelégszünk e három bájt egyszerű összegével (3. cím), a több ezer vagy akár több tízezer bájtot magában foglaló programkódnál azonban egy kifinomultabb megoldást, CRC (**C**yclic **R**edundancy **C**heck) kódot használunk (4. cím).

Mindezekkel a tervezői döntésekkel kiegészítve a rendszertervet a *BOOTSTRAP* modullal kapcsolatban már összegyűlt annyi elképzelés és elvárás, hogy annak részletes RTL-tervezését megkezdhessük. Ahogy az RTL-modellek általában, úgy a *BOOTSTRAP* modul RTL-terve is általános célú, újrafelhasználható IP-magok és a konkrét problémára egyedileg tervezett részegységek megfelelően összekapcsolt hálózata lesz. A toplevel modul egyszerűsített blokkvázlatát a 2-60. ábra szemlélteti.



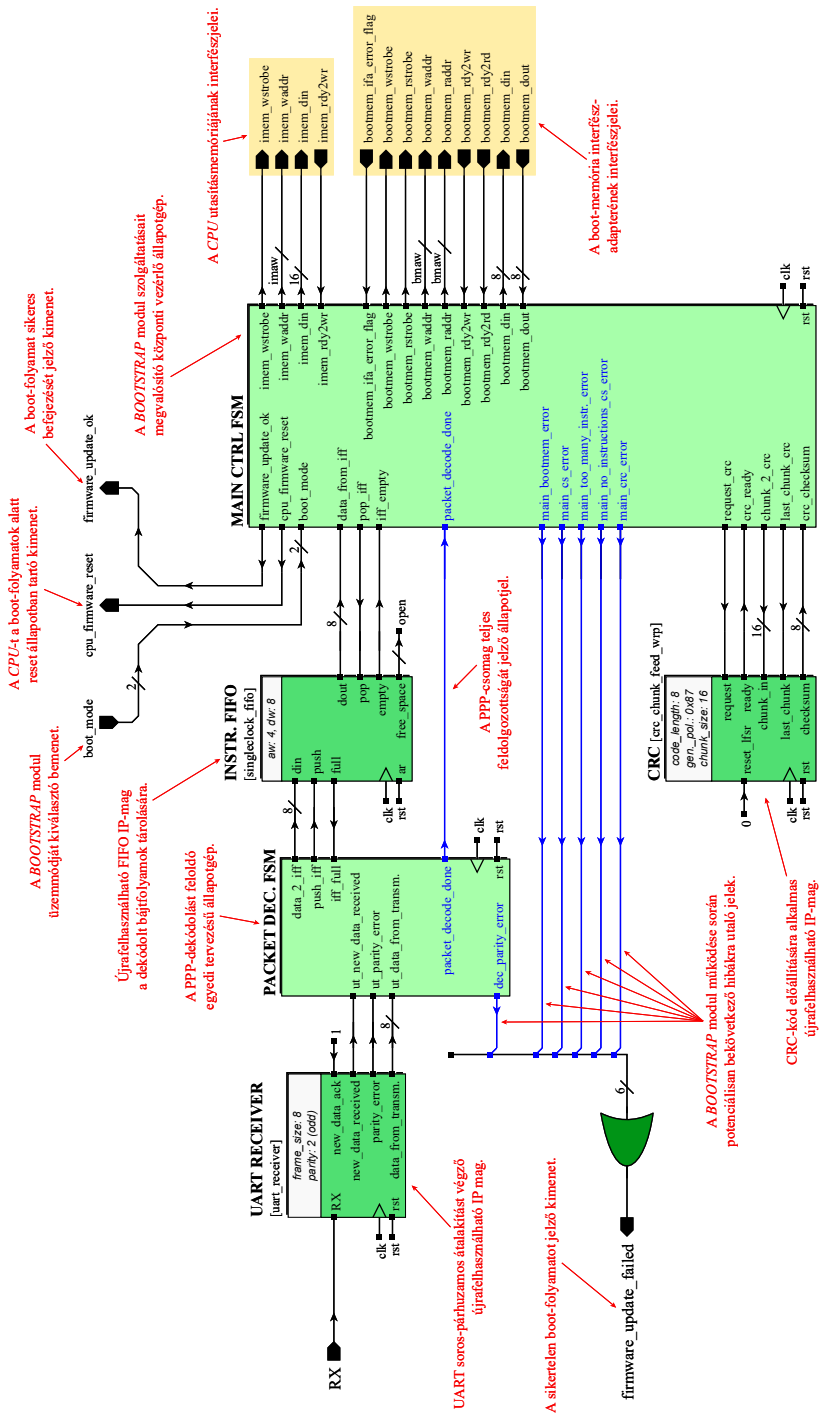
2-60. ábra. *A BOOTSTRAP modul egyszerűsített blokkvázlata*

Az UART-linken a soros-párhuzamos átalakítást a 4.3.1. pontban részletesen tárgyalt *uart_receiver* modullal végezzük el. Ez az újrafelhasználható IP-mag kiszolgáló típusú – tehát tranzakciók kezdeményezésére képes – felhasználói interfészén a 2-59. ábrán látható, PPP-kódolt bájtfolyam bájtjait egyesével adja át a PPP-kódolás feloldását végző, egyedi tervezésű állapotgépnek *(PACKET DECODER FSM)*. Ennek az állapotgépnek nincs tudomása arról, hogy a beérkező adatfolyam egyes bájtjai közül melyek alkotnak egy utasítást vagy melyek azok, amelyek ellenőrzőösszeget hordoznak,

feladata nem mutat túl az adatcsomag elejét és végét jelző 0x7E karakterek eltávolításán és az escape-szekvenciák feloldásán. A dekódolt bájtfolyamot – ami immár csak magukat az utasításokat és az azokhoz csatolt ellenőrzőösszegeket tartalmazza – egy FIFO-ba írja be *(INSTR. FIFO)*. A FIFO-ban való adattárolás – az UART soros-párhuzamos átalakításhoz hasonlóan – egy általános funkció, amit a 4.2.2. pontban bemutatott *singleclock_fifo* IP-mag egy példánya lát el. Ennek a köztes tárolónak az olvasását és a kiolvasott bájtsorozat feldolgozását egy ugyancsak egyedi tervezésű állapotgép végzi *(MAIN CTRL FSM)*. Ennek az állapotgépnek a feladata, hogy az egyes gépi utasításokhoz tartozó ellenőrzőösszegek segítségével meggyőződjön arról, hogy az UART átvitel közben a bájtfolyam nem sérült, valamint hogy a boot-memóriák interfészadaptereihez és a *CPU* belső utasításmemóriájának beíró interfészéhez kapcsolódva megvalósítsa a *BOOTSTRAP* modul rendszertervben rögzített szolgáltatásait, vagyis a memóriatartalmak inicializálását. Egy programkód boot-memóriákban való elhelyezésekor és egy már ott lévő programkód kiolvasásakor a *MAIN CTRL FSM* állapotgépnek CRC-ellenőrzést kell végeznie. Ezt a részfunkciót szintén egy újrafelhasználható IP-maggal, a 4.5.4. pontban bemutatott *crc_chunk_feed_wrp* modullal valósítjuk meg.

A *BOOTSTRAP* modulon belül és annak áramköri környezetében példányosított újrafelhasználható IP-magok interfészjelei pontosan ismertek, így a 2-60. ábrán látható blokkvázlatot egy egészen részletes RTL kapcsolási sémává fejleszthetjük tovább, ami nagy segítséget jelenthet az újonnan megtervezendő, egyedi állapotgépek HDL-leírásainak elkészítésekor. A 2-61. ábra ezt a részletes RTL-tervet szemlélteti. Figyeljük meg, hogy a részletes RTL kapcsolási sémában az egyedi állapotgépek interfészjeleinek túlnyomó része a már meghozott tervezői döntéseink következménye. Újdonságot az eddigiekhez képest csupán a két állapotgép közötti *packet_decode_done* állapotjelző és a működés során bekövetkező hibákat képviselő, a *firmware_update_failed* kimenetet egy *OR* kapun keresztül meghajtó hibajelek jelentik. E jelek szerepét a két állapotgép részletes tervezése során tisztázzuk.

A *CPU* vezérlőegységét one-shot típusú, az állapotgép minden részfunkcióját egyetlen szinkron folyamatba tömörítő HDL-leírással modelleztük. Ez az állapotgép egy különálló HDL kódrészletekkel leírt műveletvégző egység alkatrészeinek vezérlőjeleit állítja elő. A *BOOTSTRAP* modul két állapotgépénél ugyancsak a one-shot leírási módszert használjuk, de a *CPU*-étól lényegesen eltérő jellegű funkciójuk miatt a különálló műveletvégző elemek létrehozása helyett ezúttal az adatmanipulációt kifejező értékadásokat is beágyazzuk az állapotgépek leírásába (lásd még DRMSz: 135. oldal). Az állapotgépek HDL-modelljeinek vázlatait a 2-62. és 2-64. ábrák mutatják be.



2-61. ábra. *A BOOTSTRAP modul toplevel RTL kapcsolási sémája*

PPP-dekóder állapotgép ◆ A PPP-kódolás során a *CPU* programját jelentő bájtfolyam olyan átalakításon esik át, ami lehetővé teszi a változó hosszúságú adatcsomagok vevője számára a csomag elejének és végének egyértelmű azonosítását. A PPP-dekóder állapotgép feladata, hogy ebből a módosított bájtfolyamból visszaállítsa az eredeti tartalmat, vagyis a *CPU* programkódját, és az azt alkotó bájtokat az *INSTR. FIFO*-tárolóba mentse.

VHDL

```
1  L_PACKET_DECODER_FSM: process ( clk, rst ) begin
2    if ( rst = '1' ) then
3      state_dec <= idle; push_iff <= '0'; data_2_iff <= (others => '0'); esc_received_flag <= '0';
4      no_7es <= 0; packet_decode_done <= '0'; dec_parity_error <= '0';
5    elsif ( rising_edge(clk) ) then
6    |
7    |  case state_dec is
8    |  |
9    |  |  when idle => if ( ut_new_data_received = '1' ) then
10   |  |                  if ( ut_parity_error = '0' ) then state_dec <= check_data;
11   |  |                  else dec_parity_error <= '1'; state_dec <= error;
12   |  |                  end if;
13   |  |                end if;
14   |  |
15   |  |  [...]
16   |  |
17   |  |  when error  => null;
18   |  |
19   |  |  when others => null;
20   |  |
21   |  end case;
22   |
23   end if;
24 end process;
```

2-62. ábra. *A PACKET DECODER FSM állapotgép VHDL-leírása (1)*

Amíg az UART vonalon nem érkezik feldolgozandó adat, addig a PPP-dekóder várakozik (2-62. ábra: *idle* állapot). Amikor az UART-vevő áramkör – *new_data_received* kimenetének magas szintre állításával – egy új adatbájt beérkezését jelzi, akkor a PPP-dekóder annak további feldolgozása előtt meggyőződik róla, hogy a kapott bájt nem paritáshibás-e (2-62. ábra: 10. sor). Hibás paritású adat fogadása esetén az állapotgép a hibát a *dec_parity_error* kimenet magas szintre húzásával jelzi, ami a 2-61. ábrán látható *OR* kapun keresztül egyúttal a *BOOTSTRAP* modul *firmware_update_failed* kimenetén is magas szintet idéz elő. A PPP-dekóder állapotgép ekkor az *error* állapotba lép, ahonnan csak reset hatására képes kilépni, a programtár inicializálásának folyamata tehát megszakad. Ha az UART-vevőtől kapott bájt hibátlan, akkor a PPP-dekóder a *check_data* állapotba lépve elkezdi annak vizsgálatát (lásd 2-63. ábra).

A PPP-kódolásban kitüntetett szerepe van a 0x7E és 0x7D karaktereknek, nem meglepő tehát, hogy a PPP-dekóder állapotgép *check_data* állapota is ezeket „keresi" a bejövő bájtfolyamban. Egy adott bájt feldolgozásának mikéntje attól függ, hogy a közvetlenül előtte kapott bájt escape-karakter (0x7D) volt-e vagy sem. Az (n+1)-edik bájt fogadásának pillanatában az n-edik bájt az UART-vevő kimenetén már nem áll rendelkezésre, ezért az „előző karakter egy escape-karakter volt" szituációt egy egybites, belső állapotjelző *(esc_received_flag)* segítségével tartjuk számon.

Ha az éppen feldolgozott adatbájtot nem előzte meg escape-karakter, illetve maga az adatbájt nem escape-karakter és nem is a csomag elejét vagy végét jelző 0x7E jelzőkarakter, akkor azt az *INSTR. FIFO* adatbemenetére másoljuk, majd a *push_iff_1* állapotba lépve – amikor a FIFO készen áll új adat beírására – kezdeményezzük a FIFO írását (2-63. ábra: 24. sor). A FIFO írása után az *idle* állapotba visszalépve várakozunk a következő beérkező adatbájtra (2-63. ábra: 30. sor). Ha egy 0x7D escape-karakter érkezett, akkor az *esc_received_flag* állapotjelzőt magas szintre állítjuk, de

VHDL

```
1  when check_data =>
2    if ( esc_received_flag = '0' ) then
3
4      if ( ut_data_from_transmitter = X"7E" ) then
5
6        if ( no_7es < 1 ) then no_7es <= no_7es + 1; state_dec <= idle;
7        else state_dec <= end_of_packet;
8        end if;
9
10     elsif ( ut_data_from_transmitter = X"7D" ) then
11       esc_received_flag <= '1'; state_dec <= idle;
12     else
13       data_2_iff <= ut_data_from_transmitter; state_dec <= push_iff_1;
14     end if;
15
16   else
17     esc_received_flag <= '0';
18     data_2_iff <= X"20" xor ut_data_from_transmitter;
19     state_dec <= push_iff_1;
20   end if;
21
```

VHDL (folyt.)

```
22 when push_iff_1 =>
23   if ( iff_full = '0' ) then
24     push_iff <= '1';
25     state_dec <= push_iff_2;
26   end if;
27
28 when push_iff_2    =>
29   push_iff <= '0';
30   state_dec <= idle;
31
32 when end_of_packet =>
33   packet_decode_done <= '1';
```

2-63. ábra. *A PACKET DECODER FSM állapotgép VHDL-leírása (2)*

magával az escape-karakterrel mint adattal nincs feladatunk, ezért rögtön visszatérünk az újabb bájt beérkezésére várakozó *idle* állapotba (2-63. ábra: 10-11. sor). A közvetlenül az escape-karakter után beérkező bájtok esetén az UART-vevő kimenetéről leolvasott érték az eredeti (kódolatlan) adatbájt és a 0x20 karakter *XOR* kapcsolatából képződött, a visszaállítás pedig ugyanennek a műveletnek az újbóli elvégzésével lehetséges (2-63. ábra: 18. sor). A visszaállított adatbájtot ezúttal is a *push_iff_1* és *push_iff_2* állapotok töltik be az *INSTR. FIFO*-tárolóba.

A PPP-csomag végét a beérkezett 0x7E jelzőkarakterek számolásával detektáljuk; az első ilyen karakter fogadásakor a számláló növelésén túl nincs teendőnk (2-63. ábra: 6. sor), a második alkalom viszont a csomag végét jelöli (2-63. ábra: 7. sor). Ekkor az *end_of_packet* állapotba lépve magas szintre húzzuk a *MAIN CTRL FSM* állapotgépnek szóló *packet_decode_done* állapotjelzőt. Az *end_of_packet* állapotban szándékosan nem nyilatkozunk a *state_dec* regiszter értékével kapcsolatban, ami így változatlan marad, így ettől a pillanattól kezdve a PPP-dekóder állapotgép egészen a következő reset-ig inaktív lesz.

Központi vezérlő állapotgép ◆ A *BOOTSTRAP* modul központi vezérlő állapotgépe (2-61. ábra: *MAIN CTRL FSM*) a *CPU* vezérlő állapotgépénél némileg összetettebb, ezért a benne alkalmazott HDL kódolási technikáról még a vezérlési állapotok részletes vizsgálata előtt érdemes pár szót ejteni.

Pár gondolat a szubrutinszerű vezérlésiállapot-sorozatokról…

Az állapotgépek egymást követő vezérlési állapotok láncolatán keresztül hajtják végre feladatukat. Az egyes állapotokban vezérlőjeleket állítunk elő a megfelelő ütemezés szerint különféle műveletvégző elemek számára, egyéb almodulokkal lépünk kölcsönhatásba azok interfészjeleinek jól meghatározott időzítési szabályoknak (protokolloknak) megfelelő „billegtetésével”, vagy akár adatfeldolgozási lépéseket is végezhetünk, amelyek esetén ugyancsak fontos az ok-okozati viszonyokat figyelembe vevő sorrendi ütemezés pontos kézbentartása. A vezérlési állapotok sorozata tulajdonképpen egy algoritmust definiál, hiszen a bejárt állapotok függhetnek ugyan a mindenkori bemenetek értékétől, de a működés mindig egyértelmű és determinisztikus.

Az algoritmusok az elterjedten használt definíció szerint egy probléma megoldásához vezető elemi lépések sorozatai. Az „elemi lépés" persze a kontextustól függően sok minden lehet; elemi lépésnek hívhatjuk egy videófájl-tömörítő program működésében egy képkocka kétdimenziós diszkrét Fourier-transzformáltjának előállítását, dacára annak, hogy ez a művelet önmagában is számtalan egyszerűbb lépés végrehajtását takarja (lásd 5.6. Függelék). A BOOTSTRAP modul példájánál maradva mondhatjuk azt, hogy a programtárak inicializálásának „első lépése" az UART-on érkező adatfolyam PPP-dekódolása, ugyanakkor korábban láttuk, hogy maga a PPP-dekódolás is még elemibb lépések sorozatát igényli. Általánosan megfogalmazva tehát egy algoritmus egy probléma megoldásához vezető további, egyszerűbb algoritmusok hierarchikusan egymásba ágyazott struktúrája. Ezt az egymásbaágyazottságot a programozási nyelvek – legyenek azok utasításkészlet-függő assembly vagy absztrakt, magas szintű programozási nyelvek (C, Python stb.) – a „szubrutinok" fogalmán keresztül ragadják meg. A gyakran használt, bizonyos értelemben általános célú algoritmusokat szubrutinokba szervezzük, amelyeket aztán magasabb szintű algoritmusokból „hívhatunk".

Ezt a szubrutinfogalmat az állapotgépek vezérlési állapotaira is kiterjeszthetjük; ha egy összetett állapotgép által végrehajtott algoritmusban egy egyszerűbb műveletsorra többször is szükség van (pl. egy másik IP-maggal való kölcsönhatás, részszámítás elvégzése), akkor célszerű ezt az egyszerűbb műveletsort egy különálló vezérlésiállapot-sorozatként definiálni, amit azután a „főprogramot" alkotó állapotok bármelyikéből „hívhatunk". A programozási nyelvek szubrutinfogalmával való analógia teljessé tételéhez technikailag már csak azt kell lehetővé tennünk, hogy a szubritunszerű állapotsorozat végén az állapotgép ne egy konkrét állapotba, hanem a mindenkori „hívó" állapotot követő vezérlési állapotba lépjen vissza, imitálva a programozási nyelvek „return" utasításának működését.

A *BOOTSTRAP* modul központi vezérlő állapotgépének feladata elég sokrétű. Ez az a funkcionális egység, amelyben „összefutnak" a *BOOTSTRAP* modulon belüli (PPP-dekódoláson átesett utasítás-bájtfolyamot tároló FIFO és *CRC* modul) és azon kívüli (boot-memóriák interfészadapterei és a *CPU* belső utasításmemóriája) részegységek interfészei, amelyeken keresztül a rendszertervben megfogalmazott bootstrap-műveleteket végre lehet hajtani:

- *CPU* belső utasításmemóriájának közvetlen inicializálása és a programfuttatás indítása.
- Programkód betöltése a boot-memóriák egyikébe.
- Boot-memóriában tárolt program áttöltése a *CPU* belső utasításmemóriájába és a programfuttatás indítása (a *BOOTSTRAP* modul HDL-modellje erre utal „boot"-ként).

A *MAIN CTRL FSM* állapotgép VHDL-modelljének vázlatát a 2-64. ábra mutatja be. A *BOOTSTRAP* modul viselkedése a különböző üzemmódokban egy-egy aránylag összetett algoritmussal írható le. Ezek vezérlési állapotok sorozatára való „lefordítását", vagyis a *MAIN CTRL FSM* állapotgép részletes tervezését több egyszerűbb, önmagában is értelmezhető részalgoritmus definiálásával, és az azokat megvalósító szubrutinszerű vezérlésiállapot-sorozatok létrehozásával kezdjük:

- Utasítás elhelyezése a *CPU* belső utasításmemóriájában: A *CPU* utasításmemóriájának beíró interfésze egy szinkron kézfogásos interfész, amelynek késleltetését a *BOOTSTRAP* modul nem ismerheti. A kézfogás mechanizmus teljes körű megvalósítására van tehát szükség, ami ezt a késleltetést a *BOOTSTRAP* modul HDL-modellje szempontjából irrelevánssá teszi (lokalitás).
- Utasítás elhelyezése a boot-memóriában és a teljes programot lefedő CRC-kód frissítése: A *CPU* gépi utasításai két bájtból állnak, így egyetlen utasítás kiírása két, önmagában is

VHDL

```
1  L_MAIN_CTRL_FSM: process ( clk, rst ) begin
2  | if ( rst = '1' ) then [...] -- reset all outputs and internal registers...
3  | elsif ( rising_edge(clk) ) then
4  | |
5  | | case state_main is
6  | | |
7  | | | when read_boot_mode => case boot_mode is
8  | | |                        | when "00"   => state_main <= pop_fifo;     -- EXT ROM programming
9  | | |                        | when "01"   => state_main <= pop_fifo;     -- direct CPU IMEM programming
10 | | |                        | when "10"   => state_main <= boot_1;       -- boot from EXT ROM
11 | | |                        | when others => firmware_update_ok <= '1';  -- silent
12 | | |                        |                cpu_firmware_reset <= '0';
13 | | |                       end case;
14 | | |
15 | | | [...]
16 | | |
17 | | | when error  => null;
18 | | |
19 | | | when others => null;
20 | | |
21 | | end case;
22 | |
23 | end if;
24 end process;
```

2-64. ábra. *A MAIN CTRL FSM állapotgép VHDL-leírása (1)*

több vezérlési állapotot igénylő tranzakciót igényel az állapotgép és a boot-memória interfészadaptere között. Amikor egy teljes programot kiírunk a boot-memóriába, akkor az e programkód alapján kiszámított, végül ugyancsak a boot-memóriába kiírt végleges CRC-kód utasításonkénti egyszeri frissítéssel áll elő a legutolsó utasítás feldolgozásakor (a CRC-kódgenerálás folyamatát lásd 5.4. Függelék). Az ellenőrzőösszeg frissítésének lépéseit tehát célszerű beépíteni abba az állapotsorozatba, amely magukat az utasításokat a boot-memóriába írja.

- Egyetlen bájt elhelyezése vagy kiolvasása a boot-memóriából: A boot-memória interfészadapteréhez ugyancsak szinkron kézfogásos tranzakciókkal férhetünk hozzá (lásd 2-57. ábra), ráadásul ez az interfész egy hibajelet is tartalmaz, amit minden ilyen tranzakció után ellenőriznünk kell.

A 2-65. ábra állapotsorozata egyetlen utasítást ír be a *CPU* belső utasításmemóriájába. Mielőtt az állapotgép „főprogramját” képviselő vezérlési állapotok bármelyikéből a *write_instruction_imem_1* állapotba irányítanánk a végrehajtást, az utasításmemória írásához tartozó címet a *mar (**m**emory **a**ddress **r**egister)*, az adatot pedig az *mdr (**m**emory **d**ata **r**egister)* regiszterbe kell írnunk. A szubrutinszerű állapotsorozaton belül tehát a cím és az adat forrása a hívás kontextusától függetlenül mindig a *mar/mdr* regiszterpár,[53] ezek értékeit másoljuk át a *BOOTSTRAP* modulnak a *CPU* utasításmemóriájához tartozó interfészére *(imem_waddr, imem_din)*. Miután meggyőződtünk róla, hogy a *CPU* utasításmemóriája készen áll új adat beírására (2-65. ábra: 5. sor), az *imem_wstrobe* jelen előállított, a szinkron kézfogásos interfész-specifikációnak megfelelően egyetlen órajelciklus időtartamú pozitív impulzussal kezdeményezzük magát a beírást.

A 2-65. ábra 10. sorában láthatjuk a szubrutinszerű állapotsorozatok legfontosabb sajátosságát; az állapotsorozat végén – ahelyett, hogy a szokott módon egy konkrét állapotba irányítanánk

[53] Visszautalva a szubrutinszerű állapotsorozatok és a programozási nyelvek szubrutinfogalmának analógiájára; a *mar* és *mdr* regiszterek azt a szerepet töltik be a szubrutinszerű állapotsorozatok működésében, amit az assembly nyelvek szubrutinjainak vagy a magas szintű programozási nyelvek függvényeinek paraméterei.

át a végrehajtást – az állapotregiszterbe egy másodlagos állapotregiszter *(return_state_main)* értékét töltjük be. Így a szubrutinszerű állapotsorozatból való „visszatérésként" az állapotgép abban a vezérlési állapotban folytatja a végrehajtást, amit a szubrutinszerű állapotsorozatba való belépés előtt, vagyis a „szubrutin hívásakor" kijelöltünk. A *return_state_main* regiszter a fő állapotregiszterrel *(state_main)* megegyező típusú, a két VHDL jel *(signal)* értékkészlete tehát azonos, bármelyik állapot lehet „visszatérési állapot".[54]

VHDL

```
when write_instruction_imem_1 => -- imaw: instruction memory address width
                                 imem_waddr <= mar(imaw-1 downto 0);
                                 imem_din <= mdr;

                                 if ( imem_rdy2wr = '1' ) then
                                   imem_wstrobe <= '1'; state_main <= write_instruction_imem_2;
                                 end if;

when write_instruction_imem_2 => imem_wstrobe <= '0';
                                 state_main <= return_state_main;
```

2-65. ábra. *A MAIN CTRL FSM állapotgép VHDL-leírása (2)*

A 2-66. ábra azt a szubrutinszerű állapotsorozatot mutatja be, amely egy teljes utasítást helyez el a boot-memóriában, majd a kiírt utasítást a CRC-számító IP-magnak átadva a lokálisan kiszámított CRC-kódot is frissíti. A boot-memória írásához tartozó címet és adatot ez az állapotsorozat is rendre a *mar* és *mdr* regiszterekben veszi át a hívó állapottól. A 2-2. táblázatban láthatjuk, hogy a boot-memóriákban az utasítások bájtjai nem a nulladik címtől kezdődően foglalnak helyet; az első öt rekeszt a programkód egészével kapcsolatos metaadatok számára tartjuk fenn. További komplikációt jelent, hogy az utasítások 16-bitesek, a boot-memóriák viszont bájt-szervezésűek, tehát egy utasítást két részre bontva tudunk csak eltárolni. Ahhoz, hogy a *BOOTSTRAP* modul legfelsőbb szintű algoritmusainak vezérlési állapotai minél intuitívabb, jól átlátható formában kifejezhetőek legyenek, a boot-memóriák írásáért felelős szubrutinszerű állapotsorozat a címszámítást maga végzi el. A *mar* regiszterbe így betölthetjük egy, a teljes programkód összes utasításának boot-memóriába való írása során iterátorként használt, nullától indított számláló értékét (lásd később: *instr_counter*), amely alapján a boot-memória interfészadapterének címbemeneteire juttatandó címinformációt *(bootmem_waddr)* a szubrutinszerű állapotsorozat maga számítja ki (2-66. ábra: 2–5. sor és 24–27. sor). A *write_instruction_bootmem_1...7* állapotok által megvalósított írás műveletekhez előbb az *mdr* regiszter alsó (2-66. ábra: 7. sor), majd a felső (2-66. ábra: 29. sor) fele szolgáltatja magát az adatot. A 20. és 44. sorban azt is láthatjuk, hogy a boot-memória interfészadapterével elvégzett szinkron kézfogásos tranzakció végén ellenőrizni kell a *bootmem_ifa_error_flag* állapotjelző értékét. Hiba esetén a *main_bootmem_error* állapotjelző magas szintre állításával a teljes boot-folyamatot sikertelennek minősítjük, és a *MAIN CTRL FSM* állapotgépet *error* állapotba irányítjuk, amelyből – a PPP-dekódoló állapotgépnél látottakhoz hasonlóan – már csak a *reset* jelent kiutat. A *main_bootmem_error* állapotjelző a 2-61. ábrán látható *OR* kapun keresztül magas szintre húzza a *BOOTSTRAP* modul *firmware_update_failed* kimenetét is.

Az *mdr* regiszterben átvett utasítás két bájtjának boot-memóriába írása után a *calculate_crc_for_storing_1...2* állapotokban az *mdr* regiszter tartalmát átadjuk a *CRC* modulnak is (2-66. ábra:

[54] Sok állapotot tartalmazó, nagyméretű állapotregisztert tartalmazó esetekben pazarlásnak tűnhet a visszatérési állapotok miatt egy teljes méretű, másodlagos szerepű állapotregisztert létrehozni. Szerencsére a logikaiszintézis-algoritmusok elég okosak ahhoz, hogy felismerjék; a másodlagos állapotregiszterbe csak kisszámú konkrét értéket töltünk be a működés során (ezek a ténylegesen visszatérési állapotként használt állapotkódok), a *return_state_main* regisztert fizikailag megvalósító bináris vektor méretét tehát képesek minimalizálni.

VHDL

```
when write_instruction_bootmem_1 =>
  bootmem_waddr <= std_logic_vector(
                     shift_left(unsigned(mar), 1)
                     + 5
                   );

  bootmem_din <= mdr(7 downto 0);
  bootmem_wstrobe <= '1';
  state_main <= write_instruction_bootmem_2;

when write_instruction_bootmem_2 =>
  bootmem_wstrobe <= '0';
  state_main <= write_instruction_bootmem_3;

when write_instruction_bootmem_3 =>
  state_main <= write_instruction_bootmem_4;

when write_instruction_bootmem_4 =>
  if ( bootmem_rdy2wr = '1' ) then
   if ( bootmem_ifa_error_flag = '1' ) then
     main_bootmem_error <= '1';
     state_main <= error;
   else
     bootmem_waddr <= std_logic_vector(
                        unsigned(bootmem_waddr)
                        + 1
                      );

     bootmem_din <= mdr(15 downto 8);
     bootmem_wstrobe <= '1';
     state_main <= write_instruction_bootmem_5;
   end if;
  end if;

```

VHDL (folyt.)

```
when write_instruction_bootmem_5 =>
  bootmem_wstrobe <= '0';
  state_main <= write_instruction_bootmem_6;

when write_instruction_bootmem_6 =>
  state_main <= write_instruction_bootmem_7;

when write_instruction_bootmem_7 =>
  if ( bootmem_rdy2wr = '1' ) then
   if ( bootmem_ifa_error_flag = '1' ) then
     main_bootmem_error <= '1';
     state_main <= error;
   else
     state_main <= write_instruction_bootmem_8;
   end if;
  end if;

when write_instruction_bootmem_8 =>
  chunk_2_crc <= mdr;
  request_crc <= '1';
  state_main <= calculate_crc_for_storing_1;

when calculate_crc_for_storing_1 =>
  state_main <= calculate_crc_for_storing_2;

when calculate_crc_for_storing_2 =>
  if ( crc_ready = '1' ) then
    request_crc <= '0';
    state_main <= return_state_main;
  end if;
```

2-66. ábra. *A MAIN CTRL FSM állapotgép VHDL-leírása (3)*

53. sor). Az e modul szerepét betöltő, ugyancsak szinkron kézfogásos interfésszel rendelkező *crc_chunk_feed_wrp* IP-mag a teljes programkódra vonatkozó CRC-számításnak az aktuális részeredményét a *checksum* kimenetén állítja elő. Megvizsgálva a *CRC* modul belső működését (lásd 4.5.4. pont) láthatjuk, hogy ezt a kimenetet egy sor belső regiszter hajtja meg, amelyek értéke garantáltan nem változik meg mindaddig, amíg a modul felhasználói interfészén újabb számítás elvégzését nem kezdeményezzük, vagyis az IP kimenete önmagában is használható a CRC-számítás

VHDL

```
when write_byte_bootmem_1 =>
  bootmem_wstrobe <= '1';
  state_main <= write_byte_bootmem_2;

when write_byte_bootmem_2 =>
  bootmem_wstrobe <= '0';
  state_main <= write_byte_bootmem_3;

when write_byte_bootmem_3 =>
  state_main <= write_byte_bootmem_4;

when write_byte_bootmem_4 =>
  if ( bootmem_rdy2wr = '1' ) then
   if ( bootmem_ifa_error_flag = '1' ) then
     main_bootmem_error <= '1';
     state_main <= error;
   else state_main <= return_state_main;
   end if;
  end if;

```

VHDL (folyt.)

```
when read_byte_bootmem_1 =>
  bootmem_rstrobe <= '1';
  state_main <= read_byte_bootmem_2;

when read_byte_bootmem_2 =>
  bootmem_rstrobe <= '0';
  state_main <= read_byte_bootmem_3;

when read_byte_bootmem_3 =>
  state_main <= read_byte_bootmem_4;

when read_byte_bootmem_4 =>
  if ( bootmem_rdy2rd = '1' ) then
   if ( bootmem_ifa_error_flag = '1' ) then
     main_bootmem_error <= '1';
     state_main <= error;
   else state_main <= return_state_main;
   end if;
  end if;
```

2-67. ábra. *A MAIN CTRL FSM állapotgép VHDL-leírása (4)*

részeredményének tárolására a következő utasítás feldolgozásáig. A *checksum* kimenet értékét tehát nem kell sehová átmásolnunk, a szubrutinszerű állapotsorozat végén egyszerűen visszalépünk a hívó állapot által kijelölt visszatérési állapotba (2-66. ábra: 63. sor).

A 2-67. ábra két további szubrutinszerű állapotsorozatot mutat be. A bal oldali vezérlési állapotok egy bájtot írnak be a boot-memóriába, a jobb oldaliak pedig ugyancsak egyetlen bájtot olvasnak ki onnan. A korábbiaktól eltérően ezek az állapotsorozatok nem használják a *mar* és *mdr* regisztereket, hívásuk előtt tehát a hívó állapotnak a boot-memória interfészén közvetlenül el kell helyeznie a felülírandó vagy kiolvasandó rekesz címét *(bootmem_waddr, bootmem_raddr)* és írás esetén az adatot *(bootmem_din)*.

A szubrutinszerű állapotsorozatok nagy segítséget jelentenek majd a *BOOTSTRAP* modul szolgáltatásait megvalósító, a *MAIN CTRL FSM* állapotgép „főprogramját" jelentő vezérlési állapotok konstrukciója során.[55] A továbbiakban ezeknek az állapotoknak a részleteit tekintjük át.

Ahogy az a 2-64. ábra kódrészletének 8–9. soraiban is jól látható, a boot-memóriának és a *CPU* utasításmemóriájának UART-on keresztüli inicializálása egyaránt a PPP-dekóder által előkészített *INSTR. FIFO* olvasásával kezdődik. A 2-68. ábra az e folyamathoz tartozó vezérlési állapotok részleteit szemlélteti. A FIFO kiolvasása előtt természetesen ellenőrizzük, hogy abban van-e adat (2-68. ábra: 1. sor). A FIFO töltődésének ütemét alapvetően az UART-link bitsebessége határozza meg, az ürítés üteme pedig a *BOOTSTRAP* modul órajelének frekvenciájától függ. Mivel mindkettő a toplevel integrátor által meghatározott paraméter, ezért nem tudhatjuk, hogy az üres FIFO azért üres, mert a PPP-dekóder még nem töltötte bele a következő bájtot, vagy azért, mert már az összes adatot kiolvastuk belőle és új bájt érkezésére nem is kell számítani. Ne feledjük, a FIFO-ban lévő bájtfolyamba már semmilyen módon nincs belekódolva a program mérete, az csak magukat az utasításokat és az azokból képezett ellenőrzőösszegeket tárolja. Ezért van szükség a *packet_decode_done* állapotjelzőre, amin keresztül a PPP-dekóder jelezni tudja a központi vezérlő állapotgépnek, hogy a FIFO-ba több adat már nem érkezik.

A PPP-dekóder a bejövő bájtfolyam dekódolása során a bájtok sorrendjén nem változtat, tehát a FIFO-ban a 2-59. ábrán látható sorrendben találjuk meg az egyes utasítások alsó és felső bájtjait, valamint az azok összegeként kiszámított ellenőrzőösszeget. A bájt-hármasok elkülönítésére az állapotgépben létrehozunk egy belső számlálót *(byte_counter)*, amit minden bájt kiolvasásakor eggyel növelünk. A számláló értékétől függ, hogy az adott bájttal mit kezdünk; az utasítás alsó bájtját az ugyancsak belső regiszterként deklarált *recent_instr* regiszter alsó felébe, a felső bájtját annak felső felébe töltjük be (2-68. ábra: 13. és 16. sor). Az ellenőrzőösszeg kiolvasásakor pedig e két bájt összegét helyben is kiszámítjuk és eltároljuk a *local_cs* regiszterben, hogy azután azt a *check_cs* állapotban össze tudjuk hasonlítani a FIFO-ból kiolvasott értékkel (2-68. ábra: 27. sor). Ha a két összeg nem egyezik, akkor a *main_cs_error* kimenetet magas szintre állítjuk és az *error* állapotba lépünk át, vagyis a bootstrap-folyamat megszakad. A *main_cs_error* jelet a toplevel modulban a korábban említettekkel együtt egy közös *OR* kapuba kötjük be, tehát ez a hibajelenség

[55] A szubrutinszerű állapotsorozatokkal kapcsolatban érdemes megjegyezni, hogy azok különálló állapotgépekként is megvalósíthatók, ami bizonyos esetekben áttekinthetőbb HDL-leírásokhoz vezethet. A kisebb méretű, különálló állapotgépek létrehozását leginkább olyan esetekben érdemes mérlegelni, amikor a különböző szubrutinszerű állapotsorozatok nem használnak közös erőforrásokat, például nem lépnek kölcsönhatásba ugyanazon külső interfésszel vagy belső almodullal. Ennek magyarázata az, hogy a megosztott erőforrások jeleihez való hozzáférés a különálló állapotgépeket alkalmazó megoldás esetén körülményessé válhat, hiszen egy-egy jelnek csak egy meghajtója lehet, a különálló állapotgépek kimeneteit tehát multiplexálni kell. A beiktatott multiplexerek pedig többlet vezérlőjelek bevezetését is szükségessé teszik, amelyek az összevont, szubrutinszerű állapotsorozatokat tartalmazó megoldásban egyáltalán nem jelennek meg.

VHDL

```
1  when pop_fifo => if ( iff_empty = '1' ) then
2
3                   if ( packet_decode_done = '1' ) then -- no more data from packet decoder...
4                     state_main <= pop_done_1;
5                   end if;
6
7                 else pop_iff <= '1'; state_main <= get_fifo_data_1;
8                 end if;
9
10 when get_fifo_data_1 => pop_iff <= '0'; state_main <= get_fifo_data_2;
11
12 when get_fifo_data_2 => case byte_counter is
13                         when 0 => recent_instr(7 downto 0) <= data_from_iff;
14                                   byte_counter <= byte_counter + 1;
15                                   state_main <= pop_fifo;
16                         when 1 => recent_instr(15 downto 8) <= data_from_iff;
17                                   byte_counter <= byte_counter + 1;
18                                   state_main <= pop_fifo;
19                         when 2 => local_cs <= std_logic_vector(
20                                               unsigned(recent_instr(15 downto 8)) +
21                                               unsigned(recent_instr(7 downto 0))
22                                              );
23                                   byte_counter <= 0;
24                                   state_main <= check_cs;
25                        end case;
26
27 when check_cs => if ( local_cs /= data_from_iff ) then main_cs_error <= '1'; state_main <= error;
28                 else
29
30                   if ( instr_counter < 2**imaw ) then -- imaw: instruction memory address width
31
32                     instr_counter <= instr_counter + 1;
33                     mar <=  std_logic_vector(
34                               to_unsigned(instr_counter, bmaw) -- bmaw: boot memory address width
35                             );
36                     mdr <= recent_instr;
37                     return_state_main <= pop_fifo;
38
39                     if ( boot_mode = "00" ) then
40                       state_main <= write_instruction_bootmem_1; -- EXT ROM programming
41                     else state_main <= write_instruction_imem_1; -- direct CPU IMEM programming
42                     end if;
43
44                   else -- too many instructions received...
45                     main_too_many_instructions_error <= '1'; state_main <= error;
46                   end if;
47
48                 end if;
```

2-68. ábra. *A MAIN CTRL FSM állapotgép VHDL-leírása (5)*

is magas szintet idéz elő a *BOOTSTRAP* modul *firmware_update_failed* kimenetén. Ha a helyben kiszámított ellenőrzőösszeg megegyezik azzal, amit a FIFO-ból kiolvastunk, akkor folytatódhat a feldolgozás; a *recent_instr* regiszterben összeállított utasításkódot a *boot_mode* bemenet által meghatározott üzemmódtól függően vagy a boot-memóriába, vagy közvetlenül a *CPU* belső utasításmemóriájába kell beírnunk. A FIFO-ból kiolvasott teljes, 16-bites utasítások számát az *instr_counter* regiszter segítségével követjük nyomon. A 30. sorban azt látjuk, hogy a központi vezérlő állapotgép ellenőrzi, hogy a beérkező utasítások száma összhangban van-e a toplevel integrátor által beállított *imaw (**i**nstruction **m**emory **a**ddress **w**idth)* paraméter értékével. Ha a *BOOTSTRAP* modul az UART-linken a programtárak kapacitását túllépő mennyiségű adatot kap, akkor a hibát a *main_too_many_instructions_error* állapotjelzőn keresztül jelzi. Ez a hibajelenség is a teljes folyamat leállítását és a *firmware_update_failed* kimenet magas szintre húzását vonja maga után. A 2-68. ábra 32–42. soraiban láthatjuk a szubrutinszerű állapotsorozatok használatának módját a „főprogramot” jelentő állapotok szempontjából. Ezen a ponton a *recent_instr* regiszterben rendelkezésünkre

áll a FIFO-ból legutóbb kiolvasott utasítás, ezt szeretnénk elhelyezni a boot-memóriában vagy a *CPU* utasításmemóriájában. E két memóriamodul írására egy-egy szubrutinszerű állapotsorozatot definiáltunk, amelyek belépési pontjait rendre a *write_instruction_bootmem_1* és a *write_instruction_imem_1* állapotok képviselik. Mindkét szubrutinszerű állapotsorozat a *mar* és *mdr* regisztereken keresztül veszi át a hívó állapottól, hogy milyen címre milyen adatot kell elhelyezni. Ennek megfelelően híváskor a *mar* regiszterbe betöltjük az utasításszámláló értékét *(instr_counter)*, az *mdr* regiszterbe pedig magát az utasítást, majd a visszatérési állapot kijelölése (2-68. ábra: 37. sor) után a *boot_mode* bemenet értékétől függően átlépünk az egyik szubrutinszerű állapotsorozatba. Az utasítás elhelyezése után folytatnunk kell a FIFO kiolvasását, tehát a visszatérési pont mindkét esetben a *pop_fifo* állapot lesz. A FIFO végső kiürülésekor, tehát amikor az *empty* jelzéssel együtt a PPP-dekóder állapotgép *packet_decode_done* kimenete is magas szintű lesz (2-68. ábra: 3. sor), akkor a *pop_done_1* állapotban folytatjuk a műveletvégzést, ami ezen a ponton ismét kettéágazik attól függően, hogy épp melyik programtárat töltöttük fel (lásd 2-69. ábra).

VHDL

```
1  when pop_done_1 =>
2    if ( boot_mode = "00" ) then -- EXT ROM programming
3      |
4      | -- close CRC calculation...
5      | chunk_2_crc <= X"0000"; last_chunk_crc <= '1';
6      | request_crc <= '1'; state_main <= pop_done_2;
7      |
8    else -- direct CPU IMEM programming
9      firmware_update_ok <= '1';
10     cpu_firmware_reset <= '0';
11   end if;
12
```

VHDL (folyt.)

```
13 when pop_done_2 =>
14   state_main <= pop_done_3;
15
16 when pop_done_3 =>
17   if ( crc_ready = '1' ) then
18     request_crc <= '0';
19     state_main <= write_metadata_into_rom_1;
20   end if;
```

2-69. ábra. *A MAIN CTRL FSM állapotgép VHDL-leírása (6)*

Ha a *CPU* belső utasításmemóriáját töltöttük fel az UART-on érkező programkóddal, akkor a *BOOTSTRAP* modulnak már csak annyi dolga van, hogy a *firmware_update_ok* kimenet magas szintre állításával jelezze a folyamat sikeres befejeződését és a *cpu_firmware_reset* kimenet alacsony szintre állításával engedélyezze a *CPU* számára a program futtatását (2-69. ábra: 8–11. sor). Ha az UART-on érkező programkódot a boot-memóriában helyeztük el, akkor a helyzet ennél kicsit bonyolultabb; néhány extra lépést még el kell végeznünk, mielőtt a folyamatot befejezettnek tekinthetjük. A műveletvégzés ez esetben a *pop_done_2* állapotban folytatódik.

A boot-memóriában a programkódon felül az ahhoz tartozó metaadatokat is el kell helyeznünk annak érdekében, hogy később a *CPU* belső utasításmemóriáját ebből a külső tárolóból is fel tudjuk tölteni. Ahogy az a 2-2. táblázatban is látható, a metaadatok a programkód méretét, az ezt lefedő egyszerű ellenőrzőösszeget, valamint a teljes programkódot lefedő CRC-kódot jelentik. Visszatekintve a 2-66. ábrán látható szubrutinszerű állapotsorozatra, láthatjuk, hogy a programkódot lefedő CRC-kód kiszámítása az utasítások boot-memóriában való elhelyezésével párhuzamosan folyt, a *CRC* modul *checksum* kimenetén tehát rendelkezésre áll maga a CRC-kód. A *CRC* modul működéséből adódóan ahhoz, hogy a generált kód valóban a teljes programot lefedje, egy végső, lezáró számítási lépést kell végrehajtanunk, amelyben a *CRC* modulnak küldött bemenő adattöredék összes bitjét nullára állítjuk és a *last_chunk* vezérlőbemenet magas szintre állításával magának a *CRC* modulnak is a tudtára adjuk, hogy ez lesz a legutolsó adattöredék (2-69. ábra: 5-6. sor, vö. 4-223. ábra). A végleges CRC-kód kiszámítása után a műveletvégzést a *write_metadata_into_rom_1* állapotban folytatjuk.

VHDL

```vhdl
when write_metadata_into_rom_1 => -- write...
                                  -- - no. instructions into ROM[0...2]
                                  -- - checksum for no. instructions into ROM[3]
                                  -- - instructions CRC into ROM[4]

          bootmem_waddr <= std_logic_vector(to_unsigned(0, bmaw));                        -- write ROM[0]
          bootmem_din <= std_logic_vector(to_unsigned(instr_counter, 24)(7 downto 0));
          return_state_main <= write_metadata_into_rom_2;
          state_main <= write_byte_bootmem_1;

when write_metadata_into_rom_2 =>
          bootmem_waddr <= std_logic_vector(to_unsigned(1, bmaw));                        -- write ROM[1]
          bootmem_din <= std_logic_vector(to_unsigned(instr_counter, 24)(15 downto 8));
          return_state_main <= write_metadata_into_rom_3;
          state_main <= write_byte_bootmem_1;

when write_metadata_into_rom_3 =>
          bootmem_waddr <= std_logic_vector(to_unsigned(2, bmaw));                        -- write ROM[2]
          bootmem_din <= std_logic_vector(to_unsigned(instr_counter, 24)(23 downto 16));
          return_state_main <= write_metadata_into_rom_4;
          state_main <= write_byte_bootmem_1;

when write_metadata_into_rom_4 =>
          bootmem_waddr <= std_logic_vector(to_unsigned(3, bmaw));                        -- write ROM[3]
          bootmem_din <= std_logic_vector(
                           | to_unsigned(instr_counter, 24)(7 downto 0) +
                           | to_unsigned(instr_counter, 24)(15 downto 8) +
                           | to_unsigned(instr_counter, 24)(23 downto 16)
                         );
          return_state_main <= write_metadata_into_rom_5;
          state_main <= write_byte_bootmem_1;

when write_metadata_into_rom_5 =>
          bootmem_waddr <= std_logic_vector(to_unsigned(4, bmaw));                        -- write ROM[4]
          bootmem_din <= crc_checksum;
          return_state_main <= rom_write_done;
          state_main <= write_byte_bootmem_1;

when rom_write_done => firmware_update_ok <= '1';
```

2-70. ábra. *A MAIN CTRL FSM állapotgép VHDL-leírása (7)*

A *write_metadata_into_rom_1...5* állapotokban a *write_byte_bootmem_X* szubrutinszerű állapotsorozat „szolgáltatását” igénybe véve helyezünk el öt bájtot a boot-memóriában a 2-2. táblázat előírásainak megfelelően. Figyeljük meg, hogy a *mar* és *mdr* regisztereket ezúttal nem használjuk, a *write_byte_bootmem_1* állapotba való belépés előtt közvetlenül a boot-memória interfészadapteréhez kapcsolt *bootmem_waddr* és *bootmem_din* jeleket állítjuk be. A boot-memória 0...2 címeire a korábban eltárolt program méretét írjuk, amit az utasításszámláló regiszter *(instr_counter)* tárol. E regiszter 24-bites, amit három részre bontva tárolunk el a bájt-szervezésű külső tárban. A *write_metadata_4* állapotban kerül sor a program méretét leíró három bájtot lefedő ellenőrzőösszeg kiszámítására, amit a boot-memória 3. címére írunk ki. Végül a teljes programkódot lefedő CRC-kódot a *CRC* modul *checksum* kimenetéről olvassuk le és írjuk ki a boot-memória 4. címére. A folyamatot ezúttal is a *firmware_update_ok* kimenet magas szintre állításával zárjuk le. A *CPU firmware_reset* bemenetét ezúttal nem nullázzuk, hiszen az éppen befejezett bootstrap-folyamat a boot-memória feltöltésére vonatkozott, amelynek a programfuttatás elindítása – a rendszertervben rögzített elvárások alapján – nem feladata.

A *CPU* belső utasításmemóriájának a boot-memóriából való inicializálása a *BOOTSTRAP* modul legösszetettebb szolgáltatása. Az ennek megvalósításához szükséges vezérlési állapotokat a 2-71. és 2-73. ábrák mutatják be.

VHDL

```
when boot_1 => -- Read no. instructions, check checksum

               bootmem_raddr <= std_logic_vector(to_unsigned(0, bmaw));                    -- read ROM[0]
               return_state_main <= boot_2;
               state_main <= read_byte_bootmem_1;

when boot_2 => no_instructions(7 downto 0) <= unsigned(bootmem_dout);
               local_cs <= std_logic_vector(unsigned(bootmem_dout));
               bootmem_raddr <= std_logic_vector(to_unsigned(1, bmaw));                    -- read ROM[1]
               return_state_main <= boot_3;
               state_main <= read_byte_bootmem_1;

when boot_3 => no_instructions(15 downto 8) <= unsigned(bootmem_dout);
               local_cs <= std_logic_vector(unsigned(local_cs) + unsigned(bootmem_dout));
               bootmem_raddr <= std_logic_vector(to_unsigned(2, bmaw));                    -- read ROM[2]
               return_state_main <= boot_4;
               state_main <= read_byte_bootmem_1;

when boot_4 => no_instructions(23 downto 16) <= unsigned(bootmem_dout);
               local_cs <= std_logic_vector(unsigned(local_cs) + unsigned(bootmem_dout));
               bootmem_raddr <= std_logic_vector(to_unsigned(3, bmaw));                    -- read ROM[3]
               return_state_main <= boot_5;
               state_main <= read_byte_bootmem_1;

when boot_5 => if ( bootmem_dout /= local_cs ) then
                 main_no_instructions_cs_error <= '1'; state_main <= error;
               else instr_counter <= 0; state_main <= boot_6;
               end if;
```

2-71. ábra. *A MAIN CTRL FSM állapotgép VHDL-leírása (8)*

A teljes program boot-memóriából való kiolvasásának első lépése, hogy a 2-2. táblázatban rögzített címkiosztásnak megfelelően eltárolt metaadatok alapján megállapítjuk a kiolvasandó program méretét. A boot-memória bájtonkénti olvasásához a *read_byte_bootmem_X* szubrutinszerű állapotsorozatot használjuk. A 0...2 címekről beolvasott bájtokat a *no_instructions* regiszterben fűzzük össze (2-71. ábra: 7. 13. és 19. sor), és a kiolvasással párhuzamosan a *local_cs* regiszterben e bájtok egyszerű összegét is képezzük (2-71. ábra: 8. 14. és 20. sor), majd azt a boot-memória 3. címéről beolvasott, tárolt ellenőrzőösszeggel vetjük össze (2-71. ábra: 25. sor). Ha a két ellenőrzőösszeg nem egyezik meg, akkor a boot-folyamatot a *main_no_instructions_cs_error* állapotjelző magas szintre állításával megszakítjuk. E hibajel a korábbiakhoz hasonlóan a *BOOTSTRAP* modul *firmware_update_failed* kimenetét is magas szintre húzza.

Ha az ellenőrzőösszegek egyeznek, akkor tudhatjuk, hogy a *no_instructions* regiszterben eltárolt érték a kiolvasandó, majd a *CPU* belső utasításmemóriájába betöltendő 16-bites utasítások számát tárolja. A műveletvégzés ekkor maguknak az utasításoknak a kiolvasásával folytatódik a *boot_6* állapotban (lásd 2-72. ábra).

A boot-memória bájtos szervezése és a metaadatok meghatározott címkiosztásnak megfelelő elhelyezése az *instr_counter* utasításszámláló és az általa kijelölt utasítás boot-memóriabeli címei között egy átszámítást tesz szükségessé. Ezt az átszámítást – a *write_instruction_bootmem_X* szubrutinszerű állapotsorozatnál látottakhoz hasonlóan – a kiolvasásnál is el kell végezni (2-72. ábra: 3–5. és 10–12. sor). Mivel a teljes program kiolvasása után az azt lefedő tárolt CRC-kódot egy helyben újra kiszámolt CRC-kóddal kell majd összevetnünk az adatok sértetlenségének ellenőrzése céljából, ezért az egyes utasítások kiolvasása után azokat rögtön átadjuk a *CRC* modulnak (2-72. ábra: 17–18. sor). A CRC-részszámítás elvégzése után annak eredményét ezúttal sem töltjük át a központi vezérlő állapotgépnek egy belső regiszterébe, mivel az ideiglenesen megőrzendő kódot magának a *CRC* modulnak a *cheksum* kimenete eltárolja. Egy-egy utasítás boot-memóriából való kiolvasását annak a *CPU* belső utasításmemóriájába való beírása követi (lásd 2-73. ábra).

VHDL

```
1  when boot_6 => -- Read the instructions...
2
3                 bootmem_raddr <= std_logic_vector(
4                                    shift_left(to_unsigned(instr_counter, bmaw), 1) + 5
5                                  );
6                 return_state_main <= boot_7;
7                 state_main <= read_byte_bootmem_1;
8
9  when boot_7 => recent_instr(7 downto 0) <= bootmem_dout;
10                bootmem_raddr <= std_logic_vector(
11                                   shift_left(to_unsigned(instr_counter, bmaw), 1) + 5 + 1
12                                 );
13                return_state_main <= boot_8;
14                state_main <= read_byte_bootmem_1;
15
16 when boot_8 => recent_instr(15 downto 8) <= bootmem_dout;
17                chunk_2_crc <= bootmem_dout & recent_instr(7 downto 0);
18                request_crc <= '1';
19                state_main <= calculate_crc_for_checking_1;
20
21 when calculate_crc_for_checking_1 => state_main <= calculate_crc_for_checking_2;
22
23 when calculate_crc_for_checking_2 => if ( crc_ready = '1' ) then
24                                        request_crc <= '0'; state_main <= boot_9;
25                                     end if;
```

2-72. ábra. *A MAIN CTRL FSM állapotgép VHDL-leírása (9)*

A *CPU* utasításmemóriája 16-bites bejegyzéseket tárol és a programot a nullás címtől kezdődően töltjük bele, így az *instr_counter* regiszter közvetlenül használható a tároló címzésére (2-73. ábra: 5–7. sor). Az *instr_counter* regiszter értékét értelemszerűen minden beíráskor eggyel növeljük, és a boot-memória olvasásához, valamint a *CPU* utasításmemóriájának írásához tartozó lépéseket addig ismételjük, amíg az *instr_counter* el nem éri a *no_instructions* regiszterben tárolt, a teljes program méretét tároló értéket. Az iterációk során a *CRC* modul segítségével minden utasítás beolvasásakor újra és újra frissítjük a helyben kiszámított CRC-kódot, majd az utolsó utasítás feldolgozása után azt ismét egy lezáró részszámítással véglegesítjük (2-73. ábra: 20–22. sor). A *boot_12*

VHDL

```
1  when boot_9    => -- Write instructions into IMEM,
2                    -- while calculating last CRC
3                    -- Read CRC from ROM and check
4
5    mar <= std_logic_vector(
6             to_unsigned(instr_counter, bmaw)
7           );
8    mdr <= recent_instr;
9    return_state_main <= boot_10;
10   state_main <= write_instruction_imem_1;
11
12 when boot_10 =>
13   if ( instr_counter < no_instructions-1 ) then
14     instr_counter <= instr_counter + 1;
15     state_main <= boot_6;
16   else
17   | instr_counter <= 0;
18   |
19   | -- Close CRC calculation..
20   | chunk_2_crc <= X"0000";
21   | last_chunk_crc <= '1';
22   | request_crc <= '1';
23   | state_main <= boot_11;
24   end if;
25
26 when boot_11 => state_main <= boot_12;
27
```

VHDL (folyt.)

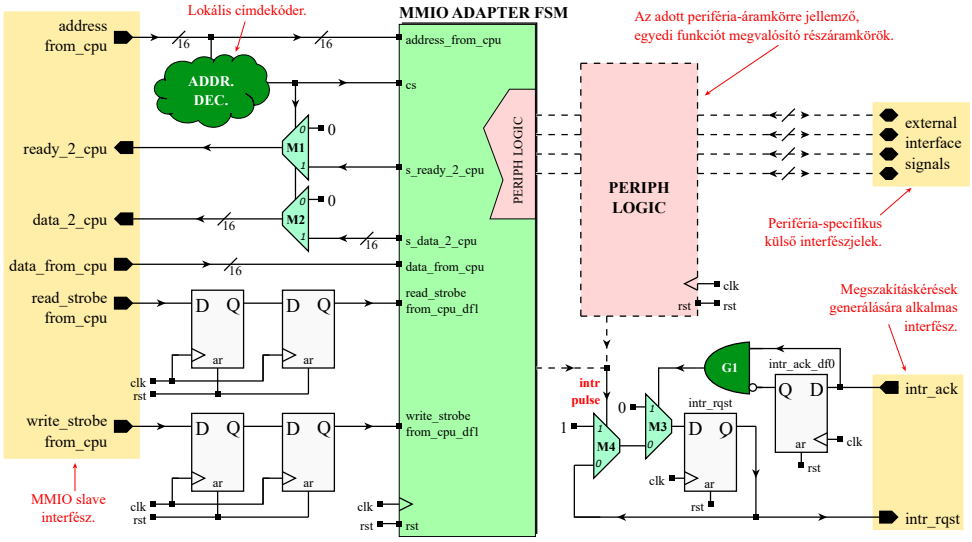
```
28 when boot_12 =>
29   if ( crc_ready = '1' ) then
30   | request_crc <= '0';
31   |
32   | -- Read CRC from ROM and check.
33   | bootmem_raddr <= std_logic_vector(
34   |                     to_unsigned(4, bmaw)
35   |                   );
36   | return_state_main <= boot_13;
37   | state_main <= read_byte_bootmem_1;
38   end if;
39
40 when boot_13 =>
41   if ( crc_checksum /= bootmem_dout ) then
42   | -- CRC error in stored data...
43   | main_crc_error <= '1';
44   | state_main <= error;
45   else state_main <= boot_done;
46   end if;
47
48 when boot_done =>
49   firmware_update_ok <= '1';
50   cpu_firmware_reset <= '0';
```

2-73. ábra. *A MAIN CTRL FSM állapotgép VHDL-leírása (10)*

állapotban újabb bootmemória-olvasást kezdeményezünk; ezúttal a 4. címen eltárolt CRC-kódot olvassuk ki, és azt a helyben kiszámított, a *CRC* modul *checksum* kimenetéről leolvasható CRC-kóddal vetjük össze (2-73. ábra: 41. sor). Ha a két kód eltér, az azt jelenti, hogy a boot-memóriában vagy annak kiolvasása során sérült az adat, a programfuttatás tehát nem indulhat el. Ezt a hibajelenséget a korábbiakkal azonos módon kezeljük; a *main_crc_error* állapotjelzőt magas szintre állítva és a központi vezérlő állapotgépet az *error* állapotba irányítva megszakítjuk a boot-folyamatot. Egyező CRC-kódok esetén a *CPU* belső utasításmemóriájában immár egy helyes programkód van, tehát egyrészt magas szintre állíthatjuk a *BOOTSTRAP* modul *firmware_update_ok* kimenetét, másrészt a *cpu_firmware_reset* kimenet nullázásával engedélyezhetjük a program futtatását (2-73. ábra: *boot_done* állapot).

2.4. Az MMIO periféria-áramkörök szerkezete

Az MMIO-interfészjelek együttese egy, a funkcióbővítő rendszer gerincét képező, csillag topológiájú multi-slave buszrendszer, amelyen keresztül az alapfunkciókat ellátó IP-magok és a bővítőmodulok egységes protokoll szerint kommunikálnak a *CPU*-val. Az e protokollnak való megfelelés kényszere az IP-magok belső felépítésében is hasonlóságokhoz vezet; a 2-74. ábra az MMIO-kompatibilis moduloknak egy általánosított modelljét szemlélteti.



2-74. ábra. *Az MMIO periféria-áramkörök általános szerkezete*

Már a rendszerterv szintjén rögzítettük, hogy a *CPU* és a periféria-áramkörök közötti órajel-szinkronizáló fokozatokat az IP-magok szerves részének tekintjük, nem meglepő tehát, hogy az MMIO-interfész órajel-szinkronizációs megoldását szemléltető 2-7. ábra *FF1...FF4* flip-flopjai a periféria-áramkörök közös szerkezeti sémájában is visszaköszönnek *(read/write_strobe_df0/1)*.[56]

[56] A periféria-áramkör *ready* jelzésének *CPU*-oldali órajel-szinkronizálásáért felelős, a 2-7. ábrán *FF5* és *FF6* azonosítóval jelölt flip-flopokat magában a *CPU*-ban helyeztük el (lásd 2-56. ábra).

VHDL

```
L_MMIO_ADAPTER_FSM: process ( clk, rst ) begin
  if ( rst = '1' ) then
    state <= idle; s_ready_2_cpu <= '0'; s_data_2_cpu <= (others => '0');
    status_reg_3 <= (others => '0'); control_reg_1 <= (others => '0');
    [...] -- reset all outputs and other internal registers...
  elsif ( rising_edge(clk) ) then

    case state is

      when idle => s_ready_2_cpu <= '1';

                   -- Handle MMIO write requests...
                   if ( write_strobe_from_cpu_df1 = '1' and cs = '1' ) then
                     s_ready_2_cpu <= '0';

                     if ( unsigned(address_from_cpu) = << address_1 >> ) then
                       control_reg_1 <= data_from_cpu;
                       state <= wait_for_deassert_strobes;
                     elsif ( unsigned(address_from_cpu) = << address_2 >> ) then
                       state <= address_2_service_state_1;

                     elsif ( [...] ) then [...]

                     else state <= wait_for_deassert_strobes;
                     end if;

                   end if;


                   -- Handle MMIO read requests...
                   if ( read_strobe_from_cpu_df1 = '1' and cs = '1') then
                     s_ready_2_cpu <= '0';

                     if ( unsigned(address_from_cpu) = << address_1 >> ) then
                       s_data_2_cpu <= control_reg_1;
                       state <= wait_for_deassert_strobes;
                     elsif ( unsigned(address_from_cpu) = << address_3 >> ) then
                       s_data_2_cpu <= status_reg_3;
                       state <= wait_for_deassert_strobes;
                     elsif ( unsigned(address_from_cpu) = << address_4 >> ) then
                       state <= address_4_service_state_1;

                     elsif ( [...] ) then [...]

                     else state <= wait_for_deassert_strobes;
                     end if;

                   end if;

      ------------------------------------------------------------------------------------

      when address_2_service_state_1 => -- perform write service assigned to address #2...
                                        [...]

      when address_4_service_state_1 => -- perform read service assigned to address #4...
                                        [...]

      [...] -- additioanl states...

      ------------------------------------------------------------------------------------

      when wait_for_deassert_strobes => if (
                                          read_strobe_from_cpu_df1 = '0' and
                                          write_strobe_from_cpu_df1 = '0'
                                        ) then state <= idle;
                                        end if;

      when others => null;

    end case;

  end if;
end process;
```

2-75. ábra. *Az MMIO-adapter állapotgép vázlata*

Az MMIO-buszrendszernek egy fontos sajátossága az elosztott címdekódolás, amelynek rendszerszintű működéséhez elengedhetetlen, hogy minden periféria-áramkörben jelen legyen egy saját címdekóder-logika. Ez a struktúra egy, a periféria-áramkör saját címeire érzékeny, kombinációs hálózatként megvalósított dekóderből és az annak kimenetével vezérelt két multiplexerből áll (2-74. ábra: *M1*, *M2*). Mindezeknek a közös szerkezeti elemeknek a HDL-modelljét a 2-8. ábra mutatja be.

Az MMIO protokollnak a slave egységek viselkedésére vonatkozó előírásait jól átlátható és könnyen módosítható formában elégíthetjük ki egy állapotgép segítségével. Ennek az állapotgépnek a feladata, hogy az egységes MMIO slave interfész és a periféria-áramkör funkciójának megfelelő, teljesen egyedi belső részáramkörök (2-74. ábra: *PERIPH LOGIC*) között teremtsen kapcsolatot, ezért az állapotgép HDL-leírása részben megegyezik minden periféria-áramkör esetén. Egy ilyen MMIO-adapter állapotgépsablon VHDL-leírását mutatja be a 2-75. ábra.

A HDL-leírásban azt vehetjük észre, hogy az állapotgép az MMIO-buszrendszer slave → master irányú *ready_2_cpu* és *data_2_cpu* jeleit nem közvetlenül, hanem egy-egy belső jelen *(s_ready_2_cpu, s_data_2_cpu)* és a címdekódoló logika részét képező multiplexereken keresztül hajtja meg. Erre azért van szükség, hogy a címdekódoló logika – eleget téve az MMIO-interfész specifikációjának – a lehető leggyorsabban, rögtön a processzor által kiadott cím *(address_from_cpu)* megjelenésekor,[57] az állapotgép működésétől függetlenül is nullába húzhassa ezeket a kimeneteket, amennyiben maga a cím nem esik bele a periféria-áramkör saját címtartományába. Ellenkező esetben, vagyis amikor a *CPU* felől érkező MMIO-kérés a periféria-áramkörnek szól, akkor a címdekóder logika a *cs (chip select)* jel magas szintre állításával mintegy átadja a kimenetek vezérlésének jogát az állapotgépnek; ilyenkor az *s_ready_2_cpu* és *s_data_2_cpu* jelek változásai egyúttal a periféria-áramkör kimenetein is megjelennek.

Az állapotgépben két, minden MMIO periféria-áramkörben megtalálható állapotot hozunk létre, ezek az *idle* és a *wait_for_deassert_strobes*. A további állapotok az adott periféria-áramkörre jellemzők, ezek szerepét később részletezzük. Az *idle* állapotban az *s_ready_2_cpu* jel magas szinten tartása mellett[58] várakozunk írás- vagy olvasáskérések érkezésére (2-75. ábra: 13. és 31. sor).[59] A *write/read_strobe_from_cpu_df1* jelek magas értéke önmagában még nem jelent az adott periféria-áramkörnek szóló kérést, hiszen a *strobe* jelek változásai a double-flop szinkronizáló fokozatokon a *CPU* által kiadott címtől függetlenül keresztüljutnak. Ezért az állapotgépen belül csak akkor tekintjük a szinkronizáló fokozatok kimenetén megjelenő magas értékeket érvényes kéréseknek, ha közben a *cs* magas szintű. Érvényes kérés érkezésekor az *s_ready_2_cpu* jelet alacsony szintre húzzuk (2-75. ábra: 14. és 32. sor, vö. 1-8. és 1-9. ábra: az MMIO-tranzakciók 2. lépése), majd a *CPU* által kiadott cím alapján megállapítjuk, hogy pontosan „mit szeretne a *CPU*"; az egyazon

[57] Az „azonnal" kifejezés csak a zero-delay jellegű RTL-modellek tisztán logikai szimulációjakor értendő szó szerint. A szintetizált áramkörben a cím megjelenése és a kimenetek nullázása között eltelik némi idő a címdekódoló logika kombinációs hálózatának késleltetése miatt.

[58] Ne feledjük, ha az *address_from_cpu* buszon olyan cím van, ami nem az adott periféria-áramkör címtartományába tartozik, akkor a címdekóder *cs* kimenete alacsony szintű, tehát a *ready_2_cpu* kimenet is az, függetlenül attól, hogy az állapotgép milyen értéket ad az *s_ready_2_cpu* jelnek.

[59] Az állapotgépben az íráskérésekre vonatkozó feltételvizsgálat mindig megelőzi az olvasáskérésekre vonatkozót, ami egyidejű írás/olvasás esetén az olvasást előnyben részesítő prioritáskezelést jelentene. Azonban az MMIO-buszrendszeren egyidőben csak egyféle tranzakció kezdeményezésére van lehetőség (a *CPU* vagy a *store* vagy a *load* gépi utasítást hajtja végre), ezért e két művelet közötti prioritás nem értelmezhető (vö. memória jellegű katalógus-áramkörök interfészadaptereinek hozzáférés-vezérlője: 4-95. ábra).

periféria-áramkör címtartományába tartozó különböző címeken kezdeményezett írás- és olvasásműveletek a periféria-áramkör konkrét funkciójától függően mást és mást jelenthetnek, továbbá a kérések feldolgozásának módja is igen változatos lehet. A 2-75. ábrán a legtipikusabb esetekre mutatunk be egy-egy példát:

- Hozzáférés írható-olvasható vezérlőregiszterhez (*address_1*: 2-75. ábra: 16. és 34. sor): Az adott cím írása vagy olvasása legegyszerűbb esetben egy tényleges regiszter tartalmának felülírását vagy kiolvasását jelenti. Az ilyen vezérlőregiszterek rendszerint az adott periféria-áramkör futásidejű konfigurációs beállításait vagy a belső működésükre utaló állapotinformációkat tárolják.
- Csak-olvasható állapotregiszter lekérdezése (*address_3*: 2-75. ábra: 37. sor): Az írható-olvasható regiszterek esetéhez hasonlóan ez a tranzakciótípus is egy fizikailag is jelenlévő regiszterre vonatkozik, annak értékét kérdezi le. A csak-olvasható regiszterek esetén azonban az írásművelet logikailag nem értelmezhető; ilyen lehet pl. egy MMIO periféria-áramkörként megvalósított FIFO-tároló *full* és *empty* állapotjelzőit tartalmazó regiszter. Mivel az MMIO-írásokat kezelő kódrészletben (2-75. ábra: 13–27. sor) nincs olyan feltételvizsgálat, ami az adott cím esetén igazra értékelődne ki, ezért az állapotgép – a címvizsgálatot lezáró *else* ág következtében – az *idle* állapotból azonnal a *wait_for_deassert_strobes* állapotba lép.[60]
- Master → slave irányú adatforgalommal járó szolgáltatás végrehajtásának kezdeményezése adott címen végrehajtott írással (*address_2*: 2-75. ábra: 19. sor): Az MMIO írásművelet ebben az esetben nem egy regiszter értékének felülírását, hanem egy összetett, akár sok vezérlési állapotot igénylő műveletsor végrehajtását kezdeményezi. A *CPU* által a *data_from_cpu* jelen küldött adat[61] – igény szerint – felhasználható a szolgáltatáskérés paramétereként. Erre a tranzakciótípusra jó példa egy soros kommunikációs interfész-vezérlő; egy MMIO UART periféria-áramkör elkészíthető úgy, hogy egy adatküldés egy adott címen végrehajtott MMIO-írással legyen kezdeményezhető (lásd 2.8.1. pont: *address_rxtx_data*). Az UART párhuzamos-soros átalakítóval való kommunikáció több vezérlési állapotot is igényelhet (lásd pl. a szinkron kézfogásos interfészt). Az összetett szolgáltatást levezénylő állapotsorozatnak mindenképp a *wait_for_deassert_strobes* állapotban kell véget érnie, ami gondoskodik az MMIO-tranzakció megfelelő lezárásáról (vö. 18. és 19. ábra: az MMIO-tranzakciók 6. lépése).
- Slave → master irányú adatforgalommal járó szolgáltatás végrehajtásának kezdeményezése adott címen végrehajtott olvasással (*address_4*: 2-75. ábra: 40. sor): Az előző pontban részletezett megoldásnak a fordított irányú adatáramlást lehetővé tevő párja. Az összetett szolgáltatást megvalósító állapotsorozatnak ezúttal is a *wait_for_deassert_strobes* állapotban kell befejeződnie úgy, hogy a periféria-áramkör által a *CPU* felé küldött adatnak az *s_data_2_cpu* vektorba kell kerülnie.[62]

[60] Assembly programozási szempontból ez azt jelenti, hogy a *CPU* által éppen végrehajtott *store* utasításnak semmi hatása nincs.

[61] A *CPU* által éppen végrehajtott *store* utasításnak a regisztertömbből kiolvasott bejegyzése: 2-1. táblázat: *RF[IR[11:8]]*.

[62] Az adatbejegyzést a *CPU* által éppen végrehajtott *load* utasítás a regisztertömbbe írja be: 2-1. táblázat: *RF[IR[11:8]]*.

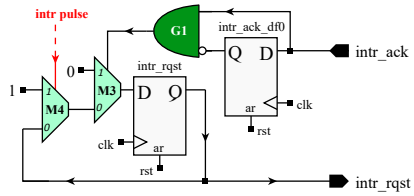
A fentiek alapján elmondható, hogy minden MMIO periféria-áramkörhöz definiálnunk kell egy ún. „címtérképet”, ami tartalmazza az adott periféria-áramkörhöz rendelt címeken végrehajtott MMIO írás- és olvasásműveletek jelentését. Ez a címtérkép lesz a periféria-áramkör dokumentációja, amely nélkülözhetetlen a *CPU* programjának elkészítéséhez. Fontos kiemelni, hogy az MMIO periféria-áramkörök HDL-modelljeiben magukat az MMIO-címeket (2-75. ábra: *address_X*) szintézis-paraméterként célszerű kezelni (VHDL *generic*, Verilog *parameter*), lehetővé téve a top-level integrátor számára a teljes soft-processzoros rendszerre vonatkozó periféria-készlet címkiosztásának testreszabását.

A *CPU*-val együttműködő periféria-áramkörök többsége esetén szükség van a megszakítási rendszerhez való kapcsolódásra is. Ehhez az adott periféria-áramkörbe be kell építenünk a 2-74. ábra jobb alsó sarkában látható *intr_rqst* flip-flopot és az annak bemenetét meghajtó *M3* és *M4* multiplexereket, valamint az *intr_ack_df0* flip-flop és a *G1 AND* kapu által képezett felfutóél-detektort. Ezek biztosítják a *CPU intr_rqst* és *intr_ack* jelpárjának 2-53. ábrán látható viselkedésével való kompatibilitást.[63]

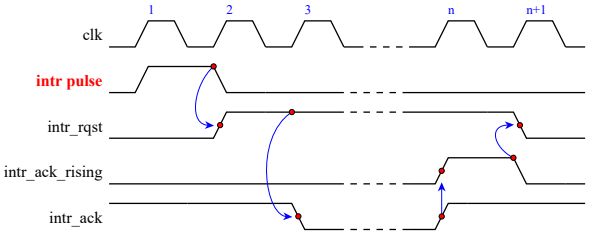
VHDL

```
1  L_INTR_GENERATION: process ( clk, rst ) begin
2    if ( rst = '1' ) then
3      intr_ack_df0 <= '0'; intr_rqst <= '0';
4    elsif ( rising_edge(clk) ) then
5
6      intr_ack_df0 <= intr_ack;
7
8      if ( intr_ack_rising = '1' ) then
9        intr_rqst <= '0';
10     elsif ( << intr pulse >> = '1' ) then
11       intr_rqst <= '1';
12     end if;
13
14   end if;
15 end process;
16 intr_ack_rising <= intr_ack and not intr_ack_df0;
```

Áramköri struktúra



Hullámforma



2-76. ábra. *Az MMIO periféria-áramkörök megszakításkérést generáló logikája*

A periféria-áramkör belsejében így nincs más dolgunk, mint egy, a periféria-áramkör lokális órajelének egyetlen periódusáig magas értékű impulzust generálni az *intr pulse* jelen minden olyan esemény bekövetkezésekor, amikor a *CPU* felé egy megszakításkérést szeretnénk küldeni (pl. egy FIFO kiürülése, egy UART adatkeret érkezése). A periféria-áramkörökben a megszakítási vonalak órajel-szinkronizálásával már nem kell foglalkoznunk, hiszen a szükséges szinkronizáló fokozatot a *CPU* tartalmazza (lásd 2-55. ábra).

[63] Ez a „viselkedés” tulajdonképpen egy adatvektorok nélküli szinkron kézfogásos interfész működésének felel meg.

A 2-74. ábrán bemutatott kapcsolási séma és az abban helyet foglaló állapotgép 2-75. ábrán látható modellje együttesen nem egy konkrét periféria-áramkört, hanem az MMIO-kompatibilis periféria-áramköröknek egy „sablonját" szemlélteti. Egy konkrét feladatot ellátó periféria-áramkörnek a csak rá jellemző sajátosságai egyedi címtérképében, és az azt megvalósító, az ábrán *PERIPH LOGIC* névvel szereplő áramköri blokkban rejlenek; ezen belül bármit elhelyezhetünk a kívánt funkció megvalósítása érdekében (újrafelhasználható IP-magok, egyedi állapotgépek, egyszerű kombinációs hálózatok stb.).

A fejezet további részeiben bemutatott MMIO periféria-áramkörök – beleértve magukat a funkcióbővítő rendszer lényegét jelentő bővítőmodulokat is – tervezésekor ebből az általános modellből indulunk ki, és azok belső működésének szemléltetése során csak azokra a részletekre térünk ki, amelyek az adott periféria-áramkör esetén egyediek, vagyis bemutatjuk azok címtérképének részleteit, valamint az e címtérkép által előírt működéshez szükséges részáramköröket.

2.5. Megszakításvezérlő

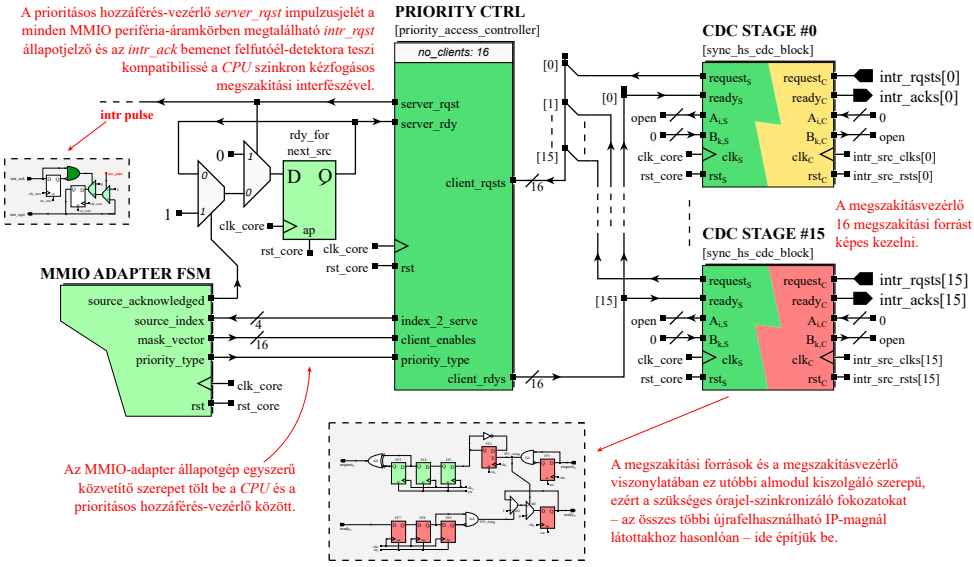
Áttanulmányozásra javasolt alfejezetek:
4.1.2. *Segédmodulok – Prioritásos hozzáférés-vezérlő*

A *CPU*-ba beépítettünk minden olyan erőforrást, ami egy teljes értékű megszakítási rendszerhez szükséges, de magának a *CPU* magnak a megszakítás-interfésze csupán egyetlen megszakítási vonalból *(intr_rqst)* és az ahhoz tartozó nyugtázójelből *(intr_ack)* áll. Az alkalmazások többségében a processzoros rendszerben több periféria-áramkör dolgozik párhuzamosan, amelyek bármelyike rendelkezhet megszakításkérések generálására alkalmas interfésszel. Ha a toplevel modulban több ilyen megszakítási forrás is van, akkor a *CPU* mellé egy megszakításvezérlőt kell beépítenünk (lásd 1-12. ábra). A megszakítási források így nem közvetlenül a *CPU*-hoz, hanem e megszakításvezérlő bemeneteihez csatlakoznak. A megszakításvezérlő alapfeladata az, hogy minden megszakítási forrás kérését továbbítsa a *CPU* egyetlen megszakítási interfésze felé úgy, hogy a *CPU*-nak lehetősége legyen megállapítani, hogy az adott pillanatban kiszolgált megszakításkérés melyik forrástól származik.

Az 1-4. és 2-53. ábrák hullámformáit összevetve láthatjuk, hogy a *CPU intr_rqst* és *intr_ack* jelei egy adatvektorok nélküli szinkron kézfogásos interfészt képeznek. A megszakítások generálására képes bármilyen áramkörnek – így a megszakításvezérlőnek is – megszakításkérő interfészén ehhez a protokollhoz kell alkalmazkodnia. Ahhoz, hogy egy megszakítási forrás közvetlenül a *CPU*-hoz és a megszakításvezérlőhöz – annak klienseként – is csatlakoztatható legyen anélkül, hogy RTL-modelljén bármiféle módosítást kellene végeznünk, kézenfekvő, hogy a megszakításvezérlő kliensinterfészeire is ezt a protokollt írjuk elő.[64] Fogalmazhatunk tehát úgy is, hogy a megszakításvezérlő alapfeladata, hogy a szinkron kézfogásos kliensinterfészein érkező kéréseket továbbítsa egy kiszolgáló-típusú, ugyancsak szinkron kézfogásos interfészre, amely egyúttal az éppen kiszolgált kliens azonosítására is alkalmas. Ez a megfogalmazás azért érdekes, mert semmi nem utal benne sem *CPU*-ra, sem periféria-áramkörökre, a körülírt funkció tehát általános

[64] Ez a megoldás egyúttal lehetőséget teremt a megszakításvezérlők kaszkádosítására is, vagyis hogy a megszakításvezérlőt egy másik megszakításvezérlő klienseként kapcsoljuk a rendszerhez, még tovább növelve a megszakítási források maximális számát.

felhasználású, újrafelhasználható IP-magként való megvalósításért kiált. Pontosan ezt az általános funkciót látja el a 4.1.2. pontban bemutatott prioritásos hozzáférés-vezérlő, a megszakításvezérlőt tehát lényegében e köré építhetjük fel, ennek az IP-magnak a szolgáltatásait tesszük elérhetővé az MMIO-buszrendszeren keresztül (lásd 2-77. ábra).



2-77. ábra. *A megszakításvezérlő belső szerkezete*

A megszakításvezérlő tulajdonképpen egy egyszerű – bár bizonyos értelemben speciális szerepkörű, a *CPU* megszakítási rendszerének „meghosszabbításaként” felfogható – MMIO periféria-áramkör, amelynek belső szerkezete megfelel a 2-74. ábrán szemléltetett általános modellnek. A megszakítási forrásokról – a rendszertervben rögzítettekkel összhangban – azt feltételezzük, hogy a megszakításvezérlőhöz képest aszinkron órajeltartományokban működnek, tehát gondoskodnunk kell a megfelelő órajel-szinkronizálásról. A kliens-kiszolgáló viszonyú almodulok közötti interfészeknél az ehhez szükséges áramköri részleteket – ugyancsak az egységesség érdekében – mindig a kiszolgálóoldal részeként kezeljük, ezért a megszakításvezérlőbe is beépítünk 16 db órajel-szinkronizáló fokozatot. Tekintve, hogy a megszakításvezérlő kliensinterfészei valójában adatvektorok nélküli szinkron kézfogásos interfészek, használhatjuk a 2-2. ábrán látható általános vektorszinkronizáló fokozatot. E fokozatoknak a megszakításvezérlő órajeléhez szinkronizált (kiszolgálóoldali) *request* kimeneteit és *ready* bemeneteit rákapcsolhatjuk a prioritásos hozzáférés-vezérlő 16-bites *client_rqsts* és *client_rdys* buszaira.

A megszakításvezérlő MMIO-címtérképét a 2-3. táblázat mutatja be.

Cím azonosítója és típusa		A címhez rendelt funkció leírása
address_mask_vector	RW	Az egyes megszakítási források engedélyezését vezérlő regiszter címe; az n-edik megszakítási forrás által generált kérésekre a megszakításvezérlő csak akkor reagál, ha a *mask_vector* regiszter n-edik bitje magas szintű (vö. 4-7. ábra: *G1 AND* kapu).
address_priority_type	RW	A megszakítási források közötti prioritáskezelés típusát beállító regiszter címe. Ha az e címre beírt adat legkisebb helyiértékű bitje alacsony szintű, akkor a megszakítási források közötti prioritást azok indexe határozza meg (az alacsonyabb indexű forrás prioritása magasabb), ha pedig magas szintű, akkor a megszakítások közötti prioritási sorrend round-robin típusú; a prioritási sorrend minden kiszolgáláskor ennek megfelelően változik.
address_source_index	R	A megszakítási forrásoktól érkező kérések *CPU* felé való továbbításakor a megszakításvezérlő az e címen elérhető regiszterben tárolja el a kiszolgálandó megszakítási forrás indexét. A *CPU* központi megszakításkiszolgáló rutinjában e regiszter kiolvasásával dönthető el, hogy melyik perifériaspecifikus megszakításkiszolgáló rutint kell meghívni (vö. 5.1.3. Függelék).
address_source_acknowledged	W	A megszakításkérés kiszolgálásának utolsó lépéseként a *CPU*-nak egy írás műveletet kell végrehajtania ezen a címen, így értesítve a megszakításvezérlőt, hogy készen áll újabb megszakításkérések fogadására.

2-3. táblázat. *A megszakításvezérlő MMIO-címtérképe*

A prioritásos hozzáférés-vezérlő működése futásidőben konfigurálható a *client_enables* és *priority_type* bemenetek segítségével. Előbbi az egyes kliensektől érkező kérések engedélyezését és tiltását szabályozza, utóbbi pedig az egyidőben érkező kérések közötti prioritáskezelés jellegét (fix vagy round-robin) határozza meg. Mindkét bemenetet az MMIO-adapter állapotgép hajtja meg, vagyis mindkettő módosítható a *CPU*-n futó program segítségével; az *address_mask_vector* és *address_priority_type* címeken végrehajtott MMIO írásműveletekkel – vagyis az ezekkel a címekkel felparaméterezett *store* utasítások végrehajtásával – e futásidejű konfigurációs beállítások bármikor módosíthatók.

Egy klienstől érkező megszakításkérés kiszolgálásának folyamata a következő: Amikor egy klienstől megszakításkérés érkezik, akkor a prioritásos hozzáférés-vezérlő a *server_rqst* kimenetén egy egyetlen órajelciklus időtartamú pozitív impulzust állít elő (vö. 4-11. ábra), miközben *index_2_serve* kimenetét a megszakításkérést küldő kliens indexére állítja be. A *server_rqst* impulzusjel – amellett, hogy alacsony szintre állítja a prioritásos hozzáférés-vezérlő *server_rdy* bemenetét meghajtó *rdy_for_next_src* állapotjelzőt – magas szintre húzza a *CPU intr_rqst* bemenetét meghajtó *intr_rqst* flip-flopot. Ennek hatására a programvégrehajtás – az aktuális gépi utasítás végrehajtását követően – a központi megszakításkiszolgáló rutin kezdőcímén folytatódik, miközben a *CPU intr_ack* kimenetének magas szintje nullába állítja a megszakításvezérlő *intr_rqst* flip-flopját, nyugtázva a megszakításkérést. A prioritásos hozzáférés-vezérlő *server_rdy* bemenetét meghajtó *rdy_for_next_src* flip-flopot nem nullázzuk mindaddig, amíg a *CPU* ki nem olvasta az éppen kiszolgált megszakítási forrás azonosítóját. A központi megszakításkiszolgáló rutin feladata, hogy a megszakításvezérlőből egy MMIO-olvasás – vagyis a megfelelő címmel felparaméterezett *load* utasítás végrehajtása – által kinyerje a megszakításvezérlőből a megszakításkérés forrásának indexét (lásd pl. 5.1.3. Függelék). Ezt az információt a megszakításvezérlő az *address_source_index* címen elérhető regiszteren keresztül bocsátja a *CPU* rendelkezésére, ami fizikailag a prioritásos hozzáférés-vezérlő *index_2_serve* kimenetét jelenti. A forrás azonosítása és a megfelelő perifériaspecifikus

szubrutin lefuttatása után a *CPU* az *address_source_acknowledged* címen végrehajtott írás művelettel állíthatja magas szintre a *rdy_for_next_src* flip-flopot, egyúttal a prioritásos hozzáférés-vezérlő *server_rdy* bemenetét, vagyis utasíthatja azt a megszakításkérés nyugtázásának forrás felé való továbbítására, a megszakítási források további monitorozására és újabb megszakításkérés fogadására. A megszakításvezérlő MMIO-adapter állapotgépének releváns részleteit a 2-78. ábra mutatja be.

VHDL

```
1  when idle => s_ready_2_cpu <= '1';
2
3               -- Handle MMIO write requests...
4               if ( write_strobe_from_cpu_df1 = '1' and cs = '1' ) then
5                 s_ready_2_cpu <= '0';
6                 |
7                 if ( unsigned(address_from_cpu) = address_mask_vector ) then
8                   mask_vector <= data_from_cpu; state <= wait_for_deassert_strobes;
9                 elsif ( unsigned(address_from_cpu) = address_priority_type ) then
10                  priority_type <= data_from_cpu(0); state <= wait_for_deassert_strobes;
11                elsif ( unsigned(address_from_cpu) = address_source_acknowledged ) then
12                  source_acknowledged <= '1'; state <= wait_for_deassert_strobes;
13                else state <= wait_for_deassert_strobes;
14                end if;
15                |
16              end if;
17
18
19              -- Handle MMIO read requests...
20              if ( read_strobe_from_cpu_df1 = '1' and cs = '1') then
21                s_ready_2_cpu <= '0';
22                |
23                if ( unsigned(address_from_cpu) = address_mask_vector ) then
24                  s_data_2_cpu <= mask_vector; state <= wait_for_deassert_strobes;
25                elsif ( unsigned(address_from_cpu) = address_priority_type ) then
26                  s_data_2_cpu(15 downto 1) <= (others => '0');
27                  s_data_2_cpu(0) <= priority_type; state <= wait_for_deassert_strobes;
28                elsif ( unsigned(address_from_cpu) = address_source_index ) then
29                  s_data_2_cpu <= std_logic_vector(to_unsigned(prior_index_2_serve, 16));
30                  state <= wait_for_deassert_strobes;
31                else state <= wait_for_deassert_strobes;
32                end if;
33                |
34              end if;
35
36 when wait_for_deassert_strobes => source_acknowledged <= '0';
37                                   if (
38                                     read_strobe_from_cpu_df1 = '0' and
39                                     write_strobe_from_cpu_df1 = '0'
40                                   ) then state <= idle;
41                                   end if;
```

2-78. ábra. *A megszakításvezérlő MMIO-adapter állapotgépének releváns részletei*

Az állapotgép HDL-modelljében piros kiemeléssel szerepelnek azok a kódrészletek, amelyek nem az általános MMIO perifériaáramkör-sablon részei, hanem a megszakításvezérlőre jellemzők, annak 2-3. táblázatbeli címtérképének megvalósításában játszanak szerepet. Látható, hogy a *mask_vector* és *priority_type* regiszterek egyszerű írható/olvasható (RW) vezérlőregiszterek (vö. 2-75. ábra: *control_reg_1*), míg az *address_source_index* cím egy csak-olvasható (R) állapotregisztert rejt (vö. 2-75. ábra: *status_reg_3*).

Külön figyelmet érdemel a megszakítások nyugtázására szolgáló *address_source_acknowledged* nevű, csak-írható (W) MMIO-cím. Az ezen a címen végrehajtott MMIO-íráskor – vagyis amikor a *CPU* egy ezzel a címmel felparaméterezett *store* utasítást hajt végre – az MMIO-adapter állapotgép egy egyetlen periódusnyi pozitív impulzust állít elő (2-78. ábra: 12. és 36. sor) a *source_acknowedged* jelen, magas szintre állítva ezáltal a prioritásos hozzáférés-vezérlő *server_rdy* bemenetét

meghajtó *rdy_for_next_src* flip-flopot. Az *address_source_acknowledged* címen tehát olvasást végrehajtani értelmetlen, továbbá íráskor ez az egyetlen impulzus az MMIO-tranzakcióban elküldött adat értékétől – tehát a *store* utasítás [11:8] adatmezőjétől – függetlenül előáll, ilyen értelemben tehát az adott címmel felparaméterezett MMIO-írás puszta végrehajtása is önálló jelentéssel bír.

2.6. Memóriarendszer

Áttanulmányozásra javasolt alfejezetek:

***4.1.2.** Segédmodulok – Prioritásos hozzáférés-vezérlő*
***4.2.** On-chip adattárolás*
***4.4.3.** Katalógus-áramkörök interfészadapterei – SDRAM*

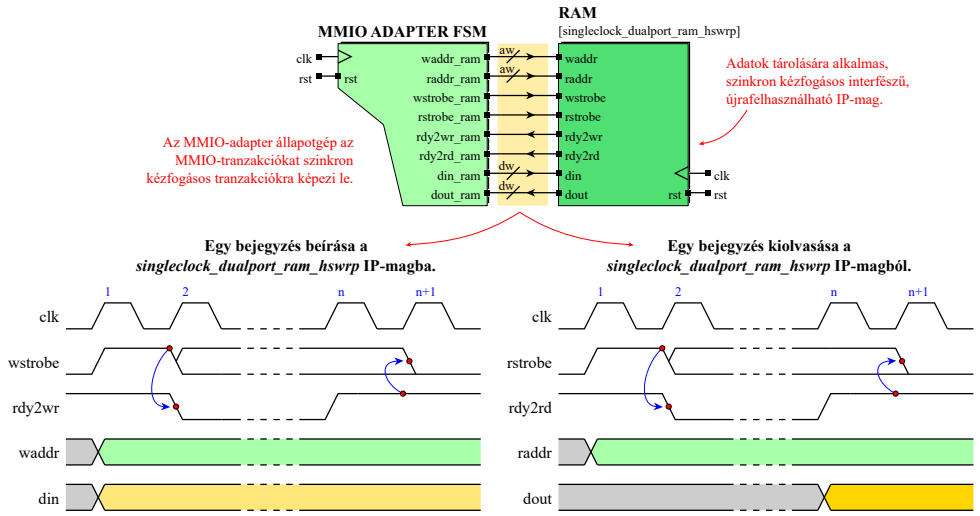
Egy tárolt programot végrehajtó mikroáramkör esetén a széles körű alkalmazhatóság és a számítási teljesítmény szempontjából is óriási jelentősége van a forrásadatokat, az adatfeldolgozó algoritmusok rész- és végeredményeit, a periféria-áramkörök állapotinformációit stb. tárolni képes, ún. operatív tár méretének. A funkcióbővítő rendszer központi processzora Harvard architektúrájú, vagyis a futtatott programot tároló utasításmemóriához a *CPU*-nak nincs általános hozzáférése, onnan kizárólag a végrehajtandó utasításokat képes beolvasni, ez tehát operatív tárként nem használható. A *CPU* mag tartalmaz 16 db 16-bites általános célú regisztert, amelyek funkcionálisan alkalmasak ugyan az operatív tár szerepének betöltésére, de ez a tárkapacitás az alkalmazások túlnyomó részében közel sem elegendő.[65] A problémát olyan periféria-áramkörök tervezésével és beépítésével orvosoljuk, amelyek feladata nem több, mint hogy lehetővé tegyék a *CPU* számára az adatok eltárolását és visszaolvasását, természetesen MMIO-interfészen keresztül.

2.6.1. On-chip statikus RAM mint operatív tár

Az adatok eltárolása talán a legalapvetőbb digitális funkció, az összetettebb részegységek kifejlesztésének elengedhetetlen kelléke. Az MMIO-kompatibilis memóriamodult – a rendszertervben *BRAM* – egy általános célú, szinkron kézfogásos interfésszel ellátott, újrafelhasználható memóriamodulnak a 2-74. ábrán látható MMIO perifériaáramkör-sablonba való integrálásával hozzuk létre (lásd 2-79. ábra).

Az újrafelhasználható *singleclock_dualport_ram_hswrp* IP-mag – amelynek részleteit a 4.2.1. és 4.2.4. pontok tárgyalják – címbuszának és adatbuszának mérete *(aw: **a**ddress **w**idth, dw: **d**ata **w**idth)* szintézis-paraméter. A tároló összkapacitásának és szervezésének ezt a rugalmas jellegét a *BRAM* modulra is célszerű átörökíteni, lehetővé téve a toplevel integrátor számára a felhasznált blokk-RAM modulok számának optimalizálását az adott alkalmazás igényei szerint. Természetesen figyelembe kell vennünk magának az MMIO-buszrendszernek a korlátait; a *BRAM* modul címbuszának és adatbuszának mérete legfeljebb 16 bit lehet.

[65] Ráadásul az általános célú regisztertömb egyes rekeszeit tipikusan valamilyen dedikált célra, egy regiszterhasználati konvenciónak megfelelő „szerepkörben" használjuk, ami az operatív tárolóként elérhető tárkapacitást tovább csökkenti. Az 5.1.2. Függelékben bemutatott regiszterhasználati konvenció esetén a 16 regiszterből csupán ötnek *(temp_5...1)* nem jut valamilyen egyedi szerep, így a konkrét program igényeitől függően használható bármire, vagyis tekinthető operatív tárnak.



2-79. ábra. *A BRAM modul szerkezete*

Az MMIO periféria-áramkörök lokális címdekódere és MMIO-adapter állapotgépe általános esetben a toplevel integrátor által a példányosításkor meghatározott, az adott periféria-áramkörhöz hozzárendelt címekkel veti össze a működés során a *CPU* által az MMIO slave interfész *address_from_cpu* buszára kapcsolt értéket (lásd 2-8. ábra: 16–20. sor, 2-75. ábra: 16–46. sor, 2-78. ábra: 7–32. sor). Az olyan periféria-áramkörök esetén, ahol e címek egymástól független vezérlő- és állapotregisztereket (vagy szolgáltatásokat) takarnak, nem szükséges, hogy az adott periféria-áramkörhöz hozzárendelt címek összefüggő tartományt alkossanak. Az ütközésmentes címkiosztási séma megalkotása teljes egészében a toplevel integrátor felelőssége. Kioszthatja az MMIO-címeket úgy, hogy az egyazon periféria-áramkörhöz tartozó címek összefüggő tartományt képezzenek, de elképzelhető olyan megoldás is, amelyben a periféria-áramkörök vezérlőregisztereinek címei alkotnak egy összefüggő címtartományt, állapotregisztereik pedig egy másikat. Az operatív memória szerepkörére szánt *BRAM* modul címdekóderét és MMIO-adapter állapotgépét – ehhez képest némileg kötöttebb módon – a következőképpen készítjük el:

- A periféria-áramkör belsejében lévő memóriamodul a teljes MMIO-címtartomány (0...65.535) egy szeletének közvetlen fizikai leképeződése lesz; minden MMIO-tranzakció egy-egy írás- vagy olvasásműveletet jelent a szinkron kézfogásos interfészen úgy, hogy az MMIO-tranzakciót előidéző *load* vagy *store* utasítások cím- és adatparaméterei megegyeznek az általuk kiváltott szinkron kézfogásos tranzakciók cím- és adatparamétereivel.
- A *BRAM* modul a teljes MMIO-címtartománynak egy összefüggő szeletét foglalja le.

A 2-80. ábra a fenti megfontolásoknak megfelelő címdekóder logikát, a 2-81. ábra pedig az MMIO-adapter állapotgép vezérlési állapotait szemlélteti.

A modul által lefoglalt címtartomány méretét és a teljes MMIO-címtartományon belüli helyét a toplevel integrátor határozza meg az *address_low* és *address_high* paraméterek beállításával (2-80. ábra: 1. sor).

Verilog

```
assign cs = (address_from_cpu >= address_low & address_from_cpu <= address_high) ? 1 : 0;
assign ready_2_cpu = (cs == 1) ? s_ready_2_cpu : 0;
assign data_2_cpu = (cs == 1) ? s_data_2_cpu : 0;
```

2-80. ábra. *A BRAM modul lokális címdekódere*

Verilog

```
idle: begin
  s_ready_2_cpu <= 1;

  // Handle MMIO write requests...
  if ( write_strobe_from_cpu_df1 == 1 && cs == 1 ) begin
    s_ready_2_cpu <= 0;
    waddr_ram <= address_from_cpu[aw-1:0] - address_low;
    din_ram <= data_from_cpu[dw-1:0];
    wstrobe_ram <= 1;
    state <= write_1;
  end

  // Handle MMIO read requests...
  if ( read_strobe_from_cpu_df1 == 1 && cs == 1 ) begin
    s_ready_2_cpu <= 0;
    raddr_ram <= address_from_cpu[aw-1:0] - address_low;
    rstrobe_ram <= 1;
    state <= read_1;
  end

end

```

Verilog (folyt.)

```
write_1: state <= write_2;

write_2:
  if ( rdy2wr_ram == 1 ) begin
    wstrobe_ram <= 0;
    state <= wait_for_deassert_strobes;
  end

read_1: state <= read_2;

read_2:
  if ( rdy2rd_ram == 1 ) begin
    rstrobe_ram <= 0;
    s_data_2_cpu <= 0;
    s_data_2_cpu[dw-1:0] <= dout_ram;
    state <= wait_for_deassert_strobes;
  end

wait_for_deassert_strobes:
  if (
    read_strobe_from_cpu_df1 == 0 &&
    write_strobe_from_cpu_df1 == 0
  ) state <= idle;
```

2-81. ábra. *A BRAM modul MMIO-adapter állapotgépének vezérlési állapotai*

A 2-81. ábrán látható, hogy az MMIO-interfész írás- és olvasástranzakcióit az MMIO-adapter állapotgép egy-egy szinkron kézfogásos tranzakcióvá alakítja át. Figyeljük meg, hogy ha a belső memóriamodul cím- és/vagy adatbuszának méretét a toplevel integrátor a maximális 16-nál kisebb értékre állítja be, akkor a periféria-áramkör a 16-bites MMIO-tranzakciók cím- és adatbuszainak csak az alsó *aw* és *dw* darab bitjét veszi figyelembe (2-81. ábra: 7–8. és 16. sor), kiolvasáskor pedig a *CPU* felé küldött adatbusz *(s_data_2_cpu → data_2_cpu)* felső 16-*dw* darab bitjét mindig nullára állítja (2-81. ábra: 36–37. sor).[66]

A *BRAM* modul operatív tárként való használata a *CPU* programozója szempontjából nagyon kényelmes, hiszen egy-egy *store* vagy *load* utasítással – amelyek tehát egy-egy MMIO-tranzakciót eredményeznek – eltárolhat vagy visszaolvashat egy-egy 16-bites szót. Ez teljesítmény szempontjából is kedvező, mivel a *BRAM* modul belsejében a bejegyzések tárolása fizikailag az FPGA blokk-RAM moduljaiban, tehát gyors hozzáférésű, az általános célú regisztertömböt képező flip-flopokkal csaknem azonos késleltetésű on-chip SRAM-cellákban történik. A megoldás hátránya, hogy a tárkapacitás növelésével – ami természetesen sosem elég – a *BRAM* modul egyre több és több címet foglal el a *CPU* teljes címtartományából. Egy 256 db 16-bites bejegyzés tárolására képes (512 bájtos) *BRAM* modul a teljes (0...65.535) címtartománynak alig 0,4%-át foglalja le, de ha az operatív tár méretét 32 kB-ra növeljük, akkor ez az arány már 25%. Ne feledjük, hogy a rendszer

[66] Az *always*-blokk szekvenciális utasításkiértékelési mechanizmusának köszönhetően a két egymást követő, egyazon jelre vonatkozó értékadás tulajdonképpen azt fejezi ki, hogy a 36. sorban az *s_data_2_cpu* jel minden bitjét nullába állítjuk, majd a 37. sorban a [*dw*-1:0] tartományba írt nullákat – még ugyanabban a szimulációs ciklusban – a belső memóriamodul *dout_ram* adatkimenetének értékével írjuk felül. Áramkörileg ez annyit jelent, hogy az *s_data_2_cpu* busz alsó *dw* darab bitjét a belső memóriamodul adatkimenete hajtja meg, a felső 16-*dw* darab bitre pedig konstans nullát kapcsolunk (vö. DRMSz: 4.4.4. alfejezet: Egyidejűség és sorrendiség a HDL-ekben).

toplevel moduljában példányosított teljes perifériaáramkör-készlet osztozik a közös címtartományon, vagyis ez utóbbi esetben az összes kommunikációs, időzítési stb. feladatot ellátó periféria-áramkörnek, sőt a bővítőmoduloknak is a fennmaradó 75%-ba kell beleférniük.

A *BRAM* modulnak az a sajátossága, hogy az adatokat az FPGA blokk-RAM moduljaiban tárolja, hasonló problémát jelenthet, hiszen ezek a memóriamodulok ugyancsak korlátozott mennyiségben állnak rendelkezésre (legfeljebb Mbit nagyságrend). Több tíz kB vagy annál nagyobb tárkapacitású operatív tárat tehát a *BRAM* modullal kialakítani nem célszerű, ilyenkor valamilyen külső memória-áramkört kell illesztenünk a rendszerhez, ahogy tettük azt a *CPU* által futtatandó programok nemillékony tárolásakor is. Az operatív tár esetében – a modul szerepköréből adódóan – nem szükséges, hogy a külső memória nemillékony legyen, fontosabb szempont a kis hozzáférési idő és a tárkapacitás. A PC-s környezetben használt SDRAM e kívánalmaknak teljesen megfelel, ezért a következő szakaszban egy ilyen katalógus-áramkör illesztését végezzük el.

2.6.2. Külső szinkron dinamikus RAM mint operatív tár

Ahogy a programokat tároló EEPROM katalógus-áramköröknél is láthattuk, a memória jellegű külső eszközöket olyan interfészadapter egységeken keresztül célszerű illeszteni, amelyek szinkron kézfogásos interfészen teszik lehetővé felhasználójuk számára a memóriához való hozzáférést, vagyis amelyek használata az FPGA-n belülről egy on-chip memóriamodulával – pl. a *BRAM* periféria-áramkörben is használt *singleclock_dualport_ram_hswrp* IP-magéval – egyezik meg. A 4.4.3. pontban tárgyalt SDRAM interfészadapter IP-mag pontosan így működik, a nagy tárkapacitást biztosító katalógus-áramkörre épülő operatívtár-rendszerünk tehát erre az újrafelhasználható IP-magra támaszkodik.

Mivel az SDRAM IFA felhasználói interfésze szándékosan a *singleclock_dualport_ram_hswrp* IP-magéval egyezik meg, azt gondolhatnánk, hogy az MMIO-buszrendszerre kapcsolásához szükséges illesztőlogika (1-13. ábra: *EXTRAM*) is meg kell hogy egyezzen a *BRAM*-nál látottakkal. Ez azonban több okból sem lehetséges:

- Az SDRAM tárkapacitása sokkal nagyobb, mint egy on-chip memóriamodulé, így az írás- és olvasásműveletekhez szükséges memóriacím 16 bitnél jellemzően nagyobb lesz. Egyetlen MMIO-tranzakció tehát semmiképp sem lesz elegendő arra, hogy a külső memóriához való egyetlen hozzáféréshez szükséges összes információt magában foglalja; egy-egy külső memóriaírás vagy -olvasás lebonyolításához több MMIO-tranzakcióra lesz szükség.
- A külső memória illesztésére részben azért van szükség, hogy úgy tudjunk nagy tárkapacitást kezelni, hogy a *CPU* MMIO-címtartományából aránylag kevés címet foglaljunk le. A *BRAM* modulnál alkalmazott közvetlen leképezés az MMIO-címtartomány és a külső memóriachip címtartománya között tehát ugyancsak kivihetetlen.
- Ha az SDRAM interfészadaptert egy MMIO-illesztőlogikán keresztül közvetlenül az MMIO-buszrendszerre kapcsolnánk, akkor bármely almodul csak a *CPU*-n keresztül tudna hozzáférni a külső memóriához, hiszen a *CPU* az egyetlen master egység az MMIO-buszon. Bár nem ismerhetjük előre a bővítőmodulok szerepét és pontos feladatát, de azt mindenesetre sejthetjük, hogy ha a megvalósított funkció nagy tárkapacitást igényel, akkor a számítási teljesítményre alighanem nincs jó hatással az, ha ehhez a tárterülethez

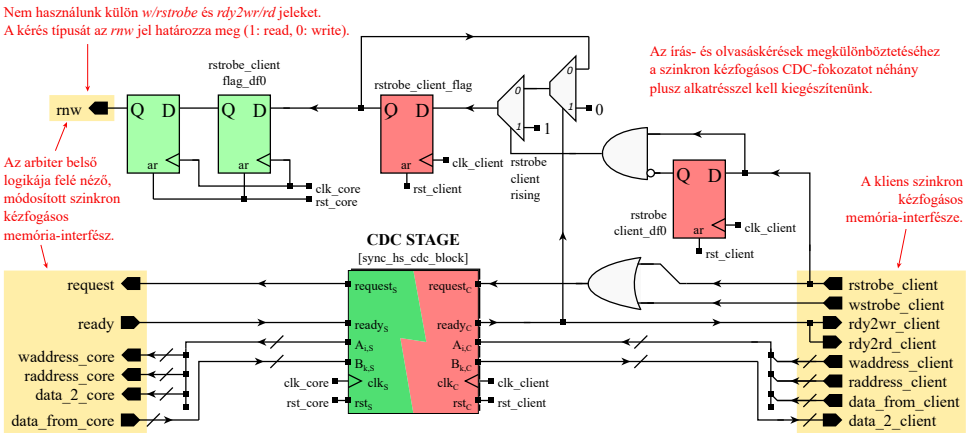
a bővítőmodul csak egy aránylag kis teljesítményű processzoron keresztül fér hozzá. Ezért a rendszerterv 1-14. ábrájának megfelelően az SDRAM interfészadaptert nem közvetlenül, hanem egy memória-arbiteren *(MEMARB)* keresztül kapcsoljuk az MMIO-buszrendszerre, ami nyolc szereplőnek biztosít közvetlen hozzáférést a külső memóriához. E nyolc szereplő közül az egyik nyilván maga a *CPU* lesz, ami – amellett, hogy egy dedikált MMIO-interfészen keresztül magát az arbitert is konfigurálja – így már nem az SDRAM interfészadapterét, hanem az arbiter adatinterfészét „látja" az MMIO-buszrendszeren, tehát az adatforgalmat bonyolító illesztőlogikának (az 1-14. ábrán *EXTRAM*) nem az SDRAM IFA, hanem az arbiter szolgáltatásait kell leképeznie MMIO-tranzakciókra.

Mielőtt tehát megterveznénk a *CPU* számára a külső SDRAM-hoz való hozzáférést biztosító *EXTRAM* almodult, tervezzünk egy egyszerű, MMIO-kompatibilis memória-arbitert.

Memória-arbiter ◆ A külső operatív memóriához való hozzáférést a rendszerterv szerint nyolc kliens számára kell lehetővé tennünk nyolc egymástól független szinkron kézfogásos interfészen. E kliensekről feltételezzük, hogy az arbitertől és egymástól is teljesen független almodulok, amelyek külön-külön aszinkron órajelforrásokról járnak. Ez azt jelenti, hogy az arbiternek a megfelelő órajel-szinkronizáló fokozatokat is magában kell foglalnia. Mivel szinkron kézfogásos memória-interfészekről van szó – amelyek mögött ezúttal nem egy on-chip memóriamodul, hanem a nulla késleltetésű működésre garantáltan alkalmatlan arbiter logika rejtőzik – felhasználhatjuk a 2.1. pontban bemutatott általános vektorszinkronizáló fokozat 2-2. ábrán látható, egyszerűbb változatát. Az arbiter belső szerkezetének egyszerűsítése érdekében az egyazon klienshez tartozó beíró és kiolvasó részinterfészeket nem kezeljük egymástól függetlenként, vagyis feltételezzük, hogy egy kliens egyidőben vagy egy írást vagy egy olvasást kezdeményez, a kettőt együtt soha. Így a szinkron kézfogásos memória-interfész beírást és kiolvasást kezdeményező *wstrobe* és *rstrobe* jelei összevonhatók, tehát a teljes interfészhez csak egy vektorszinkronizáló fokozat szükséges (lásd 2-82. ábra).[67]

A beíró és kiolvasó részinterfészek CDC-fokozatainak összevonásával 10 flip-flopot megspóroltunk ugyan, de sajnos a két művelet megkülönböztetésére az arbiter-oldalon is szükség lesz, ami néhány további alkatrész beépítését teszi szükségessé. Az arbiter-oldali, olvasáskéréskor magas, íráskor pedig alacsony szintű *rnw* kimenetet a kliens *rstrobe* jeléből származtatjuk; míg a kliens órajeltartományában egy felfutóél-detektor segítségével minden olvasáskéréskor magas szintre állítjuk az *rstrobe_client_flag* állapotjelző flip-flopot. Mivel e flip-flop az egyes tranzakciók végén a CDC-fokozat kliens-oldali *ready* jelének felfutása hatására nullázódik, ezért kimenete tulajdonképpen egy statikus jelnek tekinthető, ami a kliens olvasáskérései hatására magas szintűvé válik,

[67] Az összevont vektorszinkronizáló fokozat használatának vannak bizonyos következményei, amelyek mellett nem mehetünk el szó nélkül. A két *strobe* jel összevonása egyúttal azt is jelenti, hogy a két nyugtázójel *(rdy2wr, rdy2rd)* mindig együtt változik, függetlenül attól, hogy a kliens írás- vagy olvasáskéréssel fordult-e az arbiterhez. Ez így rendben is van; az arbiter nem tud írás és olvasás műveleteket párhuzamosan végrehajtani, tehát logikus, hogy írás közben a *rdy2rd*, olvasás közben pedig a *rdy2wr* kimenet is alacsony legyen mindaddig, amíg az adott tranzakció be nem fejeződik. Az adatvektorok összevont kezelése azt is maga után vonja, hogy beíráskor az *raddress_core* (az arbiter-oldali olvasási cím) és a *data_2_client* (kliens-oldali kiolvasott adat), kiolvasáskor pedig a *waddress_core* (az arbiter-oldali írási cím) és a *data_2_core* (arbiter-oldali beírandó adat) vektorok értéke érvénytelenné válik. Ezek közül funkcionális szempontból csak a kliensnek szóló *data_2_client* vektor beíráskor történő, látszólag indokolatlan érvénytelenné válása okozhat némi fejfájást, ha a kliens arra számított, hogy ez a vektor a következő kiolvasásig garantáltan változatlan marad. Az arbiternek ezt a sajátosságát tehát a kliens-almodulok kénytelenek figyelembe venni, nem szabad a kiolvasott adatvektort közvetítő portot „hosszú távú adattárolásra használniuk". Ez egyébként a lokalitás elve alapján amúgy sem lenne tanácsos.



2-82. ábra. *A MEMARB modul kliensinterfészeinek órajel-szinkronizáló fokozata*

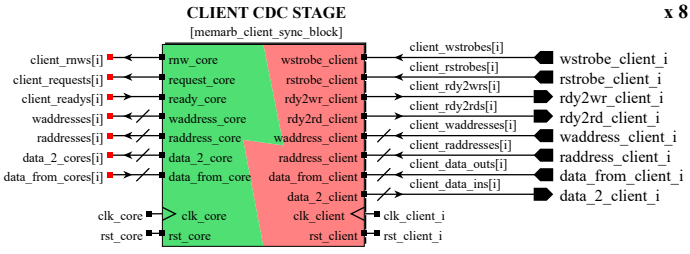
Verilog

```
1  // Array for CDC-stage instantiation
2  logic [7:0]    client_rnws;
3  logic [7:0]    client_requests;
4  logic [7:0]    client_readys;
5  logic [aw-1:0] waddresses[7:0];
6  logic [aw-1:0] raddresses[7:0];
7  logic [dw-1:0] data_2_cores[7:0];
8  logic [dw-1:0] data_from_cores[7:0];
9  logic [7:0]    client_clks;
10 logic [7:0]    client_rsts;
11 logic [7:0]    client_wstrobes;
12 logic [7:0]    client_rstrobes;
13 logic [7:0]    client_rdy2wrs;
14 logic [7:0]    client_rdy2rds;
15 logic [aw-1:0] client_waddresses[7:0];
16 logic [aw-1:0] client_raddresses[7:0];
17 logic [dw-1:0] client_data_outs[7:0];
18 logic [dw-1:0] client_data_ins[7:0];
19
```

Verilog (folyt.)

```
20 // Generate client CDC blocks
21 genvar i; generate
22   for ( i = 0; i < 8; i++ ) begin
23     memarb_client_sync_block #(aw, dw) CLIENT_CDC_STAGE (
24       .clk_core(clk_core),
25       .rst_core(rst_core),
26       .rnw(client_rnws[i]),
27       .request(client_requests[i]),
28       .ready(client_readys[i]),
29       .waddress_core(waddresses[i]),
30       .raddress_core(raddresses[i]),
31       .data_2_core(data_2_cores[i]),
32       .data_from_core(data_from_cores[i]),
33       .clk_client(client_clks[i]),
34       .rst_client(client_rsts[i]),
35       .wstrobe_client(client_wstrobes[i]),
36       .rstrobe_client(client_rstrobes[i]),
37       .rdy2wr_client(client_rdy2wrs[i]),
38       .rdy2rd_client(client_rdy2rds[i]),
39       .waddress_client(client_waddresses[i]),
40       .raddress_client(client_raddresses[i]),
41       .data_from_client(client_data_outs[i]),
42       .data_2_client(client_data_ins[i])
43     );
44   end
45 endgenerate
```

Áramköri struktúra



2-83. ábra. *A MEMARB modul CDC-fokozatainak példányosítása*

a teljes olvasástranzakció alatt magas is marad, íráskérések esetén viszont végig alacsony. Statikus jel lévén az *rstrobe_client_flag* állapotjelző egy egyszerű double-flop szinkronizáló fokozattal átvihető az arbiter órajeltartományába *(rstrobe_client_df0 → rnw)*. A 2-82. ábrán látható teljes CDC-fokozatot a *MEMARB* modulon belül nyolcszor példányosítjuk (lásd 2-83. ábra).

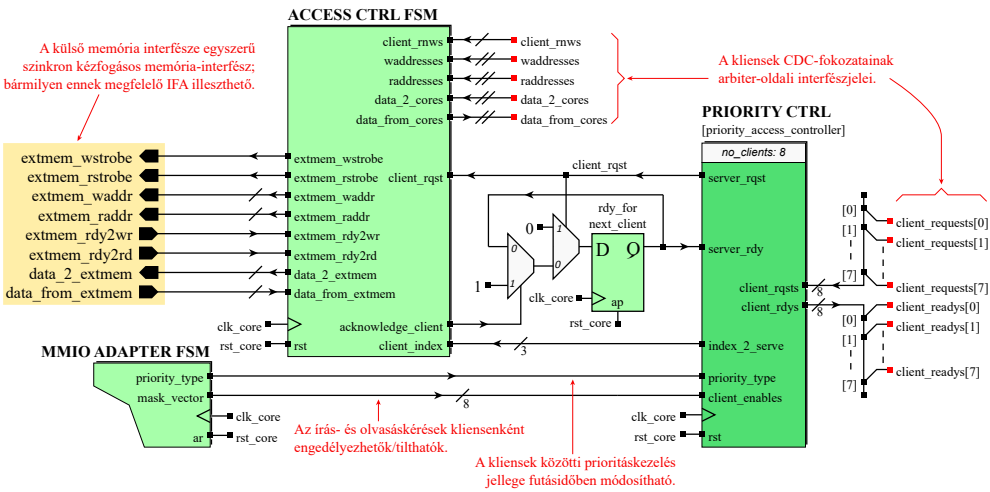
Egyazon almodul sok példányának létrehozása a *generate* utasítással fejezhető ki hatékonyan. A *generate* utasításon belüli példányosítás utasítás(ok) port-hozzárendeléseinek kifejezéséhez a CDC-fokozatok azonos funkciójú (és elnevezésű) interfészjeleit egy-egy tömbbe *(array)* kell szervezni. Az egyes CDC-fokozatok egybites interfészjelei tehát egy-egy nyolcbites vektor megfelelő helyiértékű bitjeként (pl. a 6-os CDC-fokozat *client_rnw* kimenete által meghajtott jel: *client_rnw[6]*), a több-bitesek pedig egy-egy vektortömb megfelelő indexű elemeként (pl. a 4-es indexű CDC-fokozat *waddress_core* bemenetét meghajtó jel: *waddresses[4]*) érhetők el az arbiter belső logikája számára. Ennek hangsúlyozása céljából e jelek deklarációit is feltüntettük a 2-83. ábra bal oldalán.

A CDC-fokozatok által közvetített interfészek kiszolgáló típusúak, tehát írás- és olvasástranzakciókat képesek kezdeményezni. E kezdeményezések bármikor érkezhetnek és származhatnak bármelyik klienstől, vagyis közöttük valamiféle prioritási sorrendet kell felállítani az arbiteren belül, hiszen az SDRAM IFA egyszerre csak egy kérést tud kiszolgálni. A párhuzamosan működő, szinkron kézfogásos kérések generálására alkalmas kliensek időosztásos, valamilyen prioritási séma szerinti kiszolgálásának igénye már ismerősen csenghet; a 2.5. pontban tárgyalt megszakításvezérlőnek pontosan ugyanez volt a feladata a különböző megszakítási forrásokkal kapcsolatban. A megoldás ezúttal is a kifejezetten e célra kifejlesztett, a 4.1.2. pontban részletesen bemutatott prioritásos hozzáférés-vezérlő lesz.[68] Ez az újrafelhasználható almodul gondoskodik arról, hogy a kliensek *request/ready* jelpárja a szinkron kézfogásos protokollnak megfelelően viselkedjen, a beérkező kéréseket pedig időben sorba állítva képezi le az arbiter belső logikája számára egyetlen, értelemszerűen szintén kiszolgáló típusú, ugyancsak szinkron kézfogásos protokoll szerint viselkedő jelpárra *(server_rqst/server_rdy)*, amelyet az egyes kérések forrását azonosító kimeneti vektorral egészít ki *(index_2_serve)*. Az arbiter további részáramköreinek tehát kétféle feladatot kell ellátniuk; egyrészt közvetíteniük kell a CDC-fokozatokon és a prioritásos hozzáférés-vezérlőn keresztül érkező klienskéréseket a külső SDRAM interfészadaptere felé, másrészt biztosítaniuk kell a *CPU* számára a prioritásos hozzáférés-vezérlő MMIO-buszrendszeren keresztüli konfigurálásának lehetőségét. E két részfeladat egymástól független, ezért végrehajtásukra két külön állapotgépet hozunk létre. Az arbiter ennek megfelelő RTL kapcsolási sémáját a 2-84. ábra szemlélteti.

Az írás- és olvasástranzakciók külső SDRAM felé való továbbításáért az *ACCESS CTRL FSM* állapotgép felel. HDL-modelljének vázlatát a 2-85. ábra bal, vezérlési állapotainak részleteit pedig jobb oldala mutatja be.

Az *ACCESS CTRL FSM* állapotgépnek mindaddig semmi dolga nincs, amíg a prioritásos hozzáférés-vezérlő a *client_rqst* jel magas szintre húzásával nem jelzi egy kliens kérés beérkezését (2-85. ábra jobb oldal: 3. sor). Ekkor – miközben az állapotgépen kívüli *rdy_for_next_client* állapotjelző nullázása ideiglenesen letiltja az esetleg beérkező további kliens kérések feldolgozását – az *ACCESS CTRL*

[68] Ezen a ponton idézzük fel egy pillanatra a 2. fejezet bevezetőjében említett RTL-tervezési alapelveket. A fenti példában látványosan érhető tetten az újrafelhasználhatóságra, regularitásra és modularitásra vonatkozó irányelveink létjogosultsága; egy kellően általánosan megfogalmazott részfunkciót önálló almodulként megvalósítva azt számtalan, látszólag eltérő kontextusban tudjuk felhasználni. Az egységesített (reguláris) interfészek definiálása e módszer gyakorlati kivitelezését könnyíti meg.



2-84. ábra. *A MEMARB modul belső szerkezete*

Verilog

```
1  // ACCESS CTRL FSM
2  always @ ( posedge clk_core, posedge rst_core )
3  begin
4    if ( rst_core == 1 ) begin
5      state_acc <= wait_for_request;
6      acknowledge_client <= 0;
7      extmem_waddr <= 0;
8      extmem_raddr <= 0;
9      data_2_extmem <= 0;
10     extmem_wstrobe <= 0;
11     extmem_rstrobe <= 0;
12   end else begin
13
14     // defaults...
15     acknowledge_client <= 0;
16
17     case ( state_acc )
18
19       wait_for_request: [...]
20
21       [...]
22
23       default: ;
24
25     endcase
26
27   end
28 end
29
```

Verilog

```
1  wait_for_request: begin
2
3    if ( client_rqst == 1 ) begin
4      extmem_waddr <= waddresses[client_index];
5      extmem_raddr <= raddresses[client_index];
6      data_2_extmem <= data_2_cores[client_index];
7
8      if ( client_rnws[client_index] == 0 ) begin
9        extmem_wstrobe <= 1; state_acc <= write_1;
10     end else begin
11       extmem_rstrobe <= 1; state_acc <= read_1;
12     end
13
14   end
15
16 end
17
18 write_1: state_acc <= write_2;
19
20 write_2: if ( extmem_rdy2wr == 1 ) begin
21              extmem_wstrobe <= 0;
22              acknowledge_client <= 1;
23              state_acc <= wait_for_request;
24            end
25
26 read_1: state_acc <= read_2;
27
28 read_2: if ( extmem_rdy2rd == 1 ) begin
29             extmem_rstrobe <= 0;
30             data_from_cores[client_index] <=
31             data_from_extmem;
32             acknowledge_client <= 1;
33             state_acc <= wait_for_request;
34           end
```

2-85. ábra. *Az ACCESS CTRL FSM állapotgép Verilog-leírása*

FSM állapotgép a külső SDRAM interfészadapterének címbemeneteire másolja a *client_index* által azonosított CDC-fokozat kimenetéről leolvasott írási és olvasási címeket, valamint a kliens által küldött adatvektort (2-85. ábra jobb oldal: 4–6. sor). A kliens kérésének típusát a CDC-fokozatok *rnw_core* kimenete jelöli ki (1: olvasás, 0: írás). E jeleket a nyolcbites *client_rnws* vektor fogja össze, amit a *client_index* értékével indexelve megkapjuk az éppen kiszolgált kérés típusát (2-85. ábra jobb

oldal: 8. sor). A kérés típusától függően az állapotgép vagy a *write_1...2* vagy a *read_1...2* állapotokban folytatja a műveletvégzést. Mindkét állapotpár egy-egy szokásos szinkron kézfogásos írás- vagy olvasásműveletet vezényel le a külső SDRAM interfészadaptere felé. Amikor az SDRAM IFA az *extmem_rdy2wr/rd* bemenet magas szintre állításával jelzi a művelet végét, akkor az állapotgép – amellett, hogy olvasás esetén a megfelelő CDC-fokozat adatbemenetére másolja a külső SDRAM-ból kiolvasott adatot (2-85. ábra jobb oldal: 30–31. sor) – az *acknowledge_client* kimenetén egy egyetlen periódusnyi pozitív impulzust előállítva (2-85. ábra jobb oldal: 22/32. sor) magas szintre húzza a *rdy_for_next_client* állapotjelzőt, utasítva ezáltal a prioritásos hozzáférés-vezérlőt a kliens-oldali szinkron kézfogásos tranzakció lezárására, egyúttal engedélyezve további klienskérések fogadását.

A 2-84. ábrán jól látható, hogy az arbiternek – a benne lévő prioritásos hozzáférés-vezérlő miatt – vannak bizonyos konfigurációs lehetőségei; az egyes kliensektől érkező kérések feldolgozása letiltható, a kliensek közötti prioritáskezelés módja (fix vagy round-robin) pedig változtatható. E konfigurációs lehetőségeket MMIO-interfészen keresztül biztosítjuk a *CPU* számára. Az arbiter ennek megfelelő címtérképét a 2-4. táblázat mutatja be.

Cím azonosítója és típusa		A címhez rendelt funkció leírása
address_mask_vector	RW	Az egyes klienskérések feldolgozásának engedélyezését vezérlő regiszter címe; az n-edik kliens által generált kérésekre az arbiter csak akkor reagál, ha a *mask_vector* regiszter n-edik bitje magas szintű (vö. 4-7. ábra: *G1 AND* kapu).
address_priority_type	RW	A kliensek közötti prioritáskezelés típusát beállító regiszter címe. Ha az e címre beírt adat legkisebb helyiértékű bitje alacsony szintű, akkor a kliensek közötti prioritást azok indexe határozza meg (az alacsonyabb indexű forrás prioritása magasabb). Ha a regiszterbe írt adat legkisebb helyiértékű bitje magas szintű, akkor a kliensek közötti prioritási sorrend round-robin típusú, a prioritási sorrend minden kiszolgáláskor ennek megfelelően változik.

2-4. táblázat. *A MEMARB modul MMIO-címtérképe*

A *CPU*-n futó programból tehát lehetőség van a *mask_vector* és a *priority_type* regiszterek értékének módosítására az *address_mask_vector* és/vagy az *address_priority_type* címeken végrehajtott MMIO-írás műveletek, vagyis a megfelelően felparaméterezett *store* utasítások végrehajtásával. Az MMIO-adapter állapotgép ennek megfelelő vezérlési állapotait a 2-86. ábra szemlélteti.

Fontos kiemelni, hogy a *CPU* az MMIO-interfészen kizárólag az arbiter konfigurációs regisztereit képes elérni, magát az SDRAM-ot nem. Az SDRAM írásához vagy olvasásához a *CPU*-t az arbiter valamelyik kliensinterfészéhez kell csatlakoztatnunk. Az arbiter kliensinterfészein a cím- és adatvektorok mérete szintézis-paraméter, míg az MMIO-interfész fixen 16-bites címekkel és adatokkal dolgozik. Ráadásul az arbiter alapvető célja egy külső, nagykapacitású memóriamodul rendszerbe illesztése, ezért címbuszainak mérete tipikusan lényegesen nagyobb 16 bitnél. Ahhoz tehát, hogy a *CPU* MMIO-tranzakciók segítségével, vagyis a *store* és *load* gépi utasítások végrehajtásán keresztül tudjon adatokat elhelyezni vagy kiolvasni a külső SDRAM-ból, szükség van egy „átjátszó” modulra az MMIO-buszrendszer és az arbiter szinkron kézfogásos kliensinterfésze között. Ezt a célt szolgálja a rendszertervben (lásd 1-14. ábra) *EXTRAM* névvel jelölt almodul.

A külső SDRAM elérése a *CPU*-ból ◆ Az MMIO-buszrendszer 16-bites cím- és ugyancsak 16-bites adatbuszokkal működik, vagyis egyetlen MMIO-tranzakció legfeljebb 65.536 „célpontot” tud megkülönböztetni és legfeljebb 16 bit információt képes a megcímzett célpontnak

Verilog

```
idle: begin
  s_ready_2_cpu <= 1;

  // Handle MMIO write requests...
  if ( write_strobe_from_cpu_df1 == 1 && cs == 1 ) begin
    s_ready_2_cpu <= 0;

    if ( address_from_cpu == address_priority_type ) begin
      priority_type <= data_from_cpu[0]; state_mmio <= wait_for_deassert_strobes;
    end else if ( address_from_cpu == address_mask_vector ) begin
      mask_vector <= data_from_cpu[7:0]; state_mmio <= wait_for_deassert_strobes;
    end else state_mmio <= wait_for_deassert_strobes;

  end


  // Handle MMIO read requests...
  if ( read_strobe_from_cpu_df1 == 1 && cs == 1 ) begin
    s_ready_2_cpu <= 0;

    if ( address_from_cpu == address_priority_type ) begin
      s_data_2_cpu <= 0; s_data_2_cpu[0] <= priority_type; state_mmio <= wait_for_deassert_strobes;
    end else if ( address_from_cpu == address_mask_vector ) begin
      s_data_2_cpu <= 0; s_data_2_cpu[7:0] <= mask_vector; state_mmio <= wait_for_deassert_strobes;
    end else state_mmio <= wait_for_deassert_strobes;

  end

end

wait_for_deassert_strobes: if (
                             read_strobe_from_cpu_df1 == 0 &&
                             write_strobe_from_cpu_df1 == 0
                           ) state_mmio <= idle;
```

2-86. ábra. *A MEMARB modul MMIO-adapter állapotgépének vezérlési állapotai*

elküldeni vagy onnan kiolvasni. Egy aránylag kis kapacitású tároló esetén – mint amilyen az előző pontban tárgyalt, on-chip RAM blokkokat használó *BRAM* modul – ez nem is probléma, minden MMIO-tranzakció egy-egy memóriaírásnak vagy -olvasásnak feleltethető meg. A nagy kapacitású külső tárolók MMIO rendszerbe illesztésekor azonban a 16 bit általában kevés, egyetlen MMIO-tranzakció tehát nem elegendő az egyetlen memóriahozzáférés által igényelt információmennyiség átvitelére. A problémát úgy oldjuk meg, hogy e nagy mennyiségű cím- és adatinformációt igénylő memória-hozzáféréseket nem egy, hanem több MMIO-tranzakcióval vezényeljük le. Az egyszerűség kedvéért tételezzük fel, hogy a külső memória 32-bites cím- és adatbuszokkal rendelkezik. Ekkor a külső memória írása és olvasása a következő MMIO-tranzakciókra bontható fel:

1. MMIO-írás: A külső memória egy rekeszének címét tároló 32-bites regiszter felső 16 bitjének beállítása.
2. MMIO-írás: A külső memória egy rekeszének címét tároló 32-bites regiszter alsó 16 bitjének beállítása.
3. MMIO-írás: Írás esetén a külső memóriába beírandó adatot tároló 32-bites regiszter felső 16 bitjének beállítása.
4. MMIO-írás: Írás esetén a külső memóriába beírandó adatot tároló 32-bites regiszter alsó 16 bitjének beállítása.
5. MMIO-írás: Írás vagy olvasásművelet kezdeményezése a külső memória felé a 32-bites cím- és adatregiszterekben tárolt értékek felhasználásával. Olvasás esetén a külső memóriából kiolvasott 32-bites adat elhelyezése egy 32-bites belső regiszterben.

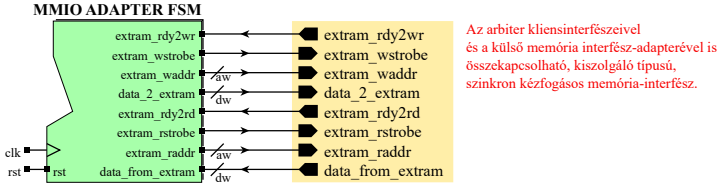
6. MMIO-olvasás: Olvasás esetén a külső memóriából kiolvasott adatot tároló 32-bites regiszter felső 16 bitjének átmásolása a *CPU* általános célú regisztertömbjébe.
7. MMIO-olvasás: Olvasás esetén a külső memóriából kiolvasott adatot tároló 32-bites regiszter alsó 16 bitjének átmásolása a *CPU* általános célú regisztertömbjébe.

A nagy kapacitású külső memória illesztéséhez tehát szükségünk lesz egy olyan MMIO perifé-ria-áramkörre, amelynek szolgáltatásai a fenti lépéssorozat végrehajtását lehetővé teszik. Ezt a célt szolgálja a rendszerterv *EXTRAM* modulja, amelynek címtérképét a 2-5. táblázat mutatja be.

Cím azonosítója és típusa		A címhez rendelt funkció leírása
address_extram_address_buffer_MSW	W	A külső memória legfeljebb 32-bites címét tároló regiszter felső (MSW) és alsó (LSW) két bájtjának címei.
address_extram_address_buffer_LSW	W	
address_extram_data_buffer_MSW	RW	A külső memóriába írandó és az onnan kiolvasott, legfeljebb 32-bites adatot tároló regiszter felső (MSW) és alsó (LSW) két bájtjának címei.
address_extram_data_buffer_LSW	RW	
address_initiate_extram_access	W	E regiszter legkisebb helyiértékű bitjébe 1-et írva a külső memória olvasása, nullát írva pedig annak írása kezdeményezhető.

2-5. táblázat. *Az EXTRAM modul MMIO-címtérképe*
*MSW: **M**ost **S**ignificant **W**ord, LSW: **L**east **S**ignificant **W**ord*

Szerkezetét tekintve az *EXTRAM* modul igen egyszerű (lásd 2-87. ábra); mivel az MMIO-adapter állapotgép közvetlenül is képes kezelni az arbiter kliensinterfészének jeleit, egyéb szerkezeti elemekre nincs szükség.[69] A toplevel integrációt megkönnyítendő célszerű az interfészjelek méretét szintézis-paraméterként *(aw: address width, dw: data width)* megvalósítani függetlenül attól, hogy maguk a belső cím- és adatregiszterek fixen 32-bitesek.



2-87. ábra. *Az EXTRAM modul belső szerkezete*

A 2-88. ábra az *EXTRAM* modul MMIO-adapter állapotgépének vezérlési állapotait mutatja be. Az állapotgép VHDL-leírásából jól látható, hogy az *extram_address_buffer* és *extram_data_buffer* regiszterek egyszerű belső regiszterekként vannak megvalósítva, MMIO-interfészen keresztüli írásuk és olvasásuk a külső memóriától tulajdonképpen független; a *CPU*-n futó program felelőssége gondoskodni arról, hogy a külső memóriához fordulás előtt e regiszterek megfelelően be legyenek állítva. Csak ezután kerülhet sor az *address_initiate_extram_access* címen végrehajtott MMIO-írás műveletre (2-88. ábra: 16. sor). Az ezt kiváltó *store* utasítás adatmezőjének legkisebb helyiértékű bitje határozza meg, hogy a külső memóriába írunk vagy onnan olvasunk (2-88. ábra: 36. sor). A külső memória írását és olvasását végző *write_1...2* és *read_1...2* vezérlési állapotok már ismerősek

[69] Érdemes megjegyezni, hogy ha a rendszerben nincs arbiter, akkor az *EXTRAM* modul – az interfészek regularitásának köszönhetően – akár közvetlenül a külső memória interfészadapterének felhasználói interfészére (lásd 4-111. ábra bal oldalá la) is csatlakoztatható.

VHDL

```
when idle =>
  s_ready_2_cpu <= '1';

  -- Handle MMIO write requests...
  if ( write_strobe_from_cpu_df1 = '1' and cs = '1' ) then
   s_ready_2_cpu <= '0';
   |
   | if ( unsigned(address_from_cpu) = address_extram_address_buffer_MSW ) then
   |   extram_address_buffer(31 downto 16) <= data_from_cpu; state <= wait_for_deassert_strobes;
   | elsif ( unsigned(address_from_cpu) = address_extram_address_buffer_LSW ) then
   |   extram_address_buffer(15 downto 0) <= data_from_cpu; state <= wait_for_deassert_strobes;
   | elsif ( unsigned(address_from_cpu) = address_extram_data_buffer_MSW ) then
   |   extram_data_buffer(31 downto 16) <= data_from_cpu; state <= wait_for_deassert_strobes;
   | elsif ( unsigned(address_from_cpu) = address_extram_data_buffer_LSW ) then
   |   extram_data_buffer(15 downto 0) <= data_from_cpu; state <= wait_for_deassert_strobes;
   | elsif ( unsigned(address_from_cpu) = address_initiate_extram_access ) then state <= extram_access;
   | else state <= wait_for_deassert_strobes;
   | end if;
   |
  end if;


  -- Handle MMIO read requests...
  if ( read_strobe_from_cpu_df1 = '1' and cs = '1') then
   s_ready_2_cpu <= '0';
   |
   | if ( unsigned(address_from_cpu) = address_extram_data_buffer_MSW ) then
   |   s_data_2_cpu <= extram_data_buffer(31 downto 16); state <= wait_for_deassert_strobes;
   | elsif ( unsigned(address_from_cpu) = address_extram_data_buffer_LSW ) then
   |   s_data_2_cpu <= extram_data_buffer(15 downto 0); state <= wait_for_deassert_strobes;
   | else state <= wait_for_deassert_strobes;
   | end if;
   |
  end if;

when extram_access => if ( data_from_cpu(0) = '0' ) then                    -- write EXTRAM
                      | extram_waddr <= extram_address_buffer(aw-1 downto 0);
                      | data_2_extram <= extram_data_buffer(dw-1 downto 0);
                      | extram_wstrobe <= '1';
                      | state <= write_1;
                      else                                                  -- read EXTRAM
                      | extram_raddr <= extram_address_buffer(aw-1 downto 0);
                      | extram_rstrobe <= '1';
                      | state <= read_1;
                      end if;

when write_1 => state <= write_2;

when write_2 => if ( extram_rdy2wr = '1' ) then
                  extram_wstrobe <= '0'; state <= wait_for_deassert_strobes;
                end if;

when read_1 => state <= read_2;

when read_2 => if ( extram_rdy2rd = '1' ) then
               | extram_rstrobe <= '0';
               | extram_data_buffer(31 downto dw) <= (others => '0');
               | extram_data_buffer(dw-1 downto 0) <= data_from_extram;
               | state <= wait_for_deassert_strobes;
               end if;

when wait_for_deassert_strobes => if (
                                    read_strobe_from_cpu_df1 = '0' and
                                    write_strobe_from_cpu_df1 = '0'
                                  ) then state <= idle;
                                  end if;
```

2-88. ábra. *Az EXTRAM modul MMIO-adapter állapotgépének vezérlési állapotai*

lehetnek, hiszen ez egy szokásos szinkron kézfogásos memória-interfész, aminek állapotgépes vezérlését már a *BRAM* (lásd 2-81. ábra: 23–45. sor) és a *MEMARB* (lásd 2-85. ábra: 18–34. sor) moduloknál is meg kellett oldanunk.

2.7. Lokális adatgyűjtés, időzítők

Áttanulmányozásra javasolt alfejezetek:

4.1.1. *Segédmodulok – Órajel-előosztó (prescaler)*
4.4.4. *Katalógus-áramkörök interfészadapterei – I²C A/D-átalakító*
4.4.5. *Katalógus-áramkörök interfészadapterei – I²C hőmérséklet-érzékelő*
4.4.6. *Katalógus-áramkörök interfészadapterei – I²C RTCC áramkör*

2.7.1. Adatgyűjtő modul

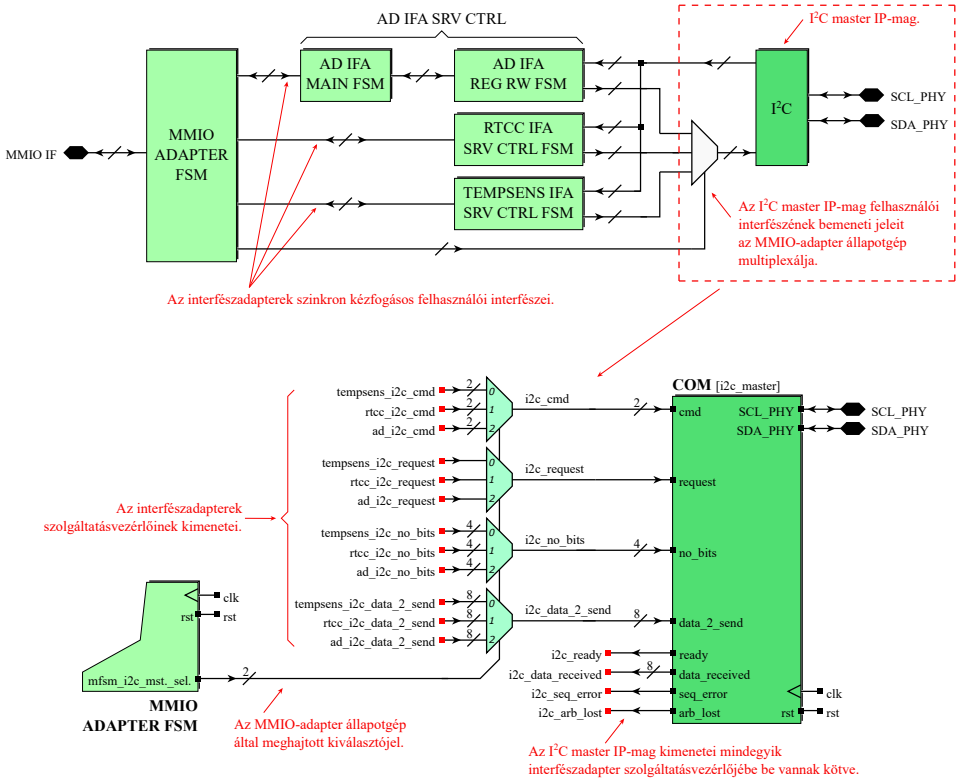
A funkcióbővítő rendszer adatgyűjtő moduljának feladata, hogy MMIO-interfészen keresztül tegye hozzáférhetővé három katalógus-áramkör; egy négycsatornás A/D-átalakító, egy hőmérséklet-érzékelő és egy valósidejű óra/naptár adatait. A memória jellegű katalógus-áramkörökhöz hasonlóan e külső eszközöket is egy-egy interfészadapteren keresztül illesztjük a rendszerhez, amelyek a katalógus-áramkörök szolgáltatásait – vagy azok egy részét – a lehető legszélesebb körű alkalmazhatóság érdekében egy-egy szinkron kézfogásos interfészre képezik le. Az interfészadapterek működésének és belső felépítésének részleteit rendre a 4.4.4., a 4.4.5. és a 4.4.6. alfejezetek mutatják be.

Mindhárom katalógus-áramkör I²C-interfésszel van felszerelve, és mindhárom interfészadapter egymagában is használható IP-magként áll rendelkezésre, vagyis mindegyikben megtalálható a felhasználó és a katalógus-áramkör közötti adatcsere I²C-protokollnak (lásd 5.2.3. Függelék) megfelelő lebonyolításáért felelős, ugyancsak újrafelhasználható I²C master IP-mag. Megtehetjük tehát, hogy az adatgyűjtésre szánt MMIO periféria-áramkör RTL-modelljében egyszerűen példányosítjuk mindhárom interfészadaptert, és azok I²C-buszjeleit a megfelelő katalógus-áramkör kivezetéseivel összekapcsoljuk. Bizonyos értelemben ez a lehető legegyszerűbb megoldás ugyan, de az így kapott áramkör erőforrás-felhasználása némileg nagyobb lesz a feltétlenül szükségesnél, ti. így nem használjuk ki azt, hogy az I²C valójában egy busz, vagyis egyetlen I²C master akár mindhárom katalógus-áramkör kezelésére is alkalmas lehet. A leendő áramkör méretének – közvetetten egyúttal fogyasztásának – csökkentése érdekében az adatgyűjtő modulban nem az interfészadapterek toplevel moduljait, hanem azoknak csak a szolgáltatásvezérlő állapotgépeit példányosítjuk, amelyekhez egyetlen, időosztásos alapon használt I²C master IP-magot illesztünk. Így két I²C master IP-magot és két pár kivezetést spórolhatunk meg, aminek az „ára" csupán az I²C master IP-mag megosztott használatát lehetővé tévő néhány multiplexer és az azok vezérléséhez szükséges többlet logika. Az így kapott struktúrát – az MMIO periféria-áramkörök általános felépítését szemléltető 2-74. ábrának megfelelően – egy MMIO-adapter állapotgéppel tesszük teljessé. Az adatgyűjtő modul egyszerűsített blokkvázlatát a 2-89. ábra szemlélteti (vö. 4-124., 4-134. és 4-140. ábra).

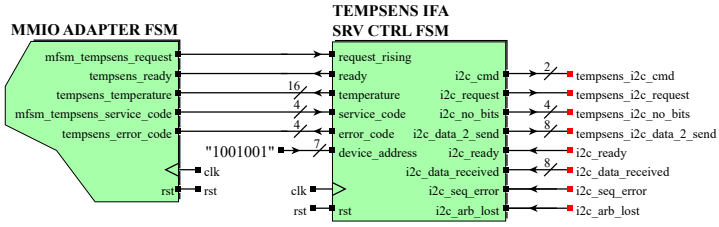
A blokkvázlat alapján jól látható, hogy az MMIO-adapter állapotgép feladata ezúttal az, hogy három különböző szinkron kézfogásos interfészt képezzen le írható és olvasható regiszterekre, miközben gondoskodik az I²C master IP-mag bemeneti multiplexereinek megfelelő vezérléséről is. E leképezés egy lehetséges megvalósítását az alábbiakban interfészadapterenként külön mutatjuk be.

A 2-90. ábrán az MMIO-adapter állapotgép és a hőmérséklet-érzékelő IFA szolgáltatásvezérlő állapotgépének kapcsolata látható. Az MMIO-adapter az IFA felhasználói interfészének szinte

minden jelét közvetlenül vezérli. Kivételt jelent a *device_address* bemenet, amit egy fix értékre kapcsolunk.[70]



2-89. ábra. *Az adatgyűjtő modul egyszerűsített blokkvázlata*



2-90. ábra. *Az MMIO-adapter állapotgép és a hőmérséklet-érzékelő IFA szolgáltatásvezérlő állapotgépének kapcsolata*

[70] Az I²C-interfészű katalógus-áramkörök többsége ad némi mozgásteret az eszköz címének az alkalmazási környezettől függő beállítására annak érdekében, hogy az egyazon buszra felfűzött eszközök között elkerülhető legyen a címütközés. A lokális adatgyűjtésre használt A/D-átalakító és hőmérséklet-érzékelő egyaránt biztosítja ezt a lehetőséget, amit az általános felhasználásra szánt interfészadapterek le is képeznek felhasználói interfészükre a *device_address* bemeneten keresztül. Az RTCC katalógus-áramkör ilyen konfigurációs lehetőséget nem biztosít. Ez az eszköz a fix 1101111_2 és 1010111_2 címeket foglalja le. Az előbbi címen az RTCC-szolgáltatások érhetők el, az utóbbi pedig egy általános célú tárolót (EEPROM) rejt, amit a 4.4.6. pontban bemutatott interfészadapter nem képez le a felhasználói interfészére. Mindezekkel az információkkal az MMIO adatgyűjtő modul felhasználóját nem akarjuk terhelni, ezért az I²C-címeket ezen a hierarchiaszinten már nem kezeljük paraméterként, azokat az MMIO adatgyűjtő modulban fix értékekre állítjuk be.

A hőmérséklet-érzékelő IFA-tól[71] a hőmérséklettel arányos, 16 biten digitalizált érték egy szinkron kézfogásos tranzakcióval kérdezhető le. A kiolvasott vektor mérete tehát összhangban van az MMIO-buszrendszer adatbuszainak méretével, ami egy nagyon egyszerű címtérkép létrehozását teszi lehetővé (lásd 2-6. táblázat).

Cím azonosítója és típusa		A címhez rendelt funkció leírása
address_temperature	R	A hőmérséklettel arányos, 16-biten digitalizált érték címe.
address_error_reg	R	A hőmérséklet-érzékelő interfészadapterétől származó hibakód címe.

2-6. táblázat. *Az adatgyűjtő modul címtérképének a hőmérséklet-érzékelőre vonatkozó részlete*

Az adatgyűjtő modul egy külön címet foglal le a hőmérséklet-adat lekérdezéséhez, így az egyetlen MMIO-olvasással, vagyis egyetlen, megfelelően felparaméterezett *load* utasítás végrehajtásával betölthető a *CPU* általános célú regisztertömbjébe.

Az adatgyűjtő modul egy közös I²C-buszra felfűzve kezeli a három katalógus-áramkört. Ezen az I²C-buszon – akár az eszközök hibája, akár jelintegritási problémák okán – bármely I²C-tranzakció során előfordulhat busz-arbitrációs hiba vagy az, hogy a slave eszköz nem küld megfelelő nyugtázást (ACK) a master kérésére. Az MMIO-adapter állapotgép erről úgy értesülhet, hogy a megszólított katalógus-áramkör interfészadapterének hibakód kimenete *(error_code)* a szinkron kézfogásos tranzakció végeztével nullától különböző értéket vesz fel. Ilyenkor az interfészadapter által visszaadott többi adatvektort érvénytelennek kell tekinteni, és a jelenségről a felhasználót valami módon értesíteni kell. E célból definiáljuk a három interfészadapter által közösen használt, csak-olvasható *error_reg* regisztert, amibe minden I²C-tranzakció után betöltjük az éppen megszólított IFA *error_code* kimenetének értékét. Az adatgyűjtő modul használatakor tehát a *CPU*-n futó programban – egy, az *address_error_reg* címen végrehajtott *load* utasítással – gondoskodni kell e hibaregiszter ellenőrzéséről minden olyan *store* vagy *load* utasítás végrehajtása után, ami I²C-tranzakciót váltott ki.

Az MMIO-adapter állapotgépnek a hőmérséklet-érzékelő 2-6. táblázatbeli címtérképét megvalósító vezérlési állapotait a 2-91. ábra mutatja be. A címtérkép nem tartalmaz a hőmérséklet-érzékelőre vonatkozó írható címeket, így az MMIO-adapter állapotgép *idle* állapotának az MMIO íráskéréseket kezelő része (2-91. ábra: 4. sor) ezúttal nem releváns. Az *address_temperature* címen végrehajtott olvasás hatására (2-91. ábra: 11. sor) az állapotgép a *read_temperature_1...3* állapotsorozaton lépked végig, ami a hőmérséklet-érzékelő IFA szinkron kézfogásos felhasználói interfészén hajt végre egy tranzakciót. Ne feledjük, hogy az I²C master IP-magot a három szolgáltatásvezérlő állapotgép megosztva használja, ezért a kézfogásos tranzakciók előtt be kell állítanunk a 2-89. ábrán látható multiplexerek kiválasztó bemenetét *(mfsm_i2c_master_selector)* úgy, hogy az I²C master modul bemeneteit az éppen megszólított IFA kimenetei hajtsák meg (2-91. ábra: 22. sor). Csak ezután kerülhet sor az IFA szolgáltatásának kiválasztására és a szinkron kézfogásos tranzakció kezdeményezésére (2-91. ábra: 23–24. sor). A tranzakció végeztével az MMIO-adapter állapotgépnek az olvasás művelet lezárása előtt még két feladata van; egyrészt az IFA által visszaadott adatot az MMIO-buszrendszer kimeneti adatbuszára kell kapcsolnia (2-91. ábra: 31. sor), illetve az interfészadapternek az I²C-buszon esetleg bekövetkezett hibákra utaló *tempsensor_error_code* kimenetét

[71] Az adatgyűjtő modulon belül valójában nem a teljes IFA-t, csak annak szolgáltatásvezérlő állapotgépét példányosítottuk. Az egyszerűbb megfogalmazás érdekében erre az állapotgépre egyszerűen csak interfészadapterként hivatkozunk.

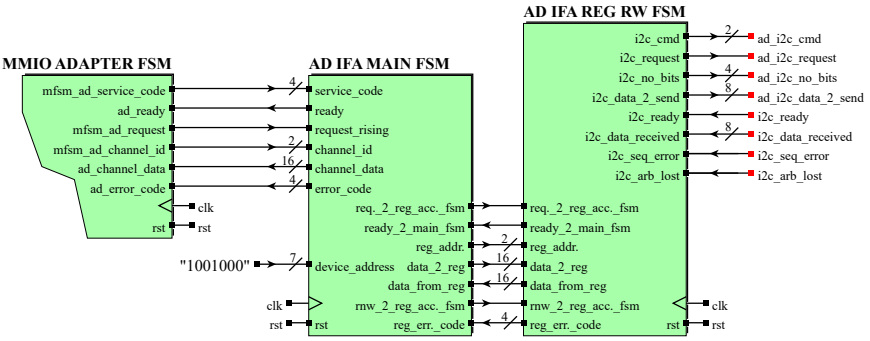
VHDL

```
1  when idle => s_ready_2_cpu <= '1';
2
3              -- Handle MMIO write requests...
4              if ( write_strobe_from_cpu_df1 = '1' and cs = '1' ) then s_ready_2_cpu <= '0'; [...] end if;
5
6
7              -- Handle MMIO read requests...
8              if ( read_strobe_from_cpu_df1 = '1' and cs = '1') then
9                s_ready_2_cpu <= '0';
10               |
11               if ( unsigned(address_from_cpu) = address_temperature ) then state <= read_tempsensor_1;
12               elsif ( [...] ) then [...]
13               elsif ( unsigned(address_from_cpu) = address_error_reg ) then
14                 s_data_2_cpu <= X"00" & mfsm_error_reg;
15                 state <= wait_for_deassert_strobes;
16               else state <= wait_for_deassert_strobes;
17               end if;
18             end if;
19
20 [...]
21
22 when read_tempsensor_1 => mfsm_i2c_master_selector <= "00";   -- tempsensor
23                          mfsm_tempsens_service_code <= X"0"; -- get temperature
24                          mfsm_tempsens_request <= '1';
25                          state <= read_tempsensor_2;
26
27 when read_tempsensor_2 => mfsm_tempsens_request <= '0';
28                          state <= read_tempsensor_3;
29
30 when read_tempsensor_3 => if ( tempsens_ready = '1' ) then
31                          | s_data_2_cpu <= tempsens_temperature;
32                          | mfsm_error_reg <= X"0" & tempsens_error_code;
33                          | state <= wait_for_deassert_strobes;
34                          end if;
35
36 [...]
37
38 when wait_for_deassert_strobes => if (
39                                     read_strobe_from_cpu_df1 = '0' and
40                                     write_strobe_from_cpu_df1 = '0'
41                                   ) then state <= idle;
42                                   end if;
```

2-91. ábra. *Az adatgyűjtő modul MMIO-adapter állapotgépének a hőmérséklet-érzékelőre vonatkozó vezérlési állapotai*

a nyolcbites *mfsm_error_reg* regiszterbe kell betöltenie (2-91. ábra: 32. sor). Ez utóbbi regisztert ugyancsak megosztva használjuk a három interfészadapter *error_code* kimeneteinek tárolására; értékét minden szinkron kézfogásos tranzakció végén az éppen megszólított IFA *error_code* kimenetével írjuk felül. E regiszter egy külön MMIO-olvasással kérdezhető le (2-91. ábra: 13–15. sor). Ahogy korábban is említettük, az adatgyűjtő modulhoz készített assembly programrészletben célszerű e hibaregisztert minden, I²C-tranzakciót igénylő művelet után kiolvasni, erre azonban a periféria-áramkör a felhasználót nem „kényszeríti rá”. Elképzelhető az, hogy egy egész sor I²C-tranzakció lezajlik, mire a hibaregiszter kiolvasására sor kerül, ilyenkor csak a legutóbb bekövetkezett hibakódot tudjuk visszaadni. Mindezek okán magába a hibaregiszterbe az IFA-k négybites *error_code* kimenetén kívül a hibakód forrását is érdemes belekódolni, erre szolgál a nyolcbites hibaregiszter felső négy bitje, amit a hőmérséklet-érzékelővel, az RTCC-vel és az A/D-átalakítóval való kölcsönhatás után rendre a 0, 1 vagy 2 értékre állítunk be (2-91. ábra: 32. sor, 2-93. ábra: 43. sor, 2-95. ábra: 42. sor, 2-96. ábra: 51. sor).

A 2-92. ábrán az MMIO-adapter állapotgép és az A/D-átalakító IFA állapotgépeinek kapcsolata látható. A szerkezet hasonló a hőmérséklet-érzékelőnél látottakhoz; az MMIO-adapter állapotgép az IFA *MAIN FSM* állapotgépéhez kapcsolódik, amelynek az I²C-címet meghatározó *device_address* bemenetét egy fix értékre állítjuk be.



2-92. ábra. *Az MMIO-adapter állapotgép és az A/D-átalakító IFA állapotgépeinek kapcsolata*

A négycsatornás, 16-bites A/D-átalakító interfészadapterétől az egyes csatornákon mérhető feszültségek digitalizált értékei csatornánként egy-egy külön szinkron kézfogásos tranzakcióval kérdezhetők le. A kiolvasandó vektorok mérete tehát ezúttal is megegyezik az MMIO-buszrendszer adatbuszainak méretével, így a címtérkép felhasználói (programozói) szempontból itt is kényelmesen használható lesz (lásd 2-7. táblázat).

Cím azonosítója és típusa		A címhez rendelt funkció leírása
address_ad_ch_0	R	A nullás indexű csatorna 16-bites digitalizált értékének címe.
address_ad_ch_1	R	Az egyes indexű csatorna 16-bites digitalizált értékének címe.
address_ad_ch_2	R	A kettes indexű csatorna 16-bites digitalizált értékének címe.
address_ad_ch_3	R	A hármas indexű csatorna 16-bites digitalizált értékének címe.
address_error_reg	R	Az A/D-átalakító interfészadapterétől származó hibakód címe.

2-7. táblázat. *Az adatgyűjtő modul címtérképének az A/D-átalakítóra vonatkozó részlete*

Az adatgyűjtő modulban tehát egy-egy külön címet foglalunk le az egyes A/D-csatornák adatainak lekérdezéséhez; bármely digitalizált érték egyetlen MMIO-olvasással, vagyis egy megfelelően felparaméterezett *load* utasítás végrehajtásával lekérdezhető. Az MMIO-adapter állapotgépnek az e címtérképet megvalósító vezérlési állapotait a 2-93. ábra mutatja be.

Ha összevetjük az MMIO-adapter állapotgépnek a hőmérséklet-érzékelő és az A/D-átalakító címtérképét megvalósító részleteit, azt látjuk, hogy a két eset nagyon hasonló, a különbség csupán az, hogy MMIO-olvasáskor ezúttal nemcsak egy, hanem az egyes A/D-csatornákra vonatkozó összes cím *(address_ad_ch_0...3)* a *read_ad_1...3* állapotsorozatba irányítja a végrehajtást (2-93. ábra: 12–16. sor), ahol egy szinkron kézfogásos tranzakció zajlik le, ezúttal az MMIO-adapter állapotgép és az A/D-átalakító IFA *MAIN FSM* állapotgépe között. E tranzakció annyiban összetettebb a hőmérséklet-érzékelőnél látottaknál, hogy mivel a négy cím az A/D-átalakító négy külön csatornájára vonatkozik, ezért a *read_ad_1* állapotban egy további címdekódolási lépést kell elvégeznünk, amiben meghatározzuk az A/D-átalakító interfészadapterének *channel_id* bemenetére kapcsolandó értéket (2-93. ábra: 30–34. sor). A szinkron kézfogásos tranzakció végén az IFA által visszaadott adatot az MMIO-buszrendszer kimeneti adatbuszára kapcsoljuk, a hibakódot pedig a közös hibaregiszterbe töltjük (2-93. ábra: 43–44. sor), majd az MMIO-olvasás tranzakciót a szokásos módon a *wait_for_deassert_strobes* állapotban zárjuk le.

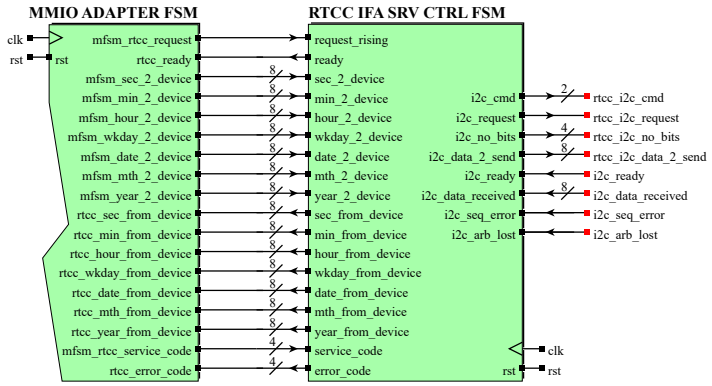
VHDL

```
when idle => s_ready_2_cpu <= '1';

             -- Handle MMIO write requests...
             if ( write_strobe_from_cpu_df1 = '1' and cs = '1' ) then s_ready_2_cpu <= '0'; [...] end if;

             -- Handle MMIO read requests...
             if ( read_strobe_from_cpu_df1 = '1' and cs = '1') then
               s_ready_2_cpu <= '0';
               |
               | if ( [...] ) then [...]
               | elsif (
               | | unsigned(address_from_cpu) = address_ad_ch_0 or
               | | unsigned(address_from_cpu) = address_ad_ch_1 or
               | | unsigned(address_from_cpu) = address_ad_ch_2 or
               | | unsigned(address_from_cpu) = address_ad_ch_3
               | ) then state <= read_ad_1;
               | elsif ( [...] ) then [...]
               | elsif ( unsigned(address_from_cpu) = address_error_reg ) then
               |   s_data_2_cpu <= X"00" & mfsm_error_reg;
               |   state <= wait_for_deassert_strobes;
               | else state <= wait_for_deassert_strobes;
               | end if;
             end if;

[...]

when read_ad_1 => mfsm_i2c_master_selector <= "10"; -- AD
                  mfsm_ad_service_code <= X"0";     -- get channel data

                  if    ( unsigned(address_from_cpu) = address_ad_ch_0 ) then mfsm_ad_channel_id <= "00";
                  elsif ( unsigned(address_from_cpu) = address_ad_ch_1 ) then mfsm_ad_channel_id <= "01";
                  elsif ( unsigned(address_from_cpu) = address_ad_ch_2 ) then mfsm_ad_channel_id <= "10";
                  else mfsm_ad_channel_id <= "11";
                  end if;

                  mfsm_ad_request <= '1';
                  state <= read_ad_2;

when read_ad_2 => mfsm_ad_request <= '0';
                  state <= read_ad_3;

when read_ad_3 => if ( ad_ready = '1' ) then
                    | s_data_2_cpu <= ad_channel_data;
                    | mfsm_error_reg <= X"2" & ad_error_code;
                    | state <= wait_for_deassert_strobes;
                  end if;

[...]

when wait_for_deassert_strobes => if (
                                    read_strobe_from_cpu_df1 = '0' and
                                    write_strobe_from_cpu_df1 = '0'
                                  ) then state <= idle;
                                  end if;
```

2-93. ábra. *Az adatgyűjtő modul MMIO-adapter állapotgépének az A/D-átalakítóra vonatkozó vezérlési állapotai*

A 2-94. ábra az MMIO-adapter állapotgép és az RTCC IFA szolgáltatásvezérlő állapotgépének kapcsolatát szemlélteti, ami a korábbiakhoz képest semmi újdonságot nem tartogat; az IFA felhasználói interfészének minden jelét hozzákapcsoljuk az MMIO-adapter állapotgéphez. Az RTCC nem teszi lehetővé az I²C-cím hardveres konfigurálását, így az interfészadapternek sincs *device_address* bemenete, ezúttal tehát annak fix értékre állításával sem kell foglalkoznunk.

A periféria-áramkör RTCC-re vonatkozó címtérképe (lásd 2-8. táblázat) a korábbiakhoz képest bonyolultabb, aminek a fő oka az, hogy a katalógus-áramkörnek az időt és dátumot számontartó belső számlálói nemcsak kiolvashatók, de be is állíthatók.



2-94. ábra. *Az MMIO-adapter állapotgép és az RTCC IFA szolgáltatásvezérlő állapotgépének kapcsolata*

Cím azonosítója és típusa		A címhez rendelt funkció leírása
address_rtcc_sec_inbuffer	W	Az RTCC katalógus-áramkör számlálóiba beírni kívánt értékeket tároló belső regiszterek címei.
address_rtcc_min_inbuffer	W	
address_rtcc_hour_inbuffer	W	
address_rtcc_wkday_inbuffer	W	
address_rtcc_date_inbuffer	W	
address_rtcc_month_inbuffer	W	
address_rtcc_year_inbuffer	W	
address_init_rtcc	W	Az e címen végrehajtott MMIO-írás kezdeményezi az RTCC IFA-nak az eszköz belső számlálóinak felülírását végző szolgáltatását.
address_update_rtcc_outbuffers	W	Az e címen végrehajtott MMIO-írás kezdeményezi az RTCC IFA-nak az eszköz belső számlálóinak kiolvasását végző szolgáltatását.
address_rtcc_sec_outbuffer	R	Az RTCC katalógus-áramkör számlálóinak legutóbb kiolvasott értékeit tároló belső regiszterek címei.
address_rtcc_min_outbuffer	R	
address_rtcc_hour_outbuffer	R	
address_rtcc_wkday_outbuffer	R	
address_rtcc_date_outbuffer	R	
address_rtcc_month_outbuffer	R	
address_rtcc_year_outbuffer	R	
address_error_reg	R	

2-8. táblázat. *Az adatgyűjtő modul címtérképének az RTCC-re vonatkozó részlete*

Az RTCC IFA egyetlen szolgáltatást definiál az eszköz összes belső számlálójának egyidejű felülírására és egyet azok ugyancsak egyidejű kiolvasására. Mivel mindkét esetben összesen 7 × 8 bit adat mozgatásáról van szó – a másik két katalógus-áramkörnél látottakkal ellentétben –, egyetlen MMIO-tranzakcióba nem tudjuk besűríteni sem a számlálók frissítését, sem azok kiolvasását. Ehelyett a periféria-áramkörön belül létrehozunk 7 db bemeneti és 7 db kimeneti buffert

a katalógus-áramkörbe beírandó és onnan kiolvasott értékek ideiglenes tárolására, és e bufferekhez egy-egy külön MMIO-címet rendelve tesszük elérhetővé azokat a *CPU* számára. A bemeneti bufferek írása és a kimeneti bufferek olvasása nem idéz elő semmilyen kölcsönhatást a periféria-áramkör és az RTCC IFA – közvetetten tehát a katalógus-áramkör – között, e célra további két, csak-írható címet definiálunk. Az *address_init_rtcc* címen végrehajtott MMIO-írás a bemeneti bufferek aktuális értékeivel kezdeményezi az RTCC IFA-nak a katalógus-áramkör belső számlálóinak felülírását végző szolgáltatását. Hasonlóképpen, az *address_update_rtcc_outbuffers* címen végrehajtott írás művelettel kezdeményezhető az IFA azon szolgáltatása, ami a katalógus-áramkörből kiolvassa az összes számláló értékét. A kiolvasott értékeket a periféria-áramkör kimeneti buffereibe töltjük.

Az RTCC katalógus-áramkör belső számlálóinak a fenti módszerrel való beállítását lehetővé tévő vezérlési állapotokat a 2-95. ábra mutatja be.

VHDL

```
when idle => s_ready_2_cpu <= '1';

             -- Handle MMIO write requests...
             if ( write_strobe_from_cpu_df1 = '1' and cs = '1' ) then
               s_ready_2_cpu <= '0';

               if ( unsigned(address_from_cpu) = address_rtcc_sec_inbuffer ) then
                 mfsm_sec_2_device <= data_from_cpu(7 downto 0); state <= wait_for_deassert_strobes;
               elsif ( unsigned(address_from_cpu) = address_rtcc_min_inbuffer ) then
                 mfsm_min_2_device <= data_from_cpu(7 downto 0); state <= wait_for_deassert_strobes;
               elsif ( unsigned(address_from_cpu) = address_rtcc_hour_inbuffer ) then
                 mfsm_hour_2_device <= data_from_cpu(7 downto 0); state <= wait_for_deassert_strobes;
               elsif ( unsigned(address_from_cpu) = address_rtcc_wkday_inbuffer ) then
                 mfsm_wkday_2_device <= data_from_cpu(7 downto 0); state <= wait_for_deassert_strobes;
               elsif ( unsigned(address_from_cpu) = address_rtcc_date_inbuffer ) then
                 mfsm_date_2_device <= data_from_cpu(7 downto 0); state <= wait_for_deassert_strobes;
               elsif ( unsigned(address_from_cpu) = address_rtcc_month_inbuffer ) then
                 mfsm_mth_2_device <= data_from_cpu(7 downto 0); state <= wait_for_deassert_strobes;
               elsif ( unsigned(address_from_cpu) = address_rtcc_year_inbuffer ) then
                 mfsm_year_2_device <= data_from_cpu(7 downto 0); state <= wait_for_deassert_strobes;
               elsif ( unsigned(address_from_cpu) = address_init_rtcc ) then state <= init_rtcc_1;
               elsif ( [...] ) then [...]
               else state <= wait_for_deassert_strobes;
               end if;

             end if;

             -- Handle MMIO read requests...
             if ( read_strobe_from_cpu_df1 = '1' and cs = '1') then s_ready_2_cpu <= '0'; [...] end if;

[...]

when init_rtcc_1 => mfsm_i2c_master_selector <= "01";
                    mfsm_rtcc_service_code <= X"1"; -- set timekeeping registers
                    mfsm_rtcc_request <= '1';
                    state <= init_rtcc_2;

when init_rtcc_2 => mfsm_rtcc_request <= '0';
                    state <= init_rtcc_3;

when init_rtcc_3 => if ( rtcc_ready = '1' ) then
                      mfsm_error_reg <= X"1" & rtcc_error_code;
                      state <= wait_for_deassert_strobes;
                    end if;
```

2-95. ábra. *Az adatgyűjtő modul MMIO-adapter állapotgépének az RTCC-re vonatkozó vezérlési állapotai (1)*

Figyeljük meg, hogy a bemeneti bufferek *(mfsm_sec_2_device ... mfsm_year_2_device)* felülírása után az MMIO-tranzakciókat a *wait_for_deassert_strobes* állapotba lépve azonnal lezárjuk, az RTCC interfészadapterrel tehát nem lépünk kapcsolatba. Az *address_init_rtcc* címen végrehajtott MMIO-írás azonban az *init_rtcc_1...3* állapotsorozatba irányítja a végrehajtást, ami egy szinkron kézfogásos

tranzakciót hajt végre az RTCC IFA felhasználói interfészén. A bemeneti bufferek közvetlenül meghajtják az RTCC IFA adatbemeneteit, így azok már be vannak állítva, amikor sor kerül az I²C master IP-mag bemeneti multiplexereinek beállítására, az RTCC IFA szolgáltatásának kiválasztására és a tranzakció kezdeményezésére (2-95. ábra: 33–35. sor). Az RTCC IFA-val végrehajtott szinkron kézfogásos tranzakció végén a közös hibaregiszter értékét ezúttal is frissítjük (2-95. ábra: 42. sor), mielőtt a *wait_for_deassert_strobes* állapotba lépve zárnánk az MMIO-írás tranzakciót. Az RTCC számlálóinak kiolvasását és az ezek értékeit tároló kimeneti bufferekhez való hozzáférést lehetővé tévő vezérlési állapotokat a 2-96. ábra szemlélteti.

VHDL

```
1  when idle => s_ready_2_cpu <= '1';
2
3                -- Handle MMIO write requests...
4                if ( write_strobe_from_cpu_df1 = '1' and cs = '1' ) then
5                  s_ready_2_cpu <= '0';
6                  |
7                  if ( [...] ) then [...]
8                  elsif ( unsigned(address_from_cpu) = address_update_rtcc_outbuffers ) then
9                    state <= read_rtcc_1;
10                 else state <= wait_for_deassert_strobes;
11                 end if;
12                 |
13               end if;
14
15               -- Handle MMIO read requests...
16               if ( read_strobe_from_cpu_df1 = '1' and cs = '1') then
17                 s_ready_2_cpu <= '0';
18                 |
19                 if ( unsigned(address_from_cpu) = address_rtcc_sec_outbuffer ) then
20                   s_data_2_cpu <= X"00" & rtcc_sec_from_device; state <= wait_for_deassert_strobes;
21                 elsif ( unsigned(address_from_cpu) = address_rtcc_min_outbuffer ) then
22                   s_data_2_cpu <= X"00" & rtcc_min_from_device; state <= wait_for_deassert_strobes;
23                 elsif ( unsigned(address_from_cpu) = address_rtcc_hour_outbuffer ) then
24                   s_data_2_cpu <= X"00" & rtcc_hour_from_device; state <= wait_for_deassert_strobes;
25                 elsif ( unsigned(address_from_cpu) = address_rtcc_wkday_outbuffer ) then
26                   s_data_2_cpu <= X"00" & rtcc_wkday_from_device; state <= wait_for_deassert_strobes;
27                 elsif ( unsigned(address_from_cpu) = address_rtcc_date_outbuffer ) then
28                   s_data_2_cpu <= X"00" & rtcc_date_from_device; state <= wait_for_deassert_strobes;
29                 elsif ( unsigned(address_from_cpu) = address_rtcc_month_outbuffer ) then
30                   s_data_2_cpu <= X"00" & rtcc_mth_from_device; state <= wait_for_deassert_strobes;
31                 elsif ( unsigned(address_from_cpu) = address_rtcc_year_outbuffer ) then
32                   s_data_2_cpu <= X"00" & rtcc_year_from_device; state <= wait_for_deassert_strobes;
33                 elsif ( unsigned(address_from_cpu) = address_error_reg ) then
34                   s_data_2_cpu <= X"00" & mfsm_error_reg; state <= wait_for_deassert_strobes;
35                 else state <= wait_for_deassert_strobes;
36                 end if;
37                 |
38               end if;
39
40 [...]
41
42 when read_rtcc_1 => mfsm_i2c_master_selector <= "01"; -- RTCC
43                     mfsm_rtcc_service_code <= X"0";   -- get timekeeping registers
44                     mfsm_rtcc_request <= '1';
45                     state <= read_rtcc_2;
46
47 when read_rtcc_2 => mfsm_rtcc_request <= '0';
48                     state <= read_rtcc_3;
49
50 when read_rtcc_3 => if ( rtcc_ready = '1' ) then
51                       mfsm_error_reg <= X"1" & rtcc_error_code;
52                       state <= wait_for_deassert_strobes;
53                     end if;
```

2-96. ábra. *Az adatgyűjtő modul MMIO-adapter állapotgépének az RTCC-re vonatkozó vezérlési állapotai (2)*

A kimeneti bufferek egy-egy csak-olvasható címen érhetők el, és kiolvasásuk a bemeneti regisztereknél látottakhoz hasonlóan nem vált ki semmilyen kölcsönhatást az RTCC IFA-val. Az RTCC belső számlálóinak az I²C-buszon keresztüli tényleges kiolvasása az *address_update_rtcc_outbuffers* címre végrehajtott írással lehetséges. Ez elsőre furcsának tűnhet, hiszen alapvetően egy, az RTCC-ből kifelé irányuló adatátvitelről van szó. Maga az MMIO-írás azonban – amit a *CPU*-n futó programban egy *store* utasítás vált ki – csupán kezdeményez egy aránylag összetett, sok adat mozgatásával járó műveletsort, tehát a *CPU* és a periféria-áramkör között adatcserére valójában nem kerül sor. Az ilyen látszólagos ellentmondásokat könnyen feloldhatjuk a periféria-áramkörhöz rendelt címek megfelelő elnevezésével. Az *„address_init_rtcc"* és *„address_update_rtcc_outbuffers"* kifejezések cselekvésre felszólító jellege utal arra, hogy e címek mögött egyik esetben sem egy fizikailag is jelenlévő regiszter rejtőzik, hanem valamilyen összetett műveletsor kezdeményezését képezik le az MMIO-interfészre.[72]

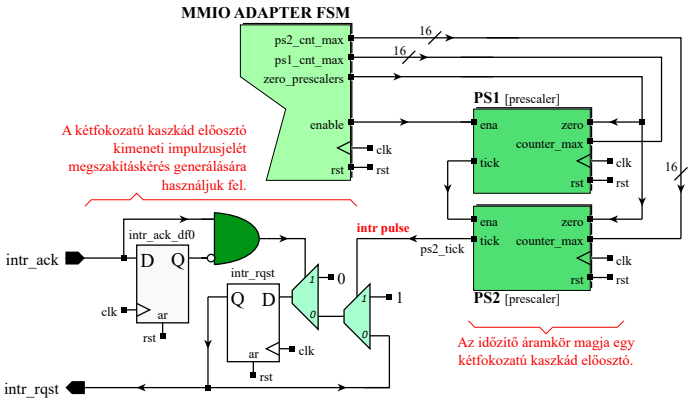
2.7.2. Időzítők

A processzoros rendszerek elmaradhatatlan kelléke a programozható időzítő áramkör. Rendszertervünk szerint a funkcióbővítő alaprendszerben két, MMIO-kompatibilis időzítőre lesz szükségünk (lásd 1-15. ábra); egyiket a *CPU*-n futó szoftverben hasznosítható, előre nem meghatározott szerepkörű, általános időzítőként, a másikat pedig e szoftver esetleges holtpontra jutása esetén a *CPU*-t újraindító watchdogként szeretnénk használni. Az általános célú időzítő feladata az, hogy beállítható időközönként egy-egy megszakításkérést generáljon a *CPU* felé, a watchdog áramköré pedig, hogy egy ugyancsak beállítható időtartam eltelte után – ha ez idő alatt nem állítjuk alapállapotba – meghúzza a *CPU* reset bemenetét. A két modul funkciója tehát alapvetően ugyanaz, ezért ahelyett, hogy két, nagyon hasonló periféria-áramkört készítenénk, megtervezzük a *CPU* megszakítási rendszerével – egyúttal a 2.5. pontban bemutatott megszakításvezérlővel – kompatibilis, a 2-74. ábrán látható általános séma szerint felépülő MMIO időzítőt, és egy egyszerű illesztőlogikával a *CPU* reset bemenetére kapcsolva ezt használjuk fel a watchdog funkció megvalósítására is.

Az adott időközönkénti triggerjel-előállítás a digitális áramkörökben egy igen gyakran előforduló igény, ezért e funkció betöltésére célszerű egy újrafelhasználható IP-magot készíteni. A 4.1.1. pontban bemutatott órajel-előosztó pontosan ezt a célt szolgálja; ha engedélyező bemenete magas szintű, akkor belső számlálójának értékét órajelciklusonként eggyel növeli, majd amikor e számláló értéke eléri a felhasználó által a *counter_max* bemenetre kapcsolt – nemnegatív egészként értelmezett – vektor értékét, akkor *tick* kimenetén egy egyetlen órajelciklus időtartamú, pozitív impulzust állít elő. Ahogy azt a 4-5. ábra is szemlélteti, a kimeneti impulzusok közötti legnagyobb elérhető időtartam több ilyen előosztó sorbakapcsolásával növelhető meg. Az MMIO időzítő-áramkörünk alapja egy ilyen, kétfokozatú kaszkád előosztó struktúra lesz (lásd 2-97. ábra).

---

[72] Annak persze semmi akadálya, hogy az *address_update_rtcc_outbuffers* címet ne csak-írható, hanem csak-olvasható címként definiáljuk, érzékeltetve, hogy itt alapvetően egy befelé irányuló adatátvitelről van szó. Ekkor a 2-96. ábra 8–9. soraiból az erre a címre vonatkozó dekódolást leíró kódrészletet az MMIO-olvasásokat feldolgozó *if-else* szerkezetbe (2-96. ábra: 16–38. sor) kell beágyaznunk. E megoldás azonban ugyancsak ellentmondásos, hiszen így a *CPU* által végrehajtott *load* utasítás nem tud érvényes adatot betölteni a *CPU* belső regisztertömbjébe, mert az RTCC számlálóinak értékei csak a periféria-vezérlőben lévő kimeneti bufferekig „jutottak el". Ezek csak további *load* utasítások végrehajtásával tölthetők be a *CPU* regisztertömbjébe. A két megoldás tehát felhasználói szemmel gyakorlatilag egyenértékű. Az ehhez hasonló ellentmondásos helyzetekben kulcsfontosságú a megfelelő dokumentáció; a periféria-áramkörök címtérképeinek felhasználói dokumentációja természetesen túlmutathat a 2-8. táblázatban rögzített, párszavas definíciókon.



2-97. ábra. *Az időzítő áramkör belső szerkezete*

A kaszkád előosztó mindkét fokozatának *counter_max* bemenetét MMIO-interfészen keresztül beállíthatóvá tesszük, így a megszakításkérések generálása között eltelt idő futásidőben változtatható lesz. Természetesen az előosztók működésének engedélyezését/tiltását és azok belső számlálóinak nullázását is lehetővé tesszük. Ez utóbbi funkció segítségével az MMIO időzítő watchdog áramkörként is használható lesz. Mindezek figyelembevételével alkotjuk meg a periféria-áramkör MMIO-címtérképét (lásd 2-9. táblázat).

Cím azonosítója és típusa		A címhez rendelt funkció leírása
address_ps1_counter_max	W	Az első órajel-előosztó fokozat számlálójának maximális értékét tároló regiszter címe.
address_ps2_counter_max	W	A második órajel-előosztó fokozat számlálójának maximális értékét tároló regiszter címe.
address_enable	W	Az órajel-előosztók működését engedélyező/tiltó, magas aktív vezérlőregiszter címe.
address_reset	W	Írás műveletet végrehajtva ezen a címen az órajel-előosztók belső számlálói nullázódnak.

2-9. táblázat. *Az időzítő-áramkör címtérképe*

Mivel a címtérkép kizárólag csak-írható címeket tartalmaz, amelyek ráadásul egyszerű statikus vagy – a *zero_prescalers* jel esetén – impulzusszerű vezérlőjeleket takarnak, ezért az időzítő MMIO-adapter állapotgépe a korábbiaknál lényegesen egyszerűbb; a minden MMIO periféria-áramkörben megtalálható *idle* és *wait_for_deassert_strobes* állapotokon kívül többre nincs is szükség (lásd 2-98. ábra).

A kaszkád előosztó második fokozatának *tick* kimenetén az *address_ps1/2_counter_max* címekre írt értékektől függő időközönként egy-egy pozitív impulzus jelenik meg. Az MMIO periféria-áramkörök általános szerkezetét szemléltető 2-74. ábrán látható, hogy ez az impulzus közvetlenül bekötethető a *CPU* megszakítási rendszerével való kompatibilitást biztosító logikába *(ps2_tick ≡ intr pulse)*.

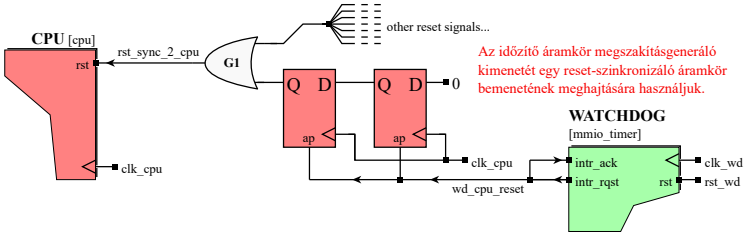
A fenti időzítő áramkör watchdog szerepben való alkalmazásakor az *intr_rqst* kimenetet valójában nem megszakítási vonalként használjuk, tehát azt nem a megszakításvezérlő valamelyik bemenetére és nem is a *CPU* megszakításkérő bemenetére kapcsoljuk, hanem annak *rst* bemenetét

Verilog

```
idle: begin
  s_ready_2_cpu <= 1;

  // Handle MMIO write requests...
  if ( write_strobe_from_cpu_df1 == 1 && cs == 1 ) begin
    s_ready_2_cpu <= 0;

    if ( address_from_cpu == address_ps1_counter_max ) begin
      ps1_cnt_max <= data_from_cpu; state <= wait_for_deassert_strobes;
    end else if ( address_from_cpu == address_ps2_counter_max ) begin
      ps2_cnt_max <= data_from_cpu; state <= wait_for_deassert_strobes;
    end else if ( address_from_cpu == address_enable ) begin
      enable <= data_from_cpu[0]; state <= wait_for_deassert_strobes;
    end else if ( address_from_cpu == address_reset ) begin
      zero_prescalers <= 1; state <= wait_for_deassert_strobes;
    end else state <= wait_for_deassert_strobes;

  end

  // Handle MMIO read requests...
  if ( read_strobe_from_cpu_df1 == 1 && cs == 1 ) begin
    s_ready_2_cpu <= 0; state <= wait_for_deassert_strobes;
  end

end

wait_for_deassert_strobes: begin
  zero_prescalers <= 0;
  if (read_strobe_from_cpu_df1 == 0 && write_strobe_from_cpu_df1 == 0 ) state <= idle;
end
```

2-98. ábra. *Az időzítő áramkör MMIO-adapter állapotgépének vezérlési állapotai*

hajtjuk meg vele. A rendszertervben rögzítettek értelmében feltételeznünk kell, hogy a *CPU* és az MMIO periféria-áramkörök – így maga a watchdogként használt időzítő is – egymástól független, aszinkron órajelekről járnak, ezért a toplevel modulban az *intr_rqst* kimenetre egy reset-szinkronizáló fokozatot kell beépítenünk. Ennek mikéntjét a 2-99. ábra egy aszinkron meghúzott, szinkron elengedett double-flop szinkronizáló példáján keresztül szemlélteti.



2-99. ábra. *Az MMIO időzítő bekötése watchdog áramkörként a CPU reset bemenetére*

A *CPU*-t általában nemcsak a watchdog áramkörrel szeretnénk reset állapotba kényszeríteni, ezért a watchdoghoz tartozó, szinkronizált reset jelet a többi reset-forrás – ugyancsak szinkronizált – reset jelével kell *OR* kapcsolatba hoznunk (lásd 2-99. ábra: *G1*).

2.8. Az üzemeltetői interfész áramkörei

Áttanulmányozásra javasolt alfejezetek:

4.2. *On-chip adattárolás*
4.3.1. *A soros kommunikáció alapáramkörei – UART-adó és -vevő (+* ***4.3.4****. Nagysebességű UART-vevő)*
4.3.5. *A soros kommunikáció alapáramkörei – Nagysebességű SPI slave*

Ahogy azt az 1-16. ábra is szemlélteti, rendszertervünk szerint az üzemeltető mikrovezérlő három különböző interfészen keresztül léphet kapcsolatba a funkcióbővítő rendszerrel. A különféle belső részegységek, periféria-áramkörök, bővítőmodulok futásidejű konfigurálására és állapotuk lekérdezésére egy kissebességű, kétirányú UART-kapcsolatot hozunk létre, a nagyobb adatcsomagok átvitelét pedig egy – az UART-hoz képest lényegesen gyorsabb – adatfolyam jellegű alkalmazásokra optimalizált SPI linken keresztül tesszük lehetővé. Az SPI interfészlogikát úgy alakítjuk ki, hogy az képes legyen az üzemeltető mikrovezérlő és a bővítőmodulok között egy, az MMIO-buszrendszert „megkerülő", közvetlen, nagysebességű adatkapcsolat létrehozására. Az UART és SPI adatinterfészek mellé még egy egyszerű, egybites GPIO (**G**eneral-**P**urpose **I**/**O**) kimenetet is beépítünk, lehetővé téve a felhasználó számára, hogy a funkcióbővítő rendszert bekösse az üzemeltető mikrovezérlő megszakítási rendszerébe.

Ahogy a funkcióbővítő rendszer minden belső részegységének, úgy az e három interfészt működtető részáramköröknek is kompatibilisnek kell lenniük a belső processzoros rendszer kommunikációs hálózatával, az MMIO-buszrendszerrel. Ebben az alfejezetben tehát megtervezzük az MMIO UART és GPIO periféria-áramköröket, valamint a némileg egzotikusabb funkciójú, az SPI-interfésznek a bővítőmodulok közötti megosztott használatát lehetővé tevő nyolccsatornás, vonalkapcsolt útválasztót, amelynek futásidejű konfigurálására természetesen szintén egy MMIO slave interfészt alakítunk ki.

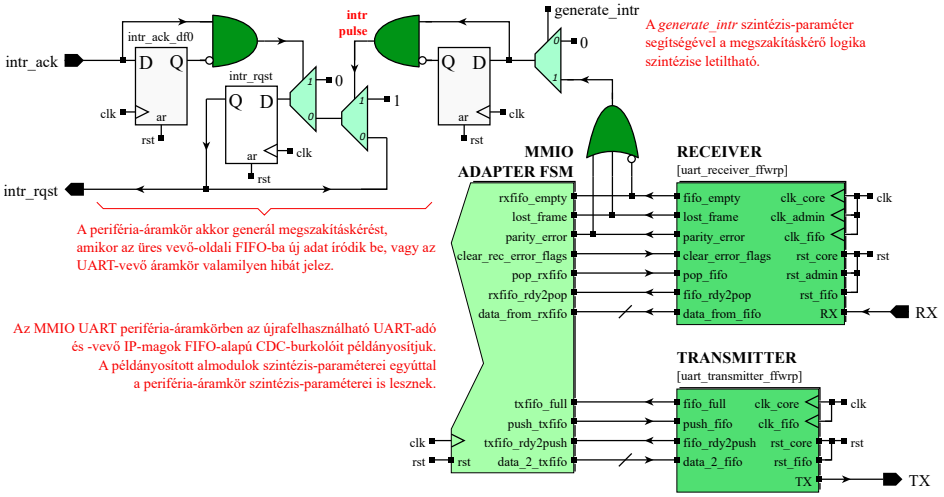
2.8.1. UART-vezérlő

Az UART szabványnak megfelelő soros-párhuzamos és párhuzamos-soros átalakítás a processzoros rendszertől és az MMIO-buszrendszertől függetlenül értelmezhető részfunkciók, amelyek megvalósítására célszerű újrafelhasználható IP-mago(ka)t készíteni. Ilyen újrafelhasználható IP-magok részletes RTL-terveit tárgyalja a 4.3.1. alfejezet. Az *uart_transmitter* és *uart_receiver* modulok a széles körű felhasználhatóság – és természetesen az oly nagyra értékelt egységesség – érdekében szinkron kézfogásos interfésszel vannak ellátva. Mindkét IP-mag példányosítható közvetlenül vagy a hozzájuk tartozó CDC-burkolómodulokon keresztül is.

Az egyszerű szinkron kézfogásos burkolómodulok a 2-2. ábrán látható általános vektorszinkronizáló fokozatokat tartalmazzák, így téve lehetővé az UART-adó és -vevő magok és beágyazó környezetük órajelének függetlenítését. Mivel az MMIO UART periféria-áramkörben a „felhasználó" tulajdonképpen az MMIO-illesztőlogika lesz, amiről – az MMIO-buszrendszer sajátosságaiból adódóan – eleve feltételezzük, hogy a beágyazó környezetéhez képest aszinkron órajelforrásról jár, ennek az egyszerű burkolómodulnak a használata nem indokolt. Úgy is fogalmazhatunk, hogy az MMIO periféria-áramkörön belül felesleges még egy külön órajeltartományt kialakítani, célszerűbb az újrafelhasználható IP-magokat és az MMIO-kompatibilitást biztosító egyéb

részáramköröket egyazon órajelről járatni.[73] Az adatfolyam jellegű alkalmazások támogatására az UART-adó és -vevő IP-magokhoz rendelkezésre állnak FIFO-alapú CDC-burkolómodulok is (lásd 4-42. és 4-45./4-46. ábra). Amellett, hogy lehetővé teszik a felhasználó és a belső IP-mag órajeleinek függetlenítését, ezek a burkolómodulok tartalmaznak belső FIFO-tárakat is az elküldendő és fogadott adatok ideiglenes tárolására, ily módon kompenzálva a soros adatátviteli csatorna és a beágyazó rendszer működése közötti feldolgozásisebesség-különbséget. E burkolómodulnak az órajelek függetlenítésére vonatkozó képességét a fentiek értelmében nem szeretnénk kihasználni, de az ideiglenes adattárolás képessége miatt érdemes megfontolni, hogy az MMIO UART periféria-áramkörben az UART-adó és -vevő IP-magok alapverzióit, vagy a FIFO-alapú burkolókat célszerű-e „MMIO-sítanunk".

Az üzemeltetői UART-interfész protokollját rendszertervünk ugyan nem rögzíti, de valószínűleg nem nyúlunk nagyon mellé, ha a két eszköz közötti kommunikációt több bájtból álló parancscsomagok és az azokra adott, ugyancsak több bájtból álló válaszcsomagok sorozataként képzeljük el. Bár a jellemzően 2-200 kb/s jelzési sebességgel működő UART egy több tíz MHz-en üzemelő *CPU* szempontjából aránylag lassúnak mondható, arról semmi információnk nincs, hogy egy-egy parancs végrehajtása vajon mennyi időt vesz igénybe. Annak érdekében, hogy az UART-on érkező bájtok feldolgozásához némi időtartalékot biztosítsunk a *CPU* számára, mindenképp érdemes az UART-vevő IP-magot annak FIFO-alapú burkolóján keresztül példányosítani. A válaszcsomagok visszaküldésével kapcsolatban hasonlóan érvelhetünk; a soros adatátvitelhez képest aránylag gyorsan működő *CPU*-nak egy-egy több bájtos válaszcsomag összeállítása után nem kell „kivárnia" az egyes bájtok elküldését, ha az UART-adót tartalmazó periféria-áramkörben van egy FIFO-tároló is, ami akár a teljes válaszcsomagot is képes lehet eltárolni. Miközben a periféria-áramkör a lassú



2-100. ábra. *Az UART-modul belső szerkezete*

[73] Lásd 2-74. ábra: a *PERIPH LOGIC* blokkban lévő részáramkörök ugyanarról az órajelről járnak, amiről az MMIO-adapter állapotgép és az egyéb illesztőlogikák. Kivételt jelent az az eset, amikor az MMIO periféria-áramkör kliens típusú interfészt ajánl ki a külvilág felé (lásd pl. megszakításvezérlő: 2-77. ábra vagy memória-arbiter: 2-83. és 2-84. ábra). Ekkor az interfészhez tartozó CDC-fokozatok kliens-oldala magának a kliensnek az órajeléről, kiszolgálóoldala viszont a periféria-áramkör magórajeléről jár.

UART-csatornára jellemző ütemben küldi a válaszcsomag bájtjait, a *CPU* foglalkozhat egyéb teendőivel, például a következő parancs feldolgozásával.

A fentiek értelmében az MMIO UART periféria-áramkör belsejében a 4.3.1. pontban bemutatott UART-adó és -vevő IP-magok FIFO-alapú burkolómoduljait példányosítjuk (lásd 2-100. ábra). A beépített almodulok szintézis-paramétereinek nem adunk konkrét értéket, ezeket az MMIO periféria-áramkör szintjén is paraméterként kezeljük.

Az MMIO-illesztőlogika szerepe a kapcsolási rajz alapján világosan látszik; elérhetővé kell tennie az UART-vevő kimeneti FIFO-jának *pop*-interfészét és az UART-adó bemeneti FIFO-jának *push*-interfészét az MMIO master, vagyis a *CPU* számára. A FIFO-k szolgáltatásainak MMIO írás- és olvasásműveletekre való leképezése sokféleképpen elképzelhető, egy lehetséges megoldást mutat be az UART periféria-áramkör címtérképét tartalmazó 2-10. táblázat.

Cím azonosítója és típusa		A címhez rendelt funkció leírása
address_rxtx_data	RW	Ezen a címen végrehajtva egy írás műveletet az MMIO mastertől érkező adat az UART-adó bemeneti FIFO-jába íródik be. Olvasás műveletet végrehajtva ezen a címen az MMIO masternek visszaküldött adat az UART-vevő kimeneti FIFO-jából kiolvasott bejegyzés lesz.
address_status	R	Ezen a címen érhetők el a periféria-áramkör belső állapotát leíró állapotjelzők. A masternak visszaadott 16-bites bejegyzés egyes mezőinek jelentése a következő: [0]: Magas-aktív paritáshiba-állapotjelző; az UART-vevő a legutóbb dekódolt adatkeretben paritáshibát érzékelt. [1]: Magas-aktív, az UART-vevő FIFO-jának telítettségéből adódó adatvesztést *(lost_frame)* jelző bit. Az UART-vevő a legutóbb dekódolt adatkeretet nem tudta eltárolni. [2]: Az UART-vevő kimeneti FIFO-jának *empty* jelzése. [3]: Az UART-adó bemeneti FIFO-jának *full* jelzése.
address_clear_error_flags	W	Az UART-vevő IP-mag állapotjelzői *(parity_error, lost_frame)* az ezen a címen végrehajtott írás művelettel törölhetők.

2-10. táblázat. *Az UART-modul címtérképe*

Az MMIO-adapter állapotgépnek az íráskérések kezeléséért felelős vezérlési állapotait a 2-101. ábra szemlélteti. Figyeljük meg, hogy az *address_rxtx_data* címen végrehajtott írás műveletkor az állapotgép az UART-adó bemeneti FIFO-jának *full* jelzését egyáltalán nem vizsgálja. Elképzelhető tehát az az eset, hogy a *CPU*-n futó program úgy hajt végre *store* utasítást ezen a címen, hogy az UART periféria-áramkör nem képes a kapott adatot eltárolni, ami nyilván funkcionális hibát jelent. A *full* jelzés persze vizsgálható volna minden írásművelet végrehajtásakor – pl. a 2-101. ábra 18. sorában lévő feltételvizsgálat kiegészítésével –, de így arról is gondoskodnunk kellene, hogy a sikertelen *push* műveletekről a felhasználó értesülhessen. Ez újabb állapotjelző bevezetését és/vagy a megszakításkéréseket generáló logika kibővítését igényelné, ami növelné a periféria-áramkör összetettségét és erőforrás-felhasználását is. Ezek a többlet erőforrások megspórolhatók a *CPU* programját készítő felhasználóval való „jóhiszemű együttműködéssel”; az UART-adó bemeneti FIFO-jának *full* jelzése bármikor lekérdezhető az *address_status* címen végrehajtott MMIO-olvasás segítségével, vagyis a periféria-áramkör dokumentációjában rögzíthetjük, hogy a felhasználónak kötelessége az *address_rxtx_data* cím írása előtt meggyőződni arról, hogy a küldendő adatot van hova beírni.

Hasonlóképpen járhatunk el az UART-vevő kimeneti FIFO-jának olvasásakor is (lásd 2-102. ábra); az *address_rxtx_data* cím olvasása az UART-vevő kimeneti FIFO-jának interfészén egy *pop*

VHDL

```vhdl
when idle => s_ready_2_cpu <= '1';

             -- Handle MMIO write requests...
             if ( write_strobe_from_cpu_df1 = '1' and cs = '1' ) then
               s_ready_2_cpu <= '0';
               
               if ( unsigned(address_from_cpu) = address_rxtx_data ) then state <= push_txfifo_1;
               elsif ( unsigned(address_from_cpu) = address_clear_error_flags ) then
                 clear_rec_error_flags <= '1'; state <= wait_for_deassert_strobes;
               else state <= wait_for_deassert_strobes;
               end if;
               
             end if;

             -- Handle MMIO read requests...
             if ( read_strobe_from_cpu_df1 = '1' and cs = '1') then s_ready_2_cpu <= '0'; [...] end if;

when push_txfifo_1 => if ( txfifo_rdy2push = '1' ) then
                        data_2_txfifo <= data_from_cpu(frame_size-1 downto 0);
                        push_txfifo <= '1';
                        state <= wait_for_deassert_strobes;
                      end if;

[...]

when wait_for_deassert_strobes => push_txfifo <= '0'; clear_rec_error_flags <= '0';
                                  if (
                                    read_strobe_from_cpu_df1 = '0' and
                                    write_strobe_from_cpu_df1 = '0'
                                  ) then state <= idle;
                                  end if;
```

2-101. ábra. *Az UART-modul MMIO-adapter állapotgépének vezérlési állapotai (1)*

VHDL

```vhdl
when idle => s_ready_2_cpu <= '1';

             -- Handle MMIO write requests...
             if ( write_strobe_from_cpu_df1 = '1' and cs = '1' ) then s_ready_2_cpu <= '0'; [...] end if;

             -- Handle MMIO read requests...
             if ( read_strobe_from_cpu_df1 = '1' and cs = '1') then
               s_ready_2_cpu <= '0';
               
               if ( unsigned(address_from_cpu) = address_rxtx_data ) then state <= pop_rxfifo_1;
               elsif ( unsigned(address_from_cpu) = address_status ) then
                 s_data_2_cpu(0) <= parity_error; s_data_2_cpu(1) <= lost_frame;
                 s_data_2_cpu(2) <= rxfifo_empty; s_data_2_cpu(3) <= txfifo_full;
                 s_data_2_cpu(15 downto 4) <= (others => '0');
                 state <= wait_for_deassert_strobes;
               else state <= wait_for_deassert_strobes;
               end if;
               
             end if;

[...]

when pop_rxfifo_1 => pop_rxfifo <= '1'; state <= pop_rxfifo_2;

when pop_rxfifo_2 => pop_rxfifo <= '0'; state <= pop_rxfifo_3;

when pop_rxfifo_3 => if ( rxfifo_rdy2pop = '1' ) then
                       s_data_2_cpu(frame_size-1 downto 0) <= data_from_rxfifo;
                       s_data_2_cpu(15 downto frame_size) <= (others => '0');
                       state <= wait_for_deassert_strobes;
                     end if;

when wait_for_deassert_strobes => push_txfifo <= '0'; clear_rec_error_flags <= '0';
                                  if (
                                    read_strobe_from_cpu_df1 = '0' and
                                    write_strobe_from_cpu_df1 = '0'
                                  ) then state <= idle;
                                  end if;
```

2-102. ábra. *Az UART-modul MMIO-adapter állapotgépének vezérlési állapotai (2)*

műveletet idéz elő anélkül, hogy az *empty* jelzést bármi módon figyelembe vennénk, vagyis a periféria-áramkör feltételezi, hogy az MMIO-olvasásra csak úgy kerülhetett sor, hogy előtte a felhasználó az állapotjelzők lekérdezésével már meggyőződött arról, hogy van mit kiolvasni a FIFO-ból.

Ahogy az a 2-100. ábrán ugyancsak látható, az MMIO UART periféria-áramkör megszakításkérést generál, ha az UART-vevő kimeneti FIFO-jának *empty* jelzésén egy lefutóél áll elő – vagyis, amikor az üres FIFO-ba egy új adat íródik be –, illetve amikor a vevő valamelyik hibajelzése aktiválódik – paritáshibás adatkeret fogadása vagy a FIFO telítettségéből adódó adatvesztés esetén. A megszakításkérések generálásához vezető események mindegyikére igaz, hogy bekövetkezésükről a *CPU* az *address_status* címen végrehajtott periodikus olvasással is értesülhet. A *CPU*-n futó program készítőjének a döntése, hogy ezeket az eseményeket az állapotjelzők periodikus lekérdezésével (polling) detektálja, vagy a megszakítási rendszerre támaszkodik. A megszakításkérések generálását emiatt letilthatóvá tesszük a *generate_intr* szintézis-paraméter segítségével. Ez a megoldás egyúttal erőforrás-optimalizációt is jelent, mivel e paraméter alacsony értéke esetén a periféria-áramkör megszakításkérő logikáját a szintézis-szoftverek – a „constant propagation” nevű logikai optimalizációs eljárás nyomán (lásd még DRMSz: 273. oldal) – be sem építik a HDL-leírásból előállított kapuszintű netlistába.

2.8.2. GPIO-vezérlő

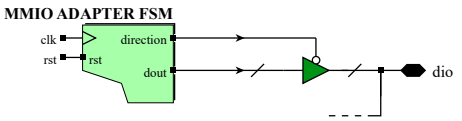
Rendszertervünk szerint az üzemeltető mikrovezérlő megszakítási rendszeréhez való kapcsolódás mindössze annyit követel meg a funkcióbővítő rendszer GPIO-vezérlőjétől, hogy képes legyen egyetlen kimenet értékének beállítására. Tekintve, hogy tervezési alapelveink között szerepel az újrafelhasználhatóságra való törekvés is, az MMIO GPIO-vezérlőt természetesen ennél sokkal általánosabb követelmények szerint készítjük el. Egyrészt nemcsak egybites jelek, hanem több-bites vektorok kezelésére is alkalmassá tesszük, másrészt gondoskodunk arról, hogy a felhasználó akár futásidőben is módosíthassa az adott port irányát.

Ahhoz, hogy egy több-bites port irányát és az adott esetben kimenetként használt kivezetések értékét MMIO-interfészen keresztül tegyük beállíthatóvá, egy aránylag egyszerű illesztőlogikára van szükségünk. (lásd 2-103. ábra).

Verilog

```
1  // TRI-STATE OUTPUT DRIVER
2  assign dio = (direction == 0) ? dout : 'Z;
```

Áramköri struktúra



2-103. ábra. *A GPIO-modul belső szerkezete (1)*

Ebben a megoldásban a háromállapotú meghajtó engedélyező és adatbemenetére kapcsolt *direction* és *dout* regisztereket egy-egy külön MMIO-címen tesszük elérhetővé. A kétirányú *dio* busz kimenetként való konfigurálása és értékének beállítása tehát két MMIO-írással – vagyis két *store* utasítás végrehajtásával – elvégezhető.

Ugyanennek a kétirányú busznak a bemenetként való használata meglepően sok kérdést vet fel. Ebben a helyzetben tehát adott egy több bites vektor *(dio)*, amelynek értékét külső tényezők (pl. egy másik eszköz: mikrovezérlő, FPGA) határozzák meg. A „külső tényezők" kifejezés láttán az első dolog, ami eszünkbe juthat, hogy a buszon végbemenő változások bármikor bekövetkezhetnek, vagyis a busz mintavételezésénél körültekintően kell eljárnunk, órajel-szinkronizáló fokozatot kell beépítenünk. Több bites busz lévén a *dio* bemenet megfelelő mintavételezéséhez valamilyen vektorszinkronizáló fokozatra van szükségünk. Lássuk a lehetőségeket:

- Az általános vektorszinkronizáló fokozatok – mint amilyeneket a szinkron kézfogásos interfésszel ellátott, belső almodulok között használunk (lásd 2-2. és 2-3. ábra) – tulajdonképpen fizikailag nem a több-bites buszt szinkronizálják, hanem hozzárendelnek a buszhoz egy egybites jelzővonalat *(request)*, amit egyszerű, egybites impulzus-szinkronizáló fokozattal visznek át az adó oldalról a vevő oldalra. Az adó e jelzővonal segítségével tájékoztatja a vevőt arról, hogy mikor garantálja a busz értékének változatlanságát, így a vevő pontosan tudja, hogy mikor mintavételezheti azt. Az általános GPIO-vezérlőnél ilyen jelzővonal nincs, tehát az általános vektorszinkronizáló fokozatok nem használhatók.
- Több-bites buszok jelzővonal nélküli átvitelére alkalmas a Gray-szinkronizáló fokozat is (lásd DRMSz: 6-32. ábra), de csak bizonyos korlátozásokkal. Ez a módszer ui. feltételezi, hogy a szinkronizálandó busz egy számláló, vagyis hogy bármely állapotváltozás esetén a régi és az új bináris kombinációt nemnegatív vagy kettes-komplemens kódolású egész számként értelmezve az ábrázolt értékek között a különbség mindig egy. Ez a feltétel az általános GPIO-vezérlő esetén nem teljesül, úgyhogy sajnos ezt a szinkronizálási módszert is kénytelenek vagyunk elvetni.
- A több-bites buszok órajeltartományok közötti átvitelének legáltalánosabb – és egyben legköltségesebb – módszere az aszinkron beíró és kiolvasó órajelekkel üzemeltetett FIFO (lásd 4-26. ábra). Ez a megoldás több okból sem felel meg nekünk. A FIFO önmagában nem képes eldönteni, hogy a bemeneti busz értékét mikor kellene beírnia a belső tárolójába, mindenképpen szükség lenne tehát némi kiegészítő logikára, ami figyelné a buszon végbemenő változásokat, és a *push* műveletet e változások után kezdeményezné. A megoldás így sem volna teljesen korrekt, hiszen a GPIO-vezérlő esetén nem a busz viselkedésének „történelmére", hanem annak mindenkori állapotára vagyunk kíváncsiak.

A jól bevált módszereink egyike sem alkalmas tehát az általános GPIO-vezérlő problémájának kezelésére, ezért kénytelenek leszünk egy új, egyedi mintavevő áramkört konstruálni. Első lépésként oldjuk meg a busz mintavételezését úgy, hogy a metastabilitás legalább fizikailag ne okozzon problémát, és állítsunk elő egy, a busz állapotának megváltozására utaló belső jelzést. Ezt a feladatot a 2-104. ábrán látható áramköri szerkezet látja el.

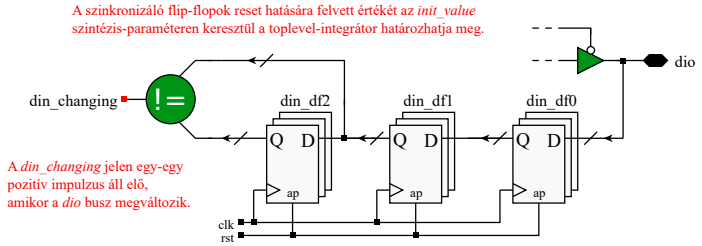
A bemeneti buszt tehát egyszerű, bitenkénti double-flop szinkronizálással mintavételezzük, ami biztosítja, hogy metastabil állapot csak a *din_df0* vektor flip-flopjaiban állhat elő.[74] Vizsgáljuk meg e kapcsolás viselkedését, amikor a *dio* bemenet tetszőleges időpontban megváltozik. A 2-104. ábra időzítési diagramja a metastabilitás nemdeterminisztikus jellegéből adódóan eltérő, lehetséges

[74] Elvileg lehetséges, hogy a *din_df0* regisztert alkotó flip-flopokban a metastabilitás egy órajelciklus időtartamnál tovább fennáll, ami a *din_df1* regisztert alkotó flip-flopokban is metastabilitást idézhet elő, de ennek a valószínűsége olyan csekély, hogy rendszerint nullának tekintjük.

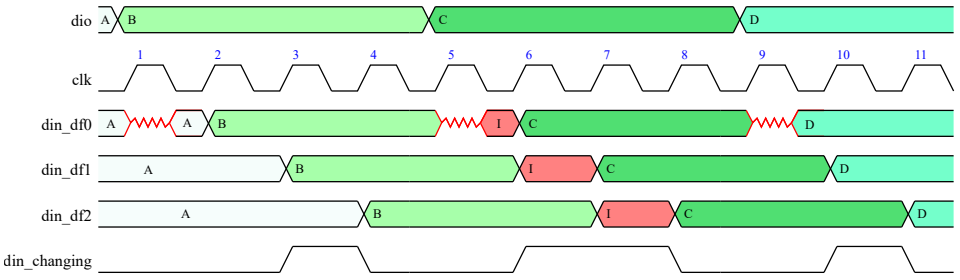
Verilog

```
// INPUT HANDLING LOGIC
parameter init_value;

always @ ( posedge clk, posedge rst ) begin
  if ( rst == 1 ) begin
    din_df0 <= init_value; din_df1 <= init_value; din_df2 <= init_value;
  end else begin
    din_df0 <= dio; din_df1 <= din_df0; din_df2 <= din_df1;
  end
end

assign din_changing = (din_df1 != din_df2) ? 1 : 0;
```

Áramköri struktúra



Hullámforma



2-104. ábra. *A GPIO-modul belső szerkezete (2)*

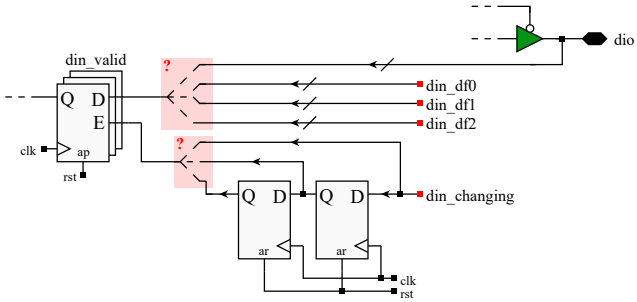
forgatókönyveket szemlélteti. A *dio* bemeneten bekövetkező A → B állapotváltozás hatására a *din_df0* regiszternek bizonyos flip-flopjai metastabil állapotba kerülhetnek. E metastabil állapotból kilépve a flip-flopok bármilyen értéket felvehetnek; az ábra azt az esetet mutatja be, amikor a *din_df0* regiszter még az órajel második felfutóéle előtt véletlenül éppen újra az A értéken stabilizálódik, majd az órajel harmadik felfutóélénél töltődik bele a B érték – ekkor a *dio* bemenet már stabil, tehát metastabilitás itt már nem alakulhat ki újra. A *din_df1* és *din_df2* regiszterek rendre az órajel harmadik és negyedik felfutóéle hatására veszik fel a B értéket, aminek következtében a *din_changing* jelen az órajel harmadik és negyedik felfutóéle között áll elő egy pozitív impulzus, jelezve a bemenet értékének megváltozását.

Az ábrán látható második eset azt szemlélteti, amikor a *dio* bemeneten bekövetkező B → C változás ugyancsak metastabilitást okoz a *din_df0* regiszter flip-flopjaiban – vagy legalábbis némelyikben –, de ezúttal a stabilizálódás egy B-től és C-től is különböző I (inkonzisztens) értéken következik be, mivel a flip-flopok egy része a változás előtti, másik része pedig a változás utáni értékre áll be. A *din_df0* regiszter tehát egy tranziens erejéig olyan értéket vesz fel, ami a *dio* buszon sosem fordult elő. Az órajel hatodik felfutóéle hatására a *din_df0* regiszterbe betöltődik ugyan C,

de az inkonzisztens érték egy-egy órajelciklus ideig a *din_df1* és *din_df2* regiszterekben is megjelenik. Ezek az inkonzisztens értékek egy két órajelciklus időtartamú magas impulzust eredményeznek a *din_df1* és *din_df2* regiszterek értékének különbözőségére érzékeny komparátor kimenetén. Fontos hangsúlyozni, hogy az inkonzisztens érték nem metastabil állapotot jelent, hanem csupán azt, hogy ez az érték a *dio* bemeneten valójában sosem állt elő, az kifejezetten a metastabilitás nemdeterminisztikus természetének a melléktermék.

A harmadik esetben, a *dio* bemenet C → D változásánál a részben metastabil állapotba kerülő flip-flopok alkotta *din_df0* regiszter még a következő órajel-felfutóél előtt felveszi D értékét, ami lényegében annak felel meg, mintha egyáltalán nem is következett volna be az időzítési feltételek megsértése miatti metastabilitás egyik flip-flopban sem. Ekkor a *dio* bemenet megváltozását jelző komparátor kimenetén – az első esethez képest egy órajelciklussal korábban – már a bemeneti változást követő első és második órajel-felfutóélek között előáll egy pozitív impulzus.

A mintavevő áramkörrel alapvető célunk az, hogy legyen egy olyan regiszterünk *(din_valid)*, ami a *dio* bemenet változásakor – ha némi késleltetéssel is, de – felveszi annak új értékét, miközben garantálható, hogy inkonzisztens, tehát a *dio* bemeneten soha elő nem forduló érték nem kerül bele még ideiglenesen sem. A bemenet megváltozását jelző *din_changing* jelet vagy annak késleltetett értékeit – e mintatároló regiszter órajel-engedélyező bemenetére kapcsolva – felhasználhatjuk a mintavétel pillanatának meghatározására (lásd 2-105. ábra).



2-105. ábra. *A GPIO-modul belső szerkezete (3)*

A kérdés csak az, hogy szükséges-e a késleltetés beiktatása, és ha igen, akkor pontosan milyen mértékű, illetve hogy a *dio* busz és a *din_df0/1/2* regiszterek közül melyiket érdemes mintavételezni.

- A *dio* busz a 2-104. ábrán vázolt forgatókönyvekben nem változik „túl gyakran", látható, hogy bármely két változás között legalább három órajelciklus telik el. A bemenet viselkedésére vonatkozó ebbéli feltételezésünk később részletezett okok miatt nem feltétlenül helytálló. Ha viszont elfogadjuk, hogy a *dio* bemeneten bármikor, bármilyen gyakorisággal bekövetkezhet változás, függetlenül attól, hogy a *din_changing* jel mikor aktív, akkor annak közvetlen rákapcsolása a *din_valid* regiszter bemenetére semmiképp sem célszerű, hiszen így a *din_valid* regiszter flip-flopjai is potenciálisan metastabil állapotba kerülhetnének, ami azt jelentené, hogy a metastabilitást még lokalizálnunk sem sikerült.
- A *din_df0* regiszter kimenetét ugyancsak nem célszerű a *din_valid* regiszter bemenetére kapcsolnunk, mert ezzel megnövelnénk az „üzemszerűen" metastabil állapotba kerülő

flip-flopok terhelését, megnövelve a stabilizálódáshoz szükséges időt is, végső soron tehát csökkentve a szinkronizáló fokozat megbízhatóságát.

- A *din_df1* regiszter betöltése a *din_valid* regiszterbe működőképes megoldás, feltéve, hogy e betöltés pillanatát körültekintően választjuk meg. A *din_changing* jel közvetlen felhasználása a *din_valid* regiszter órajel-engedélyező bemenetének meghajtására a 2-104. ábrán látható esetek közül az elsőben és a harmadikban problémamentes, a *din_valid* regiszter az órajel negyedik és tizenegyedik felfutóélénél kapja meg a *din_df1* regiszter értékét, ami mindkét esetben a *dio* bemenet megváltozása utáni stabil érték. Ha azonban a B → C átmenet által szemléltetett forgatókönyv teljesül, akkor a *din_valid* regiszter órajel-engedélyező bemenete a hetedik és a nyolcadik órajel-felfutóélnél is magas szintű, aminek eredményeként a *din_valid* regiszter ideiglenesen az inkonzisztens értéket is felveszi. Ez a helyzet elkerülhető, ha a *din_valid* regiszter órajel-engedélyező bemenetére nem a *din_changing* jelet, hanem annak egy órajelciklussal megkésleltetett változatát kötjük be.
- A *din_valid* regiszter adatbemenetét a *din_df2* regiszter kimenetével is hajthatjuk, ekkor azonban értelemszerűen a *din_changing* jelre is egy újabb késleltető flip-flopot kell beiktatnunk, ami szükségtelenül növelné a bemenet megváltozása és a mintavétel közt eltelt időt.

A fentiek értelmében tehát a 2-106. ábrán látható áramköri struktúrát érdemes kialakítani az aszinkron módon változó *dio* bemenet mintavételezésére. A hullámformán látható, hogy a *din_valid* regiszter úgy követi a *dio* busz változásait, hogy közben nem vesz fel inkonzisztens, a *dio* buszon sosem tapasztalható értéket. Ez azt jelenti, hogy az MMIO-adapter állapotgép bármelyik időpillanatban mintavételezheti a *din_valid* regisztert, az abban tárolt érték mindig tekinthető a *dio* busz mindenkori értékének.[75] A GPIO-vezérlő címtérképe a 2-103. és a 2-106. ábrák alapján már könnyen konstruálható (lásd 2-11. táblázat).

Cím azonosítója és típusa		A címhez rendelt funkció leírása
address_direction	W	A *dio* busz irányát meghatározó regiszter címe. Ha a legkisebb helyiértékű bit értéke 0, akkor a *dio* busz kimenet, ellenkező esetben bemenet.
address_data	RW	A *dio* busz mindenkori értékét reprezentáló adatregiszter címe.

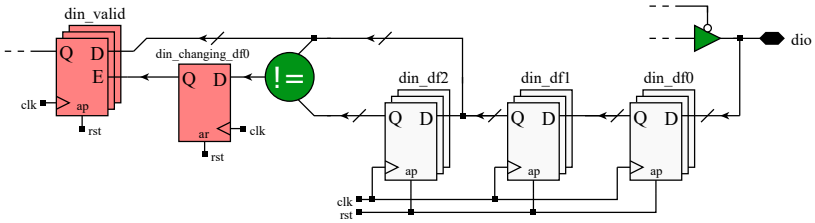
2-11. táblázat. *A GPIO modul címtérképe*

[75] Ahogy korábban utaltunk rá, a 2-106. ábrán látható kapcsolás csak akkor képes biztosítani a *din_valid* vektor konzisztenciáját, ha a *dio* busz változásai nem követik egymást túl gyakran. Ez lényegében az egybites double-flop szinkronizáló fokozatok azon korlátozottságának a következménye, mely szerint csak statikus jelek átvitelére alkalmasak. Ezt a korlátot felfoghatjuk a *dio* buszt meghajtó külső eszközre vonatkozó előírásnak is. Ha a *dio* busz minden bitje azonos forrásból származik (pl. egy másik FPGA vagy mikrokontroller párhuzamos kimenete), ahol az egyes bitek megváltozása tekinthető egyidejűnek (a megváltozásuk között eltelő idő nagyságrendekkel kisebb az órajel periódusidejénél), akkor a teljes busz változásainak gyakoriságára vonatkozó előírás nem jelent problémát, rendszertechnikailag kezelhető. Ráadásul ennek az előírásnak a megsértése sem vezet katasztrofális hibához, hiszen a szinkronizáló fokozat a metastabilitást így is képes izolálni. Ha a *dio* busz egyes bitjei külön hardverelemektől (pl. egy sor kapcsoló) származnak, akkor a teljes busz egészének egymást követő változásai bármikor bekövetkezhetnek. Ilyenkor a fenti szinkronizáló fokozat a busz konzisztenciáját nem képes biztosítani, ez azonban nem jelenthet gondot, hiszen a különböző forrásokból származó bitek halmaza esetén eleve nincs értelme konzisztenciáról beszélni. Ilyenkor a szinkronizáló fokozat feladata csupán a metastabilitás izolálása, amit ebben az esetben is teljesít.

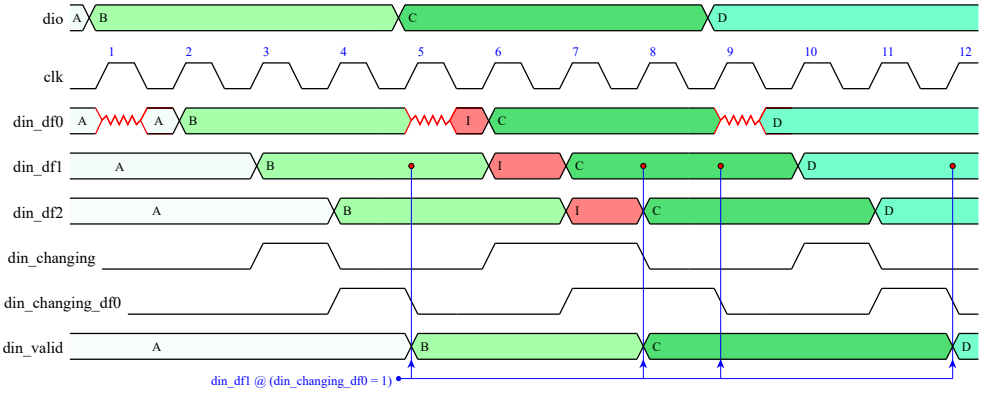
Verilog

```
1  // INPUT HANDLING LOGIC
2  always @ ( posedge clk, posedge rst ) begin
3    if ( rst == 1 ) begin
4      din_df0 <= init_value; din_df1 <= init_value; din_df2 <= init_value; din_changing_df0 <= 0;
5    end else begin
6
7      din_df0 <= dio; din_df1 <= din_df0; din_df2 <= din_df1;
8
9      din_changing_df0 <= din_changing;
10
11     if ( din_changing_df0 == 1 ) din_valid <= din_df1;
12
13   end
14 end
15
16 assign din_changing = (din_df1 != din_df2) ? 1 : 0;
```

Áramköri struktúra



Hullámforma



2-106. ábra. *A GPIO-modul belső szerkezete (4)*

A *dio* busz irányát meghatározó, csak-írható cím mögött egy, az MMIO-adapter állapotgép kimeneti buffereként megvalósított, egyszerű, egybites regiszter rejlik. Az e címen végrehajtott írás műveletben a *CPU*-tól kapott 16-bites adat felső 15 bitjét tehát figyelmen kívül hagyjuk, a *dio* busz irányát a legkisebb helyiértékű bit értéke határozza meg. Az *address_data* címen végrehajtott írásművelet az ugyancsak az MMIO-adapter állapotgép kimeneti bufferként megvalósuló, *dw* bit széles *dout* regiszterbe írja be a *CPU*-tól kapott adat alsó *dw* darab bitjét, miközben a felső 16-*dw* darab bitet figyelmen kívül hagyja. A 2-107. ábra a GPIO-vezérlő címtérképét a fentiek értelmében megvalósító MMIO-adapter állapotgép vezérlési állapotait szemlélteti.

Az eddig tárgyalt áramköri részletek lehetővé teszik a *CPU* számára, hogy a megfelelő címeken végrehajtott MMIO írás- és olvasásműveletekkel – vagyis *store* és *load* utasítások végrehajtásával – beállítsa a *dio* busz irányát, valamint hogy írja vagy olvassa annak értékét. Tehát a GPIO-vezérlő

Verilog

```
idle: begin
  s_ready_2_cpu <= 1;

  // Handle MMIO write requests...
  if ( write_strobe_from_cpu_df1 == 1 && cs == 1 ) begin
    s_ready_2_cpu <= 0;

    if ( address_from_cpu == address_direction ) begin
      direction <= data_from_cpu[0]; state <= wait_for_deassert_strobes;
    end else if ( address_from_cpu == address_data ) begin
      dout <= data_from_cpu[dw-0:0]; state <= wait_for_deassert_strobes;
    end else state <= wait_for_deassert_strobes;

  end

  // Handle MMIO read requests...
  if ( read_strobe_from_cpu_df1 == 1 && cs == 1 ) begin
    s_ready_2_cpu <= 0;

    if ( address_from_cpu == address_data ) begin
      s_data_2_cpu <= 0; s_data_2_cpu[dw-1:0] <= din_valid; state <= wait_for_deassert_strobes;
    end else state <= wait_for_deassert_strobes;

  end

end

wait_for_deassert_strobes: if (
                             read_strobe_from_cpu_df1 == 0 &&
                             write_strobe_from_cpu_df1 == 0
                           ) state <= idle;
```

2-107. ábra. *A GPIO-modul MMIO-adapter állapotgépének vezérlési állapotai*

az alapfunkcióját így már el tudja látni. További kérdéseket vet fel azonban a *CPU* megszakítási rendszeréhez való kapcsolódás lehetősége. Ha a GPIO-vezérlőben a *dio* busz irányát kimenetként határozzuk meg, akkor a megszakításkérésnek nincs értelme, de ha a buszt bemenetként használjuk, akkor annak változásai sokféleképpen képezhetők le a *CPU* felé küldött megszakításkérésekre. A 2-108. ábra egy olyan megszakításkérő logikát mutat be, ami szintézis-paraméterek segítségével az adott alkalmazás igényei szerint testreszabható.

A megszakításkéréseket generáló, aránylag összetett és sok erőforrást felemésztő logika – az MMIO UART periféria-áramkörnél látottakhoz hasonlóan – a *generate_intr* szintézis-paraméter segítségével letiltható, így – ha az adott alkalmazásban nincs rá szükség – nem kerül bele a szintetizált netlistába. A megszakításkérő logika további szintézis-paraméterek segítségével konfigurálható. Alapvetően kétféle eseménytípus bekövetkezését tudjuk vele detektálni; az *intr_gen_type* paramétert magas szintre állítva a *dio* busz bármely változása megszakításkérést idéz elő, míg ugyanennek a paraméternek az alacsony szintje mellett a *dio* busznak csak egy kiválasztott vezetékét figyeljük, annak változásaikor generálunk megszakításkérést. A megfigyelt vezeték az *intr_gen_index* paraméterrel választható ki. Ez esetben a toplevel integrátornak lehetősége van még tovább szűkíteni a megszakításkéréshez vezető események sorát; az *intr_gen_edge_type* paraméter határozza meg, hogy a kiválasztott vezetéken végbemenő mely változások – felfutóél, lefutóél vagy mindkettő – váltsanak ki megszakításkérést.

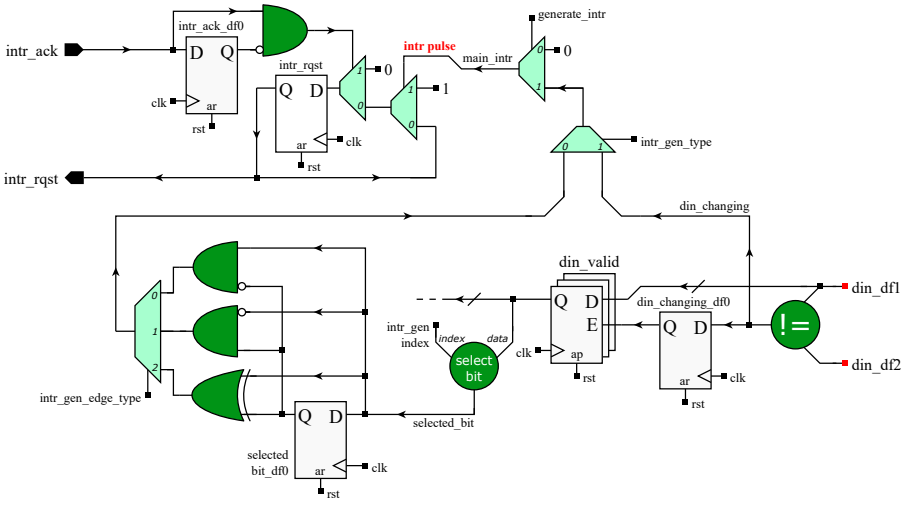
2.8.3. SPI-FIFO útválasztó

Funkcióbővítő rendszerünk lényege – ahogy arra elnevezése is utal –, hogy egyedi feladatkörű, adott alkalmazás igényei szerint kialakított céláramköröket tudunk vele hozzákapcsolni az üzemeltető

Verilog

```
parameter generate_intr;
parameter intr_gen_type;      // 0: edge of a selected bit, 1: change on the dio vector
parameter intr_gen_edge_type; // 0: rising, 1: falling, 2: both
parameter intr_gen_index;

[...]

always @ ( posedge clk, posedge rst ) begin
  if ( rst == 1 ) begin
    intr_ack_df0 <= 0; intr_rqst <= 0; selected_bit_df0 <= 0;
  end else begin

    intr_ack_df0 <= intr_ack;

    if ( intr_ack_rising == 1 ) intr_rqst <= 0;
    else if ( main_intr == 1 ) intr_rqst <= 1;

    selected_bit_df0 <= selected_bit;

  end
end

assign intr_ack_rising = intr_ack & ~intr_ack_df0;
assign selected_bit = din_valid[intr_gen_index];
assign selected_bit_rising = selected_bit & ~selected_bit_df0;
assign selected_bit_falling = ~selected_bit & selected_bit_df0;
assign selected_bit_changing = selected_bit ^ selected_bit_df0;

always @ (*) begin
  if ( generate_intr == 0 ) main_intr <= 0;
  else if ( intr_gen_type == 1 ) main_intr <= din_changing;
  else case ( intr_gen_edge_type )
        0:       main_intr <= selected_bit_rising;
        1:       main_intr <= selected_bit_falling;
        default: main_intr <= selected_bit_changing;
      endcase
end
```

Áramköri struktúra

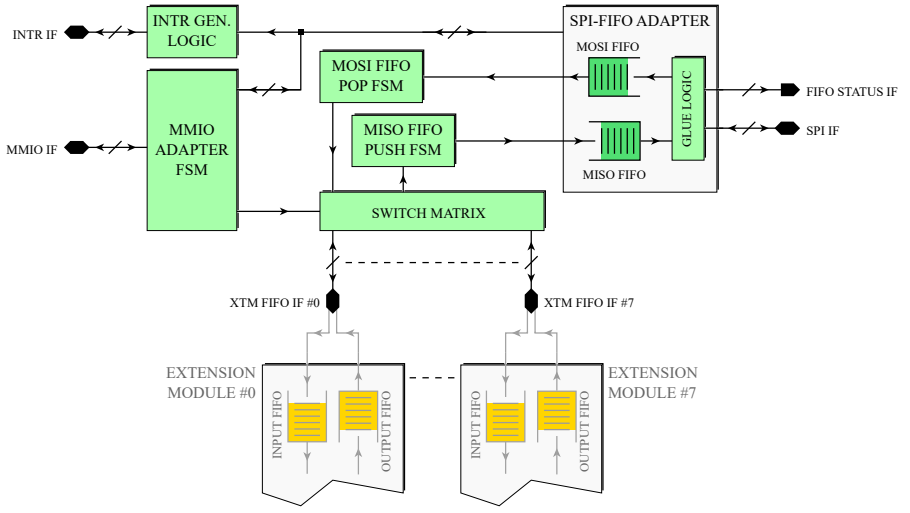


2-108. ábra. *A GPIO-modul megszakításkérő logikája*

rendszerhez, például egy korlátozott képességekkel bíró, egyszerű mikrovezérlőhöz. Az egyedi céláramkörök az MMIO-buszrendszerre kapcsolódnak, ami azt jelenti, hogy az összeköttetésük az üzemeltető rendszerrel tulajdonképpen már önmagában a 2.8.1. pontban tárgyalt üzemeltetői

UART-modul által megvalósul; az ezen az interfészen kapott adatokat a *CPU* továbbíthatja a bővítőmodulok felé és fordítva. Természetesen mindezt a *CPU* csak a közös MMIO-buszrendszeren végrehajtott írás- és olvasásműveletek sokaságával képes végrehajtani. Ha nagy mennyiségű adat mozgatására van szükség az üzemeltető rendszer és a bővítőmodulok között, akkor ez a megoldás több szempontból is problémás. Egyrészt az üzemeltetői UART egy lassú, jellemzően 2-200 kb/s adatátviteli sebességű interfész, ami az átviendő adatmennyiség növekedésével egyre zavaróbb szűk keresztmetszetet jelent, másrészt az átvitt adatoknak így az MMIO-buszrendszeren is keresztül kell jutniuk, amit azonban a rendszer összes periféria-áramköre közösen használ. A *CPU*-n futó program készítője megtehetné, hogy adatátviteli igény esetén egyáltalán nem foglalkozik a többi periféria-áramkörrel – akár még a megszakításkéréseket is figyelmen kívül hagyva –, így minimalizálva az átvitelhez szükséges időt, ez azonban semmiképp sem tekinthető egy általános megoldásnak, főleg, ha vannak a rendszerben időzítéskritikus feladatot ellátó periféria-áramkörök is.

Ahelyett, hogy a leendő felhasználó leleményességére bazíroznánk a rendszer e nyilvánvaló korlátozottságából adódó nehézségek megoldását, az üzemeltetői UART-interfészt és az MMIO-buszrendszert egy, az üzemeltető rendszer és a funkcióbővítő rendszer közé beépített SPI-interfésszel tehermentesítjük. Ennek az SPI-interfésznek a használatát egyszerre csak egy, időosztásos alapon legfeljebb nyolc bővítőmodul számára tesszük lehetővé a 2-109. ábrán szemléltetett felépítésű, MMIO-kompatibilis periféria-áramkör, az SPI-FIFO útválasztó segítségével.



2-109. ábra. *Az SPI-FIFO útválasztó modul egyszerűsített blokkvázlata a hozzákapcsolt bővítőmodulokkal*

E periféria-áramkör lelke egy olyan illesztőlogika *(SPI-FIFO ADAPTER)*, ami a soros SPI-tranzakciókat FIFO *push* és *pop* műveletekre „fordítja le". Ez a részáramkör az üzemeltető rendszer által a MOSI vonalon sorosan küldött adatbitek párhuzamosításával kapott bájtfolyamot egy FIFO-tárba *(MOSI FIFO)* írja be, miközben a visszaküldendő bájtfolyamot egy másik FIFO-tárból *(MISO FIFO)* olvassa ki, és e kiolvasott bájtok bitjeit a megfelelő sorrendben és ütemezéssel a MISO vonalra lépteti ki.

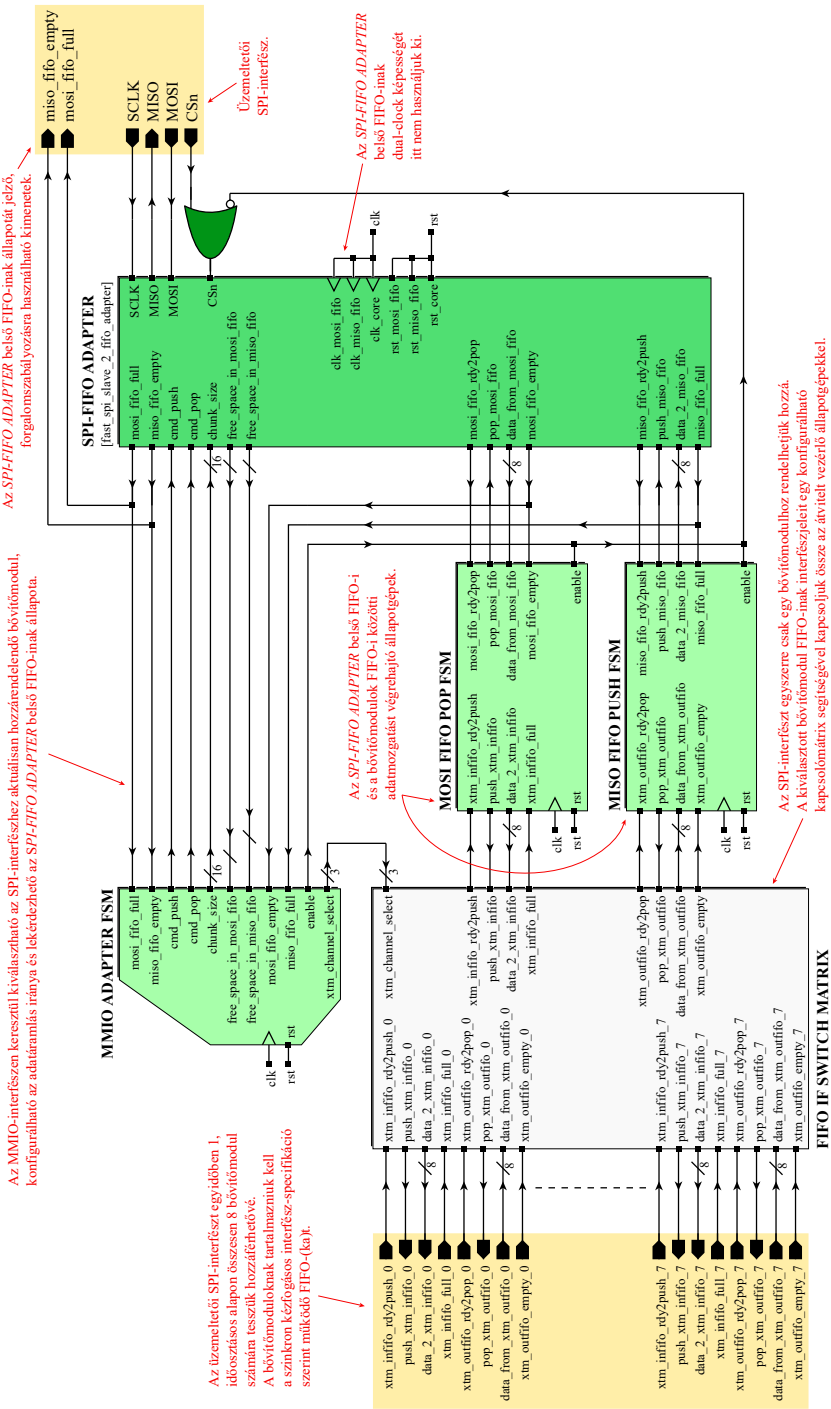
A bővítőmodulok csak akkor tudják az SPI-FIFO útválasztó szolgáltatásait igénybe venni, ha maguk is rendelkeznek a *MOSI* és *MISO FIFO*-khoz hasonló bemeneti és kimeneti FIFO-tárakkal. A bővítőmodulok FIFO-i és az *SPI-FIFO ADAPTER* FIFO-i közötti adatmozgatást állapotgépek

végzik *(MOSI FIFO POP FSM, MISO FIFO PUSH FSM)*, amelyeknek csak egy-egy darab, bővítőmodul-oldali FIFO interfészük van. Egy futásidőben konfigurálható kapcsolómátrix mindig csak az éppen kiválasztott bővítőmodullal teremt közvetlen összeköttetést, megvalósítva ezáltal az SPI-interfész bővítőmodulok közötti megosztott használatát.

Az SPI-FIFO útválasztó modul célja az, hogy a *CPU*-t megkerülő adatkapcsolatot valósítson meg a bővítőmodulok és az üzemeltető rendszer között, de ez nem jelenti azt, hogy a modul teljesen független lenne a funkcióbővítő rendszer belső kommunikációs hálózataitól. A *MOSI* és *MISO FIFO* állapotának diagnosztikai célú lekérdezése, valamint a kapcsolómátrix konfigurálása is MMIO-interfészen történik, illetve a belső állapot bizonyos – később részletezett – megváltozásai esetén az almodul megszakításkérések generálására is képes. A modul belső szerkezetének részleteit a 2-110. és a 2-115. ábrák szemléltetik.

Az *SPI-FIFO ADAPTER* modul feladatköre, mely szerint az üzemeltető rendszer által az SPI-interfészen küldött bájtokat egy FIFO-ba írja be, miközben a visszaküldendő bájtokat egy másik FIFO-ból olvassa ki és továbbítja az SPI-interfészre, kellően általános ahhoz, hogy egy újrafelhasználható IP-magot készítsünk hozzá. Ennek az IP-magnak a belső működését és felépítését a 4.3.5. pont tárgyalja. Ahhoz, hogy az SPI-FIFO útválasztóban felhasználjuk az *SPI-FIFO ADAPTER*-t, nem szükséges minden belső sajátosságát részleteiben ismernünk – pontosan ez az újrafelhasználható IP-alapú tervezés lényege –, de fontos, hogy alapvető szolgáltatásaival és felhasználói interfészeinek működésével tisztában legyünk, ezért ezeket röviden itt is összefoglaljuk:

- A modul a master és a slave közötti adatátvitelt ún. „SPI-tranzakciók" sorozataként definiálja. Egy ilyen tranzakció során a master és a slave között N × 8 bájtból álló csomagok (chunk) kicserélésére kerül sor; a master egység elküld egy csomagot a slave egységnek, ezzel egyidőben a slave is visszaküld egy csomagot a masternak.
- A csomagméretet meghatározó *chunk_size* vezérlőbemenet értéke futásidőben, minden tranzakció előtt módosítható, tranzakció közben viszont nem változhat.
- A felhasználónak lehetősége van külön-külön engedélyezni/tiltani a master → slave és slave → master irányú adatáramlást egy-egy magas-aktív vezérlőbemenet *(cmd_push, cmd_pop)* megfelelő beállításával. E vezérlőbemenetek a *chunk_size* bemenethez hasonlóan kezelendők; minden tranzakció előtt beállíthatók, de tranzakció közben nem változhatnak.
- A modul a két adatfolyam bájtjait egy-egy belső FIFO-ban tárolja. E FIFO-k – ahogy azt a 4-80. és 4-86. ábrák is szemléltetik – a 4.2.5. pontban bemutatott, újrafelhasználható *dualclock_fifo_hswrp* IP-mag példányai. Az *SPI-FIFO ADAPTER* modul integrálásánál tehát ennek a FIFO IP-magnak a kliensinterfészével kompatibilis logikát kell készítenünk.
- A modul belső FIFO-i dual-clock jellegűek, vagyis a felhasználó és a belső áramkörök közötti órajel-szinkronizálás feladatát is ellátják. Esetünkben ez a „felhasználó" – az almodul, amin belül az *SPI-FIFO ADAPTER*-t példányosítjuk – egy MMIO periféria-áramkör, amin belül nem alakítunk ki további aszinkron órajeltartományokat. Ennek oka csupán az egyszerűségre való törekvés; maga az MMIO-buszrendszer eleve lehetővé teszi, hogy a periféria-áramköröket egymástól független órajeltartományokban üzemeltessük, azokon belül további órajeltartományokat létrehozni indokolatlan lenne. Az *SPI-FIFO ADAPTER*-nek az órajel-szinkronizáló képességét tehát nem használjuk ki; a 2-110. ábrán is látható, hogy a *MOSI/MISO FIFO* kliens-oldali órajelbemeneteit a modul magórajelével közösítettük.



2-110. ábra. *Az SPI-FIFO útválasztó szerkezete (1)*

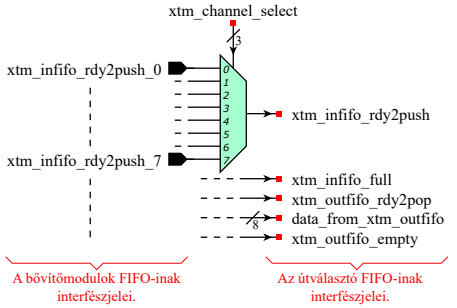
- A modul a felhasználói interfészén elérhetővé teszi a belső FIFO-tárolóinak telítettségére vonatkozó *full* és *empty* állapotjelzőket, illetve egy-egy további vektort a FIFO-kban lévő üres rekeszek számával *(free_space_in_mosi/miso_fifo)*.

Az *SPI-FIFO ADAPTER* modul belső FIFO-i és a kívülről az útválasztóra kapcsolt bővítőmodulok saját FIFO-i között egy futásidőben konfigurálható kapcsolómátrix és két állapotgép teremt kapcsolatot. A kapcsolómátrix tulajdonképpen csupán egy összetett kombinációs hálózat, ami a nyolc pár bővítőmodul-oldali FIFO interfészjel-halmaz közül mindig csak az *xtm_channel_select* bemenete által kijelölt párt kapcsolja az állapotgépekre. E kombinációs hálózat belső szerkezetét a 2-111. és 2-112. ábrák szemléltetik.

VHDL

```
1  xtm_infifo_rdy2push <= xtm_infifo_rdy2push_0 when
2                         xtm_channel_select = 0 else
3                         xtm_infifo_rdy2push_1 when
4                         xtm_channel_select = 1 else
5                         xtm_infifo_rdy2push_2 when
6                         xtm_channel_select = 2 else
7                         xtm_infifo_rdy2push_3 when
8                         xtm_channel_select = 3 else
9                         xtm_infifo_rdy2push_4 when
10                        xtm_channel_select = 4 else
11                        xtm_infifo_rdy2push_5 when
12                        xtm_channel_select = 5 else
13                        xtm_infifo_rdy2push_6 when
14                        xtm_channel_select = 6 else
15                        xtm_infifo_rdy2push_7;
16
17 xtm_infifo_full       <= [...]
18 xtm_outfifo_rdy2pop   <= [...]
19 data_from_xtm_outfifo <= [...]
20 xtm_outfifo_empty     <= [...]
```

Áramköri struktúra



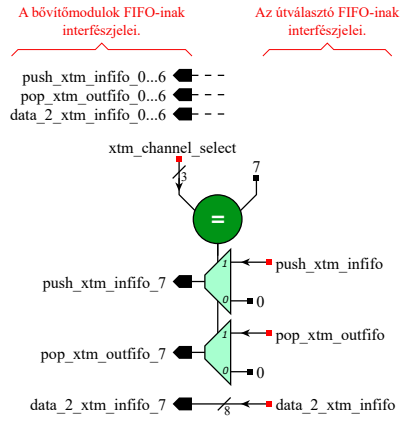
2-111. ábra. *A kapcsolómátrix belső szerkezete (1)*

A bővítőmodulok FIFO interfészeinek kimeneti jelei (pl. *xtm_infifo_rdy2push_0...7*) a kapcsolómátrixon belül egy-egy nyolcbemenetű multiplexerbe futnak be. A multiplexer kiválasztójeleként használt *xtm_channel_select* vektor értéke alapján dől el, hogy e nyolc jelből a belső állapotgép melyiket „látja”.

VHDL

```
1  push_xtm_infifo_0   <= [...]
2  pop_xtm_outfifo_0   <= [...]
3  data_2_xtm_infifo_0 <= [...]
4
5  [...]
6
7  push_xtm_infifo_6   <= [...]
8  pop_xtm_outfifo_6   <= [...]
9  data_2_xtm_infifo_6 <= [...]
10
11 push_xtm_infifo_7   <= push_xtm_infifo when
12                        xtm_channel_select = 7 else '0';
13
14 pop_xtm_outfifo_7   <= pop_xtm_outfifo when
15                        xtm_channel_select = 7 else '0';
16
17 data_2_xtm_infifo_7 <= data_2_xtm_infifo;
```

Áramköri struktúra



2-112. ábra. *A kapcsolómátrix belső szerkezete (2)*

A bővítőmodulok FIFO interfészeinek bemeneteit kezelő kombinációs hálózatok kicsit más felépítésűek. Mivel itt azt kell elérnünk, hogy az éppen nem kiválasztott bővítőmodul bemeneti FIFO-in ne kezdeményezzünk *push*, kimeneti FIFO-in pedig *pop* műveletet, ezért e jeleket egy-egy kétbemenetű multiplexerrel hajtjuk meg, ami gondoskodik róla, hogy az *xtm_channel_select* vektor által ki nem választott bővítőmodul *push* és *pop* bemenetei alacsony szintre legyenek húzva. A bővítőmodulok bemeneti FIFO-inak adatbemenetén *(data_2_xtm_infifo_0...7)* a multiplexer megspórolható, hiszen ha az adott FIFO *push* bemenete folyamatosan nulla értékű, akkor ennek az adatvektornak a változásai irrelevánsak, azokról a bővítőmodul FIFO-ja „nem vesz tudomást".

A kapcsolómátrix belső, tehát az SPI-FIFO útválasztó modul belső logikája felé néző FIFO interfészjelei mindig az *xtm_channel_select* vektor által kiválasztott bővítőmodul bemeneti és kimeneti FIFO-inak jeleit képviselik. Ezen a ponton tehát a kiválasztott bővítőmodul FIFO interfészei és az *SPI-FIFO ADAPTER* modul belső FIFO-inak interfészei néznek szembe egymással. Ezek az interfészek kliens típusú szinkron kézfogásos interfészek, vagyis adatátvitelt nem kezdeményeznek, ezért – a 2-6. ábrán látható sémának megfelelően – valamilyen illesztőlogikát kell beépítenünk, ami a FIFO-k közötti adatmozgatást végrehajtja. Ezt a szerepet két állapotgép tölti be; a *MOSI FIFO POP FSM* az *SPI-FIFO ADAPTER MOSI FIFO*-jából mozgatja át az adatbájtokat a kiválasztott bővítőmodul bemeneti FIFO-jába, a *MISO FIFO PUSH FSM* pedig fordítva; a bővítőmodul kimeneti FIFO-jának tartalmát másolja át az *SPI-FIFO ADAPTER MISO FIFO*-jába. E két állapotgép vázlatát a 2-113., vezérlési állapotaikat pedig a 2-114. ábra szemlélteti.

VHDL – MOSI FIFO POP FSM

```
1  L_MOSI_FIFO_POP_FSM: process ( clk, rst ) begin
2  | if ( rst = '1' ) then
3  | | mosifpf_state <= pop_mosi_fifo_1;
4  | | pop_mosi_fifo <= '0';
5  | | push_xtm_infifo <= '0';
6  | | data_2_xtm_infifo <= X"00";
7  | elsif ( rising_edge(clk) ) then
8  | |
9  | | case mosifpf_state is
10 | | |
11 | | | [...]
12 | | |
13 | | | when others => null;
14 | | |
15 | | end case;
16 | |
17 | end if;
18 end process;
```

VHDL – MISO FIFO PUSH FSM

```
1  L_MISO_FIFO_PUSH_FSM: process ( clk, rst ) begin
2  | if ( rst = '1' ) then
3  | | misofpf_state <= pop_xtm_fifo_1;
4  | | pop_xtm_outfifo <= '0';
5  | | push_miso_fifo <= '0';
6  | | data_2_miso_fifo <= X"00";
7  | elsif ( rising_edge(clk) ) then
8  | |
9  | | case misofpf_state is
10 | | |
11 | | | [...]
12 | | |
13 | | | when others => null;
14 | | |
15 | | end case;
16 | |
17 | end if;
18 end process;
```

2-113. ábra. *Az SPI-FIFO útválasztó belső állapotgépeinek vázlata*

Az *SPI-FIFO ADAPTER* belső FIFO-i a *dualclock_fifo_hswrp* IP-mag példányai, és ugyanezt feltételezzük a bővítőmodulok bemeneti és kimeneti FIFO-iról is. Ez azt jelenti, hogy az útválasztó állapotgépei két-két teljesen egyforma, szinkron kézfogásos interfészű FIFO között mozgatják az adatokat.[76] Ezeken az adatátviteli csatornákon a FIFO-tárolók már önmagukban is megvalósítanak egy forgalomszabályozási mechanizmust a telítettségükre utaló *empty* és *full* állapotjelzőik által. A két adatmozgató állapotgépet tehát elkészíthetnénk oly módon, hogy csupán ezeknek az

[76] Ha az egyes interfészjelek elnevezésétől elvonatkoztatva vizsgáljuk a HDL-modelleket, akkor észrevehetjük, hogy a két állapotgép működése teljesen megegyezik. Ez azt jelenti, hogy nincs akadálya e részfunkció „absztrahálásának", kiszervezésének, vagyis egy olyan almodul létrehozásának, ami két szinkron kézfogásos FIFO között végez adatmozgatást, függetlenül e FIFO-k rendszertechnikai szerepétől. Az SPI-FIFO útválasztóba azután ebből az általánosított adatmozgató almodulból beépíthetünk két külön példányt a 2-110. ábra *MOSI FIFO POP FSM* és *MISO FIFO PUSH FSM* állapotgépek helyére.

VHDL – MOSI FIFO POP FSM

```
when pop_mosi_fifo_1 =>
  if (
    enable = '1' and
    mosi_fifo_empty = '0'
  ) then
    pop_mosi_fifo <= '1';
    mosifpf_state <= pop_mosi_fifo_2;
  end if;

when pop_mosi_fifo_2 =>
  pop_mosi_fifo <= '0';
  mosifpf_state <= pop_mosi_fifo_3;

when pop_mosi_fifo_3 =>
  if ( mosi_fifo_rdy2pop = '1' ) then
    mosifpf_state <= push_xtm_fifo_1;
  end if;

when push_xtm_fifo_1 =>
  if ( xtm_infifo_full = '0' ) then
    if ( xtm_infifo_rdy2push = '1' ) then
      data_2_xtm_infifo <= data_from_mosi_fifo;
      push_xtm_infifo <= '1';
      mosifpf_state <= push_xtm_fifo_2;
    end if;
  end if;

when push_xtm_fifo_2 =>
  push_xtm_infifo <= '0';
  mosifpf_state <= pop_mosi_fifo_1;
```

VHDL – MISO FIFO PUSH FSM

```
when pop_xtm_fifo_1 =>
  if (
    enable = '1' and
    xtm_outfifo_empty = '0'
  ) then
    pop_xtm_outfifo <= '1';
    misofpf_state <= pop_xtm_fifo_2;
  end if;

when pop_xtm_fifo_2 =>
  pop_xtm_outfifo <= '0';
  misofpf_state <= pop_xtm_fifo_3;

when pop_xtm_fifo_3 =>
  if ( xtm_outfifo_rdy2pop = '1' ) then
    misofpf_state <= push_miso_fifo_1;
  end if;

when push_miso_fifo_1 =>
  if ( miso_fifo_full = '0' ) then
    if ( miso_fifo_rdy2push = '1' ) then
      data_2_miso_fifo <= data_from_xtm_outfifo;
      push_miso_fifo <= '1';
      misofpf_state <= push_miso_fifo_2;
    end if;
  end if;

when push_miso_fifo_2 =>
  push_miso_fifo <= '0';
  misofpf_state <= pop_xtm_fifo_1;
```

2-114. ábra. *Az SPI-FIFO útválasztó belső állapotgépeinek vezérlési állapotai*

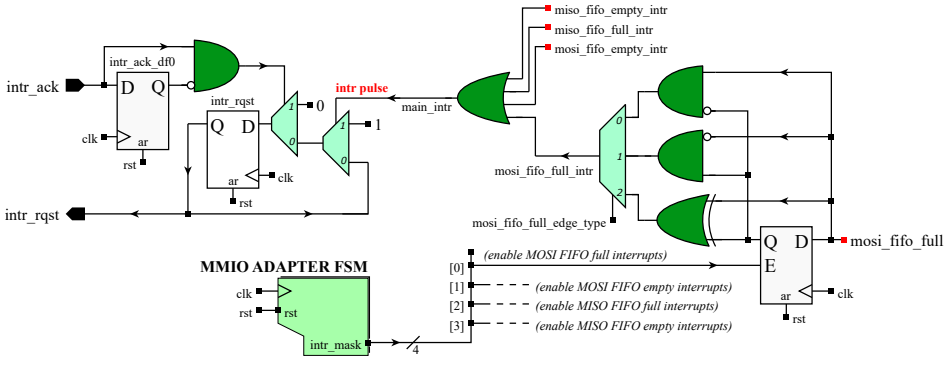
állapotjelzőknek az értékét vegyék figyelembe, vagyis ha fizikailag lehetséges egy adat átmozgatása, akkor az automatikusan történjen is meg, legyen a működés teljesen transzparens. Ugyanakkor a funkcióbővítő rendszer felhasználója számára hasznos lehet, ha ezeknek az állapotgépeknek a működését szüneteltetni tudja, megakadályozva a bővítőmodulok kimeneti FIFO-inak kiürítését és bemeneti FIFO-ik teleírását. Ennek érdekében létrehozunk egy futásidőben beállítható, magas-aktív engedélyezőjelet, amelynek alacsony értéke mellett az állapotgépek nem kezdeményeznek adatátvitelt a FIFO-k között akkor sem, ha annak egyéb akadálya nincs (2-114. ábra: 3. sor). Figyeljük meg, hogy az engedélyezőjel olvasására csupán a több lépésből álló, két szinkron kézfogásos tranzakciót végrehajtó állapotsorozatok elején kerül sor. Így biztosított, hogy az egyik oldalról kiolvasott adat akkor sem veszik el, ha a felhasználó „menet közben” tiltja le az állapotgépek működését. A tiltás ui. csak a *pop+push* műveletpár végrehajtása után juthat érvényre, adatvesztés tehát nem léphet fel.

Az üzemeltetői SPI-interfész célja, hogy tehermentesítse a *CPU*-t és az MMIO-buszrendszert az üzemeltető rendszer és a bővítőmodulok közötti adatmozgatások közvetlen megvalósításával. Az SPI-FIFO útválasztó konfigurálása után az adatmozgatást az SPI-interfész master egysége, vagyis az üzemeltető rendszer vezényli, a folyamat nem igényli a *CPU* állandó közreműködését. A *CPU*-n futó programból persze nyomon követhető az adatátvitel a beépített FIFO-k *full* és *empty* állapotjelzőin, valamint a FIFO-k szabad kapacitását leíró *free_space_in_miso/mosi_fifo* vektorokon keresztül, de az adatmozgatás ezek állandó monitorozása nélkül is megvalósítható. A nagyfokú autonómia miatt különösen fontos azon események halmazának meghatározása, amelyek megszakításkérés generálását idézik elő az SPI-FIFO útválasztóban. A 2-115. ábra egy egyszerű, de rugalmasan testreszabható megoldást mutat be.

VHDL

```
generic (
  [...]
  mosi_fifo_full_edge_type: integer range 0 to 2; -- 0: rising, 1: falling, 2: both
);

L_INTR_GEN: process ( clk, rst ) begin
  if ( rst = '1' ) then intr_ack_df0 <= '0'; intr_rqst <= '0'; mosi_fifo_full_df0 <= '0';
  elsif ( rising_edge(clk) ) then
    |
    intr_ack_df0 <= intr_ack;
    |
    if ( intr_ack_rising = '1' ) then intr_rqst <= '0';
    elsif ( main_intr = '1' ) then intr_rqst <= '1';
    end if;
    |
    if ( intr_mask(0) = '1' ) then mosi_fifo_full_df0 <= mosi_fifo_full; end if;
    |
  end if;
end process;

intr_ack_rising <= intr_ack and not intr_ack_df0;

main_intr <= mosi_fifo_full_intr or mosi_fifo_empty_intr or miso_fifo_full_intr or miso_fifo_empty_intr;

mosi_fifo_full_intr <= (mosi_fifo_full and not mosi_fifo_full_df0) when mosi_fifo_full_edge_type = 0 else
                       (not mosi_fifo_full and mosi_fifo_full_df0) when mosi_fifo_full_edge_type = 1 else
                       (mosi_fifo_full xor mosi_fifo_full_df0);
```

Áramköri struktúra



2-115. ábra. *Az SPI-FIFO útválasztó szerkezete (2)*

Ebben az esetben a megszakításkérést lényegében az *SPI-FIFO ADAPTER* belső FIFO-inak állapotváltozásaihoz kötjük; mindkét FIFO *full* és *empty* állapotjelzőinek változásait nyomon követjük egy-egy konfigurálható éldetektorral. A 2-115. ábra csak a *MOSI FIFO full* kimenetére beépített konfigurálható éldetektort szemlélteti, a többi állapotjelzőre ugyanennek a struktúrának a másolatait képzeljük el. A megszakítási rendszer testreszabhatósága abban nyilvánul meg, hogy egyrészt ezeknek az éldetektoroknak a típusa – fel/lefutóél vagy mindkettő – állapotjelzőnként egy-egy szintézis-paraméterrel választható ki, másrészt működésük futásidőben engedélyezhető vagy letiltható az *intr_mask* vektor megfelelő beállításával.

Az SPI-FIFO útválasztó modul egy MMIO periféria-áramkör, amelynek MMIO írás- és olvasásműveletekkel – tehát a *CPU*-n futó program *store* és *load* utasításaival – elérhető konfigurációs lehetőségeit és állapotjelzőit a 2-12. táblázat foglalja össze.

Cím azonosítója és típusa		A címhez rendelt funkció leírása
address_enable	W	A bővítőmodulok és az *SPI-FIFO ADAPTER* FIFO-i közötti adatátvitelt végző állapotgépek működését vezérlő, magas-aktív engedélyezőjel címe.
address_xtm_channel_select	W	Az SPI-interfészhez aktuálisan hozzárendelt bővítőmodul indexét tároló regiszter címe. Mivel az SPI-FIFO útválasztó legfeljebb nyolc bővítőmodult tud kezelni, ezért e regiszternek csak az alsó három bitje tartalmaz releváns információt, az MMIO írásműveletek adatmezőjének felső 13 bitje nem értelmezett.
address_intr_mask	W	A megszakításkérést kiváltó eseményeket maszkoló regiszter címe. E regiszter beállításával testreszabható, hogy mely események bekövetkezését szeretné a felhasználó a *CPU*-nak küldött megszakításkéréssel jelezni. [0]: változás a *MOSI FIFO full* kimenetén [1]: változás a *MOSI FIFO empty* kimenetén [2]: változás a *MISO FIFO full* kimenetén [3]: változás a *MISO FIFO empty* kimenetén
address_spi_fifos_flags	R	Az *SPI-FIFO ADAPTER* modul belső FIFO-inak állapotát leíró regiszter címe: [0]: *MOSI FIFO full* [1]: *MOSI FIFO empty* [2]: *MISO FIFO full* [3]: *MISO FIFO empty*
address_free_space_in_mosi_fifo	R	A *MOSI FIFO* szabad kapacitását tároló regiszter címe.
address_free_space_in_miso_fifo	R	A *MISO FIFO* szabad kapacitását tároló regiszter címe.
address_spi_if_cmd	W	Az *SPI-FIFO ADAPTER* modul *cmd_push* és *cmd_pop* bemeneteit meghajtó vezérlőregiszter címe: [0]: *cmd_push* [1]: *cmd_pop* E regiszter segítségével a felhasználó külön-külön engedélyezheti vagy tilthatja az SPI-interfészen a master → slave és slave → master irányú adatátvitelt.
address_spi_if_chunk_size	W	Az *SPI-FIFO ADAPTER* modul futásidejű *chunk_size* paraméterét beállító regiszter címe, ami praktikusan az SPI-interfészen zajló adatforgalom csomagméretét jelenti.

2-12. táblázat. *Az SPI-FIFO útválasztó címtérképe*

A látszólag bonyolult címtérkép minden bejegyzése valójában egyszerű írható vagy olvasható vezérlőregiszterekről szól, így az MMIO-adapter állapotgép – a többi periféria-áramköréhez képest is – egyszerűnek mondható (lásd 2-116. ábra).

VHDL

```
when idle => s_ready_2_cpu <= '1';

             -- Handle MMIO write requests...
             if ( write_strobe_from_cpu_df1 = '1' and cs = '1' ) then
               s_ready_2_cpu <= '0';

               if ( unsigned(address_from_cpu) = address_enable ) then
                 enable <= data_from_cpu(0); state <= wait_for_deassert_strobes;
               elsif ( unsigned(address_from_cpu) = address_xtm_channel_select ) then
                 xtm_channel_select <= unsigned(data_from_cpu(2 downto 0));
                 state <= wait_for_deassert_strobes;
               elsif ( unsigned(address_from_cpu) = address_intr_mask ) then
                 intr_mask <= data_from_cpu(3 downto 0); state <= wait_for_deassert_strobes;
               elsif ( unsigned(address_from_cpu) = address_spi_if_cmd ) then
                 spi_if_adapter_cmd_push <= data_from_cpu(0);
                 spi_if_adapter_cmd_pop <= data_from_cpu(1);
                 state <= wait_for_deassert_strobes;
               elsif ( unsigned(address_from_cpu) = address_spi_if_chunk_size ) then
                 spi_if_adapter_chunk_size <= data_from_cpu; state <= wait_for_deassert_strobes;
               else state <= wait_for_deassert_strobes;
               end if;

             end if;

             -- Handle MMIO read requests...
             if ( read_strobe_from_cpu_df1 = '1' and cs = '1') then
               s_ready_2_cpu <= '0';

               if ( unsigned(address_from_cpu) = address_spi_fifos_flags ) then
                 s_data_2_cpu(15 downto 4) <= (others => '0');
                 s_data_2_cpu(0) <= mosi_fifo_full;
                 s_data_2_cpu(1) <= mosi_fifo_empty;
                 s_data_2_cpu(2) <= miso_fifo_full;
                 s_data_2_cpu(3) <= miso_fifo_empty;
                 state <= wait_for_deassert_strobes;
               elsif ( unsigned(address_from_cpu) = address_free_space_in_mosi_fifo ) then
                 s_data_2_cpu(15 downto mosi_fifo_addr_width+1) <= (others => '0');
                 s_data_2_cpu(mosi_fifo_addr_width downto 0) <= free_space_in_mosi_fifo;
                 state <= wait_for_deassert_strobes;
               elsif ( unsigned(address_from_cpu) = address_free_space_in_miso_fifo ) then
                 s_data_2_cpu(15 downto miso_fifo_addr_width+1) <= (others => '0');
                 s_data_2_cpu(mosi_fifo_addr_width downto 0) <= free_space_in_miso_fifo;
                 state <= wait_for_deassert_strobes;
               else state <= wait_for_deassert_strobes;
               end if;

             end if;

when wait_for_deassert_strobes => if (
                                    read_strobe_from_cpu_df1 = '0' and
                                    write_strobe_from_cpu_df1 = '0'
                                  ) then state <= idle;
                                  end if;
```

2-116. ábra. *Az SPI-FIFO útválasztó MMIO-adapter állapotgépének vezérlési állapotai*

Pár gondolat az MMIO-címek hozzáférési típusáról…

Az SPI-FIFO útválasztó MMIO-címtérképének ebben a kialakításában csak olyan címek szerepelnek, amelyeken vagy csak írás, vagy csak olvasás művelet hajtható végre. Egy állapotjelző írásának természetesen semmi értelme nincs, ezért kézenfekvő, hogy ezeket csak-olvasható regiszterként definiáljuk, és az MMIO-adapter állapotgépben az e címekre érkező írás kéréseket egyszerűen figyelmen kívül hagyjuk (pl. 2-116. ábra: 20. sor). A csak-írható vezérlőregiszterekkel azonban más a helyzet. Mondhatnánk, hogy egy beállítható regiszter visszaolvasását lehetővé tenni felesleges, hiszen ha a CPU-n futó programon kívül semmi nem módosíthatja az adott regiszterben tárolt értéket, akkor az értelemszerűen mindig az lesz, amit a program legutóbb beleírt. Amíg a CPU-n futó program egyszerű felépítésű, addig ez így is van. Ha azonban

a CPU-n olyan programot futtatunk, ami nem csupán egyetlen főprogramból, hanem egymással látszólag párhuzamosan futó szubrutinokból (taszkokból) áll, akkor hamar kerülhetünk abba a helyzetbe, hogy egy adott szubrutinban szükségünk van egy olyan regiszter értékére, amit egy másik szubrutin állított be. E regiszter értékét kiolvasni viszont nem áll módunkban, ha azt a periféria-áramkörben csak-írható vezérlőregiszterként hoztuk létre. A probléma persze megkerülhető úgy, hogy az adott regiszter értékének tárolására definiálunk egy bejegyzést egy, minden szubrutin számára hozzáférhető általános célú tárolóban (pl. a CPU belső regisztertömbjének vagy egy MMIO operatív tárnak egy rekeszében). Ennél elegánsabb megoldás, ha az adott regisztert írható-olvasható típusúvá tesszük a periféria-áramkör HDL-modelljében. Ahogy szinte minden tervezői döntés, ez is egy kompromisszumot rejt; ha minden vezérlőregisztert írható-olvasható típusúvá teszünk, akkor a CPU-n futó program egyszerűsödik, áttekinthetőbbé válik, ugyanakkor az MMIO-címdekóderek nagyobbak lesznek, ami sok periféria-áramkör esetén akár jelentős erőforrásigény-növekményt is jelenthet.

2.9. Hibakeresést segítő modulok

Áttanulmányozásra javasolt alfejezetek:

***4.3.1.** A soros kommunikáció alapáramkörei – UART-adó és -vevő (+ **4.3.4**. Nagysebességű UART-vevő)*

A digitális rendszerek tervezése során az összetettség növekedésével a belső működés részleteinek vizsgálata egyre nehezebbé válik. Bár funkcióbővítő rendszerünk egészen „emberi léptékű", aránylag könnyen áttekinthető, a mai modern rendszerchipeknél (**S**ystem-**o**n-**C**hip, SoC) nagyságrendekkel egyszerűbb RTL-modell, mégis elképzelhetetlen, hogy a toplevel modulját ne fokozatosan, újabb és újabb almodulok hozzáadásával és a félkész modell folyamatos ellenőrzésével készítsük el. Különösen igaz ez akkor, ha új, egyedi funkciójú bővítőmodulokat készítünk, amiket önmagukban is és a rendszerbe integrálva is feltétlenül ellenőriznünk kell (lásd még DRMSz: 5.6. alfejezet: almodulszintű és toplevel verifikáció).

A toplevel RTL-modell és a *CPU*-n futó szoftver együttesének kifejlesztését – az ún. hardver-szoftver együttes tervezést (hardware-software co-design) – tehát valami módon támogatnunk kell. Akár úgy is, hogy a rendszerhez – annak költségeit növelve – kifejezetten csak a hibakeresést elősegítő hardverelemeket adunk hozzá. Funkcióbővítő rendszerünk alapkiépítésébe három ilyen almodult integrálunk.[77]

Ahogy az 1-17. ábrán is látható, az MMIO-buszrendszerre kifejezetten a *CPU*-n futó szoftver kifejlesztését segítő UART (lásd 2.8.1. pont) és GPIO (lásd 2.8.2. pont) periféria-áramköröket

[77] Ha újrakonfigurálható hardverrel dolgozunk – mint ahogy a funkcióbővítő rendszer esetében is tesszük –, akkor ez a költségnövekmény kevésbé fájdalmas, hiszen e többlet erőforrásokat akár utólag is kivehetjük a rendszerből, „helyet csinálva" újonnan felmerülő igényeket kielégítő komponenseknek. Standard cellás technológia esetén a gyártást követően nyilván ezt nem tehetjük meg, így a kifejezetten csak a tesztelést segítő erőforrások látszólag feleslegesen növelik a költségeket, hiszen a végtermékben nem csinálnak semmit. Ez azonban több szempontból is hibás érvelés. Egyrészt egy összetett, a felhasználó által utólag is módosítható szoftver-komponenst is tartalmazó rendszerben – pl. egy rendszerchip esetén – a hardver-szoftver integráció valójában csak a gyártást követően történik, amikor is a megfigyelhetőséget javító hardverelemek roppant értékesnek bizonyulhatnak. Másrészt ezeknek a többleterőforrásoknak egy része nem is a tervezési, hanem kifejezetten a gyártási hibák feltárására szolgál. Mivel az integrált áramkörök legyártásakor a kihozatal – a hibátlan és hibás chipek számának arányát kifejező mérőszám – talán sosem fogja elérni a 100%-ot, a tesztelést segítő integrált hardverelemek (pl. scan-path, boundary-scan) nélkül garantálhatatlan lenne, hogy a végfelhasználóhoz csak hibátlan termék jusson el.

kapcsolunk, amelyek a továbbiakban szerves részét képezik az alaprendszernek, függetlenül attól, hogy az alaprendszer funkcionálisan nélkülük is teljes lehetne, rendeltetését maradéktalanul betölthetné.

A harmadik, a rendszertervben *HWDBGR (**H**ard**w**are **D**e**b**u**gg**e**r)* névvel illetett almodul valójában nem egy valódi funkcionális egység, csupán annak váza, amit nagyméretű toplevel modellek összeállításakor célszerű beépíteni. Ez a váz tulajdonképpen csak egy UART adó- és vevőmodul és egy ezeket vezérlő, „félkész” állapotgép. A toplevel-integráció során elkészített hardver-prototípusok vizsgálata során e félkész egységet mindig az adott körülmények, felderítendő hibajelenségek alapján változtatjuk, fejlesztjük, kihasználva az FPGA technológia újrakonfigurálhatóságát. A 2-117. ábra egy olyan példát mutat be, amiben a *HWDBGR* modul a *CPU*-nak küldött megszakításkérések detektálásakor egy-egy üzenetet küld az UART-adón keresztül a felhasználónak.

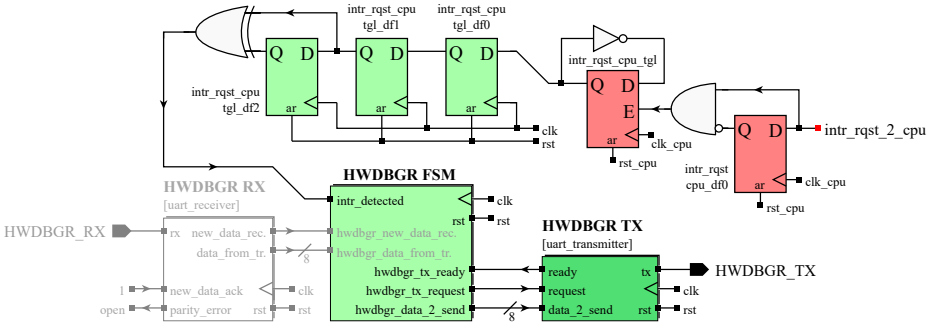
Verilog

```
1  always @ ( posedge clk_cpu, posedge rst_cpu ) begin
2  | if ( rst_cpu == 1 ) begin [...]
3  | end else begin
4  | | intr_rqst_cpu_df0 <= intr_rqst_cpu;
5  | | if ( intr_rqst_cpu_rising == 1 )
6  | |   intr_rqst_cpu_tgl <= ~intr_rqst_cpu_tgl;
7  | | end
8  | end
9
10 always @ ( posedge clk, posedge rst ) begin
11 | if ( rst == 1 ) begin [...]
12 | end else begin
13 | | intr_rqst_cpu_tgl_df0 <= intr_rqst_cpu_tgl;
14 | | intr_rqst_cpu_tgl_df1 <= intr_rqst_cpu_tgl_df0;
15 | | intr_rqst_cpu_tgl_df2 <= intr_rqst_cpu_tgl_df2;
16 | end
17 end
18
19 assign intr_rqst_cpu_rising = intr_rqst_cpu &
20                               ~intr_rqst_cpu_df0;
21
22 assign intr_detected = intr_rqst_cpu_tgl_df2 ^
23                        intr_rqst_cpu_tgl_df1;
24
25 uart_transmitter #( [...] ) // HWDBGR TX
26   L_HWDBGR_TX (
27   | .clk(clk), .rst(rst),
28   | .request(hwdbgr_tx_request),
29   | .ready(hwdbgr_tx_ready),
30   | .data_2_send(hwdbgr_data_2_send),
31   | .tx(HWDBGR_TX)
32   );
33
```

Verilog (folyt.)

```
34 // HWDBGR FSM
35 always @ ( posedge clk, posedge rst ) begin
36 | if ( rst == 1 ) begin [...]
37 | end else begin
38 | |
39 | | case ( hwdbgr_state )
40 | | | idle: if ( intr_detected == 1 ) begin
41 | | |         hwdbgr_data_2_send <= 42;
42 | | |         hwdbgr_tx_request <= 1;
43 | | |         hwdbgr_state <= s1;
44 | | |       end
45 | | |
46 | | | s1: hwdbgr_state <= s2;
47 | | |
48 | | | s2: if ( hwdbgr_tx_ready == 1 ) begin
49 | | |       hwdbgr_tx_request <= 0;
50 | | |       hwdbgr_state <= idle;
51 | | |     end
52 | | |
53 | | | [...]
54 | | |
55 | | | default:    ;
56 | | |
57 | | endcase
58 | |
59 | end
60 end
```

Áramköri struktúra



2-117. ábra. *A CPU-nak küldött megszakításkéréseket jelző HWDBGR modul megvalósítása*

Az ehhez hasonló modulok tervezésekor természetesen körültekintően kell eljárnunk. Az UART-kapcsolat egyszerűen kiépíthető, kényelmes ugyan, de sebessége messze elmarad a processzoros rendszer belsejében zajló folyamatok sebességétől. A fenti esetben például a *CPU* sok-sok megszakítást kiszolgálhat, mire az UART-kapcsolaton egyetlen üzenet elér a felhasználóhoz, így nem garantálható, hogy nem lesznek detektálatlan megszakításkérések. Ez a konkrét probléma kezelhető egy időablak definiálásával, amin belül a debugger ténylegesen minden megszakítást detektál, majd – akár egy időbélyeggel és egyéb hasznos állapotváltozókkal együtt – eltárol egy blokk-RAM modulban. Az időablak végén e blokk-RAM modul összes bejegyzését egyszerre elküldhetjük az UART-interfészen keresztül a felhasználónak. Ez a megoldás már kísértetiesen hasonlít az FPGA-alapú rendszerek prototípus-tesztelésének elterjedten használt eszközeire, az ún. integrált logikai analizátorokra (pl. Xilinx ChipScope).

3

A bővítőmodulok RTL-tervezése

Az előző fejezetben bemutatott funkcionális egységek egy teljesen általános beágyazott processzoros rendszert alkotnak. A *CPU*-n futó program segítségével kialakítható egy kliens–kiszolgáló viszony az üzemeltető mikrovezérlő és a funkcióbővítő rendszer között, de ez utóbbi szolgáltatáskészlete ebben a formában még nem adna hozzá túl sokat az üzemeltető rendszer működéséhez, legfeljebb a diagnosztikai adatok gyűjtését és a külső, nagykapacitású memóriák illesztésének lehetőségét nevezhetnénk meg az üzemeltető rendszer számára elérhetővé tett többletfunkcióként. A funkcióbővítő rendszer 2. fejezetben bemutatott alapkiépítésének célja valójában az, hogy egy jól definiált fogalmi keretet és konkrét interfész-specifikációkat definiáljon a különféle bővítőmodulok egységes és hatékony integrálásához:

- **A bővítőmodulok speciális MMIO periféria-áramkörök**, belső szerkezetük a 2-74. ábrán látható általános sémának felel meg, vagyis képesek a processzoros rendszerünk részegységei közötti kommunikáció gerincét alkotó buszrendszerre slave egységként kapcsolódni. Az MMIO slave interfész lehetővé teszi, hogy a bővítőmodulok futásidejű konfigurációját megváltoztassuk és állapotukat lekérdezzük a *CPU* által végrehajtott *store* és *load* utasítások segítségével. Annak sincs elvi akadálya, hogy feldolgozandó és feldolgozott adatok forgalmát bonyolítsuk az MMIO-interfészen keresztül a *CPU* és a bővítőmodulok vagy akár – a *CPU* közreműködésével – két bővítőmodul között, bár ebben az esetben körültekintően kell eljárnunk az adatmennyiség vonatkozásában. Mint minden MMIO periféria-áramkör, **a bővítőmodulok is rendelkezhetnek** a *CPU* megszakítási rendszerével – egyúttal a 2.5. pontban bemutatott megszakításvezérlővel – kompatibilis **megszakításkérő interfésszel**.
- **A bővítőmodulok hozzáférhetnek a külső, nagykapacitású operatív tárhoz** a 2.6.2. pontban bemutatott memória-arbiteren keresztül. A bővítőmodulok a memória-arbiter szinkron kézfogásos interfész-specifikáció szerint működő kliensinterfészeihez kapcsolódnak, amelyeken keresztül egyszerű írás- és olvasáskérésekkel fordulhatnak az arbiterhez, közvetetten tehát a külső operatív tárhoz.
- **Az üzemeltető rendszer és a bővítőmodulok között közvetlen** – az MMIO-buszrendszert és a *CPU*-t megkerülő – **adatkapcsolat kiépítésére is van lehetőség** a 2.8.3. pontban részletezett SPI-FIFO útválasztón keresztül. Ez a modul az üzemeltetői SPI-interfész és a bővítőmodulokba integrált FIFO-tárak között teremt kapcsolatot. Ehhez természetesen az kell, hogy a bővítőmodulok belső FIFO-inak *push* és *pop* interfészei kompatibilisek legyenek az SPI-FIFO útválasztó bővítőmodul-oldali – ugyancsak szinkron kézfogásos – FIFO interfészeivel. Ez legegyszerűbben úgy érhető el, ha minden bővítőmodulba a *dual-clock_fifo_hswrp* IP-mag (lásd 4.2.5. pont) példányait építjük be.

A bővítőmodulokat tehát a beágyazó rendszernek e sajátosságait figyelembe véve kell megterveznünk. Funkciójuktól függően a különféle bővítőmodulok nem feltétlenül használnak ki minden rendelkezésre álló lehetőséget, a fenti felsorolásnak tulajdonképpen bármely eleme elhagyható, ha a konkrét bővítőmodul azt feladatköréből adódóan nem igényli.

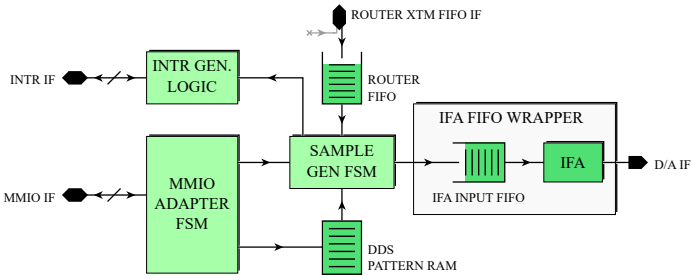
3.1. Hullámforma-generátor

Áttanulmányozásra javasolt alfejezetek:

***4.2.** On-chip adattárolás*

***4.4.8.** Katalógus-áramkörök interfészadapterei – SPI D/A-átalakító (+ **4.4.9**. Órajel-szinkronizálás)*

A digitális jelfeldolgozás tipikusan nagy számítási kapacitást igénylő alkalmazási terület, ezért kézenfekvő, hogy a kis méretűnek és korlátozott számítási teljesítményűnek tételezett üzemeltető rendszer számára az ilyen alkalmazásokban hasznosítható bővítőmodulokat készítsünk. Ebben a pontban egy, a digitális jelfeldolgozás egyik „alapfeladatának" tekinthető funkciót ellátó, a digitális mintasorozatok analóg hullámformává alakítását végző bővítőmodult mutatunk be. E bővítőmodul egyszerűsített blokkvázatát a 3-1. ábra szemlélteti.



3-1. ábra. *A hullámforma-generátor blokkvázlata*

A funkcióbővítő rendszert magában foglaló FPGA természetesen kizárólag digitális funkciók megvalósítására alkalmas, analóg jelek előállítására csak „külső segítséggel" képes. A hullámforma-generátorunk számára egy MCP4921 típusú D/A-átalakító áramkör nyújtja ezt a külső segítséget. A 12-bites, SPI-interfészen keresztül üzemeltethető katalógus-áramkör kényelmes használatát lehetővé tévő interfészadapter RTL-modelljének részleteit a 4.4.8. pont tárgyalja. Az újrafelhasználható IP-magokra jellemző módon az interfészadapter alapváltozatának az analóg mennyiséggé konvertálandó digitális mintákat egy-egy szinkron kézfogásos tranzakcióval adhatjuk át (lásd 4-149. ábra bal oldala). Mivel célunk ezúttal nem egy-egy különálló digitális érték analóggá alakítása, hanem digitális vektorokkal reprezentált minták összefüggő sorozatából álló jelfolyamok analóg jellé konvertálása, ezért az interfészadapternek a 4.4.9. pontban bemutatott, adatfolyam jellegű alkalmazásokra tervezett változatát használjuk (lásd 4-153. és 4-154. ábra).

Ez a változat valójában egy CDC-burkolómodul, ami a *dualclock_fifo_hswrp* IP-mag egy példányával egészíti ki az IFA alapverzióját. Ez a FIFO egyrészt ideiglenes bufferként szolgál az analóggá konvertálandó digitális minták számára, másrészt – dual-clock jellegének köszönhetően – a beágyazó rendszer és az IFA közötti órajel-szinkronizáló funkciót is ellátja. Ez utóbbi képességet

a bővítőmodulban nem használjuk ki, az interfészadapter bemeneti FIFO-jának *push* interfészét és belső logikáját egyazon órajelről járatjuk. A FIFO-alapú burkolómodul ugyancsak hasznos szolgáltatása, hogy a bemeneti FIFO kiolvasásának – egyúttal a külső D/A-átalakító analóggá konvertálandó mintákkal való ellátásának – ütemét egy futásidejű paraméteren (lásd 4-153. ábra: *fifo_pop_delay_cnt_max*) keresztül teszi beállíthatóvá. Így az újabb és újabb analóg értékek beállítása között eltelt idő független a FIFO írásának ütemétől.[78]

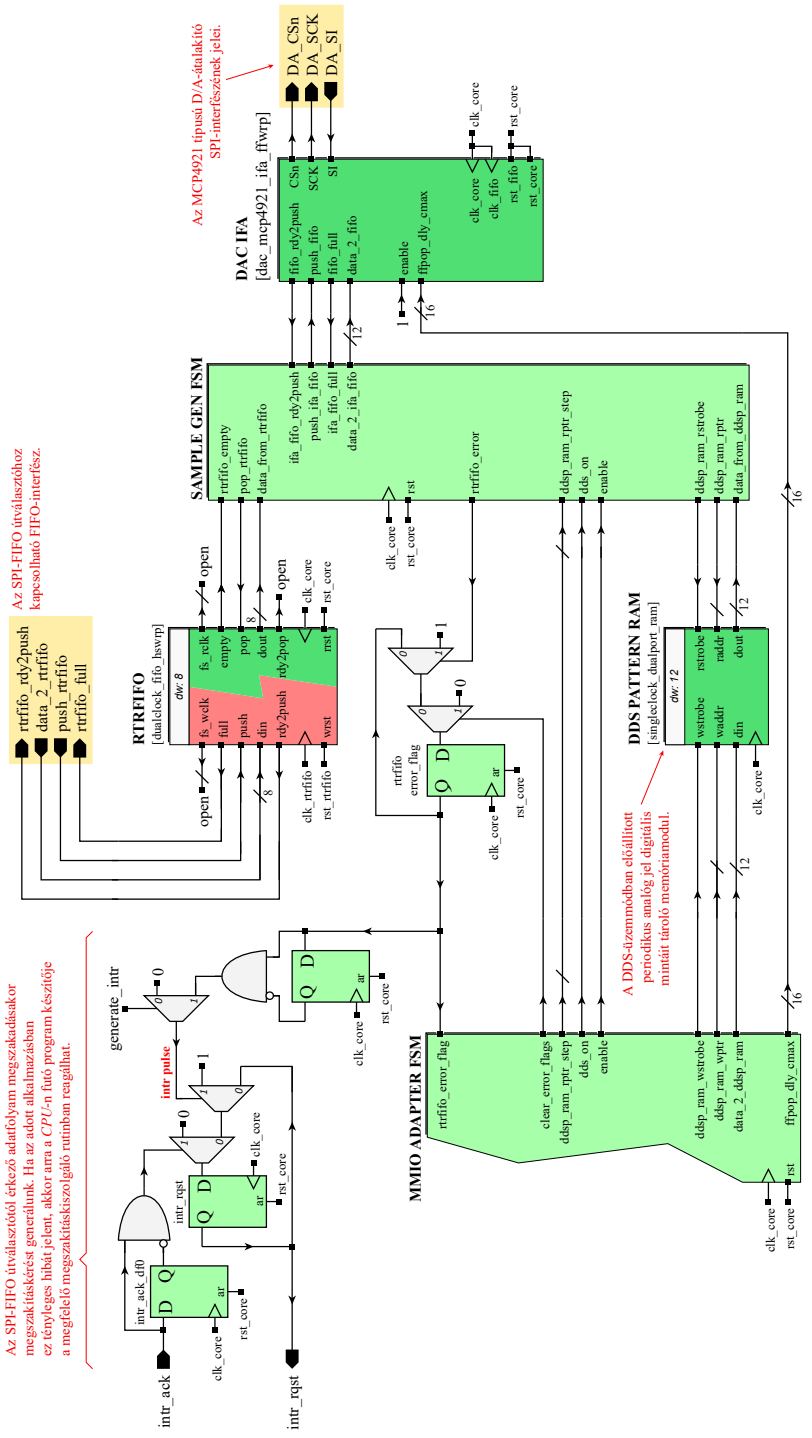
A bővítőmodul kétféle üzemmódban használható, amelyek az interfészadapter bemeneti FIFO-jába írt mintasor forrásában különböznek. Az egyik üzemmódban e mintasor forrása egy másik FIFO-tároló (3-1. ábra: *ROUTER FIFO*), amely segítségével a hullámforma-generátor a 2-109. ábrán látható séma szerint az SPI-FIFO útválasztóhoz illeszthető. Ebben az üzemmódban tehát az analóggá konvertálandó digitális mintasort az üzemeltetői SPI-interfészen keresztül juttatjuk a D/A-átalakító interfészadapterébe. A másik üzemmódban egy előre rögzített digitális mintasor periodikus konverzióját valósítjuk meg. A generálandó analóg jel egy periódusának digitális mintáit a bővítőmodul egy belső RAM-ban (3-1. ábra: *DDS PATTERN RAM*) tárolja, amelynek tartalma az MMIO-interfészen keresztül módosítható, így tetszőleges alakú periodikus analóg jel előállítható. Ezt az üzemmódot közvetlen hullámforma-szintézisnek (**D**irect **D**igital **S**ynthesis, DDS) nevezzük. A hullámforma-generátor működése közben a – szintén MMIO-interfészen keresztül – kiválasztott üzemmódtól függően vagy az SPI-FIFO útválasztó felől érkező mintákat tároló FIFO vagy a rögzített mintákat tároló RAM tartalmát töltjük át a D/A-átalakító interfészadapterének bemeneti FIFO-jába. Ezt az adatmozgatást/másolást egy belső állapotép (3-1. ábra: *SAMPLE GEN FSM*) végzi.

A hullámforma-generátor a *CPU* megszakítási rendszeréhez is képes kapcsolódni. Az SPI-FIFO útválasztótól érkező mintasor folytonosságának megszakadása egy egyértelműen detektálható esemény, amelynek jelentése a konkrét alkalmazástól függ, ezért ezt a bővítőmodul egy megszakításkéréssel jelzi. Az e megszakításkéréseket kiszolgáló szubrutin létrehozása, vagyis a megszakadó adatfolyamra adott alkalmazásfüggő „reakció" kidolgozása a *CPU*-n futó program készítőjének feladata.

A bővítőmodul RTL kapcsolási sémáját a 3-2. ábra szemlélteti. Látható, hogy nem a D/A-átalakító interfészadaptere az egyetlen példányosított újrafelhasználható IP-mag. Az SPI-FIFO útválasztótól érkező mintákat tároló FIFO – az IFA bemeneti FIFO-jához hasonlóan – a 4.2.5. pontban tárgyalt *dualclock_fifo_hswrp* IP-mag egy példánya, amelynek dual-clock képességei ezúttal nem maradnak kihasználatlanul; *push* interfészének órajele *(clk_rtrfifo)* a bővítőmodul magórajelétől *(clk_core)* független. A *clk_rtrfifo* órajelbemenetet a funkcióbővítő rendszer toplevel moduljában az SPI-FIFO útválasztó órajelével kell meghajtanunk. Így elérhető, hogy az útválasztó és a bővítőmodul egymáshoz képest aszinkron órajelekről járhasson, eleget téve a rendszertervben rögzített ezirányú alapelvnek (lásd 1.2.1.3. pont).[79] A DDS-üzemmód periodikusan ismételt mintáit tároló

[78] Ahhoz, hogy az analóg jel folytonos legyen – abban az értelemben, hogy minden D/A-konverzió során a digitális mintasor egy újabb elemének átalakítására kerüljön sor –, természetesen biztosítani kell, hogy a bemeneti FIFO menet közben ne ürüljön ki. Ha erre mégis sor kerül, akkor az IFA a legutóbbi digitális minta konverzióját ismétli addig, amíg újabb minta nem kerül a bemeneti FIFO-ba. Ilyen értelemben tehát a bemeneti FIFO írásának üteme nem teljesen független az egyes D/A-átalakítások között eltelt időtől.

[79] A külső D/A-átalakító SPI-interfésze legfeljebb 20 MHz frekvenciájú órajelet képes fogadni, amihez a hullámforma-generátor magórajelének frekvenciáját 40 MHz-re kell beállítani (ennek magyarázatát lásd 4.3.3.1. pont). Ennél magasabb frekvenciájú órajelforrást tehát a *clk_core* bemenet meghajtására nem érdemes használnunk. Ugyanakkor az SPI-FIFO útválasztó órajelének frekvenciáját érdemes lehet magasabbra állítani, hiszen ez jelentősen befolyásolja az útválasztó és a bővítőmodulok közötti adatátvitel sebességét. A konkrét bővítőmodulok sajátosságaitól függően ennek az interfésznek a sebessége a teljes rendszer számítási teljesítményére is komoly hatással lehet.



3-2. ábra. *A hullámforma-generátor RTL kapcsolási sémája*

DDS PATTERN RAM memória ugyancsak egy újrafelhasználható almodul, a 4.2.1. pontban bemutatott *singleclock_dualport_ram* IP-mag egy példánya. Mivel a minták elhelyezését az MMIO-adapter állapotgép, kiolvasását pedig a *SAMPLE GEN FSM* állapotgép végzi, amelyek mindegyike a bővítőmodul magórajeléről jár, ezért a RAM-modul esetén órajel-szinkronizáló fokozatok beépítésére nincs szükség.

Az összetett RTL-modellek szerkezetére jellemző séma, hogy az újrafelhasználható IP-magok működését egyedi, csak az adott funkcionális egységben értelmezhető illesztőlogikák hangolják össze (glue-logic). Ilyen illesztő funkciót lát el a hullámforma-generátorban a *SAMPLE GEN FSM* állapotgép is, amely a felhasználó (ez esetben a *CPU*-n futó program) által kijelölt üzemmódtól függően vagy a *ROUTER FIFO* vagy a *DDS PATTERN RAM* memóriamodul tartalmát mozgatja/másolja át a D/A-átalakító interfészadapterének bemeneti FIFO-jába. Az állapotgép VHDL-leírásának vázlatát a 3-3. ábra mutatja be.

VHDL

```
L_SAMPLE_GEN_FSM: process ( clk_core, rst_core ) begin
  if ( rst_core = '1' ) then
    state_sgen <= idle; pop_rtrfifo <= '0'; rtrfifo_error <= '0'; ddsp_ram_rstrobe <= '0';
    ddsp_ram_rptr <= (others => '0'); data_2_ifa_fifo <= (others => '0'); push_ifa_fifo <= '0';
  elsif ( rising_edge(clk_core) ) then
  |
  | -- default values of the pulse-like control signals...
  | pop_rtrfifo <= '0'; ddsp_ram_rstrobe <= '0'; push_ifa_fifo <= '0';
  |
  | case state_sgen is
  | |
  | | when idle => if ( enable = '1' ) then
  | |                 if ( dds_on = '1' ) then state_sgen <= feed_from_ddsp_ram_1;
  | |                 else state_sgen <= feed_from_rtrfifo_1;
  | |                 end if;
  | |               end if;
  | |
  | | when feed_from_ddsp_ram_1 => [...]
  | |
  | | when feed_from_rtrfifo_1  => [...]
  | |
  | | when others => null;
  | |
  | end case;
  |
  end if;
end process;
```

3-3. ábra. *A SAMPLE GEN FSM állapotgép VHDL-modelljének vázlata*

VHDL

```
when feed_from_ddsp_ram_1 => ddsp_ram_rstrobe <= '1'; state_sgen <= feed_from_ddsp_ram_2;

when feed_from_ddsp_ram_2 => state_sgen <= feed_from_ddsp_ram_3;

when feed_from_ddsp_ram_3 => data_2_ifa_fifo <= ddsp_ram_dout;

                             if ( ifa_fifo_full = '0' and ifa_fifo_rdy2push = '1' ) then
                               push_ifa_fifo <= '1';
                               state_sgen <= feed_from_ddsp_ram_4;
                             end if;

when feed_from_ddsp_ram_4 => ddsp_ram_rptr <= std_logic_vector (
                                               unsigned(ddsp_ram_rptr) + unsigned(ddsp_ram_rptr_step)
                                             );
                             state_sgen <= idle;
```

3-4. ábra. *A SAMPLE GEN FSM állapotgép vezérlési állapotai (1)*

A *ROUTER FIFO* és a D/A-átalakító interfészadapterének bemeneti FIFO-ja – természetesen nem a vakszerencse folytán – a szinkron kézfogásos interfész-specifikáció szerint működik (lásd 4-20. ábra). A *DDS PATTERN RAM* memóriamodul interfésze még ennél is egyszerűbb, nem kézfogásos jellegű, a beírás és a kiolvasás is mindig megtörténik egyetlen órajelciklus alatt (lásd 4-13. ábra). Mindezen információk birtokában kidolgozhatók az állapotgép vezérlési állapotainak részletei. A 3-4. ábra a DDS-üzemmódban érvényesülő vezérlési állapotokat mutatja be.

Figyeljük meg, hogy bár DDS-üzemmódban a *SAMPLE GEN FSM* állapotgép feladata a mintatároló RAM-ban lévő értékek periodikus átmásolása az IFA bemeneti FIFO-jába, mégis – a 15. sorban látható értékadásnak megfelelően – az egyes bejegyzések FIFO-ba írása után mindig visszatérünk az *idle* állapotba. Erre a kis „kitérőre" csupán azért van szükség, hogy a periodikus másolás közben az MMIO-interfészen érkező írás-kérések következményeként esetleg megváltozó *enable* és/vagy *dds_on* vezérlőjeleknek e változásai érvényesülhessenek, vagyis a felhasználó képes legyen a működést letiltani vagy üzemmódot váltani. Ha időközben egyik vezérlőjel sem változott, akkor a műveletvégzés ismét a periodikus másolást végrehajtó állapotsorozat első állapotában folytatódik *(feed_from_ddsp_ram_1)*.

Az SPI-FIFO útválasztótól érkező mintasorozatot tároló *ROUTER FIFO* és a D/A-átalakító IFA bemeneti FIFO-ja közötti adatmozgatást végző vezérlési állapotokat a 3-5. ábra szemlélteti.

VHDL

```
1  when feed_from_rtrfifo_1 => if ( rtrfifo_empty = '1' ) then
2                                rtrfifo_error <= '1';
3                                if ( enable = '0' ) then state_sgen <= idle; end if;
4                              else
5                                rtrfifo_error <= '0';
6                                pop_rtrfifo <= '1';
7                                state_sgen <= feed_from_rtrfifo_2;
8                              end if;
9
10 when feed_from_rtrfifo_2 => state_sgen <= feed_from_rtrfifo_3;
11
12 when feed_from_rtrfifo_3 => data_2_ifa_fifo(7 downto 0) <= rtrfifo_dout;
13                             state_sgen <= feed_from_rtrfifo_4;
14
15 when feed_from_rtrfifo_4 => if ( rtrfifo_empty = '1' ) then
16                               rtrfifo_error <= '1';
17                               if ( enable = '0' ) then state_sgen <= idle; end if;
18                             else
19                               rtrfifo_error <= '0';
20                               pop_rtrfifo <= '1';
21                               state_sgen <= feed_from_rtrfifo_5;
22                             end if;
23
24 when feed_from_rtrfifo_5 => state_sgen <= feed_from_rtrfifo_6;
25
26 when feed_from_rtrfifo_6 => data_2_ifa_fifo(11 downto 8) <= rtrfifo_dout(3 downto 0);
27
28                             if ( ifa_fifo_full = '0' and ifa_fifo_rdy2push = '1' ) then
29                               push_ifa_fifo <= '1';
30                               state_sgen <= idle;
31                             end if;
```

3-5. ábra. *A SAMPLE GEN FSM állapotgép vezérlési állapotai (2)*

E vezérlésiállapot-sorozat feladata tehát két FIFO közötti adatmozgatás. A FIFO-k mindegyike a *dualclock_fifo_hswrp* IP-mag egy példánya, a feladat tehát első ránézésre teljesen megegyezik azzal, amit az SPI-FIFO útválasztóban már láthattunk (lásd 2-114. ábra). Van azonban egy kis rendszertechnikai bökkenő; míg az SPI-FIFO útválasztó nyolcbites FIFO interfészeket képes kezelni, ennek megfelelően a *ROUTER FIFO* is nyolcbites szószélességű, addig az MCP4921 D/A-átalakító

12-bites, vagyis az interfészadapter bemeneti FIFO-ja is 12-bites adatvektorokat vár. A két FIFO példány tehát mégsem teljesen azonos, hanem egyazon IP-mag két különböző szószélességgel felparaméterezett változata. Az ellentmondást a *SAMPLE GEN FSM* állapotgép oldja fel úgy, hogy az SPI-FIFO útválasztó felől érkező bájtfolyamot valójában bájt-párok sorozataként értelmezi. A bájt-párok elsőként érkező tagja tölti ki az IFA bemeneti FIFO-jába írandó adat alsó nyolc helyiértékét (3-5. ábra: 12. sor), a másodikként érkező bájt alsó négy bitje pedig a felső négyet (3-5. ábra: 26. sor). Így tehát minden IFA-oldali FIFO írást két útválasztó-oldali FIFO olvasás előz meg.[80]

A *SAMPLE GEN FSM* állapotgépben valósul meg az SPI-FIFO útválasztó felől érkező adatfolyam megszakadásának detektálása is. Ha az állapotgép az adatmozgatás során azt tapasztalja, hogy a *ROUTER FIFO* kiürült (3-5. ábra: 1. és 15. sor), akkor az *rtrfifo_error* jel magas szintre állításának következtében az állapotgépen kívül elhelyezett *rtrfifo_error_flag* állapotjelző ugyancsak magas szintre áll be, és – amellett, hogy felfutóéle megszakításkérést generál – úgy is marad mindaddig, amíg az MMIO-interfészen a megfelelő címre érkező kérés nem törli (lásd később). A konkrét alkalmazástól függ, hogy az adatfolyam megszakadása valódi funkcionális hibát jelent-e, ezért ilyen esetekben a *SAMPLE GEN FSM* állapotgép türelmesen várakozik tovább az újabb bájt megérkezésére. A túlzott türelem azonban holtponthoz vezethet, ha az adatfolyam „végleg" megszakadt, ezért e várakozó állapotokba beépítünk egy-egy kiskaput, amelyekkel a megfelelő MMIO kéréssel az állapotgép működése várakozás közben is letiltható/újraindíthatóvá válik. Ezt szolgálják a 3. és 17. sorban látható feltételvizsgálatok, amik az MMIO-adapter állapotgép által meghajtott *enable* engedélyezőjel nullába állítása esetén azonnal *idle* állapotba kényszerítik a *SAMPLE GEN FSM* állapotgépet. A DDS-üzemmódnál látottakhoz hasonlóan folytonos bemeneti adatfolyam esetén is gondoskodni kell arról, hogy a működés letiltható, az üzemmód pedig bármikor megváltoztatható legyen, ezért a sikeres IFA FIFO írások után az állapotgép ezúttal is mindig az *idle* állapotba lép az *enable* és *dds_on* vezérlőjelek esetleges változásainak detektálása céljából.

A hullámforma-generátor címtérképét a 3-1. táblázat részletezi. A hullámforma-generátor címtérképe egyszerű csak-írható és csak-olvasható vezérlő- és állapotregisztereket takar. Az e regiszterek írását és olvasását megvalósító MMIO-adapter állapotgép vezérlési állapotait a 3-6. ábra mutatja be.

[80] A bejövő adatfolyam dekódolásának ez a koncepciója természetesen csak akkor értelmes, ha az üzemeltető rendszer az SPI-FIFO útválasztó SPI-interfészén ezt figyelembe véve küldi az analóggá konvertálandó adatokat.

Cím azonosítója és típusa		A címhez rendelt funkció leírása
address_enable	W	A modul működését engedélyező, magas-aktív vezérlőregiszter címe.
address_dds_on	W	Az üzemmódot – az analóggá alakítandó digitális adatfolyam forrását – kiválasztó vezérlőregiszter címe.
address_ddsp_ram_rptr_step	W	DDS-üzemmódban a generált periodikus hullámforma frekvenciája a tárolt minták megváltoztatása nélkül módosítható oly módon, hogy a mintatároló memória egyes bejegyzésein nem egyesével, hanem nagyobb ugrásokkal, a köztes mintákat kihagyva lépkedünk végig. Az ugrások mértéke az e címen végrehajtott írással módosítható.
address_ddsp_ram_wptr	W	A DDS-üzemmódhoz tartozó mintatároló tartalma bejegyzésenként két MMIO-írással változtatható meg; az *address_ddsp_ram_wptr* címen végrehajtott írás beállítja a felülírandó bejegyzés címét, az *address_write_ddsp_ram* címen végrehajtott pedig beírja magát az adatot a legutóbb beállított címre.
address_write_ddsp_ram	W	
address_fifo_pop_delay_cnt_max	W	A D/A-átalakító interfészadapterének hasonló nevű futásidejű paraméterét meghatározó vezérlőregiszter címe. Segítségével meghatározható, hogy az IFA milyen gyakorisággal olvassa bemeneti FIFO-ját és indítson SPI-tranzakciót a D/A-átalakító felé.
address_status	R	A bővítőmodul állapotára utaló állapotjelzők címe: [0]: Az SPI-FIFO útválasztó felől érkező adatfolyam megszakadását jelző, magas-aktív állapotjelző. [1]: Az SPI-FIFO útválasztó felől érkező mintákat tároló FIFO *empty* jelzése.
address_clear_error_flags	W	Az SPI-FIFO útválasztó felől érkező adatfolyam megszakadását jelző hiba-állapotjelző az e címen végrehajtott írással törölhető.

3-1. táblázat. *A hullámforma-generátor címtérképe*

Külön említést érdemel a DDS-üzemmód mintatároló memóriájának írása. E RAM-modul címbuszának szintézis-paraméterként megvalósított szélessége (lásd 4-13. ábra: *aw* paraméter) a tárkapacitást határozza meg, vagyis azt, hogy a hullámforma-generátor által – közvetetten a D/A-átalakítón keresztül – előállított kimeneti analóg jel egy periódusa legfeljebb hány mintából állhat.[81] A mintatároló minden bejegyzése indirekt módon, két-két MMIO-írással írható felül; az első írás kiválasztja a megszólítandó memóriarekeszt (3-6. ábra: 15. sor), a második pedig beírja az ebbe a rekeszbe szánt adatot (3-6. ábra: 18–19. sor). Ez az adat a D/A-átalakító típusából adódóan mindig 12-bites, így egyetlen MMIO-tranzakció elegendő. Viszont az, hogy a címinformáció átvitelére is csak egyetlen MMIO-tranzakciót és egyetlen, legfeljebb 16-bites vezérlőregisztert *(ddsp_ram_wptr)* használunk, valójában egy tervezői döntés, amelynek következtében a hullámforma-generátor által előállítható analóg jel egy periódusa legfeljebb 65.536 db mintából állhat.

[81] A „legfeljebb" kifejezés azért indokolt, mert a kimeneti analóg jel egy periódusának tényleges mintaszáma az *address_ddsp_ram_rptr_step* cím mögött rejlő vezérlőregiszteren keresztül csökkenthető (vö. 3-4. ábra: 13. sor).

VHDL

```
when idle => s_ready_2_cpu <= '1';

             -- Handle MMIO write requests...
             if ( write_strobe_from_cpu_df1 = '1' and cs = '1' ) then
               s_ready_2_cpu <= '0';
               
               if ( unsigned(address_from_cpu) = address_enable ) then
                 enable <= data_from_cpu(0); state_mmio <= wait_for_deassert_strobes;
               elsif ( unsigned(address_from_cpu) = address_dds_on ) then
                 dds_on <= data_from_cpu(0); state_mmio <= wait_for_deassert_strobes;
               elsif ( unsigned(address_from_cpu) = address_ddsp_ram_rptr_step ) then
                 ddsp_ram_rptr_step <= data_from_cpu(ddsp_ram_aw-1 downto 0);
                 state_mmio <= wait_for_deassert_strobes;
               elsif ( unsigned(address_from_cpu) = address_ddsp_ram_wptr ) then
                 ddsp_ram_wptr <= data_from_cpu(ddsp_ram_aw-1 downto 0);
                 state_mmio <= wait_for_deassert_strobes;
               elsif ( unsigned(address_from_cpu) = address_write_ddsp_ram ) then
                 data_2_ddsp_ram <= data_from_cpu(11 downto 0);
                 ddsp_ram_wstrobe <= '1';
                 state_mmio <= wait_for_deassert_strobes;
               elsif ( unsigned(address_from_cpu) = address_fifo_pop_delay_cnt_max ) then
                 fifo_pop_delay_cnt_max <= data_from_cpu; state_mmio <= wait_for_deassert_strobes;
               elsif ( unsigned(address_from_cpu) = address_clear_error_flag ) then
                 clear_error_flag <= '1'; state_mmio <= wait_for_deassert_strobes;
               else state_mmio <= wait_for_deassert_strobes;
               end if;
               
             end if;

             -- Handle MMIO read requests...
             if ( read_strobe_from_cpu_df1 = '1' and cs = '1') then
               s_ready_2_cpu <= '0';
               
               if ( unsigned(address_from_cpu) = address_status ) then
                 s_data_2_cpu(15 downto 2) <= (others => '0');
                 s_data_2_cpu(0) <= rtrfifo_error_flag; s_data_2_cpu(1) <= rtrfifo_empty;
                 state_mmio <= wait_for_deassert_strobes;
               else state_mmio <= wait_for_deassert_strobes;
               end if;
               
             end if;

when wait_for_deassert_strobes => ddsp_ram_wstrobe <= '0'; clear_error_flag <= '0';

                                  if (
                                    read_strobe_from_cpu_df1 = '0' and write_strobe_from_cpu_df1 = '0'
                                  ) then state_mmio <= idle;
                                  end if;
```

3-6. ábra. *A hullámforma-generátor MMIO-adapter állapotgépének vezérlési állapotai*

3.2. DSP-bővítőmodul – szűrő és spektrumanalizátor

Áttanulmányozásra javasolt alfejezetek:
4.2. *On-chip adattárolás*
4.4.7. *Katalógus-áramkörök interfészadapterei – SPI A/D-átalakító (+* ***4.4.9.*** *Órajel-szinkronizálás)*
4.4.8. *Katalógus-áramkörök interfészadapterei – SPI D/A-átalakító (+* ***4.4.9.*** *Órajel-szinkronizálás)*
4.5.5. *Aritmetika és jelfeldolgozás – FFT*
4.5.6. *Aritmetika és jelfeldolgozás – FIR-szűrő (+* ***4.5.7.*** *RC aluláteresztő szűrő)*

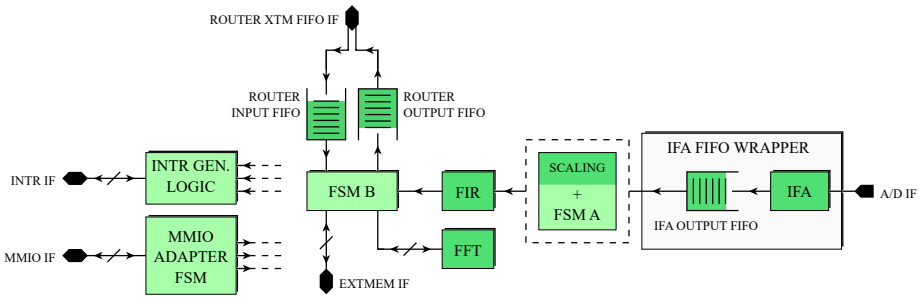
3.2.1. Követelmények és rendszerterv

Az ebben a pontban bemutatott bővítőmodul a hullámforma-generátorhoz hasonlóan ugyancsak a digitális jelfeldolgozás legalapvetőbb funkcióinak egy részét valósítja meg, jelesül egy analóg jel A/D-átalakításával nyert digitális jelfolyam előfeldolgozását (szűrését) és diszkrét Fourier-transzformáltjának, vagyis spektrumának kiszámítását. A rendszerterv felvázolása előtt néhány pontban rögzítsük a modullal szemben támasztott legalapvetőbb elvárásokat, követelményeket:

- A modulnak képesnek kell lennie egy analóg jel A/D-átalakítóval nyert digitalizált mintáinak továbbítására az üzemeltető rendszer felé az SPI-FIFO útválasztón keresztül.
- A modulnak tartalmaznia kell egy, az A/D-átalakítótól érkező digitalizált mintasor manipulálására alkalmas szűrőfokozatot, amely segítségével a mintasor által leírt jel spektrális összetétele befolyásolható.
- A modulnak képesnek kell lennie egy N-elemű mintasor diszkrét Fourier-transzformáltjának (spektrumának) kiszámítására. A transzformálandó mintasor forrása lehet az előző pontban említett szűrőfokozat kimenete (közvetetten az A/D-átalakító), az SPI-FIFO útválasztó (közvetetten az üzemeltető rendszer), valamint a memória-arbiter (közvetetten a külső, nagykapacitású operatív tár).
- A modulnak képesnek kell lennie a diszkrét Fourier-transzformáció eredményének továbbítására az SPI-FIFO útválasztó és a memória-arbiter felé.
- A modul belső számábrázolása 24-bites, kettes-komplemens kódolású fixpontos tört, amelyben az egészrész és a törtrész ábrázolására egyaránt 12 bitet szánunk.

A fenti követelménylista nem túl részletes, csupán a legalapvetőbb elvárásokat rögzíti. Ilyen elnagyolt specifikáció mellett érdemes nagy hangsúlyt fektetni a rendszertervezésre, hiszen a leendő áramkör gyakorlati használhatósága jelentősen függ a finom részletektől. Különösen igaz ez a funkcióbővítő rendszerünk periféria-áramköreire, amelyek képességei önmagukban nem, csak a CPU-val, az azon futó programmal, továbbá az üzemeltető rendszerrel együtt értelmezhetők és értékelhetők; ezekkel kölcsönhatásban oldanak meg összetett feladatokat. Ebben a szellemben tekintsük át a DSP-modul 3-7. ábrán látható egyszerűsített blokkvázlatát.

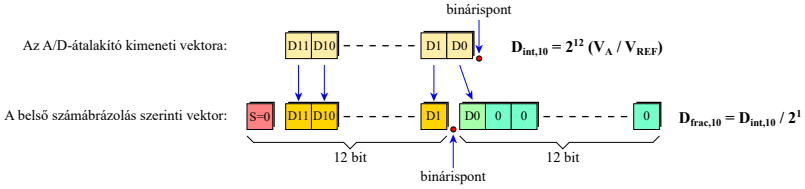
Az analóg jel digitalizálását egy, a funkcióbővítő rendszernek helyet adó FPGA-val SPI-interfészen összekapcsolt katalógus-áramkör, egy MCP3201 típusú A/D-átalakító végzi. Ez az A/D-átalakító egy 12-bites, unipoláris, a digitalizált értéket nemnegatív egész számként előállító eszköz, amelyből a bináris vektorokat egy könnyen integrálható, újrafelhasználható IP-mag, egy interfészadapter segítségével nyerjük ki (RTL-modelljének részleteit lásd 4.4.7. pont). Az



3-7. ábra. *A DSP-modul egyszerűsített blokkvázlata*

IFA alapváltozata egy egyszerű szinkron kézfogásos interfésszel van ellátva, ami a felhasználói kérések beérkezésének pillanatában egy-egy SPI-tranzakciót kezdeményez a katalógus-áramkör felé, lekérdezve az analóg bemenet aktuális értékét. Ebben az alkalmazásban azonban az IFA-t más módon használjuk; nem „véletlenszerű" pillanatokban szeretnénk egy-egy A/D-konverziót és eredményének lekérdezését kezdeményezni, hanem a folyamatosan változónak feltételezett analóg jel digitalizált mintáiból meghatározott – és állandó – időközönként végrehajtott lekérdezéssel egy digitális adatfolyamot előállítani. A 4-151. ábrán látható az A/D-átalakító interfészadapterének az ilyen adatfolyam jellegű alkalmazásokhoz tervezett CDC-burkolómodulja, ami a futásidőben konfigurálható mintavételi idő szerint periodikusan végrehajtott konverziók eredményét egy dual-clock kimeneti FIFO-tárba tölti. A DSP-bővítőmodulban e burkolómodul egy példányát használjuk az A/D-átalakítóval való kölcsönhatás részleteinek lebonyolításához (lásd 3-7. ábra: *IFA FIFO WRAPPER*). A felhasználói szempontból kulcsfontosságú mintavételi idő az *ad_query_delay_cnt_max* nevű futásidejű paraméteren keresztül az IFA saját órajel-frekvenciájának ismeretében állítható be. Mivel a bővítőmodul tervezőjeként sem annak leendő órajel-frekvenciájáról, sem az adott alkalmazásban optimális mintavételi időről nincs információnk, ezért az *ad_query_delay_cnt_max* paramétert konfigurálhatóvá tesszük MMIO-interfészen keresztül.[82]

Az A/D-átalakító a mért analóg mennyiséggel arányos 12-bites, nemnegatív egész számokat szolgáltat. E bináris vektorokból a 3-8. ábrán látható módon képezzük jelfeldolgozó modulunk 24-bites, kettes-komplemens kódolású fixpontos törtszámokat ábrázoló, 12 db egész- és 12 db törtrészbitet tartalmazó adatvektorait.



3-8. ábra. *Áttérés a DSP-modul belső számábrázolási formátumára*

Ezzel a megoldással az A/D-átalakító 0 ... 4095_{10} kimeneti tartományát a belső számábrázolás szerinti 0 ... $2047{,}5_{10}$ tartományra képezzük le, miközben a lehető legtöbb törtrészbitet tartjuk

[82] Az A/D-átalakító maximális mintavételi frekvenciája 100 ksps (**k**ilo**s**ample(s) **p**er **s**ec), ami nyilván egy felső korlátot jelent, egyúttal a bővítőmodul lehetséges alkalmazásainak körét is szűkíti, de a példában bemutatott architekturális és RTL-tervezési megoldások szempontjából ez nem jelent problémát.

fenn a későbbi számítások pontosságának megőrzése érdekében. A kettővel való leosztás a bináris pont – a fixpontos belső számábrázolás előjelbitjének beillesztéséből adódó – relatív eltolódásának következménye. Az A/D-átalakító kimeneti FIFO-jából kiolvasott vektorok tehát már ebben a lépésben is egy transzformáción esnek át, de a széles körű alkalmazhatóság érdekében a további számítások előtt a digitalizált mintasort még egy lineáris skálázásnak vetjük alá (3-7. ábra: *SCALING*). A szűrőfokozatba való továbbítás előtt minden, immár fixpontossá alakított D vektort a

$$D' = (A \times D) + B$$

összefüggés szerint transzformálunk. A kifejezésben szereplő A és B együtthatók segítségével figyelembe vehetők az A/D-átalakító meghajtófokozatának sajátosságai (továbbá a teljes A/D-átalakítási lánc erősítési és ofszet-hibája).[83] Mivel e sajátosságok a különböző alkalmazásokban eltérhetnek, ezért az A és B értékeket futásidőben – MMIO-interfészen – konfigurálhatóvá tesszük. A 3-7. ábra érzékelteti, hogy az A/D-átalakító interfészadapteréből kinyert és a fentiek szerint skálázott, immár az előírt fixpontos számábrázolásnak megfelelő formátumú minták átvitelét az IFA kimeneti FIFO-ja és a szűrőfokozat bemeneti FIFO-ja között egy állapotgép *(FSM A)* vezényli le.

A digitalizált mintasor tehát az A/D-átalakító IFA kimeneti FIFO-jából egy másik újrafelhasználható IP-mag, a 4.5.6. pontban részletesen bemutatott FIR (**F**inite **I**mpulse **R**esponse) szűrő bemeneti FIFO-jába kerül (3-7. ábra: *FIR*). Ez a funkcionális egység a bemeneti FIFO-jába töltött mintasor és a felhasználó által futásidőben – szintén a bővítőmodul MMIO-interfészén keresztül – módosítható szűrőegyütthatókból álló véges elemszámú vektor konvolúciójának eredményét tölti a kimeneti FIFO-jába. Az így előálló mintasor által leírt jel spektrális összetétele a szűrőegyütthatók függvényében változik meg.[84]

Az eddig felsorolt almodulok az A/D-átalakító által mért, időben folyamatosan változó analóg mennyiséget egy digitalizált mintasorrá képezik le, ami a FIR-szűrő kimeneti FIFO-jában tárolódik. A követelménylistából azonban kiolvasható, hogy a DSP-modulnak nemcsak az A/D-átalakítótól származó mintasorokon kell képesnek lennie diszkrét Fourier-transzformációt végezni; a transzformálandó vektor érkezhet az üzemeltető rendszertől az SPI-FIFO útválasztón keresztül és a külső operatív tárból a memória-arbiteren keresztül is. Fogalmazhatunk úgy is, hogy a DSP-modul diszkrét Fourier-transzformációs funkcióját az A/D-átalakítótól független, „off-line" üzemmódban is szeretnénk majd használni. Ennek megfelelően a DSP-modul rendszertervének további tárgyalásához definiálunk két üzemmódot:

---

[83] Az A/D-átalakítók által szolgáltatott értékek fixpontos számábrázolásra való leképezésének kérdéseit a Függelék 5.3.1.1. pontja fejti ki részletesen.

[84] A 3-7. ábra alapján azt vehetjük észre, hogy a szűrő kiiktatását ez az architektúra nem teszi lehetővé. A „szűretlen" mine tasor továbbítása vagy feldolgozása közvetetten mégis megoldható a FIR-szűrő együtthatóinak megfelelő beállításával. A szűrő kimenete a gerjesztésnek (a bemeneti mintasornak) és a szűrő impulzusválaszának konvolúciójaként áll elő. Egy FIR-szűrő impulzusválasza tulajdonképpen nem más, mint együtthatóinak rendezett listája. Ha tehát e szűrőegyütthatókat úgy állítjuk be, hogy az általuk reprezentált impulzusválasznak a bemeneti mintasorral képzett konvolúciója magát a bemeneti mintasort adja vissza, akkor a szűrőt „hatástalanítottuk". A gerjesztést érintetlenül hagyó impulzusválasz az ún. Kronecker-delta függvény (a folytonos idejű Dirac-impulzus diszkrét megfelelője), amit az [1, 0, 0, ...] együtthatók írnak le. A szűrő látszólagos kiiktatásához tehát a felhasználónak annyit kell tennie, hogy a legkisebb indexű szűrőegyütthatót 1-re, a többit pedig 0-ra állítja be.

Mintafolyam üzemmód ◆ Ebben az üzemmódban aktiváljuk az A/D-átalakító interfészadapterét, az *FSM A* állapotgépet és a FIR-szűrőt. Utóbbinak a kimeneti FIFO-jában előálló digitalizált mintasort továbbítjuk az SPI-FIFO útválasztón keresztül az üzemeltető rendszer felé.

FFT-üzemmód ◆ Ebben az üzemmódban a főszerep az N-elemű mintasorok diszkrét Fourier-transzformáltjának kiszámítására alkalmas FFT (**F**ast **F**ourier **T**ransform) almodulé, amit a 4.5.5. pontban részletesen bemutatott, újrafelhasználható IP-mag valósít meg. Ennek a modulnak a felhasználói interfésze a FIR-szűrőéhez hasonló; az N-elemű transzformálandó vektort egy bemeneti FIFO-ba kell töltenünk, a transzformált vektort pedig egy kimeneti FIFO-ból olvashatjuk ki.

A transzformálandó vektor forrását és a transzformált vektor „célállomását" a bővítőmodul MMIO-interfészén keresztül beállítható konfigurációs regiszterek rögzítik. Ha a transzformálandó vektor forrásaként a felhasználó a FIR-szűrő kimeneti FIFO-ját jelöli ki, akkor aktiváljuk az A/D-átalakító interfészadapterét, az *FSM A* állapotgépet és a FIR-szűrőt. A központi vezérlő szerepét betöltő *FSM B* állapotgép feladata ebben az esetben az, hogy periodikusan átmozgasson N db mintát a FIR-szűrő kimeneti FIFO-jából az FFT-almodul bemeneti FIFO-jába, kezdeményezze a transzformációt, majd annak végén az ugyancsak N-elemű transzformált vektort vagy az útválasztóhoz kapcsolt kimeneti FIFO-ba vagy – az *EXTMEM* interfészen és a memória-arbiteren keresztül – a külső operatív tárba írja. Az A/D-átalakítótól származó digitalizált jelfolyam feldolgozása ebben az esetben folyamatos abban az értelemben, hogy a diszkrét Fourier-transzformált kiszámítása és továbbítása alatt a mintavétel nem áll le, és N db mintánként a folyamat periodikusan ismétlődik.[85]

A transzformálandó vektor forrása a követelmények alapján lehet az SPI-FIFO útválasztó és a memória-arbiter is. Előbbi esetben az *FSM B* állapotgép egyetlen N-elemű vektort olvas ki az útválasztóhoz kapcsolt bemeneti FIFO-ból, utóbbi esetben pedig az *EXTMEM* interfészen keresztül a külső operatív tárból, majd továbbítja azt az FFT-almodulnak. A transzformáció végén az *FSM B* állapotgép hasonlóan jár el, mint az előző bekezdésben részletezett esetben; a transzformált vektort az MMIO-interfészen beállított, érvényben lévő konfiguráció függvényében vagy az útválasztóhoz kapcsolt kimeneti FIFO-ba vagy az *EXTMEM* interfészen keresztül a külső operatív tárba írja. Ebben az esetben az A/D-átalakító interfészadaptere, valamint az *FSM A* állapotgép és a FIR-szűrő inaktív, és a DSP-modul egyetlen diszkrét Fourier-transzformált kiszámítása után további utasításig tétlen állapotban várakozik.

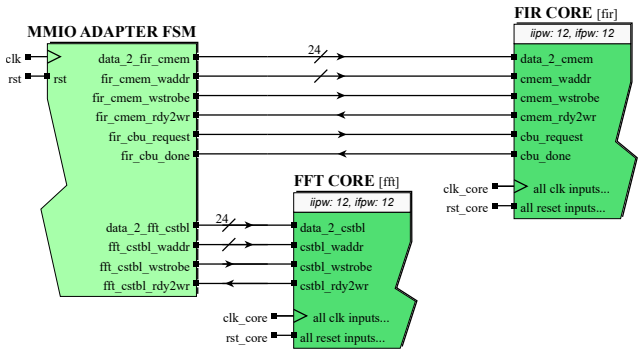
3.2.2. RTL-modell

A DSP-bővítőmodul legösszetettebb részegységei a FIR-szűrő és az FFT IP-mag. Feladatát mindkét almodul csak egy előzetes konfigurációs műveletsor elvégzése után képes ellátni. A FIR-szűrő üzemkész állapotba hozásához a felhasználónak a szintézis-paraméterként meghatározott számú szűrőegyütthatót egy belső memóriába (lásd 4-251. ábra: *CMEM*) kell betöltenie, majd kezdeményeznie kell azok áttöltését a belső konfigurációs regiszterekbe (lásd 4-248. ábra: *cbufs*). Az FFT modulnak az N-elemű diszkrét Fourier-transzformáció elvégzéséhez szüksége van a cos függvény

[85] A diszkrét Fourier-transzformált kiszámítása és továbbítása egy időigényes műveletsor, amelynek elvégzése közben az A/D-átalakító IFA kimeneti FIFO-jának, valamint a FIR-szűrő be- és kimeneti FIFO-jának az ideiglenes buffer szerepe felértékelődik. Ezek szintézis-paramétereken keresztül meghatározható kapacitását a mintavételi időnek és a DSP-modul órajel-frekvenciájának ismeretében körültekintően kell beállítani az adatvesztés elkerülése érdekében.

félperiódusának N/2 db helyettesítési értékére (lásd 4-227. ábra), amelyeket az IP-mag egy belső memóriában, az ún. cosinus-táblában (lásd 4-230. ábra: *COSTABLE*) tárol. Az N/2 db bejegyzést tároló cosinus-tábla feltöltése a felhasználó feladata. Az N elemszám a FIR-szűrő fokszámához hasonlóan szintézis-paraméter.

A FIR-szűrő együtthatóinak beállítását és a cosinus-tábla feltöltését a DSP-bővítőmodulban az MMIO-interfészen keresztül tesszük lehetővé. Szerkezeti szempontból ez azt jelenti, hogy a két IP-mag megfelelő konfigurációs interfészeit az MMIO-adapter állapotgéppel kapcsoljuk össze a 3-9. ábrán látható módon.



3-9. ábra. *A DSP-modul szerkezete (1)*

Az MMIO-adapter állapotgépe persze ezúttal is csak közvetítő szerepben van; a *CPU* felől érkező írás- és olvasáskérésekre reagálva hajt végre műveleteket a bővítőmodul belsejében. Vagyis a FIR és FFT IP-magok üzemeltetésre való előkészítése a *CPU*-n futó program feladata lesz, amihez definiálnunk kell e két almodul konfigurációs interfészeinek szolgáltatásait leképező MMIO-címeket (lásd 3-2. táblázat).

Cím azonosítója és típusa		A címhez rendelt funkció leírása
address_fir_cmem_address_buffer	W	A FIR-szűrő együttható memóriáját címző regiszter címe.
address_fir_dmem_data_buffer_MSB	W	A FIR-szűrő együttható memóriájába beírandó 24-bites érték felső (MSB) és alsó két (LSW) bájtját tároló regiszterek címei.
address_fir_dmem_data_buffer_LSW	W	
address_init_fir_cmem_write	W	Az e címen végrehajtott írás művelettel frissíthető a FIR-szűrő együttható memóriájának egy bejegyzése.
address_init_fir_cbu	W	Az e címen végrehajtott írás művelettel kezdeményezhető a FIR-szűrő belső *cbufs* regisztereinek felülírása az együttható memóriában eltárolt értékekkel *(cbu: coefficient buffers update)*.
address_fft_cstbl_address_buffer	W	Az FFT IP-mag cosinus-tábláját címző regiszter címe.
address_fft_cstbl_data_buffer_MSB	W	Az FFT IP-mag cosinus-táblájába beírandó 24-bites érték felső (MSB) és alsó két (LSW) bájtját tároló regiszterek címei.
address_fft_cstbl_data_buffer_LSW	W	
address_init_fft_cstbl_write	W	Az e címen végrehajtott írás művelettel frissíthető az FFT IP-mag cosinus-táblájának egy bejegyzése.

3-2. táblázat. *A DSP-modul MMIO-címei (1)*
*MSB: **M**ost **S**ignificant **B**yte, LSW: **L**east **S**ignificant **W**ord*

Verilog

```
idle: begin
  s_ready_2_cpu <= 1;
  
  // Handle MMIO write requests...
  if ( write_strobe_from_cpu_df1 == 1 && cs == 1 ) begin
    s_ready_2_cpu <= 0;
    
    if ( [...] ) begin [...]
    end else if ( address_from_cpu == address_fir_cmem_address_buffer ) begin
      fir_cmem_waddr <= data_from_cpu[$clog2(fir_order+1)-1:0]; state_mmio <= wait_for_deassert_strobes;
    end else if ( address_from_cpu == address_fir_dmem_data_buffer_MSB ) begin
      data_2_fir_cmem[23:16] <= data_from_cpu[7:0]; state_mmio <= wait_for_deassert_strobes;
    end else if ( address_from_cpu == address_fir_cmem_data_buffer_LSW ) begin
      data_2_fir_cmem[15:0] <= data_from_cpu; state_mmio <= wait_for_deassert_strobes;
    end else if ( address_from_cpu == address_init_fir_cmem_write ) begin
      fir_cmem_wstrobe <= 1; state_mmio <= write_fir_cmem_1;
    end else if ( address_from_cpu == address_init_fir_cbu ) begin
      fir_cbu_request <= 1; state_mmio <= fir_cbu_1;
    end else if ( address_from_cpu == address_fft_cstbl_address_buffer ) begin
      fft_cstbl_waddr <= data_from_cpu[$clog2(fft_N/2)-1:0]; state_mmio <= wait_for_deassert_strobes;
    end else if ( address_from_cpu == address_fft_cstbl_data_buffer_MSB ) begin
      data_2_fft_cstbl[23:16] <= data_from_cpu[7:0]; state_mmio <= wait_for_deassert_strobes;
    end else if ( address_from_cpu == address_fft_cstbl_data_buffer_LSW ) begin
      data_2_fft_cstbl[15:0] <= data_from_cpu; state_mmio <= wait_for_deassert_strobes;
    end else if ( address_from_cpu == address_init_fft_cstbl_write ) begin
      fft_cstbl_wstrobe <= 1; state_mmio <= write_fft_cstbl_1;
    end else if ( [...] ) begin [...]
    end else state_mmio <= wait_for_deassert_strobes;
    
  end
  
  // Handle MMIO read requests...
  if ( read_strobe_from_cpu_df1 == 1 && cs == 1 ) begin s_ready_2_cpu <= 0; [...] end
  
end

write_fir_cmem_1: state_mmio <= write_fir_cmem_2;

write_fir_cmem_2: if ( fir_cmem_rdy2wr == 1 ) begin
                    fir_cmem_wstrobe <= 0; state_mmio <= wait_for_deassert_strobes;
                  end

fir_cbu_1: state_mmio <= fir_cbu_2;

fir_cbu_2: if ( fir_cbu_done == 1 ) begin
             fir_cbu_request <= 0; state_mmio <= wait_for_deassert_strobes;
           end

write_fft_cstbl_1: state_mmio <= write_fft_cstbl_2;

write_fft_cstbl_2: if ( fft_cstbl_rdy2wr == 1 ) begin
                     fft_cstbl_wstrobe <= 0; state_mmio <= wait_for_deassert_strobes;
                   end

wait_for_deassert_strobes: begin
  [...]
  if ( read_strobe_from_cpu_df1 == 0 && write_strobe_from_cpu_df1 == 0 ) state_mmio <= idle;
end
```

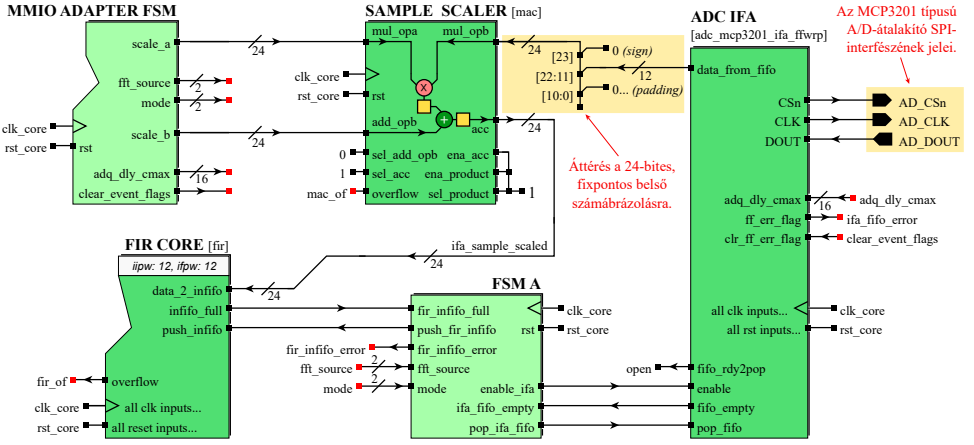
3-10. ábra. *A DSP-modul MMIO-adapter állapotgépének vezérlési állapotai (1)*

A *CPU*-n futó programból a FIR-szűrő együttható memóriájának bejegyzései csak egyesével frissíthetők a következőképpen: az *address_fir_cmem_address_buffer* címen végrehajtott írás művelettel eltároljuk a beírandó érték együttható memóriabeli címét, majd az *address_fir_dmem_data_buffer_MSB/LSW* címeken végrehajtott újabb két írás művelettel a 24-bites együttható felső és két alsó bájtjának rögzítését is elvégezzük. Az együtthatót és annak együttható memóriabeli címét tehát az MMIO-adapter állapotgép egy-egy kimeneti regisztere ideiglenesen tárolja. Maga a FIR-szűrőbe való beírás az *address_init_fir_cmem_write* címen végrehajtott írás művelettel kezdeményezhető.

E négy MMIO-írásból álló műveletsort minden együtthatóra el kell végezni. Ha minden együtthatót eltároltunk a FIR-szűrő együttható memóriájában, akkor a szűrő belső *cbufs* regisztereinek felülírása az együttható memóriába betöltött értékekkel az *address_init_fir_cbu* címen végrehajtott MMIO-írással kezdeményezhető (vö. 4.5.6.2. pont: A szűrőegyütthatók újrakonfigurálása). Ezzel a FIR-szűrő együtthatói frissültek, az almodul készen áll az üzemeltetésre.

Az FFT IP-mag cosinus-táblájának bejegyzései ugyanezzel a módszerrel inicializálhatók. A cosinus-táblához tartozó címregiszter az *address_fft_cstbl_address_buffer*, az adatregiszter pedig az *address_fft_cstbl_data_buffer_MSB/LSW* címeken keresztül érhető el. A bejegyzés beírása az *address_init_fft_cstbl_write* címen végrehajtott írással kezdeményezhető. Az FFT IP-mag esetén nincs szükség a FIR-szűrőre jellemző *cbufs* regiszterek frissítéséhez hasonló műveletre, mivel az FFT IP-mag belső logikája a cosinus-táblát a transzformáció végrehajtása közben közvetlenül képes olvasni (vö. 4.5.5.2. pont: Cosinus-tábla). A DSP-modul MMIO-adapter állapotgépének a 3-2. táblázatban látható címek kezeléséért felelős részleteit a 3-10. ábra szemlélteti.

A 3-11. ábra a DSP-modulnak az A/D-átalakító és a FIR-szűrő közötti előfeldolgozó almoduljait, azok összeköttetéseit és MMIO-interfészen keresztül konfigurálható paramétereit szemlélteti.



3-11. ábra. *A DSP-modul szerkezete (2)*

Ahogy korábban említettük, az A/D-átalakító IFA kimeneti FIFO-ja és a FIR-szűrő bemeneti FIFO-ja közötti adatátvitelt az *FSM A* állapotgép vezényli le. Az IFA kimeneti FIFO-jából kiolvasott 12-bites bináris vektorokat a legfelső helyiértéken az előjelbitet képviselő nullával, az alsó 11 helyiértéken pedig a későbbi számítások pontosságát biztosító (padding) nullákkal egészítjük ki, így állítva elő a 3-8. ábrán bemutatott belső számábrázolásnak megfelelő formátumú mintákat. A FIR-szűrő bemeneti FIFO-jába való betöltés előtt e mintákat skálázzuk (3-11. ábra: *SAMPLE SCALER*), ami egy szorzás és egy összeadás művelet elvégzését jelenti az MMIO-adapter állapotgép által szolgáltatott *scale_a* és *scale_b* együtthatókkal. A skálázás kivitelezéséhez a FIR-szűrő 4.5.6.1. pontban bemutatott műveletvégző egységéhez tervezett újrafelhasználható szorzó-akkumulátor (**M**ultiply & **Ac**cumulate, MAC) modul egy példányát használjuk (lásd 4-246. ábra, vö. 4-248. ábra). A 3-11. ábrán az is látható, hogy az *FSM A* állapotgépnek feladata végrehajtásához szüksége van az – ugyancsak az MMIO-adapter állapotgép által meghajtott – *fft_source és mode* vezérlőjelek

értékére, amelyek pontos értelmezésére később visszatérünk. A 3-11. ábrán látható áramköri részletek konfigurálásához szükséges MMIO-címeket a 3-3. táblázat foglalja össze.

Cím azonosítója és típusa		A címhez rendelt funkció leírása
address_sampling_time	W	Az A/D-átalakító interfészadapterének *ad_query_delay_cnt_max* bemenetét meghajtó vezérlőregiszter címe, ami meghatározza a rendszer mintavételi frekvenciáját (vö. 4-151. ábra).
address_scale_a_MSB	W	Az A/D-átalakító interfészadapterétől érkező minták lineáris skálázásának *A* együtthatóját tároló vezérlőregiszter felső (MSB) és alsó két (LSW) bájtjának címei.
address_scale_a_LSW	W	
address_scale_b_MSB	W	Az A/D-átalakító interfészadapterétől érkező minták lineáris skálázásának *B* együtthatóját tároló vezérlőregiszter felső (MSB) és alsó két (LSW) bájtjának címei.
address_scale_b_LSW	W	
address_config	W	A DSP-modul üzemmódját, valamint FFT-üzemmódban a transzformálandó vektor forrását és a transzformált vektor célhelyét meghatározó vezérlőregiszter címe: [1:0]: *mode* (üzemmód-kiválasztó regiszter) • 00_2: clean-up üzemmód (lásd később) • 01_2: mintafolyam üzemmód (sample stream) • 10_2: FFT-üzemmód [3:2]: *fft_source* (a transzformálandó vektor forrása) • 00_2: FIR-szűrő kimeneti FIFO-ja • 01_2: az útválasztóhoz kapcsolt bemeneti FIFO • 10_2: külső memória [4]: *fft_sink* (a transzformált vektor célhelye) • 0: az útválasztóhoz kapcsolt kimeneti FIFO • 1: külső memória

3-3. táblázat. *A DSP-modul MMIO-címei (2)*
*MSB: **M**ost **S**ignificant **B**yte, LSW: **L**east **S**ignificant **W**ord*

Az A/D-átalakító interfészadapterének *ad_query_delay_cnt_max* bemenete 16-bites, vagyis egyetlen – az *address_sampling_time* címen végrehajtott – MMIO írásművelettel beállítható. A skálázás együtthatói azonban – az előírt számábrázolási formátumnak megfelelően – 24-bitesek, vagyis az MMIO-adapter állapotgépnek az ezeket az együtthatókat képviselő *scale_a* és *scale_b* kimeneteit csak két-két MMIO-írással tudjuk beállítani *(address_scale_A/B_MSB/LSW)*.

A DSP-modul mintafolyam és FFT-üzemmódját már a rendszertervezés szintjén megneveztük, de a 3-3. táblázatban látható, hogy az RTL-tervezés során egy további üzemmódot, az ún. clean-up üzemmódot is bevezetjük. Ennek az üzemmódnak a célja, hogy teret adjon a felhasználónak a modul újrakonfigurálásához; clean-up üzemmódban az A/D-átalakító interfészadapterét, a FIR-szűrőt és az FFT IP-magot is kikapcsoljuk, a FIFO-kat pedig kiürítjük az esetleg még bennük lévő adattöredékek eldobásával.

Mivel az üzemmód-kiválasztó *mode* regiszter, valamint az FFT-üzemmódban érvényes, a transzformálandó vektor forrását és a transzformált vektor célhelyét meghatározó *fft_source* és *fft_sink* regiszterek együttesen csupán 5 bit információt hordoznak, ezért ezek beállítását egyetlen közös MMIO-címen tesszük lehetővé *(address_config)*. Az MMIO-adapter állapotgépnek az e vezérlőregiszterek beállítását szolgáló részleteit a 3-12. ábra mutatja be.

Az *FSM B* állapotgép központi szerepét jól érzékelteti a 3-13. ábra. Itt futnak össze a FIR-szűrő kimeneti FIFO-jának, az FFT modul be- és kimeneti FIFO-jának, valamint az SPI-FIFO útválasztóval való kommunikációt lehetővé tévő FIFO-knak, továbbá a memória-arbiterrel való

Verilog

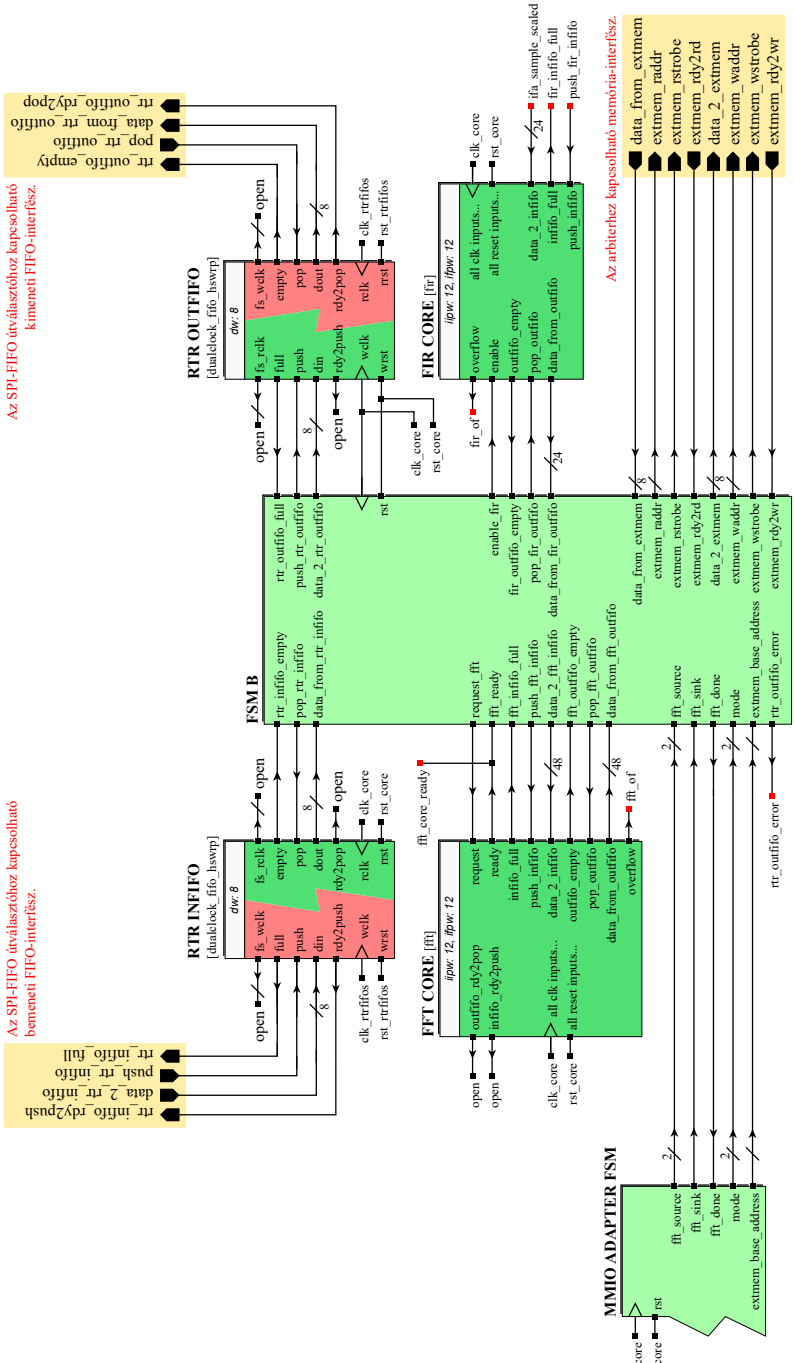
```
idle: begin
  s_ready_2_cpu <= 1;

  // Handle MMIO write requests...
  if ( write_strobe_from_cpu_df1 == 1 && cs == 1 ) begin
    s_ready_2_cpu <= 0;

    if ( address_from_cpu == address_sampling_time ) begin
      ad_query_delay_cnt_max <= data_from_cpu; state_mmio <= wait_for_deassert_strobes;
    end else if ( address_from_cpu == address_scale_a_MSB ) begin
      scale_a[23:16] <= data_from_cpu[7:0]; state_mmio <= wait_for_deassert_strobes;
    end else if ( address_from_cpu == address_scale_a_LSW ) begin
      scale_a[15:0] <= data_from_cpu; state_mmio <= wait_for_deassert_strobes;
    end else if ( address_from_cpu == address_scale_b_MSB ) begin
      scale_b[23:16] <= data_from_cpu[7:0]; state_mmio <= wait_for_deassert_strobes;
    end else if ( address_from_cpu == address_scale_b_LSW ) begin
      scale_b[15:0] <= data_from_cpu; state_mmio <= wait_for_deassert_strobes;
    end else if ( address_from_cpu == address_config ) begin
      mode <= data_from_cpu[1:0];              // 00: clean-up,       01: sample stream, 10: FFT
      fft_source <= data_from_cpu[3:2];        // 00: FIR outfifo,    01: router infifo, 10: ext. memory
      fft_sink <= data_from_cpu[4];            //  0: router outfifo,  1: ext. memory
      state_mmio <= wait_for_deassert_strobes;
    end else if ( [...] ) begin [...]
    end else state_mmio <= wait_for_deassert_strobes;

  end

  // Handle MMIO read requests...
  if ( read_strobe_from_cpu_df1 == 1 && cs == 1 ) begin s_ready_2_cpu <= 0; [...] end

end

[...]

wait_for_deassert_strobes: begin
  [...]
  if ( read_strobe_from_cpu_df1 == 0 && write_strobe_from_cpu_df1 == 0 ) state_mmio <= idle;
end
```

3-12. ábra. *A DSP-modul MMIO-adapter állapotgépének vezérlési állapotai (2)*

kompatibilitást – közvetett módon tehát a külső operatív tárhoz való hozzáférést – biztosító interfésznek a jelei is.

Az *FSM B* állapotgép természetesen az MMIO-adapter állapotgéppel is összeköttetésben áll. Feladata végrehajtásához szüksége van az *address_config* címen keresztül beállítható *mode*, *fft_source* és *fft_sink* vezérlőregiszterek, illetve egy további, a külső memóriával végzett műveletek során érvényesülő *extmem_base_address* vezérlőregiszter értékére. A követelménylista megfogalmazza, hogy FFT-üzemmódban a külső memória egyaránt kijelölhető a transzformálandó vektor forrásaként és/vagy a transzformált vektor célhelyeként. Az MMIO-adapter állapotgép *extmem_base_address* vezérlőkimenete írja elő az *FSM B* állapotgép számára, hogy a külső memóriához forduláskor honnan kell kezdenie az *N*-elemű vektorok beolvasását és/vagy kiírását.

A DSP-modul bájtos hozzáférésű külső memóriát feltételez, ezért az N db 24-bites komplex számból álló transzformálandó és transzformált vektorok N × 6 bejegyzést foglalnak el a külső tárban. A külső memória címbuszának mérete egy 32 bitben maximalizált szintézis-paraméter, ezért a báziscím rögzítéséhez két MMIO-írásra van szükség. Az ezekhez tartozó MMIO-címeket a 3-4. táblázat, az MMIO-állapotgép e két címen elérhető *extmem_base_address* vezérlőregiszterének beállításáért felelős részleteit pedig a 3-14. ábra mutatja be.



3-13. ábra. *A DSP-modul szerkezete (3)*

Cím azonosítója és típusa		A címhez rendelt funkció leírása
address_extmem_base_address_MSW	W	A külső memóriában lefoglalt tartomány báziscímét tároló regiszter felső (MSW) és alsó két (LSW) bájtjának címei.
address_extmem_base_address_LSW	W	

3-4. táblázat. *A DSP-modul MMIO-címei (3)*
*MSW: **M**ost **S**ignificant **W**ord, LSW: **L**east **S**ignificant **W**ord*

Verilog

```
idle: begin
  s_ready_2_cpu <= 1;
  
  // Handle MMIO write requests...
  if ( write_strobe_from_cpu_df1 == 1 && cs == 1 ) begin
    s_ready_2_cpu <= 0;
    
    if ( [...] ) begin [...]
    end else if ( address_from_cpu == address_extmem_base_address_MSW ) begin
      extmem_base_address[31:16] <= data_from_cpu; state_mmio <= wait_for_deassert_strobes;
    end else if ( address_from_cpu == address_extmem_base_address_LSW ) begin
      extmem_base_address[15:0] <= data_from_cpu; state_mmio <= wait_for_deassert_strobes;
    end else if ( [...] ) begin [...]
    end else state_mmio <= wait_for_deassert_strobes;
    
  end
  
  // Handle MMIO read requests...
  if ( read_strobe_from_cpu_df1 == 1 && cs == 1 ) begin s_ready_2_cpu <= 0; [...] end

end

[...]

wait_for_deassert_strobes: begin
  [...]
  if ( read_strobe_from_cpu_df1 == 0 && write_strobe_from_cpu_df1 == 0 ) state_mmio <= idle;
end
```

3-14. ábra. *A DSP-modul MMIO-adapter állapotgépének vezérlési állapotai (3)*

A DSP-modul belső részegységei közötti összeköttetések áttekintése után nekivághatunk a különböző üzemmódokban elvárt pontos viselkedést meghatározó *FSM A* és *FSM B* állapotgépek részletes tervezésének. A korábban bemutatott, összetettebb IP-magoknál – pl. *BOOTSTRAP* (2.3. pont), SPI-FIFO útválasztó (2.8.3. pont) stb. – már megfigyelhettük azt a jellemző mintázatot, hogy a komplex toplevel modulok lényegében újrafelhasználható IP-magok és az azok működését összehangoló, az adott alkalmazáshoz kifejlesztett, egyedi állapotgépek összekapcsolt hálózataként épülnek fel. Az egyedi állapotgépek tervezésekor természetesen kulcsfontosságú az újrafelhasználható IP-magok interfészeinek pontos ismerete. A DSP-modul esetén az A/D-átalakító IFA, a FIR-szűrő és az FFT modul olyan újrafelhasználható IP-magok, amelyek FIFO-kon keresztül kommunikálnak a külvilággal. Részletes RTL-modelljeiket megvizsgálva láthatjuk, hogy e FIFO-interfészek mögött az IP belsejében a 4.2.5. pontban tárgyalt *dualclock_fifo* almodul (FIR), vagy annak kézfogásos variánsa, a *dualclock_fifo_hswrp* almodul (IFA és FFT) rejtőzik,[86] az SPI-FIFO útválasztóval való összekapcsoláshoz szükséges FIFO-tárolók pedig ugyancsak a *dualclock_fifo_hswrp* IP-mag példányai. A szinkron kézfogásos interfésszel ellátott FIFO modulok sajátossága, hogy képesek modellezni a több órajelciklus késleltetésű memóriák viselkedését, lehetővé téve e késleltetés hatásának vizsgálatát a beágyazó rendszer számítási kapacitására (lásd még: 4.2.3. pont). A DSP-bővítőmodulban található FIFO-knak ezt a képességét szándékosan nem

[86] A/D-átalakító IFA: 4-151. ábra, FIR-szűrő: 4-249. ábra, FFT modul: 4-230. ábra.

használjuk ki; a példányosított, szinkron kézfogásos interfészű FIFO-kat nulla késleltetésűnek – vagyis az 1-3. ábra jobb oldalán látható időzítési diagramoknak megfelelő működésűnek – feltételezve tervezzük meg a velük kölcsönhatásba lépő állapotgépeket.[87]

A 3-15. ábra az A/D-átalakító kimeneti FIFO-ja és a FIR-szűrő bemeneti FIFO-ja közötti adatmozgatásért felelős *FSM A* állapotgép Verilog-leírásának vázlatát, a 3-16. ábra pedig vezérlési állapotainak részleteit szemlélteti.

Verilog

```
1  // FSM A
2  always @ ( posedge clk_core, posedge rst_core ) begin
3    if ( rst_core == 1 ) begin
4      state_a <= check_mode_a;
5      push_fir_infifo <= 0; enable_ifa <= 0; pop_ifa_fifo <= 0; fir_infifo_error <= 0;
6    end else begin
7    |
8    |  // default values of the pulse-like control signals...
9    |  push_fir_infifo <= 0; pop_ifa_fifo <= 0; fir_infifo_error <= 0;
10   |
11   |  case ( state_a )
12   |  |
13   |  |  check_mode_a: [...]    // 00: clean-up:      pop A/D IFA FIFO
14   |  |                         // 01: sample stream: move data
15   |  |  [...]                  // 10: FFT:           if fft_source is FIR FIFO -> move data, else clean-up
16   |  |
17   |  |  default: ;
18   |  |
19   |  endcase
20   end
21 end
```

3-15. ábra. *Az A/D-átalakító interfészadaptere és a FIR-szűrő közötti adatátvitelt vezérlő állapotgép (FSM A) Verilog-leírásának vázlata*

Az *FSM A* állapotgép fő feladata az A/D-átalakító IFA és a FIR-szűrő közötti adatátvitel, amit a *move_data_1...4* állapotok valósítanak meg. Erre csak akkor kerülhet sor, ha a DSP-modul mintafolyam üzemmódban van ($mode = 01_2$, 3-16. ábra: 2. sor), vagy ha FFT-üzemmódban az *fft_source* vezérlőbemenet a FIR-szűrő kimeneti FIFO-ját jelöli ki a diszkrét Fourier-transzformáció bemenetének forrásaként ($fft_source = 00_2$, 3-16. ábra: 4. sor).[88] A vezérlőbemenetek bármely ettől eltérő kombinációja esetén az *FSM A* állapotgép clean-up üzemmódban kell hogy működjön, amelyben egyrészt letiltja az A/D-átalakító IFA működését (3-16. ábra: 9. sor), másrészt kiolvassa (és elveti) az annak kimeneti FIFO-jában esetleg bent maradt bejegyzéseket (3-16. ábra: 10. sor).

Ahhoz, hogy az IFA elkezdjen adatokat lekérdezni az A/D-átalakítótól, természetesen engedélyezni kell a működését (3-16. ábra: 15. sor), ezután pedig nincs más feladat, mint folyamatosan ellenőrizni kimeneti FIFO-jának *empty* jelzését (3-16. ábra: 16. sor). Az *empty* kimenet alacsony szintje továbbítandó bejegyzés jelenlétére utal, ilyenkor végrehajtunk egy *pop* műveletet az IFA kimeneti FIFO-ján, majd két órajelciklusnyi késleltetés után *push* műveletet kezdeményezünk a FIR-szűrő bemeneti FIFO-ján (3-16. ábra: 26. sor). Figyeljük meg, hogy az állapotgép közvetlenül nem olvassa az IFA FIFO-jának adatkimenetét és nem írja a FIR-szűrő FIFO-jának adatbemenetét. A két FIFO közötti adatkapcsolat ui. az állapotgépen kívül, ráadásul csak közvetetten, a 12-bites A/D minták fixpontossá alakításán és a *SAMPLE SCALER* szorzó-akkumulátor által végzett skálázáson keresztül jön létre (lásd 3-11. ábra). Pontosan e kapcsolat közvetett jellege

[87] A megoldás nyilvánvaló hátránya, hogy így később nem térhetünk át több órajelciklus késleltetésű FIFO modulokra a DSP-modul állapotgépeinek módosítása nélkül, előnye viszont, hogy ezek az állapotgépek így egyszerűbbek lesznek, kevesebb vezérlési állapotot tartalmaznak, ami által a műveletvégzés is gyorsabb lesz.

[88] A FIR-szűrő kimeneti FIFO-jában csak az *FSM A* állapotgép közreműködésével jelenhet meg feldolgozható mintasor.

Verilog

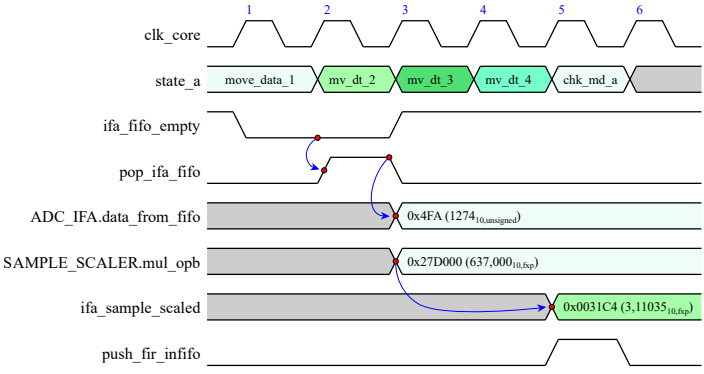
```
1  check_mode_a:
2    if ( mode == 2'b01 ) state_a <= move_data_1;
3    else if (
4      mode == 2'b10 && fft_source == 2'b00
5    ) state_a <= move_data_1;
6    else state_a <= cleanup_ifa_fifo;
7
8  cleanup_ifa_fifo: begin
9    enable_ifa <= 0;
10   if ( ifa_fifo_empty == 0 ) pop_ifa_fifo <= 1;
11   else state_a <= check_mode_a;
12 end
13
```

Verilog (folyt.)

```
14 move_data_1: begin
15   enable_ifa <= 1;
16   if ( ifa_fifo_empty == 0 ) begin
17     pop_ifa_fifo <= 1;
18     state_a <= move_data_2;
19   end
20 end
21
22 move_data_2: state_a <= move_data_3;
23 move_data_3: state_a <= move_data_4;
24
25 move_data_4: begin
26   if ( fir_infifo_full == 0 ) push_fir_infifo <= 1;
27   else fir_infifo_error <= 1;
28   state_a <= check_mode_a;
29 end
```

3-16. ábra. *Az A/D-átalakító interfészadaptere és a FIR-szűrő közötti adatátvitelt vezérlő állapotgép (FSM A) vezérlési állapotai*

az, ami miatt a *pop* és *push* műveletek között némi várakozás beiktatására van szükség; a skálázást megvalósító szorzó-akkumulátor egy szinkron működésű, két órajelciklus késleltetésű részegység. A bejegyzések átmozgatásának pontos időzítése a 3-17. ábrán követhető nyomon.



3-17. ábra. *Egy bejegyzés átmozgatása az A/D-átalakító IFA kimeneti FIFO-ja és a FIR-szűrő bemeneti FIFO-ja között (scale_a = 0,00488281, scale_b = 0,0)*

A FIR-szűrő bemeneti FIFO-jába természetesen csak akkor tudunk új bejegyzést írni, ha az nincs tele, és ezt az ellenőrzést az *FSM A* állapotgép el is végzi (3-16. ábra: 26. sor). Elvileg nem lép fel adatvesztés, ha a *push* műveletet még a következő minta megérkezése előtt be tudjuk írni a FIR-szűrő bemeneti FIFO-jába, de az egyszerűség kedvéért e FIFO *full* jelzésének detektálásakor azonnal „farkast kiáltunk”; a *fir_infifo_error* vezérlőjelen egy pozitív impulzust állítunk elő (3-16. ábra: 27. sor). Ezt a jelzést a DSP-modul megszakításkérő logikája dolgozza majd fel (lásd később). Figyeljük meg, hogy a *clean-up* állapot és az adatmozgató állapotsorozat is mindig a *check_mode_a* állapotba irányítja vissza a végrehajtást. Erre azért van szükség, mert ez az egyetlen állapot, ahol az állapotgép a vezérlőbemenetek (*mode* és *fft_source*) változásaira érzékeny, azokra reagálni tud (lásd még: 3.1. pont: hullámforma-generátor *SAMPLE GEN FSM* állapotgépe).

A 3-18. ábra az *FSM B* állapotgép Verilog-leírásának vázlatát mutatja be.

Verilog

```
1  // FSM B
2  always @ ( posedge clk_core, posedge rst_core ) begin
3  | if ( rst_core == 1 ) begin [...] -- reset all outputs and internal registers...
4  | end else begin
5  | |
6  | | // default values of the pulse-like control signals...
7  | | pop_rtr_infifo <= 0; push_rtr_outfifo <= 0; pop_fir_outfifo <= 0; request_fft <= 0;
8  | | push_fft_infifo <= 0; pop_fft_outfifo <= 0;
9  | | rtr_outfifo_error <= 0; extmem_wstrobe <= 0; extmem_rstrobe <= 0;
10 | | |
11 | | case ( state_b )
12 | | | |
13 | | | check_mode_b: if ( mode == 2'b01 ) state_b <= stream_2_router_1;      // FIR -> ROUTER
14 | | |               else if ( mode == 2'b10 ) begin                        // FFT
15 | | |               | extmem_raddr <= extmem_base_address[extmem_aw-1:0];
16 | | |               | extmem_waddr <= extmem_base_address[extmem_aw-1:0];
17 | | |               | state_b <= fft_1;
18 | | |               end else state_b <= cleanup;                           // CLEAN-UP
19 | | |
20 | | | cleanup: begin
21 | | | | enable_fir <= 0;
22 | | | | if ( fir_outfifo_empty == 0 || rtr_infifo_empty == 0 || fft_outfifo_empty == 0 ) begin
23 | | | |   pop_fir_outfifo <= 1; pop_rtr_infifo <= 1; pop_fft_outfifo <= 1;
24 | | | | end else state_b <= check_mode_b;
25 | | | end
26 | | | |
27 | | | [...]
28 | | | |
29 | | | default: ;
30 | | | |
31 | | endcase
32 | | |
33 | end
34 end
```

3-18. ábra. *A központi vezérlő állapotgép (FSM B) Verilog-leírásának vázlata*

Az *FSM A* állapotgépnél látottakhoz hasonlóan itt is a *mode* vezérlőjel határozza meg az alapvető viselkedést. Ez alapján az *FSM B* állapotgépnek három működési módja van. A *mode* vezérlőjel 01_2 értéke által képviselt mintafolyam üzemmódban feladata az, hogy a FIR-szűrő kimeneti FIFO-jának tartalmát az útválasztóhoz kapcsolt kimeneti FIFO-ba továbbítsa (*stream_2_router_1...3* állapotsorozat). Ha a *mode* vezérlőbemenet 10_2 értékű, akkor a DSP-modul FFT-üzemmódban van, vagyis az *fft_source* vezérlőbemenet értéke által meghatározott forrásból fel kell töltenie az FFT IP-mag bemeneti FIFO-ját, kezdeményeznie kell a transzformáció elvégzését, majd annak befejeződésekor a transzformált vektort az FFT modul kimeneti FIFO-jából az *fft_sink* vezérlőjel által kijelölt helyre kell továbbítania (*fft_1...5* állapotsorozat). A *mode* vezérlőjel 00_2 értéke esetén az *FSM B* állapotgép az *FSM A*-hoz hasonlóan egy „tisztogató” funkcióba lép, amely során a FIR-szűrő kimeneti FIFO-ját, az FFT modul kimeneti FIFO-ját és az útválasztóhoz kapcsolt bemeneti FIFO-tárolót is kiüríti, az itt talált adattöredékeket eldobja és mindezt addig folytatja, amíg a *mode* vezérlőbemeneten valami változás nem következik be (*cleanup* állapot, 3-18. ábra: 20-25. sor).

A *stream_2_router_1...3* állapotsorozat részleteit a 3-19. ábra szemlélteti. Bár a feladat hasonlít az *FSM A* állapotgépnél látottakhoz, hiszen ezúttal is két, nulla késleltetésű, szinkron kézfogásos interfésszel ellátott FIFO között kell adatmozgatást végezni, a *stream_2_router_3* állapot leírása alapján mégis nyilvánvaló, hogy a helyzet ezúttal egy árnyalatnyival bonyolultabb.

A különbség az *FSM A* állapotgéphez képest az, hogy míg abban az esetben a forrás- és cél-FIFO-k adatbuszai azonos méretűek voltak, addig az *FSM B* állapotgép a mintafolyam üzemmódban a FIR-szűrő 24-bites kimeneti FIFO-jából továbbítja az adatokat a 8-bites *ROUTER OUTPUT FIFO*-ba. Tehát a FIR-szűrőből kiolvasott egyetlen bejegyzés átmozgatásához három *push*

Verilog

```
1  stream_2_router_1: begin
2                     | enable_fir <= 1;
3                     | if ( fir_outfifo_empty == 0 ) begin
4                     |   pop_fir_outfifo <= 1; state_b <= stream_2_router_2;
5                     | end
6                     end
7
8  stream_2_router_2: state_b <= stream_2_router_3;
9
10 stream_2_router_3: begin
11                    |
12                    | case ( byte_cnt )
13                    | | 0:       data_2_rtr_outfifo <= data_from_fir_outfifo[23:16];   // MSB first
14                    | | 1:       data_2_rtr_outfifo <= data_from_fir_outfifo[15:8];
15                    | | default: data_2_rtr_outfifo <= data_from_fir_outfifo[7:0];
16                    | endcase
17                    |
18                    | if ( rtr_outfifo_full == 0 ) begin
19                    | | push_rtr_outfifo <= 1;
20                    | |
21                    | | if ( byte_cnt == 2 ) begin
22                    | |   byte_cnt <= 0; state_b <= check_mode_b;
23                    | | end else byte_cnt <= byte_cnt + 1;
24                    | |
25                    | end else rtr_outfifo_error <= 1;
26                    |
27                    end
```

3-19. ábra. *A központi vezérlő állapotgép (FSM B) mintafolyam (sample stream) üzemmódban érvényesülő vezérlési állapotai*

műveletet kell elvégeznünk az útválasztóhoz kapcsolt FIFO-n. Az *FSM B* állapotgépben az egyazon fixpontos minta bájtonkénti átvitelének nyomon követésére bevezetjük a *byte_cnt* regisztert. Az *FSM A* állapotgép leírásához képest egy további különbség, hogy az *FSM B* állapotgép a FIFO-k adatbuszait is közvetlenül kénytelen írni és olvasni, hiszen a *byte_cnt* számláló értékétől függően a *data_from_fir_outfifo* busz – ami a FIR-szűrő kimeneti FIFO-jának adatbusza – különböző mezőit kell a *data_2_rtr_outfifo* buszra, vagyis az útválasztóhoz kapcsolt kimeneti FIFO adatbemenetére kapcsolnia (lásd 3-19. ábra: 12–16. sor). Az adatvesztés veszélyének jelzésére az *FSM A* állapotgépnél bevezetett megoldást alkalmazzuk; ha az *FSM B* állapotgép az útválasztóhoz kapcsolt kimeneti FIFO *full* kimenetén magas szintet tapasztal (3-19. ábra: 25. sor), miközben adatot próbál beleírni, akkor az *rtr_outfifo_error* állapotjelzőn egy pozitív impulzust állít elő a hibadetektáló és megszakításkérő logika számára (lásd később).

Az *FSM B* állapotgép *fft_1...5* állapotsorozata az FFT-üzemmód vezénylését végzi. Ebben az üzemmódban az állapotgép pontos viselkedése változatosan alakulhat, mivel az elvégzendő diszkrét Fourier-transzformáció(k) bemeneti és kimeneti vektorának helye rendre az *fft_source* és *fft_sink* vezérlőjelek függvénye. A HDL-leírás áttekinthetősége érdekében a transzformálandó vektornak a különböző forrásokból az FFT IP-mag bemeneti FIFO-jába, illetve a transzformált vektornak az FFT IP-mag kimeneti FIFO-jából a különböző célhelyekre való átmozgatására egy-egy szubrutinszerű állapotsorozatot definiálunk:[89]

- Az FFT modul bemeneti FIFO-jának feltöltése a FIR-szűrő kimeneti FIFO-jából kiolvasott mintákkal: 3-20. ábra.

[89] A szubrutinszerű állapotsorozatok fogalmának részletes magyarázatát lásd 2.3. pont: *BOOTSTRAP* modul központi vezérlő állapotgépe.

- Az FFT modul bemeneti FIFO-jának feltöltése az útválasztóhoz kapcsolt bemeneti FIFO-ból (az üzemeltető rendszertől érkező mintákkal): 3-21. ábra.
- Az FFT modul bemeneti FIFO-jának feltöltése az *EXTMEM* interfészen – és a memória-arbiteren – keresztül a külső operatív tárból: 3-22. ábra.
- A transzformált vektor átmozgatása az FFT modul kimeneti FIFO-jából az útválasztóhoz kapcsolt kimeneti FIFO-ba (az üzemeltető rendszer felé): 3-23. ábra.
- A transzformált vektor átmozgatása az FFT modul kimeneti FIFO-jából az *EXTMEM* interfészen – és a memória-arbiteren – keresztül a külső operatív tárba: 3-24. ábra.

Verilog

```
1  feed_fft_from_fir_1: begin
2                         enable_fir <= 1;
3                         if ( fft_infifo_full == 1 ) state_b <= return_state_b;
4                         else if ( fir_outfifo_empty == 0 ) begin
5                           pop_fir_outfifo <= 1; state_b <= feed_fft_from_fir_2;
6                         end
7                       end
8
9  feed_fft_from_fir_2: state_b <= feed_fft_from_fir_3;
10
11 feed_fft_from_fir_3: begin
12                         data_2_fft_infifo <= { data_from_fir_outfifo , 24'h000000 };
13                         push_fft_infifo <= 1; state_b <= feed_fft_from_fir_4;
14                      end
15
16 feed_fft_from_fir_4: state_b <= feed_fft_from_fir_1;
```

3-20. ábra. *A központi vezérlő állapotgép (FSM B) FFT-üzemmódhoz tartozó szubrutinszerű állapotsorozatai (1)*

Az FFT modul által végzett transzformáció fokszáma egy szintézis-paraméter, ami egyúttal az FFT IP-mag be- és kimeneti FIFO-inak kapacitását is definiálja. Ez azt jelenti, hogy bármelyik adatmozgató állapotsorozatról legyen is szó, a periodikus, N db minta átmozgatását igénylő folyamatok végének megállapításához szükségtelen külön számlálót létrehoznunk, használhatjuk az FFT modul FIFO-inak állapotjelzőit. N db bejegyzés befogadása után a bemeneti FIFO *full* jelzése magas lesz, a kimeneti FIFO *empty* jelzése pedig ugyancsak N db kiolvasás után válik magassá. Ennek megfelelően a szubrutinszerű állapotsorozatok mindegyikére igaz, hogy a visszatérési állapotokba a megfelelő FIFO állapotjelző bebillenésekor lépnek vissza (3-20. ábra: 3. sor, 3–21. - 3-24. ábra: 1. sor).

Az FFT modul komplex számokon végez műveleteket, ezért be- és kimeneti FIFO-inak adatbusza 48-bites. A felső 24 bit képviseli az adott minta valós, az alsó 24 pedig annak képzetes részét. Így egyetlen *push* vagy *pop* művelet egy teljes értékű komplex szám mozgatását jelenti. Amikor a FIR-szűrő kimeneti FIFO-jából – közvetetten tehát az A/D-átalakítóból – érkező mintasoron szeretnénk diszkrét Fourier-transzformációt végezni, akkor e mintákat valós értékeknek tekintjük, azokat az FFT bemeneti FIFO adatbuszának felső felébe töltjük, miközben ugyanannak az adatbusznak a minta képzetes részét képviselő alsó felét nullákkal töltjük ki (3-20. ábra: 12. sor). Így lehetséges, hogy minden, a FIR-szűrőből kiolvasott 24-bites minta egyetlen *push* művelettel beíródhat az FFT modulba.

A többi esetben erre az egyszerűsítésre nincs lehetőségünk; ha a transzformálandó vektor az útválasztóhoz kapcsolt bemeneti FIFO-ból, vagyis az SPI-FIFO útválasztón keresztül érkezik (lásd 3-21. ábra), akkor nem feltételezhetjük, hogy valós mintasorról van szó, az üzemeltető rendszer nyugodtan küldhet nemnulla képzetes részű komplex számokat feldolgozásra. Mivel az útválasztóhoz kapcsolt FIFO-k adatbusza – magának az útválasztónak a sajátosságai miatt – 8-bites, ezért egyetlen

Verilog

```
1  feed_fft_from_rtr_1: if ( fft_infifo_full == 1 ) state_b <= return_state_b;
2                       else if ( rtr_infifo_empty == 0 ) begin
3                         pop_rtr_infifo <= 1; state_b <= feed_fft_from_rtr_2;
4                       end
5
6  feed_fft_from_rtr_2: state_b <= feed_fft_from_rtr_3;
7
8  feed_fft_from_rtr_3: begin
9
10                        case ( byte_cnt )
11                          0:       data_2_fft_infifo[47:40] <= data_from_rtr_infifo;   // MSB first
12                          1:       data_2_fft_infifo[39:32] <= data_from_rtr_infifo;
13                          2:       data_2_fft_infifo[31:24] <= data_from_rtr_infifo;
14                          3:       data_2_fft_infifo[23:16] <= data_from_rtr_infifo;
15                          4:       data_2_fft_infifo[15:8]  <= data_from_rtr_infifo;
16                          default: data_2_fft_infifo[7:0]   <= data_from_rtr_infifo;
17                        endcase
18
19                        if ( byte_cnt == 5 ) begin
20                          byte_cnt <= 0; push_fft_infifo <= 1;
21                        end else byte_cnt <= byte_cnt + 1;
22
23                        state_b <= feed_fft_from_rtr_4;
24
25                      end
26
27 feed_fft_from_rtr_4: state_b <= feed_fft_from_rtr_1;
```

3-21. ábra. *A központi vezérlő állapotgép (FSM B) FFT-üzemmódhoz tartozó szubrutinszerű állapotsorozatai (2)*

komplex elem FFT modulba való beírása előtt hat darab *pop* műveletet kell elvégeznünk. A megoldás alapja ezúttal is a *stream_2_router_1...3* állapotsorozatban bevezetett *byte_cnt* számlálóregiszter, amit minden *pop* művelet után inkrementálunk, miközben értékétől függően a 8-bites *data_from_rtr_infifo* busz tartalmát a 48-bites *data_2_fft_infifo* busz megfelelő mezőjébe irányítjuk (3-21. ábra: 10–17. sor).

Verilog

```
1  feed_fft_from_extmem_1: if ( fft_infifo_full == 1 ) state_b <= return_state_b;
2                          else begin
3                            extmem_rstrobe <= 1; state_b <= feed_fft_from_extmem_2;
4                          end
5
6  feed_fft_from_extmem_2: state_b <= feed_fft_from_extmem_3;
7
8  feed_fft_from_extmem_3: if ( extmem_rdy2rd == 1 ) begin
9
10                           case ( byte_cnt )
11                             0:       data_2_fft_infifo[47:40] <= data_from_extmem;   // MSB first
12                             1:       data_2_fft_infifo[39:32] <= data_from_extmem;
13                             2:       data_2_fft_infifo[31:24] <= data_from_extmem;
14                             3:       data_2_fft_infifo[23:16] <= data_from_extmem;
15                             4:       data_2_fft_infifo[15:8]  <= data_from_extmem;
16                             default: data_2_fft_infifo[7:0]   <= data_from_extmem;
17                           endcase
18
19                           if ( byte_cnt == 5 ) begin
20                             byte_cnt <= 0; push_fft_infifo <= 1;
21                           end else byte_cnt <= byte_cnt + 1;
22
23                           extmem_raddr <= extmem_raddr + 1;
24                           state_b <= feed_fft_from_extmem_4;
25
26                         end
27
28 feed_fft_from_extmem_4: state_b <= feed_fft_from_extmem_1;
```

3-22. ábra. *A központi vezérlő állapotgép (FSM B) FFT-üzemmódhoz tartozó szubrutinszerű állapotsorozatai (3)*

Verilog

```
1  send_fft_result_2_rtr_1: if ( fft_outfifo_empty == 1 ) state_b <= return_state_b;
2                           else begin
3                             pop_fft_outfifo <= 1; state_b <= send_fft_result_2_rtr_2;
4                           end
5
6  send_fft_result_2_rtr_2: state_b <= send_fft_result_2_rtr_3;
7
8  send_fft_result_2_rtr_3: begin
9
10                            case ( byte_cnt )
11                             0:       data_2_rtr_outfifo <= data_from_fft_outfifo[47:40];   // MSB first
12                             1:       data_2_rtr_outfifo <= data_from_fft_outfifo[39:32];
13                             2:       data_2_rtr_outfifo <= data_from_fft_outfifo[31:24];
14                             3:       data_2_rtr_outfifo <= data_from_fft_outfifo[23:16];
15                             4:       data_2_rtr_outfifo <= data_from_fft_outfifo[15:8];
16                             default: data_2_rtr_outfifo <= data_from_fft_outfifo[7:0];
17                            endcase
18
19                            if ( rtr_outfifo_full == 0 ) begin
20                             push_rtr_outfifo <= 1;
21
22                             if ( byte_cnt == 5 ) begin
23                               byte_cnt <= 0; state_b <= send_fft_result_2_rtr_1;
24                             end else begin
25                               byte_cnt <= byte_cnt + 1; state_b <= send_fft_result_2_rtr_4;
26                             end
27
28                            end else rtr_outfifo_error <= 1;
29
30                          end
31
32 send_fft_result_2_rtr_4: state_b <= send_fft_result_2_rtr_3;
```

3-23. ábra. *A központi vezérlő állapotgép (FSM B) FFT-üzemmódhoz tartozó szubrutinszerű állapotsorozatai (4)*

Verilog

```
1  send_fft_result_2_extmem_1: if ( fft_outfifo_empty == 1 ) state_b <= return_state_b;
2                              else begin
3                                pop_fft_outfifo <= 1; state_b <= send_fft_result_2_extmem_2;
4                              end
5
6  send_fft_result_2_extmem_2: state_b <= send_fft_result_2_extmem_3;
7
8  send_fft_result_2_extmem_3: begin
9
10                               case ( byte_cnt )
11                                0:       data_2_extmem <= data_from_fft_outfifo[47:40];   // MSB first
12                                1:       data_2_extmem <= data_from_fft_outfifo[39:32];
13                                2:       data_2_extmem <= data_from_fft_outfifo[31:24];
14                                3:       data_2_extmem <= data_from_fft_outfifo[23:16];
15                                4:       data_2_extmem <= data_from_fft_outfifo[15:8];
16                                default: data_2_extmem <= data_from_fft_outfifo[7:0];
17                               endcase
18
19                               extmem_wstrobe <= 1;
20                               state_b <= send_fft_result_2_extmem_4;
21
22                             end
23
24 send_fft_result_2_extmem_4: state_b <= send_fft_result_2_extmem_5;
25
26 send_fft_result_2_extmem_5: if ( extmem_rdy2wr == 1 ) begin
27
28                               if ( byte_cnt == 5 ) begin
29                                 byte_cnt <= 0; state_b <= send_fft_result_2_extmem_1;
30                               end else begin
31                                 byte_cnt <= byte_cnt + 1; state_b <= send_fft_result_2_extmem_3;;
32                               end
33
34                               extmem_waddr <= extmem_waddr + 1;
35
36                             end
```

3-24. ábra. *A központi vezérlő állapotgép (FSM B) FFT-üzemmódhoz tartozó szubrutinszerű állapotsorozatai (5)*

Ez az ütemezési logika alapvetően a külső memóriával való együttműködés esetén is megállja a helyét, hiszen – a lehető legszélesebb körű alkalmazhatóság érdekében – a DSP-modul bájtos szervezésű külső operatív tárat feltételez, vagyis a memória-interfész adatbuszai az útválasztóhoz kapcsolt FIFO-khoz hasonlóan 8-bitesek. Figyelembe kell azonban vennünk, hogy a DSP-modul belső FIFO-itól eltérően a külső memória – vagy az interfész tekintetében azzal egyenértékű memória-arbiter – nem képes nulla késleltetésű szinkron kézfogásos tranzakciókra, vagyis az *FSM B* állapotgépnek a megfelelő állapotaiban feltétlenül figyelembe kell vennünk a kézfogás mechanizmus *rdy2rd (ready to read)* és *rdy2wr (ready to write)* visszacsatoló jeleit (lásd 3-22 ábra: 8. sor, 3-24. ábra: 26. sor).

Az FFT modul be- és kimeneti FIFO-ihoz tartozó, kizárólag adatmozgatást végző szubrutinszerű vezérlésiállapot-sorozatok segítségével a teljes FFT műveletsort levezénylő *fft_1...5* állapotsorozat már egyszerűen létrehozható (lásd 3-25. ábra). A transzformáció állapotának visszajelzésére létrehozzuk az *fft_done* állapotjelzőt, amit a művelet megkezdésekor – tehát amikor a *mode* vezérlőregiszter 10_2 értéket vesz fel és az *FSM B* állapotgép *fft_1* állapotba kerül – nullázunk (3-25. ábra: 3. sor). Ezzel egyidőben megvizsgáljuk a transzformálandó vektor forrását definiáló *fft_source* vezérlőjel aktuális értékét, ami alapján eldönthető, hogy melyik szubrutinszerű állapotsorozattal „végeztetjük el" az FFT modul bemeneti FIFO-jának feltöltését (3-25. ábra: 5-9. sor). A bemeneti FIFO feltöltése után – függetlenül attól, hogy azt melyik szubrutinszerű állapotsorozat végezte el – az *fft_2...4* állapotokon lépkedünk végig, ahol – a megszokott módon, egy szinkron kézfogásos tranzakció segítségével – utasítjuk az FFT modult a transzformált vektor kiszámítására (3-25. ábra: 15–21. sor). Amikor az FFT modul – *ready* kimenetének magas szintre állításával – jelzi a művelet végét, akkor az *fft_sink* vezérlőjel függvényében eldöntjük, hogy melyik szubrutinszerű állapotsorozatot használjuk a transzformált vektor FFT modulból való kiolvasására és a célhelyre való továbbítására (3-25. ábra: 23–25. sor). A transzformált vektor továbbítása után, az *fft_5* állapotban az *fft_done* állapotjelzőt magas szintre húzzuk. Ezt a jelzést a *fir_infifo_error* és *rtr_outfifo_error* hibajelző impulzusjelekhez hasonlóan a DSP-modul hibadetektáló és megszakításkérő logikája dolgozza majd fel (lásd később).

Verilog

```
1  fft_1: begin
2
3    fft_done <= 0;
4
5    case ( fft_source )
6      2'b00:   state_b <= feed_fft_from_fir_1;
7      2'b01:   state_b <= feed_fft_from_rtr_1;
8      default: state_b <= feed_fft_from_extmem_1;
9    endcase
10
11   return_state_b <= fft_2;
12
13 end
14
15 fft_2: begin
16           request_fft <= 1; state_b <= fft_3;
17         end
18
19 fft_3: state_b <= fft_4;
20
```

Verilog (folyt.)

```
21 fft_4: if ( fft_core_ready == 1 ) begin
22
23           if ( fft_sink == 0 )
24             state_b <= send_fft_result_2_rtr_1;
25           else state_b <= send_fft_result_2_extmem_1;
26
27           return_state_b <= fft_5;
28
29         end
30
31 fft_5: begin
32
33   fft_done <= 1;
34
35   if ( mode == 2'b10 && fft_source == 2'b00 )
36     state_b <= fft_1;
37   else if ( mode == 2'b00 ) state_b <= cleanup;
38
39 end
```

3-25. ábra. *A központi vezérlő állapotgép (FSM B) FFT-üzemmódban érvényesülő vezérlési állapotai*

Már a rendszertervezés szintjén rögzítettük, hogy ha a diszkrét Fourier-transzformáció forrásaként a felhasználó – az *fft_source* vezérlőregiszter megfelelő beállításával – a FIR-szűrő kimeneti

FIFO-ját jelöli ki, akkor feltételezzük, hogy az A/D-átalakítótól származó adatfolyam egymást követő mintasor-részletein újabb és újabb transzformációkat szeretne elvégezni, amelyek eredményét vagy az útválasztón keresztül kérdezi le, vagy a külső memória meghatározott báziscímen kezdődő tartományából olvassa ki. A *mode* és *fft_source* vezérlőjeleknek ezt a kombinációját az *FSM B* állapotgép az *fft_5* állapotban dekódolja, és a feltételek teljesülése esetén az *fft_1* állapotba lépve újabb transzformációs ciklust kezd (3-25. ábra: 35–36. sor). Ha a fenti, a periodikus FFT-számításra vonatkozó igényt indikáló feltételek nem teljesülnek, akkor az állapotgép mindaddig a tétlen *fft_5* állapotban marad, ameddig a *mode* regiszter értékét a felhasználó 00_2-ra nem állítja, mintegy nyugtázva az egyszeri transzformáció elvégzését.

A DSP-bővítőmodul meglehetősen összetett, működése során többféle esemény is bekövetkezhet, amit valamilyen módon a felhasználónak jelezni érdemes. Bővítőmodul lévén ennek az IP-magnak is „joga van” – akár közvetlenül, akár a megszakításvezérlőn keresztül – csatlakozni a *CPU* megszakítási rendszeréhez, ami az ilyen különleges események felhasználó felé való jelzésének tökéletes eszköze. A DSP-modulban nyolc „eseményt” definiálunk (lásd 3-5. táblázat).

Esemény jelzése áramköri szinten	Esemény
Felfutóél az *fft_core_ready* jelen.	Az FFT IP-mag befejezte az aktuális transzformációt, megkezdődött a kimeneti FIFO kiolvasása.
Felfutóél az *fft_done* állapotjelzőn.	Az *FSM B* állapotgép befejezte a legutóbb elvégzett FFT művelet eredményvektorának célhelyre mozgatását.
Felfutóél az *ifa_fifo_error* jelen.	Az A/D-átalakító kimeneti FIFO-ja megtelt, a legutóbbi konverzió eredménye elveszett.
Felfutóél a *fir_infifo_error* jelen.	Az *FSM A* állapotgép a FIR-szűrő bemeneti FIFO-jának telítettsége miatt nem tudta eltárolni a következő mintát, ami így elveszett.
Felfutóél a *rtr_outfifo_error* jelen.	Az *FSM B* állapotgép – mintafolyam üzemmódban – az útválasztóhoz kapcsolt kimeneti FIFO telítettsége miatt nem tudta eltárolni a FIR-szűrőtől érkező mintát, ami így elveszett.
Felfutóél a *mac_of* jelen.	Túl- vagy alulcsordulás a *SAMPLE SCALER* modulban.
Felfutóél a *fir_of* jelen.	Túl- vagy alulcsordulás a FIR-szűrőben.
Felfutóél az *fft_of* jelen.	Túl- vagy alulcsordulás az FFT modulban.

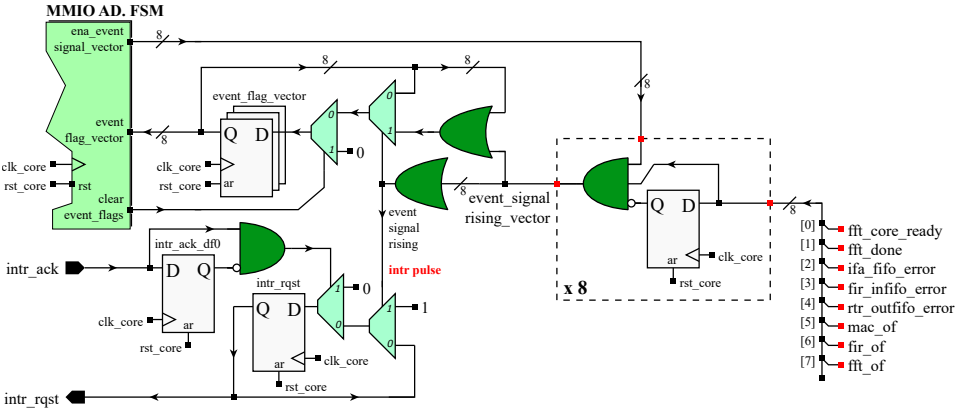
3-5. táblázat. *Megszakításkérést kiváltó események a DSP-modulban*

Áramköri szinten mindegyik eseményt egy állapotjelzőn bekövetkező felfutóél jelzi, ezért a DSP-modul megszakításkérő logikájában (lásd 3-26. ábra) mindegyik állapotjelzőhöz létrehozunk egy-egy dedikált, külön-külön engedélyezhető/letiltható felfutóél-detektort, amelyek kimeneteit egy közös, 8-bites vektorba (3-26. ábra: *event_signal_rising_vector*) fűzzük össze. Bármely éldetektor kimenetének aktiválódása megszakításkérést vált ki (3-26. ábra: 12. sor), ez azonban önmagában még nem elegendő, hiszen a *CPU*-n futó program ezáltal csak a megszakításkérés tényéről értesülhet, de a megszakításkiszolgáló rutinnak így még nem áll módjában megállapítani, hogy azt pontosan mi okozta. Ahhoz, hogy a megszakításkérés forrását is azonosíthatóvá tegyük, létrehozunk egy újabb, 8-bites állapotjelző vektort (3-26. ábra: *event_flag_vector*), amelyben a felfutóél-detektorok kimeneteinek jelzéseit permanensen regisztráljuk (3-26. ábra: 24–25. sor). Figyeljük meg, hogy az éldetektorok kimenetén megjelenő pozitív impulzusok hatására az *event_flag_vector* regiszterbe az éldetektorok kimeneteit összefogó *event_signal_rising_vector* busznak és magának az

Verilog

```
always @ ( posedge clk_core, posedge rst_core ) begin
  if ( rst_core == 1 ) begin
    intr_ack_df0 <= 0; intr_rqst <= 0;
    fft_core_ready_df0 <= 0; fft_done_df0 <= 0; ifa_fifo_error_df0 <= 0; fir_infifo_error_df0 <= 0;
    rtr_outfifo_error_df0 <= 0; mac_of_df0 <= 0; fir_of_df0 <= 0; fft_of_df0 <= 0;
    event_flag_vector <= 0;
  end else begin

    intr_ack_df0 <= intr_ack;

    if ( intr_ack_rising == 1 ) intr_rqst <= 0;
    else if ( event_signal_rising == 1 ) intr_rqst <= 1;

    fft_core_ready_df0 <= fft_core_ready;
    fft_done_df0 <= fft_done;
    ifa_fifo_error_df0 <= ifa_fifo_error;
    fir_infifo_error_df0 <= fir_infifo_error;
    rtr_outfifo_error_df0 <= rtr_outfifo_error;
    mac_of_df0 <= mac_of;
    fir_of_df0 <= fir_of;
    fft_of_df0 <= fft_of;

    if ( clear_event_flags == 1 ) event_flag_vector <= 0;
    else if ( event_signal_rising == 1 ) event_flag_vector <= event_flag_vector |
                                                              event_signal_rising_vector;

  end
end

assign event_signal_rising_vector[0] = ena_event_signal_vector[0] & fft_core_ready & ~fft_core_ready_df0;
assign event_signal_rising_vector[1] = ena_event_signal_vector[1] & fft_done & ~fft_done_df0;
assign event_signal_rising_vector[2] = ena_event_signal_vector[2] & ifa_fifo_error & ~ifa_fifo_error_df0;

assign event_signal_rising_vector[3] = ena_event_signal_vector[3] &
                                       fir_infifo_error & ~fir_infifo_error_df0;

assign event_signal_rising_vector[4] = ena_event_signal_vector[4] &
                                       rtr_outfifo_error & ~rtr_outfifo_error_df0;

assign event_signal_rising_vector[5] = ena_event_signal_vector[5] & mac_of & ~mac_of_df0;
assign event_signal_rising_vector[6] = ena_event_signal_vector[6] & fir_of & ~fir_of_df0;
assign event_signal_rising_vector[7] = ena_event_signal_vector[7] & fft_of & ~fft_of_df0;

assign event_signal_rising = (event_signal_rising_vector == 0) ? 0 : 1;

assign intr_ack_rising = intr_ack & ~intr_ack_df0;
```

Áramköri struktúra



3-26. ábra. *A DSP-modul megszakításkérő logikája*

event_flag_vector regiszternek az *OR* kapcsolata íródik be. Ezzel a megoldással az egymást követő, más-más eseményjelző vonalhoz tartozó felfutóél-detektorok egymás korábban regisztrált jelzéseit nem tudják törölni.

A megszakításkérések forrása az *event_flag_vector* segítségével már azonosítható, de természetesen csak akkor, ha ezt a regisztert a *CPU*-n futó program számára is hozzáférhetővé tesszük. A 3-6. táblázat a megszakításkérő logikához tartozó újabb MMIO-címeket és azok szerepét foglalja össze.

Cím azonosítója és típusa		A címhez rendelt funkció leírása
address_ena_event_signals	W	Az egyes eseményekhez rendelt felfutóél-detektorok engedélyezését/letiltását szolgáló vezérlőregiszter címe.
address_event_flag_vector	R	Az e címen végrehajtott olvasás művelettel lekérdezhető a megszakításkérések forrását azonosító *event_flag_vector* állapotregiszter.
address_clear_event_flags	W	Az e címen végrehajtott írás művelettel törölhető a megszakításkérések forrását rögzítő *event_flag_vector* állapotregiszter.

3-6. táblázat. *A DSP-modul megszakításkérő logikájához tartozó MMIO-címek*

A 3-27. ábra a DSP-bővítőmodul MMIO-adapter állapotgépének a megszakításkérő logikát kiszolgáló részleteit mutatja be.

Verilog

```
idle: begin
  s_ready_2_cpu <= 1;

  // Handle MMIO write requests...
  if ( write_strobe_from_cpu_df1 == 1 && cs == 1 ) begin
    s_ready_2_cpu <= 0;

    if ( [...] ) begin [...]
    end else if ( address_from_cpu == address_ena_event_signals ) begin
      ena_event_signal_vector <= data_from_cpu[7:0]; state_mmio <= wait_for_deassert_strobes;
    end else if ( address_from_cpu == address_clear_event_flags ) begin
      clear_event_flags <= 1; state_mmio <= wait_for_deassert_strobes;
    end else state_mmio <= wait_for_deassert_strobes;

  end

  // Handle MMIO read requests...
  if ( read_strobe_from_cpu_df1 == 1 && cs == 1 ) begin
    s_ready_2_cpu <= 0;

    if ( address_from_cpu == address_event_flag_vector ) begin
      s_data_2_cpu[15:8] <= 0; s_data_2_cpu[7:0] <= event_flag_vector;
      state_mmio <= wait_for_deassert_strobes;
    end else state_mmio <= wait_for_deassert_strobes;

  end

end

[...]

wait_for_deassert_strobes: begin
  clear_event_flags <= 0;
  if ( read_strobe_from_cpu_df1 == 0 && write_strobe_from_cpu_df1 == 0 ) state_mmio <= idle;
end
```

3-27. ábra. *A DSP-modul MMIO-adapter állapotgépének vezérlési állapotai (4)*

3.3. Lebegőpontos aritmetikai bővítőmodul

Áttanulmányozásra javasolt alfejezetek:
4.5.3: *Aritmetika és jelfeldolgozás – Lebegőpontos aritmetikai egység*

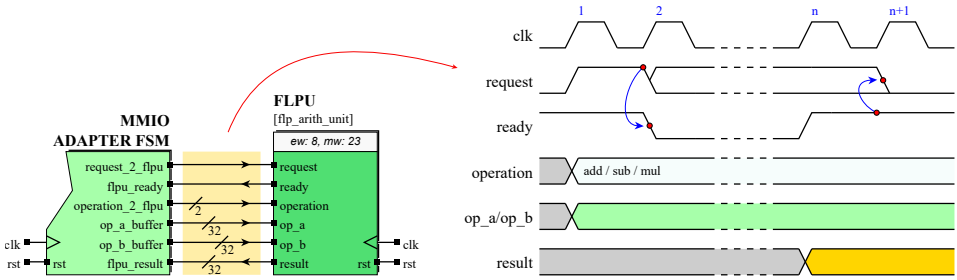
A funkcióbővítő rendszer kifejlesztésének kezdetén abból a feltételezésből indultunk ki, hogy az azt üzemeltető rendszer alapvetően egyszerű – pl. egy kisméretű és kis számítási teljesítményű mikrovezérlő –, ezért van szükség rá, hogy az adott alkalmazástól függően újabb és újabb többletfunkciókat adjunk hozzá. A lebegőpontos aritmetika – számítástechnikai összetettsége miatt – tipikusan nem képezi részét az egyszerű mikrovezérlős rendszereknek. A lebegőpontos aritmetikai alapműveletek ugyan a legegyszerűbb mikrovezérlőn is megvalósíthatók „szoftverben", a megfelelő célhardver híján azonban még ezeknek az alapműveleteknek az elvégzése is rengeteg időt igényel. Ráadásul valós problémák megoldása során jellemzően az alapműveletek sokaságát kell elvégezni, ezért a tisztán szoftveres megoldás inkább csak elvi jelentőségű, a gyakorlatban nem jellemző. A műveletvégzés sebessége lebegőpontos aritmetikai céláramkörökkel ugyan nagyságrendekkel gyorsítható, a legkisebb mikrovezérlőkbe azonban ilyen részegységeket egész egyszerűen azok erőforrásigénye és fogyasztása miatt nem építenek; egy gyakorlati feladatok megoldására is érdemben használható lebegőpontos aritmetikai egység mérete összemérhető lehet magának a mikrovezérlőnek a méretével, sőt, akár meg is haladhatja azt.

A fentiek értelmében a funkcióbővítő rendszerrel jelentős számításiteljesítmény-növekedést érhetünk el, ha a komplex lebegőpontos adatfeldolgozó algoritmusokat a *CPU*-n futó programmal valósítjuk meg úgy, hogy a lebegőpontos aritmetikai alapműveletek végrehajtására célhardvert építünk be egy bővítőmodul formájában.[90]

Egy-egy új funkcionális egység tervezése előtt – a 2. fejezet bevezetőjében tárgyalt alapelvek értelmében – mindig célszerű mérlegelni újrafelhasználható IP-magok létrehozásának lehetőségét. A lebegőpontos aritmetikai bővítőmodul esete ebből a szempontból elég magától értetődő; az alapműveletek gyors végrehajtására képes részáramköröket egy, a lehető legáltalánosabb, szinkron kézfogásos interfésszel ellátott IP-magként célszerű megtervezni, amit azután a megfelelő illesztő logika hozzáadásával – a 2-74. ábrán szemléltetett módon – tehetünk MMIO-kompatibilissé (lásd 3-28. ábra).

A szinkron kézfogásos interfészű, lebegőpontos összeadás, kivonás és szorzás műveletek végrehajtására képes IP-mag RTL-tervének részleteit a 4.5.3. pont tárgyalja. E modul magát a számformátumot, vagyis a karakterisztika és a mantissza méretét szintézis-paraméterként kezeli. Bővítőmodulként való felhasználásakor ezeket a paramétereket az ANSI/IEEE Std 754-1985 szabvány szerinti egyszeres pontosságú formátumnak megfelelően rögzítjük; a karakterisztikát 8,

[90] A lebegőpontos aritmetika ma már nem tekinthető kifejezetten egyedi funkciónak, régóta léteznek olyan – közepesen összetettnek nevezhető – mikrovezérlők, amelyek tartalmaznak lebegőpontos aritmetikai céláramköröket. Vagyis a funkcióbővítő rendszer egészének létrehozását – beleértve az annak helyet adó FPGA-t és az eddig bemutatott alaprendszert – pusztán a lebegőpontos aritmetika iránti igény nem indokolja. Ha a lebegőpontos aritmetikán kívül semmi egyébre nincs szükségünk, akkor egy megfelelő mikrovezérlő kiválasztásával rendszertechnikailag egyszerűbb és hatékonyabb megoldáshoz jutunk. Ez a gondolat valójában bármilyen bővítőmodulra igaz, hacsak nem valami extrém egyedi funkcióról beszélünk. Funkcióbővítő rendszerünk létjogosultsága nem az egyes bővítőmodulokban, hanem azoknak az adott alkalmazás igényei szerint összeállított kombinációjában rejlik.



3-28. ábra. *Az MMIO-kompatibilis lebegőpontos aritmetikai egység szerkezete*

a mantisszát pedig 23 biten ábrázoljuk, így az előjelbittel együtt 32-bites operandus- és eredményvektorokat kapunk.

Az IP-mag MMIO-buszrendszerre való csatlakoztatását lehetővé tevő illesztőlogika – az alaprendszer összetevőinek ismeretében – nagyon egyszerűen létrehozható, hiszen ezúttal is szinkron kézfogásos interfészen elérhető szolgáltatásokat kell kiajánlanunk a *CPU* számára (vö. 2-79. ábra: *BRAM* modul, 2-87. ábra: *EXTRAM* modul stb.). Az MMIO-kompatibilis lebegőpontos aritmetikai egység címtérképét a 3-7. táblázat szemlélteti.

Cím azonosítója és típusa		A címhez rendelt funkció leírása
address_op_a_MSW	W	A 32-bites *A* operandus felső 16 bitjét tároló regiszter címe.
address_op_a_LSW	W	A 32-bites *A* operandus alsó 16 bitjét tároló regiszter címe.
address_op_b_MSW	W	A 32-bites *B* operandus felső 16 bitjét tároló regiszter címe.
address_op_b_LSW	W	A 32-bites *B* operandus alsó 16 bitjét tároló regiszter címe.
address_operation	W	Az elvégzendő művelet kódját tartalmazó kétbites regiszter címe. • 00_2: összeadás • 01_2: kivonás • 10_2: szorzás Az MMIO írásműveletben a *CPU* által küldött 16-bites vektor felső 14 bitjét a modul figyelmen kívül hagyja. Az e címen végrehajtott írásművelet egyúttal kezdeményezi a modul belsejében példányosított lebegőpontos aritmetikai egység működésének elindítását is.
address_result_MSW	R	A 32-bites eredmény felső 16 bitjét tároló regiszter címe.
address_result_LSW	R	A 32-bites eredmény alsó 16 bitjét tároló regiszter címe.

3-7. táblázat. *Az MMIO-kompatibilis lebegőpontos aritmetikai egység címtérképe*
*MSW: **M**ost **S**ignificant **W**ord, LSW: **L**east **S**ignificant **W**ord*

Az e címtérképet megvalósító MMIO-adapter állapotgép vezérlési állapotai a 3-29. ábrán láthatók.

A *CPU*-n futó program feladata, hogy megfelelően „üzemeltesse” az MMIO-kompatibilis lebegőpontos aritmetikai egységet. Bármely alapművelet elvégzésekor először el kell helyezni az operandusokat a 32-bites *op_a_buffer* és *op_b_buffer* regiszterekben (3-29. ábra: 8–15. sor). Mivel az MMIO-buszrendszer csak 16-bites szavakat képes egylépésben a periféria-áramkörbe írni, ezért operandusonként két MMIO írásműveletre van szükség. Ezután az *operation_buffer* regiszterbe egy újabb MMIO írásművelettel helyezhetjük el az elvégzendő művelet kódját. Ez az írásművelet egyúttal a belső lebegőpontos aritmetikai IP-magot is működésbe hozza (3-29. ábra: 18.

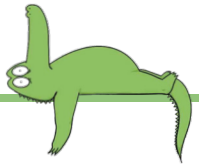
Verilog

```
idle: begin
  s_ready_2_cpu <= 1;

  // Handle MMIO write requests...
  if ( write_strobe_from_cpu_df1 == 1 && cs == 1 ) begin
    s_ready_2_cpu <= 0;

    if ( address_from_cpu == address_op_a_MSW ) begin
      op_a_buffer[31:16] <= data_from_cpu; state <= wait_for_deassert_strobes;
    end else if ( address_from_cpu == address_op_a_LSW ) begin
      op_a_buffer[15:0] <= data_from_cpu; state <= wait_for_deassert_strobes;
    end else if ( address_from_cpu == address_op_b_MSW ) begin
      op_b_buffer[31:16] <= data_from_cpu; state <= wait_for_deassert_strobes;
    end else if ( address_from_cpu == address_op_b_LSW ) begin
      op_b_buffer[15:0] <= data_from_cpu; state <= wait_for_deassert_strobes;
    end else if ( address_from_cpu == address_operation ) begin
      operation_buffer <= data_from_cpu[1:0];
      request_2_flpu <= 1;
      state <= calculate_1;
    end else state <= wait_for_deassert_strobes;

  end

  // Handle MMIO read requests...
  if ( read_strobe_from_cpu_df1 == 1 && cs == 1 ) begin
    s_ready_2_cpu <= 0;

    if ( address_from_cpu == address_result_MSW ) begin
      s_data_2_cpu <= flpu_result[31:16]; state <= wait_for_deassert_strobes;
    end else if ( address_from_cpu == address_result_LSW ) begin
      s_data_2_cpu <= flpu_result[15:0]; state <= wait_for_deassert_strobes;
    end else state <= wait_for_deassert_strobes;

  end

end

calculate_1: state <= calculate_2;

calculate_2: if ( flpu_ready == 1 ) begin
               request_2_flpu <= 0; state <= wait_for_deassert_strobes;
             end

wait_for_deassert_strobes:  if (
                              read_strobe_from_cpu_df1 == 0 &&
                              write_strobe_from_cpu_df1 == 0
                            ) state <= idle;
```

3-29. ábra. *A lebegőpontos aritmetikai egység MMIO-adapter állapotgépének vezérlési állapotai*

sor). A *calculate_2* vezérlési állapotban láthatjuk, hogy a számítás elvégzésének időtartama alatt az MMIO-tranzakció is várakozik, vagyis a *CPU*-n futó program szempontjából a számítás az *address_operation* címen végrehajtott *store* utasítás végrehajtásának részeként történik meg. Ez azt jelenti, hogy e *store* utasítást a programkódban azonnal követhetik a 32-bites eredményvektort kiolvasó két MMIO-olvasást kiváltó *load* utasítások.

4



IP-katalógus

A 2. fejezet bevezetőjében az újrafelhasználhatóságra törekvést az RTL-tervezés egyik legfontosabb irányelvének neveztük. Ez az álláspont a regiszter-transzfer elvonatkoztatási szint és a hardverleíró nyelvek bevezetése óta eltelt közel négy évtized alatt összegyűlt tapasztalatok alapján valóban megalapozottnak tűnik; a mai rendszerchipekre (**S**ystem-**o**n-**C**hip, SoC) jellemző összetettség és megbízhatóság aligha lenne elérhető, ha terveinket nem korábban elkészített és verifikált, parametrikus formában megfogalmazott, sokféle alkalmazási környezetben helytálló IP-magok közvetlen, vagy némi inkrementális továbbfejlesztés utáni integrálásával készítenénk. Ebben a fejezetben a funkcióbővítő rendszer részletes tervezése során azonosított, kontextustól függetlenül értelmezhető feladatokat ellátó részáramköröket mutatunk be. Ezek az IP-magok áramköri terveknek egy koherens katalógusát alkotják, ami legfőképpen egységes (reguláris) működésű interfészeikben nyilvánul meg, továbbá abban, hogy helyenként egymást példányosítva képeznek összetettebb funkciójú részegységeket.

4.1. Segédmodulok

4.1.1. Órajel-előosztó

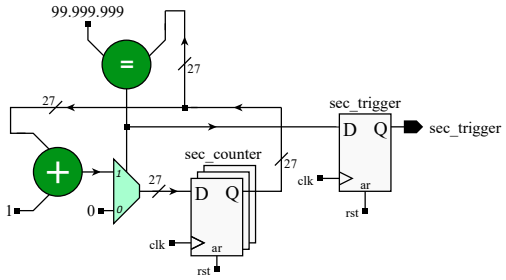
Az FPGA-kban megvalósított digitális áramkörökben az órajelek frekvenciája tipikusan a néhányszor 10 MHz tartományba esik, a legmodernebb (2025) eszközök esetén pedig akár a 200-300 MHz-et is elérheti. A magas órajel-frekvencia nagy számítási teljesítményt biztosít, ami nyilván jól jön, ha egy adatfeldolgozó rendszerről beszélünk, ugyanakkor bizonyos alkalmazásokban célszerű az áramköreink működését valami módon lelassítani. Gondoljunk egy egyszerű időzítő áramkörre, amelynek az a feladata, hogy másodpercenként előállítson egy-egy impulzust a kimenetén. Ha az időzítőt egy 100 MHz-es órajelről járatjuk, akkor a benne lévő számlálónak százmillió órajel-felfutóélt kell megszámlálnia, mielőtt egy impulzust generál a kimenetén (lásd 4-1. ábra).

A 4-1. ábrán látható áramkörben az órajel maximális frekvenciáját meghatározó, legnagyobb késleltetésű lokális adatút (kritikus út) a 27-bites összeadót megvalósító kombinációs hálózatban alakul ki. Ha az áramkör és a céltechnológia (az adott FPGA LUT-jainak és vezetékhálózatainak késleltetései) kombinációja által meghatározott maximális órajel-frekvencia nem éri el a tervezési peremfeltételként adott 100 MHz-et, akkor ez a megoldás nem használható. Az 1.2.1.3. pontban a sok almodult tartalmazó rendszerek órajelforrásaival kapcsolatban megállapítottuk, hogy a fogyasztás optimalizálásának érdekében minden almodult olyan órajellel érdemes hajtani, amelynek frekvenciája az elvégzendő feladattal összhangban van, a másodperces időzítőt 100 MHz-es

VHDL

```
1  L_SEC_CNT: process ( clk, rst ) begin
2  | if ( rst = '1' ) then
3  | | sec_counter <= 0; sec_trigger <= '0';
4  | elsif ( rising_edge(clk) ) then
5  | | if ( sec_counter = 99999999 ) then
6  | | | sec_counter <= 0; sec_trigger <= '1';
7  | | else
8  | | | sec_counter <= cnt + 1;
9  | | | sec_trigger <= '0';
10 | | end if;
11 | end if;
12 end process;
```

Áramköri struktúra



4-1. ábra. *Egy másodperces periódusú időzítő 100 MHz-es órajellel*

órajelről járatni tehát közel sem optimális. Az önálló órajeltartomány létrehozása azonban több szempontból is problémás lehet:

- Ha az órajel frekvenciája tervezési peremfeltétel, akkor annak csökkentése nem opció; nem feltétlenül engedhetjük meg magunknak, hogy a felhasználónak előírjuk az IP-mag bemenetére juttatandó órajel frekvenciáját.
- Ha az IP-mag órajelével kapcsolatban mégis megfogalmazunk valami feltételt, akkor „illik" gondoskodni a megfelelő órajel-szinkronizáló fokozatokról (lásd 2.1. alfejezet), ami egyszerű funkciójú IP-magok esetén a probléma szükségtelen túlbonyolítása lenne.

A fenti problémákra kézenfekvő megoldás, ha a nagyfrekvenciájú órajelet az IP-magon belül leosztjuk, és az áramkör belső részegységeit a leosztott órajellel hajtjuk meg. Az órajel leosztását végző áramkört órajel-előosztónak (prescaler) nevezzük. Ennek a funkcionális egységnek egy lehetséges megvalósítását mutatja a 4-2. ábra.

Az órajel-előosztó alapja egy számláló (4-2. ábra bal oldala), amelynek értéke – feltéve, hogy a *zero* bemenet alacsony, az *ena* pedig magas szintű – az órajel minden felfutóélénél eggyel növekszik, majd egy futásidőben beállítható maximális érték elérésekor átfordul, miközben egy pozitív impulzust állít elő a *limit* jelen. A *limit* jelet fogadó éldetektor kimenetén minden átforduláskor egy egyetlen órajel-periódusnyi pozitív impulzus keletkezik. E pozitív impulzusok a *counter_max* vektor mindenkori értékétől függő időközönként követik egymást, tehát felfoghatók egy másodlagos, alacsonyabb frekvenciájú ütemjelként, amit kétféleképpen használhatunk fel:

- Hozzászokhattunk, hogy az órajel rendszerint egy 50%-os kitöltési tényezőjű négyszögjel, de áramköreink túlnyomó többsége valójában vagy csak a felfutó vagy csak a lefutó éleket használja ki.[91] Tehát annak ellenére, hogy az órajel-előosztó *tick* kimenetének kitöltési tényezője csekély, a jelalak tehát nem túl „órajelszerű", nincs elvi akadálya annak, hogy – ezt a jelet órajelként értelmezve – közvetlenül más flip-flopok órajel-bemenetét hajtsuk meg vele. Ennek a módszernek van egy nagy hátránya, ami csak a fizikai tervezés szintjén kerül előtérbe. Az RTL-tervezés egyik legfontosabb absztrakciója az órajelhálózat idealitása; az egyazon órajelforrásról járó flip-flopok az RTL-modellekben mind ugyanabban

[91] Az ún. **D**ual **D**ata **R**ate (DDR) megoldások, amelyekben a flip-flopok az órajel fel- és lefutóélénél egyaránt mintavételezik bemenetüket, csak akkor alkalmazhatók, ha a céltechnológia ezt lehetővé teszi, vagyis, ha az adott FPGA-ban megtalálhatóak az erre alkalmas flip-flopok. Ilyen értelemben ez a módszer technológiafüggő, ezért ahol lehet, kerüljük.

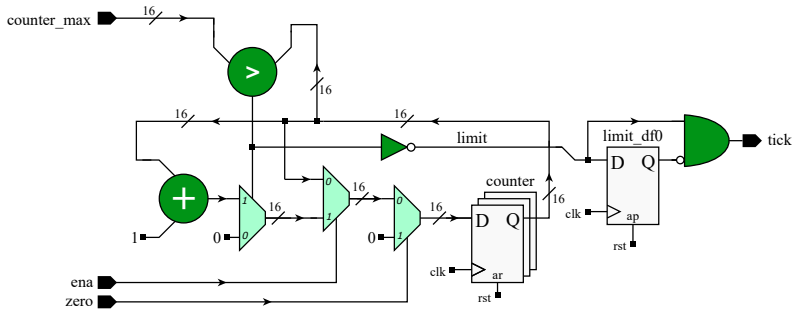
VHDL

```
1  process ( clk, rst ) begin
2  | if ( rst = '1' ) then counter <= (others => '0');
3  | elsif ( rising_edge(clk) ) then
4  | | if ( zero = '1' ) then counter <= (others => '0');
5  | | elsif ( ena = '1' ) then
6  | | | if ( counter < unsigned(counter_max) ) then
7  | | |   counter <= counter + 1;
8  | | | else counter <= (others => '0');
9  | | | end if;
10 | | end if;
11 | end if;
12 end process;
13
14 limit <= '0' when counter < unsigned(counter_max) else '1';
```

VHDL (folyt.)

```
15 process ( clk, rst )
16 begin
17 | if ( rst = '1' ) then
18 |   limit_df0 <= '1';
19 | elsif ( rising_edge(clk) ) then
20 |   limit_df0 <= limit;
21 | end if;
22 end process;
23
24 tick <= limit and not limit_df0;
```

Áramköri struktúra



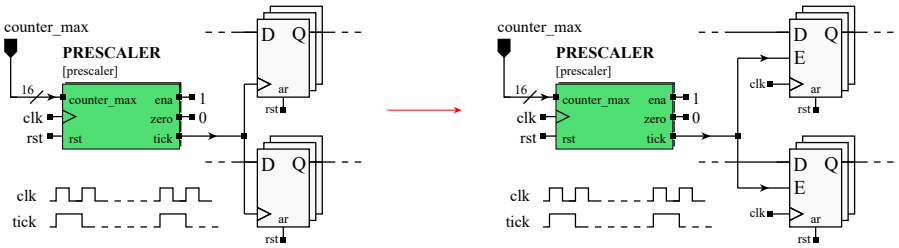
4-2. ábra. *Az órajel-előosztó egy lehetséges megvalósítása*

az időpillanatban érzékelik az órajel éleit, vagyis a modellezésnek ezen a szintjén mindig nulla késleltetésű órajelhálózattal, nulla órajelelcsúszással számolunk (lásd még DRMSz: 4.4.4. alfejezet). Ahhoz, hogy a szintézis és a fizikai tervezés után az RTL-modellel azonos viselkedésű áramköröket kapjunk, valami módon biztosítani kell, hogy ez a feltételezés a fizikailag megvalósított áramkörökre is igaz legyen. Az órajelelcsúszás minimalizálása céljából az FPGA-kban a gyártók dedikált órajelterjesztő vezetékhálózatokat, ún. órajelfákat alakítanak ki, amelyek tartalmazzák a szükséges nagy meghajtóképességű buffereket és elcsúszásmentesítő (de-skew) áramköröket. A flip-flopok órajelbemeneteit optimális esetben ezek az órajelfák hajtják meg (lásd még: DRMSz: 7-4. ábra). Ha az órajel-előosztó *tick* kimenetét közvetlenül flip-flopok órajelbemenetére kapcsoljuk, akkor a fizikai szintézist végző szoftver – az órajelelcsúszás minimalizálása érdekében – ezt a jelet is órajelként kezeli, és egy órajelfát hajt meg vele. Bár a mai modern FPGA-kban meglehetősen sok (több tíz vagy akár több száz), egymástól függetleníthető órajelfa áll rendelkezésre, bizonyos eszközökben ezek száma korlátozott.[92] Ha RTL-tervünk több órajelfát igényel, mint amennyit a céleszköz biztosítani tud, akkor a szintézis-szoftver kénytelen általános célú vezetékhálózatokon terjeszteni az órajeleket is, ami időzítési szempontból kellemetlen következményekkel jár; lecsökken a megengedhető maximális órajel-frekvencia, vagy egyes lokális adatutakon sérülhet a fogadó oldali flip-flop tartási maradékidőre vonatkozó

[92] A Microchip (Microsemi) RTAX/RTSX eszközcsaládjában például csupán két dedikált órajelfa, a HCLK (hardwired clock) és az RCLK (routed clock) áll rendelkezésre. Ez utóbbi felbontható ugyan több, külön órajellel meghajtható részhálózatra, de így is csupán néhány független órajelfa hozható létre.

időzítési feltétele („hold-time violation”), ami metastabil állapot kialakulását okozhatja (lásd még DRMSz: 6. fejezet).

- Az órajel-előosztó *tick* kimenetét a fenti problémák miatt rendszerint nem a flip-flopok órajelbemenetére, hanem azok órajel-engedélyező *(E)* bemenetére kapcsoljuk (lásd 4-3. ábra).



4-3. ábra. *Az órajel-előosztó használata*

Figyeljük meg, hogy ebben az esetben az áramkör összes flip-flopja egyazon nagyfrekvenciájú órajelről *(clk)* jár, vagyis a szintézis során csupán egyetlen órajelfát kell felhasználnunk, ugyanakkor az órajel-előosztó kimenete által engedélyezett flip-flopok a nagyfrekvenciájú órajelnek csak minden (*counter_max*+1)-edik élét „veszik figyelembe”, működésük tehát látszólag egy alacsonyabb frekvenciájú órajelről járó flip-flopéval egyezik meg.

Az órajel-előosztó használatával a másodperc-időzítő a 4-4. ábrán látható módon valósítható meg.

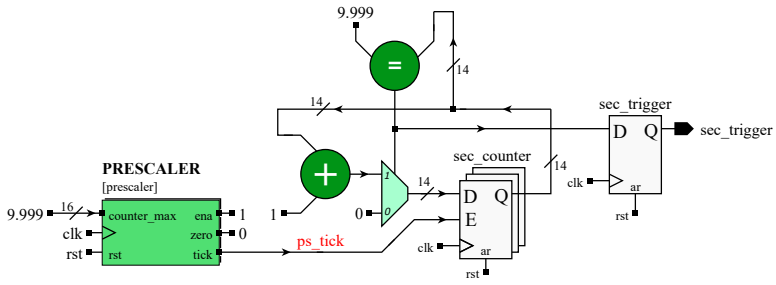
VHDL

```
1  L_PRESCALER: entity work.prescaler(rtl)
2  port map (
3  | clk         => clk,
4  | rst         => rst,
5  | ena         => ena,
6  | zero        => '0',
7  | counter_max => X"270F", -- 9999
8  | tick        => ps_tick
9  );
10
```

VHDL (folyt.)

```
11 L_SEC_CNT: process ( clk, rst ) begin
12 | if ( rst = '1' ) then
13 |   sec_counter <= 0; sec_trigger <= '0';
14 | elsif ( rising_edge(clk) ) then
15 |   if ( ps_tick = '1' ) then
16 |   | if ( sec_counter = 9999 ) then
17 |   |   sec_counter <= 0; sec_trigger <= '1';
18 |   | else
19 |   |   sec_counter <= sec_counter + 1;
20 |   |   sec_trigger <= '0';
21 |   | end if;
22 |   end if;
23 | end if;
24 end process;
```

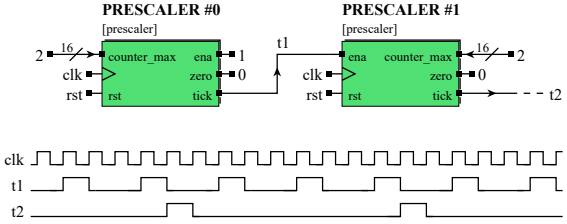
Áramköri struktúra



4-4. ábra. *Az egy másodperces periódusú időzítő megvalósítása órajel-előosztóval*

Az ábrán jól megfigyelhető, hogy az órajel-előosztó használatával az egy másodperces periódusú impulzussorozat előállításához elegendő csupán tízezerig elszámolni a korábbi százmillió helyett, hiszen a számlálás „látszólagos" periódusa az órajel-előosztó jóvoltából a tízezerszeresére nőtt. A 27-bites számláló helyett tehát elegendő egy 14-bites, ami időzítési szempontból kedvezőbb; az összeadóban kialakuló kritikus út késleltetése sokkal kisebb lesz. Fontos hangsúlyozni, hogy a teljes időzítő áramkör erőforrás-felhasználása nem lett kisebb, hiszen az órajel-előosztó maga is egy számláló. Fogalmazhatunk úgy is, hogy a 4-1. ábrán látható egyetlen, 27-bites számláló funkcióját két kisebb számláló sorbakapcsolásával valósítottuk meg.

Az órajel-előosztó leosztásának mértéke a *counter_max* bemenet segítségével futásidőben módosítható. E bemenet esetünkben 16-bites, vagyis a maximális leosztás mértéke $2^{16} = 65.536$. Ha ennél nagyobb mértékű leosztásra van szükségünk, akkor létrehozhatunk ún. kaszkád előosztót az egyfokozatú előosztók 4-5. ábrán látható séma szerinti sorbakapcsolásával.

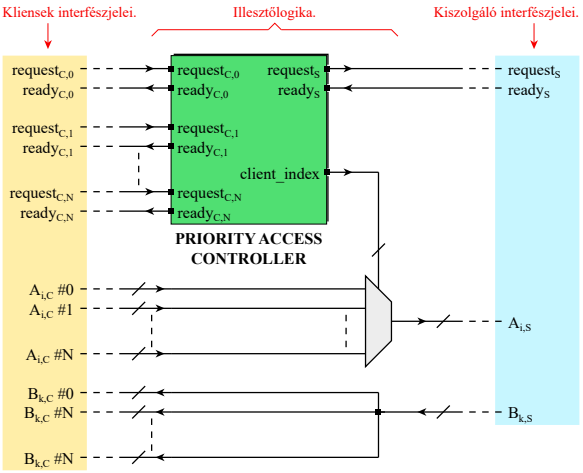


4-5. ábra. *Nagymértékű leosztás létrehozása az órajel-előosztók sorbakapcsolásával*

4.1.2. Prioritásos hozzáférés-vezérlő

Az RTL-tervezésnek a 2. fejezet bevezetőjében említett egyik legfontosabb alapelve a regularitás, ami az egyes almodulok közötti kommunikációs interfészek kialakításában is érvényesül; a különféle részfunkciókat ellátó IP-magokat szinkron kézfogásos vagy MMIO-interfésszel látjuk el. A szinkron kézfogásos interfész specifikációja pont–pont kapcsolatot ír le egyetlen kliens és egyetlen kiszolgáló között. Ebben a kapcsolatban a kliens az 1-4. ábrán látható hullámformának megfelelően kezdeményezheti a kiszolgáló valamely szolgáltatásának igénybevételét. Az IP-magokat jellemzően egyetlen kliensinterfésszel látjuk el, ugyanakkor összetett rendszerek esetén gyakran előfordul, hogy egy adott kiszolgáló IP-mag egy adott példányának szolgáltatásait – időosztásos alapon – több kliens almodul is igénybe szeretné venni. Ehhez olyan illesztőlogikát kell tervezünk, ami az összes felhasználói kérést képes a kiszolgáló jellegű IP-mag egyetlen kliensinterfészére továbbítani úgy, hogy az időosztásos működés a szereplők számára transzparens legyen; sem a kliensek, sem a kiszolgáló nem tud róla, hogy a tranzakciók immár nem egy direkt kapcsolaton keresztül zajlanak le.

A szinkron kézfogásos interfész specifikációjának általánosított A_i és B_k vektorai az adott kiszolgálóra jellemző méretűek és számosságúak, ami az illesztőlogika funkciójának újrafelhasználható formában való kifejezését eléggé körülményessé tenné. Ehelyett egy olyan megoldást mutatunk be, ami a szinkron kézfogásos interfésznek csak a *request* és *ready* jeleivel dolgozik, ugyanakkor jelzi a beágyazó környezet számára, hogy egy adott kérés melyik felhasználótól származik, így téve lehetővé az interfészhez tartozó adatvektorok multiplexálását (lásd 4-6. ábra). Az IP-magra a továbbiakban prioritásos hozzáférés-vezérlőként hivatkozunk.



4-6. ábra. *Illesztőlogika a kiszolgáló több kliens közötti megosztásához*

A prioritásos hozzáférés-vezérlő időosztásos elvű működéséből következik, hogy az egyes kliensek kérései nem azonnal továbbítódnak a kiszolgáló felé, hiszen egy másik kliens kiszolgálása nem szakítható meg. Ugyanakkor a szinkron kézfogásos interfész specifikációja lehetővé teszi a kliens számára, hogy kérését mindössze egy egyetlen órajelciklus időtartamú pozitív impulzussal jelezze a *request* vonalon. Ahhoz, hogy az éppen elfoglalt kiszolgálót aktuális feladatának elvégzése után értesíthessük az időközben beérkezett új kérésről, az ilyen *request* impulzusokat regisztrálnunk kell egy-egy állapotjelző segítségével. A 4-7. ábra egy erre alkalmas áramköri részletet mutat be.

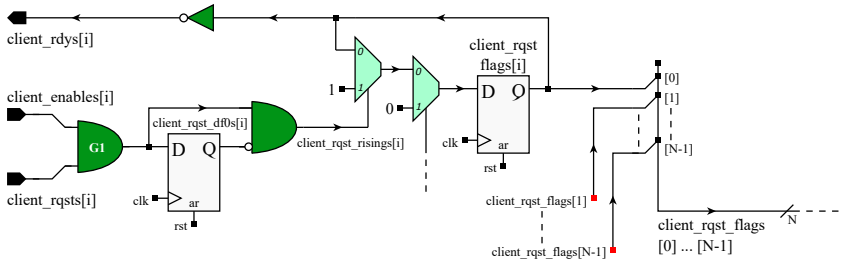
A kliensek számát szintézis-paraméterként valósítjuk meg, és az egyes kliensekkel kapcsolatos teendőink a hozzáférés-vezérlőn belül megegyeznek, ezért a leendő áramkör sok ismétlődő szerkezetet tartalmaz majd. Szerencsére ez nem jelenti azt, hogy a hardverleíró nyelvű modellnek is ismétlődő kódrészletekből kell állnia; az ilyen struktúrák elegánsan kifejezhetők a *for-generate* utasítással. A kliensek *request* jeleit egy-egy felfutóél-detektorral fogadjuk; ha az adott kliens engedélyezett *(client_enables[i] = 1)*, akkor a *request* jel felfutóéle a *client_rqst_flags* vektor megfelelő bitjét egybe állítja, egyúttal – a szinkron kézfogásos interfész előírásainak eleget téve – rögtön alacsony szintre húzza az adott *request* jelhez tartozó *ready* kimenetet. A *client_rqst_flags* vektor szerepét tehát felfoghatjuk úgy is, hogy annak minden bitje egy-egy kliens legutóbb érkezett kérésének „függőben van” állapotát jelzi. Ezeket az állapotjelzőket az adott kérés kiszolgálásának végén persze nullázni kell, ami által egyúttal a megfelelő *ready* jel magas szintűvé válik, jelezve az adott kliensnek a szinkron kézfogásos tranzakció végét. Arra, hogy ennek milyen feltételek esetén kell megtörténnie, később visszatérünk, egyelőre foglalkozzunk azokkal az áramköri részletekkel, amik a regisztrált kéréseket időben sorba rendezve továbbítják a kiszolgáló felé.

A *client_rqst_flags* vektor alapján a kiszolgáló *request* jelét könnyen előállíthatjuk; ha van akár csak egy függőben lévő felhasználói kérés is, vagyis a *client_rqst_flags* vektor bámely bitje magas szintű, akkor azt a kiszolgáló felé azonnal jelezni kell, tehát a *server_rqst* kimenet értékét magas szintűre kell állítani. A kiszolgáló ennek hatására nullába húzza saját *ready* kimenetét, majd amikor elvégezte feladatát, akkor ismét magas szintűre állítja azt. A kiszolgáló *ready* jelének felfutóélét egy éldetektor segítségével a *request* jel nullázására használjuk. E mechanizmust a 4-8. ábrán látható

Verilog

```
1  genvar i; generate // Client request registering
2
3    for ( i = 0; i < N; i++ ) begin
4
5      always @ ( posedge clk, posedge rst ) begin
6        if ( rst == 1 ) begin
7          client_rqst_df0s[i] <= 0; client_rqst_flags[i] <= 0;
8        end else begin
9
10         client_rqst_df0s[i] <= client_rqsts[i] & client_enables[i];
11
12         if ( [...] ) client_rqst_flags[i] <= 0;
13         else if ( client_rqst_risings[i] == 1 ) client_rqst_flags[i] <= 1;
14
15       end
16     end
17
18     assign client_rqst_risings[i] = client_rqsts[i] & ~client_rqst_df0s[i];
19     assign client_rdys[i] = ~client_rqst_flags[i];
20
21   end
22
23 endgenerate
```

Áramköri struktúra



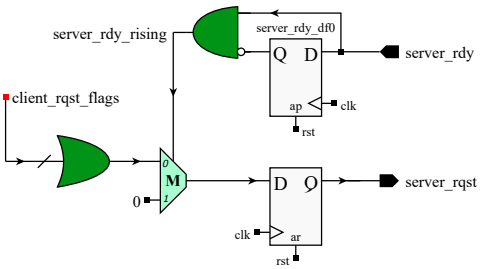
4-7. ábra. *A prioritásos hozzáférés-vezérlő RTL-modellje (1)*

áramköri sémával valósíthatjuk meg. Az *M*-mel jelölt multiplexer biztosítja azt, hogy a *request* jel akkor is legalább egy órajelciklus ideig alacsony szintű legyen, ha a *client_rqst_flags* vektorban a folyamatosan érkező klienskérések miatt állandóan van egyes értékű bit.

VHDL

```
1  always @ ( posedge clk, posedge rst ) begin
2    if ( rst == 1 ) begin
3      server_rqst <= 0; server_rdy_df0 <= 1;
4    end else begin
5
6      server_rdy_df0 <= server_rdy;
7
8      if ( server_rdy_rising == 1 )
9        server_rqst <= 0;
10     else
11       if ( client_rqst_flags != 0 )
12         server_rqst <= 1;
13       else server_rqst <= 0;
14
15   end
16 end
17
18 assign server_rdy_rising = server_rdy &
19                           ~server_rdy_df0;
```

Áramköri struktúra

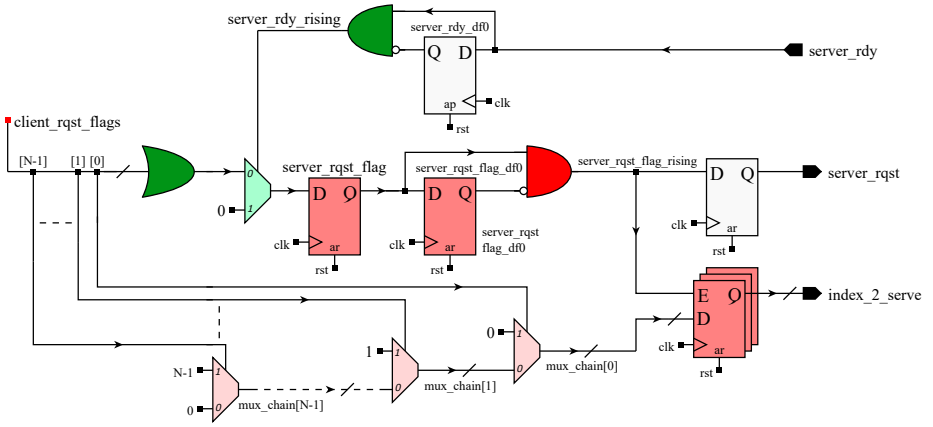


4-8. ábra. *A prioritásos hozzáférés-vezérlő RTL-modellje (2)*

Verilog

```
1  always @ ( posedge clk, posedge rst ) begin
2    if ( rst == 1 ) begin
3      server_rqst_flag <= 0; server_rqst_flag_df0 <= 0;
4      index_2_serve <= 0;
5      [...]
6    end else begin
7
8      server_rdy_df0 <= server_rdy;
9
10     if ( server_rdy_rising == 1 ) server_rqst_flag <= 0;
11     else if ( client_rqst_flags != 0 ) server_rqst_flag <= 1;
12     else server_rqst_flag <= 0;
13
14     server_rqst_flag_df0 <= server_rqst_flag;
15     server_rqst <= server_rqst_flag_rising;
16
17     if ( server_rqst_flag_rising == 1 ) index_2_serve <= mux_chain[0];
18
19   end
20 end
21
22 assign server_rdy_rising = server_rdy & ~server_rdy_df0;
23 assign server_rqst_flag_rising = server_rqst_flag & ~server_rqst_flag_df0;
24
25 // MUX chain
26 genvar j; generate
27   for ( j = 0; j < N-1; j++ ) begin
28     assign mux_chain[j] = (clint_rqst_flags[j] == 1) ? j : mux_chain[j+1];
29   end
30 endgenerate
31 assign mux_chain[N-1] = (client_rqst_flags[N-1] == 1) ? N-1 : 0;
```

Áramköri struktúra



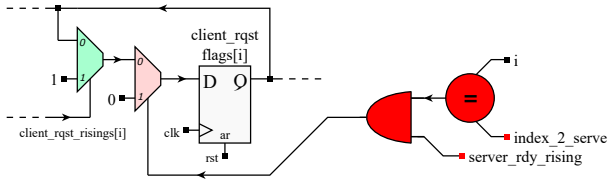
4-9. ábra. *A prioritásos hozzáférés-vezérlő RTL-modellje (3)*

Ezzel a megoldással elértük azt, hogy a kliensek kérései egy-egy különálló kérésként jelenjenek meg a kiszolgáló interfészén, de hogy melyik kiszolgálóoldali kérést melyik kliens idézte elő, arról egyelőre semmi információnk nincs. A kérések forrásának azonosítására egy kombinációs hálózatként működő multiplexerláncot hozunk létre (lásd 4-9. ábra), amely a *client_rqst_flags* vektor minden változásakor megállapítja, hogy annak új értékében melyik a legkisebb, egyes értéket tartalmazó bitpozíció. Ha tehát a *client_rqst_flags* vektor több egyest is tartalmaz, vagyis amikor több kliens kérése is függőben van, akkor a multiplexerlánc a legkisebb helyiértékűt jelöli meg kiszolgálandó indexként, ezt tekinti a legmagasabb prioritásúnak. A *mux_chain[0]* buszon tehát folyamatosan rendelkezésre áll a kiszolgálandó kliens indexe. Egy-egy kiszolgálóoldali szinkron kézfogásos

Verilog

```
genvar i; generate // Client request registering

 for ( i = 0; i < N; i++ ) begin

  always @ ( posedge clk, posedge rst ) begin
   if ( rst == 1 ) begin [...] end
   else begin

    [...]

    if ( server_rdy_rising == 1 && index_2_serve == i ) client_rqst_flags[i] <= 0;
    else if ( [...] ) client_rqst_flags[i] <= 1;

   end
  end

  [...]

 end

endgenerate
```

Áramköri struktúra

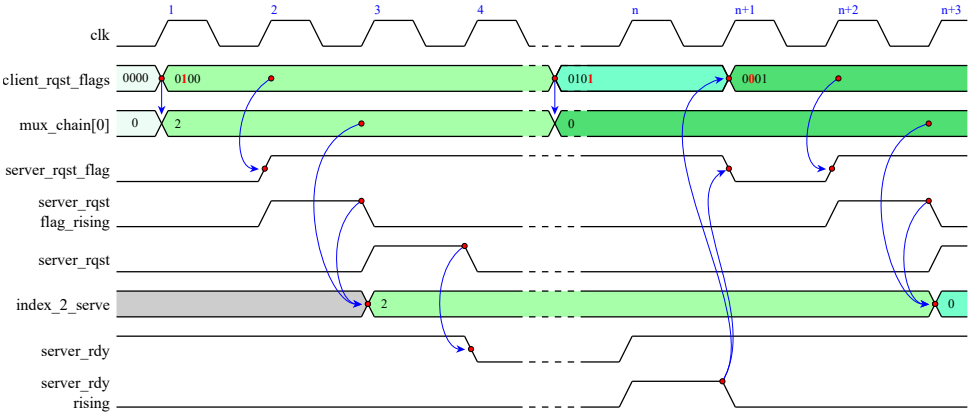


4-10. ábra. *A prioritásos hozzáférés-vezérlő RTL-modellje (4)*

tranzakció közben bármely kliens szabadon kezdeményezhet kéréseket, tehát a *client_rqst_flags* vektorban bármikor újabb egyes értékek bukkanhatnak fel, de a hozzáférés-vezérlő kimenetén az éppen kiszolgált kliens indexe a kiszolgálóoldali tranzakció végéig nem változhat meg, ezért a multiplexerlánc kimenetét a tranzakció indításakor „be kell fagyasztanunk". Ezt a multiplexerlánc kimenetének impulzusszerű mintavételezésével tehetjük meg a következőképpen; a kiszolgáló *request* jelét meghajtó flip-flop *(server_rqst)* elé egy, a kiszolgáló felé küldött kérés függő állapotát jelző belső állapotjelzőt *(server_rqst_flag)* és egy felfutóél-detektort iktatunk be. A kiszolgálóoldali szinkron kézfogásos tranzakció megkezdését a *server_rqst_flag* állapotjelző egyesbe billenése előzi meg. Az e flip-flop kimenetére beépített felfutóél-detektor kimenetét használjuk arra, hogy a multiplexerlánc kimenetének aktuális értékét, vagyis a kiszolgálandó kliens indexét az *index_2_serve* kimeneti regiszterbe töltsük át. Mivel az éldetektor kimenete csupán egyetlen órajelciklus ideig magas szintű, ezért a multiplexerlánc kimenetének mintavételezése valóban impulzusszerű; az *index_2_serve* kimenet értéke a *server_rqst_flag* következő felfutóéléig garantáltan változatlan marad, függetlenül a kliensek kérései alapján folyamatosan változó *client_rqst_flags* vektor változásaitól.

A kiszolgálóoldali szinkron kézfogásos tranzakció végét a *server_rdy* bemenet felfutása jelzi. Ezt a felfutóélt – amellett, hogy a következő kiszolgálóoldali kérést elindítjuk vele – arra is felhasználjuk, hogy nullázzuk a *client_rqst_flags* vektornak a legutóbbi kiszolgálóoldali kérést kiváltó klienshez tartozó bitjét, kifejezve, hogy ez a felhasználói kérés immár kiszolgáltatott (lásd 4-10. ábra). Az állapotjelző törlése egyúttal magas szintre állítja a kliens *ready* jelét is, lezárva a kliensoldali szinkron kézfogásos tranzakciót.

A 4-11. ábrán egy konkrét példán keresztül követhető nyomon a prioritásos hozzáférés-vezérlő működése. E példában négy kliens pályázik a kiszolgáló szolgáltatásainak igénybevételére, ezért a *client_rqst_flags* vektor négybites. Az órajel első felfutóélénél a 2. helyiértékű bit magas szintet vesz fel, vagyis a kettes indexű kliens kérést kezdeményezett. Mivel a multiplexerlánc egy kombinációs hálózat, ezért annak *mux_chain[0]* kimenetén a *client_rqst_flags* vektor megváltozásakor azonnal megjelenik a kiszolgálandó kliens indexe.



4-11. ábra. *A prioritásos hozzáférés-vezérlő működése*

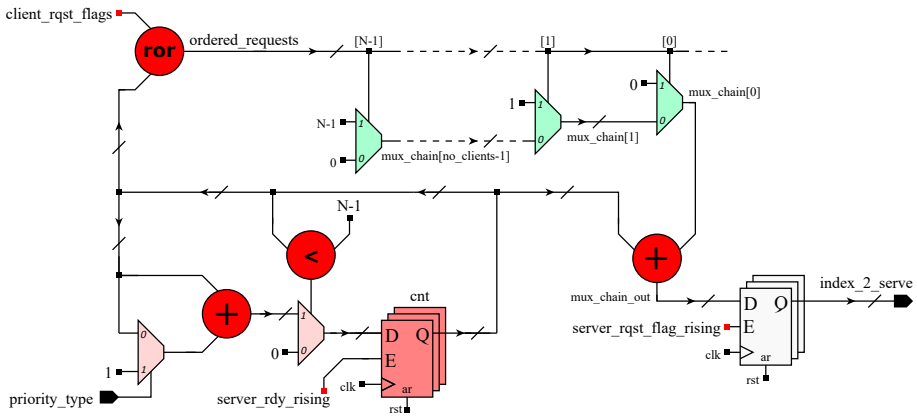
Az órajel második felfutóélénél – mivel a *client_rqst_flags* vektor értéke immár nullától különböző – a *server_rqst_flag* magas szintű lesz, ami egy egyetlen órajel-periódus időtartamú pozitív impulzust idéz elő a *server_rqst_flag_rising* jelen. Ez az impulzus egyrészt a kiszolgáló *request* jelén *(server_rqst)* is megjelenik egy órajelciklus késleltetéssel, másrészt előidézi a multiplexerlánc aktuális értékének áttöltését az *index_2_serve* kimeneti regiszterbe az órajel harmadik felfutóélénél. A szinkron kézfogásos interfész specifikációjának megfelelően működő kiszolgáló a *ready* jelét az órajel negyedik felfutóélénél nullába, majd a kérés kiszolgálása után újra egybe húzza. Időközben a *client_rqst_flags* értékét figyelve azt látjuk, hogy az ismét megváltozik; a nullás helyiértékű bitje magas szintű lesz, ami azt jelzi, hogy időközben a nullás indexű klienstől is érkezett egy kérés. A multiplexerlánc kimenete erre azonnal reagál, de az *index_2_serve* kimenet értéke nem változik meg, hiszen épp folyamatban van a kettes indexű kliens kiszolgálása. A kiszolgáló *ready* jelének felfutása hatására az órajel (n+1)-edik felfutóélénél a *client_rqst_flags* második helyiértékű bitje és a *server_rqst_flag* is nullázódik. Ez utóbbi állapotjelző csupán egyetlen órajelciklus ideig lesz nulla értékű, hiszen a *client_rqst_flags* vektor nulladik helyiértékű bitje időközben magas szintű lett. Az (n+2)-edik órajel-felfutóélnél tehát újabb kiszolgálóoldali kérés kezdeményezése indul meg, amelyben a kiszolgálandó kliens indexe nulla.

Ebben a formájában a hozzáférés-vezérlő csak erős jóindulattal hívható „prioritásosnak”, hiszen éppen a kliensek közötti prioritás szempontjából elég rugalmatlan. Mivel az egyes kliensek kéréseinek a *client_rqst_flags* vektor bitjeire való leképezése állandó, és a multiplexerlánc mindig e vektor legkisebb helyiértékű egyes bitjének indexét jelöli ki a következő kiszolgálandó kliens azonosítójaként, ezért a működés fix prioritást valósít meg; minél kisebb egy kliens indexe, annál nagyobb a prioritása. Ez a megoldás ugyan sok alkalmazásban megfelelő – sőt, akár kifejezetten kívánatos is lehet –, de magában rejti az alacsonyabb prioritású kliensek kiéheztetésének veszélyét, vagyis azt,

Verilog

```
always @ ( posedge clk, posedge rst ) begin
  if ( rst == 1 ) begin cnt <= 0; [...] end
  else begin
    |
    | if ( server_rdy_rising == 1 ) // Priority variation counter
    |   if ( cnt < N-1 ) begin
    |     if ( priority_type == 1 ) cnt <= cnt + 1;
    |   end else cnt <= 0;
    |
    | [...]
    |
    | if ( server_rqst_flag_rising == 1 ) index_2_serve <= mux_chain_out;
    |
  end
end

assign ordered_requests <= (client_rqst_flags << (N-cnt)) | (client_rqst_flags >> cnt); // rotate right
assign server_rdy_rising = server_rdy & ~server_rdy_df0;
assign server_rqst_flag_rising = server_rqst_flag & ~server_rqst_flag_df0;

genvar j; generate // MUX chain
  for ( j = 0; j < N-1; j++ ) begin
    assign mux_chain[j] = (ordered_requests[j] == 1) ? j : mux_chain[j+1];
  end
endgenerate
assign mux_chain[N-1] = (ordered_requests[N-1] == 1) ? N-1 : 0;
assign mux_chain_out = mux_chain[0] + cnt;
```

Áramköri struktúra



4-12. ábra. *A prioritásos hozzáférés-vezérlő RTL-modellje (5)*

hogy a magas prioritású kliensek gyakori kérései miatt az alacsonyabb prioritásúak kiszolgálására sosem kerül sor. A hozzáférés-vezérlőt egy igazságosabb prioritáskezelés irányába fejlesztjük tovább.

Ahogy korábban említettük, az eddig tárgyalt megoldásban a fix prioritás annak köszönhető, hogy egy adott kliens kérései a multiplexerlánc egy adott fokozatának kiválasztó-bemenetére képződnek le. A nullás indexű klienst képviselő állapotjelző mindig a multiplexerlánc nulladik fokozatának kiválasztó-bemenetét hajtja, az egyes indexűé az egyes fokozatét és így tovább. E leképezés dinamikussá tételével egyúttal a kliensek közötti prioritási sorrend is változtatható lenne. Az időben változó leképezésre a 4-12. ábrán láthatunk példát.

Ebben a megoldásban a *client_rqst_flags* vektor nem közvetlenül, hanem egy logikai jobbra forgatás *(ror: **ro**tate **r**ight)* műveleten keresztül képződik le a multiplexerek kiválasztó bemeneteire, tehát a multiplexerlánc immár nem a *client_rqst_flags* vektorban, hanem az *ordered_requests* vektorban

keresi a legkisebb helyiértékű egyes bitet. A forgatás mértékét egy számlálóval minden kiszolgálóoldali tranzakció végén eggyel növeljük. Mivel a multiplexerlánc kimenetén továbbra is mindig a legkisebb helyiértékű egyes bit indexe szerepel, de ez az index immár nem az ugyanezen indexű klienst képviseli, ezért az *index_2_serve* kimenet értékének megállapításakor azt is figyelembe kell vennünk, hogy az eredeti állapothoz képest hány forgatást hajtottunk eddig végre. A *mux_chain_out* kimenetű összeadó tehát azt számítja ki, hogy adott pillanatban az *ordered_requests* vektor legkisebb helyiértékű egyes bitjének indexe melyik kliens kérését reprezentálja. A kliensek közötti prioritás négy kliens esetén a 4-1. táblázatban szemléltetett módon változik.

A számláló értéke	*client_rqst_flags* bitjei	*ordered_requests* bitjei	Kliens index	0	1	2	3
0	$C_3C_2C_1C_0$	$C_3C_2C_1C_0$	Prioritási sorrend*	3	2	1	0
1		$C_0C_3C_2C_1$		0	3	2	1
2		$C_1C_0C_3C_2$		1	0	3	2
3		$C_2C_1C_0C_3$		2	1	0	3

4-1. táblázat. *A kliensek közötti prioritási sorrend változása a számláló értékének változásakor*
*C_n: Az n-edik kliens kérését képviselő állapotjelző. * A nagyobb szám magasabb prioritást jelent*

A kliensek számának megfelelő mennyiségű növelés után a számláló ismét nulla értéket vesz fel, ekkor az eredeti prioritási sorrend áll vissza. Ezt a prioritáskezelési sémát „round-robin" prioritásnak nevezzük. Ez a megoldás megakadályozza az alacsony prioritású kliensek kiéheztetését, hiszen azok minden kiszolgálóoldali tranzakció után eggyel magasabb prioritásúvá válnak, miközben az éppen kiszolgált kliens a prioritási sor végére kerül.

A 4-12. ábra 7. sorában látható, hogy a számláló értékének növelése csak akkor következhet be, ha a *priority_type* bemenet értéke egy, ellenkező esetben a számláló a működés teljes időtartama alatt nulla lesz, amivel az eredeti, fix prioritáskezelést kapjuk vissza. Így tehát nemcsak a kliensek közötti prioritási sorrend, de maga a prioritáskezelési séma is futásidőben módosítható.

4.2. On-chip adattárolás

Az alcímben szereplő „on-chip" kifejezés arra utal, hogy ebben az alfejezetben kifejezetten olyan adattároló megoldásokkal foglalkozunk, amelyek az FPGA belső erőforrásait (LUT-jait, flip-flopjait és blokk-RAM moduljait) használják ideiglenes, illékony adattárolási funkciót ellátó áramkörök létrehozására (vö. 4.4.1., 4.4.2. és 4.4.3. alfejezet; külső memória-áramkörök illesztése). Az egyszerű D flip-flopokból felépített regisztertömbök (lásd DRMSz: 4.5.4. pont) is ebbe a kategóriába sorolhatóak ugyan, de gyakorlati hasznuk kedvezőtlen erőforrás-kihasználási tulajdonságaikból adódóan elég korlátozott, ezért ezekkel itt nem foglalkozunk. Kifejezetten olyan megoldásokat mutatunk be, amelyek FPGA gyártók és áramkörcsaládok széles skáláján használhatóak a belső blokk-RAM modulok képességeinek kiaknázására. Az újrafelhasználhatóságra és technológiafüggetlenségre való törekvésünk szempontjából némi nehézséget okoz, hogy a különféle FPGA gyártók és eszközcsaládok blokk-RAM moduljai áramköri szinten eltérő működésűek lehetnek. A helyzetet tovább nehezíti, hogy a szintézis-szoftverek is erősen befolyásolják azt, hogy e modulok képességeit milyen HDL kódolási szabályok betartásával lehet „előcsalogatni". Az egyes gyártók a saját eszközeikhez rendszerint rendelkezésre bocsátják azok részletes dokumentációját, valamint egy sor ajánlást azzal kapcsolatban, hogy a hardverleíró nyelvű modelleket

hogyan kell elkészíteni az egyes viselkedési minták megvalósításához. Az újrafelhasználhatóságra való törekvés ennek ellenére nem veszett ügy, készíthetők olyan leírások, amelyek a blokk-RAM moduloknak kizárólag a legalapvetőbb jellegzetességeire támaszkodnak, így gyakorlatilag bármely gyártó és bármely eszközcsalád esetén használhatók.

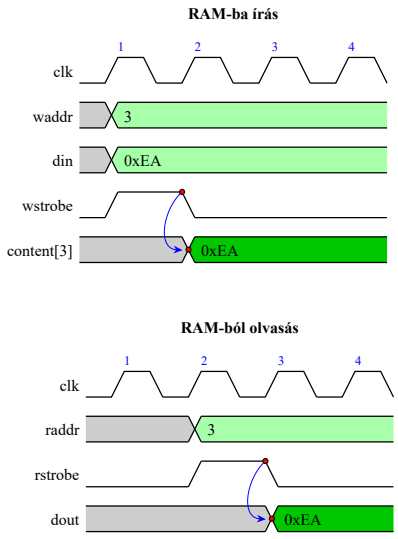
4.2.1. Véletlen elérésű tárolómodulok (RAM-ok)

A legegyszerűbb, blokk-RAM modulok segítségével is megvalósítható memória IP a *singleclock_dualport_ram*. Ez lényegében bináris vektorok tömbje, amelyben a vektorok mérete és száma is szintézis-paraméter. A modul egyetlen beíró és egy attól független kiolvasó interfészt biztosít a felhasználónak. A modul Verilog-leírását és interfészeinek viselkedését a 4-13. ábra szemlélteti.

Verilog

```
1  module singleclock_dualport_ram
2  #(parameter aw, dw)
3  (
4    input  logic clk, wstrobe, rstrobe,
5    input  logic [aw-1:0] waddr,
6    input  logic [aw-1:0] raddr,
7    input  logic [dw-1:0] din,
8    output logic [dw-1:0] dout
9  );
10
11   logic [dw-1:0] content [2**aw-1:0];
12
13   always @ ( posedge clk )
14   begin
15     if ( rstrobe == 1'b1 ) dout <= content[raddr];
16     if ( wstrobe == 1'b1 ) content[waddr] <= din;
17   end
18
19 endmodule
```

Hullámforma

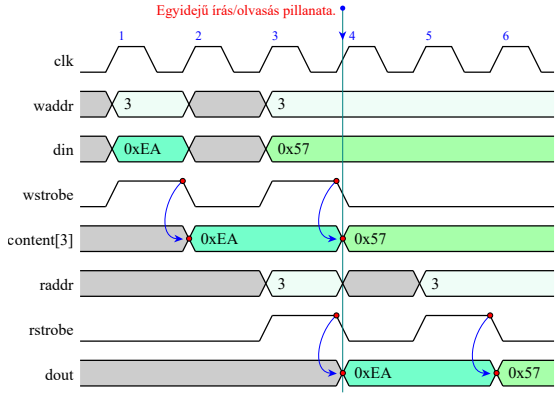


4-13. ábra. *A singleclock_dualport_ram IP Verilog modellje és interfészeinek viselkedése*

A *singleclock_dualport_ram* IP két fontos tulajdonsága, hogy szinkron működésű – tehát adatkimenete csak az órajel felfutóélénél képes megváltozni –, illetve hogy egyáltalán nincs *reset* bemenete, tartalma és adatkimenete, tehát szimulációs környezetben és egy konkrét FPGA áramkörben megvalósítva – annak tápfeszültségre kapcsolásakor – is inicializálatlan. Ezek a tulajdonságok valamelyest nyilván korlátozzák a felhasználót, ugyanakkor elfogadva ezeket a korlátokat gyakorlatilag biztosak lehetünk benne, hogy RTL-modelljeink technológiafüggetlenek lesznek, ti. ezt a működést bármely gyártó bármely eszköze képes biztosítani.

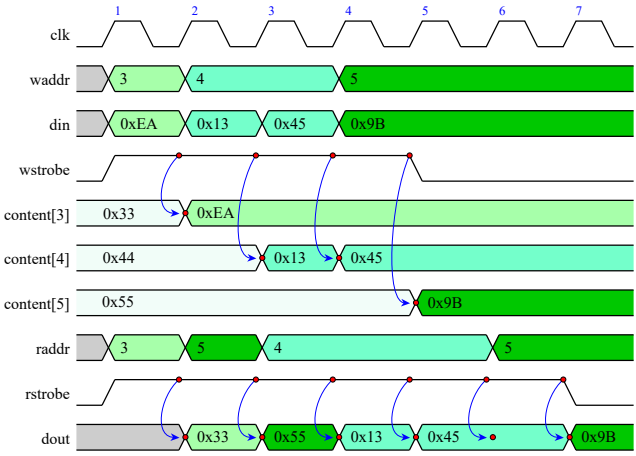
Egyidejű írás/olvasás ◆ A két külön interfészen írható és olvasható memóriák esetén az egyidejű írás/olvasás kérdése külön figyelmet érdemel. A különböző FPGA eszközök blokk-RAM moduljai rengeteg működési módot kínálnak, amelyek jellemzőinek – és persze HDL-modellezésük mikéntjének – megértése kritikus a szimulációban és a valós áramkörben tapasztalt viselkedés eltérésének elkerülése érdekében. A *singleclock_dualport_ram* IP-mag ebből a szempontból is a lehető

legegyszerűbb változat megvalósítására törekszik; ha a memóriamodul egy rekeszét ugyanabban az órajelciklusban írjuk és olvassuk, akkor a 4-13. ábrán látható HDL-modellből szintetizált áramkör úgy viselkedik, mintha a megszólított memóriarekesz egyszerű D flip-flopokból álló vektor lenne, tehát a beírás pillanatában a memóriamodul adatkimenetén a megcímzett rekesznek az előző értéke látható. Az egyidejű írás/olvasás időzítését a 4-14. ábra mutatja be.



4-14. ábra. *Egyidejű írás/olvasás a singleclock_dualport_ram IP esetén*

Ezt a működési módot az FPGA gyártók „READ FIRST” üzemmódnak nevezik, utalva arra, hogy az egyidejű írás/olvasás esetén a kimeneti port úgy viselkedik, mintha az olvasás előbb történne meg, mint az írás. Ez valóban a „szokásos”, flip-flop-szerű viselkedés; a memóriarekesz írása egyazon szimulációs ciklusban, az órajelnek egyazon felfutóéle hatására történik meg, mint az olvasás, vagyis a kimeneti adatbusz frissítése. Ez a kimeneti adatbusz felfogható a memóriarekesz után kapcsolt kimeneti regiszternek is, amely a bemenetét meghajtó, véges késleltetésű vezetékezés miatt a kiolvasás pillanatában a memóriarekesznek még csak az előző értékét láthatja.[93]



4-15. ábra. *A singleclock_dualport_ram modul folyamatos írása/olvasása*

[93] Vö. DRMSz: 4.4.4. alfejezet, VHDL delta-késleltetés, Verilog blokkoló vs. nem-blokkoló értékadás.

Folyamatos írás/olvasás ◆ A 4-13. és 4-14. ábrák időzítési diagramjain egyaránt látható, hogy a *singleclock_dualport_ram* modulnak nincs szüksége több órajelciklusra ahhoz, hogy a hozzá intézett kéréseket kiszolgálja, ezért nincs akadálya annak, hogy az írási és olvasási kéréseket késleltetés nélkül fűzzük egymás után. Ezt a *strobe* jelek magas szinten tartásával és a címbemenetek órajelciklusonkénti változtatásával érhetjük el, ahogy azt a 4-15. ábrán is láthatjuk.

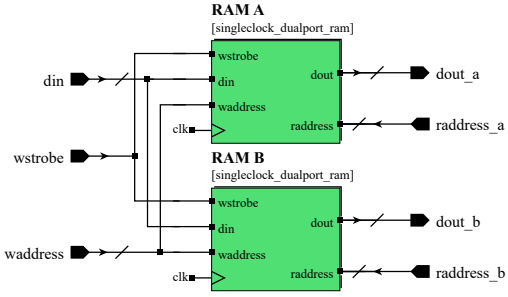
Gyakran nagyon hasznos, ha egy memóriamodulból egyszerre nemcsak egy, hanem két rekesz tartalmát is ki tudjuk olvasni. A *singleclock_dualport_ram* IP-magot könnyen továbbfejleszthetjük úgy, hogy egy további kiolvasó interfészt adunk hozzá a 4-16. ábrán látható VHDL-leírásnak megfelelően. Ezt a memóriamodult *singleclock_tripleport_ram*-nak hívjuk.

VHDL

```
entity singleclock_tripleport_ram is
  generic ( aw: integer range 2 to 16; dw: integer range 4 to 128 );
  port (
    clk, rstrobe_a, rstrobe_b, wstrobe: in  std_logic;
    raddr_a, raddr_b, waddr:            in  std_logic_vector (aw-1 downto 0);
    dout_a, dout_b:                     out std_logic_vector (dw-1 downto 0);
    din:                                in  std_logic_vector (dw-1 downto 0)
  );
end entity singleclock_tripleport_ram;

architecture rtl of singleclock_tripleport_ram is

  type memory_content_t is array (natural range <>) of std_logic_vector;
  signal content: memory_content_t (0 to 2**aw-1)(dw-1 downto 0);

begin

  L_RAM: process ( clk ) begin
    if ( rising_edge(clk) ) then
      if ( rstrobe_a = '1' ) then dout_a <= content(to_integer(unsigned(raddr_a))); end if;
      if ( rstrobe_b = '1' ) then dout_b <= content(to_integer(unsigned(raddr_b))); end if;
      if ( wstrobe = '1' ) then content(to_integer(unsigned(waddr))) <= din; end if;
    end if;
  end process;

end architecture rtl;
```

4-16. ábra. *A singleclock_tripleport_ram IP VHDL-modellje*

Ebben a leírásban olyan működést fogalmaztunk meg, amelyben a memóriamodul az órajelnek egyazon felfutóéle hatására három külön műveletet végez el; egy beírást és két kiolvasást. Ezt a viselkedést a *singleclock_dualport_ram* IP két példányának 4-17. ábrán szemléltetett összekapcsolásával is elérhetjük.



4-17. ábra. *A singleclock_tripleport_ram viselkedésének megvalósítása a singleclock_dualport_ram két példányával*

A szintézis-szoftverek ugyancsak az FPGA-k blokk-RAM moduljainak többszörözésével valósítják meg a *singleclock_tripleport_ram* IP viselkedését, tehát számítanunk kell arra, hogy a fizikailag felhasznált memóriarekeszek száma duplája lesz annak, amit logikailag meg tudunk címezni. Természetesen nincs akadálya annak, hogy további kiolvasó interfészeket is adjunk a leíráshoz, ekkor azonban a felhasznált memóriablokkok száma minden újabb kiolvasó interfésszel eggyel nő.

4.2.2. FIFO-tároló

Az egyik legszélesebb körben alkalmazott memóriatípus a FIFO (**F**irst-**I**n **F**irst-**O**ut) tároló, amelynek két szolgáltatása a *push*, amely során egy bejegyzést helyezhetünk el a tárban, illetve a *pop*, amellyel kiolvashatjuk azt a bejegyzést, amit legrégebben töltöttünk bele. Egy ilyen memóriamodul Verilog-leírását mutatja be a 4-18. ábra.

Verilog

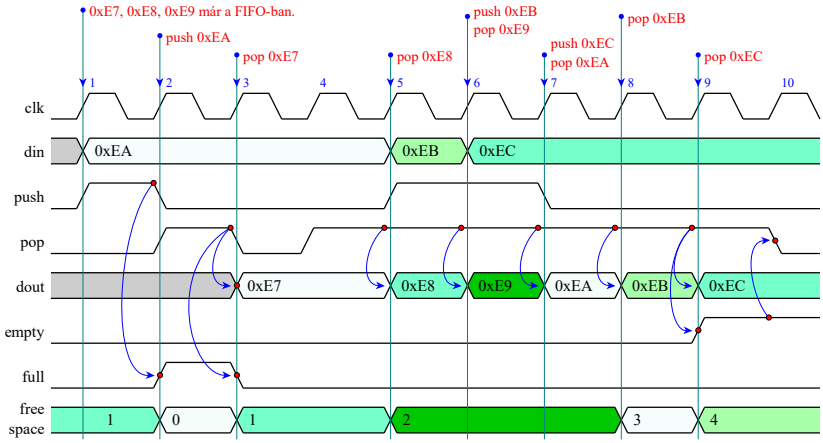
```
module singleclock_fifo #(parameter aw, dw)
(
  input  logic clk, rst, push, pop,
  output logic full, empty,
  output logic [dw-1:0] dout,
  input  logic [dw-1:0] din,
  output logic [aw:0] free_space
);

  logic [dw-1:0] content [2**aw-1:0];
  logic [aw-1:0] read_pointer;
  logic [aw-1:0] write_pointer;

  always @ (posedge clk, posedge rst) begin
    if ( rst == 1 ) begin
      write_pointer <= 0; read_pointer <= 0; full <= 0; empty <= 1; dout <= 0; free_space <= 2**aw;
    end else begin

      if ( push && !full ) begin
        content[write_pointer] <= din; write_pointer = write_pointer + 1; free_space = free_space - 1;
        if ( write_pointer == read_pointer ) full = 1;
        else full = 0;
        empty = 0;
      end

      if ( pop && !empty ) begin
        dout <= content[read_pointer]; read_pointer = read_pointer + 1; free_space = free_space + 1;
        if ( write_pointer == read_pointer ) empty = 1;
        else empty = 0;
        full = 0;
      end

    end
  end

endmodule
```

4-18. ábra. *FIFO-tároló Verilog modellje*

Megvalósítását tekintve a 4-18. ábrán látható *singleclock_fifo* modul valójában egy egyedi módon címzett véletlen elérésű memória. A felhasználó sem a beíráshoz *(push)*, sem a kiolvasáshoz *(pop)* nem ad meg memóriacímet, hanem azt maga a FIFO modul számítja ki mindkét esetben egy-egy számláló (4-18. ábra: *write_pointer*, *read_pointer*) segítségével. *Push* művelet esetén az egyik, *pop* művelet esetén a másik számláló értékét növeli meg eggyel. A számlálók csökkentésére sosem kerül sor, azok a maximális értékük elérése utáni első műveletkor egyszerűen átfordulnak, aminek az az eredménye, hogy az újabb adatok beírására tulajdonképpen azokat a rekeszeket használjuk fel, ahonnan időközben a felhasználó már kiolvasta az információt. Ezt a megoldást cirkuláris

buffernek nevezzük. Ebben a megvalósításban a teli *(full)* és üres *(empty)* állapotjelzők karbantartása egyszerű; ha a *push* művelet következményeként megnövelt *write_pointer* értéke egyenlővé válik a *read_pointer*-ével (4-18. ábra: 21. sor), akkor a tároló megtelt, ha pedig a *pop* művelet által megnövelt *read_pointer* esetén áll elő ugyanez a helyzet (4-18. ábra: 28. sor), az a tároló kiürülését jelenti. A FIFO-tároló a benne lévő szabad bejegyzéshelyek számát is a felhasználó rendelkezésére bocsátja a *free_space* kimenetén keresztül.

Időzítési szempontból a FIFO-tároló a véletlen elérésű memóriamodulhoz hasonlóan működik. A *push* és *pop* jelek ugyanúgy használandók, mint a RAM *wstrobe* és *rstrobe* jelei; a *push/pop* bemenetek magas szintre állításakor a beírás, illetve a kimeneti adatbusz és a *full/empty* jelzések frissítése rögtön az órajel következő felfutóélénél megtörténik. A 4-19. ábra a FIFO modul időzítési modelljét mutatja be egy 4 bejegyzés tárolására képes példány esetén, amelybe előzetesen 3 adatot már beírtunk.



4-19. ábra. *A singleclock_fifo modul időzítési modellje*

Az időzítési modellen azt is megfigyelhetjük, hogy a FIFO modul is használható „folyamatos üzemben", vagyis a beírás és kiolvasás műveletek a maximális sebesség elérése érdekében szorosan egymás után fűzhetők a *push/pop* jelek magas szinten tartásával.

4.2.3. Memória IP-k kézfogásos burkolói

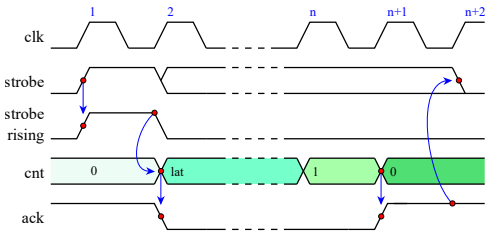
Az eddig bemutatott RAM és FIFO tárolómodulok számára az órajelnek egyetlen felfutóéle is elegendő feladatuk elvégzéséhez, mivel működésük az FPGA-k belső blokk-RAM moduljaira épül. Általános esetben egy memóriamodul nem feltétlenül képes erre; ha a beírt/kiolvasott adatot kódolni/dekódolni kell (pl. EDAC-védelemmel ellátott memória IP-k esetén), vagy maga a tárolás egy külső, lassú működésű eszközben történik, akkor a memória IP interfészén szükségünk van a szinkron kézfogásos interfész által biztosított visszacsatolási mechanizmusra. A lokalitás és modularitás jegyében a memóriamodulok beágyazó környezetét – pl. a memóriába való írást és az onnan való olvasást végző állapotgépeket – ezért célszerű kézfogásos interfészű memóriákat feltételezve elkészíteni, így a rendszer a gyors, egyetlen órajelciklus késleltetésű, és a bonyolult belső megvalósítású, lassú memória IP-kkal is képes lesz együttműködni. Ennek a megoldásnak a demonstrálása

céljából a RAM és FIFO modulokhoz elkészítünk egy kiegészítő logikát, amelyet felhasználva a szinkron kézfogásos interfész-specifikációnak megfelelő viselkedésű memóriamodulokat kaphatunk úgy, hogy azok késleltetése paraméterként beállítható. Így a memóriát is tartalmazó teljes rendszer számítási teljesítményének függése a benne lévő memóriák sebességétől könnyen vizsgálhatóvá válik. A kiegészítő logika VHDL-leírása és RTL kapcsolási sémája a 4-20. ábrán látható.

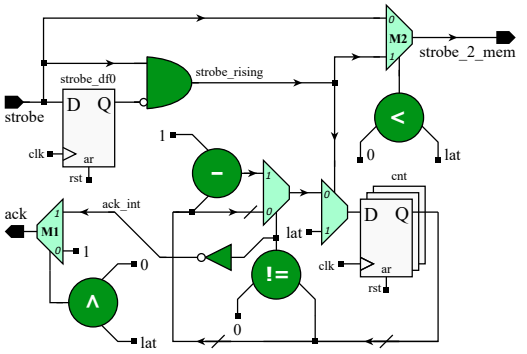
VHDL

```
1  L_HS_LOGIC: process ( clk, rst ) begin
2    if ( rst = '1' ) then
3      strobe_df0 <= '0'; cnt <= 0;
4    elsif ( rising_edge(clk) ) then
5
6      strobe_df0 <= strobe;
7
8      if ( strobe_rising = '1' ) then cnt <= lat;
9      elsif ( cnt /= 0 ) then cnt <= cnt - 1;
10     end if;
11
12   end if;
13 end process;
14
15 strobe_rising <= strobe and not strobe_df0;
16 ack_int <= '0' when cnt /= 0 else '1';
17 ack <= ack_int when lat > 0 else '1';
18 strobe_2_mem <= strobe_rising when lat > 0
19                else strobe;
```

Hullámforma



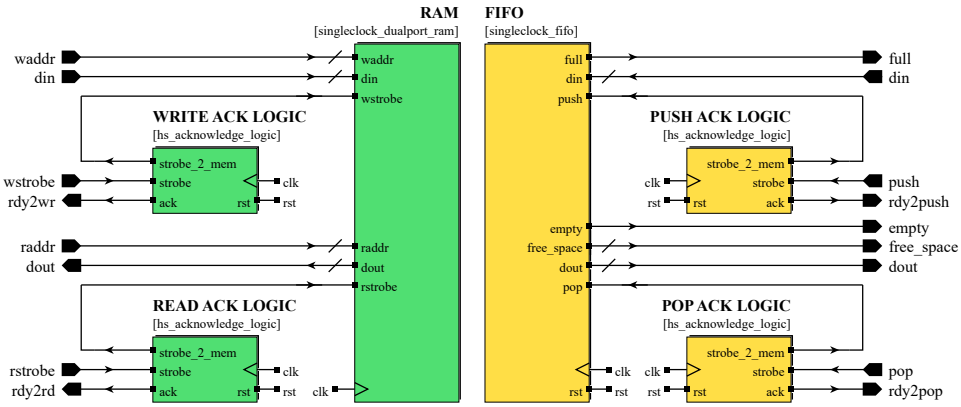
Áramköri struktúra



4-20. ábra. *A memória IP-k szinkron kézfogásos interfész-specifikáció szerinti működését megvalósító logika*

A megoldás alapvetően egy lefelé számláló áramkör, amelybe a *strobe* jel felfutóélének detektálásakor az órajelciklusokban kifejezett késleltetés értékét *(lat)* töltjük be, majd a számláló értékét az órajel minden felfutóélénél eggyel csökkentjük mindaddig, amíg az el nem éri a nullát. A kliens modulnak szóló visszacsatolást jelentő *ack* kimenetet a számláló aktuális értékéből egy dekóderrel állítjuk elő úgy, hogy az mindaddig alacsony szintű legyen, ameddig a lefelé számlálás folyik, így imitálva a memóriamodul belső késleltetését. Ha a kívánt késleltetés értékét képviselő *lat* paramétert nullára állítjuk, akkor az *M1* multiplexer – a szinkron kézfogásos interfész-specifikáció e szélsőséges esetének megfelelően (lásd 1-5. ábra) – magas szinten tartja az *ack* kimenetet, míg az *M2* multiplexer közvetlenül továbbítja a kliens *strobe* jelét a belső memóriának.

A 4-20. ábra áramköre a véletlen elérésű tároló IP-kkal és a FIFO-tároló IP-kkal is használható a 4-21. ábrán szemléltetett módon.



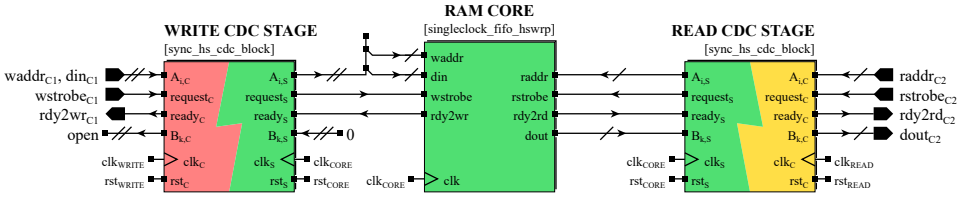
4-21. ábra. *RAM és FIFO modulok kézfogásos logikával kiegészítve*

A kézfogásos logikát is tartalmazó, kiegészített IP-magokra a *„_hswrp" (**h**and-**s**hake **wr**a**p**per)* utótaggal hivatkozunk: *singleclock_dualport_ram_hswrp*, *singleclock_fifo_hswrp*. A kifejezés arra utal, hogy az így kapott IP-mag egy, az eredeti funkciót kiegészítő, azt új képességekkel felruházó burkolómodul.

4.2.4. CDC a tároló IP-magok interfészein

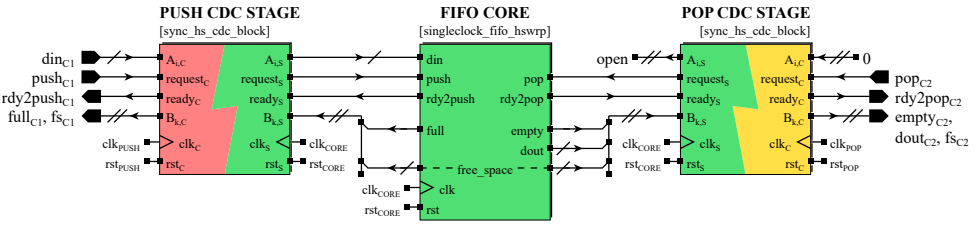
A *singleclock_dualport_ram_hswrp* és a *singleclock_fifo_hswrp* IP-magok beíró és kiolvasó interfészei szinkron kézfogásos interfészek. Ha ezeknek az interfészeknek a kliensei a memóriamoduléhoz képest aszinkron órajeltartományban működnek, akkor használhatjuk a szinkron kézfogásos interfészhez kifejlesztett, általános vektorszinkronizáló fokozatot. Fontos hangsúlyozni, hogy ezeknek a memóriamoduloknak a késleltetése a kézfogásos logika által megvalósított paraméter, értéke nullára is állítható, ezért a 2.1. pontban bemutatott CDC áramkörsémának azt a változatát kell használnunk, ami ennek az esetnek a kezelésére is alkalmas (lásd 2-3. ábra).

A 4-22. ábra szemlélteti, hogy az univerzális vektorszinkronizáló fokozatok általánosított A_i és B_k vektorai a RAM-modullal összekapcsolva annak adat- és címbuszait foglalják magukban. Az ábra az órajeltartományokat különböző színekkel érzékelteti; látható, hogy általános esetben a beírást és a kiolvasást egy-egy külön kliens végzi, mindkettő a saját órajeltartományából.



4-22. ábra. *Órajel-szinkronizáló fokozatok a véletlen elérésű memória interfészein*

A FIFO IP esetén a helyzet némileg bonyolultabb. Itt a vektorszinkronizálók A_i és B_k vektorai az adatbuszokon kívül a *full* és *empty* állapotjelzőket, valamint a szabad helyek számát hordozó *free_space* vektort is magukban foglalják (lásd 4-23. ábra).



4-23. ábra. *Órajel-szinkronizáló fokozatok a FIFO modul interfészein*

4.2.5. CDC dual-clock blokk-RAM modulokkal

Az előző pontban bemutatott, az IP-magok és klienseik aszinkronitásának kezelésére adott megoldás általánosan használható nemcsak memória jellegű, hanem bármilyen funkciójú IP-mag, és nemcsak szinkron kézfogásos, hanem egyéb interfészek esetén is. Kifejezetten memória jellegű IP-k esetén ezt a problémát egy teljesen más megközelítéssel is megoldhatjuk; kihasználhatjuk azt, hogy az FPGA-k blokk-RAM moduljai beíró és kiolvasó interfészeik részeként egy-egy külön órajelbemenetet bocsátanak rendelkezésünkre, amelyek akár egymáshoz képest aszinkron órajelekkel is meghajthatók. Így tulajdonképpen a blokk-RAM modul maga válhat órajelszinkronizáló fokozattá. A 4-24. ábra a dualport RAM modulnak azt a változatát mutatja be, ami a beíráshoz és a kiolvasáshoz egy-egy külön órajelet használ.

Verilog

```
module dualclock_dualport_ram #(parameter aw, dw)
(
  input  logic wclk, rclk, wstrobe, rstrobe,
  input  logic [aw-1:0] waddr,
  input  logic [aw-1:0] raddr,
  input  logic [dw-1:0] din,
  output logic [dw-1:0] dout
);

  logic [dw-1:0] content [2**aw-1:0];

  always @ ( posedge wclk ) if ( wstrobe == 1 ) content[waddr] <= din;

  always @ ( posedge rclk ) if ( rstrobe == 1 ) dout <= content[raddr];

endmodule
```

4-24. ábra. *Véletlen elérésű dualport memória külön beíró és kiolvasó órajelbemenettel*

A megoldás akár több kiolvasó interfésszel rendelkező RAM modulok esetén is működik, hiszen a szintézis-szoftverek az újabb és újabb hozzáadott kiolvasó interfészek megvalósítását a blokk-RAM modulok többszörözésével oldják meg (lásd 4-17. ábra), tehát minden új kiolvasó interfészhez egy újabb órajelbemenet is tartozik. Így állítható elő a *singleclock_tripleport_ram* modul apró módosításával a *dualclock_tripleport_ram* modul a 4-25. ábrán látható módon.

Verilog

```
entity dualclock_tripleport_ram is
  generic ( aw: integer range 2 to 16; dw: integer range 4 to 128 );
  port (
    wclk, rclk, rstrobe_a, rstrobe_b, wstrobe: in  std_logic;
    raddr_a, raddr_b, waddr:                   in  std_logic_vector (aw-1 downto 0);
    dout_a, dout_b:                            out std_logic_vector (dw-1 downto 0);
    din:                                       in  std_logic_vector (dw-1 downto 0)
  );
end entity dualclock_tripleport_ram;

architecture rtl of dualclock_tripleport_ram is

  type memory_content_t is array (natural range <>) of std_logic_vector;
  signal content: memory_content_t (0 to 2**aw-1)(dw-1 downto 0);

begin

  L_WRITE_RAM: process ( wclk ) begin
    if ( rising_edge(wclk) ) then
      if ( wstrobe = '1' ) then content(to_integer(unsigned(waddr))) <= din; end if;
    end if;
  end process;

  L_READ_RAM: process ( rclk ) begin
    if ( rising_edge(rclk) ) then
      if ( rstrobe_a = '1' ) then dout_a <= content(to_integer(unsigned(raddr_a))); end if;
      if ( rstrobe_b = '1' ) then dout_b <= content(to_integer(unsigned(raddr_b))); end if;
    end if;
  end process;

end architecture rtl;
```

4-25. ábra. *A dualclock_tripleport_ram IP VHDL-modellje*

A blokk-RAM moduloknak ez a speciális tulajdonsága a FIFO-tárolók esetén is kihasználható. A *push* és *pop* műveletet végrehajtó – vagyis a *write/read_pointer* regisztereket kezelő – áramköri részleteket létrehozhatjuk két különálló órajeltartományban, ami a felhasználó szempontjából csak annyit jelent majd, hogy a FIFO IP-nak egy-egy külön órajele van a *push* és *pop* interfészekhez. Természetesen ez azt is maga után vonja, hogy az IP-magon belül a két tartomány közötti adatcserét meg kell oldani. A 4-26. ábra egy ilyen FIFO megoldást mutat be.

Ebben a megvalósításban a *write/read_pointer* regiszterek bitszélessége az adatok tárolására használt belső RAM címbuszának bitszélességénél eggyel nagyobb. Ilyenkor a FIFO teli állapotát jelző *full* állapotjelzőt akkor állítjuk magas szintre, ha a *write_pointer* és a *read_pointer* legmagasabb helyiértékű bitjei különbözők, miközben összes többi bitjük azonos, az üres állapotot jelző *empty* jelzés pedig akkor magas értékű, ha a *write_pointer* és a *read_pointer* azonos értékűek. Mivel mindkét állapotjelző előállításához egyaránt szükségünk van a *write_pointer* és *read_pointer* regiszterekre, ezért mindkét vektort át kell vinnünk a beíró/kiolvasó órajeltartományok határán. Kihasználva azt, hogy mindkét vektor egy-egy felfelé számláló, alkalmazhatunk Gray-szinkronizáló fokozatokat (lásd DRMSz: 6.3.1. alfejezet). A 4-26. ábrán szereplő Verilog modellben magát a Gray-enkódolást és dekódolást egy-egy függvény végzi, ezek kifejtését az 5.5. Függelék tartalmazza. A FIFO-ban lévő szabad helyek számát úgy állapíthatjuk meg, hogy a *read_pointer* értékéből kivonjuk a *write_pointer* értékét, majd az eredményhez hozzáadjuk a teljes kapacitást. Mivel a szabad helyek száma mind a beírást, mind a kiolvasást végző kliens számára hasznos lehet, ezért azt a *push* és *pop* interfészen egyaránt elérhetővé tesszük.

Természetesen nincs akadálya annak, hogy a több órajelbemenetet biztosító, ezáltal órajel-szinkronizáló fokozatként használható RAM és FIFO memóriamodulokat is kiegészítsük a 4-20. ábrán látható logikával, létrehozva azok szinkron kézfogásos burkolóit. Ezeknél a *read/*

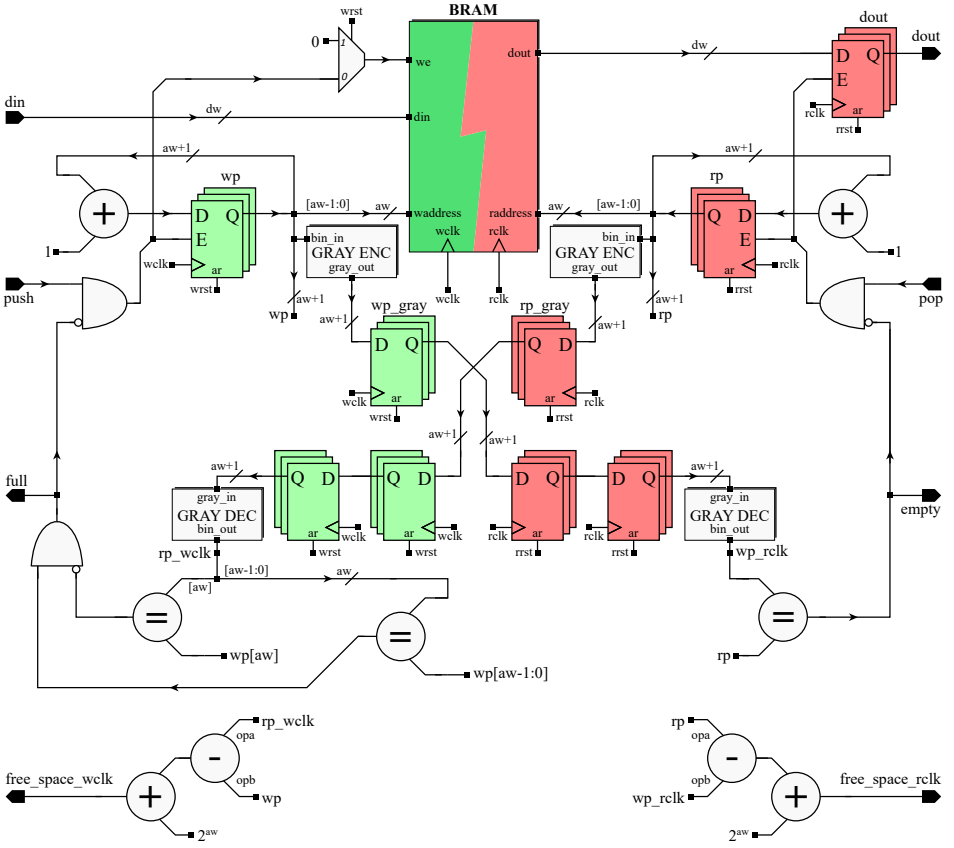
Verilog – a push műveletet megvalósító logika

```
always @ ( posedge wclk, posedge wrst )
begin
  if ( wrst == 1'b1 ) begin
    wp <= 0; wp_gray <= 0;
    rp_gray_df0 <= 0; rp_gray_df1 <= 0;
  end else begin
    if ( push == 1'b1 && full == 1'b0 ) begin
      content[wp[aw-1:0]] <= din;
      wp <= wp + 1;
    end
    wp_gray <= gray_encoder(wp);
    rp_gray_df0 <= rp_gray;
    rp_gray_df1 <= rp_gray_df0;
  end
end

assign full = ((wp[aw-1:0] == rp_wclk[aw-1:0]) &&
               (wp[aw] != rp_wclk[aw])) ?
               1'b1 : 1'b0;
assign rp_wclk = gray_decoder(rp_gray_df1);
assign free_space_wclk = (rp_wclk - wp) + 2**aw;
```

Verilog – a pop műveletet megvalósító logika

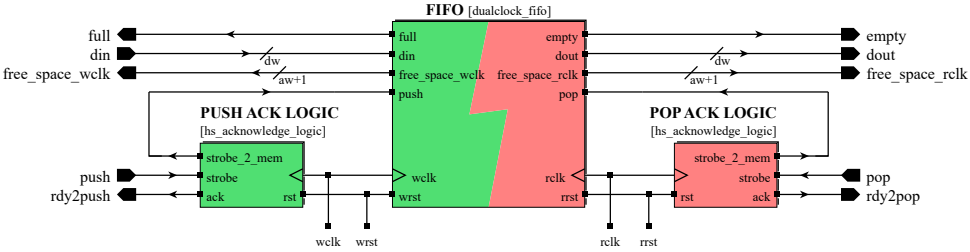
```
always @ ( posedge rclk, posedge rrst )
begin
  if ( rrst == 1'b1 ) begin
    rp <= 0; rp_gray <= 0;
    wp_gray_df0 <= 0; wp_gray_df1 <= 0;
    dout <= 0;
  end else begin
    if ( pop == 1'b1 && empty == 1'b0 ) begin
      dout <= content[rp[aw-1:0]];
      rp <= rp + 1;
    end
    rp_gray <= gray_encoder(rp);
    wp_gray_df0 <= wp_gray;
    wp_gray_df1 <= wp_gray_df0;
  end
end

assign empty = (wp_rclk == rp) ? 1'b1 : 1'b0;
assign wp_rclk = gray_decoder(wp_gray_df1);
assign free_space_rclk = (rp - wp_rclk) + 2**aw;
```

Áramköri struktúra



4-26. ábra. *Az aszinkron dual-clock FIFO modul belső szerkezete*

write_strobe vagy a *push/pop* bemenetekhez tartozó visszacsatoló jeleket előállító kézfogásos logikák külön órajelekről járnak. A 4-27. ábra az így kapott *dualclock_fifo_hswrp* burkolómodul kapcsolási sémáját mutatja be.[94]



4-27. ábra. *A dualclock_fifo_hswrp IP-mag RTL kapcsolási sémája*

Hasonló módon készíthető el a *dualclock_dualport_ram* modul kibővítésével a kézfogásos interfésszel rendelkező, a beírást és a kiolvasást külön-külön órajelhez szinkronizáltan végrehajtani képes *dualclock_dualport_ram_hswrp* almodul, vagy annak két kiolvasó interfésszel rendelkező variánsa, a *dualclock_tripleport_ram_hswrp* almodul is.

4.2.6. A memóriatartalom inicializálása

Visszatekintve a memória jellegű IP-magok RTL-modelljeire, azt láthatjuk, hogy akár véletlen elérésű memóriáról, akár FIFO-ról legyen szó, a tároló funkciót a HDL-ekben logikai vektorok tömbjeként írjuk le. A leírás tehát egyszerűen flip-flopokból álló vektorok halmazának tűnik, a szintézis-szoftverek azonban – szándékunknak megfelelően – ezeket a modelleket az FPGA-kban lévő blokk-RAM modulok segítségével valósítják meg. Ennek egyik alapvető feltétele, hogy a HDL-modellekben nem használhatjuk a regiszterek esetén megszokott *reset* jelet a tömbök tartalmának inicializálására, hiszen ezzel olyan viselkedést írnánk le, amit a blokk-RAM modulok fizikai kialakításukból adódóan nem tudnak megvalósítani, az SRAM-celláknak ui. nincs *reset* bemenete. Ez egyúttal azt is jelenti, hogy a memória IP-k tartalma a HDL szimulációk kezdetén, illetve a szintetizált áramkörben a bekapcsolás pillanatában inicializálatlan. Komoly bonyodalmakat okozhat, hogy az „inicializálatlanság” szimulációs környezetben és egy fizikailag létező áramkörben egészen mást jelent; míg szimulációban az inicializálatlan állapot a logikai jeleknek egy, az alacsony és a magas szinttől is különböző lehetséges értéke, addig az FPGA-n belül az inicializálatlan bit – legyen az flip-flop vagy SRAM-cella – a bekapcsolási tranziens végén vagy alacsony vagy magas szintű. Nem tudhatjuk, hogy melyik, de bizonyosan a kettő közül valamelyik lesz, több lehetséges érték nincs. Ez azt jelenti, hogy a szimulációs környezetben és a valóságban tapasztalt viselkedés eltérhet.[95]

[94] A *dualclock_fifo_hswrp* IP-mag alkalmas a 2-4., 2-5. és 2-6. ábrákon vázolt órajel-szinkronizálási séma megvalósítására. A regularitás jegyében ezt az IP-magot használjuk a FIFO jellegű interfész-burkolómodulok létrehozására az összes többi újrafelhasználható IP-mag esetén.

[95] Az angol nyelvű szakirodalom „simulation-synthesis mismatch”-ként hivatkozik a jelenségre, ami az RTL-tervezők legrosszabb rémálma. A CDC tárgykörének mélyén ugyanez a probléma rejlik, vagyis az, hogy a szimulátor által alkalmazott absztrakció pontatlanul modellezi a valóságot.

Az SRAM-cellák tartalmának inicializálására léteznek ugyan megoldások (pl. memória-inicializációs fájl, kezdetiérték mint HDL nyelvi elem felhasználása), ezek azonban a különböző gyártók, sőt, azon belül is a különböző eszközcsaládok esetén eltérőek, így rájuk hagyatkozva csak technológiafüggő megoldásokhoz juthatunk. Újrafelhasználható és technológiafüggetlen IP-magok fejlesztése esetén a legáltalánosabb dolog, amit tehetünk, hogy rendszerszinten gondoskodunk arról, hogy az inicializálatlan memóriatartalom ne okozhasson funkcionális hibát. Erre láthatunk példát funkcióbővítő rendszerünkben is; a *BOOTSTRAP* modul (lásd 2.3. pont) egy olyan egység, ami biztosítja, hogy a *CPU* utasításmemóriája mindenképp fel legyen töltve egy külső EEPROM-ból, mielőtt elkezdődhetne a programvégrehajtás. Az FFT IP (lásd 4.5.5. pont) esetén a cosinus-tábla, a FIR IP (lásd 4.5.6. pont) esetén pedig az együtthatókat tároló memória inicializálatlan az áramkör bekapcsolásakor. Ez utóbbi két IP nem tartalmaz a *BOOTSTRAP*-nél látottakhoz hasonló mechanizmust, ami fizikailag lehetetlenné tenné az inicializálatlan memóriákkal való elindulást, ugyanakkor biztosítják e memóriamodulok inicializálhatóságát egy külső interfészen keresztül. Dokumentációjukban tehát rögzíthető, sőt rögzítendő, hogy miképpen hozandók „üzemkész" állapotba ezek az IP-k a kérdéses memóriatartalmak feltöltésével.

A fenti példák mindegyikére igaz, hogy az inicializációs folyamat szimulálható, ugyanakkor ez sokszor körülményes (legalább verifikációs komponensként modelleznünk kell az inicializálást végző „külső szereplőt"), nagyméretű memóriák esetén pedig megengedhetetlenül időigényes is lehet. Érdemes tehát olyan megoldást találni, ami ugyan nem szintetizálható, de legalább szimulációs környezetben használható a blokk-RAM-ként megvalósítandó memóriamodulok tartalmának inicializálására. A Verilog rendelkezésünkre bocsátja a *readmemh* nevű rendszerhívást (system task), amelyet egy *initial*-blokkban elhelyezve a kijelölt tömböt egy szövegfájl tartalma alapján inicializálhatjuk a szimuláció indításakor (lásd 4-28. ábra). A *readmemb* rendszerhívás működése hasonló, segítségével a kívánt kezdeti értékeket bináris formában is megadhatjuk.

Verilog

```
1  module singleclock_dualport_ram #(parameter aw, dw)
2  (
3    input  logic clk, wstrobe, rstrobe,
4    input  logic [aw-1:0] waddr,
5    input  logic [aw-1:0] raddr,
6    input  logic [dw-1:0] din,
7    output logic [dw-1:0] dout
8  );
9
10   logic [dw-1:0] content [0:2**aw-1];
11
12   initial $readmemh("C:/TMP/init_hex.dat", content);
13   // initial $readmemb("C:/TMP/init_bin.dat", content);
14
15   always @ ( posedge clk ) begin
16     if ( rstrobe == 1 ) dout <= content[raddr];
17     if ( wstrobe == 1 ) content[waddr] <= din;
18   end
19
20 endmodule
```

Szövegfájlok a kezdeti értékekkel

```
// "C:/TMP/init_hex.dat"
1  67
2  03
3  11
4  7D
5  EB
6  12
7  45
8  32


// "C:/TMP/init_bin.dat"
1  01100111
2  00000011
3  00010001
4  01111101
5  11101011
6  00010010
7  01000101
8  00110010
```

4-28. ábra. *A véletlen elérésű memóriamodul tartalmának inicializálása*

Újrafelhasználható IP-magként tekintve a véletlen elérésű memóriamodulra a 4-28. ábrán látható megoldás nem praktikus, hiszen ilyen formában a modul minden példányát ugyanabból a fájlból próbálnánk inicializálni. Ez már csak azért sem értelmes, mert a memóriamodul adat- és címbuszának mérete olyan paraméter, aminek a felhasználó csak a példányosításkor ad értéket. Célszerű tehát a modellt úgy kiegészíteni, hogy a felhasználó a példányosításkor adhassa meg az

inicializálásra használt szövegfájl elérési útját, illetve azt is, hogy egyáltalán szeretné-e fájlból inicializálni a tartalmat. Ezt a két apró kiegészítést a 4-29. ábra szemlélteti.

Verilog

```
1  module singleclock_dualport_ram #(parameter aw, dw, init, initfile_path)
2  (
3    input  logic clk, wstrobe, rstrobe,
4    input  logic [aw-1:0] waddr,
5    input  logic [aw-1:0] raddr,
6    input  logic [dw-1:0] din,
7    output logic [dw-1:0] dout
8  );
9
10   logic [dw-1:0] content [0:2**aw-1];
11
12   initial if (init) $readmemh(initfile_path, content);
13
14   always @ ( posedge clk ) begin
15     if ( rstrobe == 1 ) dout <= content[raddr];
16     if ( wstrobe == 1 ) content[waddr] <= din;
17   end
18
19 endmodule
```

4-29. ábra. *Az inicializálás miként je paraméterekkel megfogalmazva*

A VHDL egy, a Verilognál sokkal általánosabb célú modellező nyelv, aminek következtében a *readmemh/b* rendszerhíváshoz hasonló eszköz ugyan „gyárilag” nincs benne, de a nyelv egyéb lehetőségeit kihasználva könnyen készíthetünk ezzel egyenértékű megoldást. A 4-30. ábra egy csomagban (package) definiált függvényt mutat be, amely a VHDL szabványos *textio* csomagjára támaszkodva valósítja meg ugyanazt a funkciót, amit a Verilog *readmemh* és *readmemb* rendszerhívásai nyújtanak.

A *mem_init* függvény egy, a felhasználó által megadott méretű vektortömböt *(tmp)* épít fel egy szöveges fájl tartalma alapján, és ezt a vektortömböt visszatérési értékként adja vissza. A paraméterként megadott elérési úton lévő szövegfájl megnyitása (4-30. ábra: 31. sor) után a függvény egy, a címbusz mérete alapján meghatározott hosszúságú ciklussal végighalad annak sorain (4-30. ábra: 32. sor). Az egyes sorok – amelyek egy-egy logikai vektor kívánt értékét tartalmazzák bináris vagy hexadecimális alakban (lásd 4-28. ábra jobb oldala) – feldolgozása a következőképpen megy végbe: A *readline* függvény beolvas egy teljes sort a *line_buffer* változóba (4-30. ábra: 33. sor). A paraméterként megadott formátum szerint vagy a *read* vagy a *hread* függvényt használjuk arra, hogy a *line* típusú karaktersorozatot dekódoljuk, és képezzünk belőle egy *std_logic_vector* típusú változót (4-30. ábra: 34–37. sor). Ezt a változót betöltjük egy logikai vektorokat tartalmazó tömbnek a ciklusváltozóval címzett rekeszébe (4-30. ábra: 38. sor). A ciklikus, soronkénti beolvasás végére a *tmp* változóban előáll logikai vektoroknak egy tömbje, amely már a szövegfájlban meghatározott értékeket tartalmazza. Ezt a tömböt visszatérési értékként adjuk át a felhasználónak (4-30. ábra: 41. sor). Ha a felhasználó az *init* paramétert *false*-ra állítva hívja meg a függvényt, akkor egy megfelelő méretű, de üres (inicializálatlan) tömböt adunk vissza (4-30. ábra: 42. sor).

A 4-30. ábra függvényét a VHDL kezdetiérték-adás utasításával együtt kell használnunk ahhoz, hogy egy tetszőleges, logikai vektorokat tartalmazó tömböt inicializáljunk a szimuláció elején. A 4-31. ábra a véletlen elérésű memória VHDL-modelljén mutatja be az inicializáló függvény használatát.

Az inicializáló függvény paraméterei egyúttal magának a memória IP-magnak is paraméterei, vagyis a felhasználó az inicializálás módját annak minden példányára külön határozhatja meg.

VHDL

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_textio.all;
use std.textio.all;

package storage_pkg is

  type memory_content_t is array (natural range <>) of std_logic_vector;

  impure function mem_init (
    init: boolean;
    initfile_path, initfile_format: string;
    aw, dw: integer range 2 to 128
  ) return memory_content_t;

end package storage_pkg;

package body storage_pkg is

  impure function mem_init (
      init: boolean;
      initfile_path, initfile_format: string;
      aw, dw: integer range 2 to 128
  ) return memory_content_t is
      file initfile: text;
      variable line_buffer: line;
      variable recent_word: std_logic_vector (dw-1 downto 0);
      variable tmp: memory_content_t (0 to 2**aw-1)(dw-1 downto 0);
  begin
    if ( init = true ) then
      file_open(initfile, initfile_path, read_mode);
      for i in 0 to 2**aw-1 loop
        readline(initfile, line_buffer);
        if ( initfile_format = "BIN" ) then read(line_buffer, recent_word);
        elsif ( initfile_format = "HEX" ) then hread(line_buffer, recent_word);
        else assert false report "Initfile format error" severity failure;
        end if;
        tmp(i) := recent_word;
      end loop;
      file_close(initfile);
      return tmp;
    else return tmp;
    end if;
  end function;

end package body storage_pkg;
```

4-30. ábra. *VHDL függvény a tömbök inicializálásához*

VHDL

```vhdl
entity singleclock_dualport_ram is
  generic (
    aw, dw: integer range 2 to 128
    init:   boolean;
    initfile_path, initfile_format: string
  );

  port ( [...] );

end entity singleclock_dualport_ram;

architecture rtl of singleclock_dualport_ram is

   signal content: memory_content_t (0 to 2**aw-1)(dw-1 downto 0) :=
     mem_init(init, initfile_path, initfile_format, aw, dw);

begin ( [...] ) end architecture rtl;
```

4-31. ábra. *Az inicializáló függvény használata a véletlen elérésű memória IP-mag esetén*

4.3. A soros kommunikáció alapáramkörei

Ebben a fejezetben a beágyazott rendszerekben legelterjedtebben használt soros adatátviteli interfész-megoldásokat, az UART-ot, az SPI-t és az I^2C-t megvalósító soros-párhuzamos és párhuzamos-soros átalakító áramkörökkel foglalkozunk. Mindhárom protokoll specifikációjára igaz, hogy funkcionalitás szempontjából jól skálázhatók; aránylag könnyen készíthetők a szabályrendszerük legegyszerűbb változatának megfelelő áramkörök, amelyek a teljesítmény – ez esetben az adatátviteli sebesség és/vagy a megbízhatóság – növelése érdekében megfelelő alapot nyújthatnak a továbbfejlesztéshez. Az itt bemutatott áramköri megoldások csak az interfész-protokollok 5.2. Függelékben tárgyalt alapfunkcióit valósítják meg, a különféle többletfunkciók (pl. UART hardver-átvitelvezérlés, SPI órajel-polaritás és -fázisüzemmódok stb.) ezeknek az alapáramköröknek a kisebb-nagyobb módosításaival beépíthetők.

4.3.1. UART-adó és -vevő

> ***Áttanulmányozásra javasolt alfejezetek:***
> ***5.2.1.*** *Egyszerű soros kommunikációs interfészek – UART*
> ***4.2.*** *On-chip adattárolás*

Ahogy az az 5.2.1. Függelékben is olvasható, az UART-adó és -vevő modulok működése tulajdonképpen egyetlen vezeték megfelelő ütemezésű írását vagy olvasását jelenti. Az e feladatot ellátó áramkörök tervezésekor feltételezzük, hogy mind az adó, mind a vevő oldalon rendelkezésünkre áll egy-egy, néhányszor 10 MHz frekvenciájú helyi órajelforrás, az UART-kapcsolat adatátviteli sebessége pedig 2-200 kb/s, vagyis egyetlen bit átvitele legalább 50 órajelciklus ideig tart. A pontos ütemezési szabályok szerint működő áramkörök létrehozásához a lokális órajelforrás mellett időzítőszámlálókra van szükség, amelyek saját működését és az ezeket körülvevő egyéb részáramkörök együttműködését regiszter-transzfer szinten nagyon sokféleképpen fogalmazhatjuk meg (pl. önálló IP-magként lásd 4.1.1. pont: Órajel-előosztó, vagy DRMSz: 4-7. alfejezet: Számláló-alapú funkcionális egységek). Egyszerű problémák esetén jól áttekinthető, rugalmasan módosítható (javítható) leírást kapunk, ha az időzítő-számlálót állapotgéppel kombinálva írjuk le. A modell kompaktságát és olvashatóságát tovább javítja, ha az állapotgép modellezéséhez a one-shot leírási módszert választjuk, ami lehetővé teszi, hogy a számláló-időzítő és az állapotgép működését egyetlen „összeolvasztott" HDL-modellel fejezzük ki. Ezt a módszert alkalmazzuk az UART-adó és -vevő áramkörök RTL-terveinek elkészítésekor.

UART-adó (párhuzamos-soros átalakító) ◆ A párhuzamos-soros átalakító áramkört leíró IP-magot szinkron kézfogásos interfésszel látjuk el oly módon, hogy egyetlen adatkeret soros bitfolyammá alakítását tekintjük az IP egyetlen szolgáltatásának. E szolgáltatás egyszeri végrehajtását akkor tekintjük befejezettnek, amikor a start-bit és a felhasználótól kapott adatvektor bitenkénti elküldése után az adatkeret stop-bitjének megfelelő időintervallum is eltelt. Újabb elküldendő adatvektorokat ez idő alatt az IP nem képes fogadni a felhasználótól. A HDL-modell vázlatát a 4-32. ábra mutatja be.

Verilog

```
1  always @ ( posedge clk, posedge rst ) begin  // REQUEST RISING EDGE DETECTOR
2    if ( rst == 1 ) request_df0 <= 1'b0;
3    else request_df0 <= request;
4  end
5  assign request_rising = request & ~request_df0;
6
7  always @ ( posedge clk, posedge rst ) begin // MAIN FSM
8    if ( rst == 1 ) begin
9      state <= idle; ready <= 1; TX <= 1; bit_delay_counter <= 0; bit_counter <= 0; even_parity <= 0;
10   end else begin
11   |
12   |  case ( state )
13   |  |
14   |  |  idle:    if ( request_rising == 1 ) begin [...]
15   |  |  |
16   |  |  [...]
17   |  |  |
18   |  |  default: ;
19   |  |
20   |  endcase
21   |
22   end
23 end
```

4-32. ábra. *Az UART-adó HDL-modelljének vázlata*

A 4-32. ábra HDL-modelljének 1-5. soraiban definiált felfutóél-detektor biztosítja az adó áramkör kompatibilitását a szinkron kézfogásos interfész 1-4. ábrán látható specifikációjával; a felhasználó az adatküldésre vonatkozó kérését jelezheti a *request* jelen előállított, egyetlen órajel-periódus időtartamú pozitív impulzussal, vagy megteheti, hogy a *request* jelet mindaddig magas szinten tartja, amíg a *ready* jelzés felfutó élét nem érzékeli. A 4-33. ábra az adatküldés megkezdését leíró vezérlési állapotokat szemlélteti.

Verilog

```
1  idle: if ( request_rising == 1 )
2  begin
3    ready <= 0;
4    state <= send_start;
5  end
6
```

Verilog (folyt.)

```
7  send_start: begin
8    TX <= 0;
9    if ( delay_counter < delay_counter_max )
10     delay_counter <= delay_counter + 1;
11   else begin
12     delay_counter <= 0;
13     state <= send_data;
14   end
15 end
16
17 send_data: [...]
```

4-33. ábra. *Az UART-adó működését leíró állapotgép (1)*

A 8. sorban látható, hogy a sorosított adatkeret mindig az alacsony értékű start-bittel kezdődik, amit a felhasználó által – a helyi órajel frekvenciájának ismeretében – definiált *delay_counter_max* szintézis-paraméterrel beállított időtartamig tartunk fenn. A start-bit előállítása után az állapotgép a *send_data* állapotban folytatja a műveletvégzést (lásd 4-34. ábra). Ebben az állapotban a felhasználó által küldeni kívánt vektor bitjeit – a *bit_counter* számlálóval kiválasztva – sorban rákapcsoljuk a *TX* kimenetre úgy, hogy az egyes bitváltások között ezúttal is a *delay_counter_max* paraméter által meghatározott számú órajelciklus teljen el.

A hasznos adatbitek *TX* vonalra kapcsolgatása után, amikor a *frame_size* szintézis-paraméter által meghatározott számú bitet elküldtük, akkor vagy a paritásbit vagy a stop-bit következik (4-34. ábra: 11–13. sor). Egy UART-kapcsolatban az átvitt adat helyességének ellenőrzését szolgáló paritásbit használata és típusa (páros vagy páratlan) is opcionális, ezért annak előállítását a *parity* szintézis-paraméter értékétől tesszük függővé (0: nincs paritásbit-generálás, 1: páros, 2: páratlan).

Verilog

```
1  send_data: begin
2    TX <= data_2_send[bit_counter];
3
4    if ( delay_counter < delay_counter_max )
5      delay_counter <= delay_counter + 1;
6    else begin
7      delay_counter <= 0;
8      if ( bit_counter < frame_size - 1 )
9        bit_counter <= bit_counter + 1;
10     else begin
11       bit_counter <= 0;
12       if ( parity != 0 ) state <= send_parity;
13       else state <= send_stop;
14     end
15   end
16
17 end
18
```

Verilog (folyt.)

```
19 send_parity: begin
20
21   even_parity = 0;
22   for ( int i = 0; i < frame_size; i = i+1 ) begin
23     even_parity = even_parity ^ data_2_send[i];
24   end
25
26   if ( parity == 2 ) TX <= ~even_parity;
27   else TX <= even_parity;
28
29   if ( delay_counter < delay_counter_max )
30     delay_counter <= delay_counter + 1;
31   else begin
32     delay_counter <= 0;
33     state <= send_stop;
34   end
35
36 end
```

4-34. ábra. *Az UART-adó működését leíró állapotgép (2)*

A paritásbit kiszámítására és elküldésére a *send_parity* vezérlési állapotban kerül sor. A 4-34. ábra 21–24. soraiban láthatjuk, hogy a páros paritásbitet *(even_parity)* egy *for*-ciklussal számítjuk ki a felhasználótól érkező *data_2_send* vektor adatbitjeinek *XOR* kapcsolatát képezve. A blokkoló értékadás (= operátor) és a *for*-ciklus nyelvi eszközök kombinációja kaszkád *XOR* struktúrát eredményez (vö. DRMSz: 4-31. ábra: Paritásgenerátor-áramkör leírása *for*-ciklussal), ami azt jelenti, hogy az *even_parity* bit értékét kiszámító áramköri részlet egy, az állapotgép kimeneti logikájának részét képező kombinációs hálózat lesz. Az *even_parity* bitet a *parity* paraméter értékétől függően vagy közvetlenül vagy negálva kapcsoljuk a *TX* vonalra (4-34. ábra: 26–27. sor).

Verilog

```
1  send_stop: begin
2    TX <= 1;
3    if ( delay_counter < delay_counter_max ) delay_counter <= delay_counter + 1;
4    else begin
5      delay_counter <= 0; ready <= 1; state <= idle;
6    end
7  end
```

4-35. ábra. *Az UART-adó működését leíró állapotgép (3)*

Az adatkeret küldését a logikai magas szintű stop-bit elküldése zárja le. A 4-35. ábrát a 4-33. ábrával összevetve láthatjuk, hogy a start- és stop-bitek generálása csupán a *TX* vezetékre juttatott értékben különbözik, illetve abban, hogy a stop-bit időablakának végén a felhasználót a *ready* kimenet magas szintre húzásával értesítjük az adatküldés befejezéséről (4-35. ábra: 5. sor).[96]

UART-vevő (soros-párhuzamos átalakító) ◆ A vevő IP-mag leírására az adóéhoz nagyon hasonló felépítésű HDL-modellt készítünk; megoldásunk alapja itt is egy belső időzítő-számlálót tartalmazó állapotgép. A HDL-modell vázlatát a 4-36. ábra mutatja be.

[96] Az UART protokoll a stop-bit időtartamát 1, 1,5 vagy 2 bitidőben határozza meg (lásd 5-3. ábra). E három opció közül az itt bemutatott IP-mag az 1 bitidőnyi verziót valósítja meg. Ha egy konkrét alkalmazásban – pl. a vevőoldal sajátosságai miatt – az adatkeretek között több időnek kell eltelnie, akkor az adó IP-mag felhasználójának figyelnie kell arra, hogy az UART-adó *ready* jelének felfutása után némi várakozást iktasson be, mielőtt újra adásra szólítja fel az adó áramkört.

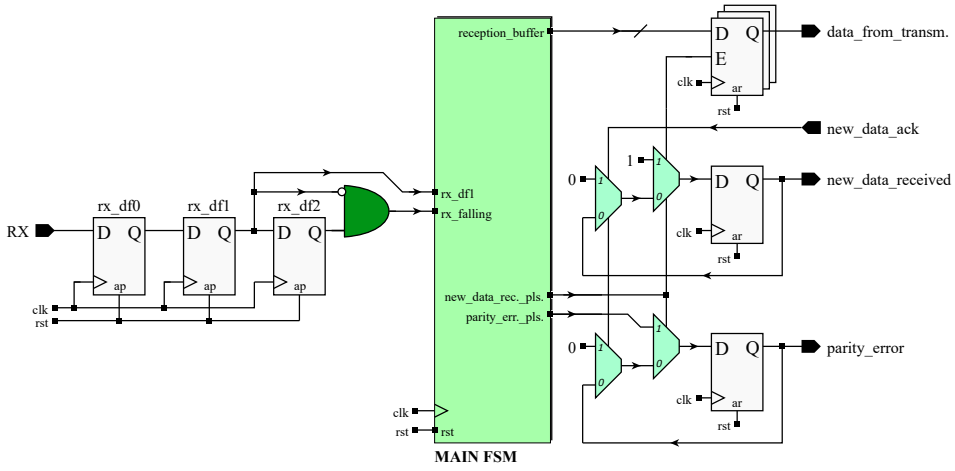
Verilog

```
// RX FALLING EDGE (START-BIT) DETECTOR
always @ ( posedge clk, posedge rst ) begin
  if ( rst == 1 ) begin
    rx_df0 <= 1; rx_df1 <= 1; rx_df2 <= 1;
  end else begin
    rx_df0 <= RX; rx_df1 <= rx_df0; rx_df2 <= rx_df1;
  end
end
assign rx_falling = rx_df2 & ~rx_df1;

//--------------------------------------

always @ ( posedge clk, posedge rst ) begin
  if ( rst == 1 ) begin
    new_data_received <= 0; parity_error <= 0;
    data_from_transmitter <= 0;
  end else begin
    if ( new_data_received_pulse == 1 ) begin
      new_data_received <= 1;
      data_from_transmitter <= reception_buffer;
      parity_error <= parity_error_pulse;
    end else if ( new_data_ack == 1 ) begin
      new_data_received <= 0;
      parity_error <= 0;
    end
  end
end

```

Verilog (folyt.)

```
// MAIN FSM
always @ ( posedge clk, posedge rst ) begin
  if ( rst == 1 ) begin
    state <= idle;
    delay_counter <= 0; bit_counter <= 0;
    new_data_received_pulse <= 0;
    parity_error_pulse <= 0;
    even_parity <= 0;
    reception_buffer <= 0;
    received_parity <= 0;
  end else begin

    // defaults...
    new_data_received_pulse <= 0;
    parity_error_pulse <= 0;

    case ( state )

      idle: if ( rx_falling == 1 ) [...]

      [...]

      default: ;

    endcase

  end
end
```

Áramköri struktúra



4-36. ábra. *Az UART-vevő RTL-modelljének vázlata*

Az *RX* jelet egy adó áramkör hajtja meg, ami tipikusan egy másik eszköz (mikrovezérlő, egy másik FPGA stb.) része, vagyis az e jelen végbemenő élváltások a vevő áramkör órajeléhez nem szinkronizáltak. A double-flop szinkronizáló fokozat tehát feltétlenül szükséges a metastabilitás elkerüléséhez. A szinkronizálót egy lefutóél-detektor egészíti ki, ami a start-bit bekövetkezését egy egyetlen órajelciklus hosszúságú pozitív impulzussal jelzi az állapotgépnek.[97] Az állapotgép az *RX* vonalon vett biteket sorban betölti a belső *reception_buffer* regiszterébe, majd amikor egy

[97] A lefutóél-detektor persze az *RX* vonal minden lefutóélénél jelez. A lefutóélek sokaságáról valójában csak az állapotgép tudja eldönteni (az *RX* vonal korábbi viselkedésének ismeretében), hogy melyik jelenti a start-bit kezdetét.

teljes adatkeretet beolvasott, akkor a *new_data_received_pulse* jelen előállított impulzussal egyrészt áttölti azokat a *data_from_transmitter* regiszterbe, másrészt magas szintre állítja a *new_data_received* állapotjelzőt, így értesítve a felhasználót az új adatkeret megérkezéséről. A felhasználó értesítésére akkor is sor kerül, ha a beolvasott adatkeretben az állapotgép paritáshibát fedez fel; ilyenkor a *new_data_received_pulse* jellel együtt a *parity_error_pulse* jelen is előállít egy impulzust, ami magas szintre húzza a felhasználónak szóló *parity_error* állapotjelzőt. Mindkét kimeneti állapotjelző a *new_data_ack* bemenet magas szintre állításával törölhető. A 4-37. ábra az állapotgép start-bitre várakozó, illetve a start-bit kezdetét jelző lefutóél utáni várakozást megvalósító vezérlési állapotait mutatja be.

Verilog

```
1  idle: if ( rx_falling == 1 ) state <= get_start;
2
3  get_start: if ( delay_counter < delay_counter_max ) delay_counter <= delay_counter + 1;
4             else begin
5               delay_counter <= 0; state <= get_data;
6             end
```

4-37. ábra. *Az UART-vevő működését leíró állapotgép (1)*

E két állapot nagyon hasonlít az adó állapotgépének első két állapotára, hiszen a feladat is szinte ugyanaz; egy külső esemény hatására egy bitidőnyi várakozás beiktatása a műveletvégzésbe. Míg az adó esetén ez a külső esemény a felhasználói kérés (vö. 4-33. ábra: 1. sor: *request_rising*), addig a vevő számára az *RX* jelen előálló lefutóél (4-37. ábra: 1. sor: *rx_falling*), vagyis a start-bit kezdete. A bitidőnek megfelelő időtartamú várakozást ugyancsak az adónál látottakkal megegyező módon, az állapotgépbe ágyazott időzítő-számlálóval valósítjuk meg. A start-bit időablakának letelte után az állapotgép a *get_data* állapotba lép, itt kerül sor az adatkeret bitjeinek megfelelő időzítésű mintavételezésére (lásd 4-38. ábra).

Verilog

```
1  get_data: begin
2    if ( delay_counter == delay_counter_max/2 ) reception_buffer[bit_counter] <= rx_df1;
3
4    if ( delay_counter < delay_counter_max ) delay_counter <= delay_counter + 1;
5    else begin
6      delay_counter <= 0;
7      if ( bit_counter < frame_size-1 ) bit_counter <= bit_counter + 1;
8      else begin
9        bit_counter <= 0;
10       if ( parity != 0 ) state <= get_parity;  // parity: 0: no parity, 1: even, 2: odd
11       else begin
12         new_data_received_pulse <= 1;
13         state <= idle;
14       end
15     end
16   end
17
18 end
```

4-38. ábra. *Az UART-vevő működését leíró állapotgép (2)*

A *get_data* állapot olyan periodikus viselkedést ír le, amelyben a *delay_counter* regiszterrel megvalósított, egy bitidőnyi várakozás után a *bit_counter* regiszter értéke inkrementálódik, miközben – az adott bit időablakának közepén, a bitidő felénél – mintát veszünk az *RX* vonal double-flop szinkronizálójának kimenetéről *(rx_df1)*, és a mintát a *reception_buffer* regiszter *bit_counter* által kijelölt bitjébe írjuk (4-38. ábra: 2. sor). A *frame_size* szintézis-paraméter által meghatározott számú iteráció után a *reception_buffer* regiszterben előáll a teljes adatkeret. Ezután a *parity*

szintézis-paraméter értékétől függően vagy továbblépünk a paritásbitet beolvasó vezérlési állapotba (4-38. ábra: 10. sor), vagy az adatkeret vételét sikeresnek tekintve előállítunk egy pozitív impulzust a *new_data_received_pulse* jelen és visszatérünk az újabb start-bit kezdetét jelző lefutóélre várakozó *idle* állapotba (4-38. ábra: 12–13. sor).[98] Ha paritásbitet használunk a fogadott adatkeret esetleges sérülésének detektálására, akkor azt egy újabb mintavételi ciklussal kell beolvasnunk, majd a fogadott bitet a *reception_buffer* regiszterben korábban eltárolt bitek alapján helyben kiszámított paritásbittel kell összevetnünk. Az e részfunkciókat leíró vezérlési állapot a 4-39. ábrán látható.

Verilog

```
1  get_parity: begin
2    if ( delay_counter == delay_counter_max/2 ) received_parity <= rx_df1;
3
4    if ( delay_counter < delay_counter_max ) delay_counter <= delay_counter + 1;
5    else begin
6      delay_counter <= 0;
7
8      even_parity = 0;
9      for ( int i = 0; i < frame_size; i = i+1 ) begin
10       even_parity = even_parity ^ reception_buffer[i];
11     end
12
13     if (
14          (parity == 1 & (received_parity != even_parity)) |          // parity: 0: no parity
15          (parity == 2 & (received_parity == even_parity))            //         1: even
16     ) parity_error_pulse <= 1;                                       //         2: odd
17
18     new_data_received_pulse <= 1;
19     state <= idle;
20
21   end
22
23 end
```

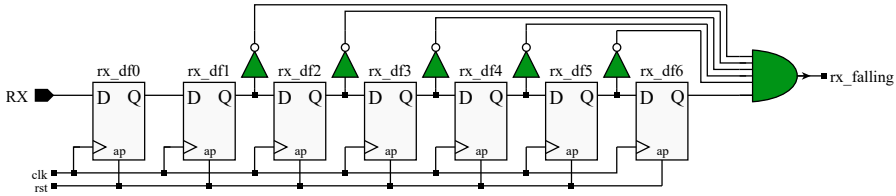
4-39. ábra. *Az UART-vevő működését leíró állapotgép (3)*

A zajérzékenység csökkentése ◆ Az UART-adó és -vevő áramkörök tervezésekor legfeljebb 200 kb/s adatátviteli sebességet tűztünk ki célul legalább 10 MHz-es lokális órajelforrás mellett. Ez azt jelenti, hogy egyetlen bit átvitele legalább 50 órajelciklus ideig tart, ami lehetőséget ad arra, hogy az adatátvitel megbízhatóságát a vevő áramkör apró továbbfejlesztéseivel jelentősen növeljük.

Az 50 órajelciklusnyi bitidő azt jelenti, hogy a vevő áramkör *RX* bemenetén minden, ennél rövidebb időtartamú tranziens csak valamilyen hiba vagy a környezetből becsatolt zaj következménye lehet. Ha az inaktív vezetéken egy 1 → 0 → 1 tranziens a helyi órajelforrás periódusidejénél hosszabb időtartamú, akkor azt a vevő start-bitként értelmezi, és megkezdi az adatkeret mintavételezését. A vevő-oldali felhasználó tehát egy soha el nem küldött adatkeret vételét tapasztalja, ami nyilván nem megengedhető. Ez a forgatókönyv az *RX* vonal lefutóél-detektorának 4-40. ábrán látható módosításával megakadályozható.

A módosított éldetektor csak akkor jelez, ha az *RX* bemenet lefutóéle után a vonal legalább annyi ideig alacsony szintű marad, amíg ez az alacsony szint be nem töltődik az *rx_df1* … *rx_df5* flip-flopok mindegyikébe. Ha az *rx_df1* flip-flop újra magas szintbe billen, mielőtt az alacsony szint

[98] A *new_data_received_pulse* jel egy egyetlen órajel-periódusig aktív impulzusjel, amelynek alapértelmezett alacsony értékét az állapotgépet leíró *case*-szerkezet elé helyeztük (lásd 4-36. ábra: 42. sor).



4-40. ábra. *Módosított lefutóél-detektor az RX vonal rövid tranzienseinek kiszűrésére*

„elérné” az *rx_df5* flip-flopot, akkor az éldetektor kimeneti *AND* kapuja nem jelez. További flip-flopok hozzáadásával a kiszűrt negatív impulzushossz tovább növelhető.

Az 50 órajelciklusnyi bitidő azt is lehetővé teszi, hogy egyetlen bitből több mintát is vegyünk, és a nem kívánt tranziensek miatt esetleg eltérő minták összessége alapján állapítsuk meg, hogy mi lehetett az adó eredeti szándéka. A többszörös mintavétel legegyszerűbb formája, ha a bitidő alatt három mintát veszünk az *RX* jelből, majd a három mintából „kétharmados szavazással” határozzuk meg, hogy az adott bitet egyesnek vagy nullának tekintjük-e. E funkció megvalósításához a vevő IP-mag állapotgépének módosítására van szükség (lásd 4-41. ábra).

Verilog

```
get_data: begin

  if ( delay_counter == delay_counter_max/3 ) sample_0 <= rx_df1;
  if ( delay_counter == delay_counter_max/2 ) sample_1 <= rx_df1;
  if ( delay_counter == (2*delay_counter_max)/3 ) sample_2 <= rx_df1;

  if ( delay_counter < delay_counter_max ) delay_counter <= delay_counter + 1;
  else begin
    delay_counter <= 0;
    |
    reception_buffer[bit_counter] <= (sample_0 & sample_1) |
    |                                (sample_0 & sample_2) |
    |                                (sample_1 & sample_2);
    |
    if ( bit_counter < frame_size-1 ) bit_counter <= bit_counter + 1;
    else begin
      bit_counter <= 0;
      if ( parity != 0 ) state <= get_parity;  // parity: 0: no parity, 1: even, 2: odd
      else begin
        new_data_received_pulse <= 1;
        state <= idle;
      end
    end
  end

end
```

4-41. ábra. *Háromszori mintavétel megvalósítása az UART-vevőben (vö. 4-38. ábra)*

A 4-41. ábra 3-5. soraiban látható, hogy a három mintát az adott bit időablakában elosztva, a bitidő harmadánál, felénél és kétharmadánál vesszük.[99] A mintákat rendre a *sample_0*, *sample_1* és *sample_2* regiszterekbe töltjük. A bitidő végén a három mintából egy *MAJ3* döntőlogikával

[99] A rövid negatív impulzusok kiszűrését végző éldetektor a sorbakapcsolt flip-flopok miatt késleltetést okoz a start-bit kezdetét jelző lefutóél detektálásában. Ennek következtében az *rx_df1* vonal mintavételezése valójában kissé eltolódik. Az itt bemutatott konkrét esetben, amikor a bitidő minimumát 50 órajelciklusban határoztuk meg, és az éldetektorban öt flip-flopot használtunk, az eltolódás mértéke nem okoz gondot. Ha azonban változatlan helyi órajel-frekvencia mellett csökkentjük a bitidőt, illetve az éldetektort további flip-flopokkal egészítjük ki, akkor a mintavétel időzítési eltolódását érdemes körültekintően megvizsgálni.

képezzük a *reception_buffer* regiszterbe betöltendő értéket (4-41. ábra: 11–13. sor). Természetesen a háromszori mintavételt alkalmazó megoldásnak is vannak korlátai, hiszen ezzel hallgatólagosan azt feltételezzük, hogy a három mintából legfeljebb egy lehet hibás, ami az összeköttetés fizikai kialakításától és az áramköri környezettől függően nem feltétlenül igaz. Rossz minőségű, sok-sok nemkívánt tranzienssel terhelt összeköttetés esetén ez a módszer sem garantálja a hibátlan adatátvitelt, a megoldás mindenesetre lényegesen zavartűrőbb az egyszeri mintavételt alkalmazó verziónál.

4.3.1.1. Órajel-szinkronizálás

Az UART-adó áramkörben a bitidőt egy állapotgépbe épített időzítőszámláló határozza meg a helyi órajelforrás frekvenciájának leosztásával. Minél nagyobb az eltérés a bitidő és az órajelforrás periódusideje között, annál nagyobb leosztásra van szükség, vagyis annál tovább kell számolnia az időzítőnek egy-egy bit *TX* vonalon tartása közben. A nagyméretű számláló összeadó áramköre időzítési problémákhoz vezethet (lásd még 4.1.1. pont), ráadásul sok erőforrást igényel. Ezek alapján megállapíthatjuk, hogy az UART-adó lokális órajelének frekvenciáját általánosságban érdemes korlátozni. Ne feledkezzünk meg arról sem, hogy a bitidő a lokális órajelforrás periódusidejének egész számú többszöröse lesz. Mindezek alapján látható, hogy az UART-adó IP-mag saját órajel-frekvenciáját célszerű függetleníteni áramköri környezete órajel-frekvenciáitól. Az adó felhasználói interfészének működése megfelel a 1.2.1.1. pontban bemutatott szinkron kézfogásos interfész-specifikációnak, vagyis használhatjuk a 2-2. ábra órajel-szinkronizáló fokozatát, az általánosított $A_{i,S}$ vektort az UART-adó *data_2_send* bemeneti vektorának megfeleltetve.

A vevő áramkör esetén a belső időzítőszámlálóval kapcsolatos, a lokális órajel frekvenciájának csökkentése mellett szóló érvelés ugyancsak érvényes. Itt azonban érdemes megjegyezni, hogy a vevő áramkörben a megbízhatóság növelését célzó megoldások szempontjából a nagy frekvenciájú lokális órajel ad némi mozgásteret; minél több órajelciklus áll rendelkezésünkre egy bit vétele alatt, annál több mintát vehetünk, és annál kifinomultabb logikákat találhatunk ki annak eldöntésére, hogy mi lehetett az adott bit értéke. A fenti két érv ellentmond egymásnak, tehát a konkrét alkalmazás igényei alapján kell döntenünk, vagy – újrafelhasználható IP-mag tervezésekor – észszerű kompromisszumra kell törekednünk. Akárhogy is, az adónál látottakhoz hasonlóan a vevő által kiszámított bitidő is a lokális órajel periódusidejének egész számú többszöröse lesz, vagyis az órajel-frekvenciát a kívánt bitidő szerint finomhangolnunk kell. A vevőre is igaz tehát, hogy áramköri környezetétől független órajeltartományt érdemes létrehoznunk neki.

A vevő IP-mag működése felfogható úgy is, hogy az adó által a soros UART protokoll szabályainak megfelelően végrehajtott adatátviteli tranzakciókat párhuzamosakká alakítja. A modul az egyes UART adatkeret-fogadások végén a *new_data_received*, *parity_error*, *data_from_transmitter* és *new_data_ack* jelek alkotta interfészen adatcserét kezdeményez. E jelek működése ugyancsak megfelel a szinkron kézfogásos interfész-specifikációnak, vagyis a 2-2. ábra órajel-szinkronizáló fokozata itt is beépíthető a 4-2. táblázatban foglalt jelösszerendeléseknek megfelelően.

Általános CDC-logika	UART-vevő modul
$A_{i,C}$	*data_from_transmitter, parity_error*
$B_{k,C}$	-

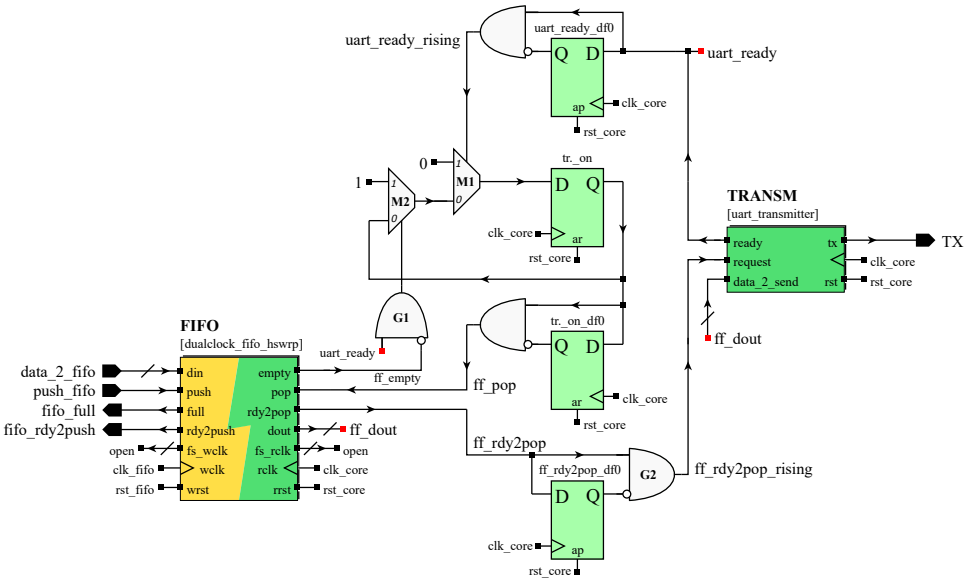
4-2. táblázat. *Az UART-vevő modul és az általánosított CDC-logika jeleinek megfeleltetése*

Figyeljük meg, hogy az adómodulnál látottakkal ellentétben itt a vevőmodul kezdeményező szerepben van, tehát az általánosított CDC-logika kliensoldali jeleit feleltetjük meg a vevő modul interfészjeleinek.

FIFO-alapú órajel-szinkronizáló burkolók ◆ A szinkron kézfogásos CDC-logikával tehát az UART áramkörök könnyen leválaszthatók környezetük órajeltartományáról, ez azonban nem az egyetlen elképzelhető módszer. Az alábbiakban egy, a 2.1. pontban általánosan megfogalmazott, FIFO-alapú megoldást is bemutatunk.

Egy két eszköz közötti UART-kapcsolatot az alkalmazások többsége egyfajta adatfolyam-csatornaként használ; az adó egy bájtfolyamot (pl. egy N db, logikailag összetartozó bájtból álló csomagot) küld el a vevőnek, aki ezt feldolgozza, tárolja, továbbítja stb. Ilyenkor az adóoldali felhasználó szempontjából nem feltétlenül fontos az, hogy egy konkrét bájt elküldésére mikor kerül sor, neki az számít, hogy az adatcsomagot a csatornába beküldhesse, amikor az számára kényelmes. A vevőoldali felhasználó számára pedig praktikus, ha az egyes bájtok beérkezésére nem kell feltétlenül azonnal reagálnia, hanem a saját tempójában olvashatja a csatorna kimenetét. Mindkét oldalon célszerű tehát egy-egy FIFO-tárolót elhelyezni; az adó egy FIFO-ba tölti be a küldendő adatfolyam bájtjait, a vevő pedig ugyancsak egy FIFO-ból olvassa ki a csatornán keresztül beérkezett bájtokat. A két FIFO ily módon a két fél közötti, ideiglenesen fennálló feldolgozásisebesség-különbséget is kompenzálni képes a tárkapacitásától függő mértékben. Az UART-adó és -vevő IP-magokhoz olyan burkolómodulokat készítünk, amelyekben a FIFO funkciókat egy-egy, a 4.2.5. pontban tárgyalt *dualclock_fifo_hswrp* modul tölti be, így azok a fenti előnyök biztosításán felül az adó- és vevőoldali felhasználók és az UART interfészáramkörök közötti CDC-feladatokat is ellátják.

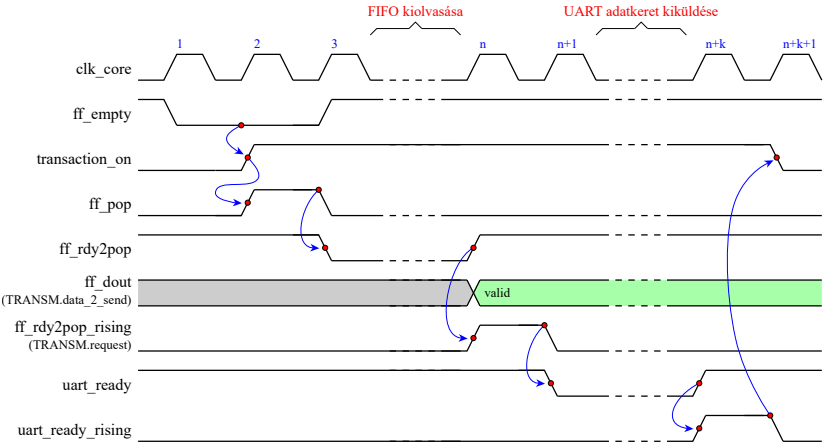
Az UART-adó burkolómoduljának szerepe tehát az, hogy egy bemeneti FIFO-ból folyamatosan kiolvassa a felhasználó által betöltött bájtokat, és azokat az UART-adónak továbbítsa. Mind



4-42. ábra. *Illesztőlogika az UART-adó bemeneti FIFO-ja és párhuzamos-soros átalakítója között*

a FIFO *pop*-interfésze, mind az UART-adó felhasználói interfésze kliens-típusú – vagyis magától egyik sem fog semmilyen adatátvitelt kezdeményezni – ezért kettejük közé egy illesztőlogikát kell tervezni. Erre egy lehetséges megoldást a 4-42. ábrán láthatunk (vö. 2-6. ábra).

A FIFO-ból a párhuzamos-soros átalakítóba való adatmozgatásra akkor kerülhet sor, ha az előbbi nem üres, az utóbbi pedig készen áll újabb adat fogadására. E két feltétel fennállását a 4-42. ábrán *G1*-gyel jelölt *AND* kapu figyeli. Ha a két feltétel teljesül, akkor a *transaction_on* állapotjelző egybe billen, jelezve, hogy megkezdődött egy adatbejegyzés mozgatása. Az állapotjelző kimenetére beépített felfutóél-detektor állítja elő a FIFO kiolvasásához szükséges pozitív impulzust az *ff_pop* jelen, aminek hatására a FIFO *rdy2pop* kimenete – a szinkron kézfogásos interfész-specifikációnak megfelelően – azonnal alacsony szintű lesz. Amikor a FIFO a *rdy2pop* kimenetét újra magas szintre húzza, akkor az UART-adó adatbemenetére közvetlenül rákapcsolt *ff_dout* buszon rendelkezésre áll az UART-adóba betöltendő adat. A FIFO *rdy2pop* jelének felfutóéle tehát alkalmas arra, hogy belőle – ugyancsak egy felfutóél-detektorral – előállítsuk az UART-adó *request* bemenetét meghajtó jelet *(ff_rdy2pop_rising)*. A ciklust az UART-adó *ready* kimenetének felfutása zárja; az e felfutóélből előállított pozitív impulzussal nullázzuk a *transaction_on* állapotjelzőt *(uart_ready_rising)*.[100] A FIFO-ból az UART-adóba való adatmozgatás folyamatát a 4-43. ábrán látható hullámforma szemlélteti.



4-43. ábra. *Adatmozgatás a FIFO-ból a párhuzamos-soros átalakítóba*

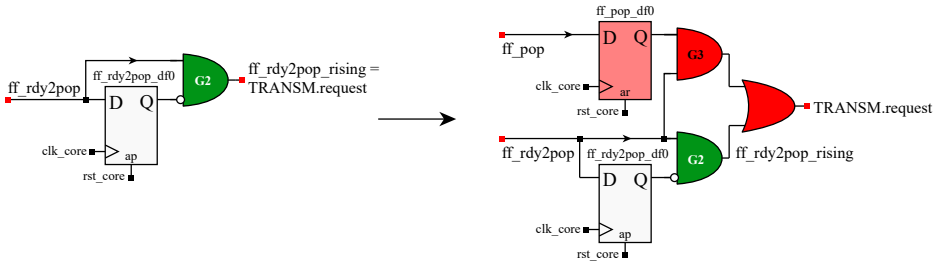
Az illesztőlogika két szinkron kézfogásos interfésszel ellátott almodul között teremt kapcsolatot. Egy apróságot azonban nem vettünk figyelembe; a FIFO almodul egy szintézis-paraméter megfelelő beállításával a szinkron kézfogásos interfésznek a nulla késleltetésű változatát is képes megvalósítani. Ebben az esetben a FIFO *rdy2pop* kimenete sosem lesz alacsony értékű, mert az órajelnek az *ff_pop* jel felfutóéle utáni legelső felfutóélénél rögtön előáll az adat az *ff_dout* buszon (vö. 4-19. ábra). Így az UART almodul *request* bemenetét hajtó éldetektor sosem jelez. Ennek az esetnek a kezelésére az illesztőlogikát a 4-44. ábrán látható elemekkel kell kiegészítenünk.

[100] Figyeljük meg, hogy a 4-42. ábrán az *M1* és *M2* multiplexerek által definiált prioritási sorrend miatt a *transaction_on* állapotjelző legalább egy órajelciklus időre biztosan alacsony szintű lesz, akkor is, ha a tranzakció végén a FIFO-ban továbbra is van adat, tehát *empty* kimenete mindvégig alacsony szintű. Így garantált, hogy az *ff_pop* jelen előáll a következő FIFO-olvasást kezdeményező pozitív impulzus.

VHDL

```
1  L_GLUE_LOGIC_EXTENSION: process ( clk_core, rst_core ) begin
2    if ( rst_core = '1' ) then ff_pop_df0 <= '0';
3    elsif ( rising_edge(clk_core) ) then ff_pop_df0 <= ff_pop;
4    end if;
5  end process;
6
7  uart_request <= ff_rdy2pop_rising or (ff_pop_df0 and ff_rdy2pop);
```

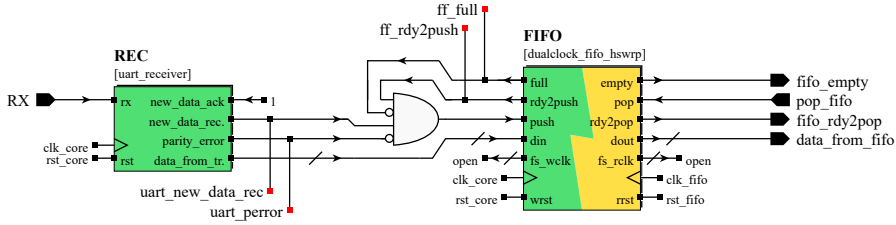
Áramköri struktúra



4-44. ábra. *Az illesztőlogika kiegészítése a nulla késleltetésű FIFO kezeléséhez*

Bevezetjük a FIFO *pop* bemenetét meghajtó éldetektor kimenetét egyetlen órajelciklussal megkésleltető *ff_pop_df0* flip-flopot. Ha e flip-flop kimenetének a felfutóélénél a FIFO *rdy2pop* jele még mindig magas szintű (ez csak akkor állhat elő, ha a FIFO kézfogás-logikája nulla késleltetésűre van beállítva), akkor a *G3* kapu kimenetén megjelenik az a pozitív impulzus, amit a korábbi megoldás nem tudott előállítani. A két impulzusgeneráló részáramkör kimeneteit egy *OR* kapun keresztül kapcsoljuk rá az UART-adó *request* bemenetére, így mindkettő hatása érvényesül.

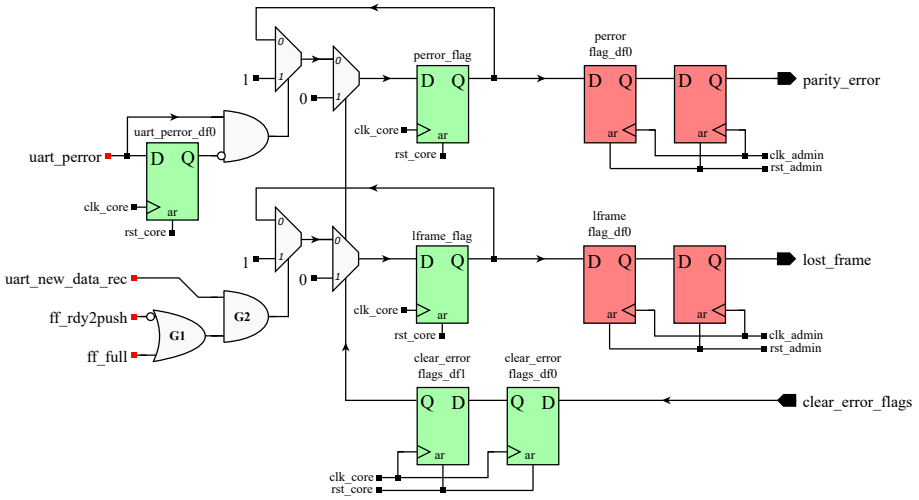
Az UART-vevő FIFO-alapú órajel-szinkronizáló burkolómoduljának szerkezete az adóéhoz nagyon hasonló; a soros-párhuzamos átalakító egy egyszerű illesztő logikával összekapcsolható *dualclock_fifo_hswrp* modullal (lásd 4-45. ábra).



4-45. ábra. *Az UART-vevő FIFO-alapú órajel-szinkronizáló burkolója (1)*

Az UART-vevő párhuzamos interfésze kiszolgáló típusú, hiszen minden adatkeret beérkezésekor egy-egy párhuzamos tranzakciót kezdeményez egy kiszolgáló, ez esetben egy FIFO felé. Ha a *new_data_ack* bemenetét állandó magas szintre kapcsoljuk, akkor ezek a kezdeményezések a *new_data_received* kimeneten megjelenő, egyetlen órajelciklus időtartamú pozitív impulzusokat jelentenek. Ezt a kimenetet felhasználhatjuk a FIFO *push* bemenetének meghajtására. A két port közvetlen összekapcsolása azonban több okból sem lehetséges; ha az adatkeret megérkezésekor a FIFO már tele van vagy még az előző adatkeret belső memóriába írásával van elfoglalva, tehát *rdy2push* jele alacsony, akkor a *push* bemenetet nem rángathatjuk, mert az ellentmondana a FIFO interfész használatára vonatkozó szabályoknak. Paritáshibás adatkeretet pedig nem szeretnénk elhelyezni

a FIFO-ban még akkor sem, ha maga a FIFO készen állna annak tárolására. Ezért az UART-vevő *new_data_received* kimenetét egy *AND* kapun keresztül kötjük a FIFO *push* bemenetére, ami mindezeket az ellenőrzéseket elvégzi.[101] A FIFO *push* interfészét az *AND* kapuval „megvédtük”, de ezzel a feladatot még nem végeztük el maradéktalanul; a paritáshibás adatkeret érkezéséről és az adattárolásra átmenetileg éppen képtelen FIFO miatti adatkeretvesztésről a felhasználót értesítenünk kell. Ezeknek az eseményeknek a rögzítésére egy-egy állapotjelzőt készítünk a 4-46. ábrán látható kapcsolási séma szerint.



4-46. ábra. *Az UART-vevő FIFO-alapú órajel-szinkronizáló burkolója (2)*

A vevő almodul paritáshibát jelző kimenetére helyezett felfutóél-detektor, illetve a *G1* és *G2* kapukból álló, egyszerű dekóderlogika a fent említett eseményeket egy-egy pozitív impulzussal jelzi. Ezek az impulzusok állítják magas szintre a *perror_flag* és *lframe_flag* állapotjelzőket, amelyek egy-egy double-flop szinkronizáló fokozaton keresztül hajtják a burkolómodul *parity_error* és *lost_frame* kimeneteit. Az állapotjelző kimeneteket a FIFO interfész órajelétől független órajelhez *(clk_admin)* szinkronizáljuk.[102] Az állapotjelzők törlésének lehetőségét természetesen ugyancsak biztosítanunk kell, ezt a célt szolgálja a *clear_error_flags* bemenet, amit a burkolómodul órajeléhez szintén egy double-flop fokozattal szinkronizálunk.

[101] Ne feledjük, hogy a FIFO kézfogásos burkolója csupán modellezni hivatott a lassú belső működésű FIFO almodulokat, amelyekkel esetleg később helyettesíteni szeretnénk ezt az egyszerű, blokk-RAM alapú kivitelt (vö. 4.2.3. pont). A *rdy-2push* kimenetet tehát nem hagyhatjuk figyelmen kívül mondván, hogy „az UART adatkeret fogadása biztosan több időt vesz igénybe, mint egy bejegyzés beírása a FIFO-ba”. Hasonló a helyzet a FIFO *full* jelzése esetén; a *dualclock_fifo_hswrp* almodul belsejében működő FIFO modellt úgy készítettük el, hogy a *full* kimenet magas értéke esetén végrehajtott *push* műveletet egyszerűen figyelmen kívül hagyja (vö. 4-26. ábra bal oldal: 7. sor). Erre a képességre azonban a lokalitás elve miatt nem támaszkodhatunk egy, a FIFO-t körülvevő áramkör tervezésénél. Nem tudhatjuk, hogy egy később beépítendő, nem általunk készített FIFO hogyan viselkedik ilyen esetekben.

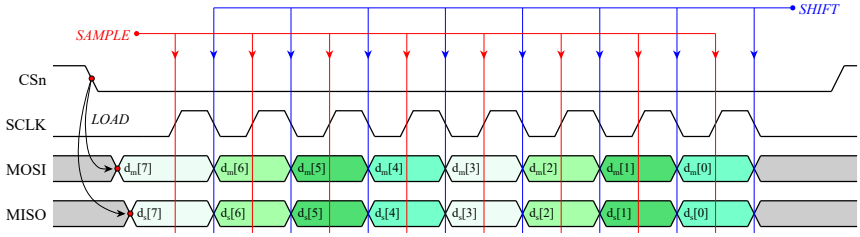
[102] Az órajel-szinkronizáló burkolómodulok tervezésekor feltételezzük, hogy az adott IP-mag különféle interfészeit más-más órajelről járó külső egységek használják. Az UART-vevőmodul példájában a FIFO olvasása, valamint a *lost_frame* és *parity_error* állapotjelzők monitorozása és törlése két különálló interfészen elérhető funkciók, amelyekhez saját órajelet rendelünk *(clk_fifo, clk_admin)*. Ha az adott felhasználási körülmények között nincs szükség ennyi órajelre (pl. mert mindkét interfészt ugyanaz a külső áramkör használja), akkor az órajelek az UART-vevő IP-magon kívül közösíthetők (lásd pl. 2-100. ábra).

4.3.2. SPI master és slave

Áttanulmányozásra javasolt alfejezetek:

5.2.2. *Egyszerű soros kommunikációs interfészek – SPI*
4.2. *On-chip adattárolás*

Ahogy az 5.2.2. Függelékben is olvasható, egy SPI-kapcsolatban egy adatcsere-tranzakció felfogható a master és a slave egységek belsejében kialakított, majd a *MISO* és *MOSI* vezetékek segítségével gyűrűbe kapcsolt léptetőregiszterek közötti bitenkénti tartalomcsereként, amelyet a master egység által előállított órajel ütemez a 4-47. ábrán látható időzítési séma szerint.



4-47. ábra. *Egy nyolcbites SPI-tranzakció időzítése*

Célszerű ezt a léptetőregiszter-sémát az RTL modellezés szintjén közvetlenül létrehozni. Figyeljük meg, hogy akár a master, akár a slave szempontjából vizsgáljuk a tranzakciót, olyan léptetőregiszterre van szükség, amelynek tartalma (minden bitje) egy vezérlőjellel egyetlen lépésben betölthető (4-47. ábra: *LOAD*), soros bemenetét egy másik vezérlőjel hatására képes mintavételezni (4-47.

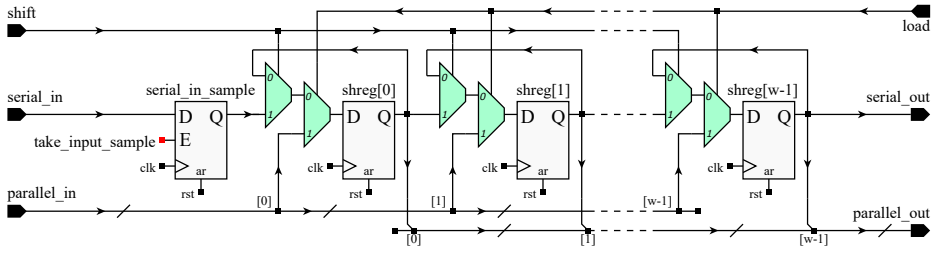
Verilog

```
1  // INPUT SAMPLE
2  always @ ( posedge clk, posedge rst ) begin
3    if ( rst == 1 ) serial_in_sample <= 0;
4    else if ( take_input_sample == 1 )
5           serial_in_sample <= serial_in;
6  end
7
8  // OUTPUT ASSIGNMENTS
9  assign serial_out = shreg[w-1];
10 assign parallel_out = shreg;
11
```

Verilog (folyt.)

```
12 // SHIFT-REGISTER
13 always @ ( posedge clk, posedge rst ) begin
14   if ( rst == 1 ) shreg <= 0;
15   else
16     if ( load == 1 ) shreg <= parallel_in;
17     else if ( shift == 1 ) begin
18           shreg[0] <= serial_in_sample;
19           for ( int i = 1; i < w; i = i + 1 )
20             shreg[i] <= shreg[i-1];
21         end
22 end
```

Áramköri struktúra

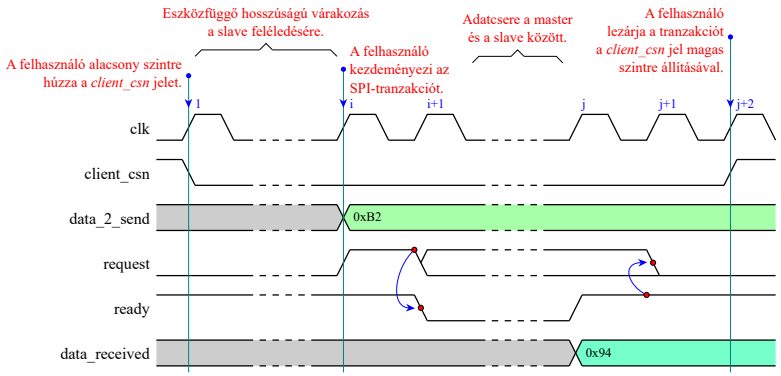


4-48. ábra. *Az SPI interfészáramkörök magját képező léptetőregiszter*

ábra: *SAMPLE*) és soros kimenetét egy harmadik vezérlőjel segítségével képes megváltoztatni, ami a léptetés funkcióval egyenértékűnek tekinthető (4-47. ábra: *SHIFT*). A 4-48. ábrán egy ezekkel a képességekkel felruházott léptetőregiszter RTL-modellje látható, amely változatlan formában, újrafelhasználható részegységként egyaránt beépíthető a master és slave IP-magba.

A léptetőregiszter egyes flip-flopjainak órajelbemenetét nem a master egység által generált *SCLK* jel, hanem egy attól független, lokális órajel hajtja meg. Az itt bemutatott master és slave IP-magok esetén azzal a feltételezéssel élünk, hogy ennek a lokális órajelnek a frekvenciája lényegesen nagyobb, legalább tíz-húszszorosa az SPI-kommunikációt ütemező *SCLK* jel frekvenciájának. Ez a feltételezés nyilván egy korlátot jelent az SPI-interfész adatátviteli sebességére vonatkozóan, amely korlát a konkrét alkalmazás jellegétől függően értékelendő. Erre a problémára később még visszatérünk, de előbb tekintsük át, miként lehet e léptetőregisztert felhasználva teljesértékű SPI master és slave IP-magokat készíteni.

SPI master ◆ A master egység *(basic_spi_master)* tervezésekor az áttekinthetőség kedvéért célszerű különválasztani a *CSn* és *SCLK* jelek generálását a léptetőregiszter vezérlésétől. Ha előállt egy olyan áramkör, ami képes a *CSn* és *SCLK* jelek 4-47. ábrán látható mintázatának létrehozására, akkor a léptetőregiszter betöltő *(load)* és léptető *(shift)* bemeneteit meghajtó logika beépítése már könnyen elvégezhető. A *CSn* jel esetén egy meglehetősen lusta, de annál rugalmasabb megoldást választunk; átadjuk a vezérlés lehetőségét az SPI master modul felhasználójának. Ez azt jelenti, hogy a master modulnak lesz egy *client_csn* bemenete a felhasználói interfészen és egy *CSn* kimenete, ami közvetlenül a slave egység *CSn* bemenetét hajtja meg, a kettőt pedig egyszerűen összekötjük a master IP belsejében. A *client_csn* jelen felül a felhasználói interfészen további négy jelet hozunk létre, amelyeket a 4-49. ábrán szemléltetett módon használva hajthatunk végre SPI adatcsere-tranzakciókat.

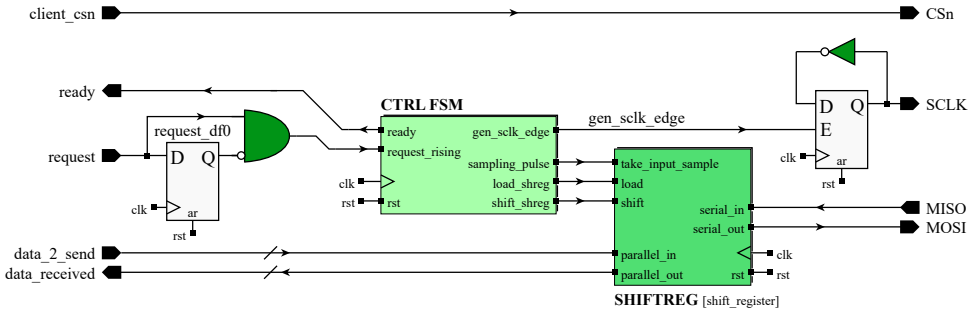


4-49. ábra. *Az SPI master modul felhasználói interfészének működése*

A *client_csn* jel tehát közvetlenül rá van kapcsolva a slave egység *CSn* bemenetére. Ennek alacsony szintre állításával a master megszólítja a slave eszközt. E lefutóél és az *SCLK* első felfutóéle között némi időnek kell eltelnie ahhoz, hogy a slave felkészülhessen a tranzakcióra. Ez a minimális időtartam a konkrét slave (pl. egy A/D-átalakító, szenzor stb.) jellemzője, amit a felhasználónak kell figyelembe vennie úgy, hogy a *client_csn* lehúzása és a tranzakció kezdeményezése – vagyis a *request*

jel felhúzása – közé némi várakozást iktat be.[103] Látható, hogy a master modul felhasználói interfésze – talán nem meglepő módon – valójában a szinkron kézfogásos interfésznek a specifikusan az SPI-ra jellemző *client_csn* jellel kibővített változata. A felhasználó a *data_2_send* buszra kapcsolja a slave-nek küldendő adatot, majd a *request* jel magas szintre állításával kezdeményezi az átvitelt. A *ready* jel felfutása jelzi az adatcsere végét, ekkor a *data_received* buszon a slave-től beolvasott adatot látjuk. Ezek után a felhasználó – ugyancsak némi eszközfüggő várakozás beiktatása után – magas szintre állítja a *client_csn* jelet, amivel a tranzakció befejezettnek tekinthető.

A teljes SPI master funkciót, vagyis az *SCLK* ciklusok generálását, valamint a *MOSI/MISO* vonalak megfelelő időzítésű meghajtását és mintavételezését egy egyszerű, one-shot típusú állapotgép és a 4-48. ábrán látható léptetőregiszter kombinációjával valósítjuk meg a 4-50. ábrán látható kapcsolási séma szerint.



4-50. ábra. *Az SPI master IP-mag (basic_spi_master) belső szerkezete*

Az ábrán jól látható, hogy a vezérlést végző állapotgép meghajtja a léptetőregiszter mintavételező *(take_input_sample)*, betöltő *(load)* és léptető *(shift)* bemeneteit, valamint a *gen_sclk_edge* jelen keresztül az *SCLK* kimenet állapotváltozásait is elő tudja idézni, vagyis minden adott ahhoz, hogy egy szabályos SPI-tranzakciót levezényeljen. A 4-51. ábra ennek az állapotgépnek a vázlatát mutatja be.

Az állapotgép leírásának 14. sorában láthatjuk, hogy az *SCLK* jelen állapotváltozást előidéző *gen_clk_edge* jelnek, valamint a léptetőregiszter vezérlőjeleinek a lokális órajel minden felfutóélénél adunk egy alapértelmezett alacsony értéket. Ezt az aktuális vezérlési állapot felülírhatja, így idézve elő egyetlen órajel-periódus ideig magas szintű vezérlőimpulzusokat az adott jelen (vö. 2-34. ábra: a *CPU* állapotgépe által generált vezérlőjelek működése). A tranzakció állapotát a 4-51. ábra 9. sorában piros színnel kiemelt két számlálóval követjük. Az első számláló *(sclk_period_counter)* szerepe, hogy a lokális, az *SCLK* jelénél nagyobb frekvenciájú órajelet leosztva meghatározza az *SCLK* kimenet periódusidejét. A leosztás mértékét a felhasználó egy szintézis-paraméter segítségével állíthatja be, így hangolva össze a lokális órajel frekvenciáját az SPI-interfész ütemjelének kívánt frekvenciájával. A másik számláló *(bit_counter)* – ahogy arra neve is utal – azt tartja számon,

[103] Részben az eszközfüggő várakozás miatt adjuk át a *CSn* jel vezérlését közvetlenül a felhasználónak; az újrafelhasználható SPI master IP-mag fejlesztőjeként a slave-függő feléledési időtartamot nem ismerhetjük. Persze lehetséges lenne a várakozást egy számlálóval az IP-magon belül megoldani, de akkor a számláló maximális értékének meghatározása ugyancsak a felhasználó feladata lenne, hiszen az IP-mag órajelének frekvenciáját – ami a számláló által megvalósított késleltetés mértékét definiálja – szintén csak ő tudhatja.

Verilog

```
always @ ( posedge clk, posedge rst ) begin       // REQUEST RISING EDGE DETECTOR
  if ( rst == 1 ) request_df0 <= 1'b0;
  else request_df0 <= request;
end
assign request_rising = request & ~request_df0;

always @ ( posedge clk, posedge rst ) begin       // CTRL FSM
  if ( rst == 1 ) begin
    state <= idle; ready <= 1; sclk_period_counter <= 0; bit_counter <= 0;
    gen_sclk_edge <= 0; sampling_pulse <= 0; load_shreg <= 0; shift_shreg <= 0;
  end else begin

    // default values of the pulse-like control signals...
    gen_sclk_edge <= 0; sampling_pulse <= 0; load_shreg <= 0; shift_shreg <= 0;

    case ( state )

      idle:    if ( request_rising == 1 ) begin [...]

      [...]

      default: ;

    endcase

  end
end
```

4-51. ábra. *Az SPI master modul vezérlő állapotgépének vázlata*

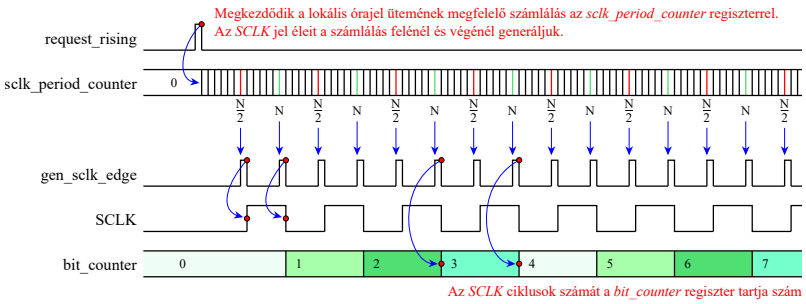
Verilog

```
idle: if ( request_rising == 1 )
begin
  ready <= 0;
  state <= s1;
end

```

Verilog (folyt.)

```
s1: if ( sclk_period_counter < N ) begin
      sclk_period_counter <= sclk_period_counter + 1;
      if ( sclk_period_counter == N / 2 ) gen_sclk_edge <= 1;
    else begin
      sclk_period_counter <= 0; gen_sclk_edge <= 1;

      if ( bit_counter < frame_size-1 ) begin
         bit_counter <= bit_counter + 1;
      end else begin
        bit_counter <= 0;
        state <= s2;
      end

    end
end

s2: [...]
```

Hullámforma



4-52. ábra. *Az SPI master modul állapotgépének működése (1)*

hogy az adott tranzakcióban eddig hány *SCLK* impulzust generáltunk, vagyis a master és a slave egységek léptetőregisztereinek bitjei közül hány cserélődött már ki. A 4-52. ábra az állapotgépnek az *SCLK* ciklusok generálását végző vezérlési állapotát mutatja be.

A *request* jel felfutásának hatására – a szinkron kézfogásos interfész-specifikációnak megfelelően – alacsony szintre húzzuk a *ready* jelet (4-52. ábra: 3. sor), majd az *s1* állapotba lépünk, ahol megkezdődik az *sclk_period_counter* regiszter órajelciklusonkénti inkrementálása. (4-52. ábra: 8. sor). Ha a számlálás eléri a szintézis-paraméterként megadott *N* értékének felét (4-52. ábra: 9. sor), vagy magát az *N* értéket (4-52. ábra: 10. sor), akkor az *SCLK* kimeneten a *gen_sclk_edge* vezérlőjelen előállított impulzussal előidézünk egy élváltást. Amikor az *sclk_period_counter* eléri N értékét, vagyis amikor lezajlott egy *SCLK* ciklus, akkor a *bit_counter* értékét eggyel növeljük (4-52. ábra: 14. sor). Az egy SPI-tranzakcióban átvitt bitek száma ugyancsak szintézis-paraméter *(frame_size)*. Ha az *SCLK* ciklusok száma eléri ezt az értéket (4-52. ábra: 15. sor), akkor továbblépünk az *s2* állapotba, ami az SPI-tranzakció végét jelenti. Az eddig elmondottak alapján az állapotgépünk tehát már képes arra, hogy az átviendő bitek számának megfelelő mennyiségű *SCLK* impulzust állítson elő, de az adatvonalak mintavételezése és léptetése így még nem valósul meg. A 4-53. ábrán azt láthatjuk, hogy miként kell kiegészíteni az állapotgép leírását ahhoz, hogy a megfelelő pillanatokban a léptetőregiszter léptetése – egyúttal a *MOSI* kimenet megváltoztatása –, valamint a *MISO* bemenet mintavételezése is megtörténjen.

Az ábrán jól látható, hogy a léptetőregiszter vezérlőbemeneteit meghajtó jelekre vonatkozó értékadások megfelelő elhelyezésével az SPI master modul funkciója könnyen teljessé tehető. A *request* jel felfutásakor a *load_shreg* jelen előállított impulzussal a léptetőregiszterbe betöltjük a *data_2_send* bemenet tartalmát, vagyis azt az adatot, amit a felhasználó el szeretne küldeni a slave eszköznek. Az *s1* állapotban – miközben a két számláló inkrementálódik – egy-egy pozitív impulzust generálunk a *sampling_pulse* jelen az *SCLK* felfutó, a *shift_shreg* jelen pedig annak lefutó éleinél. Így az *SCLK* jel felfutó éleinél megtörténik a *MISO* bemenet beolvasása a léptetőregiszter *serial_in_sample* flip-flopjába, az *SCLK* lefutó éleinél pedig ezt a mintát, és a léptetőregiszterben már bent lévő biteket is eggyel léptetjük, aminek hatására a *MOSI* kimenet is frissül. A tranzakció végén, amikor a *bit_counter* számláló eléri a *frame_size* paraméter értékét (4-53. ábra: 19–21. sor), akkor az utolsó *shift_shreg* impulzus hatására a léptetőregiszterben előáll a slave egységtől beérkezett teljes adatkeret. Mivel a léptetőregiszter teljes tartalma a *parallel_out* kimeneten keresztül közvetlenül a master egység *data_received* kimeneti buszára van kapcsolva, ezért az *s2* állapotban nincs más dolgunk, mint magas szintre húzni a *ready* jelet, értesítve a felhasználót a tranzakció befejezéséről (4-53. ábra: 27. sor).

SPI slave ◆ A slave-oldali soros-párhuzamos átalakító *(basic_spi_slave)* tervezését is a felhasználói interfész felől közelítve tárgyaljuk. Az SPI-tranzakciót mindig a master kezdeményezi, ezért a slave-oldali felhasználói interfész feladata, hogy a tranzakció kezdetekor átvegye a master számára küldendő adatot, illetve hogy jelezze a tranzakció végét, egyúttal továbbítsa a master által küldött adatot a felhasználónak.

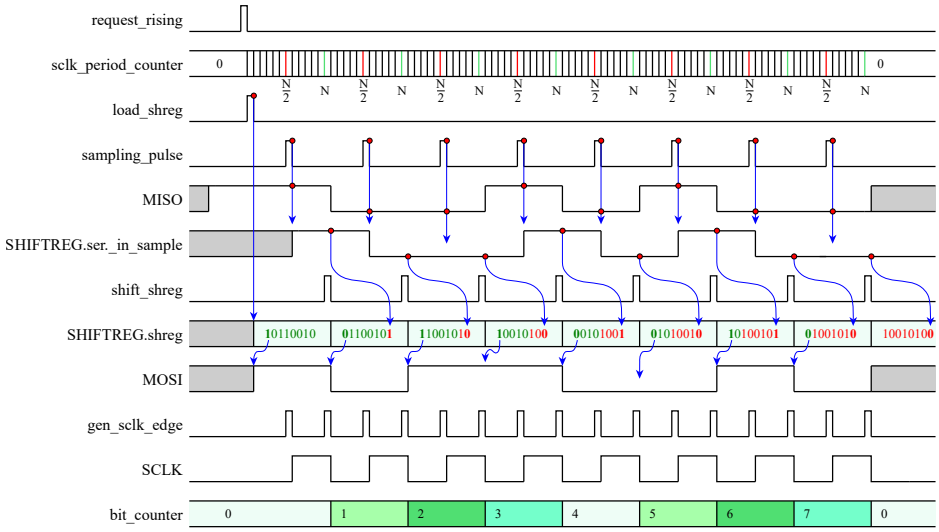
Verilog

```
1  idle: if ( request_rising == 1 )
2  begin
3  |  ready <= 0;
4  |  load_shreg <= 1;
5  |  state <= s1;
6  end
7
```

Verilog (folyt.)

```
8  s1: if ( sclk_period_counter < N ) begin
9  |     sclk_period_counter <= sclk_period_counter + 1;
10 |     if ( sclk_period_counter == N / 2 ) begin
11 |       gen_sclk_edge <= 1; sampling_pulse <= 1;
12 |     end
13 |   else begin
14 |     sclk_period_counter <= 0; gen_sclk_edge <= 1;
15 |     shift_shreg <= 1;
16 |
17 |     if ( bit_counter < frame_size-1 ) begin
18 |        bit_counter <= bit_counter + 1;
19 |     end else begin
20 |       bit_counter <= 0;
21 |       state <= s2;
22 |     end
23 |
24 |   end
25 end
26
27 s2: begin ready <= 1; state <= idle; end
```

Hullámforma



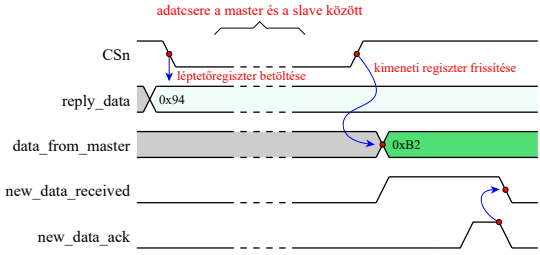
4-53. ábra. *Az SPI master modul állapotgépének működése (2)*

- A tranzakció kezdetét a *CSn* jel lefutása jelzi. A felhasználói interfészen létrehozunk egy *reply_data* bemenetet, amelyet a *CSn* jel lefutóélének detektálásakor a slave modul betölt a saját, belső léptetőregiszterébe annak *load* vezérlőbemenete segítségével.[104]
- A tranzakció végét a *CSn* jel felfutása jelzi. A master modul működésének tárgyalásakor láthattuk, hogy ekkorra a master és slave egységek léptetőregisztereinek tartalma már teljesen kicserélődött, vagyis a slave modul léptetőregiszterének *parallel_out* kimenetén a master által küldött adatot kell látnunk. A *CSn* jel felfutását detektálva ezt az adatot egy kimeneti regiszterbe töltjük, amelyet a slave modul *data_from_master* kimenetére kap-

[104] A betöltés pillanatában ennek az adatnak már rendelkezésre kell állnia, ami úgy oldható meg, hogy a slave modul felhasználója azt már az előző SPI-tranzakció végén előkészíti, vagy úgy, hogy folyamatos frissítéssel karbantart egy regisztert, amely mindig a master felé küldendő adatot tartalmazza.

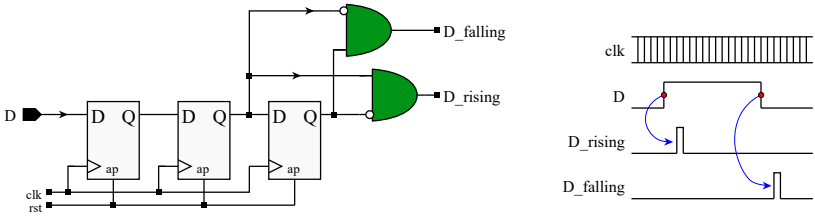
csolunk. A tranzakció végéről természetesen értesítenünk is kell a felhasználót, ezért létrehozunk egy *new_data_received* állapotjelzőt, amit a kimeneti adatregiszter betöltésével egyidőben magas szintre állítunk, illetve egy, az állapotjelző nullázására szolgáló bemenetet *(new_data_ack)*.

Egy SPI-tranzakció tehát a slave egység felhasználója szempontjából a 4-54. ábrán látható hullámforma szerint alakul.



4-54. ábra. *Egy SPI-tranzakció a slave egység felhasználója szempontjából*

A slave modulban a *CSn* és az *SCLK* bemenetek élei idéznek elő változásokat, ezért célszerű ezeket az eseményeket a slave egység órajeléhez szinkronizált impulzusokká alakítani. Ezt egyszerű éldetektorokkal megtehetjük, ne feledjük azonban, hogy a *CSn* és *SCLK* jeleket a master egység állítja elő egy másik eszközben, a saját lokális órajeléhez szinkronizáltan. A *CSn* és *SCLK* jelek tehát a slave oldalon aszinkron bemeneteknek tekintendők, amiket a lokális órajelhez szinkronizálni kell. Ezért az éldetektorokat egy-egy double-flop szinkronizáló fokozattal kell kiegészítenünk a 4-55. ábrán látható séma szerint.[105]



4-55. ábra. *A lokális órajelhez képest ritkán változó (statikus) jelek változásainak átalakítása szinkronizált impulzusokká*

A *CSn* és *SCLK* jelek éleiből előállított vezérlőimpulzusok szerepe a következő:

- A *CSn* lefutóéléből generált impulzus a felhasználó által megadott *reply_data* vektor értékét betölti a léptetőregiszterbe.

[105] Figyeljük meg, hogy a double-flop szinkronizáló fokozatok és az éldetektorok némi késleltetést okoznak; az adott bemenőjel változásait jelző impulzusok késve követik a tényleges változásokat. E késleltetés mértéke a lokális órajel periódusidejétől függ, ráadásul – a double-flop szinkronizáló fokozat működési mechanizmusából adódóan – egy órajel-periódusnyi jitter-rel terhelt (vö. DRMSz: 6-28. ábra). Ennek mindaddig nincs jelentősége, amíg az *SCLK* jel periódusideje sokszorosa a lokális órajel periódusidejének. Erre a kérdésre az alfejezet végén még visszatérünk.

- A *CSn* felfutóéléből előállított vezérlőimpulzus az SPI-tranzakció végén a léptetőregiszter tartalmát áttölti a *data_from_master* kimeneti regiszterbe.
- Az *SCLK* felfutóéléből generált vezérlőimpulzus mintavételezi a *MOSI* jelet.
- Az *SCLK* lefutóéleiből előállított vezérlőimpulzus lépteti a léptetőregisztert, egyúttal frissíti a *MISO* kimenet értékét.

Mindezek alapján a funkcionálisan teljes slave-oldali soros-párhuzamos átalakító a 4-56. ábrán látható módon építhető fel.

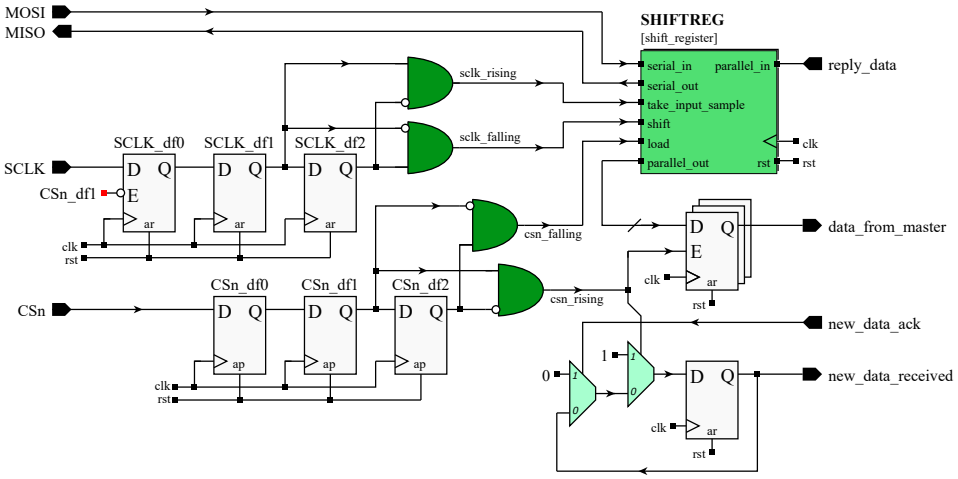
Verilog

```
always @ ( posedge clk, posedge rst ) begin
  if ( rst == 1 ) begin
    CSn_df0 <= 1; CSn_df1 <= 1; CSn_df2 <= 1;
    SCLK_df0 <= 0; SCLK_df1 <= 0; SCLK_df2 <= 0;
    new_data_received <= 0; data_from_master <= 0;
  end else begin

    CSn_df0 <= CSn;
    CSn_df1 <= CSn_df0;
    CSn_df2 <= CSn_df1;

    if ( CSn_df1 == 0 ) SCLK_df0 <= SCLK;
    SCLK_df1 <= SCLK_df0;
    SCLK_df2 <= SCLK_df1;

    if ( csn_rising == 1 ) new_data_received <= 1;
    else if ( new_data_ack == 1 )
      new_data_received <= 0;

    if ( csn_rising == 1 )
      data_from_master <= shreg_parallel_out;

  end
end

```

Verilog (folyt.)

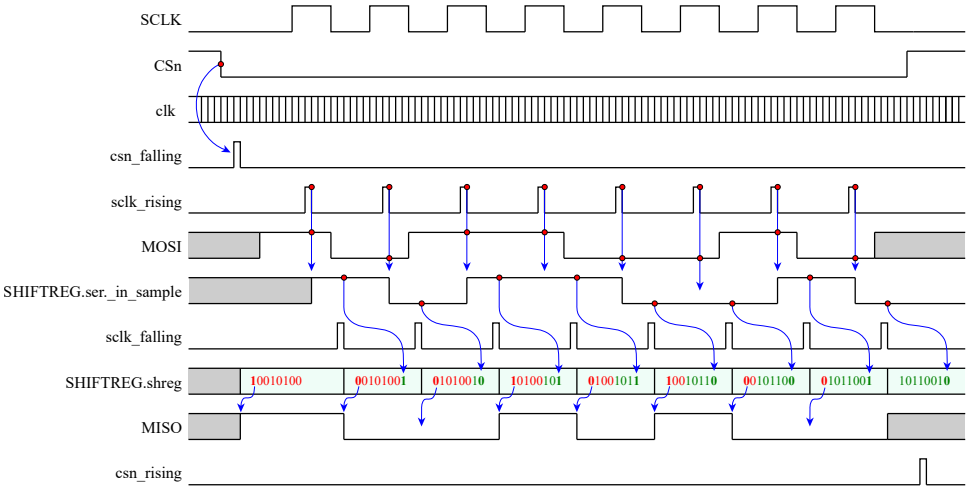
```
assign csn_falling = CSn_df2 & ~CSn_df1;
assign csn_rising = CSn_df1 & ~CSn_df2;
assign sclk_falling = SCLK_df2 & ~SCLK_df1;
assign sclk_rising = SCLK_df1 & ~SCLK_df2;

shift_register
#(frame_size) L_SHIFTREG (
  .clk(clk),
  .rst(rst),
  .take_input_sample(sclk_rising),
  .load(csn_falling),
  .shift(sclk_falling),
  .serial_in(MOSI),
  .serial_out(MISO),
  .parallel_in(reply_data),
  .parallel_out(shreg_parallel_out)
);
```

Áramköri struktúra



4-56. ábra. *Az SPI slave modul Verilog modellje és RTL kapcsolási sémája*

A 4-57. ábra a slave modul belső jeleinek változásait szemlélteti egy teljes SPI-tranzakció során.



4-57. ábra. *Egy teljes SPI-tranzakció a slave egység szempontjából*

4.3.2.1. Órajel-szinkronizálás

Láthattuk, hogy az SPI-interfész-áramkörök órajel-frekvenciája szorosan összefügg az *SCLK* jel frekvenciájával, amit az elvárt kommunikációs sebesség határoz meg. A master oldalon az *SCLK* jel a lokális órajelből egy frekvenciaosztás révén áll elő, vagyis a lokális órajel periódusideje az *SCLK* félperiódusának egész számú többszöröse kell hogy legyen. A slave esetén ez az összefüggés abban áll, hogy a lokális órajel frekvenciája jócskán meg kell hogy haladja az *SCLK* jel frekvenciáját az órajel-szinkronizáló fokozatok késleltetése miatt. Tehát a rendszerszinten az SPI-interfész sebességére vonatkozóan megfogalmazott kívánalmak egyúttal a master és slave IP-magok lokális órajelére vonatkozóan is kötöttséget jelentenek.

Ugyanarra a következtetésre juthatunk tehát, amire az UART interfészáramkörök esetén; célszerű lehetővé tenni, hogy az SPI soros-párhuzamos/párhuzamos-soros átalakítók is különálló órajeltartományban működhessenek. A master egység esetén egyszerű dolgunk van, mivel felhasználói interfészének működése megfelel a 1.2.1.1. pontban bemutatott szinkron kézfogásos interfész-specifikációnak, vagyis használhatjuk a 2-2. ábra órajel-szinkronizáló fokozatát. Az általánosított $A_{i,S}$ és $B_{k,S}$ vektorok a 4-3. táblázat szerint feleltethetők meg az SPI master modul interfészjeleinek.

Általános CDC-logika	SPI master modul
$A_{i,S}$	*data_2_send*
$B_{k,S}$	*data_received*

4-3. táblázat. *Az SPI master modul és az általánosított CDC-logika jeleinek megfeleltetése*

A slave modul órajel-szinkronizáló burkolójánál az UART-vevőnél látottakhoz hasonlóan járhatunk el. Az SPI slave modul feladata tulajdonképpen nem más, mint hogy a master által a soros

SPI protokoll szerint végrehajtott adatátviteli tranzakciókat párhuzamosakká alakítsa. Az egyes SPI-tranzakciók végén a *new_data_received*, *new_data_ack*, *data_from_master* és a *reply_data* jelek alkotta interfészen adatcserét kezdeményez a saját felhasználója felé. E jelek működése ugyancsak megfelel a szinkron kézfogásos interfész-specifikációnak, vagyis a 2-2. ábra órajel-szinkronizáló fokozata itt is beépíthető a 4-4. táblázatban foglalt jelösszerendeléseknek megfelelően.

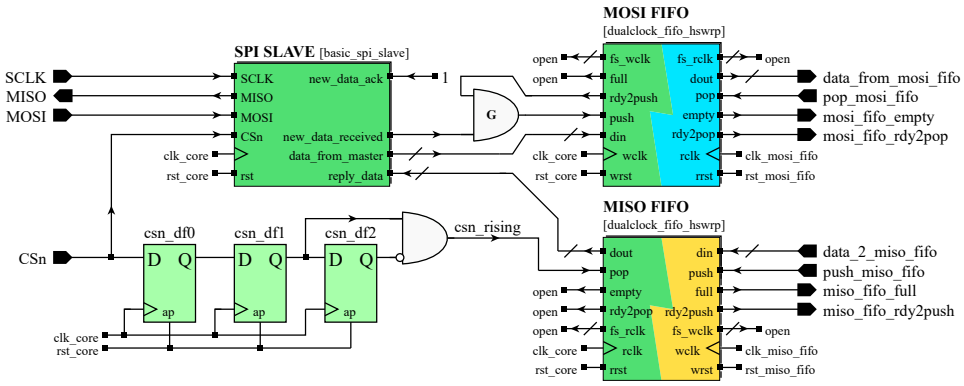
Általános CDC-logika	SPI slave modul
$A_{i,C}$	*data_from_master*
$B_{k,C}$	*data_received*

4-4. táblázat. *Az SPI slave modul és az általánosított CDC-logika jeleinek megfeleltetése*

Mivel a slave modul – az UART-vevőhöz hasonlóan – kezdeményező szerepben van, az általánosított CDC-logika kliens-oldali jeleit feleltetjük meg a slave modul interfészjeleinek.

FIFO-alapú órajel-szinkronizálás ◆ Minden SPI-tranzakcióban két adatkeret átvitele történik meg; a master küld egy adatot a slave-nek, a slave pedig visszaküld egy korábban – praktikusan a megelőző SPI-tranzakció végét jelző *CSn* felfutóéle hatására – előkészített másik adatot a master-nek. Bizonyos alkalmazásokban az egymást követő SPI-tranzakciókban átvitt adatkereteket egymáshoz lazán csatolt, vagy akár egymástól teljesen független adatfolyamoknak is tekinthetjük. Ahogy azt az UART-adó és -vevő modulok példájában is láthattuk, az ilyen adatfolyamokat kezelő áramköri részletek közötti órajel-szinkronizálásra a dual-clock FIFO modulok is kiválóan alkalmasak. Az alábbiakban az SPI slave egység FIFO-alapú órajel-szinkronizáló burkolójának egy lehetséges megvalósítását mutatjuk be részletesen.

Az SPI slave modul FIFO-alapú órajel-szinkronizáló burkolója a master felől érkező adatfolyamot egy dual-clock FIFO-ba *(MOSI FIFO)* tölti be, a visszaküldendő adatfolyamot pedig egy másik dual-clock FIFO-ból *(MISO FIFO)* olvassa ki. E két FIFO modul felhasználó felé kiajánlott interfészei egymáshoz és az SPI slave moduléhoz képest is aszinkron órajelekről működtethetők. A 4.2.5. pontban bemutatott *dualclock_fifo_hswrp* IP-magból példányosított modulok az SPI slave egységhez egyszerűen illeszthetők a 4-58. ábrán látható módon.

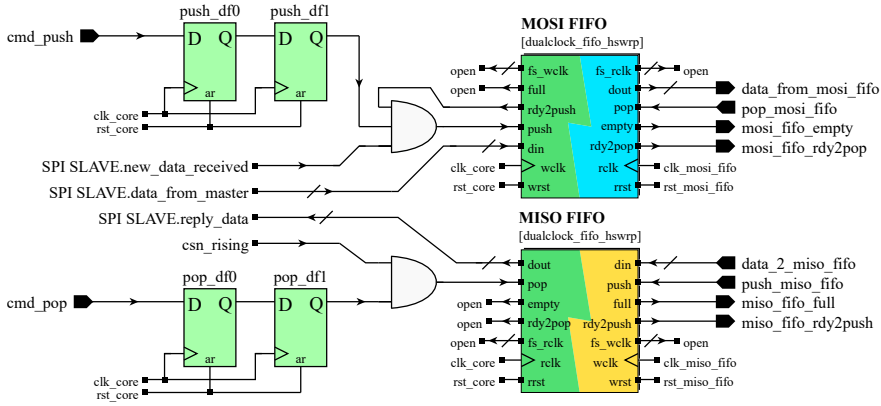


4-58. ábra. *A dualclock_fifo_hswrp IP-mag illesztése az SPI slave modulhoz*

A slave modul *new_data_ack* bemenetét konstans magas szintre állítjuk, így minden SPI-tranzakció azzal végződik, hogy a slave modul *new_data_received* kimenetén egy egyetlen periódusnyi pozitív impulzus jelenik meg, amelynek felfutóélénél a *data_from_master* buszon előáll a master által küldött adat. Ha ebben a pillanatban a *MOSI FIFO* készen áll új adat fogadására, vagyis a *rdy2push* kimenete magas szintű, akkor a slave modul *new_data_received* kimenetén előálló impulzus megjelenik a FIFO *push* bemenetén a *G AND* kapun keresztül. A *MOSI FIFO* adatbemenetét a slave modul *data_from_master* kimenetével összekötve azt érjük el, hogy az SPI-tranzakciók végén a master által küldött adat automatikusan beíródik a *MOSI FIFO*-ba. A *MISO FIFO* tartalmának eljuttatása a master-oldalra kicsit körülményesebb. Az egyértelmű, hogy a *MISO FIFO* adatkimenetét rá kell kapcsolnunk a slave modul *reply_data* bemenetére. A slave modul a *reply_data* bemenetét a *CSn* jel lefutóélénél tölti be a belső léptetőregiszterébe, vagyis az SPI-tranzakció legelső – a slave számára előre nem látható – mozzanatakor a *MISO FIFO* adatkimenetén már ott kellene lennie a master felé küldendő adatnak. Ezt a problémát úgy oldjuk meg, hogy a *MISO FIFO* kiolvasását a *CSn* jel felfutóéléhez, vagyis az SPI-tranzakciók végéhez kötjük. Ne feledjük, hogy a *CSn* jel egy aszinkron bemenet, tehát a felfutóél-detektort double-flop szinkronizálóval kell kiegészítenünk, ahogy tettük azt a slave modul belsejében is. A felfutóél-detektor kimenetét a *MISO FIFO pop* bemenetére kapcsolva azt érjük el, hogy minden SPI-tranzakció végén egyúttal előkészítjük a következőt azáltal, hogy a *MISO FIFO*-ból kiolvasunk egy adatot és azt a slave modul *reply_data* bemenetére juttatjuk. Jogosan merülhet fel a kérdés, hogy mi a helyzet a reset utáni legelső SPI-tranzakcióval. Ki készíti elő a legelső átküldendő adatot? Nos, a rövid válasz az, hogy senki; az első tranzakcióban a master felé küldött adat a *MISO FIFO* adatkimenetének alapértéke lesz. Ez az érték a *dualclock_fifo_hswrp* modul HDL-modelljéből adódóan szimulációs környezetben határozatlan vagy definiálatlan (vö. 4-26. ábra jobb oldal: 4–5. sor: nincs a *dout* adatkimenetre vonatkozó értékadás), ami egy szintetizált, fizikailag megvalósított áramkörben azt jelenti, hogy az adat véletlenszerű, gyakorlatilag bármi lehet. Ezt a problémát rendszerszinten többféleképpen is kezelhetjük. Megtehetjük, hogy az első tranzakció során a master-oldalon visszakapott adatot egyszerűen eldobjuk, tudomásul véve, hogy az első tranzakció csak arra való, hogy előcsalogassa az adatot a slave-oldali FIFO-ból. Egy másik megoldás, hogy a master modul segítségével a *CSn* jelen előállítunk egy lefutóél → felfutóél párost anélkül, hogy a kettő között SPI-tranzakciót kezdeményeznénk. A *CSn* lefutóéle így még azelőtt kiolvassa az első adatot a slave-oldali *MISO FIFO*-ból, hogy valódi SPI-tranzakcióra került volna sor. Nem szabad azonban elfelejteni, hogy a *CSn* jel felfutóéle a slave modul *new_data_received* kimenetén ebben az esetben is egy pozitív impulzus generálásával jár, tehát a *MOSI FIFO*-ban meg fog jelenni egy nem kívánt bejegyzés, amit a slave-oldali felhasználónak el kell távolítania, mielőtt a hasznos adatcsere elindul.

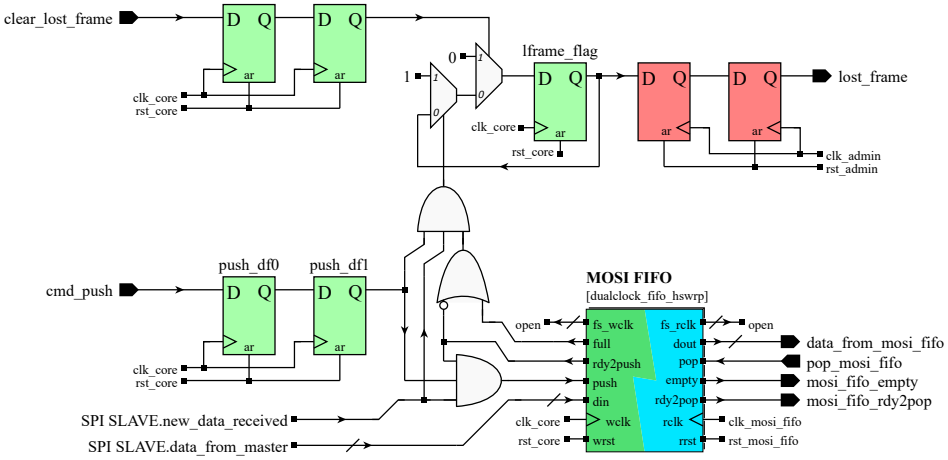
A 4-58. ábrán látható egyszerű áramkör adatfolyam-jellegű információcserét tesz lehetővé, de a két adatfolyam csak egyszerre vihető át, nincs mód csak az egyik, vagy csak a másik ág működtetésére; a slave-oldali *MOSI FIFO* írása mindig együtt jár a *MISO FIFO* olvasásával. A két irányt két vezérlőjel bevezetésével könnyen szétválaszthatjuk (lásd 4-59. ábra).

A *cmd_push* és *cmd_pop* jelekkel kapuzva a slave-oldali FIFO-k *push* és *pop* bemeneteit vezérelhetővé tettük, hogy épp melyik adatátviteli ágat szeretnénk működtetni. A *cmd_push* és *cmd_pop* bemeneteken a double-flop szinkronizáló fokozatokra azért van szükség, mert a slave IP tervezőjeként nem tudhatjuk, hogy e két vezérlő bemenetet ki és milyen órajelről hajtva használja majd.
A 4-59. ábra kapcsolási rajzán láthatjuk, hogy a slave modul FIFO-inak telítettségét jelző *full* és *empty* kimeneteket nem használjuk semmire. Ez a kialakítás tehát teljes körű forgalomszabályozásra



4-59. ábra. *Vezérlőjelek bevezetése a két adatátviteli irány egymástól független engedélyezéséhez/tiltásához*

semmiképp nem alkalmas, ehhez ui. a master modulnak ismernie kellene e két állapotjelző mindenkori értékét (vö. 1.2.3.5. pont: a funkcióbővítő rendszer üzemeltetői SPI-interfésze). A teljeskörű forgalomszabályozás biztosítása ugyan nem célunk, azonban a slave modulnak – az UART-vevő FIFO burkolójánál látottakhoz kísértetiesen hasonlító (vö. 4-45. ábra) – kis kiegészítésével lehetővé tehetjük a master → slave irányú adatvesztés slave-oldali detektálását (lásd 4-60. ábra).



4-60. ábra. *Master → slave irányú adatvesztés detektálására alkalmas kiegészítő logika*

A master → slave irányú adatvesztés azt jelenti, hogy egy SPI-tranzakció végén a slave modul nem tudja beírni a kapott adatot a *MOSI FIFO*-ba, mert az már az előző tranzakció során megtelt. A 4-60. ábrán látható kombinációs hálózat segítségével ez a szituáció könnyen detektálható, és létrehozható egy állapotjelző kimenet *(lost_frame)*, amelyen keresztül a slave-oldali felhasználó értesíthető. Az állapotjelzőt a két FIFO-interfész órajeleitől független órajelhez *(clk_admin)* szinkronizáljuk. Az állapotjelző-kimenethez természetesen létre kell hoznunk az annak nyugtázására szolgáló bemenetet is *(clear_lost_frame)*.

4.3.3. I²C master

Áttanulmányozásra javasolt alfejezetek:

5.2.3. *Egyszerű soros kommunikációs interfészek – I²C*

Az I²C interfész-protokoll sajátossága, hogy a fizikai vezetékeket háromállapotú (tri-state) buszokként használja, illetve hogy a master és a slave egységek egyaránt képesek írni és olvasni az *SCL* és az *SDA* vezetéket. Az *SDA* vezeték esetén a helyzet világos; ez hordozza az adatot mindkét irányban, tehát a master és a slave modulok a tranzakciók különböző fázisaiban (lásd 5.2.3. Függelék) felváltva írják és olvassák azt. Az *SCL* vezeték értékét alapesetben a teljes tranzakció során a master határozza meg, a kétirányúságra ebben az esetben csak az órajelnyújtás mechanizmusa miatt van szükség. Egy adatbit átvitele során a megcímzett slave egység „időt kérhet" az *SCL* vezeték alacsony szinten tartásával, rákényszerítve a master-t az *SDA* vezeték mintavételezésének elhalasztására. Ez a mechanizmus természetesen csak akkor működik, ha a master nemcsak írni, hanem olvasni is képes az *SCL* vezeték értékét.

Az I²C-interfésznek e sajátosságait a HDL-modellezés során is figyelembe kell vennünk annak érdekében, hogy a logikai szimulációban tapasztalt viselkedés minél jobban hasonlítson arra, amire a majdani valós áramköri környezetben is számíthatunk. A 4-61. ábra szemlélteti az I²C-jelek master IP-n belüli és tesztkörnyezetbeli VHDL-leírását.

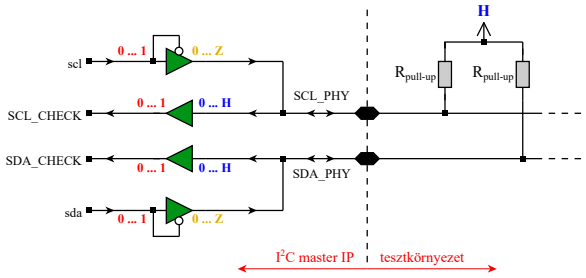
VHDL – (I²C master IP)

```
1  SCL_PHY <= '0' when scl = '0' else 'Z';
2  SCL_CHECK <= '0' when SCL_PHY = '0' else '1';
3  SDA_PHY <= '0' when sda = '0' else 'Z';
4  SDA_CHECK <= '0' when SDA_PHY = '0' else '1';
```

VHDL – tesztkörnyezet

```
1  SCL_PHY <= 'H';
2  SDA_PHY <= 'H';
```

Áramköri struktúra



4-61. ábra. *Az I²C-busz modellezése a master egységben és annak tesztkörnyezetében*

A VHDL *ieee* könyvtárában deklarált *std_logic* nevű felsorolt típus a szokásos 0, 1, Z (nagyimpedanciás) és X (határozatlan) értékeken kívül további öt különböző értéket definiál, amelyek pontosan az ilyen helyzetek leírására használhatók. Az I²C-busz két vezetékének – a master és slave egységektől független – alapértékét a tesztkörnyezet részének tekintett felhúzó ellenállások határozzák meg. A tesztkörnyezet folytonos értékadásai H, vagyis gyenge magas szintet rendelnek a vezetékeket reprezentáló jelekhez. A „gyenge" kifejezés arra utal, hogy ha bármely más szereplő a szokásos (erős) 0 szintet kényszeríti az adott vezetékre, akkor ez a 0 „győz" a gyenge felhúzással szemben, a jel tehát 0 értékű lesz. Mivel sem a master, sem a slave nem kényszerítheti erős magas

szintet a vezetékekre, ezért a busz állapota csak 0 vagy H lehet. Az I²C master modul szempontjából mindkét port kétirányú *(inout)*, hiszen mindkettőt írja és olvassa is. E két port a busz mindenkori állapotát reprezentálja *(SCL_PHY, SDA_PHY)*. A modul belsejében e portok írását és olvasását élesen szétválasztjuk egymástól:

- Íráskor az *scl/sda* jelnek adunk értéket, a 4-61. ábra bal oldalának 1. és 3. sorában lévő folytonos értékadások pedig gondoskodnak arról, hogy az erős 1 érték helyett Z (nagyimpedanciás) érték jelenjen meg a külső buszon. Ezzel tulajdonképpen azt fejezzük ki, hogy a modul háromállapotú meghajtón keresztül hajtja meg a külső buszt, és a logikai magas szintet a busz elengedésével szándékozik előidézni.
- A buszjelek olvasására létrehozzuk a belső *SCL_CHECK* és *SDA_CHECK* jeleket. A 4-61. ábra bal oldalának 2. és 4. soraiban látható folytonos értékadások biztosítják, hogy a modulon belül az adott buszjel H állapotát a szokásos erős 1-ként érzékeljük. Erre a pusztán formális átalakításra azért van szükség, mert ha az adott buszjel H értékét egy belső jel *(signal)* vagy változó *(variable)* erős 1 értékű bitjével vetnénk össze, akkor az összehasonlító operátor ezeket különbözőnek tekintené, ami funkcionálisan hibás, hiszen a H logikailag valójában egyenértékű az erős 1-gyel. Az RTL kapcsolási sémán ezt az átalakítást egy egyszerű buffer szimbólummal érzékeltettük, de ez – a kimenetet meghajtó háromállapotú meghajtóval ellentétben – fizikailag nem igényel többleterőforrást.

Az SPI-interfésszel összevetve az I²C-protokoll – a belezsúfolt többletfunkciók (multi-master jelleg, arbitráció) és a vezetékszám-minimalizálás igénye miatt – lényegesen összetettebb. Ez az összetettség az I²C master interfészáramkör RTL-modelljére is rányomja a bélyegét; ez az IP-mag hasonló szerkezetű, de viselkedését tekintve lényegesen bonyolultabb lesz, mint SPI-os megfelelője. Az I²C master funkciót két rétegre bontva kezeljük; készítünk egy olyan almodult, ami a buszon végrehajtható egybites műveletek megfelelő időzítésű levezénylésére képes *(i2c_bit_driver)*, majd ehhez létrehozunk egy burkolólogikát *(i2c_master)*, ami az egybites buszszolgáltatások felhasználásával képes megvalósítani több-bites adatátvitelt. E több-bites kölcsönhatások megfelelően felparaméterezett sorozatával a felhasználó teljes értékű, a buszra kapcsolt slave eszközök sajátosságainak megfelelő I²C-tranzakciókat hozhat létre.

I²C bit driver ◆ A master egység által végrehajtható egybites kölcsönhatások alatt a start- és stopfeltétel-generálást, valamint egyetlen bit buszra írását vagy onnan való beolvasását értjük. Ez négy különálló szolgáltatásként fogható fel, amelyek mindegyikéhez az *SCL* és *SDA* vezetékek megfelelően koreografált meghajtása és mintavételezése tartozik (lásd 5.2.3. Függelék). E négy szolgáltatás megvalósítására létrehozunk egy one-shot típusú állapotgépet, amit – az egységesség érdekében – szinkron kézfogásos interfésszel látunk el. Ezt az állapotgépet az I²C-buszjelek 4-61. ábrán látható általános használatát, valamint a több mastert tartalmazó buszok arbitrációs célú megfigyelését megvalósító áramkörökkel egészítjük ki. Az almodul RTL-modelljét a 4-62. ábra szemlélteti.

Az almodul szinkron kézfogásos interfészén a négy szolgáltatás közül a kétbites *cmd* bemenet segítségével választhatunk. A *seq_error* kimenet azt hivatott jelezni, ha a felhasználó helytelen sorrendben kezdeményezte a szolgáltatások végrehajtását (pl. startfeltétel generálása nélkül kísérelt meg írni a buszra), az *arb_lost* kimenet pedig akkor jelez, ha az almodul a busz foglaltsága miatt

nem képes végrehajtani az adott szolgáltatást. Ilyenkor az almodul nem várakozik a buszra, a hozzáférést a felhasználónak újra kezdeményeznie kell.

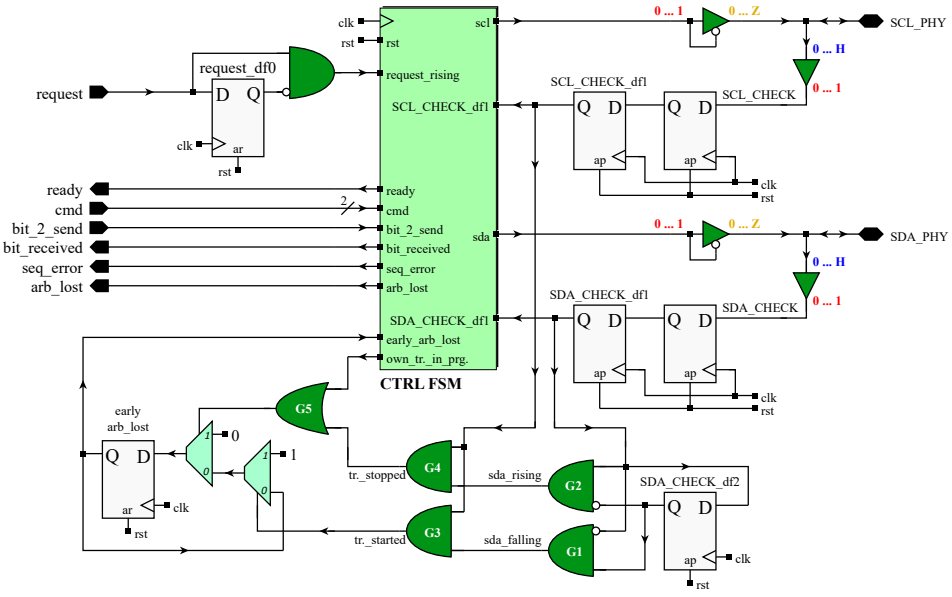
VHDL

```vhdl
L_REQUEST_DETECTOR: process ( clk, rst )
begin
  if ( rst = '1' ) then request_df0 <= '0';
  elsif ( rising_edge(clk) ) then
    request_df0 <= request;
  end if;
end process;
request_rising <= request and
                  not request_df0;

L_CTRL_FSM: process ( clk, rst ) begin
  if ( rst = '1' ) then
    state <= idle; return_state <= idle;
    ready <= '1'; scl <= '1'; sda <= '1';
    halfcycle_counter <= 0;
    bit_received <= '0'; start_sent <= '0';
    own_transaction_in_progress <= '0';
    seq_error <= '0'; arb_lost <= '0';
  elsif ( rising_edge(clk) ) then

    case state is

      when idle => [...]

      [...]

      when others => null;

    end case;

  end if;
end process;
```

VHDL

```vhdl
SCL_PHY <= '0' when scl = '0' else 'Z';
SCL_CHECK <= '0' when SCL_PHY = '0' else '1';
SDA_PHY <= '0' when sda = '0' else 'Z';
SDA_CHECK <= '0' when SDA_PHY = '0' else '1';

L_BUS_MONITORING: process ( clk, rst ) begin
  if ( rst = '1' ) then
    SCL_CHECK_df0 <= '1'; SCL_CHECK_df1 <= '1';
    SDA_CHECK_df0 <= '1'; SDA_CHECK_df1 <= '1';
    SDA_CHECK_df2 <= '1'; early_arb_lost <= '0';
  elsif ( rising_edge(clk) ) then

    SCL_CHECK_df0 <= SCL_CHECK;
    SCL_CHECK_df1 <= SCL_CHECK_df0;
    SDA_CHECK_df0 <= SDA_CHECK;
    SDA_CHECK_df1 <= SDA_CHECK_df0;
    SDA_CHECK_df2 <= SDA_CHECK_df1;

    if ( transaction_stopped = '1' or
         own_transaction_in_progress = '1' ) then
      early_arb_lost <= '0';
    elsif ( transaction_started = '1' ) then
      early_arb_lost <= '1';
    end if;

  end if;
end process;

sda_rising <= SDA_CHECK_df1 and not SDA_CHECK_df2;
sda_falling <= SDA_CHECK_df2 and not SDA_CHECK_df1;
transaction_stopped <= SCL_CHECK_df1 and sda_rising;
transaction_started <= SCL_CHECK_df1 and sda_falling;
```

Áramköri struktúra



4-62. ábra. *Az i2c_bit_driver almodul RTL-modelljének vázlata*

Az *SCL_CHECK* és *SDA_CHECK* jeleket double-flop szinkronizáló fokozatokon keresztül kötjük be az állapotgépbe. Ez elsőre furcsának tűnhet, hiszen az I²C egy szinkron protokoll, aminek az elvi lényege éppen az, hogy az adatvezeték(ek) változásainak várható időpontja a szinkronizáló jelhez képest kötött, tehát ismert. Az adatvezeték(ek) mintavételezésének időpontját pontosan kézben tarthatjuk, módunkban áll gondoskodni arról, hogy csak stabil adatvezeték(ek)et mintavételezzünk, ily módon a double-flop szinkronizáló fokozatok elvileg feleslegesek.[106] Az *i2c_bit_driver* modulba mégis beépítjük őket a következő érvek mentén:

- Az I²C-protokoll szerint az *SDA* buszjelet bármely slave modul csak az *SCL* magas szintje mellett változtathatja meg. Ha az *SCL* felfutása után – a buszon lévő slave modulok egyedi időzítési tulajdonságait figyelembe véve – az *SDA*-t pontosan kiszámított idő elteltével mintavételezzük, amikor az már garantáltan stabil, akkor az *SDA_CHECK* vonalon a double-flop szinkronizáló valóban felesleges. Az *SCL* buszjel esetén azonban az órajelnyújtás során a slave által előidézett felfutóél pontos pillanatát a master elvileg sem ismerheti, ez a jel tehát a lokális órajeltől teljesen függetlennek tekintendő, így a double-flop szinkronizáló fokozat nem mellőzhető.
- Az *SDA* buszjelet fizikailag a buszon lévő slave-ek is képesek meghajtani (legalábbis nullába húzni). Ez azt jelenti, hogy hibás slave-működés esetén előfordulhat, hogy az *SDA* vonal értéke – az I²C időzítési szabályainak ellentmondva – magas szintű *SCL* mellett változik meg, éppen akkor, amikor a master mintát akar venni belőle. Ha az *SDA_CHECK* vonalat double-flop szinkronizáló fokozat nélkül kötjük be az állapotgépbe, akkor a lokális órajelhez képest aszinkron módon, váratlan időpillanatban megváltozó *SDA_CHECK* jel az állapotregiszterben metastabilitást, azon keresztül pedig holtpontot idézhet elő, ami nem egyszerűen logikai, hanem áramköri hiba, ezt pedig feltétlenül el kell kerülnünk.

Az I²C master modult felkészítjük multi-master környezetben való működésre, vagyis beépítjük az arbitrációhoz szükséges áramköri részleteket. A 4-62. ábra jobb oldalának 19–24. és 29–32. sorai olyan logikát írnak le, amely a buszjelek állapotát figyelve képes megállapítani, hogy folyik-e másik master által kezdeményezett tranzakció a buszon. Az *SDA_CHECK_df2* flip-flop, valamint a *G1* és *G2* kapuk által képezett éldetektorok rendre az *SDA* vonal lefutó- és felfutóéleire érzékenyek. Ha magas *SCL* mellett *(G3, G4)* az *SDA* vonalon lefutóélt érzékelünk, akkor az *early_arb_lost*[107] állapotjelző bebillen, és megakadályozza az állapotgépet a buszhoz való hozzáférésben. Az *early_arb_lost* jelzés mindaddig fennáll, ameddig a másik éldetektor magas *SCL* mellett felfutóélt nem detektál az *SDA* vonalon, ami a stopfeltételnek felel meg. Ha az állapotgép sikeresen elindítja egy startfeltétel generálását, akkor az *early_arb_lost* jelzés bebillenését az *own_transaction_in_progress* jel magas szinten tartásával *(G5)* megakadályozza mindaddig, amíg a felhasználó stopfeltétel generálására nem utasítja.

Az egybites szolgáltatásokat végrehajtó almodul magja egy one-shot állapotgép, ami a *request* jel felfutásának detektálásáig tétlen állapotban várakozik (lásd 4-63. ábra).

---

[106] Vö. 4-48. ábra: Az SPI interfészáramköröknél a master *MISO* bemenetén és a slave *MOSI* bemenetén nincs órajel-szinkronizálás, hiszen mindkét jelet az *SCLK* megfelelő élének detektálása pillanatában (a léptetőregiszter *take_input_sample* bemenetén előállított impulzussal) mintavételezzük, amikor az adott jelek már biztosan stabilak.

[107] A „korai" jelző arra utal, hogy a busz használati jogának elveszítésére egy későbbi fázisban is sor kerülhet (lásd 5.2.3. Függelék: *SDA*-arbitráció), ezt az esetet az állapotgép részletes tárgyalásánál vizsgáljuk.

VHDL

```
1  when idle => if ( request_rising = '1' ) then            -- cmd:
2                                                           -- 00: GENERATE START-CONDITION
3               ready <= '0';                               -- 01: SEND BIT
4                                                           -- 10: RECEIVE BIT
5               if ( early_arb_lost = '0' ) then            -- 11: GENERATE STOP-CONDITION
6                 seq_error <= '0'; arb_lost <= '0';
7
8                 if ( cmd = "00" ) then own_transaction_in_progress <= '1'; state <= start_1;
9                 elsif ( cmd = "01" or cmd = "10" ) then state <= bit_tr_1;
10                else state <= stop_1;
11                end if;
12
13              else arb_lost <= '1'; state <= done;
14              end if;
15
16            end if;
17
18 when done => ready <= '1'; state <= idle;
```

4-63. ábra. *Az egybites I²C-szolgáltatásokat végrehajtó állapotgép tétlen állapota*

A *request* jel felfutása után az állapotgép ellenőrzi az *early_arb_lost* jelzést (4-63. ábra: 5. sor). Ha már folyamatban van egy másik master által kezdeményezett tranzakció a buszon, akkor az *arb_lost* kimenetét magas szintre húzva azonnal zárja a szolgáltatás végrehajtására vonatkozó kérést a felhasználó felé (4-63. ábra: 13. sor). Ha a busz a *request* jel felfutásakor szabad, akkor az állapotgép – startfeltétel generálására vonatkozó kérés esetén – az *own_transaction_in_progress* belső állapotjelzőt magas szintre állítja, megakadályozva ezzel az állapotgépen kívüli *early_arb_lost* jel bebillenését (vö. 4-62. ábra: *G5*), majd a *cmd* bemenet értéke alapján megkezdi a kiválasztott szolgáltatás végrehajtását (4-63. ábra: 8–11. sor).

A 4-64. ábra az egybites írás és olvasás műveletéhez tartozó vezérlési állapotokat mutatja be. Az egybites műveletek elvégzése előtt a *start_sent* jel vizsgálatával ellenőrizzük, hogy megtörtént-e már a startfeltétel generálása (4-64. ábra: 2. sor). Ha a legutóbbi stopfeltétel-generálás óta erre nem került sor, akkor a *seq_error* kimeneten jelezzük a hibát a felhasználónak. Írás esetén az *sda* jelre kapcsoljuk a küldendő bitet (4-64. ábra: 6. sor), majd három órajelciklusnyi várakozás után – közvetetten, az *SDA_CHECK_df1* jel olvasásával – ellenőrizzük, hogy az *SDA* vonal valóban a kívánt értékre állt-e be (4-64. ábra: 18. sor). A várakozóciklusokra az *SDA* vonalra beépített double-flop szinkronizáló fokozat miatt van szükség. A 18. sorban elhelyezett feltételvizsgálat valósítja meg az arbitráció második rétegét; ha az *SDA* vonal nem vette fel a kívánt értéket, akkor feltételezzük, hogy egy másik master egység használja éppen a buszt. Ebben az esetben elengedjük a buszjeleket, és értesítjük a felhasználót a busz használati jogának elveszítéséről (4-64. ábra: 22–25. sor). Ha az *SDA* vonalat sikerült a kívánt értékre beállítani (4-64. ábra: 19–20. sor), vagy ha a felhasználó olvasás művelet végrehajtását kérte (4-64. ábra: 9–10. sor), akkor az *SCL* félperiódusának megfelelő időtartamú várakozást iktatunk be. A várakozást az SPI master modulnál látottakhoz hasonlóan egy, az állapotgépbe ágyazott számláló valósítja meg *(halfcycle_counter)*, amelynek maximális értékét a felhasználó egy szintézis-paraméter *(halfcycle_counter_max)* segítségével állíthatja be, meghatározva ez által az *SCL* jel frekvenciáját. Mivel e várakozásra a többi szolgáltatás végrehajtásakor is szükség lesz, ezért magát a számlálást egy különálló vezérlési állapotba szervezzük ki (4-64. ábra: 40–46. sor). Az ilyen „szubrutinszerű", több helyen is hivatkozott vezérlési állapotokba (vagy állapotsorozatokba) való belépés előtt rögzítenünk kell, hogy melyik állapotba kell majd visszatérnünk onnan, ezért bevezetjük a *return_state* másodlagos állapotregisztert (lásd még: 2.3. pont, *BOOTSTRAP* almodul). Ennek típusa a *state* állapotregiszter típusával egyezik meg. Fél *SCL* periódusnyi várakozás után a *bit_tr_3* állapotban az *scl* jel magas szintre húzásával elengedjük az *SCL*

VHDL

```
1  when bit_tr_1 =>
2    if ( start_sent = '0' ) then
3      seq_error <= '1'; state <= done;
4    else
5      if ( cmd = "01" ) then
6        sda <= bit_2_send;
7        state <= wait_sda_1;
8      else
9        return_state <= bit_tr_3;
10       state <= wait_halfcycle;
11     end if;
12   end if;
13
14 when wait_sda_1 => state <= wait_sda_2;
15 when wait_sda_2 => state <= wait_sda_3;
16 when wait_sda_3 => state <= bit_tr_2;
17
```

VHDL (folyt.)

```
18 when bit_tr_2 => if ( SDA_CHECK_df1 = bit_2_send ) then
19                    return_state <= bit_tr_3;
20                    state <= wait_halfcycle;
21                  else
22                    scl <= '1'; sda <= '1';
23                    arb_lost <= '1'; start_sent <= '0';
24                    own_transaction_in_progress <= '0';
25                    state <= done;
26                  end if;
27
28 when bit_tr_3 => scl <= '1'; state <= bit_tr_4;
29
30 when bit_tr_4 => if ( SCL_CHECK_df1 = '1' ) then
31                    if ( cmd = "10" ) then
32                      bit_received <= SDA_CHECK_df1;
33                    end if;
34                    return_state <= bit_tr_5;
35                    state <= wait_halfcycle;
36                  end if;
37
38 when bit_tr_5 => scl <= '0'; sda <= '1'; state <= done;
39
40 when wait_halfcycle =>
41   if ( halfcycle_counter < halfcycle_counter_max ) then
42     halfcycle_counter <= halfcycle_counter + 1;
43   else
44     halfcycle_counter <= 0;
45     state <= return_state;
46   end if;
```

4-64. ábra. *Az egybites írás és olvasás szolgáltatás vezérlési állapotai*

buszjelet. Az *SCL* jel magas szintre billenése az órajelnyújtásra képes slave egységek miatt nem feltétlenül történik meg azonnal, tehát a *bit_tr_4* állapotban ki kell várnunk, amíg valóban előáll a magas szint az *SCL* buszjelen (vö. 5-12. ábra). Amikor ez megtörténik, akkor olvasás művelet esetén az *SDA* vonal aktuális értékét – amit a double-flop szinkronizáló kimenetén látunk *(SDA_CHECK_df1)* – beírjuk a *bit_received* regiszterbe. Függetlenül attól, hogy írást vagy olvasást hajtunk éppen végre, ismét az *SCL* félperiódusának megfelelő időtartamú várakozás következik (4-64. ábra: 34–35. sor), amelynek végén az *SDA* buszjelet elengedjük, az *SCL*-t pedig alacsony szintre húzzuk, így zárva az egybites átvitelt (4-64. ábra: 38. sor).

A 4-65. ábra a start-, a 4-66. pedig a stopfeltétel generálását végző vezérlési állapotok részleteit mutatja be. A startfeltétel generálása előtt ugyancsak ellenőrizni kell, hogy a legutóbbi stopfeltétel generálása után volt-e már startfeltétel-kérés (4-65. ábra: 1. sor). Ez alapján azt kell eldöntenünk, hogy egyszerű startfeltételt vagy megismételt startfeltételt kell-e létrehoznunk a buszjeleken. Egyszerű startfeltétel esetén abból indulunk ki, hogy a legutóbb végrehajtott művelet egy

VHDL

```
1  when start_1 => if ( start_sent = '0' ) then
2                    sda <= '0';
3                    return_state <= start_2;
4                    state <= wait_halfcycle;
5                  else
6                    sda <= '1'; state <= start_3;
7                  end if;
8
9  when start_2 => scl <= '0'; sda <= '1';
10                 start_sent <= '1';
11                 state <= done;
12
```

VHDL (folyt.)

```
13 when start_3 => scl <= '1'; state <= start_4;
14
15 when start_4 => if ( SCL_CHECK_df1 = '1' ) then
16                   return_state <= start_5;
17                   state <= wait_halfcycle;
18                 end if;
19
20 when start_5 => sda <= '0';
21                 return_state <= start_6;
22                 state <= wait_halfcycle;
23
24 when start_6 => scl <= '0'; sda <= '1';
25                 state <= done;
```

4-65. ábra. *A startfeltételt generáló vezérlési állapotok*

stopfeltétel-generálás volt, tehát mindkét buszjel magas állapotban van. Innen kiindulva a startfeltétel létrehozásához előbb lehúzzuk az *SDA* vonalat (4-65. ábra: 2. sor), majd fél *SCL* periódusnyi várakozást beiktatva az *SCL*-t lehúzzuk, miközben az *SDA*-t felengedjük (4-65. ábra: 9. sor, vö. 5-9. ábra: piros kiemelés), a *start_sent* belső jelet pedig magas szintre állítjuk, rögzítve, hogy folyamatban van egy általunk kezdeményezett tranzakció. Megismételt startfeltétel esetén abból indulunk ki, hogy legutóbb egy bit-írás vagy -olvasás történt, az *SCL* buszjel tehát alacsony, az *SDA* pedig magas szintű. Ebből az állapotból megismételt startfeltételt úgy állíthatunk elő, hogy először felengedjük az *SCL* vonalat (4-65. ábra: 13. sor), majd a szokásos félperiódusnyi várakozás után lehúzzuk az *SDA*-t (4-65. ábra: 20. sor), újabb félperiódusnyi várakozás után pedig az *SCL*-t is, miközben az *SDA*-t felengedjük (4-65. ábra: 24. sor). Figyeljük meg, hogy az *scl* jel magasba állítása – vagyis az *SCL* buszjel felengedésére vonatkozó szándékunk kifejezése – után ezúttal is ki kell várnunk, amíg az valóban magasba billen, mert a slave egységek itt is alkalmazhatnak órajelnyújtást (4-65. ábra: 15–18. sor).

VHDL

```
1  when stop_1 => if ( start_sent = '0' ) then
2                   seq_error <= '1'; state <= done;
3                 else
4                   sda <= '0';
5                   state <= wait_halfcycle;
6                   return_state <= stop_2;
7                 end if;
8
9  when stop_2 => scl <= '1';
10                state <= stop_3;
11
```

VHDL (folyt.)

```
12 when stop_3 => if ( SCL_CHECK_df1 = '1' ) then
13                  state <= wait_halfcycle;
14                  return_state <= stop_4;
15                end if;
16
17 when stop_4 => sda <= '1'; start_sent <= '0';
18                own_transaction_in_progress <= '0';
19                state <= wait_halfcycle;
20                return_state <= done;
```

4-66. ábra. *A stopfeltételt generáló vezérlési állapotok*

A stopfeltétel generálása (lásd 4-66. ábra) némileg egyszerűbb, de ezúttal is ellenőriznünk kell, hogy sor került-e korábban legalább egy egyszerű startfeltétel generálására. Ha nem, akkor a stopfeltétel létrehozását célzó kérés felhasználói hibát jelent (4-66. ábra: 1–2. sor). Ha ilyen jellegű hibát nem észlelünk, akkor feltételezhetjük, hogy a legutóbbi művelet egy bit-írás vagy -olvasás volt, az *SCL* buszjel tehát alacsony, az *SDA* pedig magas szintű. A stopfeltétel előidézéséhez előbb le kell húznunk az *SDA* jelet (4-66. ábra: 4. sor), majd félperiódusnyi várakozás után fel kell engednünk az *SCL*-t (4-66. ábra: 9. sor). A potenciális órajelnyújtást természetesen ezúttal is figyelembe kell vennünk (4-66. ábra: 12. sor). Az *SCL* vonal tényleges magasba billenése után félperiódusnyi idő elteltével az *SDA*-t is felengedjük (4-66. ábra: 17. sor), majd a *start_sent* és *own_transaction_in_progress* belső állapotjelzőket alacsony szintre húzva rögzítjük, hogy véget ért egy általunk kezdeményezett tranzakció, a buszt elengedtük.

I²C master burkolólogika ◆ Annak érdekében, hogy az I²C master interfészáramkör használatát felhasználói szempontból egyszerűbbé tegyük, létrehozunk egy burkolómodult *(i2c_master)*, ami az egybites műveletek végrehajtására képes *i2c_bit_driver* modul szolgáltatásait felhasználva több-bites tranzakciókat képes autonóm módon végrehajtani. Ennek a burkolónak a felhasználói interfésze szintén a szokásos szinkron kézfogásos interfész lesz, ami a *request* és *ready* vezérlőjeleken kívül a következő be- és kimeneteket definiálja:

- *cmd*: Kétbites bemenet, az *i2c_bit_driver* almodul megegyező nevű bemenetére kapcsolva.
- *seq_error*, *arb_lost*: Egybites hibajelző kimenetek, az *i2c_bit_driver* almodul megegyező nevű kimenetei által meghajtva.

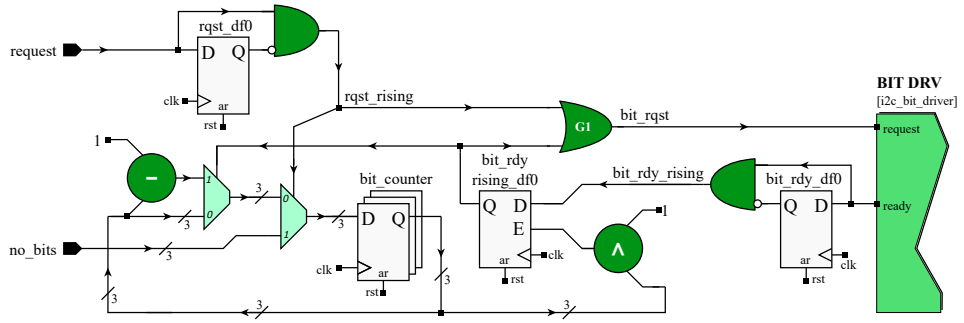
VHDL

```
1  L_GLUE_LOGIC_1: process ( clk, rst ) begin
2  | if ( rst = '1' ) then
3  |   rqst_df0 <= '0'; bit_counter <= 0;
4  | elsif ( rising_edge(clk) ) then
5  |
6  |   rqst_df0 <= request;
7  |
8  |   if ( rqst_rising = '1' ) then
9  |     bit_counter <= no_bits;
10 |   elsif ( bit_rdy_rising_df0 = '1' ) then
11 |     bit_counter <= bit_counter - 1;
12 |   end if;
13 |
14 | end if;
15 end process;
16
17 rqst_rising <= request and not rqst_df0;
18 bit_rqst <= rqst_rising or bit_rdy_rising_df0;
19
```

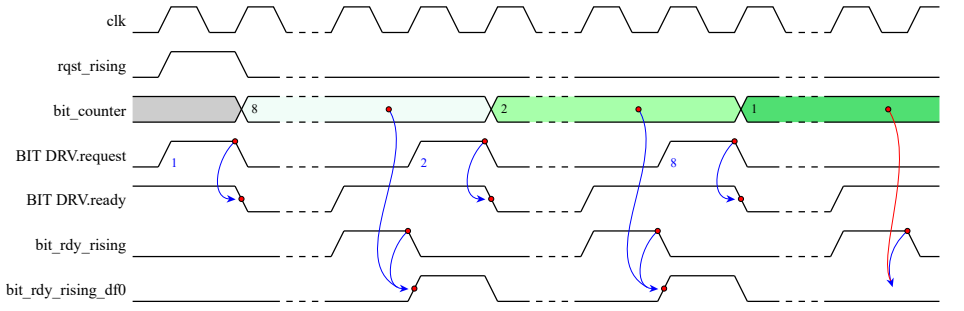
VHDL (folyt.)

```
20 L_GLUE_LOGIC_2: process ( clk, rst ) begin
21 | if ( rst = '1' ) then
22 |   bit_rdy_df0 <= '0'; bit_rdy_rising_df0 <= '0';
23 | elsif ( rising_edge(clk) ) then
24 |
25 |   bit_rdy_df0 <= bit_ready;
26 |
27 |   if ( bit_counter > 1 ) then
28 |     bit_rdy_rising_df0 <= bit_rdy_rising;
29 |   end if;
30 |
31 | end if;
32 end process;
33
34 bit_rdy_rising <= bit_ready and not bit_rdy_df0;
```

Áramköri struktúra



Hullámforma



4-67. ábra. *Burkolómodul az i2c_bit_driver almodulhoz (1)*

- *no_bits*: Írás és olvasás esetén *(cmd = 01_2/10_2)* a buszra írandó vagy onnan olvasandó bitek számát határozza meg.
- *data_2_send*: Nyolcbites bemeneti vektor, amelynek bitjeit a burkolómodul megfelelő sorrendben és ütemezéssel az *i2c_bit_driver* almodul *bit_2_send* bemenetére kapcsolva *no_bits* darab írás műveletet kezdeményez.
- *data_received*: Nyolcbites kimeneti vektor, amelynek bitjeit a burkolómodul az *i2c_bit_driver* almodul *bit_received* kimenetének értékeivel tölti fel az egybites tranzakciók végén.

A burkolólogika alapvető feladata, hogy a szinkron kézfogásos interfészén kapott, egyetlen kérésből *no_bits* számú, megegyező formátumú kérést generáljon az *i2c_bit_driver* almodul számára. A 4-67. ábrán egy ezt a részfeladatot ellátó áramköri sémát láthatunk.

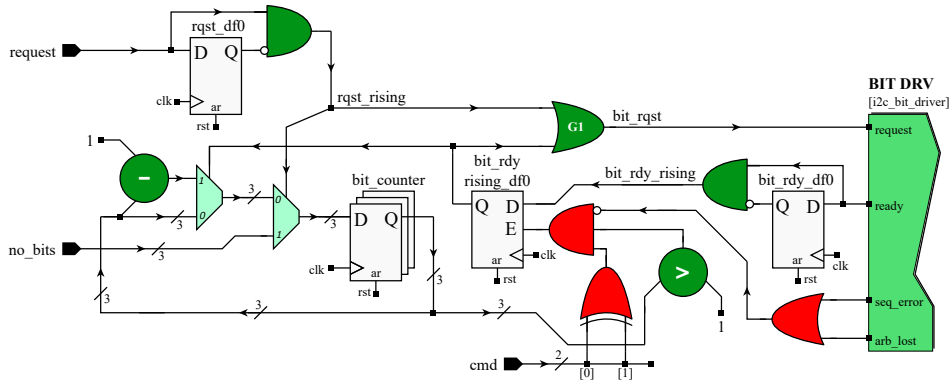
Az autonóm módon generálandó kérések számát a burkolómodul *request* jelének felfutásakor egy számlálóba *(bit_counter)* töltjük, egyúttal a kérést rögtön az *i2c_bit_driver* almodulnak is továbbítjuk a *G1* kapun keresztül. A burkolómodul az *i2c_bit_driver* almodulnak szóló további kéréseket önmaga állítja elő; az almodul *ready* kimenetére helyezett éldetektor minden egybites művelet végrehajtása után egy-egy felfutóélt állít elő. Ezt a felfutóélt a *bit_rdy_rising_df0* regiszteren keresztül csatoljuk vissza az almodul *request* bemenetére. A *bit_rdy_rising_df0* regiszterre azért van szükség, hogy a kellő számú kérés előállítása után valami módon meg tudjuk állítani a folyamatot; a *bit_counter* regiszter értékét minden, automatikusan generált kéréskor eggyel csökkentjük, és amikor ez az érték eléri az egyet, akkor a *bit_rdy_rising_df0* működését letiltjuk, megakadályozva, hogy a *no_bits* bemenet által definiáltnál több kérés keletkezzen.

Az *i2c_bit_driver* almodul *ready* jelének hatására generálódó újabb kéréseket egyéb esetekben is tiltanunk kell. Egyrészt a start- és stopfeltétel előállítására vonatkozó kérések érkezésekor egyáltalán nincs értelme biteket számolni, másrészt a folyamatot a *bit_counter* értékétől függetlenül is le kell tiltanunk, ha az *i2c_bit_driver* almodul hibát jelez a *seq_error* vagy az *arb_lost* kimenetén. Ezeket az eseteket a burkolólogika 4-68. ábrán látható kiegészítésével vehetjük figyelembe.

VHDL

```
20 L_GLUE_LOGIC_2: process ( clk, rst ) begin
21   if ( rst = '1' ) then
22     bit_rdy_df0 <= '0'; bit_rdy_rising_df0 <= '0';
23   elsif ( rising_edge(clk) ) then
24
25     bit_rdy_df0 <= bit_ready;
26
27     if ( cmd(0) /= cmd(1) and bit_counter > 1 and seq_error = '0' and arb_lost = '0' ) then
28       bit_rdy_rising_df0 <= bit_rdy_rising;
29     end if;
30
31   end if;
32 end process;
33
34 bit_rdy_rising <= bit_ready and not bit_rdy_df0;
```

Áramköri struktúra



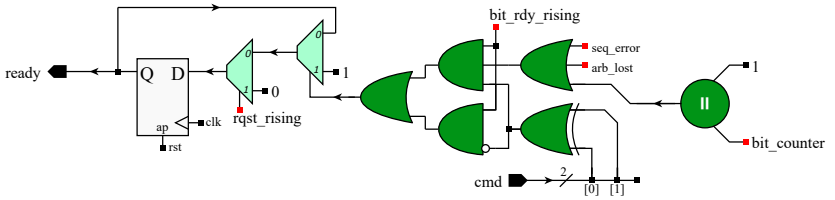
4-68. ábra. *Burkolómodul az i2c_bit_driver almodulhoz (2)*

A burkolómodul eddig bemutatott részletei csak arról gondoskodnak, hogy a megfelelő számú egybites művelet megtörténjen. Biztosítanunk kell azonban azt is, hogy a burkolómodul *ready* jelzése is a szinkron kézfogásos interfész-specifikációnak megfelelően működjön, vagyis külső kérés hatására nullázódjon, a több-bites tranzakció végén pedig – függetlenül annak sikeres vagy sikertelen kimenetelétől – ismét magas szintre álljon be. Az ehhez szükséges áramköri részleteket a 4-69. ábra szemlélteti.

VHDL

```
1  L_GLUE_LOGIC_3: process ( clk, rst ) begin
2  | if ( rst = '1' ) then ready <= '1';
3  | elsif ( rising_edge(clk) ) then
4  | |
5  | | if ( rqst_rising = '1' ) then ready <= '0';
6  | | elsif (
7  | | | (
8  | | |   bit_rdy_rising = '1' and
9  | | |   cmd(0) /= cmd(1) and
10 | | |   (bit_counter = 1 or seq_error = '1' or arb_lost = '1')
11 | | | ) or (
12 | | |   bit_rdy_rising = '1' and cmd(0) = cmd(1)
13 | | | )
14 | | ) then ready <= '1';
15 | | end if;
16 | |
17 | end if;
18 end process;
```

Áramköri struktúra



4-69. ábra. *Burkolómodul az i2c_bit_driver almodulhoz (3)*

A *ready* jelzést – a szokott módon – a *request* jel felfutóéle nullázza (4-69. ábra: 5. sor). A magas szintet az *i2c_bit_driver* almodul *ready* jelzésekor *(bit_ready_rising = 1)* a következő feltételek bármelyikének fennállása esetén vissza kell állítani:

- A felhasználó írás vagy olvasás műveletet kért (*cmd* = 01_2/10_2, 4-69. ábra: 9. sor), az *i2c_bit_driver* almodul pedig befejezte a megfelelő számú egybites írás/olvasás műveletet vagy hibát jelzett (*seq_error = 1* vagy *arb_lost = 1*, 4-69. ábra: 10. sor).
- A felhasználó start- vagy stopfeltétel-generálást kért (*cmd* = 00_2/11_2, 4-69. ábra: 12. sor).

Az egybites írás műveletek generálásához a megfelelő időpillanatokban az *i2c_bit_driver* almodul adatbemenetére *(bit_2_send)* mindig a burkolómodul *data_2_send* bemenetének soron következő bitjét kell kapcsolni, olvasás esetén pedig az ugyancsak egybites adatkimenetből *(bit_received)* egy *no_bits* számú érvényes bitet tartalmazó adathalmazt kell képezni, amit aztán a burkolómodul *data_received* kimenetére juttathatunk. A soros/párhuzamos és párhuzamos/soros átalakítást az SPI interfészáramköröknél látottakhoz hasonlóan egy-egy léptetőregiszterrel oldjuk meg (lásd 4-70. ábra).

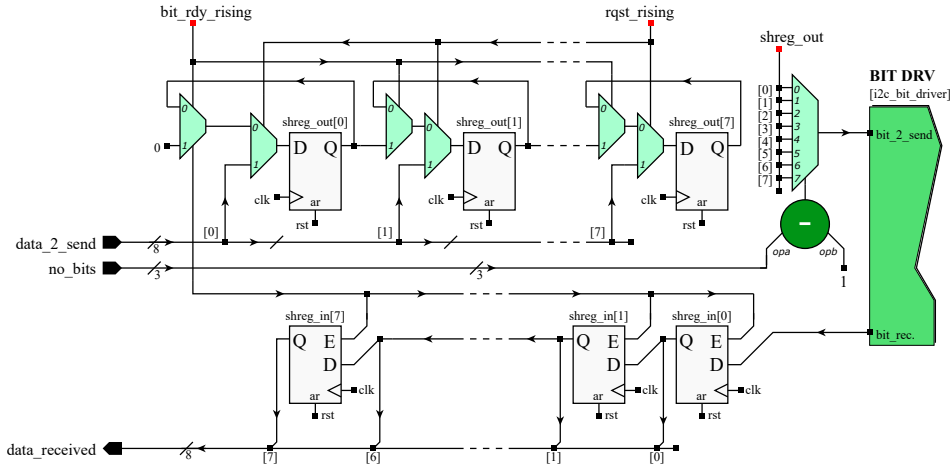
VHDL

```
1  L_GLUE_LOGIC_4: process ( clk, rst ) begin
2    if ( rst = '1' ) then
3       shreg_out <= X"00"; shreg_in <= X"00";
4    elsif ( rising_edge(clk) ) then
5
6      if ( rqst_rising = '1' ) then
7        shreg_out <= data_2_send;
8      elsif ( bit_rdy_rising = '1' ) then
9        shreg_out(0) <= '0';
10       shreg_out(7 downto 1) <= shreg_out(6 downto 0);
11     end if;
12
13     if ( bit_rdy_rising = '1' ) then
14       shreg_in(0) <= bit_received;
15       shreg_in(7 downto 1) <= shreg_in(6 downto 0);
16     end if;
17
18   end if;
19 end process;
20
```

VHDL (folyt.)

```
21 L_BIT_DRV: entity work.i2c_bit_driver(rtl)
22   generic map (halfcycle_counter_max)
23   port map (
24     clk          => clk,
25     rst          => rst,
26     SCL_PHY      => SCL_PHY,
27     SDA_PHY      => SDA_PHY,
28     cmd          => cmd,
29     request      => bit_rqst,
30     ready        => bit_ready,
31     bit_2_send   => shreg_out(no_bits-1),
32     bit_received => bit_received,
33     seq_error    => seq_error,
34     arb_lost     => arb_lost
35   );
```

Áramköri struktúra



4-70. ábra. *Burkolómodul az i2c_bit_driver almodulhoz (4)*

Az *i2c_bit_driver* almodul *bit_2_send* bemenetére kapcsolt léptetőregiszterbe *(shreg_out)* a burkolómodul *request* jelének felfutásakor betöltjük az összes átvivendő bitet (4-70. ábra: 6–7. sor), majd ezt az értéket minden, az *i2c_bit_driver* almodul számára autonóm módon generált kérés pillanatában, vagyis a *bit_rdy_rising* impulzus hatására balra[108] léptetjük (4-70. ábra: 8–10. sor). A *bit_2_send* bemenetet a léptetőregiszter *(no_bits-1)*-edik helyiértékű bitje hajtja, amiből az következik, hogy az átvitel „MSB first” jellegű lesz. Az *i2c_bit_driver* almodul *bit_received* kimenete hajtja meg az olvasáshoz használt léptetőregiszter *(shreg_in)* legalsó helyiértékű bitjét. A léptetést ebben az esetben is a *bit_rdy_rising* jel hatására végezzük el az egybites műveletek végén (4-70. ábra: 13–15. sor). Fontos kiemelni, hogy a burkolómodul *data_received* kimenetén minden tranzakció után csak az alsó *no_bits* darab bit értéke érvényes.

[108] Az ábra sajátosságai miatt látszólag nem balra, hanem jobbra léptetés történik. A léptetési irány kifejezésében azonban az áramköri séma és az áramkör fizikai elhelyezkedése nem játszik szerepet; általánosan elterjedt konvenció szerint a „balra” léptetés a magasabb, a „jobbra” léptetés pedig az alacsonyabb helyiértékű bitek felé való eltolást jelenti.

4.3.3.1. Órajel-szinkronizálás

Az SPI master IP-maghoz hasonlóan az I²C master is a helyi órajel leosztásával állítja elő a kommunikáció ütemezését biztosító *SCL* jel periodikus mintázatát (vö. SPI: *sclk_period_counter*, I²C: *halfcycle_counter*). Ha az I²C master IP-magból szintetizált áramkört az FPGA technológián közepesnek tekinthető 50 MHz-es órajellel hajtjuk meg, akkor 100 kHz-es I²C-interfész esetén az *i2c_bit_driver* almodul állapotgépe a *wait_halfcycle* állapotban 500 órajelciklust várakozik. E várakozás során fizikailag a számláló növelésén túl semmi nem történik, ugyanakkor az áramkör órajelhálózatán 10 ns-onként bekövetkezik egy fel- vagy lefutóél, ami sok energiát igényel. A fogyasztás optimalizálása érdekében tehát az I²C master IP-mag saját órajelének frekvenciáját érdemes lecsökkenteni. Az interfészáramkör beágyazó környezetének órajelére természetesen más optimalizációs szempontok vonatkozhatnak, amelyekről az újrafelhasználható IP tervezőjeként nem tudhatunk. Feladatunk csupán az, hogy biztosítsuk az I²C master IP-maghoz a megfelelő órajel-szinkronizáló fokozatokat tartalmazó burkolómodult, ami lehetővé teszi a toplevel-integrátor számára a magórajelnek és a felhasználói interfész órajelének szétválasztását. Az I²C master IP-mag felhasználói interfészének működése megfelel az 1.2.1.1. pontban bemutatott szinkron kézfogásos interfész-specifikációnak, vagyis e CDC-burkolómodulban használhatjuk a 2-2. ábra órajel-szinkronizáló fokozatát. Az általánosított $A_{i,S}$ és $B_{k,S}$ vektorok a 4-5. táblázat szerint feleltethetők meg a modul interfészjeleinek.

Általános CDC-logika	I²C master modul
$A_{i,S}$	*cmd, data_2_send, no_bits*
$B_{k,S}$	*data_received, seq_error, arb_lost*

4-5. táblázat. *Az I²C master modul és az általánosított CDC-logika jeleinek megfeleltetése*

Pár gondolat a lokális órajelforrás periódusidejéről…

Mindhárom soros kommunikációsinterfész-protokoll áramköreiben éltünk azzal a feltételezéssel, hogy a lokális órajel periódusideje lényegesen kisebb, mint az adott kommunikációs interfészre jellemző, egy bit átviteléhez szükséges idő. Láthattuk, hogy ez a feltételezés nagy rugalmasságot ad, megkönnyíti az áramkörtervezést. Ugyanakkor órajel-szinkronizáló burkolómodulokat is készítettünk a célból, hogy a toplevel-integráció során lehetőség legyen a soros-párhuzamos és párhuzamos-soros átalakítókat – úgy is fogalmazhatunk, hogy a bitszinkronizálásért felelős áramköri részleteket – a saját feladatukhoz illő mértékben lecsökkentett órajelforrásról járatni fogyasztásuk és méretük optimalizálása céljából. Érdemes tehát megvizsgálni, hogy a lokális órajelek frekvenciájának csökkentése milyen hatással van ezeknek az IP-magoknak a működésére.

A „proaktív” oldal ◆ *Az UART-adó-, az SPI master és az I²C master modulokban a lokális órajellel egy-egy időzítőszámlálót hajtottunk meg, amivel a vezérlő állapotgépekbe tulajdonképpen egy-egy órajel-előosztó funkciót építettünk be (vö. 4.1.1. pont). A leosztás mértékét – ami UART esetén a bitidőt, SPI esetén az SCLK, I²C-nél pedig az SCL periódusok időtartamát határozza meg – az IP-magok felhasználója egy-egy szintézis-paraméterrel állíthatja be.*
Az UART-adónál a helyi órajel leosztásának legkisebb, még megfelelő működést biztosító értéke 0, ami azt jelenti, hogy a bitidő az adó órajelének periódusidejével egyenlő. Ezt kétféleképpen értelmezhetjük:

- *Az UART-adó órajelének periódusidejét akár egészen a kívánt bitidőig növelhetjük, az adó által generált mintázatok a TX vezetéken így is a protokollnak megfelelők lesznek.*
- *Az FPGA-kban jellemző, több tíz MHz frekvenciájú órajellel hajtva az adót több tíz Mb/s-os adási sebességet képviselő mintázatokat állíthatunk elő a TX vezetéken. Ez az alkalmazások túlnyomó részében több mint elegendő, különös tekintettel arra, hogy ilyen sebességek esetén a vevő oldalon a soros-párhuzamos átalakító hasonló technológián történő megvalósítása egyáltalán nem magától értetődő (lásd 4.3.4. pont).*

Az SPI master modul esetén a helyi órajel leosztásának legkisebb értelmes értéke 1, ami azt jelenti, hogy az SCLK kimenet frekvenciája a lokális órajel frekvenciájának fele lesz. A lokális órajel frekvenciája tehát egészen az adott alkalmazásra jellemző SCLK frekvencia duplájáig csökkenthető. De milyenek is ezek az alkalmazások? Ha egy SPI-interfésszel ellátott katalógusáramkört (pl. egy egyszerű szenzort) szeretnénk illeszteni a master modul felhasználásával, akkor az FPGA-kban tipikusnak tekintett, több tíz vagy akár 100 MHz frekvenciájú órajelet bőven lecsökkenthetjük, mivel az ilyen periféria-áramkörök tipikusan legfeljebb néhány MHz-es SCLK frekvenciát tudnak kezelni. Nagyobb sebességű A/D- és D/A-átalakítók esetén, amelyek akár 20-50 MHz-es SCLK-t is tudnak fogadni, az itt bemutatott master modul használható, bár ebben az esetben az SPI-interfész fizikai kialakítása is kritikus és az időzítések részletes vizsgálata (lásd DR-MSz: 6. fejezet: statikus időzítésvizsgálat) elkerülhetetlen. Ugyanez igaz akkor is, ha az SPI master egységet gépek közötti kommunikációban használjuk úgy, hogy a slave funkciót egy másik eszköz tartalmazza. Léteznek nagysebességű SPI-interfésszel ellátott katalógusáramkörök is (tipikusan memóriák), amelyek akár 100-200 MHz frekvenciájú SCLK jelet is igényelhetnek. Az SPI master modul lokális órajelének frekvenciáját tehát ezekhez az értékekhez képest minimum duplázni kell, e frekvencianövelésnek azonban határt szab maga az FPGA a benne található flip-flopok, LUT-ok és vezetékhálózatok saját késleltetései miatt.

Az I^2C master IP-mag esetén a helyzet némileg bonyolultabb; az áramkör belső sajátosságai okán és amiatt, hogy az I^2C-tranzakciókban az egy bit átviteléhez szükséges időt a megszólított slave egység is erősen befolyásolja, a belső időzítőszámláló értéke és az SCL jel periódusideje között csak közvetett kapcsolat van. Logikai szimulációval ellenőrizhető, hogy a halfcycle_counter_max paraméter 0 értékénél az IP-mag – ahol ez csak rajta múlik – öt órajelciklusnyi SCL félperiódusokat generál, vagyis elmondható, hogy a master IP órajelének periódusideje nagyjából a kívánt SCL periódusidő tízszereséig növelhető.

A „reaktív" oldal ◆ *Az UART-vevő és az SPI slave modul órajel-frekvenciájának csökkentése több problémát vet fel, ezeknek az áramköröknek a tervezésénél ui. sokkal inkább támaszkodtunk a viszonylag gyors helyi órajelre. Az UART-vevőben az RX, az SPI slave-ben pedig a CSn és az SCLK jeleket statikus jelnek tekintettük, vagyis azt feltételeztük, hogy változásaik a lokális órajel periódusidejénél sokkal ritkábban követik egymást. Úgy is fogalmazhatunk, hogy „túlmintavételezést" alkalmaztunk. A fel- és lefutóélek detektálása így óhatatlanul pontatlan (ezt a 4-55. és 4-57. ábrák érzékeltetik is), mivel a változásoknak keresztül kell haladniuk a double-flop + éldetektor struktúrákon ahhoz, hogy a belső áramkörök számára is feldolgozható „egyszeri eseményekké", a lokális órajelhez szinkronizált impulzusokká formálódjanak. A fel- és lefutóélek és azok érzékelése között tehát idő telik el, és ez az idő annál nagyobb, minél nagyobb a lokális órajel periódusideje.*

Az UART-vevőben ez azt eredményezi, hogy az RX vezetéken a start-bit kezdetét jelző lefutóélt később érzékeljük, mint kellene, ezért az összes bit mintavételi időpillanata kissé későbbre tolódik, ráadásul az érvénytelen negatív impulzusok kiszűrését végző léptetőregiszter (4-40. ábra) ezt az eltolódást tovább növeli. A bitidő és a lokális periódusidő arányának csökkenésével az eltolódás mértéke egyre jelentősebb. Az eltolódást könnyen kompenzálhatjuk úgy, hogy a mintákat az időzítőszámlálónak nem pontosan az egyharmadánál, felénél és kétharmadánál vesszük, hanem kicsivel korábban (4-41. ábra: 3–5. sor). A mintavételi idő eltolódása tehát jól kezelhető, logikai szimulációval könnyen ellenőrizhető, de a háromszori mintavétel értelemszerűen határt szab a bitidő csökkentésének; mindhárom mintavételnek garantáltan az adott bit időablakán belülre kell esnie. Mindezek alapján kijelenthető, hogy az UART-vevőnek ez a változata bizonyosan megbízhatóan működik, ha a bitidő legalább ötvenszerese az órajel periódusidejének.

Az SPI slave modulban a CSn jelet fogadó double-flop + éldetektor fokozat okozta késleltetés kevésbé problémás, de az SCLK esetén ugyanez a struktúra azt okozza, hogy a MOSI jel mintavételezése is és a MISO kimenet frissítése is később következik be a kelleténél. A SPI protokoll „játékszabályai" szerint mindaddig nincs gond, amíg a mintavételezés és a kimenet frissítése is megtörténik az SCLK következő éle előtt. Az SCLK periódusidejét csökkentve a slave éldetektorainak késleltetése egyre jelentősebbé válik. Ha az SCLK jel egy változásának slave egységbeli detektálása megközelíti az SCLK jel következő változását, akkor az interfész csak igen alapos, a fizikai kialakítás pontos vizsgálatát is magában foglaló időzítés-analízissel kezelhető. Ökölszabályként megfogalmazható, hogy az itt bemutatott, egyszerű SPI slave modul akkor alkalmazható e részletes időzítésvizsgálat mellőzésével, ha órajelének frekvenciája legalább tíz-húszszorosa az SCLK jel frekvenciájának.

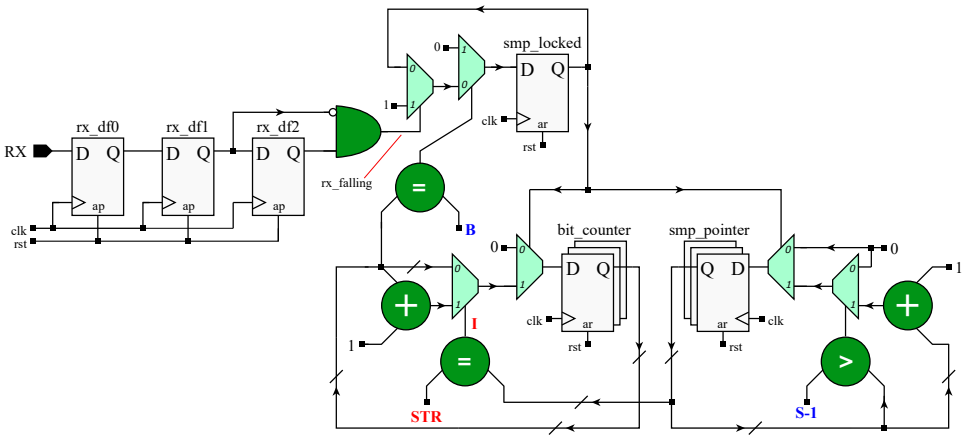
VHDL

```
1  L_START_DETECT: process ( clk, rst ) begin
2    if ( rst = '1' ) then
3      rx_df0 <= '1'; rx_df1 <= '1';
4      rx_df2 <= '1'; smp_locked <= '0';
5    elsif ( rising_edge(clk) ) then
6
7      rx_df0 <= RX;
8      rx_df1 <= rx_df0;
9      rx_df2 <= rx_df1;
10
11     if ( bit_counter = B ) then
12       smp_locked <= '0';
13     elsif ( rx_falling = '1' ) then
14       smp_locked <= '1';
15     end if;
16
17   end if;
18 end process;
19
20 rx_falling <= rx_df2 and not rx_df1;
21
```

VHDL (folyt.)

```
22 L_COUNTERS: process ( clk, rst ) begin
23   if ( rst = '1' ) then
24     bit_counter <= 0; smp_pointer <= 0;
25   elsif ( rising_edge(clk) ) then
26
27     if ( smp_locked = '1' ) then
28       if ( smp_counter < S-1 ) then
29         smp_counter <= smp_counter + 1;
30       else smp_counter <= 0;
31       end if;
32     else smp_counter <= 0;
33     end if;
34
35     if ( smp_locked = '0' ) then
36       bit_counter <= 0;
37     elsif ( smp_counter = STR ) then
38       bit_counter <= bit_counter + 1;
39     end if;
40
41   end if;
42 end process;
```

Áramköri struktúra

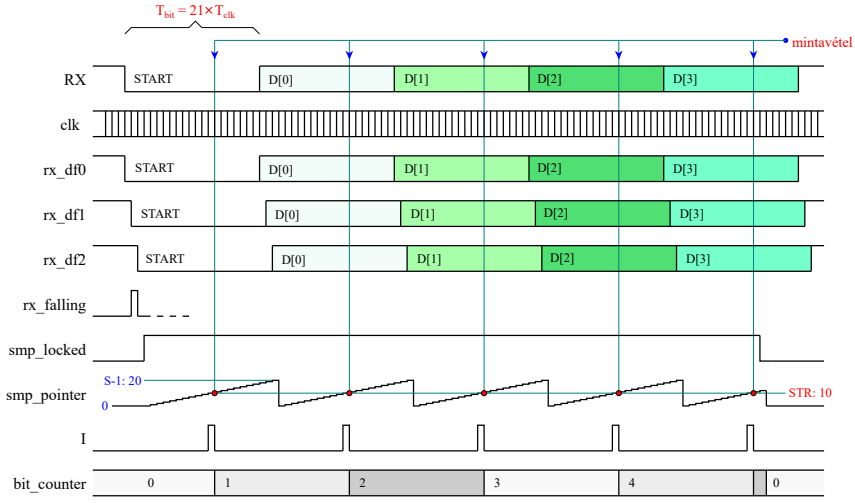


4-71. ábra. *UART bitfolyam vételének időzítésére alkalmas áramkör RTL-modellje*

4.3.4. Nagysebességű UART-vevő

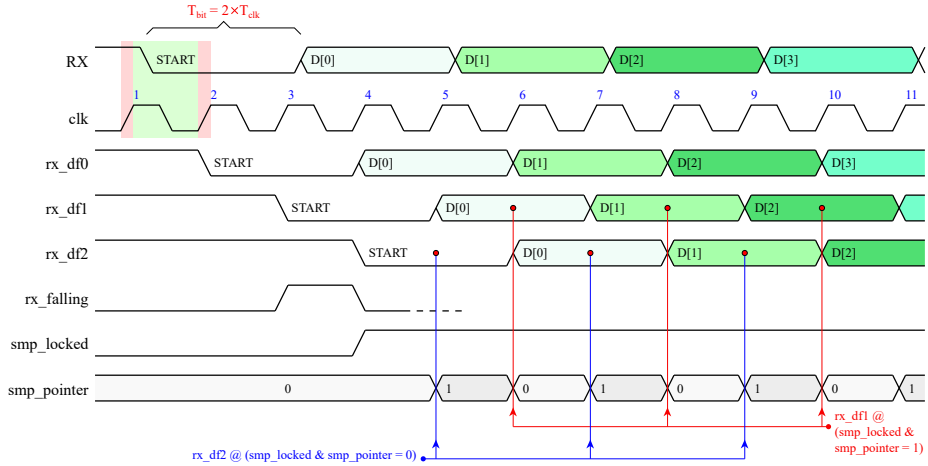
A 4.3.1. pontban bemutatott UART adó- és vevőáramkörök tervezésekor kihasználtuk azt a peremfeltételt, hogy egyetlen bit átvitelének időtartama legalább ötvenszerese a lokális órajelforrás periódusidejének. Az alábbiakban részletezett nagysebességű UART-vevő esetén azt vizsgáljuk, hogy adott órajel-frekvencia esetén mennyire csökkenthető le a bitidő úgy, hogy az aszinkron kommunikáció még megbízhatóan működjön, illetve, hogy milyen áramköri megoldásokat kíván a bitidő és az órajel-periódus arányának elvi minimumra csökkentése.

Induljunk ki a 4-71. ábrán látható, UART bitfolyamok vételének időzítésére alkalmas áramkörből. Az adó által az áramkör *RX* bemenetén előállított, az UART adatkeretnek megfelelő bitfolyam az alacsony szintű start-bittel kezdődik. Az *RX* vonal lefutóélét detektálva az *smp_locked* állapotjelző magas szintre billen, aminek hatására az *smp_pointer* számláló elkezd órajelciklusonként inkrementálódni. Az *S* paraméter által meghatározott maximális érték elérésekor a számlálás újrakezdődik. Az *smp_pointer* számláló tehát tulajdonképpen egy olyan állapotváltozónak tekinthető, amely a start-bit kezdetekor elindulva azonos időközöket definiál. A számláló adott értéke mindig $S \times T_{clk}$ idő elteltével áll elő újra és újra, vagyis egy konkrét értéket (az ábrán *STR*) dekódolva előállíthatjuk akár az *RX* bemenetet, akár az *rx_df0 ... rx_df2* flip-flopok mintavételezésének pillanatát jelző impulzust *(I)*, ami egyúttal a beolvasott bitek számát tároló *bit_counter* regiszter értékét is megnöveli. Amikor ez utóbbi regiszter értéke eléri a *B* paraméter értékét, akkor az *smp_locked* állapotjelzőt nullázzuk, ezzel az áramkör ismét az *RX* bemenet start-bitet jelző lefutóélére várakozó állapotba kerül. A 4-72. ábra hullámformája ezt a folyamatot szemlélteti egy olyan UART-kapcsolat esetén, ahol a bitidő a vevő órajel-periódusának huszonegyszerese, az adatkeret négy adatbitet tartalmaz, a mintavevő impulzust pedig a bitidő felénél állítjuk elő *(S = 21, B = 5, STR = 10)*.



4-72. ábra. *Az UART-vevő időzítőáramkörének működése, ha a bitidő az órajel periódusidejének sokszorosa*

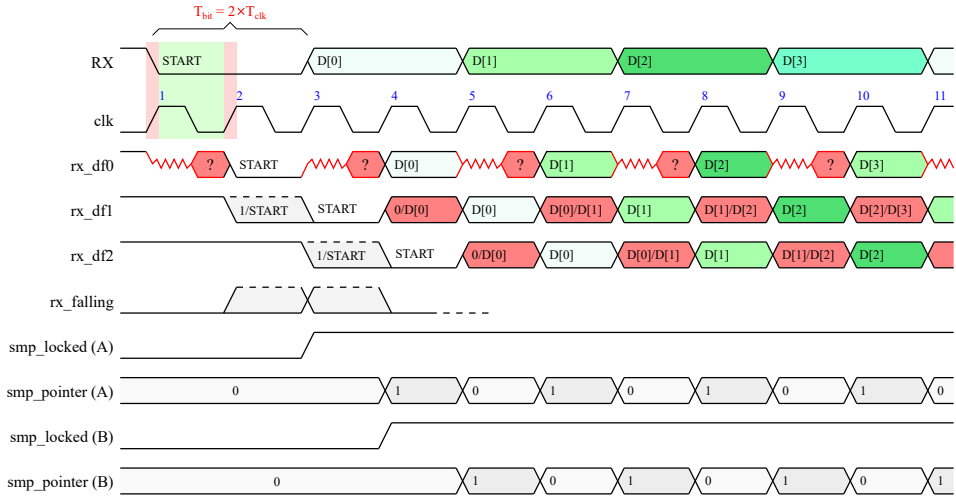
Jól látható a mintavétel pillanatának a korábban is említett, az *RX* bemenet double-flop szinkronizáló fokozatának és éldetektorának késleltetése okozta elcsúszása, de ennek az ábrán látható *S = 21* esetben nincs jelentősége. Huszonegy – vagy a korábban rögzített feltétel szerinti ötven – órajelciklus alatt rengeteg mozgásterünk van a mintavétel megoldására, ahogy azt láthattuk is a többszöri mintavétel esetén. Vizsgáljuk meg, hogy miként viselkedik ugyanez az időzítő áramkör, ha a bitidő csupán duplája a lokális órajel periódusidejének. Ezt az esetet a 4-73. ábra hullámformája mutatja be.



4-73. ábra. *Az UART-vevő időzítőáramkörének működése, ha a bitidő az órajel periódusidejének duplája (1)*

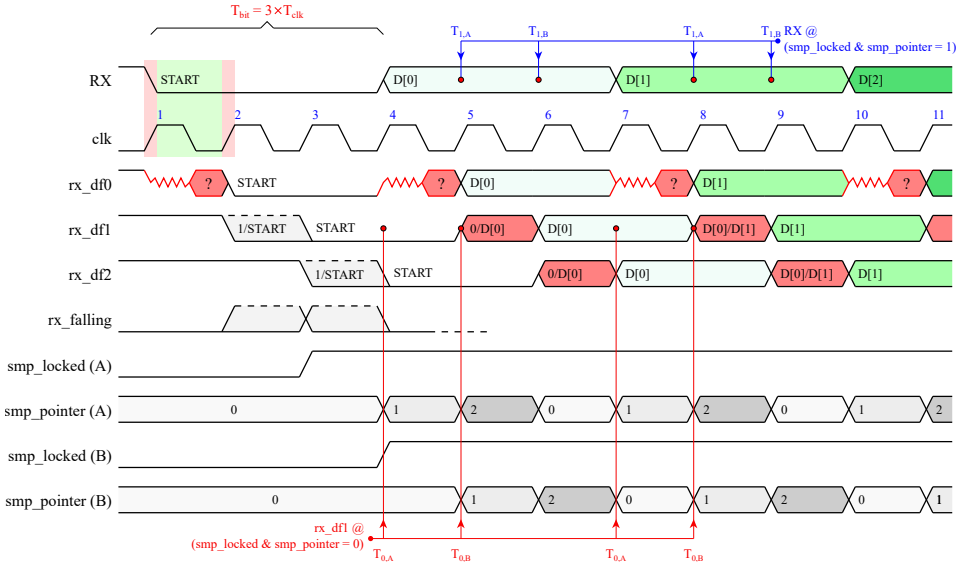
S = 2 esetén az *smp_pointer* számláló értéke nulla és egy között váltakozik, de ez látszólag nem jelent gondot, a hullámforma alapján arra következtethetünk, hogy a mintavétel pillanata többféleképpen is meghatározható; a start-bit kezdetének detektálása, tehát az *smp_locked* állapotjelző magas szintre billenése után megtehetjük, hogy az *smp_pointer* 0 értékeinél mintavételezzük az *rx_df2* vagy 1 értékeinél az *rx_df1* flip-flop kimenetét.[109] Látszólag mindkét megoldás működik, csak azt az apró különbséget kell figyelembe vennünk, hogy utóbbi esetben a start-bitről „lemaradtunk", de ez tulajdonképpen nem is baj; a start-bit ezen a ponton már betöltötte szerepét azzal, hogy a mintavételt elindította, értékét tárolni felesleges, csak az adatbitek fontosak számunkra. Mindezek alapján arra a következtetésre juthatnánk, hogy akár a $2 \times T_{clk}$ bitidővel jellemezhető UART-kapcsolat is gond nélkül megvalósítható. Ezzel a koncepcióval azonban van egy kis probléma; az *RX* bemenet double-flop szinkronizálójának késleltetését ugyan figyelembe vettük, de azt nem, hogy ennek a késleltetésnek a mértéke valójában nem determinisztikus. Ha az *RX* bemenet – aszinkron természetéből adódóan nem felróható módon – az *rx_df0* flip-flop beállítási vagy tartási idejére vonatkozó időzítési feltételeket megsértve változik meg, akkor az metastabil állapotba kerülhet, amelyből kilépve kimenete egyforma valószínűséggel lehet alacsony vagy magas, a metastabilitást előidéző él iránya erre nincs befolyással.

[109] Az „*smp_pointer* X értékeinél" való mintavételezés úgy értendő, hogy az órajelnek az *smp_pointer* X értékre való beállását követő felfutóélénél.



4-74. ábra. *Az UART-vevő időzítőáramkörének működése, ha a bitidő az órajel periódusidejének duplája (2)*

A jelenséget a 4-74. ábrán szemléltetett módon vizsgálhatjuk; az *RX* minden változásánál feltételezzük, hogy az metastabilitást idéz elő az *rx_df0* flip-flopban, amelynek kimenete így ismeretlen értéken stabilizálódik, még az órajel következő felfutóéle előtt. Az *RX* vonal új értéke tehát teljes bizonyossággal csak az órajel következő felfutóéle hatására töltődik be az *rx_df0* flip-flopba. Ez a bizonytalanság az áramkör további részeiben (közvetlenül az *rx_df1* és *rx_df2* flip-flopokban) úgy



4-75. ábra. *Az UART-vevő időzítőáramkörének működése, ha a bitidő az órajel periódusidejének háromszorosa*

nyilvánul meg, hogy azok értéke adott pillanatban nem feltétlenül ismerhető. A metastabil állapot kialakulásának veszélyét az *rx_df0* flip-flop ugyan elhárította, de az *rx_df1* és *rx_df2* flip-flopokba az *RX* jel mintáinak értékei egy órajelciklusnyi késleltetésbizonytalansággal kerülnek be. A 4-74. ábra hullámformája ezt úgy szemlélteti, hogy a piros hátterű időablakokban a flip-flop mindkét lehetséges értékét beírtuk – pl. az órajel negyedik felfutóéle hatására az *rx_df1* flip-flopba vagy a start-bit nullája, vagy már a *D[0]* adatbit értéke íródik be. Az *RX* bemenet változásait tehát egy órajelciklusnyi bizonytalansággal tudjuk csak detektálni (lásd *rx_falling* jelalakja), így a start-bit tényleges kezdete és az *smp_locked* állapotjelző magas szintre billenése között eltelt idő is két szélsőséges eset között változhat. E két szélsőséges esetet a 4-74. ábrán az *smp_locked* és *smp_pointer* regiszterek (A)-val és (B)-vel jelölt verziói szemléltetik. A kérdés már csak az, hogy ki tudunk-e választani az *RX*, *rx_df0*, *rx_df1* és *rx_df2* jelek közül egyet, amely az *smp_pointer* egy konkrét értékénél mindig helyes, gond nélkül mintavételezhető stabil értéket vesz fel, függetlenül attól, hogy az (A) és (B) forgatókönyvek közül melyik valósul meg.

- ***RX @ (smp_locked & smp_pointer = 0)***: Az (A) esetben nincs probléma, a (B) esetben viszont egy éppen megváltozó jelet mintavételeznénk, ami a mintát tároló flip-flop metastabil állapotba kerülését okozhatja.[110]
- ***rx_df0 @ (smp_locked & smp_pointer = 0/1)***: Két eset lehetséges; vagy egy stabil, de ismeretlen értéket (pl. az órajel negyedik felfutóélénél) vagy egy éppen potenciálisan metastabillá váló kimenetet mintavételeznénk.[111]
- ***rx_df1 @ (smp_locked & smp_pointer = 0)***: Az (A) forgatókönyv esetén a mintavétel helyes lenne, de a (B) esetben ismeretlen értéket mintavételeznénk.

[110] A hullámforma itt sajnos kicsit félrevezető; sok helyen találkozhatunk azzal, hogy egy regiszterbe látszólag éppen megváltozó értéket töltünk be. Ez csak akkor nem okoz gondot, ha a kibocsátó flip-flop kimenetének változását ugyanaz az órajel-felfutóél váltja ki, mint amelyik a fogadó flip-flopban a mintavételezést okozza, vagyis, ha a két flip-flop egyazon lokális adatúthoz tartozik. Ekkor a logikai szimulációban nullának feltételezett órajelelcsúszás (clock skew), a flip-flopok szintén nullának feltételezett tartási ideje (hold-time), illetve az a további feltevés, hogy a kibocsátó flip-flop kimeneti késleltetésének (clock-to-output delay) és a két flip-flop közötti jelterjedési időnek (propagation delay) az összege a tartási időnél (ez esetben nullánál) nagyobb, együtt azt eredményezik, hogy a mintavétel garantáltan sikeres lesz (lásd még DRMSz: 6. fejezet). Az absztrakt (időzítetlen) HDL-modellezésben mindezeket a feltevéseket a VHDL a delta-késleltetés, a Verilog pedig a nem-blokkoló értékadás fogalmába sűríti bele (lásd még DRMSz: 4.4.4. alfejezet). Az RTL-modelljeinkből szintetizált áramkörök jórészt automatizált fizikai tervezése során az azt végző algoritmusoknak egyrészt van információja a valós flip-flopok tulajdonságairól (kimeneti késleltetés, tartási idő stb.), másrészt van befolyásuk az órajelelcsúszás és a flip-flopok közötti vezetékezés késleltetésére különféle órajelmeghajtó-bufferek használatán és a térbeli elhelyezésen keresztül. Így biztosítható – vagy legalábbis ellenőrizhető – az RTL elvonatkoztatási szinten alkalmazott időzítési feltételezések teljesülése a fizikai valóságban is. Elérhető, hogy az órajelelcsúszás mértéke minden egyéb késleltetéstípusnál kisebb – tehát látszólag nulla – legyen, illetve, hogy a kibocsátó flip-flop kimeneti késleltetésének és a két flip-flop közötti vezetékezés késleltetésének az összege nagyobb legyen, mint a fogadó flip-flop tartási ideje, így a mintavétel sikeressége biztosított. A fenti példában azonban az *RX* jel és az azt mintavételező flip-flop nem egyazon lokális adatút két vége, az *RX* jel egy külső, aszinkron jel, amelynek a változásai ugyan egybeeshetnek a lokális órajel éleivel, de azokat nem e lokális órajel élei váltják ki.

[111] Az éppen a mintavétel pillanatában potenciálisan metastabillá váló *rx_df0* flip-flop mintavételezése önmagában még nem feltétlenül lenne gond; az *rx_df0* és a mintát tároló flip-flop egyazon lokális adatút részei, így a mintavétel a logikai szimulációnak a korábban említett, időzítési viszonyokra vonatkozó feltételezései alapján elvileg sikeres lehetne. Egy double-flop szinkronizáló fokozat első flip-flopjának *(rx_df0)* kimenetét azonban nem célszerű a szinkronizálóhoz tartozó második flip-flopon *(rx_df1)* kívül egyéb részáramkörökkel (pl. egy mintatároló regiszterrel és az ahhoz vezető vezetékezéssel) terhelni, mert a többletterhelés megnöveli a metastabil állapot időtartamát (lásd még DRMSz: 6.3.1. alfejezet), ami közvetetten a szinkronizáló fokozat megbízhatóságát csökkenti.

- ***rx_df1 @ (smp_locked & smp_pointer = 1)***: A (B) forgatókönyv esetén a mintavétel helyes lenne, de az (A) esetben ismeretlen értéket mintavételeznénk.
- ***rx_df2 @ (smp_locked & smp_pointer = 0)***: A (B) forgatókönyv esetén a mintavétel helyes lenne, de az (A) esetben ismeretlen értéket mintavételeznénk.
- ***rx_df2 @ (smp_locked & smp_pointer = 1)***: Az (A) forgatókönyv esetén a mintavétel helyes lenne, de a (B) esetben ismeretlen értéket mintavételeznénk.

Ezek alapján belátható, hogy a 4-71. ábra áramköre nem alkalmas arra, hogy az órajel-periódus duplájának megfelelő bitidejű UART kommunikációt valósítsunk meg vele. Vizsgáljuk hát meg azt az esetet, amikor a bitidő az órajel-periódus háromszorosa (lásd 4-75. ábra).

A double-flop szinkronizáló fokozat sajátosságait ezúttal is figyelembe vesszük; a potenciálisan metastabil állapotba kerülő *rx_df0* flip-flop miatt az *rx_df1* és *rx_df2* flip-flopokba beíródó, ismeretlen értékeket feltüntettük. Az *smp_pointer* regiszter a start-bit kezdetét jelző lefutóél detektálása után ezúttal a 0 → 1 → 2 ciklust ismétli. A double-flop szinkronizáló fokozat okozta, egy órajel-ciklusnyi bizonytalanság miatt az *smp_locked* állapotjelzőnek és az *smp_pointer* számlálónak itt is

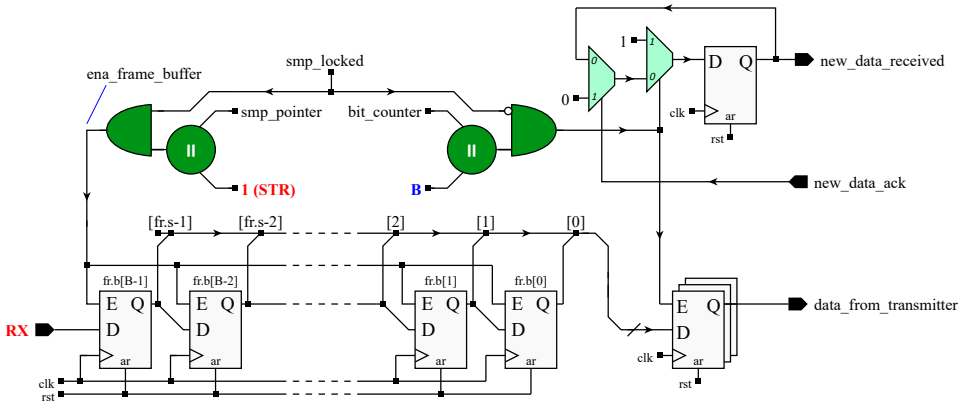
VHDL

```
1  ena_frame_buffer <= '1' when smp_locked = '1' and
2                              smp_pointer = 1 else
3                      '0';
4
5  L_FRAME_BUFFER: process ( clk, rst ) begin
6    if ( rst = '1' ) then
7      frame_buffer <= (others => '0');
8    elsif ( rising_edge(clk) ) then
9      |
10     if ( ena_frame_buffer = '1' ) then
11     | frame_buffer(B-1) <= RX;
12     | frame_buffer(B-2 downto 0) <=
13     |   frame_buffer(B-1 downto 1);
14     end if;
15     |
16   end if;
17 end process;
18
```

VHDL (folyt.)

```
19 L_USER_IF: process ( clk, rst ) begin
20   if ( rst = '1' ) then
21     new_data_received <= '0';
22     data_from_transmitter <= (others => '0');
23   elsif ( rising_edge(clk) ) then
24     |
25     if (
26       bit_counter = B and
27       smp_locked = '0'
28     ) then
29       new_data_received <= '1';
30       data_from_transmitter <= frame_buffer;
31   elsif ( new_data_ack = '1' ) then
32       new_data_received <= '0';
33   end if;
34     |
35   end if;
36 end process;
```

Áramköri struktúra



4-76. ábra. *Az UART-vevő mintatároló léptetőregisztere és felhasználói interfésze*

(A) és (B) verziójáról beszélhetünk, ezek közül a valóságban bármelyik megvalósulhat az egymást követő adatkeretek vétele során. A kérdés ezúttal is az, hogy az *RX* bemenet, illetve az *rx_df0*, *rx_df1* és *rx_df2* flip-flopok között van-e olyan, amelynek értéke az *smp_pointer* egy adott értékénél az (A) és (B) forgatókönyv esetén is helyesen mintavételezhető. A hullámformán azt látjuk, hogy akár több megoldás közül is választhatunk; vehetünk mintát közvetlenül az *RX* bemenetből az *smp_pointer* 1 értékénél vagy az *rx_df1* flip-flop kimenetéből az *smp_pointer* 0 értékénél.[112] A mintavétel megfelelő pillanatának kijelölése tehát az *smp_pointer* dekódolásával megoldható.

A 4-71. ábrán látható áramköri részlet csupán a mintavétel időzítéséhez szükséges funkciókat látja el. Ezt a kapcsolást a 4-76. ábrán látható mintatároló léptetőregiszter és a felhasználói interfészt megvalósító alkatrészek teszik teljes értékű UART-vevővé.

A közvetlenül az RX bemenetet – az *smp_pointer* segítségével kijelölt pillanatban – mintavételező léptetőregiszter *(frame_buffer)* mellé beépítünk egy kimeneti adatbuffert *(data_from_transmitter)* és egy, az adatkeretek vételének végét jelző kimenetet *(new_data_received)*, ami a *new_data_ack* bemenet magas szintre húzásával nullázható. Az UART-vevő felhasználói interfészének működése így a korábban bemutatott, kisebb bitsebességű vevőével megegyezik.

A nagysebességű UART-vevő nem foglalkozik külön a paritásbittel. Mivel az adatkeret bitszáma paraméter, ezért ezt a hiányosságot – legalább részben – könnyen kezelhetjük úgy, hogy a kilencbitesre beállított adatkeret egyik bitjét „kinevezzük" paritásbitnek. Ez a megoldás azért csak részleges, mert a paritásbit ellenőrzésének feladata az UART-vevő áramkör felhasználójára hárul.

Mivel a fenti áramkör felhasználói interfészének működése megegyezik a 4.3.1. pontban bemutatott UART-vevőnél látottakkal, ezért az ott javasolt, általános órajel-szinkronizáló burkolómodul használható a nagysebességű kivitel esetében is (lásd 4-2. táblázat).

[112] Figyeljük meg az *rx_df1* flip-flop kimenetének az órajel ötödik és nyolcadik felfutóélénél való mintavételezését; az *rx_df1* flip-flop és a mintát tároló flip-flop egyazon lokális adatút részei, a helyes mintavételhez szükséges időzítési feltételek teljesülése tehát biztosítottnak tekinthető úgy is, hogy a start-bit nullája vagy a *D[0]* adatbit után – vagyis a mintavétel pillanata után – az *rx_df1*-be beíródó értéket nem tudhatjuk biztosan.

4.3.5. Nagysebességű SPI slave

Áttanulmányozásra javasolt alfejezetek:
4.2. *On-chip adattárolás*

A szinkron soros SPI protokoll slave moduljának feladatkörét ellátó áramkör 4.3.2. pontban bemutatott változatában az *SCLK* jel úgy tölti be szinkronizáló funkcióját, hogy áramkörileg valójában adatként, statikus jelként kezeljük; változásait egy double-flop szinkronizáló fokozat és egy éldetektor kombinációjával alakítjuk át a lokális órajelhez szinkronizált impulzussokká, kihasználva azt, hogy e változások a lokális órajelforrás periódusidejénél sokkal nagyobb időközönként követik egymást. Az alábbiakban az SPI slave modulnak egy olyan változatát tervezzük meg, amely az *SCLK* jelet áramköri szempontból is valódi szinkronjelként, flip-flopok órajeleként használja.[113] Ez a megoldás lehetővé teszi, hogy jelentősen lecsökkentsük az *SCLK* jel és a lokális órajel periódusidejének arányát, vagyis növeljük az adott órajel-frekvencia esetén elérhető maximális adatátviteli sebességet. A slave modul tervezését néhány alapvető peremfeltétel rögzítésével kezdjük:

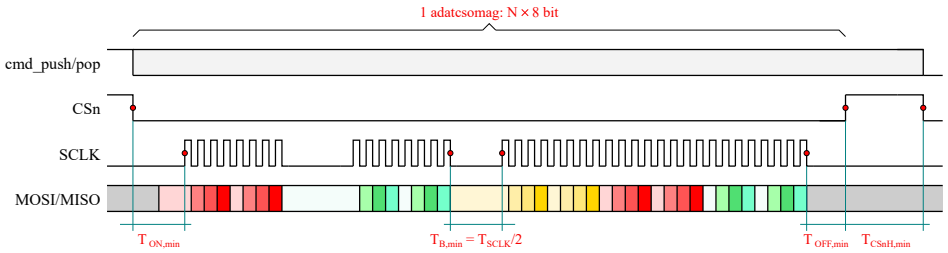
- A 4.3.2. pontban bemutatott, kissebességű változatnál látottakhoz hasonlóan ezúttal is az SPI protokoll MODE-0 üzemmódját valósítjuk meg; a *MISO* és *MOSI* vonalak mintavételezése az *SCLK* jel felfutó-, változtatása pedig mindig annak lefutóéleinél történik (lásd 4-47. ábra).
- A slave egységet a 4.3.2.1. pontban bemutatott példához hasonlóan FIFO-alapú felhaszs nálói interfésszel látjuk el, az IP-mag tehát felfogható egy SPI-FIFO soros-párhuzamos interfészadapterként, amely a mastertől érkező és annak küldendő adatok tárolásán felül órajel-szinkronizáló funkciót is ellát.
- Az SPI-tranzakció fogalmát kibővítjük; egy tranzakcióban egy N × 8 bites adatcsomag (chunk) kerül át a master egységből a slave-be és fordítva. N értéke futásidejű, az adatcsomagok átvitele között megváltoztatható paraméter. A *CSn* jel a teljes adatcsomag átvitele alatt alacsony értékű.
- A 4.3.2.1. pontban bemutatott példához hasonlóan itt is lehetővé tesszük a felhasználó számára, hogy a master → slave és slave → master irányú adatfolyamok átvitelét egymástól függetlenül engedélyezze vagy tiltsa. A vevő funkciót ellátó részáramkörök működése a *cmd_push*, az adó funkciókat ellátóké pedig a *cmd_pop* bemenettel engedélyezhető/tiltható. E két bemenet szintje egy adatcsomag átvitele közben nem változtatható, értéküket legkésőbb a *CSn* jel lefutóélével együtt stabilizálni kell.
- Az *SCLK* jel felfutó- és lefutóélei között feltételezünk egy minimális időtartamot ($T_{SCLK}/2$). Egyazon adatcsomagon belül az *SCLK* jelnek az n-edik bájt átviteléhez tartozó utolsó lefutóéle és az (n+1)-edik bájt átviteléhez tartozó első felfutóéle közötti minimális időtartam

[113] Vö. Ahogy elnevezése is utal rá, az U**A**RT egy aszinkron soros adatátviteli módszer, vagyis az adatvonal viselkedésének időzítése abszolút időmérésen alapul, nem áll rendelkezésre dedikált szinkronjel, amelynek élei az adatvonal mintavételezését időzíthetnék. Ezért aztán a kis- és nagysebességű vevő áramkörök működésében – az SPI-jal ellentétben – nincs alapvető különbség; az *RX* bemenetet egy double-flop szinkronizáló fokozat fogadja, amelynek kimenetét a megfelelő pillanatban egy léptetőregiszterbe töltjük.

szintén $T_{SCLK}/2$, vagyis az adatcsomagon belül a bájtok „bájtközi szünet” közbeiktatása nélkül, akár közvetlenül is követhetik egymást.

- A *CSn* jelnek az adatcsomag átvitelének kezdetét jelző lefutóéle és az *SCLK* jel első felfutóéle közötti minimális időtartam $T_{ON,min}$. Az *SCLK* jelnek az adatcsomag utolsó bitjének átviteléhez tartozó lefutóéle és a *CSn* jelnek a tranzakció végét jelző felfutóéle közötti minimális időtartam $T_{OFF,min}$. Két adatcsomag átvitele között a *CSn* jel legalább $T_{CSnH,min}$ időtartamig magas értékű. E három időzítési paraméterrel kapcsolatban a slave egység tervezése során fogalmazunk meg bizonyos feltételeket.

A nagysebességű slave modul időzítési modelljét a 4-77. ábra szemlélteti.



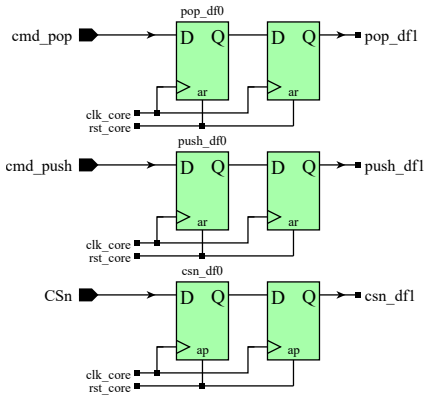
4-77. ábra. *Egy 4 bájtból álló adatcsomag átvitele a nagysebességű SPI csatornán*

A *CSn*, *cmd_push* és *cmd_pop* bemeneteket statikus jelként kezeljük, amelyeknek értéke egy adatcsomag átvitele alatt nem változik. A *CSn* jelet a távoli master egység hajtja meg, így e bemenetet feltétlenül double-flop szinkronizáló fokozattal kell fogadnunk. A *cmd_push* és *cmd_pop* vezérlőbemenetek beállítását a konkrét felhasználási körülményektől függően a távoli master egység és a slave modulnak egy másik órajeltartományban működő helyi felhasználója is végezheti, így e bemeneteket is double-flop szinkronizáló fokozatokkal látjuk el (lásd 4-78. ábra).

Verilog

```
1  // DOUBLE-FLOPS
2  always @ ( posedge clk_core, posedge rst_core )
3  begin
4  |  if ( rst_core == 1 ) begin
5  |  |  push_df0 <= 0; push_df1 <= 0;
6  |  |  pop_df0 <= 0; pop_df1 <= 0;
7  |  |  csn_df0 <= 1; csn_df1 <= 1;
8  |  end else begin
9  |  |  push_df0 <= cmd_push; push_df1 <= push_df0;
10 |  |  pop_df0 <= cmd_pop; pop_df1 <= pop_df0;
11 |  |  csn_df0 <= CSn; csn_df1 <= csn_df0;
12 |  end
13 end
```

Áramköri struktúra



4-78. ábra. *Double-flop szinkronizálók a modul statikus bemenetein*

A slave modulban az események időzítésének kézbentartására két számlálót készítünk; az egyik az adatcsomagon belüli bájtokat *(byte_counter)*, a másik pedig az adott pillanatban éppen átvitt bájton belüli biteket *(bit_counter)* számlálja. Mindkét számlálót az *SCLK* jel lefutóélei hajtják. A számlálók RTL-modelljét, valamint viselkedésüket egy négybájtos adatcsomag átvitele során a 4-79. ábra szemlélteti.

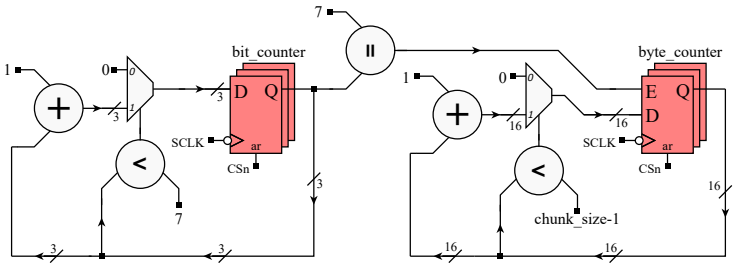
Verilog

```
1  // BIT-SYNCHRONIZATION
2  always @ ( negedge SCLK, posedge CSn ) begin
3    if ( CSn == 1 ) bit_counter <= 0;
4    else
5      if ( bit_counter < 7 )
6        bit_counter <= bit_counter + 1;
7      else bit_counter <= 0;
8  end
9
```

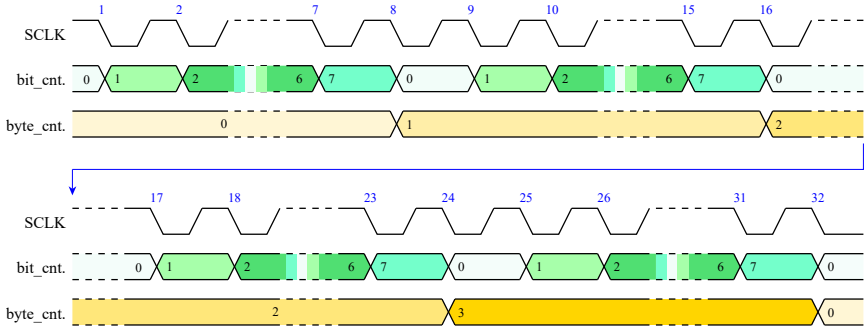
Verilog (folyt.)

```
10 // BYTE-SYNCHRONIZATION
11 always @ ( negedge SCLK, posedge CSn ) begin
12   if ( CSn == 1 ) byte_counter <= 0;
13   else
14     if ( bit_counter == 7 )
15       if ( byte_counter < chunk_size-1 )
16         byte_counter <= byte_counter + 1;
17       else byte_counter <= 0;
18 end
```

Áramköri struktúra



Hullámforma

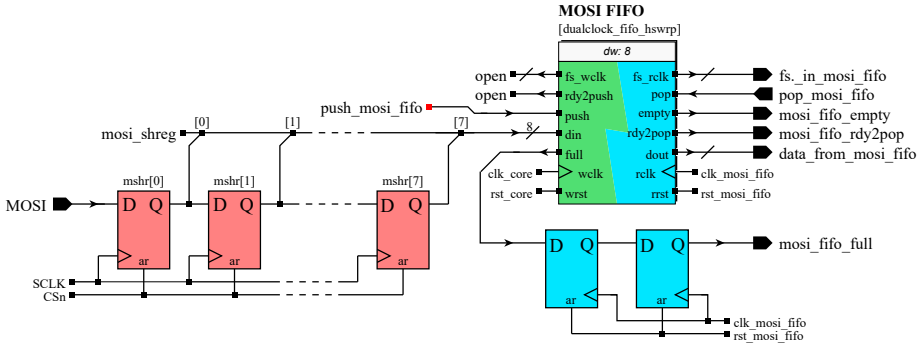


4-79. ábra. *Az adatcsomagok átvitelét ütemező számlálók*

Az SPI-FIFO interfészadapter áramkör vevő és adó funkciója egymástól jól szétválasztható, ezért ezeket külön tárgyaljuk.

Push-logika ◆ Amikor a master egy adatcsomagot küld a slave-nek, akkor az SPI protokoll szabályai szerint a *MOSI* vonal az *SCLK* jel felfutóéleinél mindig stabil, mintavételezhető. A 4.3.2. pontban bemutatott léptetőregiszteres struktúra tehát itt is kézenfekvő megoldás az adatbitek párhuzamosítására. A korábbi példával ellentétben azonban ezúttal magát az *SCLK* jelet használjuk a léptetőregisztert képező, sorbakapcsolt flip-flopok órajelbemeneteinek meghajtására. Így az

SCLK jel felfutóélei nem egy éldetektoron keresztül, hanem közvetlenül idézik elő a *MOSI* vonal mintavételezését (lásd 4-80. ábra).



4-80. ábra. *A nagysebességű SPI slave push-logikája (1)*
*dw: **d**ata **w**idth, fs: **f**ree **s**pace*

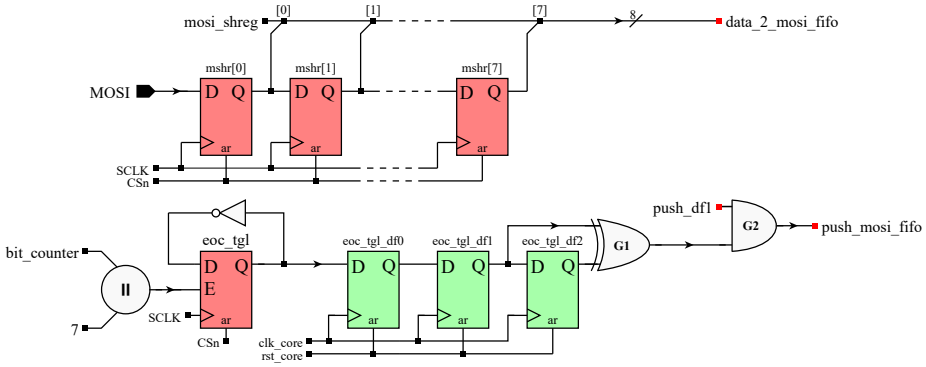
Az *SCLK* jellel meghajtott léptetőregiszter flip-flopjai képezte nyolcbites adatbuszt *(mosi_shreg)* a 4.2.5. pontban bemutatott *dualclock_fifo_hswrp* FIFO modul egy példányának adatbemenetére kapcsoljuk. A következő feladat az, hogy létrehozzuk a FIFO *push* bemenetén a megfelelő pillanatokban egy-egy pozitív impulzust előállító logikát. A „megfelelő pillanatok" minden nyolcadik mintavételezést követően adódnak, amikor a léptetőregiszter bitjei egy összetartozó bitsorozatot alkotnak. E pillanatok a *bit_counter* regiszter értéke alapján dekódolhatók; a nyolcadik adatbit az *SCLK* bemenet nyolcadik felfutóélénél töltődik be a léptetőregiszterbe, vagyis a *push* műveletet a *bit_counter* számláló hetes értékénél kell kezdeményeznünk (lásd 4-81. ábra).

A *bit_counter* számláló az *SCLK* bemenetről, a FIFO *push* interfésze viszont a modul *clk_core* magórajeléről jár, vagyis a *bit_counter* értéke alapján kijelölt bármely esemény csak valamilyen órajel-szinkronizáló fokozaton keresztül érzékeltethető a FIFO számára. Ez az órajel-szinkronizáló fokozat esetünkben egy visszacsatolatlan impulzusszinkronizáló (vö. DRMSz: 6-29. ábra); az *SCLK* bemenetnek a *bit_counter = 7* feltétel előállását követő felfutóélénél invertáljuk az *eoc_tgl* jelet. E változás egy double-flop szinkronizálón és egy éldetektoron keresztül egy, a *clk_core* órajelhez szinkronizált, egyetlen periódusnyi pozitív impulzust eredményez a *G1 XOR* kapu kimenetén. A *G1* kapu kimenete már alkalmas lenne arra, hogy a FIFO *push* bemenetét közvetlenül meghajtsa, a *G2 AND* kapu szerepe csupán annyi, hogy a FIFO-ba való tényleges beírást a *cmd_push* bemenet értékétől tegye függővé, vagyis a vevő áramkör működését letilthatóvá tegye.

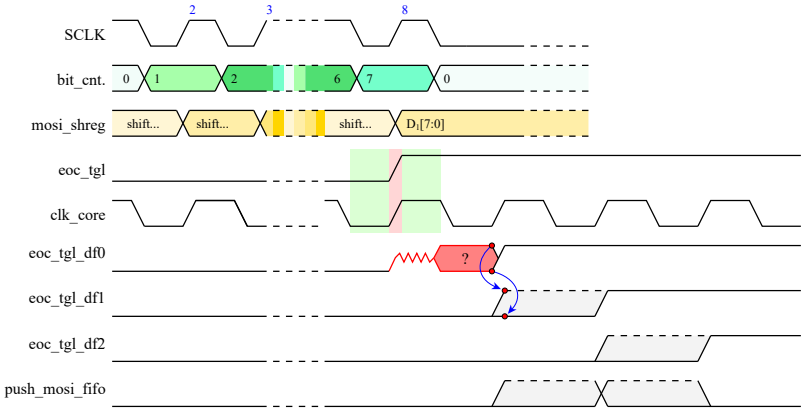
Az eddig bemutatott áramköri részletek alkalmasak a soros-párhuzamos átalakítás alapötletének szemléltetésére, de ha elmélyedünk az áramkör működésének részleteiben, észrevehetjük, hogy ez a megoldás így még több ponton is hibás. A továbbiakban három apró, e hibákat kiküszöbölő iterációt végzünk az RTL-modellen.

Az első probléma abból adódik, hogy a FIFO-ba való beírást végző visszacsatolatlan impulzusszinkronizálóhoz tartozó adatbusz, vagyis a *mosi_shreg* léptetőregiszter a *push* művelet kezdeményezésének és végrehajtásának időtartama alatt nem feltétlenül őrzi meg stabil értékét. A specifikációban rögzítettük, hogy egyazon adatcsomagon belül az egyes bájtok folyamatosan követhetik egymást, ami azt eredményezi, hogy egy-egy bájt FIFO-ba írásának tranziense alatt a *mosi_shreg* regiszterbe már elkezdhet beíródni a következő bájt. Ezt a folyamatot a 4-82. ábra hullámformája szemlélteti.

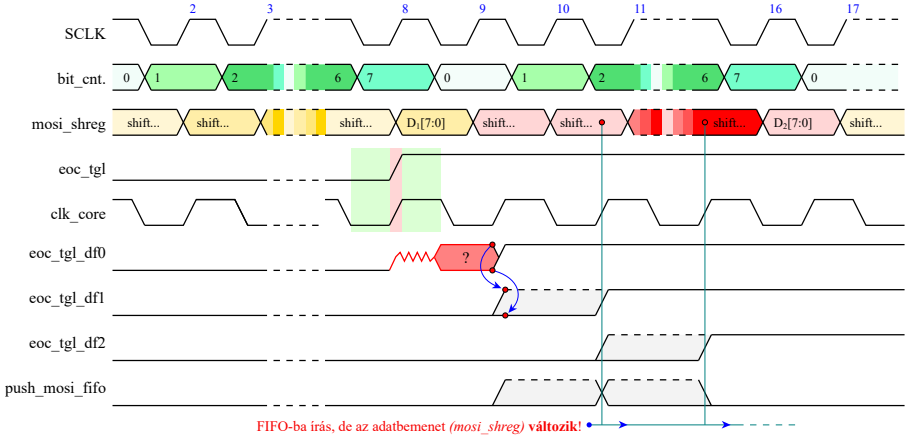
Áramköri struktúra



Hullámforma



4-81. ábra. *A nagysebességű SPI slave push-logikája (2)*
eoc_tgl: end of chunk toggle

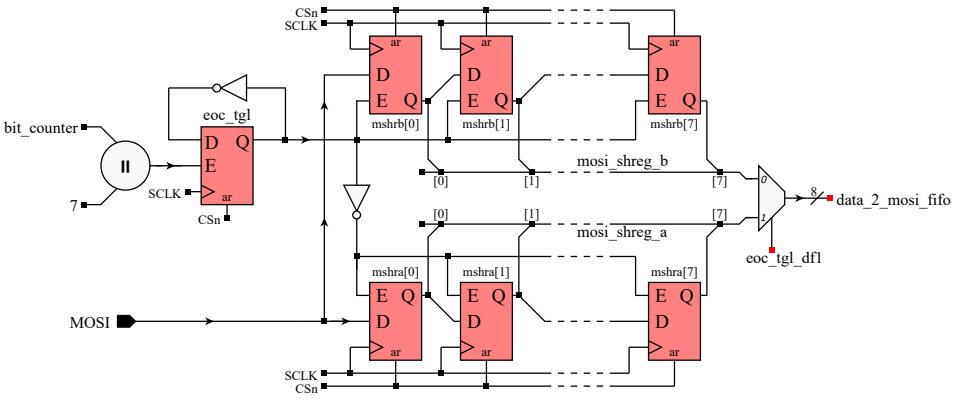


4-82. ábra. *A nagysebességű SPI slave push-logikája (3)*

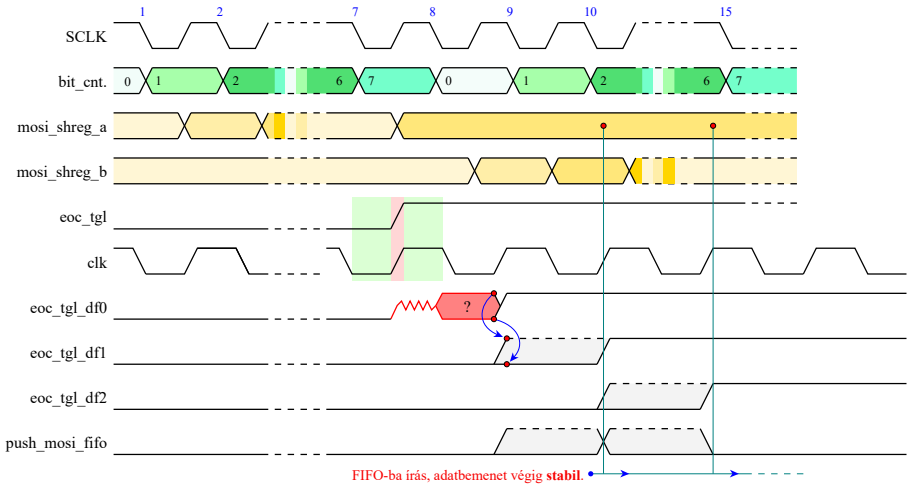
Verilog

```
1  // MOSI SHREGS
2  always @ ( posedge SCLK, posedge CSn ) begin
3    if ( CSn == 1 ) begin eoc_tgl <= 0; mosi_shreg_a <= 0; mosi_shreg_b <= 0; end
4    else begin
5
6      if ( bit_counter == 7 ) eoc_tgl <= ~eoc_tgl;
7
8      if ( eoc_tgl == 0 ) begin
9        mosi_shreg_a[0] <= MOSI;
10       for ( int i = 1; i < 8; i = i+1 ) mosi_shreg_a[i] <= mosi_shreg_a[i-1];
11     end else begin
12       mosi_shreg_b[0] <= MOSI;
13       for ( int i = 1; i < 8; i = i+1 ) mosi_shreg_b[i] <= mosi_shreg_b[i-1];
14     end
15
16   end
17 end
18
19 assign data_2_mosi_fifo = (eoc_tgl_df1 == 1) ? mosi_shreg_a : mosi_shreg_b;
```

Áramköri struktúra



Hullámforma

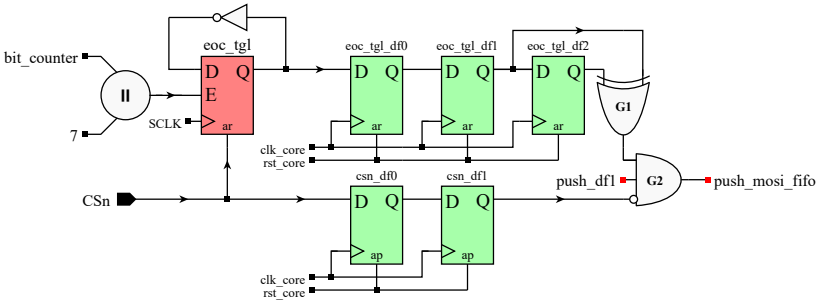


4-83. ábra. *A nagysebességű SPI slave push-logikája (4)*

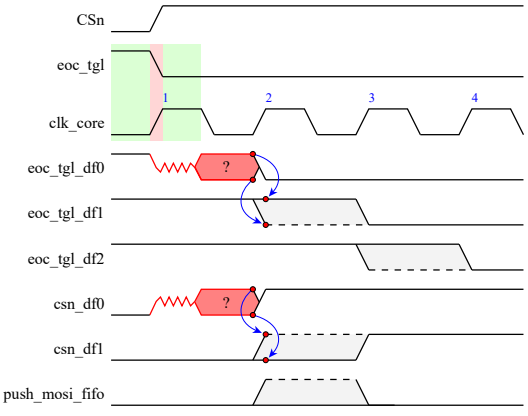
Az *eoc_tgl* szinkronizáló fokozat és a FIFO-ba való beírás együttes késleltetése nem okozna gondot, ha a teljes tranziens még az *SCLK* bemenet következő felfutóéle előtt végbemenne, ez azonban csak akkor lehetséges, ha a magórajel periódusideje sokkal kisebb, mint az *SCLK* periódusideje. Ez ellentmond a nagysebességű SPI slave modul alapkövetelményének, ezért a tervezés során szándékosan azt az esetet vizsgáljuk, amelyben az *SCLK* bemenet és a magórajel frekvenciája nagyjából azonos. A 4-82. ábrán érzékeltetett problémát egy második léptetőregiszter beiktatásával oldhatjuk meg (lásd 4-83. ábra).

Ebben a megoldásban egy bájt beléptetését követően az *eoc_tgl* jelet nemcsak arra használjuk, hogy kezdeményezzük vele a FIFO-ba való beírást, hanem arra is, hogy aktuális állapotától függően váltsunk a két léptetőregiszter között. Amíg az egyik léptetőregiszterbe az *SCLK* jel beléptei a *MOSI* vonalon érkező biteket, addig a másik léptetőregiszter állapota stabil marad. A két léptetőregiszter szerepe az *eoc_tgl* jel minden megváltozásakor, vagyis minden beérkezett bájt után felcserélődik. Az *SCLK* és *clk_core* órajelek viszonyát tekintve természetesen ez a megoldás sem mindenható; a FIFO-ba való beírás teljes tranziensének az *eoc_tgl* jel megváltozása utáni nyolcadik *SCLK* felfutóél előtt meg kell történnie. Ez azt jelenti, hogy a magórajelnek legalább olyan gyorsnak kell lennie, hogy az *SCLK* bemenet nyolc periódusába beleférjen az órajel-szinkronizáló

Áramköri struktúra



Hullámforma



4-84. ábra. *A nagysebességű SPI slave push-logikája (5)*

fokozat $3 \times T_{clk_core}$ késleltetése és a FIFO beírási késleltetése. Ezeket a feltételeket természetesen ellenőrizni kell minden alkalmazásban, de az kijelenthető, hogy ezzel a módszerrel a magórajel frekvenciájának nemhogy nagyságrendileg nem kell nagyobbnak lennie az *SCLK* jel frekvenciájánál, de kellően kis késleltetésű FIFO esetén akár annál kisebb is lehet.

Megfigyelve az eddig bemutatott áramköri sémákat, láthatjuk, hogy az *SCLK* órajeltartományban a flip-flopok aszinkron reset bemenetét a *CSn* jel hajtja meg. Ez egyrészt azt jelenti, hogy a *CSn* jel magas értékénél ezek a flip-flopok nem aktívak, az áramkör tehát – helyesen – nem reagál az *SCLK* jel esetleges változásaira, másrészt pedig azt, hogy a *CSn* jel felfutóéle pillanatában kimenetük nullázódik. Ez utóbbi sajátosság az impulzus-szinkronizáló fokozatban gondot okoz. Minden adatcsomag átvitele a *CSn* jel felfutásával végződik. Ha az adatcsomag páros számú bájtból állt, akkor ez a *CSn* felfutóél az *eoc_tgl* flip-flop magas értékénél következik be, vagyis a *CSn* felfutóéle egy állapotváltást idéz elő az impulzus-szinkronizáló fokozat bemenetén. Közvetetten tehát a *CSn* jelnek a tranzakció végét jelző felfutása is *push* műveletet idéz elő, ami funkcionálisan nyilván hibás. Ezt úgy küszöbölhetjük ki, hogy a *push* műveletet csak alacsony szintű *CSn* esetén engedélyezzük, vagyis a *G2 AND* kaput egy további bemenettel egészítjük ki (lásd 4-84. ábra).

Tekintve, hogy aszinkron órajeltartományok közötti adatátvitelről van szó, érdemes ezt a látszólag egyszerű megoldást kicsit tüzetesebben is megvizsgálni. A 4-84. ábra hullámformáján követhető, hogy mi történik a tranzakciók végét lezáró *CSn* felfutóél hatására. Az *eoc_tgl* flip-flop aszinkron reset bemenetére kapcsolt *CSn* jel felfutása a flip-flop kimenetének azonnali[114] nullázását idézi elő. Ez a változás egybeeshet a magórajel felfutóélével, vagyis az *eoc_tgl_df0* flip-flop – szerepköréből adódóan üzemszerűnek tekintendő módon – metastabil állapotba kerülhet. Ezzel párhuzamosan a *CSn* jel felfutása a *csn_df0* flip-flopban közvetlenül is metastabil állapotot idézhet elő. A két metastabillá váló flip-flop egymástól teljesen független, és mindegyikre igaz, hogy kimenete a metastabil állapotot előidéző él irányától független szinten fog stabilizálódni. Ha az *eoc_tgl_df0* flip-flop az órajel második felfutóéle előtt alacsony, a *csn_df0* flip-flop pedig magas szintre áll be, akkor az órajel második és harmadik felfutóéle között a *push_mosi_fifo* jelen előáll egy nemkívánatos pozitív impulzus, vagyis a 4-84. ábrán látható megoldás a tranzakcióvégi indokolatlan *push* műveletek kiszűrésére kudarcot vallott. A problémát az okozza, hogy a *csn_df1* jel magas értéke bizonyos esetekben túl későn érkezhet ahhoz, hogy az éldetektor kimenetén előálló impulzust a *G2 AND* kapu kiszűrje. A *CSn* jel megérkezését a saját double-flop szinkronizálójának kimenetére felgyorsítani nem tudjuk, de az *eoc_tgl* jelhez tartozó éldetektor késleltetését egy további flip-flop hozzáadásával könnyen megnövelhetjük (lásd 4-85. ábra).

Az *eoc_tgl_df3* flip-flop beiktatásával eredményül kapott időzítési viszonyokat szemléltető hullámformán jól látszik, hogy így a metastabil állapotok kialakulása ellenére a *CSn* jel felfutóéle semmiképp sem tud nemkívánatos impulzust okozni a *push_mosi_fifo* jelen, mivel a *csn_df1* flip-flop garantáltan korábban fut fel, minthogy az *eoc_tgl* jel változásának hatása megjelenne a *G1 XOR* kapu kimenetén.[115]

[114] A kimenet fizikailag valójában csak a flip-flop reset-késleltetése után válik nullává, de vizsgálódásunkban ennek most nincs jelentősége.

[115] Megjegyzendő, hogy a módosított impulzus-szinkronizáló fokozat késleltetése immár $4 \times T_{clk_core}$-ra nőtt, vagyis a *push* művelet teljes tranziensének időtartamát T_{clk_core} mértékben megnöveltük. Ezt a késleltetésnövekményt az *SCLK* és *clk_core* órajelek frekvenciaviszonyainak vizsgálatakor figyelembe kell venni.

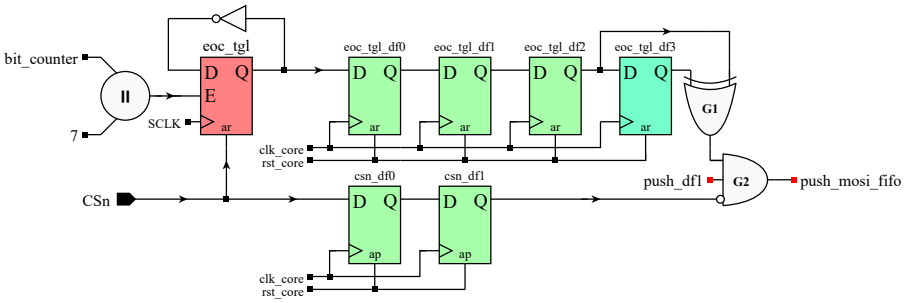
Verilog

```
1  // END-OF-CHUNK SIGNALLING
2  always @ ( posedge SCLK, posedge CSn )
3  begin
4    if ( CSn == 1 )
5      eoc_tgl <= 0;
6    else if ( bit_counter == 7 )
7      eoc_tgl <= ~eoc_tgl;
8  end
9
```

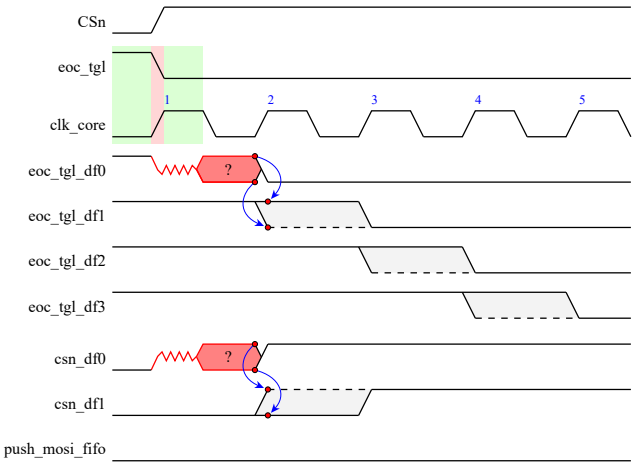
Verilog (folyt.)

```
10 // MOSI FIFO PUSH CONTROL
11 always @ ( posedge clk_core, posedge rst_core )
12 begin
13   if ( rst_core == 1 ) begin
14     eoc_tgl_df0 <= 0; eoc_tgl_df1 <= 0;
15     eoc_tgl_df2 <= 0; eoc_tgl_df3 <= 0;
16   end else begin
17     eoc_tgl_df0 <= eoc_tgl;     eoc_tgl_df1 <= eoc_tgl_df0;
18     eoc_tgl_df2 <= eoc_tgl_df1; eoc_tgl_df3 <= eoc_tgl_df2;
19   end
20 end
21
22 assign push_mosi_fifo = push_df1 & ~csn_df1 &
23                         (eoc_tgl_df2 ^ eoc_tgl_df3);
```

Áramköri struktúra

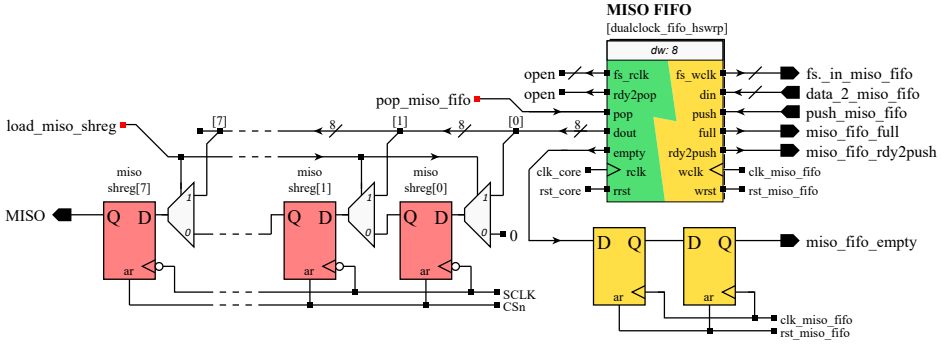


Hullámforma



4-85. ábra. *A nagysebességű SPI slave push-logikája (6)*

Pop-logika ◆ A párhuzamos-soros átalakítást végző *pop*-logika alapstruktúrája ugyancsak egy *dualclock_fifo_hswrp* FIFO modulnak és egy léptetőregiszternek a kombinációjaként készíthető el. A léptetőregiszter flip-flopjainak órajelbemenetét ezúttal is közvetlenül az *SCLK* jel hajtja. Mivel a *MISO* kimenetet a léptetőregiszter legfelső helyiértékű bitje adja, és az SPI protokoll szabályrendszere szerint e kimenet csak az *SCLK* lefutóéleinél változhat, ezért a léptetőregiszter órajelbemeneteit ezúttal negáljuk (lásd 4-86. ábra).



4-86. ábra. *A nagysebességű SPI slave pop-logikája (1)*
fs: free space

Az adatcsomagok átvitelét ütemező *bit_counter* és *byte_counter* számlálók felhasználásával olyan logikát kell létrehoznunk, amely egyrészt képes kiolvasni a FIFO-ból a masternek küldendő adatot, másrészt a FIFO-ból kiolvasott bájtot mindig a megfelelő pillanatban tölti át a *MISO* kimenetet hajtó léptetőregiszterbe. Első megoldásként a *pop* művelet kezdeményezését kössük az *SCLK* jelnek a bájthatárokat követő első lefutóéléhez. Ehhez létrehozzuk a *pop_sclk* flip-flopot, amelynek értékét a *bit_counter = 0* feltétel fennállásakor negáljuk. E változást a *push*-logikánál is használt impulzus-szinkronizáló fokozattal visszük át a FIFO órajeltartományába (lásd 4-87. ábra).

Az impulzus-szinkronizáló fokozat *G2* kapujának szerepe ugyanaz, mint a *push*-logika esetén; a *pop* műveletet csak akkor kezdeményezzük, ha a *cmd_pop* bemenet magas szintű, tehát a felhasználó a tranzakció elején nem tiltotta le a slave → master irányú adatátvitelt, a *CSn* bemenet double-flop szinkronizálón keresztül ellenőrzött értéke pedig megakadályozza, hogy a tranzakciókat lezáró *CSn* felfutás *pop* műveletet idézzen elő.

A *pop_sclk* jelnek az (n+1)-edik bájt FIFO-ból való kiolvasását kezdeményező változása áthalad az impulzus-szinkronizáló fokozaton, miközben az n-edik bájtnak a léptetőregiszterben lévő bitjeit az *SCLK* lefutóélei kiléptetik a *MISO* vonalra. A FIFO adatkimenetét a *bit_counter = 7* feltétel fennállásakor töltjük át a léptetőregiszterbe. Az n-edik bájt átvitele és az (n+1)-edik bájt FIFO-ból való kiolvasása tehát átlapoltan megy végbe, ami a magórajel és az *SCLK* bemenet frekvenciaviszonyaira vonatkozóan a *push*-logikánál látottakhoz hasonló következménnyel jár; a magórajelnek legalább olyan gyorsnak kell lennie, hogy az *SCLK* nyolc periódusa alatt a *pop* művelet kezdeményezése elérjen a FIFO *pop* bemenetéig és a FIFO kiolvasása is megtörténjen.

E megoldás hátránya, hogy az első adatcsomag első bájtjának átvitelekor a léptetőregiszterben még nincs hasznos adat, hiszen a léptetőregiszter első betöltésére csak az első *pop* művelet után kerül sor.[116] A problémát úgy orvosolhatjuk, hogy a FIFO kiolvasását nemcsak a bájthatárokon, hanem már rögtön a *CSn* jel lefutóélekor is kezdeményezzük, illetve a FIFO-ból kiolvasott adat áttöltését a léptetőregiszterbe korábban, már a *bit_counter = 0* feltétel teljesülésekor végrehajtjuk. Mivel ebben az esetben egy adott adatcsomag első bájtjának FIFO-ból való kiolvasását maga a *CSn*

[116] A második adatcsomagnál ez a probléma már nem áll fenn, mert annak első bájtját az első adatcsomag utolsó bájtjának kiléptetésével párhuzamosan kiolvassuk a FIFO-ból.

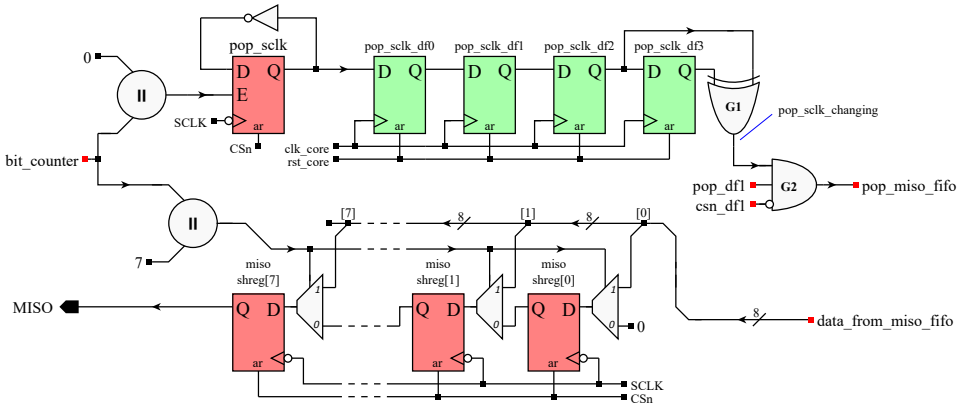
Verilog

```
1  // POP SIGNALLING
2  always @ ( negedge SCLK, posedge CSn )
3  begin
4    if ( CSn == 1 ) pop_sclk <= 0;
5    else if ( bit_counter == 0 ) pop_sclk <= ~pop_sclk;
6  end
7
8  // MISO SHREG
9  always @ ( negedge SCLK, posedge CSn )
10 begin
11   if ( CSn == 1 ) miso_shreg <= 0;
12   else
13     if ( bit_counter == 7 )
14       miso_shreg <= data_from_miso_fifo;
15     else begin
16       miso_shreg[0] <= 0;
17       for ( int i = 1; i < 8; i = i+1 )
18         miso_shreg[i] <= miso_shreg[i-1];
19     end
20 end
21
22 // MISO GENERATION
23 assign MISO = miso_shreg[7];
```

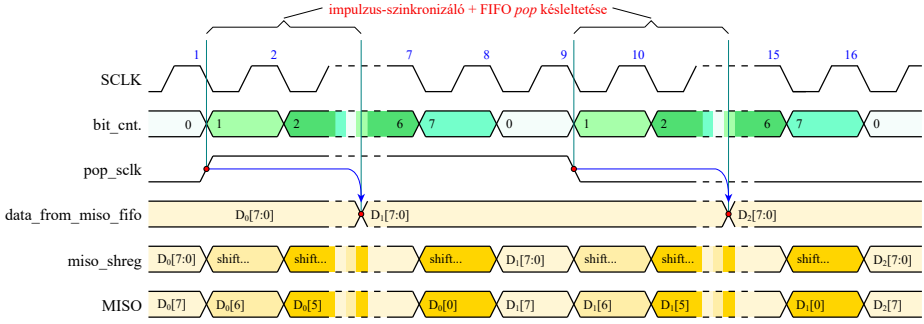
Verilog (folyt.)

```
24 // MISO FIFO POP CONTROL
25 always @ ( posedge clk_core,posedge rst_core )
26 begin
27   if ( rst_core == 1 ) begin
28     pop_sclk_df0 <= 0;
29     pop_sclk_df1 <= 0;
30     pop_sclk_df2 <= 0;
31     pop_sclk_df3 <= 0;
32   end else begin
33     pop_sclk_df0 <= pop_sclk;
34     pop_sclk_df1 <= pop_sclk_df0;
35     pop_sclk_df2 <= pop_sclk_df1;
36     pop_sclk_df3 <= pop_sclk_df2;
37   end
38 end
39
40 assign pop_sclk_changing = pop_sclk_df2 ^
41                            pop_sclk_df3;
42 assign pop_miso_fifo = pop_sclk_changing &
43                        pop_df1 &
44                        ~csn_df1;
```

Áramköri struktúra



Hullámforma



4-87. ábra. *A nagysebességű SPI slave pop-logikája (2)*

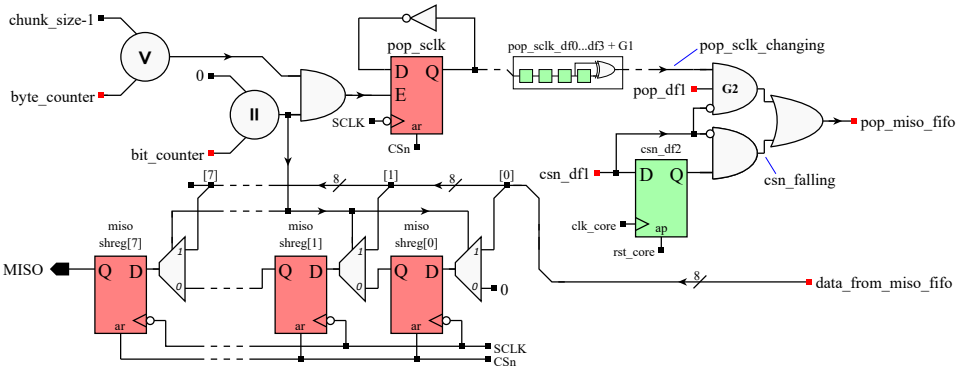
Verilog

```
1  // POP SIGNALLING
2  always @ ( negedge SCLK, posedge CSn )
3  begin
4    if ( CSn == 1 ) pop_sclk <= 0;
5    else if (
6     (bit_counter == 0) &
7     (byte_counter < chunk_size-1)
8    ) pop_sclk <= ~pop_sclk;
9  end
10
11 // MISO SHREG
12 always @ ( negedge SCLK, posedge CSn )
13 begin
14   if ( CSn == 1 ) miso_shreg <= 0;
15   else
16     if ( bit_counter == 0 )
17       miso_shreg <= data_from_miso_fifo;
18     else begin
19       miso_shreg[0] <= 0;
20       for ( int i = 1; i < 8; i = i+1 )
21         miso_shreg[i] <= miso_shreg[i-1];
22     end
23 end
24
```

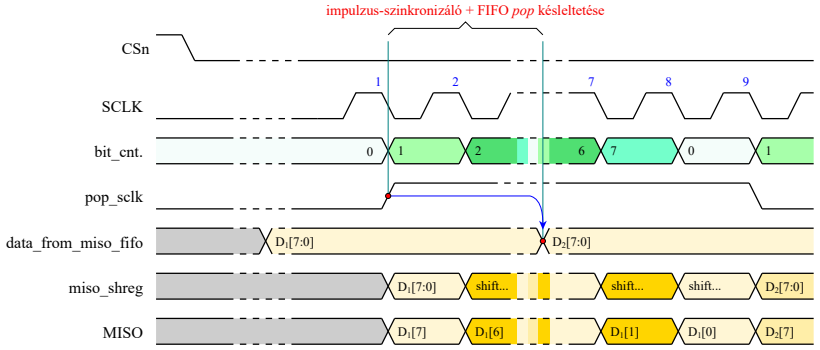
Verilog (folyt.)

```
25 // MISO GENERATION
26 assign MISO = miso_shreg[7];
27
28 // MISO FIFO POP CONTROL
29 always @ ( posedge clk_core, posedge rst_core )
30 begin
31   if ( rst_core == 1 ) csn_df2 <= 1;
32   else csn_df2 <= csn_df1;
33 end
34 assign csn_falling = ~csn_df1 & csn_df2;
35
36 assign pop_sclk_changing = pop_sclk_df2 ^
37                            pop_sclk_df3;
38
39 assign pop_miso_fifo = (pop_sclk_changing &
40                         pop_df1 & ~csn_df1) |
41                         csn_falling;
```

Áramköri struktúra



Hullámforma



4-88. ábra. *A nagysebességű SPI slave pop-logikája (3)*

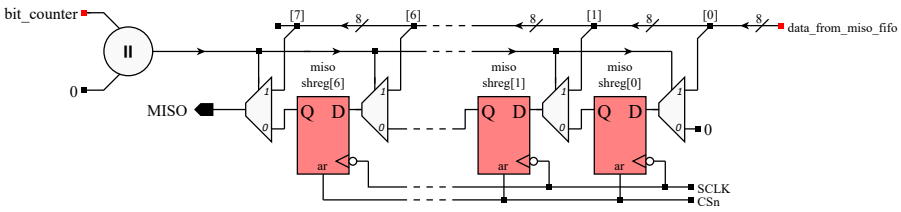
jel lefutása kezdeményezi[117], ezért a *pop_sclk* jel változását az adatcsomag utolsó bájtjának átvitele alatt – a *byte_counter = chunk_size-1* feltétel fennállása esetén – már le kell tiltanunk. Az e megfontolások mentén módosított áramköri sémát és annak viselkedését a 4-88. ábra szemlélteti.

A hullámformát tüzetesen megvizsgálva észrevehetjük, hogy mivel a léptetőregiszter betöltésének pillanatát megváltoztattuk, ezért a *MISO* vonalon a léptetőregiszterben tárolt bitek egy ciklus késéssel jelennek csak meg. Ennek oka, hogy bár a *CSn* bemenet lefutóél-detektora képes kezdeményezni a FIFO kiolvasását, de a kiolvasott adat léptetőregiszterbe való áttöltésére csak az *SCLK* első lefutóéle hatására kerülhet sor, hiába igyekeztünk a *bit_counter = 0* feltétellel az áttöltést

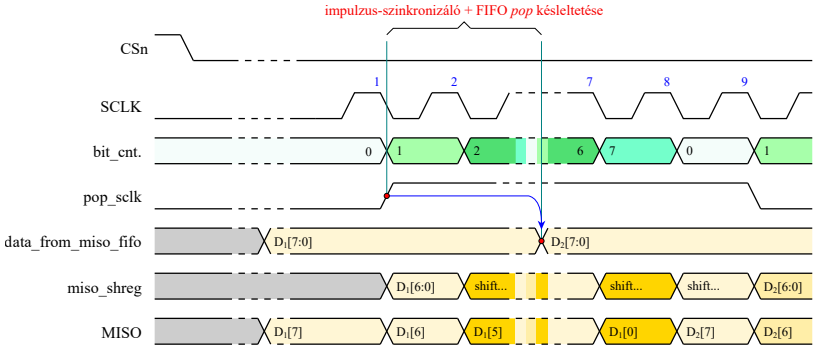
Verilog

```
11 // MISO SHREG
12 always @ ( negedge SCLK, posedge CSn ) begin
13   if ( CSn == 1 ) miso_shreg <= 0;
14   else if ( bit_counter == 0 ) miso_shreg <= data_from_miso_fifo;
15        else begin
16          miso_shreg[0] <= 0;
17          for ( int i = 1; i < 7; i = i+1 )
18            miso_shreg[i] <= miso_shreg[i-1];
19        end
20 end
21
22 // MISO GENERATION
23 assign MISO = ( bit_counter == 0 ) ? data_from_miso_fifo[7] : miso_shreg[6];
```

Áramköri struktúra



Hullámforma



4-89. ábra. *A nagysebességű SPI slave pop-logikája (4)*

[117] A *CSn* jel lefutásának felhasználása a FIFO első kiolvasására azt az időzítési peremfeltételt hozza magával, hogy a *CSn* lefutása és az *SCLK* első felfutása előtt legalább annyi időnek el kell telnie, hogy a *CSn* double-flop szinkronizáló + éldetektor struktúrán a lefutóél hatása keresztüljuthasson, és a FIFO tényleges kiolvasása is megtörténhessen. E feltétel teljesülését ugyancsak vizsgálni kell minden alkalmazásban az *SCLK* jel és a magórajel frekvenciájának összehangolása során.

a lehető legkorábbi pillanatra hozni. Így azonban az *SCLK* első felfutóélekor a master még nem láthat érvényes adatot a *MISO* vonalon. A helyzetet úgy kezelhetjük, hogy a léptetőregiszter legfelső helyiértékű flip-flopját egyszerűen eltávolítjuk (lásd 4-89. ábra).

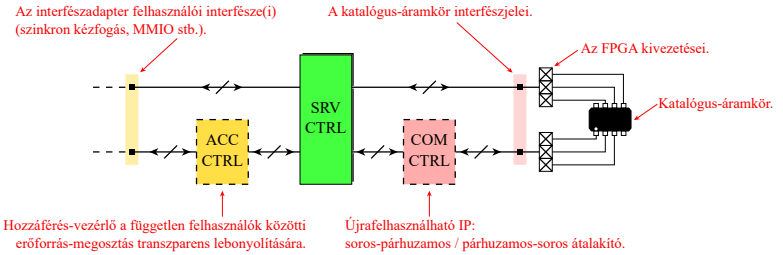
Így a *CSn* lefutóéle által kezdeményezett *pop* művelet hatására a FIFO adatkimenetén megjelenő bájt legfelső helyiértékű bitje közvetlenül a *MISO* kimenetre íródik, tehát az *SCLK* első felfutóélekor az már érvényes adatot mutat a master felé annak ellenére, hogy az immár csak hétbites léptetőregiszterbe csak az *SCLK* első lefutóélekor íródik be a FIFO-ból kiolvasott adat. Az eddig a léptetőregiszter legfelső helyiértékű bitjének bemenetét hajtó multiplexert nem mozdítottuk el, vagyis a FIFO adatkimenete csak a *bit_counter = 0* feltétel fennállása, tehát a FIFO-ból kiolvasott bájtok legelső bitjének kiléptetése idejére kap jogot a *MISO* kimenet közvetlen meghajtására. Az összes többi bit esetén a *MISO* kimenetet a léptetőregiszter 6. helyiértékű bitje adja.

4.4. Katalógus-áramkörök interfészadapterei

Funkcióbővítő rendszerünk feladata – ahogy azt az 1-1. ábra is érzékelteti –, hogy egy egyszerűnek és olcsónak feltételezett üzemeltető mikrovezérlő számára különféle többletképességeket biztosítson egy egységesített interfészen keresztül. E többletképességek egy részét magában a funkcióbővítő rendszert tartalmazó FPGA-ban létrehozott bővítő áramkörök biztosítják (pl. jelfeldolgozást végző IP-magok), lesz azonban olyan többletfunkció is, amelyet az FPGA-hoz kapcsolt katalógus-áramkörök (pl. A/D-átalakítók, érzékelők) valósítanak meg. Ez esetben a funkcióbővítő rendszer afféle közvetítő szerepben van, a katalógus-áramkörök szolgáltatásait teszi elérhetővé az üzemeltető rendszer számára.

A katalógus-áramkörök interfészei feladatkörüktől függően sokfélék lehetnek, de a gyártók – termékük beépítését megkönnyítendő – igyekeznek legalább részben szabványos megoldásokat használni. Ezért az ilyen áramköröknél elterjedtek az 5.2. Függelékben tárgyalt egyszerű soros kommunikációs interfészek, de – ha az átviteli sebességre vonatkozó elvárásokat a soros kommunikációs technikák nem tudják kielégíteni – találkozhatunk sok kivezetést igénylő, gyors párhuzamos interfészekkel is, amelyek ugyancsak lehetnek szabványosak (pl. SDRAM-interfész) vagy teljesen egyediek is.

Ahhoz, hogy lehetővé tegyük az üzemeltető mikrovezérlő számára a katalógus-áramkörök szolgáltatásaihoz való hozzáférést, olyan illesztőlogikák megtervezésére van szükség, amelyek hidat képeznek az adott katalógus-áramkör saját interfésze és a funkcióbővítő rendszer almoduljainak interfészei között. Azokat az áramköröket, amelyek egy konkrét katalógus-áramkör szolgáltatásait egy IP-k között használatos interfészen – esetünkben ez a szinkron kézfogásos interfész vagy az



4-90. ábra. *Az interfészadapter IP-magok szerkezete*

MMIO – teszik elérhetővé, interfészadapter (**I**nter**f**ace **A**dapter, IFA) áramköröknek nevezzük. Az interfészadapterek tervezése során is érdemes a modularitás és a regularitás alapelveit szem előtt tartani, ennek megfelelően az itt bemutatott IFA áramkörök RTL-modelljei a 4-90. ábrán szemléltetett, egységes séma szerint épülnek fel.

Az egységes szerkezeti séma központi eleme az eszközspecifikus szolgáltatásvezérlő *(SRV CTRL)*, ami általában egy vagy több, a katalógus-áramkör sajátosságai alapján megtervezett, összetett állapotgép. Ez az állapotgép közvetlenül, és/vagy egy újrafelhasználható IP-magként rendelkezésre álló párhuzamos-soros/soros-párhuzamos átalakító almodulon *(COM CTRL)* keresztül kapcsolódik a katalógus-áramkör kivezetéseihez.[118] Az általános séma harmadik, ugyancsak opcionális szerkezeti eleme a hozzáférés-vezérlő *(ACC CTRL)*, amely akkor jut szerephez, ha a katalógus-áramkörhöz az interfészadapteren keresztül több, egymástól független felhasználó szeretne (látszólag) egyszerre hozzáférni. Ilyenkor a hozzáférés-vezérlő egyfajta arbiter szerepet tölt be, gondoskodik a katalógus-áramkör időosztásos használatának transzparens lebonyolításáról.[119] „Transzparencia" alatt itt azt értjük, hogy az egyes felhasználók szempontjából úgy tűnik, mintha egyedül ők lépnének közvetlen kölcsönhatásba a szolgáltatásvezérlővel, nem kell tehát azzal foglalkozniuk, hogy rajtuk kívül más almodulok is használják az interfészadapteren keresztül a közös erőforrást (a katalógus-áramkört). A hozzáférés-vezérlő biztosítja, hogy a felhasználói interfészek működési mechanizmusa a felhasználók számától független legyen. Érdemes megjegyezni, hogy ez csak olyan interfész-specifikáció esetén kivitelezhető, amelyben a tranzakciókhoz szükséges időtartam nincs előre rögzítve (lásd pl. a kézfogásos mechanizmust akár a szinkron kézfogásos, akár az MMIO-interfész esetén).

Az interfészadapter áramkörök alapvető feladata, hogy a katalógus-áramkörök szolgáltatásait kényelmesen használható interfészeken keresztül tegyék elérhetővé, de elkészítésük során mindig mérlegelendő, hogy az adott katalógus-áramkörnek minden szolgáltatására szükségünk van-e. A katalógus-áramkörök egyik legfontosabb jellemzője, hogy nem egy konkrét alkalmazásra készülnek, hanem az alkalmazásoknak a lehető legszélesebb körét igyekeznek lefedni, aminek eredményeképpen rengeteg funkciót, üzemmódot és paramétert bocsátanak a felhasználó rendelkezésére.[120] A szerteágazó szolgáltatáskészletet maradéktalanul leképező interfészadapterek meglepően bonyolultak lehetnek, ezért az ilyen illesztőlogikák tervezése során gyakran a kiszemelt katalógus-áramkör funkcióinak csak egy részét ajánljuk ki a felhasználónak, így gyorsítva és egyszerűsítve a fejlesztést. Az e fejezetben bemutatott, memória jellegű katalógus-áramkörök interfészadaptereivel szemben támasztott elsődleges követelmény, hogy tegyék transzparenssé

[118] A funkcióbővítő rendszerhez kiválasztott konkrét katalógus-áramkörök típusából adódóan a *COM CTRL* almodul esetünkben az SPI master vagy az I^2C master IP-mag egy-egy példánya lesz. Összetettebb katalógus-áramkörök (pl. komplex autóipari szenzorok) esetén egyéb soros kommunikációs interfészek (UART, LIN, CAN stb.) is szóba jöhetnek.

[119] A memória jellegű katalógus-áramkörökhöz egyszerűsített hozzáférés-vezérlőket készítünk, amelyek feladata az egymástól függetlennek tekintett beíró és kiolvasó felhasználói interfészek közötti erőforrás-megosztás. A hozzáférés-vezérlés általános megvalósítására a 4.1.2. pontban tárgyalt prioritásos hozzáférés-vezérlő használható (lásd pl. 2.5. pont: MMIO megszakításvezérlő, 2.6.2. pont: MMIO memória-arbiter).

[120] Ebben az esetben is az újrafelhasználhatóságra és rugalmasságra való törekvésnek az ellentmondását látjuk, amit a mikrovezérlők általános felhasználhatóságra szánt perifériakészletével kapcsolatban az 1. fejezet bevezetőjében, a technológiafüggetlen RTL-modellek vonatkozásában pedig a 2.2.2. pont végén említettünk. Annak érdekében, hogy sok alkalmazásban használhatók legyenek, egyes katalógus-áramkörök rengeteg funkciót tartalmaznak, amelyeknek a nagy része egy-egy konkrét alkalmazási környezetben óhatatlanul kihasználatlan lesz. Korábbi megállapításunk, mely szerint készíthetők olyan megoldások, amelyek sokféle felhasználási környezetben „elég jók", de olyanok nem, amelyek sokféle alkalmazásban optimálisak, ezúttal is helytállónak bizonyul.

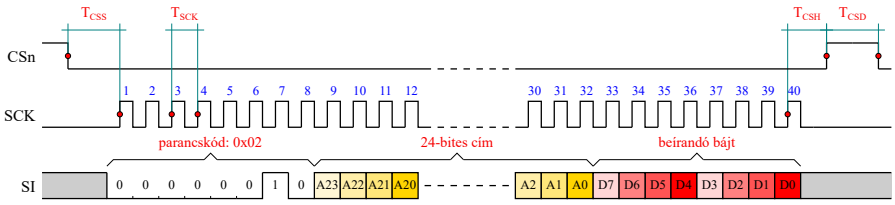
a memóriaszerű működés technológiai/fizikai részleteit, vagyis a felhasználó felé kiajánlott interfészük működése egyezzen meg a kézfogásos logikával kiegészített, on-chip RAM-modulokat használó IP-k interfészeinek működésével *(singleclock_dualport_ram)*. Ennek megfelelően az itt bemutatott, memória-funkciójú katalógus-áramköröknek csupán az egyetlen tranzakcióban egyetlen adategységet (bájt vagy duplabájt) beíró vagy kiolvasó szolgáltatásait vesszük figyelembe, dacára annak, hogy szinte minden memóriaszerű katalógus-áramkör biztosítja a blokkos adatátvitel lehetőségét, a memóriatartalom részleges vagy teljes törlését, az állapotinformációk lekérdezését, üzemmódváltást stb.

4.4.1. SPI EEPROM

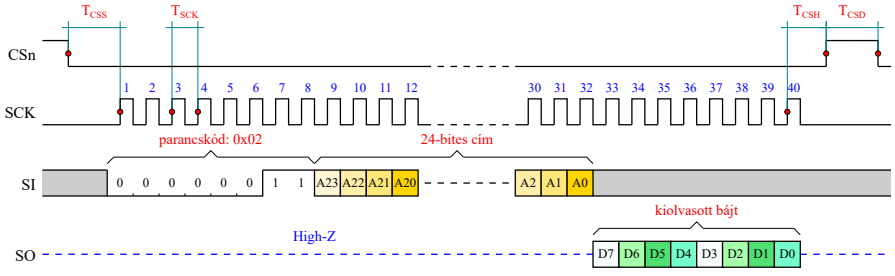
Áttanulmányozásra javasolt alfejezetek:

***4.3.2.** A soros kommunikáció alapáramkörei – SPI master és slave (+ **4.3.5:** Nagysebességű SPI slave)*

A 25LC1024 típusjelű katalógus-áramkör egy, a Microchip Technology Inc. által gyártott, 1.024 kbit (131.072 bájt) kapacitású, SPI-interfésszel szerelt EEPROM chip. Adatlapja rengeteg információt rejt használatával kapcsolatban, de a hozzá tartozó interfészadapter megtervezéséhez számunkra csupán néhány időzítési diagram releváns, amelyek a beírás és kiolvasás műveletének részleteit rögzítik. Az EEPROM chip bájtos szervezésű, és egyetlen bájt beírása és kiolvasása is egy-egy 4 × 8-bites SPI-tranzakciót igényel (lásd 4-91. és 4-92. ábra).

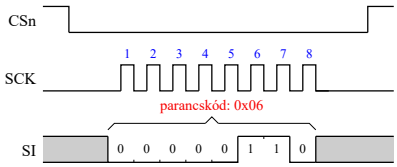


4-91. ábra. *A 25LC1024 EEPROM chip írásához tartozó SPI-tranzakciók időzítése*



4-92. ábra. *A 25LC1024 EEPROM chip olvasásához tartozó SPI-tranzakciók időzítése*

A 4 × 8-bites tranzakciók a *CSn* jel alacsony szintre húzásával indíthatók, amit a nyolcbites parancskód, majd a 24-bites[121] cím kiküldése követ. A parancskód írás és olvasás esetén nyilván eltér; beíráskor a 0x02, kiolvasáskor a 0x03 kódot léptetjük ki az SPI master interfész soros adatkimenetén *(MOSI)*, amire az EEPROM adatlapja *SI (**S**erial **I**nput)* néven hivatkozik. A parancskód és a cím kiküldését írás esetén a beírandó bájt kiküldése, olvasáskor pedig a megcímzett rekesz tartalmának az eszköz soros *SO (Serial Output)* adatkimenetéről – az SPI master interfész *MISO* jelének megfelelő kimenetről – való beolvasása követi. Az EEPROM adatlapja szerint minden írás műveletet külön engedélyezni kell, ami – ahogy azt a 4-93. ábra szemlélteti – a 0x06 parancskód kiküldésével tehető meg. Az interfészadapter tervezésekor ugyancsak figyelembe veendő tulajdonság, hogy az EEPROM chip írásakor az adat tényleges elhelyezése a megfelelő memóriarekeszbe 6 ms alatt történik meg, függetlenül attól, hogy az írástranzakció mennyi idő alatt megy végbe.[122]



4-93. ábra. *A beírást engedélyező parancs időzítése*

A 4-91. és 4-92. ábrákon az SPI-interfésszel szerelt katalógus-áramkörök illesztésekor ellenőrizendő legfontosabb időzítési tulajdonságokat is feltüntettük, amelyek betartása elengedhetetlen az eszköz helyes működéséhez.[123] Ezek magyarázatát a 4-6. táblázat foglalja össze.

Időzítési jellemző neve	Időzítési jellemző leírása	Időzítési jellemző értéke
T_{SCK}	Az SPI master interfész *SCLK* jelének megfelelő *SCK* jel periódusideje.	≥ 100 ns
T_{CSS}	A *CSn* lefutása és az *SCK* első felfutása közötti időtartam.	≥ 50 ns
T_{CSD}	Időtartam két SPI-tranzakció között, amíg a *CSn* jel magas szintű.	≥ 50 ns
T_{CSH}	Időtartam a tranzakció utolsó órajel-felfutóéle és a *CSn* felfutása között.	≥ 100 ns

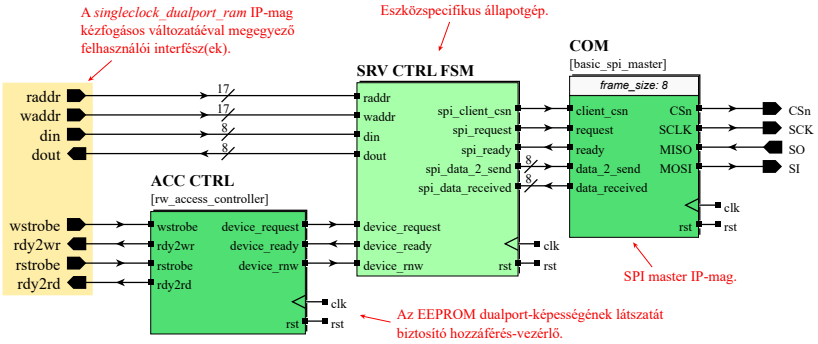
4-6. táblázat. *Az EEPROM SPI-interfészének legfontosabb időzítési jellemzői*

[121] Felmerülhet a kérdés, hogy a 131.072 darab egybájtos rekesz megcímzésére miért használunk 24 bitet, hiszen 17 is elég lenne. Bár az SPI protokoll nem rögzíti az egy tranzakción belül átvitt bitek számát (a 4.3.2. pontban bemutatott SPI master modulunk ezt paraméterként kezeli), de a mikrovezérlőkbe épített SPI master áramkörök kevésbé rugalmasak, jellemzően csak N × 8-bites tranzakciókat támogatnak. A másik ok a számunkra is oly fontos regularitás alapelvét idézi fel; a gyártó a katalógus-áramkör különböző kapacitású változataihoz ugyanazt az SPI slave interfész-áramkört építi be, így az eszköz különféle változatai felhasználói szempontból egyenértékűnek tekinthetők. Egy FPGA-ban megvalósított interfészadapter IP vagy egy mikrovezérlőben futó programkód-részlet (pl. egy EEPROM-ba író függvény) egészen 16 MB kapacitásig változtatás nélkül használható. A 16 MB-nál kisebb kapacitású eszközök a cím felső bitjeit egyszerűen figyelmen kívül hagyják. A tárkapacitás korlátját a felhasználónak nyilván figyelembe kell vennie, de az adatátvitel mechanizmusa ettől eltekintve egységes.

[122] A lehetséges legmagasabb *SCK* frekvencia (10 MHz) esetén az írás és olvasástranzakciók kb. 4,1 μs alatt lezajlanak.

[123] Az eszköz adatlapjában további időzítési jellegzetességeket és peremfeltételeket is találunk (pl. clock-to-output késleltetés az eszköz *SO* kimenetére vonatkozóan, az *SI* bemenet beállítási és tartási idői), amelyeket különösen akkor érdemes tüzetesen megvizsgálni, ha az interfész – az EEPROM példájával ellentétben – nagysebességű (> 10 MHz, vö. 4.4.3.1. pont).

Mivel az interfészadapterrel az egyszerű bájtíráson és -olvasáson túl egyéb szolgáltatást nem szeretnénk leképezni, ezért a 4-91. és 4-92. ábrák diagramjai elegendő információval szolgálnak az interfészadapter RTL-modelljének elkészítéséhez, amelynek kapcsolási sémáját a 4-94. ábra mutatja be.



4-94. ábra. *A 25LC1024 EEPROM interfészadapterének szerkezete*
*rnw: **r**ead **n**ot **w**rite (olvasáskor magas, íráskor alacsony szintű vezérlőjel)*

Az interfészadapterek általános sémájának komponensei közül az eszközspecifikus szolgáltatásvezérlő szerepét egy egyedi állapotgép, a soros-párhuzamos/párhuzamos-soros átalakítóét a 4.3.2. pontban bemutatott SPI master modul *(basic_spi_master)*, a hozzáférés-vezérlőét pedig egy ugyancsak újrafelhasználható almodul *(rw_access_controller)* tölti be.

Ez utóbbi részegység feladata az, hogy több felhasználó esetén transzparens módon kezelje az egymástól függetlenül, tehát potenciálisan egyszerre érkező felhasználói kéréseket. Általános esetben több felhasználóról persze csak akkor beszélhetünk, ha az interfészadapter ténylegesen ki is ajánl több felhasználói interfészt. Ebben a példában egy memória-jellegű katalógus-áramkörről van szó, ezért az interfészadapter felhasználói interfészét a dual-port on-chip RAM modulokhoz készített, szinkron kézfogásos interfészű IP-magéval *(singleclock_dualport_ram_hswrp)* megegyező módon szeretnénk kialakítani, megkönnyítve a toplevel-integrátor számára a két memóriaszerű részegység egymással való kicserélését. A „többfelhasználós” jelleget ebben az esetben értelmezhetjük úgy, hogy a beírás- és a kiolvasásszolgáltatást egy-egy külön felhasználóhoz rendeljük, így a hozzáférés-vezérlő szerepe úgy is megfogalmazható, hogy ez az almodul teszi lehetővé az egyébként egyszerre vagy csak írás vagy csak olvasás művelet végrehajtására képes katalógus-áramkör dual-port használatát. A dual-port jelleg persze csak látszólagos, a hozzáférés-vezérlő ui. az egyszerre érkező írás és olvasás kéréseket egymás után szolgálja ki egy egyszerű prioritáskezelési séma szerint (lásd később).[124] A hozzáférés-vezérlő RTL-modelljének vázlatát a 4-95. ábra szemlélteti.

[124] A dual-port memóriaszerű funkcionális egységeknek ez a felhasználási módja elég elterjedt. Ezzel a megoldással, amelyben tehát az egyik felhasználó csak írja, a másik pedig csak olvassa az adott memóriamodul tartalmát, lényegében egy megosztott memória jön létre, amelyen keresztül a beírásért felelős felhasználó információt oszthat meg a kiolvasásért felelős felhasználóval. A kétirányú kommunikáció egy másik, fordított szereposztású memóriapéldány létrehozásával ugyancsak könnyen megvalósítható. A többfelhasználós jelleg ugyanakkor másképpen is kialakítható; ha a dual-port memória-interfész egészét egyetlen interfésznek tekintjük, amelyen keresztül írni és olvasni is lehet a memóriát, akkor több felhasználó esetén az interfészjeleket többszörözni kell. Ez esetben a hozzáférés-vezérlőnek nemcsak az írás- és olvasáskérések, hanem az egyes felhasználók között is prioritási sort kell felállítania és kezelnie, hiszen maga a katalógus-áramkör továbbra is csak egy írás vagy olvasás műveletet képes egyidőben kiszolgálni.

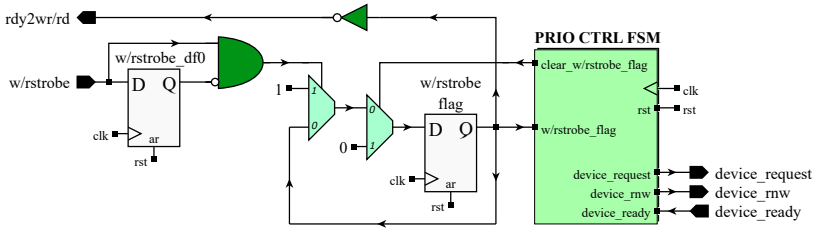
Verilog

```
1  // STROBE RISING EDGE DETECTORS
2  always @ ( posedge clk, posedge rst )
3  begin
4    if ( rst == 1 ) begin
5      wstrobe_df0 <= 0; rstrobe_df0 <= 0;
6    end else begin
7      wstrobe_df0 <= wstrobe; rstrobe_df0 <= rstrobe;
8    end
9  end
10 assign wstrobe_rising = wstrobe & ~wstrobe_df0;
11 assign rstrobe_rising = rstrobe & ~rstrobe_df0;
12
13 // STROBE FLAGS
14 always @ ( posedge clk, posedge rst )
15 begin
16   if ( rst == 1 ) begin
17     wstrobe_flag <= 0; rstrobe_flag <= 0;
18   end else begin
19     if ( clear_wstrobe_flag == 1 ) wstrobe_flag <= 0;
20     else if ( wstrobe_rising == 1 ) wstrobe_flag <= 1;
21     if ( clear_rstrobe_flag == 1 ) rstrobe_flag <= 0;
22     else if ( rstrobe_rising == 1 ) rstrobe_flag <= 1;
23   end
24 end
25 assign rdy2wr = ~wstrobe_flag;
26 assign rdy2rd = ~rstrobe_flag;
27
```

Verilog (folyt.)

```
28 // PRIO CTRL FSM
29 always @ ( posedge clk, posedge rst )
30 begin
31   if ( rst == 1 ) begin
32     state <= write_performed;
33     clear_wstrobe_flag <= 0;
34     clear_rstrobe_flag <= 0;
35     device_rnw <= 0; device_request <= 0;
36   end else begin
37
38     // defaults...
39     clear_wstrobe_flag <= 0;
40     clear_rstrobe_flag <= 0;
41     device_request <= 0;
42
43     case ( state )
44
45       write_performed: [...]
46
47       [...]
48
49       default:    ;
50
51     endcase
52
53   end
54 end
```

Áramköri struktúra



4-95. ábra. *Az rw_access_controller modul RTL-modellje*

Az RTL kapcsolási sémán látható, hogy az almodul a *strobe* jeleken előállított felfutóélekkel jelzett írás- és olvasáskéréseket egy-egy állapotjelző *(w/rstrobe_flag)* segítségével regisztrálja. Erre azért van szükség, mert a szinkron kézfogásos interfész-specifikáció szabályainak megfelelően a felhasználó megteheti, hogy a *strobe* jeleken csak egy-egy, egyetlen órajel-periódusnyi pozitív impulzust állít elő, ugyanakkor – az on-chip RAM modulokat használó IP-maggal ellentétben – nem garantálható e kérések azonnali kiszolgálása, hiszen egy-egy EEPROM művelet – egyrészt a lassú, soros kommunikáció, másrészt az EEPROM chip saját belső késleltetései miatt – sok-sok órajelciklust igényelhet. A kéréseket regisztráló állapotjelzők egyúttal a felhasználónak szóló *rdy2wr* és *rdy2rd* nyugtázójeleket is előállítják egy-egy inverteren keresztül, jelezve, hogy amíg az állapotjelző magas szintű, addig a modul nem képes újabb kérés fogadására.

Az állapotjelzők értékét egy egyszerű állapotgép figyeli, és az írás- és olvasáskéréseket az eszközspecifikus szolgáltatásvezérlőnek továbbítja (lásd 4-96. ábra).

A korábban elemzettekkel ellentétben a hozzáférés-vezérlő állapotgépének nem egy, hanem két „tétlen” állapota van. Amíg nem érkezik felhasználói kérés, addig az állapotgép a legutóbb kiszolgált kérés típusától függően vagy a *write_performed* vagy a *read_performed* állapotban várakozik. E várakozó állapotok csupán egyetlen apróságban térnek el; ha az órajel felfutóélénél az írás- és az

Verilog

```
1  write_performed: begin
2    if ( rstrobe_flag == 1 ) begin
3      device_rnw <= 1;
4      device_request <= 1;
5      state <= wait_1;
6    end else if ( wstrobe_flag == 1 ) begin
7      device_rnw <= 0;
8      device_request <= 1;
9      state <= wait_1;
10   end
11 end
12
13 read_performed: begin
14   if ( wstrobe_flag == 1 ) begin
15     device_rnw <= 0;
16     device_request <= 1;
17     state <= wait_1;
18   end else if ( rstrobe_flag == 1 ) begin
19     device_rnw <= 1;
20     device_request <= 1;
21     state <= wait_1;
22   end
23 end
24
```

Verilog (folyt.)

```
25 wait_1: state <= wait_2;
26
27 wait_2:
28   if ( device_ready == 1 ) begin
29     if ( device_rnw == 1 ) clear_rstrobe_flag <= 1;
30     else clear_wstrobe_flag <= 1;
31     state <= wait_3;
32   end
33
34 wait_3:
35   if ( device_rnw == 1 ) state <= read_performed;
36   else state <= write_performed;
```

4-96. ábra. *Az rw_access_controller modul állapotgépének Verilog-leírása*

olvasáskérés beérkezésére utaló mindkét állapotjelző magas szintű, akkor az állapotgép eltérő prioritás szerint reagál (lásd 4-96. ábra: 2./6. sor vs. 14./18. sor). Ha a legutóbb kiszolgált kérés típusa írás volt, következésképp az állapotgép a *write_performed* állapotban „tétlenkedik”, akkor az egyidőben érkező kérések közül az olvasást részesíti előnyben, ha azonban legutóbb egy olvasást szolgált ki, ami által a művelet végén a *read_performed* állapotba került, akkor az egyidejű kérések közül az írást szolgálja ki előbb. Ez a megoldás tulajdonképpen egy kétszereplős, egyenrangú körbeforgó (round-robin) prioritási séma, amely garantálja, hogy egyik felhasználói interfészt sem éheztetjük ki, függetlenül a különböző kéréstípusok gyakoriságától.

A felhasználói kérések továbbítása az eszközspecifikus szolgáltatásvezérlő felé egy kiszolgáló jellegű, ugyancsak szinkron kézfogásos interfészen történik, amely mindkét kéréstípust ki tudja fejezni; ha a *device_rnw* jel értéke a *device_request* jel felfutásakor alacsony, akkor a szolgáltatásvezérlő – a közvetlenül rendelkezésére álló cím- és adatbuszok birtokában – egy írás, ellenkező esetben egy olvasás műveletet kezdeményez az EEPROM áramkör SPI-interfészén (lásd később). Amikor a megfelelő SPI-tranzakciót a szolgáltatásvezérlő befejezte, akkor a *device_ready* jelet magas szintre állítja, aminek hatására a hozzáférés-vezérlő állapotgép törli a legutóbbi kérést regisztráló állapotjelzőt (4-96. ábra: 29. vagy 30. sor), majd az épp kiszolgált kérés típusától függően a *write_performed* vagy a *read_performed* állapotba lép, várva a következő felhasználói kérést. Mivel az állapotjelzők egy-egy inverteren keresztül a felhasználói interfész *rdy2wr* és *rdy2rd* jeleinek értékét is meghatározzák, ezért a művelet befejezéséről egyúttal a kérést küldő felhasználó is értesül.

Az interfészadapter központi eleme a szolgáltatásvezérlő, amelynek feladata tehát az, hogy a hozzáférés-vezérlőtől érkező, a *device_rnw* jel által meghatározott típusú kéréseket „lefordítsa” az EEPROM katalógus-áramkör SPI-interfészének nyelvére, amihez az SPI master modul segítségét veszi igénybe. RTL megvalósítását tekintve a szolgáltatásvezérlő a hozzáférés-vezérlőhöz hasonlóan egy one-shot típusú állapotgép, amelynek vázlatát a 4-97. ábra szemlélteti, a változatosság kedvéért ezúttal VHDL nyelven megfogalmazva.

VHDL

```
L_SRV_CTRL_FSM: process ( clk, rst ) begin
  if ( rst = '1' ) then
    state <= idle; return_state <= write_2; device_ready <= '1'; spi_client_csn <= '1';
    spi_data_2_send <= (others => '0'); spi_request <= '0'; delay_cnt <= 0; dout <= X"00";
  elsif ( rising_edge(clk) ) then

    case state is

      when idle => device_ready <= '1';
                   if ( device_request = '1' ) then
                     device_ready <= '0';
                     if ( device_rnw = '0' ) then state <= write_1; -- WRITE a single byte
                     else state <= read_1;                          -- READ a single byte
                     end if;
                   end if;

      [...]

      when others => null;

    end case;

  end if;
end process;
```

4-97. ábra. *A 25LC1024 EEPROM IFA szolgáltatásvezérlő állapotgépének vázlata*

A 4-91. és 4-92. ábrák időzítési diagramjain látható, hogy az EEPROM-mal való kommunikáció nyolcbites SPI-tranzakciókon keresztül valósul meg. A szolgáltatásvezérlő állapotgépben az írást és az olvasást is egy-egy állapotsorozat képviseli majd, amelyek tehát ilyen nyolcbites tranzakciók végrehajtására utasítják az SPI master modult, az *spi_data_2_send* vektor beállításával határozva meg azt, hogy az egyes tranzakciókban mi legyen az EEPROM felé kiküldött adat. E tranzakciók levezénylése – az SPI master modul szinkron kézfogásos interfészéből adódóan – több vezérlési állapotot igényel, amelyeket érdemes az állapotgép bármely állapotából „hívható", szubrutinszerű állapotsorozatként megvalósítani (lásd 4-98. ábra).[125]

VHDL

```
when spi_master_request_1 => if ( delay_cnt < tcs_cnt_max + 1 ) then delay_cnt <= delay_cnt + 1;
                             else
                               delay_cnt <= 0;
                               spi_request <= '1';
                               state <= spi_master_request_2;
                             end if;

when spi_master_request_2 => spi_request <= '0'; state <= spi_master_request_3;

when spi_master_request_3 => if ( spi_ready = '1' ) then
                               if ( delay_cnt < tcs_cnt_max + 1 ) then delay_cnt <= delay_cnt + 1;
                               else
                                 delay_cnt <= 0;
                                 state <= return_state;
                               end if;
                             end if;
```

4-98. ábra. *Szubrutinszerű állapotsorozat az SPI master IP-val való kommunikáció megvalósítására*

Ez az állapotsorozat az SPI master modullal való kommunikáción túl egyúttal az EEPROM T_{CSS} és T_{CSH} időzítési paramétereinek minimumára vonatkozó peremfeltételek teljesülését is biztosítja azáltal, hogy az *spi_request* jel felhúzása, tehát az SPI-tranzakció tényleges kezdeményezése előtt és annak végrehajtása után némi várakozást iktat be (lásd 4-98. ábra: 1. és 11. sor). Így, ha

[125] A szubrutinszerű állapotsorozatokról bővebben: 2.3. pont.

a szolgáltatásvezérlő bármely állapota közvetlenül az SPI master modul *client_csn* bemenetének – amely közvetlenül meghajtja a külső SPI-interfész *CSn* jelét (lásd 4-50. ábra) – lehúzása után igényli az SPI-tranzakciót, vagy az SPI-tranzakció végrehajtása után azonnal magas szintre állítja a *client_csn* jelet, akkor is garantált a megfelelő késleltetések megléte.[126] E késleltetések minimális értéke a 4-6. táblázatban foglaltak szerint 50 ns, amelynek megfelelő beállítását a *tcs_cnt_max* paraméter létrehozásával kénytelenek vagyunk a felhasználóra bízni, hiszen a példányosított interfészadapter órajelének frekvenciáját csak ő ismerheti.[127] A várakozás beiktatása után az *spi_request* jelen előállított pozitív impulzussal kezdeményezzük az SPI-tranzakciót (4-98. ábra: 4. és 8. sor), amelyhez a kiküldendő adatot a szubrutinszerű állapotsorozatot „hívó" vezérlési állapot korábban az *spi_data_2_send* buszra helyezte (lásd később). Az SPI-tranzakció – és az azt követő várakozás – után visszatérünk a hívó által a műveletvégzés folytatásának helyeként kijelölt állapotba (*return_state*, 4-98. ábra: 14. sor), ahol az SPI slave – ezúttal az EEPROM – által visszaküldött adat az *spi_data_received* vektorból olvasható ki.

Az SPI-tranzakciókat „önállóan intéző" szubrutinszerű állapotsorozat segítségével a teljes bájtírás és -olvasás műveleteket megvalósító vezérlésiállapot-sorozatok már nagyon szemléletesen konstruálhatók (lásd 4-99. és 4-100. ábra).

VHDL

```
1  when write_1 => -- enable write
2    spi_client_csn <= '0';
3    spi_data_2_send <= X"06";
4    return_state <= write_2;
5    state <= spi_master_request_1;
6
7  when write_2 =>
8    spi_client_csn <= '1';
9    if ( delay_cnt < tcs_cnt_max + 1 ) then
10     delay_cnt <= delay_cnt + 1;
11   else
12     delay_cnt <= 0;
13
14     -- send write instruction
15     spi_client_csn <= '0';
16     spi_data_2_send <= X"02";
17     return_state <= write_3;
18     state <= spi_master_request_1;
19   end if;
20
21 when write_3 => -- send address byte #2
22   spi_data_2_send <= "0000000" & waddr(16);
23   return_state <= write_4;
24   state <= spi_master_request_1;
25
```

VHDL (folyt.)

```
26 when write_4 => -- send address byte #1
27   spi_data_2_send <= waddr(15 downto 8);
28   return_state <= write_5;
29   state <= spi_master_request_1;
30
31 when write_5 => -- send address byte #0
32   spi_data_2_send <= waddr(7 downto 0);
33   return_state <= write_6;
34   state <= spi_master_request_1;
35
36 when write_6 => -- send data byte
37   spi_data_2_send <= din;
38   return_state <= wait_write_cycle;
39   state <= spi_master_request_1;
40
41 when wait_write_cycle =>
42   spi_client_csn <= '1';
43   if ( delay_cnt < write_cycle_time_cnt_max ) then
44     delay_cnt <= delay_cnt + 1;
45   else
46     delay_cnt <= 0;
47     state <= idle;
48   end if;
```

4-99. ábra. *A 25LC1024 EEPROM-ba egy bájtot beíró vezérlési állapotok*

[126] Figyeljük meg, hogy itt valójában nem az interfészadapter felhasználója számára, sokkal inkább saját magunk, mint az interfészadapter tervezője számára hoztunk létre biztosítékokat az időzítési feltételek betartására vonatkozóan. A *client_csn* jelhez a felhasználó ui. nem fér hozzá, annak kezelése a szolgáltatásvezérlő állapotgép feladata. A fenti megoldás tehát azt teszi lehetővé, hogy a $T_{CSS/H,min}$ késleltetések megfelelő beiktatásával az állapotgép tervezése során csak egy helyen kelljen foglalkoznunk.

[127] Ha a felhasználó megfogadja a fogyasztásoptimalizálással kapcsolatos tanácsunkat, mely szerint minden IP-magot csak olyan magas frekvenciájú órajellel érdemes hajtani, amelyet a feladata megkíván, akkor a *tcs_cnt_max* paraméter értéke nem lehet túl nagy. Ha például az EEPROM *SCK* jelét 1 MHz frekvenciájúra szeretnénk beállítani, akkor az SPI master modul – egyúttal a teljes interfészadapter – órajel-frekvenciájának a lehető legkisebb fogyasztást eredményező elvi minimuma 2 MHz, a periódusideje tehát 500 ns. Ez tízszerese az EEPROM $T_{CSS/H,min}$ paramétereinek, vagyis a várakozóciklusok beiktatására nincs is szükség, a *tcs_cnt_max* paraméter akár nullára is állítható. Ettől függetlenül a paramétert nem spórolhatjuk meg, hiszen a felhasználó nincs rákényszerítve az interfészadapter órajelének megfelelő leskálázására; 100 MHz-es órajelforrás esetén az 50 ns $T_{CSS/H,min}$ késleltetések már öt órajelciklust igényelnek!

Beíráskor az első lépés az írásművelet általános engedélyezése, vagyis a 0x06 parancskód kiküldése. Ez a szolgáltatásvezérlő állapotgépében az SPI-tranzakciók végrehajtásának kiszervezése után már egyszerű; alacsony szintre húzzuk az *spi_client_csn* jelet (4-99. ábra: 2. sor), beállítjuk az EEPROM felé küldendő adatot tároló *spi_data_2_send* vektort (4-99. ábra: 3. sor), a *return_state* állapotba beírjuk annak az állapotnak a kódját, amellyel az SPI-tranzakció után folytatni szeretnénk a műveletvégzést (4-99. ábra: 4. sor), majd az SPI-tranzakciókat kezelő állapotsorozat első állapotába irányítjuk a végrehajtást (4-99. ábra: 5. sor). Az SPI-tranzakció végeztével az *spi_client_csn* jelet újra magas szintűre állítjuk (4-99. ábra: 8. sor), lezárva a műveletet. Figyelembe véve az EEPROM T_{CSD} paraméterének minimumára vonatkozó előírást, a következő SPI-tranzakció megkezdése előtt ugyancsak némi várakozás beiktatására van szükség, amit helyben kell megvalósítanunk, ui. az *spi_master_request_X* állapotok erről a késleltetésről nem gondoskodnak (4-99. ábra: 9–10. sor). E várakozást követően a beírás művelete a fentiekkel csaknem azonos módon kivitelezett további négy SPI-tranzakciót igényel, a különbség csupán az, hogy az írás műveletéhez tartozó SPI-tranzakciók közben az *spi_client_csn* jelet végig alacsony szinten tartjuk. A 0x02 parancs kiküldésével jelezzük az EEPROM-nak, hogy írás következik (4-99. ábra: 15–18. sor), majd három SPI-tranzakcióban kiküldjük az összesen 24 bitnyi címet (4-99. ábra: 21–34. sor), amit a beírandó adat követ (4-99. ábra: 36–39. sor). Az *spi_client_csn* jel felhúzása után ismét egy várakozás beiktatására van szükség, mivel az EEPROM-nak 6 ms-ra van szüksége ahhoz, hogy a kapott adatot fizikailag is eltárolja (4-99. ábra: *wait_write_cycle* állapot).[128] Az interfészadapter órajel-frekvenciájának ismeretében a felhasználó a 6 ms késleltetést a *write_cycle_time_cnt_max* paraméterrel állíthatja be.

VHDL

```
1  when read_1 => -- send "read byte" instruction
2    spi_client_csn <= '0';
3    spi_data_2_send <= X"03";
4    return_state <= read_2;
5    state <= spi_master_request_1;
6
7  when read_2 => -- send address byte #2
8    spi_data_2_send <= "0000000" & raddr(16);
9    return_state <= read_3;
10   state <= spi_master_request_1;
11
12 when read_3 => -- send address byte #1
13   spi_data_2_send <= raddr(15 downto 8);
14   return_state <= read_4;
15   state <= spi_master_request_1;
16
```

VHDL (folyt.)

```
17 when read_4 => -- send address byte #0
18   spi_data_2_send <= raddr(7 downto 0);
19   return_state <= read_5;
20   state <= spi_master_request_1;
21
22 when read_5 => -- send a dummy byte
23   spi_data_2_send <= X"00";
24   return_state <= read_6;
25   state <= spi_master_request_1;
26
27 when read_6 =>
28   spi_client_csn <= '1';
29   dout <= spi_data_received;
30   state <= idle;
```

4-100. ábra. *A 25LC1024 EEPROM egy bejegyzésének kiolvasását végző vezérlési állapotok*

A bájtolvasás megvalósítása az írásnál némileg egyszerűbb; egyrészt nincs szükség az általános írásengedélyező parancskód kiküldésére, másrészt a folyamat végén nem kell kivárni az eszköz beírási késleltetését sem. Így az olvasás tulajdonképpen a 4-92. ábrán látható, négy egymást követő SPI-tranzakcióra egyszerűsödik. További eltérés az írás végrehajtásához képest, hogy az utolsó

[128] Amíg ez nem történik meg, addig nem tekinthetjük lezártnak a beírás folyamatát akkor sem, ha az ehhez szükséges SPI-tranzakciók már véget értek. Ha a 6 ms várakozás beiktatása nélkül megadnánk a lehetőséget a felhasználónak újabb kérés kezdeményezésére, akkor előfordulhatna, hogy épp az imént felülírt memóriacímen hajtana végre egy olvasást, ahol a korábban kezdeményezett írás eredményét még nem láthatná (ne feledjük, hogy az írásművelethez tartozó SPI-tranzakciók alig több mint 40 µs alatt lezajlanak, még az aránylag lassú, 1 MHz-es *SCK* frekvencia mellett is). Ez a felhasználó szempontjából nyilván megengedhetetlen.

SPI-tranzakcióban az interfészadapter nem küld hasznos adatot az EEPROM-nak (4-100. ábra: 23. sor), hiszen ennek a tranzakciónak a célja a megcímzett adatrekesz tartalmának kinyerése az EEPROM-ból. A kiolvasott bájtot az utolsó SPI-tranzakció végeztével az SPI master modul *data_received* kimenetéről olvashatjuk le (4-100. ábra: 29. sor).

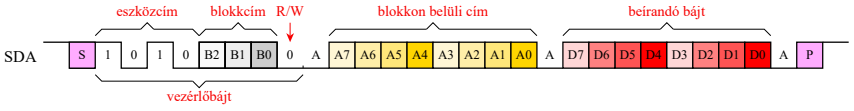
Akár írás-, akár olvasásműveletről legyen szó, a szolgáltatásvezérlő az *idle* állapotba visszatérve magas szintre állítja a hozzáférés-vezérlőnek szóló *device_ready* jelzést (4-97. ábra: 9. sor). Ennek hatására a hozzáférés-vezérlő törli az éppen végrehajtott műveletet elindító, egyúttal az interfészadapter *rdy2wr/rd* kimenetét egy inverteren keresztül meghajtó állapotjelzőt (4-96. ábra: 29. vagy 30. sor), így zárva le a felhasználói interfész kézfogásos tranzakcióját.

4.4.2. I²C EEPROM

Áttanulmányozásra javasolt alfejezetek:

4.3.3. *A soros kommunikáció alapáramkörei – I²C master*

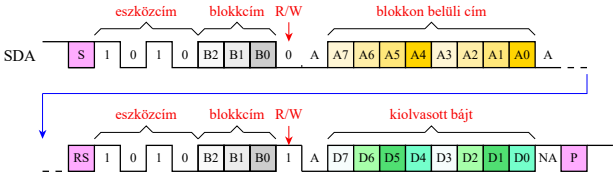
Az ugyancsak Microchip gyártmányú 24LC16B típusú katalógus-áramkör szintén egy EEPROM chipet rejt, ez a kivitel azonban kisebb tárkapacitású (16 kbit = 8 blokk × 256 bájt = 2048 bájt), és I²C-interfészen keresztül írható és olvasható. Az egyetlen bájt írásához és olvasásához tartozó I²C-tranzakciók időzítését rendre a 4-101. és a 4-102. ábrák szemléltetik.



4-101. ábra. *A 24LC16B EEPROM chip írásához szükséges I²C-tranzakció időzítése*
*S: **s**tartfeltétel, P: sto**p**feltétel, A: **a**cknowledge*

Az írásművelet egy startfeltétel generálásával indul, ami után egy ún. vezérlőbájtot küldünk el az eszköznek. Ennek felső négy bitje az eszköz mint I²C slave egység címét rejti (1010_2), 1–3. helyiértékű bitjei a megcímezni kívánt adatblokkot jelölik ki (B2, B1, B0), az utolsó bitje pedig nulla, jelezve, hogy a master egység egy I²C-írást kezdeményez. Miután az EEPROM – azáltal, hogy a következő *SCL* ciklusban nullában tartja az *SDA* vonalat – nyugtázza a vezérlőbájt vételét, a következő nyolcbites tranzakcióban a memóriacím, majd – ismét egy nyugtázását követően – a beírandó adat átvitelére és nyugtázására kerül sor. A beíráshoz tartozó I²C-busztranzakciók sorát egy stopfeltétel zárja. Ekkor az EEPROM chip megkezdi a belső bájtírás művelet végrehajtását, ami – az SPI-interfészű kivitelnél látottakhoz hasonlóan – sokkal lassabban megy végbe, mint maguk az I²C-tranzakciók; a megcímzett rekesz tartalma csak mintegy 5 ms késleltetés után íródik be, míg a 4-101. ábrán látható I²C-buszműveletek 100 kHz-es *SCK* frekvencia mellett 290 µs, 400 kHz esetén pedig 72,5 µs alatt lezajlanak. Ezt a sajátosságot az interfészadapter tervezésekor ugyancsak figyelembe kell vennünk.

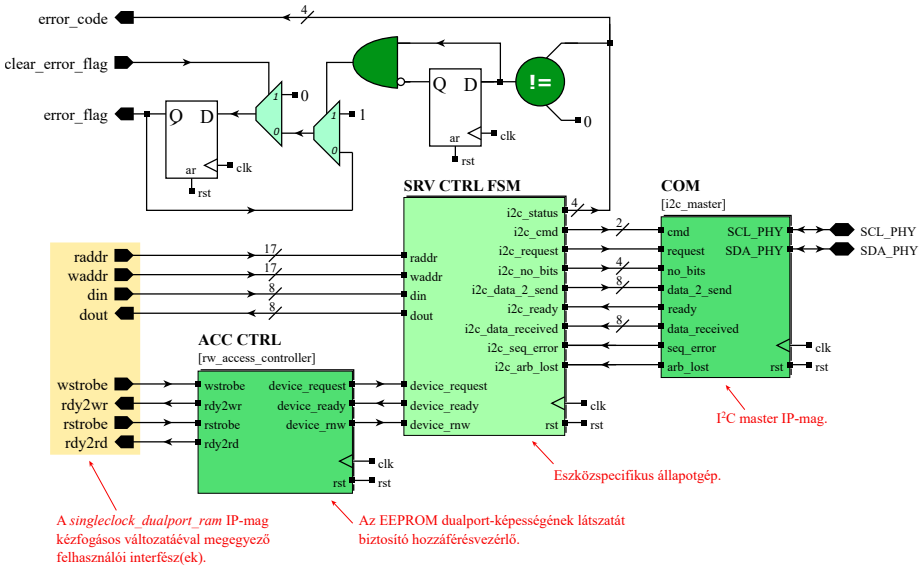
A bájtolvasáshoz tartozó időzítési diagram némileg összetettebb. A startfeltétel és a vezérlőbájt kiküldése az írásnál látottakkal megegyező módon zajlik le. Figyeljük meg, hogy a vezérlőbájt utolsó bitje ezúttal is nulla, ami írásműveletre utal, annak ellenére, hogy épp kiolvasni szeretnénk egy bájtot az EEPROM-ból. Az „írás” kifejezés azonban ebben a kontextusban nem az EEPROM



4-102. ábra. *A 24LC16B EEPROM chip olvasásához szükséges I²C-tranzakciók időzítése*
*S: **s**tartfeltétel, RS (repeated **s**tart): megismételt startfeltétel, P: sto**p**feltétel, (N)A: (**n**ot) **a**cknowledge*

adatrekeszére, hanem annak egy belső regiszterére vonatkozik; a következő nyolc bit átvitele ui. az EEPROM kiolvasandó rekeszének blokkon belüli címe lesz, amelyet az EEPROM chipen belül egy külön regiszter tárol. Ezt a regisztert egy I²C-írás tranzakcióval felül kell írnunk a rekesz tényleges kiolvasása előtt. A vezérlőbájt nyugtázását tehát az olvasási cím kiküldése és nyugtázása követi. Ezután – annak érdekében, hogy több master egység jelenléte esetén se „veszítsük el" az I²C-buszt annak arbitrációs mechanizmusa miatt az olvasás befejezése előtt – egy megismételt startfeltételt állítunk elő, amelyet egy újabb vezérlőbájt követ. E vezérlőbájtban a címbitek – az eszköz I²C-címe: 1010_2 és a kiválasztott blokk címe: B2, B1, B0 – megegyeznek az előző vezérlőbájtban elküldött értékekkel, az utolsó bit azonban egyes, ami olvasásra utal. Ez az egyes bit jelzi, hogy a vezérlőbájt nyugtázását követő nyolcbites átvitelben az EEPROM hajtja majd meg az *SDA* vonalat, bitenként visszaküldve a megcímzett adatrekesz tartalmát. Az adatbájt kiolvasását a master úgy zárja le, hogy a kilencedik *SCL* ciklusban az *SDA* vonal magas szinten tartásával szándékosan nem nyugtázza az előző tranzakciót, majd egy stopfeltételt generál, amivel a bájtolvasás művelete master és slave szempontból egyaránt lezártnak tekinthető.

Az I²C-interfésszel szerelt EEPROM katalógus-áramkör interfészadapterének szerkezetét a 4-103. ábra mutatja be.



4-103. ábra. *A 24LC16B EEPROM interfészadapterének szerkezete*

A modul alapvető szerkezete az SPI-os kiviteléhez hasonló, sőt, a hozzáférés-vezérlő teljes egészében felhasználható ebben a verzióban is, a hasonlóságok sora azonban itt véget is ér. A soros-párhuzamos/párhuzamos-soros átalakításért felelős kommunikációs almodul szerepét ezúttal a 4.3.3. pontban bemutatott I²C master IP-mag tölti be. Ez az almodul – bár a szokásos szinkron kézfogásos interfészen keresztül használható – egészen más szolgáltatáskészlettel üzemeltetendő, mint SPI-os megfelelője, a szolgáltatásvezérlő állapotgépet tehát egyedileg kell megterveznünk. Az SPI master modullal ellentétben az I²C master modul felhasználói interfészén kiajánl két, az I²C-buszon potenciálisan előforduló hibára utaló jelet (*seq_error*[129] és *arb_lost*), amiket a szolgáltatásvezérlőnek valami módon figyelembe kell vennie. Az itt bemutatott példában a hibakezelésnek egy nagyon egyszerű, de könnyen továbbfejleszthető változatát valósítjuk meg. A szolgáltatásvezérlőben létrehozunk egy négybites státuszindikátor vektort *(i2c_status)*, amely tizenöt különböző hibaesetet képes megkülönböztetni. Ebben a konkrét példában most csak három kombináció lesz érvényes: a 0x0 kód jelzi a hibátlan I²C-tranzakciókat, a 0x1 kombináció azt mutatja, hogy az I²C master egység szekvenciahibát vagy busz-arbitrációs hibát észlelt, a 0x2 kód pedig arra utal, hogy a katalógus-áramkör valamelyik I²C-tranzakciót várakozásunkkal ellentétben nem nyugtázta. A 4-103. ábra kapcsolási rajzának felső részén lévő logika a szolgáltatásvezérlő *i2c_status* kimenetét figyeli, és ha 0x0-tól eltérő értéket lát a működés során bármikor, akkor magas szintre állítja az *error_flag* nevű állapotjelzőt, amit – a hibajelzés és egyúttal a kért írás vagy olvasás művelet

VHDL

```
1  L_SRV_CTRL_FSM: process ( clk, rst ) begin
2  | if ( rst = '1' ) then
3  | | state <= idle; return_state <= write_2; device_ready <= '1';
4  | | i2c_cmd <= "00"; i2c_no_bits <= 1; i2c_data_2_send <= (others => '0'); i2c_request <= '0';
5  | | i2c_status <= (others => '0'); delay_cnt <= 0; dout <= (others => '0');
6  | elsif ( rising_edge(clk) ) then
7  | |
8  | | case state is
9  | | |
10 | | | when idle => device_ready <= '1';
11 | | | |            if ( device_request = '1' ) then
12 | | | |              device_ready <= '0';
13 | | | |              if ( device_rnw = '0' ) then state <= write_1; -- WRITE a single byte
14 | | | |              else state <= read_1;                          -- READ a single byte
15 | | | |              end if;
16 | | | |            end if;
17 | | | |
18 | | | [...]
19 | | | |
20 | | | when others => null;
21 | | | |
22 | | end case;
23 | |
24 | end if;
25 end process;
```

4-104. ábra. *A 24LC16B EEPROM IFA szolgáltatásvezérlő állapotgépének vázlata*

[129] Az I²C master modul *seq_error* kimenete akkor jelez, ha a felhasználó stopfeltétel-generálásra vagy adatátviteli tranzakcióra utasítja anélkül, hogy előtte startfeltétel-generálást kért volna. Az I²C master modul felhasználója ebben az esetben a szolgáltatásvezérlő állapotgép, vagyis az interfészadapter felhasználójának esélye sincs ilyen hibát kiváltani az I²C master modulban, hiszen ahhoz nincs közvetlen hozzáférése. Ilyen értelemben tehát ez a hibakimenet feleslegesnek tűnhet. Ugyanakkor magának a szolgáltatásvezérlőnek a kifejlesztése során követhetünk el hibákat, amelyekre a *seq_error* hibajelzés nyomonkövetésével könnyebben fény derülhet.

meghiúsulásának nyugtázásaként – a felhasználó a *clear_error_flag* bemenettel törölhet.[130] A hiba okáról a felhasználó az *error_code* kimenet értékének leolvasásával tájékozódhat.

A szolgáltatásvezérlő ebben a verzióban is egy VHDL nyelven megfogalmazott, one-shot típusú állapotgép, amelynek vázlatát a 4-104. ábra mutatja be.

Az I²C-kommunikációs modul szinkron kézfogásos interfésze miatt ezúttal is több vezérlési állapotra van szükség ahhoz, hogy egy-egy I²C-tranzakciót végrehajtsunk. A szolgáltatásvezérlő az írás és olvasás műveletek során sok ilyen tranzakciót igényel majd, ezért az I²C master modullal való kommunikációt végző állapotokat – az SPI-os példánál látottakhoz hasonlóan – egy szubrutinszerű állapotsorozatba szervezzük ki (lásd 4-105. ábra).[131] Ezt a megoldást az összes I²C- és SPI-interfészű interfészadapternél egységesen alkalmazzuk.

VHDL

```
1  when i2c_master_request_1 => i2c_request <= '1'; state <= i2c_master_request_2;
2
3  when i2c_master_request_2 => i2c_request <= '0'; state <= i2c_master_request_3;
4
5  when i2c_master_request_3 => if ( i2c_ready = '1' ) then
6                               if ( i2c_seq_error = '1' or i2c_arb_lost = '1' ) then
7                                 i2c_status <= X"1"; state <= idle;
8                               else state <= return_state;
9                               end if;
10                            end if;
```

4-105. ábra. *Szubrutinszerű állapotsorozat az I²C master IP-val való kommunikáció megvalósítására*

A szubrutinszerű állapotsorozatba való belépéskor az I²C master modul által végrehajtandó szolgáltatást (start- vagy stopfeltétel-generálás vagy adatküldés/fogadás) az *i2c_cmd* vektor beállításával határozzuk meg, illetve adatküldés vagy -fogadás esetén az átviendő biteket és azok számát is itt definiáljuk az *i2c_data_2_send* és az *i2c_no_bits* vektorok segítségével (lásd később). A szubrutinszerű állapotsorozatnak így annyi feladata marad, hogy előállítsa a kommunikációs modul *request* bemenetén a működést elindító pozitív impulzust (4-105. ábra: 1-3. sor), a *ready* jel felfutásakor – I²C-hiba detektálása esetén – beállítsa az *i2c_status* vektor értékét és az *idle* állapotba visszatérve zárja a hibás tranzakciót (4-105. ábra: 6-7. sor), vagy – ha nem történt hiba – visszatérjen a *return_state* állapotba (4-105. ábra: 8. sor). Az I²C-tranzakciók kiszervezésével a 4-101. és 4-102. ábrákon definiált memóriaírás és -olvasás műveleteket végrehajtó vezérlési állapotok a 4-106. és 4-107. ábrákon látható módon konstruálhatók.

Ahogy minden I²C-tranzakció, úgy az EEPROM-ba való beírás első lépése is a startfeltétel generálása. Ennek előállítása az I²C-buszjeleken az I²C master modul önálló szolgáltatása, tehát az interfészadapter szolgáltatásvezérlő állapotgépében nincs más dolgunk, mint kijelölni ezt a szolgáltatást az *i2c_cmd* jellel, majd átlépni az I²C master modult vezérlő állapotsorozat első állapotába (4-106. ábra: *write_1* állapot). Ha az I²C-tranzakció rendben lezajlott, akkor a végrehajtás a *write_2* állapotban folytatódik, ahol – a 4-101. ábra időzítési diagramjának megfelelően – megkezdődik

[130] A hibakezelés továbbfejleszthető további hibakódok definiálásával, pl. úgy, hogy az I²C master modul által detektált szekvencia- és arbitrációs hibákat a fentiekkel ellentétben nem „olvasztjuk egybe", vagy úgy, hogy az EEPROM-mal való kommunikáció során végrehajtott, összetett I²C-tranzakciók során bekövetkező hibák között azok bekövetkezésének körülményei szerint is különbséget teszünk. Külön hibakódot rendelhetünk pl. a memóriaolvasás során elküldött vezérlőbájt nyugtázásának hiányához, ami arra utal, hogy az EEPROM – pl. a busz fizikai sérülése miatt – nem észleli az I²C-buszon lezajló eseményeket.

[131] A szubrutinszerű állapotsorozatokról bővebben: 2.3. pont.

VHDL

```
1  when write_1 => -- send start-condition
2    i2c_cmd <= "00";
3    return_state <= write_2;
4    state <= i2c_master_request_1;
5
6  when write_2 => -- send I2C address & nW bit
7    i2c_cmd <= "01";
8    i2c_data_2_send <= "1010" & waddr(10 downto 8) & '0';
9    i2c_no_bits <= 8;
10   return_state <= write_3;
11   state <= i2c_master_request_1;
12
13 when write_3 => -- receive ACK
14   i2c_cmd <= "10";
15   i2c_no_bits <= 1;
16   return_state <= write_4;
17   state <= i2c_master_request_1;
18
19 when write_4 =>
20   if ( i2c_data_received(0) = '0' ) then
21     -- ACK received
22     i2c_cmd <= "01"; -- send byte address
23     i2c_data_2_send <= waddr(7 downto 0);
24     i2c_no_bits <= 8;
25     return_state <= write_5;
26     state <= i2c_master_request_1;
27   else -- no ACK received
28     i2c_status <= X"2";
29     state <= idle;
30   end if;
31
32 when write_5 => -- receive ACK
33   i2c_cmd <= "10";
34   i2c_no_bits <= 1;
35   return_state <= write_6;
36   state <= i2c_master_request_1;
37
```

VHDL (folyt.)

```
38 when write_6 =>
39   if ( i2c_data_received(0) = '0' ) then
40     -- ACK received
41     i2c_cmd <= "01"; -- send data to write
42     i2c_data_2_send <= din;
43     i2c_no_bits <= 8;
44     return_state <= write_7;
45     state <= i2c_master_request_1;
46   else -- no ACK received
47     i2c_status <= X"2";
48     state <= idle;
49   end if;
50
51 when write_7 => -- receive ACK
52   i2c_cmd <= "10";
53   i2c_no_bits <= 1;
54   return_state <= write_8;
55   state <= i2c_master_request_1;
56
57 when write_8 =>
58   if ( i2c_data_received(0) = '0' ) then
59     -- ACK received
60     i2c_cmd <= "11"; -- send stop-condition
61     return_state <= wait_write_cycle;
62     state <= i2c_master_request_1;
63   else -- no ACK received
64     i2c_status <= X"2";
65     state <= idle;
66   end if;
67
68 when wait_write_cycle =>
69   if (
70     delay_cnt < write_cycle_time_cnt_max
71   ) then delay_cnt <= delay_cnt + 1;
72   else
73     delay_cnt <= 0;
74     state <= idle;
75   end if;
```

4-106. ábra. *A 24LC16B EEPROM-ba egy bájtot beíró vezérlési állapotok*

a vezérlőbájt kiküldése. Ez a bájt tartalmazza az EEPROM chip I²C-címét (1010_2), a kiválasztott blokk hárombites címét – amelyet a felhasználótól kapott memóriacím felső három bitjéből képezünk[132] – illetve a tranzakció irányát kijelölő bitet, amely ez esetben nulla (4-106. ábra: 8. sor). A startfeltétel-generálással ellentétben ez egy adatátviteli művelet lesz, amelyhez az I²C master modulnak az átvivendő bitek számát is meg kell adnunk (4-106. ábra: 9. sor). A vezérlőbájt átvitelét annak nyugtázása követi, ami a master egység szempontjából egy egybites I²C-olvasást jelent (4-106. ábra: *write_3* állapot). A nyugtázás akkor sikeres, ha az EEPROM a vezérlőbájt utáni *SCL* ciklusban nullába húzta az *SDA* vonalat, vagyis ha az egybites olvasás eredményeként az I²C master modul *data_received* kimenetén előálló legkisebb helyiértékű bit nulla. Ezt az ellenőrzést minden nyugtázáskor végre kell hajtanunk a következő I²C-tranzakció megkezdése előtt (4-106. ábra: 20., 39. és 58. sor). Ha az eszköz az adott I²C-műveletet nem nyugtázta, akkor az *i2c_status* jel értékét 0x2-re állítva visszatérünk az *idle* állapotba, jelezve, hogy az írásművelet sikertelenül zárult. A vezérlőbájt kiküldése után ugyanilyen módon kerül sor az írási cím alsó nyolc bitjének, majd a beírandó adatnak a kiküldésére is, egy-egy nyolcbites I²C-tranzakció segítségével (4-106. ábra: *write_4* és *write_6* állapot). A beírást egy stopfeltétel előállítása zárja (4-106. ábra *write_8* állapot), ami után az EEPROM-chip beírási késleltetését kiváró állapot következik, ami garantálja, hogy

[132] Ez az a pont, ahol „a felhasználó elől eltakarjuk” azt a tényt, hogy az EEPROM chip 256 bájtos blokkokba szervezve tárolja az adatot. Erre az információra ti. a felhasználónak semmi szüksége, amikor a *singleclock_dualport_ram* moduléval megegyező interfészen egy összefüggő memóriaterületet szeretne látni.

VHDL

```
when read_1 => -- send start-condition
  i2c_cmd <= "00";
  return_state <= read_2;
  state <= i2c_master_request_1;

when read_2 => -- send I2C address & nW bit
  i2c_cmd <= "01";
  i2c_data_2_send <= "1010" &
                     raddr(10 downto 8) & '0';
  i2c_no_bits <= 8;
  return_state <= read_3;
  state <= i2c_master_request_1;

when read_3 => -- receive ACK
  i2c_cmd <= "10";
  i2c_no_bits <= 1;
  return_state <= read_4;
  state <= i2c_master_request_1;

when read_4 =>
  if ( i2c_data_received(0) = '0' ) then
  | -- ACK received
  | i2c_cmd <= "01"; -- send byte address
  | i2c_data_2_send <= raddr(7 downto 0);
  | i2c_no_bits <= 8;
  | return_state <= read_5;
  | state <= i2c_master_request_1;
  else -- no ACK received
    i2c_status <= X"2";
    state <= idle;
end if;

when read_5 => -- receive ACK
  i2c_cmd <= "10";
  i2c_no_bits <= 1;
  return_state <= read_6;
  state <= i2c_master_request_1;

when read_6 =>
  if ( i2c_data_received(0) = '0' ) then
  | -- ACK received
  | i2c_cmd <= "00"; -- send (repeated)
  |                  -- start-condition
  | return_state <= read_7;
  | state <= i2c_master_request_1;
  else -- no ACK received
    i2c_status <= X"2";
    state <= idle;
end if;

```

VHDL (folyt.)

```
when read_7 => -- send I2C address & R bit
  i2c_cmd <= "01";
  i2c_data_2_send <= "1010" &
                     raddr(10 downto 8) & '1';
  i2c_no_bits <= 8;
  return_state <= read_8;
  state <= i2c_master_request_1;

when read_8 => -- receive ACK
  i2c_cmd <= "10";
  i2c_no_bits <= 1;
  return_state <= read_9;
  state <= i2c_master_request_1;

when read_9 =>
  if ( i2c_data_received(0) = '0' ) then
  | -- ACK received
  | i2c_cmd <= "10"; -- get data
  | i2c_no_bits <= 8;
  | return_state <= read_10;
  | state <= i2c_master_request_1;
  else -- no ACK received
    i2c_status <= X"2";
    state <= idle;
  end if;

when read_10 =>
  dout <= i2c_data_received;
  i2c_cmd <= "01"; -- send NACK
  i2c_data_2_send <= X"01";
  i2c_no_bits <= 1;
  return_state <= read_11;
  state <= i2c_master_request_1;

when read_11 => -- send stop-condition
  i2c_cmd <= "11";
  return_state <= idle;
  state <= i2c_master_request_1;
```

4-107. ábra. *A 24LC16B EEPROM egy bejegyzésének kiolvasását végző vezérlési állapotok*

a felhasználó szempontjából a beírás művelete a szinkron kézfogásos interfészen mindaddig ne érjen véget, amíg az adat fizikailag be nem íródott az EEPROM-ba. E beírási késleltetésről tudhatjuk, hogy ennél a konkrét EEPROM áramkörnél 5 ms, de mivel az interfészadapter leendő órajelének frekvenciáját nem ismerjük, ezért a késleltetés pontos értékét – az SPI-os verziónál látottakkal azonos módon – a *write_cycle_time_cnt_max* paraméteren keresztül tesszük beállíthatóvá.

A 4-102. ábrán látható bájtolvasás folyamatát megvalósító vezérlési állapotok (lásd 4-107. ábra) az írásnál látottakhoz képest újdonságot nem jelentenek, ezek részletes vizsgálatát az olvasóra bízzuk.

4.4.3. SDRAM

A dinamikus memória-áramkörök (**D**ynamic **R**andom **A**ccess **M**emory, DRAM) az EEPROM-oknál tipikusan nagyobb tárkapacitású és összetettebb működésű eszközök. A különféle gyártók termékeinek interfésze – legalábbis a legalapvetőbb szolgáltatások tekintetében – hasonló, ezért egy parametrikusan leírt DRAM interfészadapter IP sokféle konkrét chippel kompatibilis lehet. Ebben az alfejezetben egy olyan DRAM IFA-t készítünk, amely az illesztendő katalógus-áramkör tárkapacitása és annak időzítési tulajdonságai alapján is rugalmasan testreszabható. Annak érdekében, hogy a DRAM IFA szolgáltatásai és felhasználói interfésze is teljesen megegyezhessen az EEPROM IFA áramköröknél látottakkal, az itt bemutatott megoldás sem használja ki a DRAM chip minden képességét; egy kézfogásos tranzakcióban csupán egyetlen bejegyzés beírására vagy kiolvasására lesz lehetőség.[133]

A DRAM chipekben az adatok tárolása az EEPROM-okhoz képest bonyolultabb struktúrákban történik. A teljes memória B db bankot tartalmaz, egy bank pedig egy S db sorból (row) és C db oszlopból (column) álló, mátrixszerű elrendezésben tárolja a jellemzően 8 vagy 16 bitből álló bejegyzéseket. Egy bejegyzés címe tehát a bankcím (bank address), a sorcím (row address) és az oszlopcím (column address) hármasából áll össze. Egy bejegyzés minden bitjét egy kisértékű – néhányszor 10 fF-os – kapacitásban (cellában) tárolt töltésmennyiség képviseli; ha a kapacitás tápfeszültségre feltöltött állapotban van, akkor az adott bitet egynek tekintjük, a földpotenciálra kisütött cella pedig a nullát jelképezi.

Nemcsak maga a struktúra, de az egyes bejegyzésekhez való hozzáférés „koreográfiája" is komplikáltabb, mint egy egyszerű EEPROM esetén. Egy bejegyzés írása vagy olvasása során első lépésként aktiválni kell az adott bejegyzést tartalmazó bank megfelelő sorát. Az aktiválás eredményeként a teljes kiválasztott sor összes bejegyzésének bitjei egy-egy ún. érzékelőerősítő belső flip-flopjaiba íródnak át. Mivel ez az „átíródás" fizikailag a töltésmegosztás elvén alapul, ezért a folyamat során az információt hordozó elemi kapacitásokban tárolt töltésmennyiség megváltozik, az aktiválás tehát destruktív, de a korábban az elemi cellákban tárolt információ az érzékelőerősítők flip-flopjaiban továbbra is rendelkezésre áll. Az egy teljes sornyi érzékelőerősítőből az oszlopcím választja ki azt az egyet, amelynek flip-flopjait épp írni vagy olvasni szeretnénk. Beíráskor a kiválasztott érzékelőerősítő flip-flopjainak tartalma a DRAM kétirányú adatbuszára kívülről rákapcsolt értékekkel íródik felül, kiolvasáskor pedig éppen fordítva; a flip-flopok tartalma kiíródik az adatbuszra. A beírás- és kiolvasásműveletek közös zárólépése az érzékelőerősítők flip-flopjaiban tárolt aktuális értékek visszatöltése az elemi kapacitásokba (pre-charge). Ahol a flip-flopban tárolt érték 1, ott az aktivált sor megfelelő elemi kapacitását az érzékelőerősítő tápfeszültségre tölti fel, ahol 0, ott pedig földpotenciálra süti ki. A visszatöltés végeredménye kiolvasás esetén az, hogy az aktivált sor eredeti tartalma áll vissza, beíráskor pedig az, hogy az oszlopcím által kiválasztott cellákba az adatbuszról beolvasott adat, a ki nem választott cellákba pedig azok eredeti tartalma íródik vissza.

[133] A DRAM-ok kifejezetten blokkos adatátvitelre optimalizált eszközök, vagyis igazán hatékony és gyors működést akkor érhetünk el velük, ha az adatmozgatás alapegysége nem egyetlen bejegyzés, hanem ezek sokszorosa. A blokkos adatátvitel hatékonysága leginkább az összetett beágyazott rendszerekben fontos, a hierarchikus (cache) és virtuális tárkezelés miatt. A nagyobb adategységek kezelése magasabb absztrakciós szinteken meginkább jellemző. Egy operációs rendszerben az adatokat fájlokban tároljuk, az ezeken végzett műveletek lebonyolítására pedig absztrakt adatszerkezeteket és azokat kezelő algoritmusokat, ún. fájlrendszereket hozunk létre. A fájlrendszerekben az adatok legkisebb logikai egysége a többezer bájtot tartalmazó „szektor", ezért az ilyen alkalmazásokban az operatív és háttértárak is blokkos adatátvitelre optimalizáltak. A DRAM áramköröket – illékony jellegükből adódóan – operatív tárként használjuk..

Az elemi kapacitások sajátossága, hogy – a flip-flopokkal ellentétben – a tápfeszültség jelenléte esetén sem képesek az információ hosszú távú megőrzésére, mivel a bennük tárolt töltésmennyiség az őket felépítő félvezető struktúrák (pn-átmenetek) szivárgó áramai miatt idővel lecsökken, ami végső soron az információ elvesztéséhez vezet. Ennek megakadályozása érdekében a DRAM-okban tárolt információt időnként frissíteni kell, ami lényegében az összes bejegyzés periodikus kiolvasását és visszaírását jelenti (innen az elnevezés: dinamikus RAM). A maximális frissítési periódus jellemzően néhány µs időtartamú.

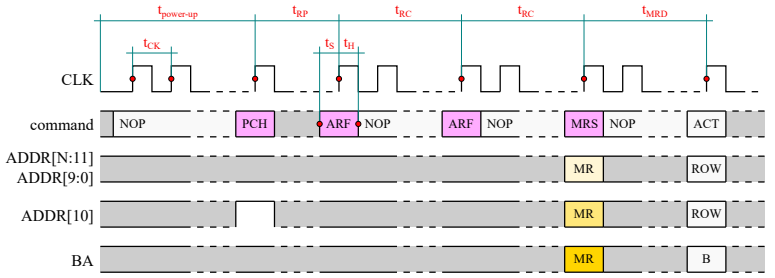
A beírás és kiolvasás műveletsorának végrehajtására a DRAM áramkörök egy néhány vezérlőbemenettel kiegészített párhuzamos interfészt bocsátanak a felhasználó rendelkezésére. Az itt bemutatott példában ez az interfész szinkron működésű, vagyis a vezérlő- és adatbemenetek mintavételezésére az ugyancsak a felhasználó által generált órajel felfutóéleinél kerül sor. Az ilyen interfésszel szerelt eszközöket szinkron DRAM-oknak (**S**ynchronous DRAM, SDRAM) nevezzük. A 4-7. táblázat az interfészjelek nevét és szerepét sorolja fel.

<table>
<tr><th colspan="2">Interfészjel neve</th><th>Interfészjel funkciója</th></tr>
<tr><td>CKE</td><td>Clock Enable</td><td>Magas-aktív órajel-engedélyező bemenet. Alacsony értéke mellett az eszköz nem veszi figyelembe az órajelbemenet változásait. Az IFA áramkörünkben ezt a bemenetet fixen magas szinten tartjuk, az órajel letiltásának lehetőségét tehát nem használjuk ki.</td></tr>
<tr><td>CLK</td><td>Clock</td><td>Órajelbemenet.</td></tr>
<tr><td>CSn</td><td>Chip Select</td><td rowspan="4">Alacsony-aktív vezérlőbemenetek, ezek együttese képezi az eszköznek küldött parancsszót (command). A tizenhat lehetséges parancs közül az itt bemutatott példában csak nyolcat valósítunk meg:
<table>
<tr><th>CSn</th><th>RASn</th><th>CASn</th><th>WEn</th><th>Parancs neve</th></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td>device deselect</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>1</td><td>no operation</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>1</td><td>read & pre-charge</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td>write & pre-charge</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>1</td><td>bank activate</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>0</td><td>pre-charge</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>1</td><td>auto-refresh</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>mode register set</td></tr>
</table></td></tr>
<tr><td>RASn</td><td>Row Address Strobe</td></tr>
<tr><td>CASn</td><td>Column Address Strobe</td></tr>
<tr><td>WEn</td><td>Write Enable</td></tr>
<tr><td>BA</td><td>Bank Address</td><td>Bankcím.</td></tr>
<tr><td>ADDR</td><td>Address</td><td>A kiválasztott bejegyzés bankon belüli címét jelölő bemenetek. A teljes busz $\log_2(S)$ db bemenetéből az alsó $\log_2(C)$ db szerepe kettős; az adatátvitel első fázisában a teljes címbusz a kiválasztott bejegyzést tartalmazó sor azonosítóját képviseli, a sor aktiválása után azonban az alsó $\log_2(C)$ db bemenet az oszlopcím szerepét tölti be, vagyis kijelöli a soron belül a megcímzett rekeszt.</td></tr>
<tr><td>DQ</td><td>Data I/O</td><td>Adat be- és kimenet.</td></tr>
<tr><td>UDQM</td><td>Upper Data I/O Mask</td><td rowspan="2">A több-bájtos adatbusszal rendelkező eszközök lehetővé teszik az adatbusz bájtonkénti maszkolását, vagyis annak meghatározását, hogy az adott írás- vagy olvasásművelet az adatbusz melyik bájtjára vonatkozzon. Az itt bemutatott IFA áramkör e két bemenetet – a CKE jelhez hasonlóan – fix értékre (0) kapcsolja, vagyis a maszkolás funkciót szintén nem használjuk ki.</td></tr>
<tr><td>LDQM</td><td>Lower Data I/O Mask</td></tr>
</table>

4-7. táblázat. *Az SDRAM katalógus-áramkör interfészjelei*

A 4-7. táblázatban definiált interfészjelek és parancsok segítségével az SDRAM áramkörnek az interfészadapterünk által is leképezett alapszolgáltatásai a 4-108–110. ábrákon szemléltetett időzítési diagramok szerint érhetők el.

Bekapcsolás után az SDRAM-ot inicializálni kell, ami a 4-108. ábrán látható műveletsor végrehajtását jelenti.

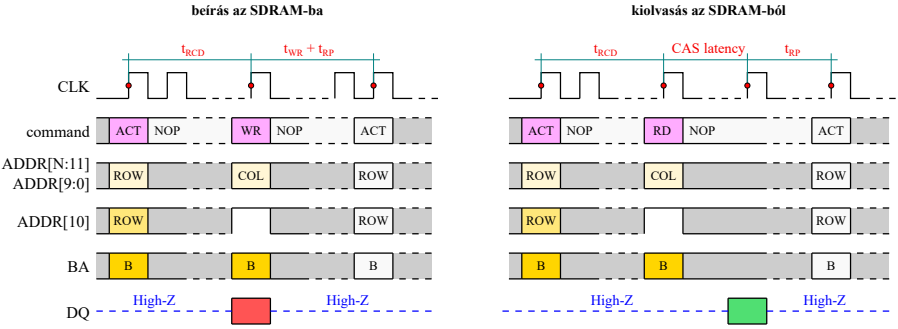


4-108. ábra. *Az SDRAM inicializálásának folyamata*

A tápfeszültség felfutása után legalább $t_{power\text{-}up}$ időtartamot várakoznunk kell a *CSn*, *RASn*, *CASn* és *WEn* bemeneteket a *no operation (NOP)* parancsnak megfelelő értékeken tartva. Ezután kiadjuk a *pre-charge (PCH)* parancsnak megfelelő kombinációt, magas szinten tartva az *ADDR* busz 10. helyiértékű bitjét, jelezve, hogy a visszatöltés műveletét az összes bankra egyszerre szeretnénk végrehajtani. Egy újabb, t_{RP} időtartamú várakozás után két *auto-refresh (ARF)* parancs kiküldésére van szükség. Ha tüzetesen megfigyeljük a 4-108–110. ábrák időzítési diagramjait, azt láthatjuk, hogy innentől kezdve az egyes parancsok elküldése egy jól áttekinthető minta szerint történik; a parancsvektor értékét egyetlen órajel-periódus ideig a *NOP*-tól eltérő értékre állítjuk, majd a parancsra – és a konkrét SDRAM-chipre – jellemző ideig várakozunk. Az inicializálás folyamata a két *auto-refresh* parancs kiküldése után a *mode register set (MRS)* paranccsal folytatódik, amely segítségével az eszköz globális paramétereit állíthatjuk be. E parancs végrehajtása során a beállítandó paraméterek kívánt értékének meghatározására a *BA* és *ADDR* bemeneteket használjuk fel[134] (a részleteket lásd később). A *mode register set* parancs elküldését egy t_{MRD} időtartamú várakozás kell hogy kövesse, csak ezután kerülhet sor az első írás vagy olvasás műveletet elindító *bank activate (ACT)* parancs elküldésére.

Az egyetlen bejegyzés beírásához és kiolvasásához szükséges interakciókat a 4-109. ábra szemlélteti. Mindkét műveletsort a kiválasztott sor aktiválásával kezdjük, a *bank activate* parancs elküldésével. Az eszköz a parancskód mintavételezésével egyidőben a *BA* és *ADDR* bemeneteit is beolvassa. A *BA* bemenet jelöli ki a megcímzett rekeszt tartalmazó bankot, az *ADDR* bemenet bitjei pedig a bankon belüli sor címét adják *(ROW)*. Beírás esetén az aktiválást követő t_{RCD} időtartamú várakozást a *write & pre-charge* (írás-visszatöltés, *WR*), olvasás esetén pedig a *read & pre-charge* (olvasás-visszatöltés, *RD*) parancs elküldése követi. E parancsokkal egyidőben az *ADDR* busz alsó $\log_2(C)$ vezetékére az oszlopcímet kapcsoljuk *(COL)*, így jelölve ki a teljes aktivált soron belül a felülírandó vagy kiolvasandó rekeszt, az *ADDR* busz 10. helyiértékű vezetékét pedig magas szintre állítjuk,

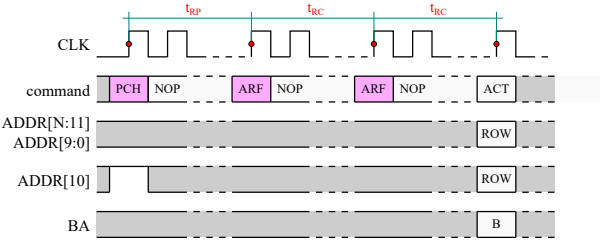
[134] Az eszköz címbemeneteit időosztásos alapon több célra is használjuk; a *mode register set* parancs végrehajtásakor paramétereket adunk át velük, illetve íráskor és olvasáskor az *ADDR* bemenet alsó bitjei előbb a sorcímet, majd egy lépéssel később az oszlopcímet hordozzák. E megoldás célja az eszköz kivezetésszámának csökkentése, a több kivezetés ti. nagyobb méretű tokot, egyúttal nagyobb gyártási költséget is jelent.



4-109. ábra. *Az SDRAM írásának és olvasásának folyamata*

amivel az írás/olvasás utáni automatikus visszatöltés végrehajtására utasítjuk az SDRAM-chip belső vezérlő logikáját. Beírás esetén ugyanebben az órajelciklusban kapuzzuk rá a beírandó adatot is a *DQ* buszra, majd egy $t_{WR} + t_{RP}$ időtartamú várakozás után a műveletet befejezettnek tekinthetjük. Olvasáskor a megcímzett rekesz tartalma az oszlopcím beléptetését követően a CAS latency paraméter által meghatározott késleltetés (2 vagy 3 órajelciklus az eszköz konfigurációjától függően) után kerül a *DQ* adatbuszra.

A teljes memóriatartalom periodikus kiolvasása és visszaírása a 4-109. ábrán látható műveletek segítségével elvileg lehetséges, de felhasználói szempontból igencsak körülményes lenne. Szerencsére az SDRAM ennél kényelmesebb lehetőséget is biztosít; külön szolgáltatást definiál a memóriatartalom automatikus frissítésére, amit a 4-110. ábra mutat be.



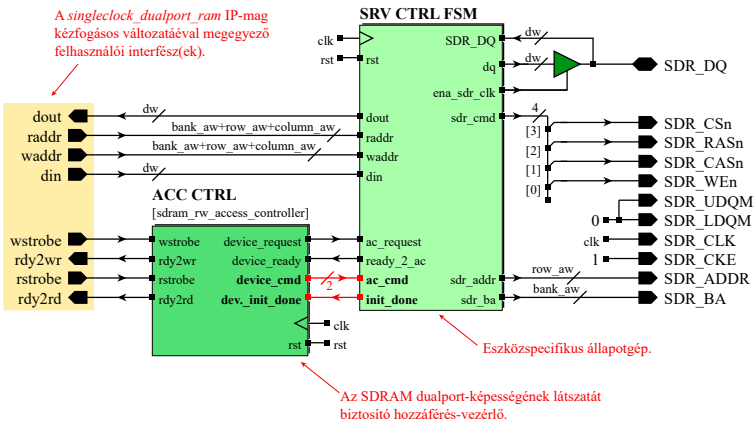
4-110. ábra. *Az SDRAM automatikus frissítésének folyamata*

A frissítési periódusok végén a felhasználó dolga csupán annyi, hogy elküldje a *pre-charge + auto-refresh + auto-refresh* parancssorozatot, természetesen beiktatva a megfelelő várakozásokat is. A *pre-charge* paranccsal egyidőben az *ADDR* busz 10. helyiértékű vezetékét magas szinten tartva jelezhetjük, hogy az összes bank frissítésére igényt tartunk.[135]

Az SDRAM interfészadapter 4-111. ábrán szemléltetett szerkezete két ponton jelentősen eltér az EEPROM IFA-knál látottaktól. Egyrészt, mivel az SDRAM nem soros kommunikációs interfésszel van ellátva, ezért soros-párhuzamos/párhuzamos-soros átalakítóra nincs szükség, az eszköz kivezetéseit közvetlenül az IFA központi elemét képező szolgáltatásvezérlő állapotgép kezeli.

[135] Lehetőség van a bankonkénti frissítésre is, ekkor az *ADDR* busz 10. helyiértékű vezetékét nulla szintűre kell állítanunk, a *BA* bemenetekre pedig a frissítésre kijelölt bank címét kell kapcsolnunk. A bankonkénti frissítés lehetőségét az itt bemutatott interfészadapterben nem használjuk ki.

A másik jelentős eltérés, hogy a hozzáférés-vezérlő feladata ebben az esetben már nem merül ki a felhasználói kérések priorizálásában és a szolgáltatásvezérlő felé való továbbításában; a periodikus frissítésekre vonatkozó kéréseket is ő generálja, egy szintézis-paraméter segítségével beál-



4-111. ábra. *Az SDRAM interfészadapter szerkezete*

lítható időközönként.[136] Ehhez a hozzáférés-vezérlő és a szolgáltatásvezérlő közötti interfészt ki kell bővítenünk. Az alapvetően szinkron kézfogásos interfész egybites *device_rnw* jelét egy kétbites parancskóddal *(device_cmd)* helyettesítjük, amely az íráson és olvasáson kívül így már további kétféle kérést is képes közvetíteni. A hozzáférés-vezérlőnek ugyancsak gondoskodnia kell arról, hogy a felhasználói kérésekkel az SDRAM áramkört mindaddig ne zargassuk, amíg annak bekapcsolás utáni inicializálása nem zárult le. A szolgáltatásvezérlő az inicializálás befejeződését az *init_done* kimenetén jelzi a hozzáférés-vezérlőnek, ami csak ennek felfutása után kezdi meg a felhasználói interfész kéréseinek feldolgozását és a periodikus frissítések időzített indítását. Az ezek alapján módosított hozzáférés-vezérlő RTL-modelljét a 4-112. ábra mutatja be.

A felhasználói *w/rstrobe* és *rdy2wr/rd* jelek kezelése csaknem teljesen megegyezik az EEPROM interfészadaptereknél látottakkal, az egyetlen apró különbség az, hogy ezúttal a *w/rstrobe_flag* állapotjelzők reset hatására nem nullára, hanem egyre inicializálódnak, első nullázásukat a hozzáférés-vezérlő állapotgép csak az SDRAM inicializálásának végén hajtja végre (lásd 4-113. ábra: *wait_for_init* állapot). A felhasználó tehát az eszköz rendelkezésre állásáig azt látja, hogy a *rdy2wr/rd* kimenetek alacsony szintűek, vagyis az eszköz sem írásra, sem olvasásra „nem áll készen”.

Ahogy az a 4-112. ábra alsó részén látható, a hozzáférés-vezérlő tartalmazza a periodikus frissítést időzítő számlálót *(refr_timeout_counter)*. Miután a szolgáltatásvezérlő jelezte az SDRAM inicializálási folyamatának végét *(device_init_done)*, e számláló órajelciklusonként eggyel növekszik, mígnem eléri a *refr_timeout_counter_max* paraméter értékét. Ekkor a *refr_timeout_flag* állapotjelző egybe billen, jelezve a hozzáférés-vezérlő állapotgépnek, hogy ideje kezdeményezni – a szolgáltatásvezérlőn

[136] Ily módon az SDRAM áramkörnek azzal a sajátos igényével, mely szerint tartalmát periodikusan frissíteni kell, a felhasználónak nincs feladata. Az on-chip RAM-okat használó *singleclock_dualport_ram_hswrp* IP, a külső EEPROM-okat illesztő IFA-k és az SDRAM IFA így minimális munkával „cserélgethetők” egy összetett toplevel modulban, ami a modularitásra és regularitásra vonatkozó alapelvek következetes betartásának köszönhető.

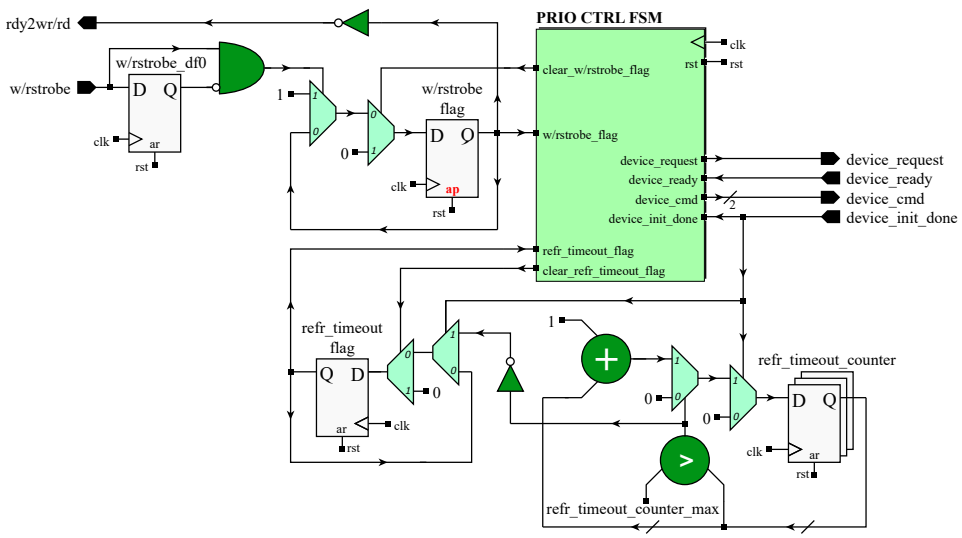
Verilog

```
1  // STROBE LOGIC
2  always @ ( posedge clk, posedge rst ) begin
3    if ( rst == 1 ) begin
4      wstrobe_df0 <= 0; rstrobe_df0 <= 0;
5      wstrobe_flag <= 1; rstrobe_flag <= 1;
6    end else begin
7      wstrobe_df0 <= wstrobe; rstrobe_df0 <= rstrobe;
8      if ( clear_wstrobe_flag == 1 ) wstrobe_flag <= 0;
9      else if ( wstrobe_rising == 1 ) wstrobe_flag <= 1;
10     if ( clear_rstrobe_flag == 1 ) rstrobe_flag <= 0;
11     else if ( rstrobe_rising == 1 ) rstrobe_flag <= 1;
12   end
13 end
14 assign wstrobe_rising = wstrobe & ~wstrobe_df0;
15 assign rstrobe_rising = rstrobe & ~rstrobe_df0;
16 assign rdy2wr = ~wstrobe_flag;
17 assign rdy2rd = ~rstrobe_flag;
18
19 // REFRESH TIMEOUT COUNTER
20 always @ ( posedge clk, posedge rst ) begin
21   if ( rst == 1 ) begin
22     refr_timeout_counter <= 0; refr_timeout_flag <= 0;
23   end else begin
24
25     if ( device_init_done == 1 ) begin
26       if ( refr_timeout_counter < refr_timeout_counter_max )
27         refr_timeout_counter <= refr_timeout_counter + 1;
28       else begin
29         refr_timeout_counter <= 0; refr_timeout_flag <= 1;
30       end
31     end else refr_timeout_counter <= 0;
32
33     if ( clear_refr_timeout_flag == 1 )
34       refr_timeout_flag <= 0;
35
36   end
37 end
38
```

Verilog (folyt.)

```
39 // PRIO CTRL FSM
40 always @ ( posedge clk, posedge rst )
41 begin
42   if ( rst == 1 ) begin
43     state <= wait_for_init;
44     clear_wstrobe_flag <= 0;
45     clear_rstrobe_flag <= 0;
46     clear_refr_timeout_flag <= 0;
47     device_request <= 0;
48     device_cmd <= 0;
49     last_access_type <= 0;
50   end else begin
51
52     // defaults...
53     clear_wstrobe_flag <= 0;
54     clear_rstrobe_flag <= 0;
55     clear_refr_timeout_flag <= 0;
56     device_request <= 0;
57
58     case ( state )
59
60       wait_for_init: [...]
61
62       [...]
63
64       default:    ;
65
66     endcase
67
68   end
69 end
```

Áramköri struktúra



4-112. ábra. *Az sdram_rw_access_controller modul RTL-modellje*

Verilog

```
1  wait_for_init: if ( device_init_done == 1 ) begin
2                   clear_rstrobe_flag <= 1;
3                   clear_wstrobe_flag <= 1;
4                   state <= wait_3;
5                 end
6
7  wait_1: state <= wait_2;
8
9  wait_2: if ( device_ready == 1 ) begin
10           |
11           | if ( device_cmd == 1 )
12           |   clear_rstrobe_flag <= 1;
13           | else if ( device_cmd == 0 )
14           |   clear_wstrobe_flag <= 1;
15           | else clear_refr_timeout_flag <= 1;
16           |
17           | state <= wait_3;
18           |
19         end
20
21 wait_3: state <= idle;
22
```

Verilog (folyt.)

```
23 idle:
24   if ( refr_timeout_flag == 1 ) begin
25   | device_cmd <= 2; // refresh
26   | device_request <= 1;
27   | state <= wait_1;
28   end else begin
29   | if ( last_access_type == 0 ) begin // write
30   | |
31   | | if ( rstrobe_flag == 1 ) begin
32   | |   device_cmd <= 1; // read
33   | |   device_request <= 1;
34   | |   last_access_type <= 1;
35   | |   state <= wait_1;
36   | | end else if ( wstrobe_flag == 1 ) begin
37   | |   device_cmd <= 0; // write
38   | |   device_request <= 1;
39   | |   last_access_type <= 0;
40   | |   state <= wait_1;
41   | | end
42   | |
43   | end else begin // read
44   | |
45   | | if ( wstrobe_flag == 1 ) begin
46   | |   device_cmd <= 0; // write
47   | |   device_request <= 1;
48   | |   last_access_type <= 0;
49   | |   state <= wait_1;
50   | | end else if ( rstrobe_flag == 1 ) begin
51   | |   device_cmd <= 1; // read
52   | |   device_request <= 1;
53   | |   last_access_type <= 1;
54   | |   state <= wait_1;
55   | | end
56   | |
57   | end
58   end
```

4-113. ábra. *Az SDRAM hozzáférés-vezérlő állapotgépének Verilog-leírása*

keresztül – az SDRAM teljes tartalmának automatikus frissítését. A hozzáférés-vezérlő állapotgép *idle* állapotában látható, hogy az időzítő a felhasználói kéréseknél magasabb prioritású, a frissítés tehát az éppen folyamatban lévő írás vagy olvasás befejezésekor ugyan azonnal, de annál semmiképp sem előbb kezdődik meg. Ezért fontos, hogy a két frissítés között eltelő időt meghatározó *refr_timeout_counter_max* paraméter értékét úgy állítsuk be, hogy az legalább egy írásnyi/olvasásnyi idővel kisebb legyen, mint a konkrét SDRAM áramkör adatlapja alapján a frissítések között megengedett maximális időtartam, máskülönben a tárolt információ elveszhet. A hozzáférés-vezérlő állapotgép az írás- és olvasáskérések közötti prioritást hasonlóan kezeli, mint az EEPROM-okhoz készített verzió, ezúttal azonban a legutóbbi kérés típusát nem az állapotgép kétféle tétlen állapota, hanem egy dedikált állapotjelző *(last_access_type)* kódolja (4-113. ábra: 29. sor).

A szolgáltatásvezérlő állapotgép VHDL-leírásának vázlatát a 4-114. ábra szemlélteti. Ennek az almodulnak a feladata az SDRAM áramkörnek a 4-108–110. ábrákon bemutatott szolgáltatásainak kezdeményezése, közvetlenül az eszköz kivezetésein keresztül. Ennek megfelelően az állapotgép HDL-modellje négy jól elkülöníthető állapotsorozatot definiál; egyet-egyet az inicializálás, az írás-visszatöltés, az olvasás-visszatöltés, valamint az automatikus frissítés műveletsorokra.

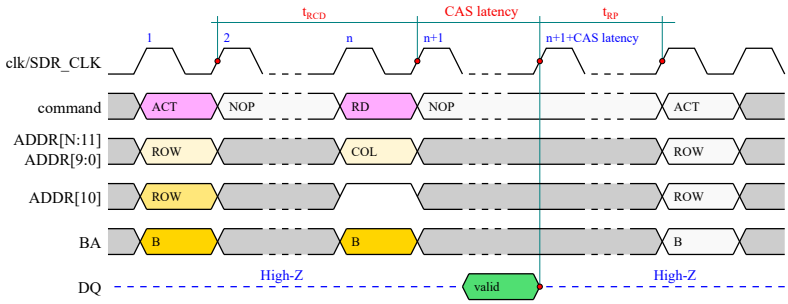
A vezérlési állapotok részletes vizsgálata előtt érdemes egy rövid kitérőt tennünk az interfészadapter és az SDRAM áramkör interfészjeleinek időzítési viszonyaival kapcsolatban annak érdekében, hogy pontosan lássuk, milyen működést várunk el az interfészadaptertől. Az SDRAM katalógus-áramkör egy szinkron DRAM, vagyis minden bemenetét az órajel felfutóélénél mintavételezi, és kimenetein a változásokat olyan módon idézi elő, hogy azok az órajel felfutóéleinél

VHDL

```
   [...]                                                                    -- CSn_RASn_CASn_WEn
   constant cmd_device_deselect:     std_logic_vector (3 downto 0)    := B"1_0_0_0";
   constant cmd_nop:                 std_logic_vector (3 downto 0)    := B"0_1_1_1";
   constant cmd_read_and_precharge:  std_logic_vector (3 downto 0)    := B"0_1_0_1";
   constant cmd_write_and_precharge: std_logic_vector (3 downto 0)    := B"0_1_0_0";
   constant cmd_bank_activate:       std_logic_vector (3 downto 0)    := B"0_0_1_1";
   constant cmd_precharge:           std_logic_vector (3 downto 0)    := B"0_0_1_0";
   constant cmd_auto_refresh:        std_logic_vector (3 downto 0)    := B"0_0_0_1";
   constant cmd_mode_register_set:   std_logic_vector (3 downto 0)    := B"0_0_0_0";
   [...]
   ------------------------------------------------------------------------------------------------------
   [...]
1  L_SRV_CTRL_FSM: process ( clk, rst ) begin
2  | if ( rst = '1' ) then
3  | | state <= initialize_1; delay_cnt <= 0; init_done <= '0'; ready_2_ac <= '1';
4  | | ena_dq <= '0'; dq <= (others => '0'); sdr_cmd <= cmd_device_deselect;
5  | | SDR_ADDR <= (others => '0'); SDR_BA <= (others => '0'); dout <= (others => '0');
6  | elsif ( rising_edge(clk) ) then
7  | |
8  | | case state is
9  | | |
10 | | | when initialize_1 => [...]
11 | | |
12 | | | [...]
13 | | |
14 | | | when others => null;
15 | | |
16 | | end case;
17 | |
18 | end if;
19 end process;
```

4-114. ábra. *Az SDRAM IFA szolgáltatásvezérlő állapotgépének vázlata*

mintavételezhetőek legyenek. Ennek hangsúlyozása céljából a 4-108–110. ábrák – az eszköz adatlapjában is megtalálható egyszerűsített időzítési diagramokhoz hasonlóan – azt sugallják, hogy a vezérlő-, cím- és adatbuszok pontosan az órajel lefutóéleinél változnak meg, így biztosítva, hogy az órajel felfutóéleinél már stabil, egyértelműen mintavételezhető állapotban legyenek. Ez a modell jól használható arra, hogy az eszköz különféle szolgáltatásainak működését szemléltesse, de fizikai szempontból pontatlan. Az interfészadapter szerkezetét bemutató 4-111. ábrán látható, hogy az SDRAM *CLK* bemenetét közvetlenül az interfészadapter saját órajelével hajtjuk meg, vagyis a szolgáltatásvezérlő állapotgép generálta buszjelek változásait a közös órajel felfutóélei váltják ki. Mindezek alapján elkészíthetők a két eszköz közötti adatcserét leíró időzítési diagramoknak a pontosított – bár bizonyos szempontból még mindig idealizált – változatai. A 4-115. ábra az olvasás műveletének ezt a pontosított modelljét szemlélteti.



4-115. ábra. *Az SDRAM olvasásának pontosított időzítési modellje*

Ez a hullámforma egy tipikus „időzítetlen” (zero-delay), a leendő áramkörök valós késleltetéseinek hatását mellőző, tisztán logikai szimulációkra jellemző modell, ahol bármely adatbusz megváltozása és mintavételezése is az órajel felfutóéle hatására, látszólag azzal egyidőben történik meg. Ez a modell tehát még mindig idealizált abban az értelemben, hogy az órajel által kiváltott változások egy valós áramkörben nyilván nem történhetnek meg azonnal, tehát az adatbuszok változásai csak az azokat kiváltó órajel-felfutóélek után következhetnek be. Másrészt a mintavételezés pillanata sem eshet egybe pontosan a mintavett adatbuszok megváltozásával, hiszen akkor sérülnének a fogadóoldali flip-flopok beállítási és tartási időkre vonatkozó előírásai. Az automatizált szintézis feladata, hogy az RTL-tervezés során használt időzítési modellünknek ezt a pontatlanságát elfedve garantálja a tervezés során tett hallgatólagos feltételezéseink teljesülését, vagyis hogy olyan áramkört hozzon létre az ismert késleltetésű alapelemkészletből, amely végül logikailag ugyanazt a működést valósítja meg, amelyet az RTL-modellünk leír. Az automatizált szintézis ezt a feladatot akkor képes ellátni, ha az RTL-modellhez tartozó időzítési peremfeltételeket is rögzítjük. Legegyszerűbb esetben ez annyit jelent, hogy előírjuk a leendő áramkörtől elvárt maximális órajel-frekvenciát, ami alapján az egyazon lokális adatút kibocsátó és fogadó flip-flopjai közötti időzítési viszonyok kiszámíthatók. Az SDRAM-interfészáramkör esetén a helyzet kicsit bonyolultabb. Azzal, hogy a szolgáltatásvezérlő állapotgép órajelét és az SDRAM órajelbemenetét a tervezés során lényegében egynek tekintjük, azt állítva, hogy a két fél közötti interfészjelek egy-egy lokális adatutat képeznek, nehéz helyzetbe hozzuk a szintézisszoftvert, hiszen pusztán az órajel frekvenciájának ismerete nem elegendő az időzítési viszonyok kiszámításához, ahhoz az SDRAM belső

VHDL

```
when initialize_1 =>
  sdr_cmd <= cmd_nop;

  if (
    delay_cnt <
    cycles_in_power_up_delay
  ) then
    delay_cnt <= delay_cnt + 1;
  else
    delay_cnt <= 0;
    state <= initialize_2;
  end if;

when initialize_2 =>
  sdr_cmd <= cmd_precharge;
  SDR_ADDR(10) <= '1';

  if ( delay_cnt < trp_cycles ) then
    delay_cnt <= delay_cnt + 1;
  else
    delay_cnt <= 0;
    sdr_cmd <= cmd_auto_refresh;
    state <= initialize_3;
  end if;

when initialize_3 =>
  sdr_cmd <= cmd_nop;

  if ( delay_cnt < trc_cycles-1 ) then
    delay_cnt <= delay_cnt + 1;
  else
    delay_cnt <= 0;
    sdr_cmd <= cmd_auto_refresh;
    state <= initialize_4;
  end if;

```

VHDL (folyt.)

```
when initialize_4 =>
  sdr_cmd <= cmd_nop;

  if ( delay_cnt < trc_cycles-1 ) then
    delay_cnt <= delay_cnt + 1;
  else
    delay_cnt <= 0;
    sdr_cmd <= cmd_mode_register_set;
    SDR_ADDR(2 downto 0) <= "000"; -- burst length: 1
    SDR_ADDR(6 downto 4) <= "011"; -- CAS latency: 3
    SDR_ADDR(8 downto 7) <= "00";  -- standard operation
    SDR_ADDR(9) <= '1';            -- single location access
    SDR_ADDR(row_aw-1 downto 10) <= (others => '1');
    SDR_BA <= (others => '1');
    state <= initialize_5;
  end if;

when initialize_5 => sdr_cmd <= cmd_nop;
                     init_done <= '1';
                     state <= idle;

-------------------------------------------------------------

when idle =>
  ready_2_ac <= '1';

  if ( ac_request = '1' ) then
    ready_2_ac <= '0';
    case ac_cmd is
      when "00"    => state <= write_and_precharge_1;
      when "01"    => state <= read_and_precharge_1;
      when "10"    => state <= auto_refresh_1;
      when others  => null;
    end case;
  end if;
```

4-116. ábra. *Az SDRAM inicializálását végző vezérlési állapotok*

flip-flopjainak időzítési jellemzőit is ismerni kellene. E flip-flopokról még az eszköz adatlapja sem árul el közvetlenül semmit, de rögzíti azt, hogy az eszköz kivezetésein milyen időzítési peremfeltételeknek kell teljesülniük az órajel és az adatvezetékek viszonyára vonatkozóan. Ezeket az adatlapi információkat úgy adjuk át a szintézisszoftvernek, hogy az órajelfeltétel (clock constraint) mellett az interfészjelek be- és kimeneti késleltetéseire (input/output delay) vonatkozóan is előírásokat fogalmazunk meg SDC (**S**ynopsys **D**esign **C**onstraint) formátumban. Ezeket az időzítési peremfeltételeket a 4.4.3.1. pontban tárgyaljuk részletesen, a szolgáltatásvezérlő állapotgépének vizsgálatakor pedig egyelőre feltételezzük azt, hogy az IFA és az SDRAM egyazon órajelről járnak és interfészjeleik egy-egy szokásos lokális adatutat képeznek.

Az SDRAM inicializálását végző állapotsorozat (lásd 4-116. ábra) első állapotában a *no operation* parancsnak megfelelő kombinációt írjuk az SDRAM vezérlőbemeneteit *(CSn, RASn, CASn, WEn)* meghajtó *sdr_cmd* regiszterbe (4-116. ábra: 2. sor), és ezen mindaddig nem is változtatunk, amíg el nem telt a *cycles_in_power_up_delay* paraméter által meghatározott számú órajelciklus. Azután az *initialize_2* állapotban az *SDR_ADDR* kimenet 10. helyiértékű bitjét magas szinten tartva kiadjuk a *pre-charge* parancsot (4-116. ábra: 15. sor), amelyet a *trp_cycles* paraméter által meghatározott ideig kint is tartunk a vezérlőkimeneteken.

Ezután egyetlen órajel-periódus ideig az *auto-refresh* parancs kerül a vezérlőkimenetekre (4-116. ábra: 22. sor), amit a *no operation* parancs követ egy, a *trc_cycles* paraméterrel meghatározott várakozás erejéig. Az *auto-refresh + no operation* parancskombinációt még egyszer megismételjük, mielőtt az *initialize_4* állapotban a *mode register set* parancs és a címbemenetek megfelelő beállításával beírjuk a globális konfigurációs információt az SDRAM-ba (4-116. ábra: 44–50. sor). Ezzel az inicializálás folyamata befejeződött, amiről az *initialize_5* állapotban az *init_done* kimenet magas szintre húzásával a hozzáférés-vezérlőt és a periodikus frissítésekért felelős számlálót is értesítjük (4-116. ábra: 55. sor). Az inicializálás után a szolgáltatásvezérlő az *idle* állapotba lép, ahol nincs más dolga, mint várakozni a hozzáférés-vezérlőtől érkező kérésekre (4-116. ábra: 63. sor), és azok beérkezése esetén a kért szolgáltatást kijelölő *ac_cmd* bemenet értéke alapján a megfelelő állapotsorozat első állapotára ugrani.

Az SDRAM-ba írás folyamatához tartozó vezérlési állapotokat a 4-117. ábra mutatja be. A *bank activate* parancs kiküldésével egyidőben az SDRAM címbemeneteit is beállítjuk (4-117. ábra: *write_and_precharge_1* állapot). Az interfészadapter felhasználói interfészének *waddr* és *raddr* címbemenetei az SDRAM teljes címtartományát lefedik, hiszen a felhasználónak nincs tudomása arról, hogy maga az SDRAM bankcím + sorcím + oszlopcím felosztásban kezeli a címinformációt. Az SDRAM *BA* és *ADDR* bemeneteire kapcsolandó értékeket tehát a *w/raddr* bemenetekből kell képeznünk. A beírás (és kiolvasás) első lépésében az SDRAM *BA* bankcím-bemenetei a felhasználói cím felső *bank_aw* db bitjét kapják (4-117. ábra: 7–8. sor), az *ADDR* bemenetek pedig a sorcímet, amit a felhasználói cím bankcím alatti *row_aw* db bitje képez (4-117. ábra: 4–5. sor). A bank- és soraktiválás után – a *trcd_cycles* paraméterben definiált várakozás elteltével – az SDRAM *ADDR* bemeneteit az oszlopcím beírására használjuk (4-117. ábra: 21–22. sor), egyúttal magas szintre húzzuk az *ADDR* bemenet 10. helyiértékű bitjét (4-117. ábra: 20. sor), ami az SDRAM specifikációja alapján a beírás művelet utáni automatikus visszatöltést írja elő az SDRAM vezérlőlogikája számára. Ebben az órajelciklusban az *ena_dq* belső vezérlőjel segítségével az SDRAM *DQ* kimenetét meghajtó háromállapotú kimeneti buffert is engedélyezzük (4-117. ábra: 23. sor), miközben magára a *DQ* kimenetre a felhasználótól érkező adatot, vagyis a *din* vektor értékét írjuk (4-117. ábra: 24. sor). Az adatot a *DQ* buszon csak egyetlen órajel-periódus ideig tarthatjuk kint, ezért

VHDL

```
1  when write_and_precharge_1 =>
2    sdr_cmd <= cmd_bank_activate;
3
4    SDR_ADDR <= waddr(waddr'length-bank_aw-1 downto
5                      column_aw);
6
7    SDR_BA <= waddr(waddr'length-1 downto
8                    waddr'length-bank_aw);
9
10   state <= write_and_precharge_2;
11
12 when write_and_precharge_2 =>
13   sdr_cmd <= cmd_nop;
14
15   if ( delay_cnt < trcd_cycles-1 ) then
16     delay_cnt <= delay_cnt + 1;
17   else
18     delay_cnt <= 0;
19     sdr_cmd <= cmd_write_and_precharge;
20     SDR_ADDR(10) <= '1';
21     SDR_ADDR(column_aw-1 downto 0) <=
22        waddr(column_aw-1 downto 0);
23     ena_dq <= '1';
24     dq <= din;
25     state <= write_and_precharge_3;
26   end if;
27
```

VHDL (folyt.)

```
28 when write_and_precharge_3 =>
29   ena_dq <= '0';
30   sdr_cmd <= cmd_nop;
31
32   if ( delay_cnt < twr_cycles+trp_cycles ) then
33     delay_cnt <= delay_cnt + 1;
34   else
35     delay_cnt <= 0;
36     state <= idle;
37   end if;
```

4-117. ábra. *Az SDRAM-ba egy bejegyzést beíró vezérlési állapotok*

VHDL

```
1  when read_and_precharge_1 =>
2    sdr_cmd <= cmd_bank_activate;
3
4    SDR_ADDR <= raddr(raddr'length-bank_aw-1 downto
5                      column_aw);
6
7    SDR_BA <= raddr(raddr'length-1 downto
8                    raddr'length-bank_aw);
9
10    state <= read_and_precharge_2;
11
12 when read_and_precharge_2 =>
13   sdr_cmd <= cmd_nop;
14
15   if ( delay_cnt < trcd_cycles-1 ) then
16     delay_cnt <= delay_cnt + 1;
17   else
18     delay_cnt <= 0;
19     sdr_cmd <= cmd_read_and_precharge;
20     SDR_ADDR(10) <= '1';
21     SDR_ADDR(column_aw-1 downto 0) <=
22        raddr(column_aw-1 downto 0);
23     state <= read_and_precharge_3;
24   end if;
25
```

VHDL (folyt.)

```
26 when read_and_precharge_3 =>
27   sdr_cmd <= cmd_nop;
28
29   if ( delay_cnt < cas_latency-1 ) then
30     delay_cnt <= delay_cnt + 1;
31   else
32     delay_cnt <= 0;
33     state <= read_and_precharge_4;
34   end if;
35
36 when read_and_precharge_4 =>
37   dout <= sdr_dq;
38   state <= read_and_precharge_5;
39
40 when read_and_precharge_5 =>
41   if ( delay_cnt < trp_cycles-1 ) then
42     delay_cnt <= delay_cnt + 1;
43   else delay_cnt <= 0; state <= idle;
44   end if;
```

4-118. ábra. *Az SDRAM-ból egy bejegyzést kiolvasó vezérlési állapotok*

a *write_and_precharge_3* állapotban az *ena_dq* vezérlőjelet rögtön vissza is vesszük nullába (4-117. ábra: 29. sor), majd a *twr_cycles* és *trp_cycles* paraméterek meghatározta késleltetés után visszatérünk *idle* állapotba, amivel a beírás folyamata véget ér.

A kiolvasás vezérlési állapotaiban (lásd 4-118. ábra) a címkezelés csaknem teljesen megegyezik a beírásnál látottakkal, a különbség csak annyi, hogy az SDRAM *BA* és *ADDR* bemeneteinek értékeit ezúttal a felhasználótól érkező olvasási cím *(raddr)* alapján határozzuk meg. A *read_and_precharge_2* állapot ugyancsak hasonló az írás megfelelő állapotához, ezúttal azonban értelemszerűen nem foglalkozunk sem a háromállapotú kimeneti bufferrel, sem annak adatbemenetével, hiszen a *DQ*

VHDL

```
when auto_refresh_1 => sdr_cmd <= cmd_precharge;
                       SDR_ADDR(10) <= '1';
                       state <= auto_refresh_2;

when auto_refresh_2 =>
  sdr_cmd <= cmd_nop;

  if ( delay_cnt < trp_cycles-1 ) then
    delay_cnt <= delay_cnt + 1;
  else
    delay_cnt <= 0;
    sdr_cmd <= cmd_auto_refresh;
    state <= auto_refresh_3;
  end if;

```

VHDL (folyt.)

```
when auto_refresh_3 =>
  sdr_cmd <= cmd_nop;

  if ( delay_cnt < trc_cycles-1 ) then
    delay_cnt <= delay_cnt + 1;
  else
    delay_cnt <= 0;
    sdr_cmd <= cmd_auto_refresh;
    state <= auto_refresh_4;
  end if;

when auto_refresh_4 =>
  sdr_cmd <= cmd_nop;

  if ( delay_cnt < trc_cycles-1 ) then
    delay_cnt <= delay_cnt + 1;
  else delay_cnt <= 0; state <= idle;
  end if;
```

4-119. ábra. *Az SDRAM tartalmának frissítését végző vezérlési állapotok*

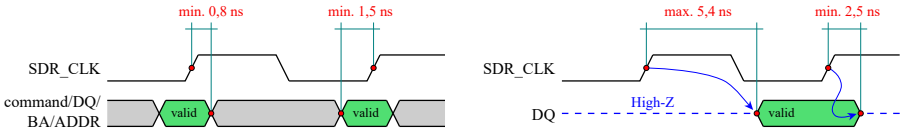
buszt ezúttal olvasni fogjuk. A *read_and_precharge_3* állapot így tulajdonképpen egy várakozásra egyszerűsödik, amely után a *read_and_precharge_4* állapotban kerül sor az SDRAM *DQ* kimeneti buszának a *dout* regiszterbe töltésére, ami egyúttal a felhasználói interfész adatkimenetét is jelenti. A *trp_cycles* paraméterben definiált várakozás után ismét az *idle* állapotba térünk vissza, amivel az olvasás befejezettnek tekinthető.

A szolgáltatásvezérlő által megvalósított harmadik vezérlésiállapot-sorozat az SDRAM automatikus tartalomfrissítés szolgáltatását hivatott levezényelni (lásd 4-119. ábra). Az SDRAM adat- és címbemenetei közül ebben a folyamatban csak az *ADDR* bemenet 10. helyiértékű bitje szerepel, ennek értékét az *auto_refresh_1* állapotban magas szintre húzva jelezzük, hogy az összes bank tartalmát szeretnénk frissíteni (4-119. ábra: 2. sor).

Az automatikus frissítés a *pre-charge* utasítást követő két *auto-refresh* parancs kiküldésével kezdeményezhető, ennek megfelelően az ehhez a művelethez tartozó vezérlési állapotok a parancskód megfelelő ütemezésű váltogatását végzik. A szolgáltatásvezérlő ennek az állapotsorozatnak a végén is az *idle* állapotba tér vissza, ahol várakozik a hozzáférés-vezérlőtől érkező kérésekre.

4.4.3.1. Időzítési peremfeltételek

Az SDRAM áramkör és az interfészadapter közötti időzítési viszonyok finomhangolásához a 4-108–110. ábrák egyszerűsített diagramjai nem elegendők, az eszköz adatlapjából további információkat kell kikeresnünk. A számunkra legfontosabb – az órajel, valamint a vezérlő-, cím- és adatbuszok időzítési viszonyaira vonatkozó – előírásokat a 4-120. ábra szemlélteti egy konkrét SDRAM katalógus-áramkör (Integrated Silicon Solutions Inc. IS42S16320F) esetén.



4-120. ábra. *Egy konkrét SDRAM katalógus-áramkör időzítési tulajdonságai*

Az SDRAM áramkör adatlapja tehát azt fogalmazza meg, hogy az eszköz vezérlő- *(CSn, RASn, CASn, WEn)*, cím- *(ADDR, BA)* és adatbemeneteinek *(DQ)* az órajelbemenet felfutóélei előtt legalább 1,5 ns-mal, és azt követően még legalább további 0,8 ns ideig stabilnak kell lenniük.[137] A másik időzítési sajátosság az, hogy olvasás esetén az eszköz a *DQ* kimenetét a megfelelő órajel-felfutóél után 5,4 ns-mal garantáltan stabilizálja, és a következő (az adatkimenet mintavételezésére használt) órajel-felfutóél után még legalább 2,5 ns ideig stabilan is tartja.

Azokat az interfészeket, ahol az órajelet a két fél nem egy közös külső forrástól kapja, hanem azt az egyik fél állítja elő, „forrás-szinkron" interfészeknek nevezzük. Ezt a megoldást előszeretettel alkalmazzuk nagysebességű párhuzamos adatátvitelre, mivel az órajel és az adatvonalak időzítési viszonyai így egy helyen (a forrás oldalán) finoman hangolhatók. Esetünkben ez a finomhangolás azt jelenti, hogy a 4-120. ábrán az SDRAM kivezetéseire megfogalmazott időzítési előírásokat az FPGA kivezetéseire vonatkozó feltételekként fejezzük ki a fizikai szintézis és a statikus időzítésvizsgálat algoritmusai számára. Háromféle peremfeltétel-típust használunk. Az órajelfeltétel rögzíti a referencia-órajel frekvenciáját, ami esetünkben az SDRAM órajelbemenete lesz. A kimeneti késleltetés (output delay) típusú időzítési kényszerekkel (timing constraint) fejezhetjük ki az SDRAM órajelének és bemeneteinek, a bemeneti késleltetés (input delay) típusúakkal pedig az órajelnek és az SDRAM által az olvasás művelet során meghajtott *DQ* busznak az időzítési viszonyait. A 4-121. ábra az e kényszereket formálisan rögzítő SDC fájl tartalmát mutatja be.

Az SDC fájlban első lépésként definiálnunk kell az *SDR_CLK* kimenetet mint referenciaórajelet, amelyhez képest a be- és kimeneti késleltetéseket értelmezzük (4-121. ábra: 1–3. sor). Esetünkben ezt az órajelet egy, az FPGA-ban lévő PLL állítja elő. Mivel a tervezőrendszer az adott PLL konfigurációját – így kimeneti frekvenciáját is – „ismeri", ezzel egyúttal az órajel frekvenciáját is definiáltuk (ebben a konkrét példában ez 100 MHz). Az SDC fájl 5–11. soraiban látható kimenetikésleltetés-előírások azt fejezik ki, hogy a megnevezett vezetékeken terjedő jelek legfeljebb 1,5 ns-mal több késleltetést szenvednek el, mint maga az *SDR_CLK* kimeneten terjedő változások. Úgy is fogalmazhatunk, hogy az automatikus fizikai szintézis algoritmusok számára előírjuk, hogy az interfészadapter áramköri struktúráját (ezáltal az *SDR_CLK* kimenetnek és az adatkimeneteknek

[137] Bár az adatlap ezt nem fejezi ki egyértelműen, de feltételezhetjük, hogy az ezeket a bemeneteket mintavételező belső flip-flopok beállítási (setup) és tartási (hold) időire vonatkozó kritériumok állnak a dolog hátterében.

SDC (Synopsys Design Constraint)

```
1  create_generated_clock -name SDR_CLK
2                         -source [get_pins {L_PLL|altpll_component|auto_generated|pll1|clk[0]}]
3                         [get_ports {SDR_CLK}]
4
5  set_output_delay -clock SDR_CLK -max 1.5 [get_ports SDR_DQ*]
6  set_output_delay -clock SDR_CLK -max 1.5 [get_ports SDR_ADDR*]
7  set_output_delay -clock SDR_CLK -max 1.5 [get_ports SDR_BA*]
8  set_output_delay -clock SDR_CLK -max 1.5 [get_ports SDR_WEn]
9  set_output_delay -clock SDR_CLK -max 1.5 [get_ports SDR_RASn]
10 set_output_delay -clock SDR_CLK -max 1.5 [get_ports SDR_CASn]
11 set_output_delay -clock SDR_CLK -max 1.5 [get_ports SDR_CSn]
12
13 set_output_delay -clock SDR_CLK -min -0.8 [get_ports SDR_DQ*]
14 set_output_delay -clock SDR_CLK -min -0.8 [get_ports SDR_ADDR*]
15 set_output_delay -clock SDR_CLK -min -0.8 [get_ports SDR_BA*]
16 set_output_delay -clock SDR_CLK -min -0.8 [get_ports SDR_WEn]
17 set_output_delay -clock SDR_CLK -min -0.8 [get_ports SDR_RASn]
18 set_output_delay -clock SDR_CLK -min -0.8 [get_ports SDR_CASn]
19 set_output_delay -clock SDR_CLK -min -0.8 [get_ports SDR_CSn]
20
21 set_input_delay -clock SDR_CLK -min 2.5 [get_ports SDR_DQ*]
22 set_input_delay -clock SDR_CLK -max 5.4 [get_ports SDR_DQ*]
```

4-121. ábra. *Az SDRAM interfészjeleinek időzítésére vonatkozó előírásokat rögzítő SDC fájl*

az FPGA-oldali viszonyait) úgy alakítsák ki, hogy ha az adatvezetékek az FPGA-n kívül az órajelnél 1,5 ns-mal több késleltetést szenvednek el, akkor az n-edik *SDR_CLK* felfutóél által kiváltott változás éppen egybeessen az (n+1)-edik felfutóéllel. Mivel e késleltetéstöbblet valójában nem lesz jelen a rendszerben, ezért az adatkimenet nem az (n+1)-edik órajel-felfutóél pillanatában, hanem legrosszabb esetben is azelőtt 1,5 ns-mal következik be, pontosan úgy, ahogy az SDRAM adatlapja azt előírja.

A 13–19. sor előírásai ezzel szemben arra utalnak, hogy a legkisebb késleltetéstöbblet, amelyet az adatvezetékek az órajelhez képest elszenvednek az FPGA-n kívül -0,8 ns. A negatív érték furcsának tűnhet, de ne feledjük, hogy az időzítési kényszerek megfogalmazásakor használt adatútmodellben elvileg az *SDR_CLK* jel és az adatkimenetek is szenvedhetnek el késleltetést az FPGA-n kívül, tehát ha az adatvonalak késleltetéstöbblete negatív, az csak annyit jelent, hogy kevesebb késleltetést szenvednek el, mint az *SDR_CLK* jel. Ezzel az időzítési feltétellel tehát azt fejezzük ki, hogy a fizikai szintézis algoritmusoknak úgy kell kialakítaniuk az *SDR_CLK* kimenet és az adatkimenetek időzítési viszonyát, hogy ha az adatvezeték 0,8 ns-mal kevesebb késleltetést szenved el, mint az *SDR_CLK* jel, akkor az adatvezetékek változása az SDRAM oldalán éppen egybeessen az n-edik órajel-felfutóéllel. Mivel az órajel 0,8 ns-os késleltetéstöbblete ezúttal is fiktív, ezért valójában az n-edik órajel-felfutóél után az adatvezetékek még legrosszabb esetben is legalább 0,8 ns-ig változatlanok lesznek, pontosan az adatlap elvárásainak megfelelően.

Az SDC fájl 21–22. soraiban definiált bemenetikésleltetés-előírások azt fejezik ki, hogy az *SDR_DQ* bemenet változását az *SDR_CLK* jel idézi elő úgy, hogy annak felfutása és az *SDR_DQ* bemenet megváltozása között legalább 2,5 ns, de legfeljebb 5,4 ns telik el (ez az SDRAM clock-to-output késleltetése). Az SDRAM áramkörnek valóban időre van szüksége, hogy kimeneteit a megfelelő órajel-felfutóél után beállítsa, tehát ez a késleltetés – ellentétben a kimenetikésleltetés-kényszerekben hivatkozottakkal – tekinthető valódi fizikai késleltetésnek. Az FPGA-n belül tehát az *SDR_CLK* és az *SDR_DQ* bemenetet mintavételező áramköri részleteket a fizikai szintézis algoritmusoknak úgy kell kialakítaniuk, hogy az n-edik *SDR_CLK* felfutóél által kiváltott változás e késleltetés ellenére is hibátlanul mintavételezhető legyen az FPGA-n belül az (n+1)-edik *SDR_CLK* felfutóél pillanatában. Ezt a mintavételt valójában nem közvetlenül az *SDR_CLK* felfutóéle váltja ki,

hiszen az FPGA-n belül a flip-flopokat az interfészadapter saját órajele hajtja meg, de az *SDR_CLK* bemenet és a mintavételező flip-flop órajelbemenetének elcsúszása a fizikai szintézis algoritmusok számára ugyancsak ismert, számításba vehető információ.[138]

4.4.4. I^2C A/D-átalakító

Áttanulmányozásra javasolt alfejezetek:

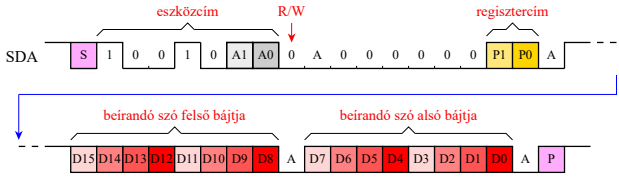
4.3.3. *A soros kommunikáció alapáramkörei – I^2C master*

Ebben az alfejezetben a Texas Instruments ADS1115 típusú kisfogyasztású, nagypontosságú, négycsatornás A/D-átalakítójához készítünk egy egyszerű, de könnyen bővíthető, továbbfejleszthető interfészadaptert. Belső felépítését tekintve az eszköz egy bemeneti multiplexerből, egy programozható előerősítőből, egy 16-bites ΣΔ (szigma-delta) A/D-átalakító magból és egy I^2C interfészlogikából áll. A katalógus-áramkörök többségéhez hasonlóan ez az A/D-átalakító is elég sokoldalú, többféle konfigurációban alkalmazható és sok tulajdonsága programozható (pl. differenciális vagy aszimmetrikus bemeneti mintavétel, előerősítő erősítése, I^2C frekvenciája). Mindezen szolgáltatásokból az itt bemutatott interfészadapter csak a legalapvetőbbeket képezi le úgy, hogy az RTL-modell aránylag könnyen kibővíthető legyen az eszköz további funkcióinak elérése érdekében.

A sok funkciót tartalmazó katalógus-áramkörökre jellemző módon az ADS1115 is csak kétféle tranzakciót, a regiszterírás és -olvasás műveleteit definiálja soros interfészén (vö. 1.2.1.2. pont). A programozható tulajdonságok a konfigurációs regiszterek megfelelő bitjeinek beállításával testreszabhatók, az eszköz állapotát és a mérési adatokat pedig státuszregiszterekből

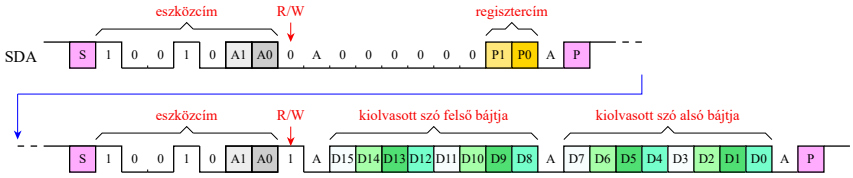
[138] A 4-121. ábra SDC fájlja azt mutatja be, hogy a be- és kimenetikésleltetés-kényszerek hogyan használhatóak egy külső eszköz bemeneteire vonatkozó időzítési előírásoknak és kimeneteire vonatkozó időzítési sajátosságoknak a figyelembevételére úgy, hogy magukat az előírt időzítési kényszereket valójában közvetlenül az FPGA kivezetéseire fogalmazzuk meg. Amikor a kimeneti késleltetés maximumára írunk elő egy feltételt, akkor a kimeneti késleltetés fogalma (lásd DRMSz: 6. fejezet) szerint arról nyilatkozunk, hogy az FPGA-n kívül mekkora késleltetéseket szenved még el az adott jel egy referenciajelhez képest. Amikor az SDRAM adatlapi információit szerepeltetjük az SDC fájlban, akkor a kimenetikésleltetés-kényszerek fogalmát valójában nem arra használjuk, „amire valók". Ebben az esetben nem valódi, fizikai késleltetéskomponensekről van szó, hanem arról, hogy a külső eszköz időzítési előírásait képezzük le fiktív késleltetéskomponensekké. Ez persze nem jelenti azt, hogy a kimenetikésleltetés-kényszerek definíció szerinti szerepüket ne tölthetnék be. Az FPGA és az SDRAM közötti vezetékezésnek is van némi késleltetése, ami pontosan az, aminek modellezésére e kényszerek fogalmát kitalálták. A vezetékezés késleltetései nem feltétlenül ismertek, ilyenkor csak becsléssel vehetjük őket figyelembe. Mivel a kimenetikésleltetés-kényszerek lényege éppen az, hogy figyelembe tudják venni a referencia-órajel és az adatjel késleltetései közötti eltérést, ezért az SDRAM adatlapi értékeinek számszerinti szerepeltetése az SDC fájlban tulajdonképpen arra a feltételezésre utal, hogy a referencia- és adatjelek egyforma késleltetésű vezetékszakaszokon terjednek. Ez a feltételezés általában helyes – vagy legalábbis szolgálhat előírásként a PCB-tervező számára –, de ha mégsem lenne az, akkor az SDC fájlban a megfelelő kényszerek hangolásával a különbségek kompenzálhatók. Ha pl. a 4-121. ábra 8. sorában az *SDR_WEn* kimenetre vonatkozóan nem 1,5 ns-os, hanem 1,7 ns-os kimenetikésleltetés-feltételt írunk elő, azzal azt fejezzük ki, hogy az *SDR_WEn* jel vezetéke 200 ps-mal nagyobb késleltetést okoz, mint az *SDR_CLK* jelé. Természetesen a fordított viszony is megfogalmazható; ha az 1,5 ns helyett 1,2 ns-ot írunk elő, az azt jelenti, hogy az *SDR_WEn* jel vezetéke 300 ps-mal kisebb késleltetésű, mint az *SDR_CLK* jelé. Ez a módszer természetesen a bemenetikésleltetés-kényszerek esetén is használható; ha az SDRAM adatlapi clock-2-output késleltetésnél nagyobb értéket adunk meg bemeneti késleltetésnek az SDC fájlban, azzal kifejezhetjük, hogy az adatvezetékek késleltetése a PCB-n nagyobb, mint a referencia-órajelé, ha kisebbet, akkor pedig fordítva.

olvashatjuk ki. A 4-122. és 4-123. ábrák rendre a regiszterírás és -olvasás műveleteinek I²C-tranzakcióit szemléltetik.



4-122. ábra. *Az ADS1115 A/D-átalakító egy regiszterének írásához szükséges I²C-tranzakció*
*S: **s**tartfeltétel, P: sto**p**feltétel, A: **a**cknowledge*

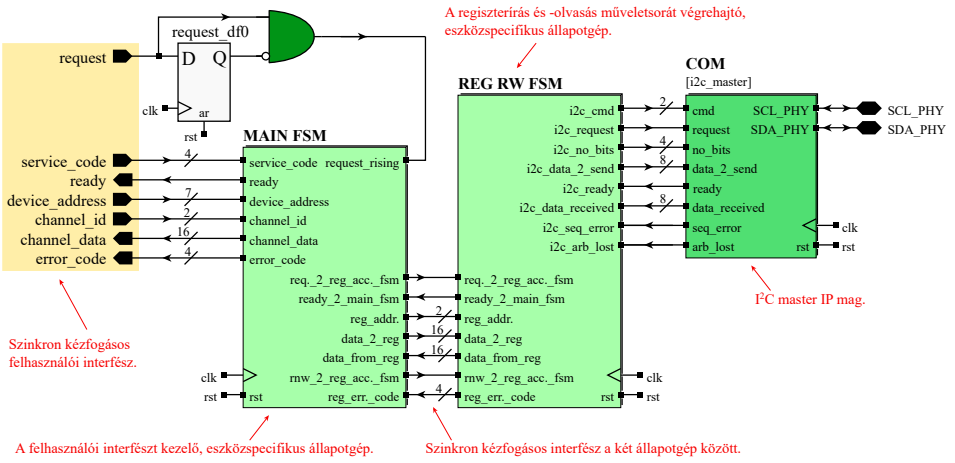
Az eszköz I²C-címe bizonyos mértékben ugyancsak konfigurálható; az *ADDR* kivezetést tápfeszültségre, földpotenciálra, az *SCL* vagy az *SDA* kivezetésekre kapcsolva négyféle cím alakítható ki. A 4-122. és 4-123. ábrákon e négy lehetőséget az I²C-tranzakciók elején, a startfeltétel után elküldött eszközcím A0 és A1 bitjei képviselik. Az eszközcímet tartalmazó bájt alsó bitje (R/W) nulla, jelezve, hogy a következő bájt-átvitelben a master egység küld adatot, nevezetesen a kiválasztott regiszter címét. Összesen négy konfigurációs és állapotregiszter áll rendelkezésünkre, ezek mindegyike 16-bites. A felülírni vagy kiolvasni kívánt regisztert az eszközcím után elküldött bájt alsó két bitje (P0 és P1) jelöli ki. Íráskor a regisztercím elküldését a két beírandó bájt elküldése követi, a műveletsort pedig a stopfeltétel zárja.



4-123. ábra. *Az ADS1115 A/D-átalakító egy regiszterének olvasásához szükséges I²C-tranzakciók*
*S: **s**tartfeltétel, P: sto**p**feltétel, A: **a**cknowledge*

Olvasáskor a regisztert kiválasztó bájt elküldése után – mivel ilyenkor az I²C-buszon az adatáramlás irányát is meg kell fordítani – egy stopfeltételt kell generálnunk. Egy újabb startfeltétel, majd az ezúttal magas szintűre állított R/W bittel elküldött eszközkiválasztó bájt után két bájtot olvasunk be az I²C-buszról, amelyek a megcímzett regiszter felső és alsó felének tartalmát hordozzák. Az olvasás műveletsorát szintén egy stopfeltétel zárja.

A 4-122. és 4-123. ábrákon látható regiszterírás és -olvasás műveletek sorozatával tetszőlegesen bonyolult kölcsönhatások valósíthatók meg az A/D-átalakító és annak használója között. Végrehajthatunk periodikus lekérdezést vagy akár a mérési adatoktól függő adaptív újrakonfigurálást is. E bonyolultságot az interfészadapter felhasználója elől természetesen igyekszünk eltakarni, nem várjuk el, hogy az ezekhez az összetett viselkedésekhez tartozó, adott esetben rengeteg regiszter-hozzáférést igénylő algoritmusokat ő valósítsa meg. Ennek okán – egyúttal a modularitás alapelvének jegyében – az interfészadapterek általános sémája (lásd 4-90. ábra) szerinti szolgáltatásvezérlő funkciót ezúttal két részre bontva kezeljük (lásd 4-124. ábra).



4-124. ábra. *Az ADS1115 A/D-átalakító interfészadapterének szerkezete*[139]

A kommunikációs almodul szerepét a 4.3.3. pontban részletesen bemutatott *i2c_master* IP-mag tölti be. Ez az almodul felel azért, hogy az interfészadapter az I²C-szabványnak megfelelő (lásd 5.2.3. Függelék) tranzakciókat generáljon a buszon. E komponens szolgáltatásait veszi igénybe a *REG RW FSM* állapotgép, amelynek tehát – az EEPROM-ok szolgáltatásvezérlő állapotgépeihez hasonlóan – a soros interfészen végbemenő tranzakciók pontos részleteivel már nem kell foglalkoznia, feladata, hogy a 4-122. és 4-123. ábrákon bemutatott specifikáció alapján megfelelően felparaméterezett kéréseket fogalmazzon meg az *i2c_master* IP-mag felé. Ennek az állapotgépnek nincs tudomása arról, hogy az egyes regisztereknek mi a szerepe az A/D-átalakítón belül. Az őt vezérlő *MAIN FSM* állapotgéptől csak azt az információt kapja meg, hogy a kiválasztott regiszternek mi a címe (P0 és P1) és hogy írás esetén mit kell beleírni, olvasás esetén pedig ugyancsak nem foglalkozik a kiolvasott regiszter tartalmával, azt közvetlenül a *MAIN FSM* állapotgépnek továbbítja

Jel neve	Bitek száma	Jel funkciója
request_2_reg_access_fsm	1	A regiszterírást vagy -olvasást kezdeményező magas-aktív vezérlőjel. A szinkron kézfogásos interfész *request* jele.
ready_2_main_fsm	1	A regiszterírás vagy -olvasás befejeződését jelző magas-aktív nyugtázójel. A szinkron kézfogásos interfész *ready* jele.
reg_address	2	A felülírandó vagy kiolvasandó regiszter címe.
data_2_reg	16	Írás esetén a regiszterbe beírni kívánt új érték.
data_from_reg	16	Olvasás esetén a regiszterből kiolvasott érték.
rnw_2_reg_access_fsm	1	A hozzáférés típusát kijelölő vezérlőbit: olvasás: 1 (R); írás: 0 (nW)
reg_error_code	4	A *REG RW FSM* állapotgéptől származó hibakód.

4-8. táblázat. *A regiszter-hozzáférést és a felhasználói interfészt megvalósító állapotgépek közötti kommunikáció jelei*

[139] A nem memória jellegű funkciót ellátó interfészadapterek esetén csak egyetlen felhasználói interfészt hozunk létre. Több felhasználói interfész kialakítása, és azok prioritásos kiszolgálásának megvalósítása a 4.1.2. pontban bemutatott prioritásos hozzáférés-vezérlő beépítésével lehetséges.

további feldolgozásra. A két állapotgép között ezeknek az információknak az átadására – a memóriaillesztő IFA-k hozzáférés-vezérlőiénél látottakhoz hasonlóan – egy szinkron kézfogásos interfészt alakítunk ki, ennek jeleit és azok szerepét a 4-8. táblázat foglalja össze.

A *MAIN FSM* állapotgép feladata a szintén szinkron kézfogásos felhasználói interfész működtetése, vagyis az ott érkező kérések regiszterírások és -olvasások sorozatává alakítása. Az interfészadapter felhasználói interfészén ebben a példában csupán egyetlen szolgáltatást – egy A/D-konverzió végrehajtását és eredményének lekérdezését – ajánlunk ki, de magát az interfészt és az azt működtető állapotgépet úgy készítjük el, hogy az interfészadapter szolgáltatásainak körét igény szerint utólag könnyen bővíthessük. A 4-125. ábra a regiszterírások és -olvasások lebonyolításáért felelős állapotgép vázlatát mutatja be.

VHDL

```
L_REG_RW_FSM: process ( clk, rst ) begin
 if ( rst = '1' ) then
  reg_state <= idle; reg_return_state <= write_2;
  i2c_request <= '0'; i2c_cmd <= "00"; i2c_no_bits <= 1; i2c_data_2_send <= X"00";
  reg_error_code <= (others => '0'); ready_2_main_fsm <= '1'; data_from_reg <= X"0000";
 elsif ( rising_edge(clk) ) then

  case reg_state is

   when idle => ready_2_main_fsm <= '1';
                if ( request_2_reg_access_fsm = '1' ) then
                 ready_2_main_fsm <= '0'; reg_error_code <= X"0";

                 if ( rnw_2_reg_access_fsm = '0' ) then reg_state <= write_1;
                 else reg_state <= read_1;
                 end if;

                end if;

   [...]

   when others => null;

  end case;

 end if;
end process;
```

4-125. ábra. *A regiszterírások és -olvasások végrehajtását végző REG RW FSM állapotgép vázlata*

A *REG RW FSM* állapotgép az *i2c_master* modulon keresztül, annak felhasználói interfészét igénybe véve kezdeményez különféle tranzakciókat az I²C-buszon. Az *i2c_master* modulnak szóló, megfelelően felparaméterezett kérések elküldését, a művelet befejezésének kivárását és az esetleges buszhibák lekezelését a regiszterírás és -olvasás algoritmusai során sokszor végre kell hajtani, ezért az *i2c_master* modullal való kommunikációt egy szubrutinszerű állapotsorozatba szervezzük ki (lásd 4-126. ábra).

VHDL

```
when i2c_master_request_1 => i2c_request <= '1'; state <= i2c_master_request_2;

when i2c_master_request_2 => i2c_request <= '0'; state <= i2c_master_request_3;

when i2c_master_request_3 => if ( i2c_ready = '1' ) then
                              if ( i2c_seq_error = '1' or i2c_arb_lost = '1' ) then
                               i2c_status <= X"1"; state <= idle;
                              else state <= return_state;
                              end if;
                             end if;
```

4-126. ábra. *Szubrutinszerű állapotsorozat az I²C master IP-val való kommunikáció megvalósítására*

Ezek után nincs más dolgunk, mint hogy létrehozzuk a 4-122. és 4-123. ábráknak megfelelő I²C-busztranzakciókat megvalósító vezérlési állapotokat. Az írás algoritmusához tartozó állapotokat a 4-127., az olvasást végzőket pedig a 4-128. ábra mutatja be.

Figyeljük meg a regularitás alapelvének érvényesülését ebben az esetben is. Az ADS1115 A/D-átalakító interfészadaptere ugyanazt a soros kommunikációs almodult példányosítja, mint amit

VHDL

```
when write_1 => -- send start-condition
  i2c_cmd <= "00";
  reg_return_state <= write_2;
  reg_state <= i2c_master_request_1;

when write_2 => -- send I2C address & nW bit
  i2c_cmd <= "01";
  i2c_data_2_send <= device_address & '0';
  i2c_no_bits <= 8;
  reg_return_state <= write_3;
  reg_state <= i2c_master_request_1;

when write_3 => -- receive ACK
  i2c_cmd <= "10";
  i2c_no_bits <= 1;
  reg_return_state <= write_4;
  reg_state <= i2c_master_request_1;

when write_4 =>
  if ( i2c_data_received(0) = '0' ) then
  | -- ACK received
  | i2c_cmd <= "01"; -- send address pointer
  | i2c_data_2_send <= "000000" & reg_address;
  | i2c_no_bits <= 8;
  | reg_return_state <= write_5;
  | reg_state <= i2c_master_request_1;
  else -- no ACK received
    reg_error_code <= X"2";
    reg_state <= idle;
  end if;

when write_5 => -- receive ACK
  i2c_cmd <= "10";
  i2c_no_bits <= 1;
  reg_return_state <= write_6;
  reg_state <= i2c_master_request_1;

when write_6 =>
  if ( i2c_data_received(0) = '0' ) then
    -- ACK received
    reg_state <= write_7;
  else -- no ACK received
    reg_error_code <= X"2";
    reg_state <= idle;
  end if;

```

VHDL (folyt.)

```
when write_7 => -- send register data [15:8]
  i2c_cmd <= "01";
  i2c_data_2_send <= data_2_reg(15 downto 8);
  i2c_no_bits <= 8;
  reg_return_state <= write_8;
  reg_state <= i2c_master_request_1;

when write_8 => -- receive ACK
  i2c_cmd <= "10";
  i2c_no_bits <= 1;
  reg_return_state <= write_9;
  reg_state <= i2c_master_request_1;

when write_9 =>
  if ( i2c_data_received(0) = '0' ) then
    -- ACK received
    reg_state <= write_10;
  else -- no ACK received
    reg_error_code <= X"2";
    reg_state <= idle;
  end if;

when write_10 => -- send register data[7:0]
  i2c_cmd <= "01";
  i2c_data_2_send <= data_2_reg(7 downto 0);
  i2c_no_bits <= 8;
  reg_return_state <= write_11;
  reg_state <= i2c_master_request_1;

when write_11 => -- receive ACK
  i2c_cmd <= "10";
  i2c_no_bits <= 1;
  reg_return_state <= write_12;
  reg_state <= i2c_master_request_1;

when write_12 =>
  if ( i2c_data_received(0) = '0' ) then
    -- ACK received
    reg_state <= write_13;
  else -- no ACK received
    reg_error_code <= X"2";
    reg_state <= idle;
  end if;

when write_13 => -- send stop-condition
  i2c_cmd <= "11";
  reg_return_state <= idle;
  reg_state <= i2c_master_request_1;
```

4-127. ábra. *Az ADS1115 A/D-átalakító egy regiszterének írását végző vezérlési állapotok*

VHDL

```
when read_1 => -- send start-condition
  i2c_cmd <= "00";
  reg_return_state <= read_2;
  reg_state <= i2c_master_request_1;

when read_2 => -- send I2C address & nW bit
  i2c_cmd <= "01";
  i2c_data_2_send <= device_address & '0';
  i2c_no_bits <= 8;
  reg_return_state <= read_3;
  reg_state <= i2c_master_request_1;

when read_3 => -- receive ACK
  i2c_cmd <= "10";
  i2c_no_bits <= 1;
  reg_return_state <= read_4;
  reg_state <= i2c_master_request_1;

when read_4 =>
  if ( i2c_data_received(0) = '0' ) then
    -- ACK received
    i2c_cmd <= "01";  -- send address pointer
    i2c_data_2_send <= "000000" & reg_address;
    i2c_no_bits <= 8;
    reg_return_state <= read_5;
    reg_state <= i2c_master_request_1;
  else -- no ACK received
    reg_error_code <= X"2";
    reg_state <= idle;
  end if;

when read_5 => -- receive ACK
  i2c_cmd <= "10";
  i2c_no_bits <= 1;
  reg_return_state <= read_6;
  reg_state <= i2c_master_request_1;

when read_6 =>
  if ( i2c_data_received(0) = '0' ) then
    -- ACK received
    reg_state <= read_7;
  else -- no ACK received
    reg_error_code <= X"2";
    reg_state <= idle;
  end if;

when read_7 => -- send stop-condition
  i2c_cmd <= "11";
  reg_return_state <= read_8;
  reg_state <= i2c_master_request_1;

```

VHDL (folyt.)

```
when read_8 => -- send start-condition
  i2c_cmd <= "00";
  reg_return_state <= read_9;
  reg_state <= i2c_master_request_1;

when read_9 => -- send I2C address & R bit
  i2c_cmd <= "01";
  i2c_data_2_send <= device_address & '1';
  i2c_no_bits <= 8;
  reg_return_state <= read_10;
  reg_state <= i2c_master_request_1;

when read_10 => -- receive ACK
  i2c_cmd <= "10";
  i2c_no_bits <= 1;
  reg_return_state <= read_11;
  reg_state <= i2c_master_request_1;

when read_11 =>
  if ( i2c_data_received(0) = '0' ) then
    -- ACK received
    i2c_cmd <= "10"; -- receive reg. data [15:8]
    i2c_no_bits <= 8;
    reg_return_state <= read_12;
    reg_state <= i2c_master_request_1;
  else -- no ACK received
    reg_error_code <= X"2";
    reg_state <= idle;
  end if;

when read_12 =>
  data_from_reg(15 downto 8) <= i2c_data_received;

  i2c_cmd <= "01"; -- send ACK
  i2c_data_2_send <= X"00";
  i2c_no_bits <= 1;
  reg_return_state <= read_13;
  reg_state <= i2c_master_request_1;

when read_13 => -- receive reg. data [7:0]
  i2c_cmd <= "10";
  i2c_no_bits <= 8;
  reg_return_state <= read_14;
  reg_state <= i2c_master_request_1;

when read_14 =>
  data_from_reg(7 downto 0) <= i2c_data_received;

  i2c_cmd <= "01"; -- send ACK
  i2c_data_2_send <= X"00";
  i2c_no_bits <= 1;
  reg_return_state <= read_15;
  reg_state <= i2c_master_request_1;

when read_15 => -- send stop-condition
  i2c_cmd <= "11";
  reg_return_state <= idle;
  reg_state <= i2c_master_request_1;
```

4-128. ábra. *Az ADS1115 A/D-átalakító egy regiszterének olvasását végző vezérlési állapotok*

az I²C EEPROM interfészadapterébe is beépítettünk, ráadásul az ezt az almodult vezérlő állapotgépek is egyforma szerkezetűek, sőt, a kommunikációs modult vezérlő állapotaik teljesen azonosak. Ennek következtében a 4-127. és 4-128. ábrákon látható vezérlési állapotok kísértetiesen hasonlítanak az I²C EEPROM szolgáltatásvezérlő állapotgépének vezérlési állapotaira (vö. 4-106. és 4-107. ábra). Az állapotok száma és az *i2c_master* modulnak szóló kérések felparaméterezése az adott katalógus-áramkör interfész-specifikációjának sajátosságait tükrözik.[140]

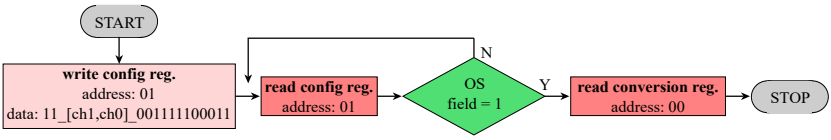
A regiszterírások és -olvasások részleteit tehát a *REG RW FSM* állapotgép eltakarja, a *MAIN FSM* állapotgép létrehozásakor így aránylag könnyű dolgunk van; az A/D-átalakító felhasználói interfészére kiajánlott szolgáltatásokat regiszterírás és -olvasás műveletekből álló algoritmusokként kell megfogalmaznunk. Az állapotgép VHDL-leírásának vázlatát a 4-129. ábra szemlélteti.

VHDL

```
1  L_MAIN_FSM: process ( clk, rst ) begin
2  | if ( rst = '1' ) then
3  | | main_state <= idle; main_return_state <= get_channel_data_2; request_2_reg_access_fsm <= '0';
4  | | reg_address <= "00"; data_2_reg <= X"0000"; rnw_2_reg_access_fsm <= '0';
5  | | error_code <= X"0"; ready <= '1'; channel_data <= (others => '0');
6  | elsif ( rising_edge(clk) ) then
7  | |
8  | | case main_state is
9  | | |
10 | | | when idle => ready <= '1';
11 | | |              if ( request_rising = '1' ) then
12 | | |              | ready <= '0'; error_code <= X"0";
13 | | |              |
14 | | |              | case service_code is
15 | | |              | | when X"0" => main_state <= get_channel_data_1;
16 | | |              | | [...]
17 | | |              | | when others => null;
18 | | |              | end case;
19 | | |              |
20 | | |              end if;
21 | | |
22 | | | [...]
23 | | |
24 | | | when others => null;
25 | | |
26 | | end case;
27 | |
28 | end if;
29 end process;
```

4-129. ábra. *A felhasználói interfész szolgáltatásait megvalósító állapotgép vázlata*

[140] Az I²C katalógus-áramkörök nemcsak I²C jellegük miatt hasonlóak, általában regisztereik írásának és olvasásának módja is csak apróságokban tér el. Ha összevetjük a 24LC16B EEPROM és az ADS1115 A/D-átalakító interfész-műveleteinek specifikációit, akkor azt látjuk, hogy a beírás csupán a beírt bájtok számában különbözik – előbbi esetben 1, utóbbinál 2. Egyetlen memóriarekesz vagy regiszter kiolvasásakor pedig a két I²C-tranzakció a kiolvasott adat méretén felül csak abban különbözik, hogy az adatáramlás irányának váltása előtt az A/D-átalakító esetén egy stopfeltétel + startfeltétel párost kell generálnunk, míg az EEPROM ehelyett megelégszik egy megismételt startfeltétellel, továbbá abban, hogy az olvasás műveletet az EEPROM-nál egy NACK (not acknowledge, *SDA* magas szintű), az A/D-átalakítónál pedig egy ACK (acknowledge, *SDA* alacsony szintű) jelzéssel kell lezárnunk. Az alapvetően nagyon hasonló működés arra sarkallhat, hogy a regiszterírás és -olvasás műveletsorait az *i2c_master* IP-magon keresztül végrehajtó funkcionális egységet (az A/D-átalakítónál a *REG RW FSM,* az EEPROM-nál az *SRV CTRL FSM* nevű állapotgépet) a kommunikációs almodulokhoz hasonlóan újrafelhasználható IP-ként hozzuk létre. Ennek valóban nincs akadálya, a fent említett apró eltérések a katalógus-áramkörök között jellemzően nem jelentősek, egy újrafelhasználható IP készítésekor a szintézisparaméter-készlet körültekintő megtervezésével jól kezelhetők. E megoldás nyilvánvaló előnye, hogy minden I²C-alapú interfészadapterbe beépíthető lenne, hátránya ugyanakkor, hogy nagyon egyszerű katalógus-áramkörök esetén (lásd pl. 4.4.5. pont: I²C hőmérséklet-érzékelő) feleslegesen növelné az IFA belső komplexitását és közvetetten az erőforrás-felhasználását, illetve, hogy nem tenné lehetővé a blokkos adatátvitelt, amivel egyetlen I²C-tranzakción belül több regisztert is írhatnánk vagy olvashatnánk, növelve a kommunikáció hatékonyságát sebesség és energiafogyasztás szempontjából egyaránt (lásd 4.4.6. pont: I²C RTCC áramkör).



4-130. ábra. *Egy kiválasztott csatorna lekérdezése az ADS1115 A/D-átalakítóból*

Felhasználói szempontból egy A/D-átalakítóval kapcsolatban kézenfekvő szolgáltatás egy kiválasztott csatornán az A/D-konverzió elvégzése és a mintavett analóg jel digitális reprezentációjának lekérdezése. Ez az ADS1115 A/D-átalakítónál a 4-130. ábrán látható algoritmus szerint végezhető el. Már egy ilyen egyszerű algoritmus is több regiszter-hozzáférést igényel, a bonyolultabb szolgáltatások megvalósításához pedig még többre lehet szükség, ezért a *REG RW FSM* állapotgéppel való kommunikációt itt is érdemes egy szubrutinszerű állapotsorozatba kiszervezni, ahogy tettük azt a *REG RW FSM* állapotgép belsejében az *i2c_master* IP-maggal való kölcsönhatások leírásakor. Ezt a szubrutinszerű állapotsorozatot a 4-131. ábra mutatja be.

VHDL

```
when call_reg_access_fsm_1 => request_2_reg_access_fsm <= '1';
                              main_state <= call_reg_access_fsm_2;

when call_reg_access_fsm_2 => request_2_reg_access_fsm <= '0';
                              main_state <= call_reg_access_fsm_3;

when call_reg_access_fsm_3 => if ( ready_2_main_fsm = '1' ) then
                                if ( reg_error_code = X"0" ) then main_state <= main_return_state;
                                else
                                  error_code <= reg_error_code;
                                  main_state <= idle;
                                end if;
                              end if;
```

4-131. ábra. *Szubrutinszerű állapotsorozat a REG RW FSM állapotgéppel való kommunikáció megvalósítására*

A 4-130. ábrán látható algoritmus megvalósítása ezek után már könnyen elvégezhető; a benne foglalt regiszterírás és -olvasás műveletek végrehajtásához nincs más feladatunk, mint hogy a *REG RW FSM* állapotgép felhasználói interfészét megfelelően felparaméterezzük, majd az állapotgépet a *call_reg_access_fsm_1* állapotba irányítsuk (lásd 4-132. ábra).

VHDL

```
when get_channel_data_1 => -- set config reg.:
  data_2_reg(15) <= '1'; -- start conversion
  data_2_reg(14 downto 12) <= '1' & channel_id;
  data_2_reg(11 downto 9) <= "001"; -- FSR: +/- 4.096V
  data_2_reg(8) <= '1'; -- single-shot mode
  data_2_reg(7 downto 5) <= "111"; -- DR: 860 SPS
  data_2_reg(4 downto 0) <= "00011"; -- irrelevant

  reg_address <= "01"; -- config reg.
  rnw_2_reg_access_fsm <= '0'; -- write
  main_state <= call_reg_access_fsm_1;
  main_return_state <= get_channel_data_2;

when get_channel_data_2 =>
  -- polling the OS field of the config reg.:
  reg_address <= "01";  -- config reg.
  rnw_2_reg_access_fsm <= '1'; -- read
  main_state <= call_reg_access_fsm_1;
  main_return_state <= get_channel_data_3;

```

VHDL (folyt.)

```
when get_channel_data_3 =>
  if ( data_from_reg(15) = '1' ) then
    -- conversion done
    main_state <= get_channel_data_4;
  else
    -- conversion in progress
    main_state <= get_channel_data_2;
  end if;

when get_channel_data_4 =>
  -- read conversion reg.
  reg_address <= "00"; -- conversion reg.
  rnw_2_reg_access_fsm <= '1'; -- read
  main_state <= call_reg_access_fsm_1;
  main_return_state <= get_channel_data_5;

when get_channel_data_5 =>
  channel_data <= data_from_reg;
  main_state <= idle;
```

4-132. ábra. *Egy adott csatorna lekérdezésének algoritmusát megvalósító vezérlési állapotok*

A *get_channel_data_1* állapotban kezdeményezzük az eszköz konfigurációs regiszterének írását. A konfigurációs regiszter egyes mezőinek szerepét az A/D-átalakító adatlapjából deríthetjük ki:

- [15]: magas szintre állítása kezdeményezi magát a konverziót
- [14]: a [13:12] bitek által kijelölt analóg csatorna potenciálját a földpotenciálhoz viszonyítjuk
- [13:12]: analóg bemeneti csatorna kiválasztása
- [11:9]: előerősítő beállítása; az analóg bemenetek mérési tartományának meghatározása
- [8]: single-shot üzemmód kiválasztása; egyetlen konverzió végrehajtása
- [7:5]: mintavételi frekvencia kiválasztása (single-shot módban irreleváns)
- [4:0]: egyéb, az IFA által nem leképezett képességekre vonatkozó beállítások

A konverzió indítása után annak állapotáról ugyanennek a regiszternek a 15. helyiértékű bitjét lekérdezve tájékozódhatunk; az indítás után 25 µs-mal – tehát jóval az előtt, hogy a konfigurációs regisztert ki tudnánk olvasni – az eszköz nulla szintre húzza a 15. bitet, majd a konverzió végén állítja azt újra magas szintre. A konfigurációs regiszter periodikus lekérdezését és a 15. helyiértékű bit értékének vizsgálatát a *get_channel_data_2/3* állapotok végzik. A konverzió befejezésekor a kiválasztott analóg bemenet digitalizált értékét a konverziós regiszterből olvashatjuk ki (*get_channel_data_4* állapot). A kiolvasott adatot az IFA felhasználói interfészére továbbítjuk (4-132. ábra: 38. sor), majd az *idle* állapotba visszatérve a *ready* jel magas szintre állításával (4-129. ábra: 10. sor) zárjuk a felhasználói kérés kiszolgálását.

Az adott csatorna lekérdezésén túlmutató, összetettebb felhasználói szolgáltatások algoritmusai a 4-132. ábrán láthatóhoz hasonlóan hozhatók létre, újabb vezérlésiállapot-sorozatok hozzáadásával.

4.4.4. I²C hőmérséklet-érzékelő

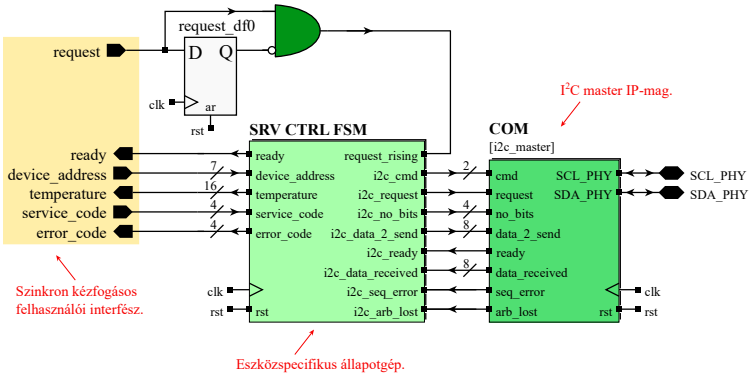
Áttanulmányozásra javasolt alfejezetek:

4.3.3. *A soros kommunikáció alapáramkörei – I²C master*

A Microchip Technology Inc. által gyártott, TCN75A típusjelű katalógus-áramkör egy egyszerű, I²C-interfésszel szerelt hőmérséklet-érzékelő. Az eszköz egyszerűsége ellenére biztosít néhány konfigurációs lehetőséget (felbontás, kisfogyasztású üzemmód stb.), ezeket azonban az itt bemutatott példában nem használjuk ki. A chip I²C-címe három kivezetés segítségével testreszabható. Ha minden konfigurációs lehetőséget mellőzve, kizárólag az alapértelmezett beállításokkal szeretnénk használni az eszközt, akkor a hozzá tervezett interfészadapternek tulajdonképpen egyetlen fajta I²C-tranzakció levezénylésére kell képesnek lennie, amely az aktuális hőmérsékletet reprezentáló 16-bites regiszter kiolvasását jelenti (lásd 4-133. ábra).

4-133. ábra. *Az TCN75A hőmérséklet-érzékelő mérési adatának kiolvasásához szükséges I²C-tranzakció*
*S: **s**tartfeltétel, P: sto**p**feltétel, (N)A: (**n**ot) **a**cknowledge*

A szenzorhoz készített interfészadapter tervezésekor tehát ezúttal – az I²C-s A/D-átalakítónál látottakkal ellentétben – a lehető legegyszerűbb felépítésre törekszünk. A felhasználói interfész kezelését, valamint a regiszterírás és -olvasás absztrakt funkcióját ezúttal egyetlen állapotgépbe sűrítve fogalmazzuk meg. Az IFA szerkezetét a 4-134. ábra szemlélteti.



4-134. ábra. *A TCN75A hőmérséklet-érzékelő interfészadapterének szerkezete*

A kommunikációs almodul szerepét ezúttal is az *i2c_master* IP-mag tölti be, ezt veszi igénybe a szolgáltatásvezérlő állapotgép a felhasználói interfészen érkező kérések kiszolgálásához. Maga a felhasználói interfész szinkron kézfogásos típusú, és bár csupán egyetlen szolgáltatás megvalósítását tűztük ki célul, az egységesség kedvéért megőrizzük a négybites szolgáltatásazonosító bemenetet *(service_code)*, amelynek segítségével később könnyen bővíthetjük az RTL-modellt újabb szolgáltatások hozzáadásával. A szolgáltatásvezérlő állapotgép vázlata a 4-135. ábrán látható.

VHDL

```
1  L_SRV_CTRL_FSM: process ( clk, rst ) begin
2  | if ( rst = '1' ) then
3  | | state <= idle; return_state <= read_temp_2;
4  | | i2c_cmd <= "00"; i2c_request <= '0'; i2c_no_bits <= 1; i2c_data_2_send <= X"00";
5  | | error_code <= X"0"; ready <= '1'; temperature <= (others => '0');
6  | elsif ( rising_edge(clk) ) then
7  | |
8  | | case state is
9  | | |
10 | | | when idle => ready <= '1';
11 | | |              if ( request_rising = '1' ) then
12 | | |                ready <= '0'; error_code <= X"0";
13 | | |                |
14 | | |                case service_code is
15 | | |                | when X"0" => state <= read_temp_1;
16 | | |                | [...]
17 | | |                | when others => null;
18 | | |                end case;
19 | | |                |
20 | | |              end if;
21 | | |
22 | | | [...]
23 | | |
24 | | | when others => null;
25 | | |
26 | | end case;
27 | |
28 | end if;
29 end process;
```

4-135. ábra. *A TCN75A IFA szolgáltatásvezérlő állapotgépének vázlata*

A 4-133. ábrán látható I²C-tranzakció megvalósítása érdekében többször kell az *i2c_master* almodulhoz fordulnunk, ezért a vele való kommunikációt – a korábbi példáknál látottakkal teljesen azonos módon – ezúttal is egy szubrutinszerű állapotsorozat segítségével valósítjuk meg (lásd 4-136. ábra).

VHDL

```
when i2c_master_request_1 => i2c_request <= '1'; state <= i2c_master_request_2;

when i2c_master_request_2 => i2c_request <= '0'; state <= i2c_master_request_3;

when i2c_master_request_3 => if ( i2c_ready = '1' ) then
                               if ( i2c_seq_error = '1' or i2c_arb_lost = '1' ) then
                                 i2c_status <= X"1"; state <= idle;
                               else state <= return_state;
                               end if;
                             end if;
```

4-136. ábra. *Szubrutinszerű állapotsorozat az I²C master IP-val való kommunikáció megvalósítására*

Összehasonlítva az I²C-s A/D-átalakító kiolvasását ábrázoló 4-123. ábrát a hőmérséklet-érzékelő lekérdezését bemutató 4-133. ábrával azt vehetjük észre, hogy ez utóbbi folyamat csaknem teljesen azonos az A/D-átalakító kiolvasásának második startfeltételt követő szakaszával, különbség csupán az eszköz megcímzésében és a tranzakció lezárásában van (ACK vs. NACK). Ahogy korábban említettük, az I²C-interfészű katalógus-áramkörök használata jobbára szabványos, így e hasonlóság nem meglepő. Az A/D-átalakító kiolvasásának a második startfeltételt megelőző szakasza a kiolvasandó regiszter címének beállítását szolgálja. A hőmérséklet-érzékelő esetén ezt a lépést megspórolhatjuk, kihasználva, hogy a belső címregiszter alapértelmezett értéke 0x00, ami épp a hőmérsékletmérés mindenkori legfrissebb eredményét tároló regisztert jelöli. Az eszközök interfészén tapasztalható viselkedés hasonlóságából adódóan a hőmérséklet-érzékelő IFA szolgáltatásvezérlő állapotgépének vezérlési állapotai (lásd 4-137. ábra) közel azonosak az A/D-átalakító IFA *REG RW FSM* nevű állapotgépének *read_8...15* állapotaival (lásd 4-128. ábra: 52–109. sor).

VHDL

```
when read_temp_1 => -- send start-condition
  i2c_cmd <= "00";
  return_state <= read_temp_2;
  state <= i2c_master_request_1;

when read_temp_2 => -- send I2C address & R bit
  i2c_cmd <= "01";
  i2c_data_2_send <= device_address & '1';
  i2c_no_bits <= 8;
  return_state <= read_temp_3;
  state <= i2c_master_request_1;

when read_temp_3 => -- receive ACK
  i2c_cmd <= "10";
  i2c_no_bits <= 1;
  return_state <= read_temp_4;
  state <= i2c_master_request_1;

when read_temp_4 =>
  if ( i2c_data_received(0) = '0' ) then
    -- ACK received
    i2c_cmd <= "10"; -- receive reg. data [15:8]
    i2c_no_bits <= 8;
    return_state <= read_temp_5;
    state <= i2c_master_request_1;
  else-- no ACK received
    error_code <= X"2";
    state <= idle;
  end if;

```

VHDL (folyt.)

```
when read_temp_5 =>
  temperature(15 downto 8) <= i2c_data_received;

  i2c_cmd <= "01";    -- send ACK
  i2c_data_2_send <= X"00";
  i2c_no_bits <= 1;
  return_state <= read_temp_6;
  state <= i2c_master_request_1;

when read_temp_6 => -- receive reg. data [7:0]
  i2c_cmd <= "10";
  i2c_no_bits <= 8;
  return_state <= read_temp_7;
  state <= i2c_master_request_1;

when read_temp_7 =>
  temperature(7 downto 0) <= i2c_data_received;

  i2c_cmd <= "01"; -- send NACK
  i2c_data_2_send <= X"01";
  i2c_no_bits <= 1;
  return_state <= read_temp_8;
  state <= i2c_master_request_1;

when read_temp_8 => -- send stop-condition
  i2c_cmd <= "11";
  return_state <= idle;
  state <= i2c_master_request_1;
```

4-137. ábra. *A TCN75A hőmérséklet-érzékelő mérési adatát kiolvasó vezérlési állapotok*

4.4.5. I²C RTCC áramkör

Áttanulmányozásra javasolt alfejezetek:

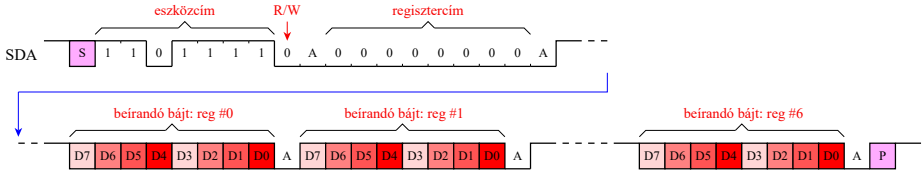
4.3.3. *A soros kommunikáció alapáramkörei – I²C master*

A Microchip gyártmányú MCP79410 típusjelű katalógus-áramkör egy ún. RTCC (**R**eal-**T**ime **C**lock/**C**alendar), vagyis egy olyan eszköz, amely hosszú távon képes nagy pontossággal mérni az idő múlását és azt dátum-óra-perc-másodperc, vagyis naptár-óra formátumban szolgáltatja.[141] Az I²C-interfészen konfigurálható és kiolvasható eszköz regiszterei közül az első héttel foglalkozunk. Ezek azok a regiszterek, amelyek a mindenkori dátumot és pontos időt tárolják binárisan kódolt decimális (**B**inary-**C**oded **D**ecimal, BCD) formátumban (lásd 4-9. táblázat).

Regiszter címe	Regiszter tartalma
0	Másodperc-számláló
1	Perc-számláló
2	Óra-számláló
3	Héten belüli nap-számláló
4	Hónapon belüli nap-számláló
5	Hónap-számláló
6	Év-számláló

4-9. táblázat. *Az MCP79410 RTCC áramkör első hét regisztere*

E regiszterek természetesen írhatók is, hiszen tápfeszültség hiányában a chip nem képes működtetni belső számlálóit, vagyis a felhasználónak minden bekapcsoláskor frissítenie kell a regiszterekben tárolt dátumot és időt.[142] A regiszterek a korábbiakhoz hasonló I²C-műveletekkel is elérhetők, de ezúttal kihasználjuk az eszköz I²C slave interfész-áramkörének az automatikus címinkrementálási képességét; a hét darab, számunkra érdekes regiszter frissítését és kiolvasását egy-egy, nyolc adatkeretet tartalmazó I²C-tranzakcióval végezzük el. A beíráshoz és kiolvasáshoz tartozó I²C-tranzakciókat rendre a 4-138. és 4-139. ábrák mutatják be.



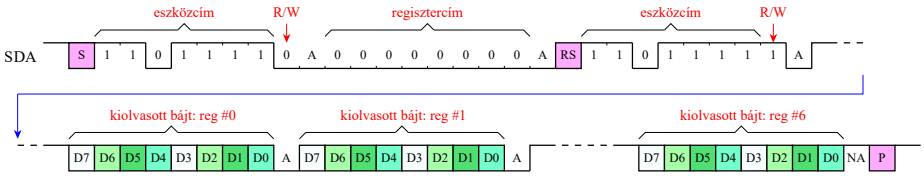
4-138. ábra. *Az MCP79410 RTCC első hét regiszterének írásához szükséges I²C-tranzakció időzítése*
*S: **s**tartfeltétel, P: sto**p**feltétel, A: **a**cknowledge*

Az eszköz I²C-címét tartalmazó címzőkeret elküldése után íráskor és olvasáskor is csak egyetlen olyan adatkeret elküldésére kerül sor, amelyben regisztercímet adunk át az eszköznek. Ezt a címet

[141] Az RTCC funkción felül az áramkör tartalmaz egy kiskapacitású, SRAM típusú memóriablokkot is, ezt azonban az itt bemutatott interfészadapter nem teszi hozzáférhetővé.

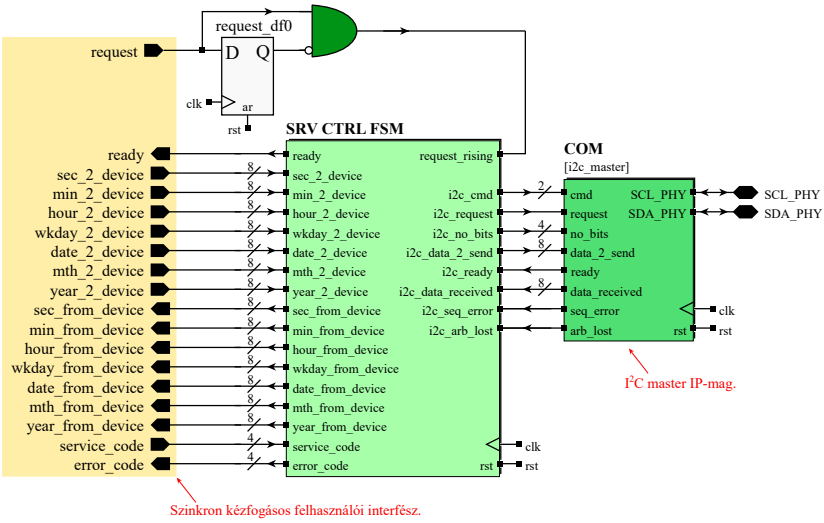
[142] Az eszköz akkumulátorról vagy elemről is üzemeltethető, ekkor a benne lévő számlálók követik az idő múlását, az I²C-interfész azonban ilyenkor inaktív.

az eszköz egy kezdőcímként tárolja el, amit a további adatkeretek elküldése vagy fogadása után automatikusan inkrementál. Ezzel a megoldással minimalizálhatjuk az I²C-busz forgalmát.



4-139. ábra. *Az MCP79410 RTCC első hét regiszterének olvasásához szükséges I²C-tranzakció időzítése*
*S: **s**tartfeltétel, RS (**r**epeated **s**tart): megismételt startfeltétel, P: sto**p**feltétel, (N)A: (**n**ot) **a**cknowledge*

Az interfészadapter felépítése (lásd 4-140. ábra) a hőmérséklet-érzékelőhöz tartozóéval megegyező; az I²C-busztranzakciókkal kapcsolatos feladatokat az *i2c_master* IP-mag látja el, a szinkron kézfogásos felhasználói interfészen érkező kéréseket pedig egy állapotgép értelmezi és „fordítja le” az *i2c_master* modulnak szóló kérésekre.



4-140. ábra. *Az MCP79410 RTCC áramkör interfészadapterének szerkezete*

Az IFA felhasználói interfészén ezúttal két szolgáltatást ajánlunk ki; a *service_code* bemenet 01_2 kombinációjával a felhasználó beállíthatja, a 00_2 kombinációval pedig kiolvashatja az RTCC áramkör regisztereit. A hét darab regiszter felhasználói szempontból tehát csak „csomagban”, egyszerre írható vagy olvasható.[143] A szolgáltatásvezérlő állapotgép vázlatát a 4-141. ábra szemlélteti.

A 4-138. és 4-139. ábrákon látható I²C-tranzakciók végrehajtásához a szolgáltatásvezérlő állapotgépnek meglehetősen sokszor kell az *i2c_master* modulhoz fordulnia, így a két részegység közötti kommunikációt – a korábbi példák alapján talán nem meglepő módon – ezúttal is egy szubrutinszerű állapotsorozatba szervezzük (lásd 4-142. ábra).

[143] Az RTCC áramkör – az A/D-átalakítóval és a hőmérséklet-érzékelővel ellentétben – nem teszi lehetővé I²C-címének kivezetéseken keresztüli konfigurálását, ezért az IFA felhasználói interfészéből hiányzik a *device_address* bemenet.

VHDL

```
L_SRV_CTRL_FSM: process ( clk, rst ) begin
 if ( rst = '1' ) then
  state <= idle; return_state <= read_2; cnt <= 0; ready <= '1'; error_code <= X"0";
  i2c_cmd <= "00"; i2c_request <= '0'; i2c_no_bits <= 1; i2c_data_2_send <= X"00";
  sec_from_device <= X"00"; min_from_device <= X"00"; hour_from_device <= X"00";
  wkday_from_device <= X"00"; date_from_device <= X"00"; mth_from_device <= X"00";
  year_from_device <= X"00";
 elsif ( rising_edge(clk) ) then

  case state is

   when idle => ready <= '1';
                if ( request_rising = '1' ) then
                 ready <= '0'; error_code <= X"0";

                 case service_code is
                  when X"0" => state <= read_1;
                  when X"1" => state <= write_1;
                  [...]
                  when others => null;
                 end case;

                end if;

   [...]

   when others => null;

  end case;

 end if;
end process;
```

4-141. ábra. *Az MCP79410 IFA szolgáltatásvezérlő állapotgépének vázlata*

VHDL

```
when i2c_master_request_1 => i2c_request <= '1'; state <= i2c_master_request_2;

when i2c_master_request_2 => i2c_request <= '0'; state <= i2c_master_request_3;

when i2c_master_request_3 => if ( i2c_ready = '1' ) then
                              if ( i2c_seq_error = '1' or i2c_arb_lost = '1' ) then
                                i2c_status <= X"1"; state <= idle;
                              else state <= return_state;
                              end if;
                             end if;
```

4-142. ábra. *Szubrutinszerű állapotsorozat az I²C master IP-val való kommunikáció megvalósítására*

A regiszterek frissítését és kiolvasását végző vezérlési állapotokat rendre a 4-143. és 4-144. ábrák részletezik.

Az I²C-s EEPROM, A/D-átalakító és hőmérséklet-érzékelő IFA áramkörök HDL-modelljeinek vizsgálata után az RTCC IFA szolgáltatásvezérlő állapotgépe már kevés izgalmat tartogat, egy dologra azonban korábban nem láthattunk példát. Figyeljük meg, hogy az RTCC chip I²C slave interfész-áramkörében a címzőregiszter automatikus inkrementálásának képessége miként teszi lehetővé a hét regiszter írásának és olvasásának ciklusba szervezését. Íráskor az I²C-szabvány szerinti címzőkeret és az eszköz saját – később automatikusan inkrementálódó – címregiszterének kezdeti értékét meghatározó adatkeret elküldése és nyugtázásának fogadása után (4-143. ábra: *write_1…6* állapotok) a szolgáltatásvezérlő állapotgép a *transmit_data_loop_1…3* állapotok által képviselt ciklusba lép, ahol a ciklusváltozó szerepét a *cnt* regiszter tölti be. E regiszter aktuális értéke alapján dől el, hogy az adott iterációban melyik vektor értékét továbbítjuk az *i2c_master* IP-magnak (4-143. ábra: 48–55.sor). A kiválasztott bájt elküldése és a küldés nyugtázásának

VHDL

```
1  when write_1 => -- send start-condition
2    i2c_cmd <= "00";
3    return_state <= write_2;
4    state <= i2c_master_request_1;
5
6  when write_2 => -- send I2C addr. & nW bit
7    i2c_cmd <= "01";
8    i2c_data_2_send <= B"1101111" & '0';
9    i2c_no_bits <= 8;
10   return_state <= write_3;
11   state <= i2c_master_request_1;
12
13 when write_3 => -- receive ACK
14   i2c_cmd <= "10";
15   i2c_no_bits <= 1;
16   return_state <= write_4;
17   state <= i2c_master_request_1;
18
19 when write_4 =>
20   if ( i2c_data_received(0) = '0' ) then
21     -- ACK received
22     i2c_cmd <= "01"; -- send reg. address
23     i2c_data_2_send <= X"00";
24     i2c_no_bits <= 8;
25     return_state <= write_5;
26     state <= i2c_master_request_1;
27   else -- no ACK received
28     error_code <= X"2";
29     state <= idle;
30   end if;
31
32 when write_5 => -- receive ACK
33   i2c_cmd <= "10";
34   i2c_no_bits <= 1;
35   return_state <= write_6;
36   state <= i2c_master_request_1;
37
38 when write_6 =>
39   if ( i2c_data_received(0) = '0' ) then
40     -- ACK received
41     state <= transmit_data_loop_1;
42   else -- no ACK received
43     error_code <= X"2";
44     state <= idle;
45   end if;
46
```

VHDL (folyt.)

```
47 when transmit_data_loop_1 =>
48   case cnt is
49     when 0 => i2c_data_2_send <= sec_2_device;
50     when 1 => i2c_data_2_send <= min_2_device;
51     when 2 => i2c_data_2_send <= hour_2_device;
52     when 3 => i2c_data_2_send <= wkday_2_device;
53     when 4 => i2c_data_2_send <= date_2_device;
54     when 5 => i2c_data_2_send <= mth_2_device;
55     when others => i2c_data_2_send <= year_2_device;
56   end case;
57
58   i2c_cmd <= "01";
59   i2c_no_bits <= 8;
60   return_state <= transmit_data_loop_2;
61   state <= i2c_master_request_1;
62
63 when transmit_data_loop_2 => -- receive ACK
64   i2c_cmd <= "10";
65   i2c_no_bits <= 1;
66   return_state <= transmit_data_loop_3;
67   state <= i2c_master_request_1;
68
69 when transmit_data_loop_3 =>
70   if ( i2c_data_received(0) = '0' ) then
71     -- ACK received
72     if ( cnt < 6 ) then
73       cnt <= cnt + 1;
74       state <= transmit_data_loop_1;
75     else
76       cnt <= 0;
77       state <= write_7;
78     end if;
79   else -- no ACK received
80     error_code <= X"2";
81     cnt <= 0;
82     state <= idle;
83   end if;
84
85 when write_7 => -- send stop-condition
86   i2c_cmd <= "11";
87   return_state <= idle;
88   state <= i2c_master_request_1;
```

4-143. ábra. *Az MCP79410 RTCC áramkör adatregisztereit frissítő vezérlési állapotok*

megérkezése után a *cnt* regiszter értékét megnöveljük (4-143. ábra: 73. sor) és a *transmit_data_loop_1* állapotba visszalépve folytatjuk a műveletvégzést a következő bájt kiküldésével, miközben az RTCC áramkör a saját, belső címregiszterét automatikusan inkrementálja, vagy – ha megtörtént a hetedik regiszter frissítése is, vagyis a *cnt* értéke 6 (4-143. ábra: 73. sor) – nullázzuk a *cnt* regisztert és folytatjuk az I²C-tranzakciót a stopfeltétel generálásával (4-143. ábra: *write_7* állapot).

Olvasás esetén a helyzet – az I²C-interfészre általában is jellemző módon – bonyolultabb, mivel a címregiszter kezdőértékének beállítása után (4-144. ábra: *read_1...5* állapotok) meg kell fordítani az adatáramlás irányát, amit csak egy megismételt startfeltétel generálásával tehetünk meg (4-144. ábra: *read_6* állapot). Az újabb startfeltétel után az I²C-szabvány újabb címzőkeret kiküldését és nyugtázását írja elő (4-144. ábra: *read_7...9* állapotok), csak ezek végrehajtása után kerülhet sor a hét adatregiszter ciklusba szervezett kiolvasására (4-144. ábra: *receive_data_loop_1...2* állapotok). Szépérzékünket némileg bánthatja, hogy a hét iteráció – az írásnál látottakkal ellentétben – nem tökéletesen egyforma. A *receive_data_loop_2* állapotban azt látjuk, hogy míg a 0...5

VHDL

```
1  when read_1 => -- send start-condition
2    i2c_cmd <= "00";
3    return_state <= read_2;
4    state <= i2c_master_request_1;
5
6  when read_2 => -- send I2C addr. & nW bit
7    i2c_cmd <= "01";
8    i2c_data_2_send <= B"1101111" & '0';
9    i2c_no_bits <= 8;
10   return_state <= read_3;
11   state <= i2c_master_request_1;
12
13 when read_3 => -- receive ACK
14   i2c_cmd <= "10";
15   i2c_no_bits <= 1;
16   return_state <= read_4;
17   state <= i2c_master_request_1;
18
19 when read_4 =>
20   if ( i2c_data_received(0) = '0' ) then
21     -- ACK received
22     i2c_cmd <= "01"; -- send reg. address
23     i2c_data_2_send <= X"00";
24     i2c_no_bits <= 8;
25     return_state <= read_5;
26     state <= i2c_master_request_1;
27   else
28     -- no ACK received
29     error_code <= X"2";
30     state <= idle;
31   end if;
32
33 when read_5 => -- receive ACK
34   i2c_cmd <= "10";
35   i2c_no_bits <= 1;
36   return_state <= read_6;
37   state <= i2c_master_request_1;
38
39 when read_6 =>
40   if ( i2c_data_received(0) = '0' ) then
41     -- ACK received
42     i2c_cmd <= "00"; -- send (repeated)
43                      -- start-condition
44     return_state <= read_7;
45     state <= i2c_master_request_1;
46   else -- no ACK received
47     error_code <= X"2";
48     state <= idle;
49   end if;
50
51 when read_7 => -- send I2C addr. & R bit
52   i2c_cmd <= "01";
53   i2c_data_2_send <= B"1101111" & '1';
54   i2c_no_bits <= 8;
55   return_state <= read_8;
56   state <= i2c_master_request_1;
57
```

VHDL (folyt.)

```
58 when read_8 => -- receive ACK
59   i2c_cmd <= "10";
60   i2c_no_bits <= 1;
61   return_state <= read_9;
62   state <= i2c_master_request_1;
63
64 when read_9 =>
65   if ( i2c_data_received(0) = '0' ) then
66     -- ACK received
67     state <= receive_data_loop_1;
68   else -- no ACK received
69     error_code <= X"2";
70     state <= idle;
71   end if;
72
73 when receive_data_loop_1 =>
74   i2c_cmd <= "10"; -- receive data
75   i2c_no_bits <= 8;
76   return_state <= receive_data_loop_2;
77   state <= i2c_master_request_1;
78
79 when receive_data_loop_2 =>
80   if ( cnt < 6 ) then
81     case cnt is
82       when 0 => sec_from_device <= i2c_data_received;
83       when 1 => min_from_device <= i2c_data_received;
84       when 2 => hour_from_device <= i2c_data_received;
85       when 3 => wkday_from_device <= i2c_data_received;
86       when 4 => date_from_device <= i2c_data_received;
87       when 5 => mth_from_device <= i2c_data_received;
88       when others => null;
89     end case;
90
91     i2c_cmd <= "01"; -- send ACK
92     i2c_data_2_send <= X"00";
93     i2c_no_bits <= 1;
94     cnt <= cnt + 1;
95     return_state <= receive_data_loop_1;
96     state <= i2c_master_request_1;
97   else
98     year_from_device <= i2c_data_received;
99
100    i2c_cmd <= "01"; -- send NACK
101    i2c_data_2_send <= X"01";
102    i2c_no_bits <= 1;
103    cnt <= 0;
104    return_state <= read_10;
105    state <= i2c_master_request_1;
106  end if;
107
108 when read_10 => -- send stop-condition
109   i2c_cmd <= "11";
110   return_state <= idle;
111   state <= i2c_master_request_1;
```

4-144. ábra. *Az MCP79410 RTCC áramkör adatregisztereit kiolvasó vezérlési állapotok*

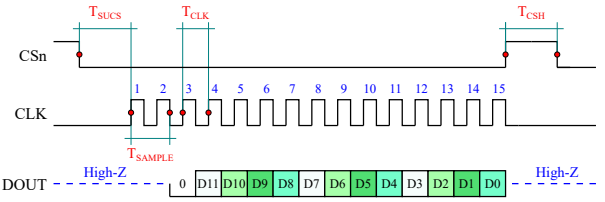
indexű regiszterek olvasását az IFA egy-egy ACK (alacsony szintű *SDA*) szimbólummal nyugtázza (4-144. ábra: 91–92. sor), addig az utolsó, 6-os indexű regiszter olvasását NACK (magas szintű *SDA*) szimbólummal kell lezárni, jelezve az RTCC áramkörnek, hogy a kiolvasást nem szeretnénk folytatni, befejezheti a belső címregiszterének automatikus inkrementálását. Ez az apró különbség teszi szükségessé a *receive_data_loop_2* állapot legkülső *if-else* szerkezetének beépítését. A hét regiszter blokkos kiolvasásának folyamatát a szokásos módon egy stopfeltétel generálásával zárjuk le a *read_10* állapotban.

4.4.6. SPI A/D-átalakító

Áttanulmányozásra javasolt alfejezetek:

4.3.2. *A soros kommunikáció alapáramkörei – SPI master és slave*

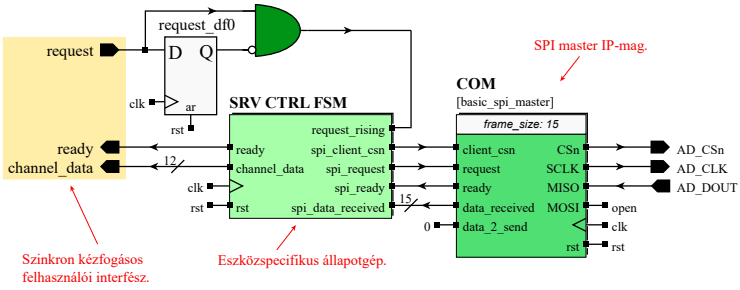
Az MCP3201 jelű katalógus-áramkör egy Microchip gyártmányú, egycsatornás, unipoláris, 12-bites, SPI-interfésszel ellátott A/D-átalakító. Az eszköz használata nagyon egyszerű, a korábban tárgyalt katalógus-áramkörökkel szemben konfigurációs lehetőségeket nem biztosít. Az A/D-konverzió kezdeményezése, egyúttal a mintavett analóg bemenő feszültség digitalizált értékének kiolvasása a 4-145. ábrán látható SPI-tranzakcióval lehetséges.



4-145. ábra. *Az A/D-konverzió kezdeményezése és az eredmény kiolvasása az MCP3201 chipből*

Az eszköz az analóg bemenetét a *CSn* jel lefutása utáni első órajel-felfutóélnél kezdi mintavételezni. A mintavétel az órajel második lefutóéléig tart, ezután a digitalizált érték egyes bitjei az órajel 4...15. felfutóéleinél olvashatók le a *DOUT* kimenetről. A helyes működés érdekében a felhasználónak gondoskodnia kell arról, hogy a *CSn* jel lefutása és az órajel első felfutása között legalább 100 ns (T_{SUCS}) idő teljen el, illetve, hogy a *CSn* jel minden felfutás után legalább 625 ns (T_{CSH}) ideig magas is maradjon, mielőtt újabb konverzió kezdeményezésére kerülne sor. Az áramkör órajelbemenete legfeljebb 1,6 MHz frekvenciájú négyszögjellel hajtható meg ($T_{CLK} \geq 0{,}625$ µs).

Az A/D-átalakítóhoz készített interfészadapter a megszokott módon épül fel; a kommunikációs modul szerepét az SPI master IP-mag *(basic_spi_master)* tölti be, a felhasználói interfészen érkező kérések SPI master modulnak szóló kérésekké transzformálását pedig a szolgáltatásvezérlő állapotgép *(SRV CTRL FSM)* végzi.



4-146. ábra. *Az MCP3201 A/D-átalakító interfészadapterének szerkezete*

Az IFA felhasználói interfésze természetesen ezúttal is szinkron kézfogásos jellegű, egyetlen adatbusza az A/D-átalakítóból kiolvasott, 12-bites adatot hordozza *(channel_data)*. A szolgáltatásvezérlő állapotgép VHDL-leírását a 4-147. ábra szemlélteti.

VHDL

```
1  L_SRV_CTRL_FSM: process ( clk, rst ) begin
2   if ( rst = '1' ) then
3     state <= idle; ready <= '1'; channel_data <= (others => '0');
4     spi_client_csn <= '1'; spi_request <= '0'; delay_cnt <= 0;
5   elsif ( rising_edge(clk) ) then
6   |
7   |  case state is
8   |  |
9   |  |  when idle => ready <= '1';
10  |  |               if ( request_rising = '1' ) then ready <= '0'; state <= get_data_1; end if;
11  |  |
12  |  |  when get_data_1 => if ( delay_cnt = spi_framing_delay_cnt_max ) then
13  |  |                       delay_cnt <= delay_cnt + 1; spi_client_csn <= '0';
14  |  |                     elsif ( delay_cnt = 2*spi_framing_delay_cnt_max ) then
15  |  |                       delay_cnt <= 0; spi_request <= '1'; state <= get_data_2;
16  |  |                     else delay_cnt <= delay_cnt + 1;
17  |  |                     end if;
18  |  |
19  |  |  when get_data_2 => spi_request <= '0'; state <= get_data_3;
20  |  |
21  |  |  when get_data_3 => if ( spi_ready = '1' ) then
22  |  |                       channel_data <= spi_data_received(11 downto 0); spi_client_csn <= '1';
23  |  |                       state <= idle;
24  |  |                     end if;
25  |  |
26  |  |  when others => null;
27  |  |
28  |  end case;
29  |
30  end if;
31 end process;
```

4-147. ábra. *Az MCP3201 IFA szolgáltatásvezérlő állapotgépének VHDL-modellje*

Az A/D-átalakító egyszerűségéből adódóan a szolgáltatásvezérlő állapotgép is csak néhány vezérlési állapotból áll. A felhasználói kérés beérkezése után (4-147. ábra: 10. sor) az eszköz T_{CSH} és T_{SUCS} időzítési előírásainak való megfelelést úgy biztosítjuk, hogy mielőtt az SPI master modul *client_csn* jelét – ami egyúttal az eszköz *CSn* bemenetét is közvetlenül meghajtja (lásd 450. ábra) – alacsony szintre állítanánk, az *spi_framing_delay_cnt_max* paraméternek megfelelő számú órajelciklus várakozást iktatunk be (4-147. ábra: 12–13. sor), majd a *CSn* jel lehúzása után még egyszer ennyi időt várunk, mielőtt az SPI-tranzakciót ténylegesen kezdeményeznénk (4-147. ábra: 14–15. sor). Az *spi_framing_delay_cnt_max* paramétert a példányosításkor az IP órajel-frekvenciájának ismeretében úgy kell beállítani, hogy az órajelciklusonként inkrementált *delay_cnt* számlálónak legalább T_{CSH} (625 ns) ideig tartson e paraméter értékét elérni. Az egyszerűség érdekében a T_{SUCS} időzítési feltétel teljesülését is ugyanezen paraméterhez kapcsoltuk, ami – tekintve, hogy ennek értéke mindössze 100 ns – némi túlbiztosítást jelent.[144]

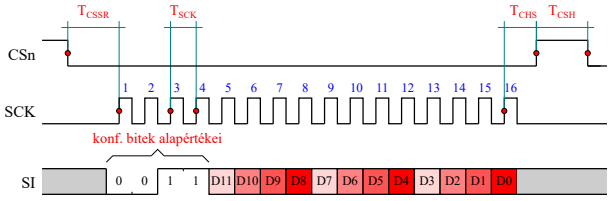
[144] Megjegyzendő, hogy ezzel az egyszerűsítéssel – még ha csak alig több, mint fél µs-mal is, de – megnöveltük azt a minimális időtartamot, ami alatt egy konverzió eredményét az A/D-átalakítóból kinyerhetjük. Időzítéskritikus alkalmazásokban ez nem feltétlenül megengedhető, ilyenkor ti. arra kell törekedni, hogy az eszköz képességeit a lehető legjobban kihasználjuk.

4.4.8. SPI D/A-átalakító

Áttanulmányozásra javasolt alfejezetek:

4.3.2. *A soros kommunikáció alapáramkörei – SPI master és slave*

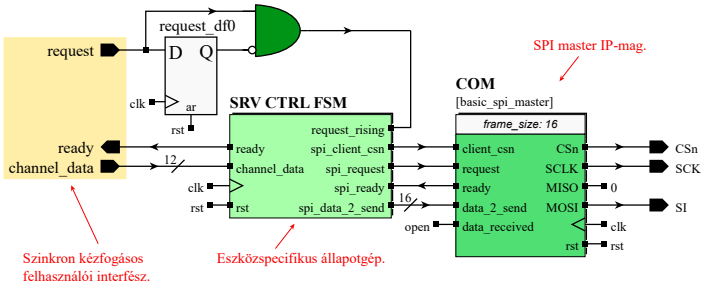
Az MCP4921 típusjelű katalógus-áramkör egy Microchip gyártmányú, egycsatornás, 12-bites, SPI-interfésszel szerelt D/A-átalakító. Az előző pontban bemutatott A/D-átalakítóhoz hasonlóan nagyon egyszerű eszköz; konfigurációs lehetőségei igen korlátozottak, mindössze négy biten kódolhatók. E konfigurációs lehetőségeket az itt bemutatott interfészadapter áramkör nem képezi le a felhasználói interfészre, az eszközt az alapértelmezett beállításaival használjuk. Ennek megfelelően a 4-148. ábrán látható SPI-tranzakció az egyetlen, amit az IFA megvalósít egy-egy felhasználói kérés beérkezésekor.



4-148. ábra. *Az MCP4921 D/A-átalakító adatregiszterét frissítő SPI-tranzakció*

A többi SPI-interfészű katalógus-áramkörhöz hasonlóan az MCP4921 D/A-átalakító is megfogalmaz néhány időzítési feltételt. Órajelének frekvenciája legfeljebb 20 MHz lehet ($T_{SCK} \geq 50$ ns), a *CSn* jel lefutása és az első órajel-felfutóél között legalább 40 ns-nak (T_{CSSR}), a 16. órajel-felfutóél és a *CSn* jel felfutása között pedig legalább 15 ns-nak kell eltelnie (T_{CHS}), illetve két SPI-tranzakció között a *CSn* jelet ugyancsak legalább 15 ns ideig magas szinten kell tartani (T_{CSH}).

A D/A-átalakító interfészadapterét az MCP3201 A/D-átalakító IFA „párjaként” készítjük el; a két IFA szerkezete megegyezik, és felhasználói interfészük is csaknem teljesen azonos, leszámítva persze az adatbusz irányának az alapvető funkcióra jellemző adatáramlási irányból adódó különbözőségét; egy A/D-átalakításkor a katalógus-áramkör küld adatot a felhasználónak, egy D/A-konverzió esetén pedig fordítva. A D/A-átalakító interfészadapterének felépítését a 4-149. ábra szemlélteti.



4-149. ábra. *Az MCP4921 D/A-átalakító interfészadapterének szerkezete*

Az IFA felhasználói interfésze szinkron kézfogásos jellegű, és az MCP3201 A/D-átalakítónál látottakhoz nagyon hasonló; egyetlen adatbusza a D/A-átalakítónak kiküldendő, 12-bites adatot hordozza *(channel_data)*. A szolgáltatásvezérlő állapotgép VHDL-leírását a 4-150. ábra szemlélteti.

VHDL

```
1  L_SRV_CTRL_FSM: process ( clk, rst ) begin
2   if ( rst = '1' ) then
3     state <= idle; ready <= '1'; spi_client_csn <= '1'; spi_request <= '0'; delay_cnt <= 0;
4   elsif ( rising_edge(clk) ) then
5   |
6   |  case state is
7   |  |
8   |  | when idle => ready <= '1';
9   |  |              if ( request_rising = '1' ) then ready <= '0'; state <= send_data_1; end if;
10  |  |
11  |  | when send_data_1 => spi_data_2_send <= "0011" & channel_data;
12  |  |
13  |  |                     if ( delay_cnt = spi_framing_delay_cnt_max ) then
14  |  |                       delay_cnt <= delay_cnt + 1; spi_client_csn <= '0';
15  |  |                     elsif ( delay_cnt = 2*spi_framing_delay_cnt_max ) then
16  |  |                       delay_cnt <= 0; spi_request <= '1'; state <= send_data_2;
17  |  |                     else delay_cnt <= delay_cnt + 1;
18  |  |                     end if;
19  |  |
20  |  | when send_data_2 => spi_request <= '0'; state <= send_data_3;
21  |  |
22  |  | when send_data_3 => if ( spi_ready = '1' ) then spi_client_csn <= '1'; state <= idle; end if;
23  |  |
24  |  | when others => null;
25  |  |
26  |  end case;
27  |
28  end if;
29 end process;
```

4-150. ábra. *Az MCP4921 IFA szolgáltatásvezérlő állapotgépének VHDL-modellje*

A szolgáltatásvezérlő állapotgép részletei is nagyon hasonlóak az SPI-os A/D-átalakító példájában láthatókhoz. Az eszköz által támasztott időzítési feltételeknek való megfelelést ez a megoldás pontosan ugyanúgy kezeli; az SPI vezetékeken végrehajtott állapotváltozásokat egy belső számláló *(delay_cnt)* ütemezi. A felhasználói kérés detektálása után (4-150. ábra: 9. sor) a szolgáltatásvezérlő az SPI master modul bemeneti adatbuszára helyezi a kiküldendő adatot (4-150. ábra: 11. sor), de az *spi_client_csn* jelnek – egyúttal az eszköz *CSn* bemenetének (lásd 4-50. ábra) – a lehúzására csak az *spi_framing_delay_cnt_max* paraméter által meghatározott számú órajelciklus után kerül sor (4-150. ábra: 13–14. sor). E paraméter megfelelő beállítása esetén biztosított a T_{CSH} időzítési feltétel teljesülése. Újabb, ugyanakkora mértékű várakozás után következik az SPI-tranzakció kezdeményezése (4-150. ábra: 15–16. sor), amivel a T_{CSSR} feltétel is teljesíthető. A T_{CHS} időzítési feltétel teljesülését részben maga az SPI master IP-mag biztosítja; az SPI-tranzakció végét jelző felfutóél az *spi_ready* jelen csak az órajel 16. lefutóéle után következik be (vö. 4-53. ábra: 27. sor). Az eszköz órajel-frekvenciájára vonatkozó $T_{SCK} \geq 50$ ns feltételt betartva tehát biztosak lehetünk benne, hogy a 16. órajel-felfutóél és az *spi_ready* jelzés felfutása között legalább 25 ns (+ az IFA saját órajelének egy periódusa) telik el, ami már önmagában nagyobb, mint az eszköz által előírt minimális 15 ns, ezért a szolgáltatásvezérlő állapotgépbe ezen a ponton újabb várakozás beiktatására nincs szükség.[145]

[145] Vö. Az SPI EEPROM T_{CSH} időzítési feltétele ezt az egyszerűsítést nem teszi lehetővé. Az EEPROM *SCK* órajelének minimális periódusideje 100 ns, vagyis az utolsó *SCK* felfutóél és az SPI master modul *ready* jelének felfutása közötti minimális időtartam csupán 50 ns (+ az IFA saját órajelének periódusideje). Mivel az EEPROM esetén a T_{CSH} késleltetés minimuma 100 ns, ezért az SPI-tranzakció végén további késleltetés beiktatására van szükség a *CSn* jel felhúzása előtt (lásd 4-98. ábra: 11. sor).

4.4.9. Órajel-szinkronizálás

Áttanulmányozásra javasolt alfejezetek:
4.2. *On-chip adattárolás*

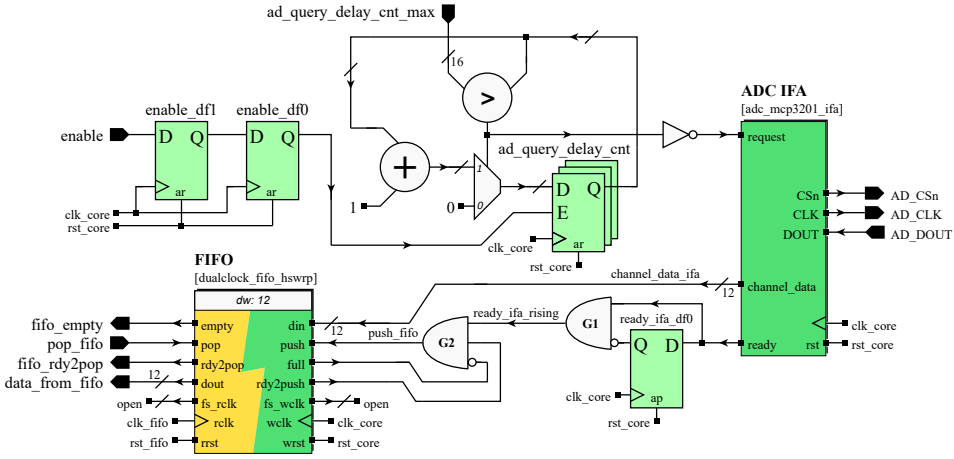
Ahogy minden IP-magra, úgy az interfészadapterekre is igaz, hogy lokális órajelük frekvenciáját célszerű feladatukhoz mérten a lehető legkisebbre állítani a fogyasztás és az erőforrás-felhasználás csökkentése érdekében. Az interfészadapterek alapvető feladata az adott katalógus-áramkör szolgáltatásainak leképezése egy kényelmesen használható párhuzamos felhasználói interfészre, eltakarva a konkrét eszköz működésének és saját, az esetek többségében soros interfészének sajátosságait. Ezért az ilyen IP-magok lokális órajelének optimális frekvenciáját jellemzően e soros interfész sebessége határozza meg. Az IFA belső részegységeit ugyan nem érdemes a kommunikációs interfész által meghatározottnál magasabb frekvenciájú órajelről járatni, ugyanakkor – mint minden újrafelhasználható IP-mag esetén – az IFA órajelét szeretnénk függetleníteni a beágyazó környezet órajelétől. E két érv mentén arra a következtetésre juthatunk, hogy az interfészadapterek belsejében a soros kommunikációs almodulokat azok órajel-szinkronizáló burkolói nélkül célszerű példányosítani, ugyanakkor magukhoz az interfészadapterekhez is érdemes saját órajel-szinkronizáló burkolókat készíteni.

Az ebben az alfejezetben bemutatott interfészadapterek felhasználói interfésze mindig egy, az adott katalógus-áramkör szolgáltatásainak leképezéséhez szükséges számú és méretű adatbuszt tartalmazó szinkron kézfogásos interfész, ami az órajel-szinkronizálás kérdését eléggé leegyszerűsíti; a 2.1. pontban tárgyalt általános vektorszinkronizáló fokozatok mindegyik esetben használhatók. Mivel egyik interfészadapter sem képes nulla késleltetésű szinkron kézfogásos tranzakció végrehajtására, ezért a két CDC-fokozat közül használhatjuk az egyszerűbbet (lásd 2-2. ábra).[146]

FIFO-alapú órajel-szinkronizáló burkolók ◆ Ahogy a soros-párhuzamos és párhuzamos-soros átalakítók esetén, úgy az interfészadaptereknél is elképzelhető olyan alkalmazás, amelyben az IFA egy adatfolyam továbbításában vesz részt. Ilyen esetekben hasznos lehet FIFO-alapú órajel-szinkronizáló fokozatok beépítése. Ez a felhasználás leginkább a digitális jelfeldolgozó rendszerek A/D- és D/A-átalakítóinak interfészadaptereire jellemző, ezért ezt a megoldást az MCP3201 és MCP4921 katalógus-áramkörökhöz készített interfészadaptereken szemléltetjük.

Az MCP3201 A/D-átalakítóhoz készített IFA a felhasználói interfészén érkező kérés hatására utasítja az A/D-átalakítót egy konverzió végrehajtására, majd annak végeztével az analóg jel egy mintájának digitalizált értékét adja át a felhasználónak. Adatfolyam jellegű alkalmazásokban kézenfekvő, hogy ez az „átadás” az adatnak egy FIFO-ba való beírását jelentse, amely FIFO a CDC-funkciót is megvalósító *dualclock_fifo_hswrp* IP-mag (lásd 4-27. ábra) egy példánya. Ugyancsak az adatfolyamszerű működés igényéből fakad, hogy az A/D-konverziók kezdeményezését „automatizálnunk” kell, olyan illesztőlogikát kell terveznünk az IFA *request* jelének meghajtására, amely

[146] Az interfészadapterek felhasználói interfészei kliens típusúak, vagyis az adott katalógus-áramkör szolgáltatásainak igénybevételét a felhasználó kezdeményezi. Kivételt képvisel a 24LC16B EEPROM IFA hibainterfésze, amely az I²C-buszon előforduló hibákról azok bekövetkezésének pillanatában azonnal tájékoztatja a felhasználót. Ez az interfész tehát kiszolgáló (kezdeményező) jellegű, de viselkedése megfelel a szinkron kézfogásos interfész-specifikációnak, tehát az általánosított CDC-logika itt is megállja a helyét (vö. 4.3.1.1. pont, 4-2. táblázat: UART-vevő órajel-szinkronizáló fokozata).



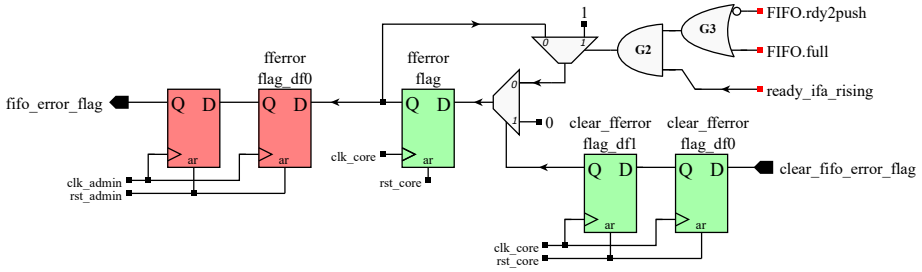
4-151. ábra. *FIFO-alapú órajel-szinkronizáló burkoló az MCP3201 IFA áramkörhöz (1)*

felhasználói beavatkozás nélkül is folyamatosan működteti az IFA-t, azon keresztül pedig magát az A/D-átalakítót. A 4-151. ábrán egy olyan példát láthatunk, amely ezt a folyamatos lekérdezést egy számlálóval valósítja meg. Ebben a megoldásban az órajel minden felfutóélénél inkrementálódó *ad_query_delay_cnt* számláló egy futásidejű paraméterként megadott maximális érték *(ad_query_delay_cnt_max)* elérésekor nullázódik, miközben egy pozitív impulzust állít elő az IFA *request* bemenetén. Az egyes A/D-konverziók végét az IFA *ready* jelének felfutóéle jelzi. Ha a konverzió végén a FIFO készen áll újabb adat beírására (*rdy2push* kimenete magas, *full* kimenete pedig alacsony szintű), akkor a *G2 AND* kapun keresztül a *ready* jelet figyelő felfutóél-detektor által generált pozitív impulzus a FIFO *push* bemenetére jut, eltárolva az IFA adatkimenetének értékét, vagyis a legutóbbi A/D-konverzió eredményét.[147]

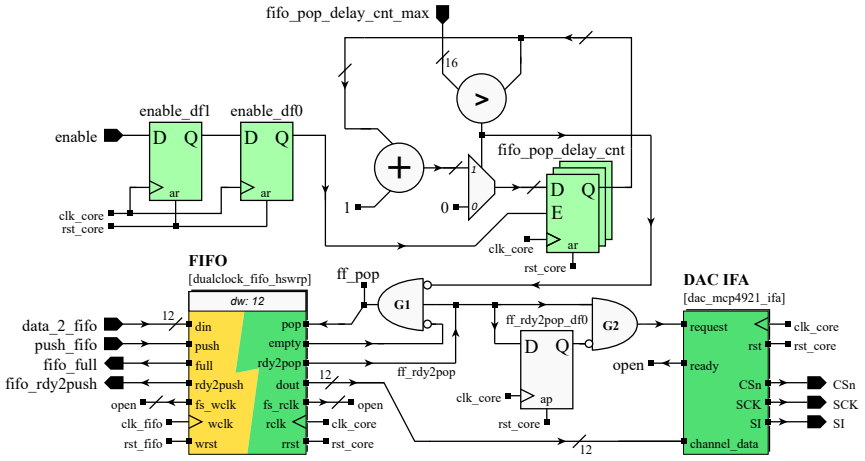
Ha az A/D-konverzió végén a FIFO még nem áll készen újabb adat beírására, akkor adatvesztés lép fel, amiről a felhasználót tájékoztatnunk kell. A burkolómodul 4-152. ábrán látható kiegészítésével könnyen létrehozhatunk egy erre alkalmas állapotjelzőt; ha a *G2* és *G3* kapuk képezte egyszerű dekóder hibát jelez, akkor az *fferror_flag* flip-flopot egybe állítjuk. Statikus állapotjelző lévén az *fferror_flag* értékét egy egyszerű double-flop szinkronizáló fokozaton keresztül adhatjuk át a felhasználónak, aki az ugyancsak double-flop szinkronizálón keresztül fogadott *clear_fifo_error_flag* bemenet magas szintre húzásával törölheti azt.

Az MCP4921 D/A-konverter interfészadapteréhez egy nagyon hasonló felépítésű és működési elvű CDC-burkolómodult készíthetünk a 4-153. ábrán látható módon.

[147] Az A/D-átalakító 4-151. ábrán látható órajel-szinkronizáló burkolója jól illeszkedik a jelfeldolgozó rendszerekben használatos architekturális sémába, ahol az egyes adatfeldolgozó fokozatok aszinkron órajeltartományokban működnek, mintafeldolgozási ütemük eltérő és egymásnak dual-clock FIFO-kon keresztül adják át az adatokat (lásd pl. 4-249. ábra: FIR-szűrő). Az ilyen felépítésű rendszerekben a 4-151. ábrán látható áramkörnek kiemelt szerep jut; az *ad_query_delay_cnt_max* paraméter értéke és az IFA *clk_core* órajelének frekvenciája együttesen meghatározzák azt, hogy milyen gyakran utasítjuk az A/D-átalakítót új konverzió elvégzésére, vagyis hogy mekkora a teljes jelfeldolgozó rendszer szempontjából kulcsfontosságú mintavételi frekvencia.



4-152. ábra. *FIFO-alapú órajel-szinkronizáló burkoló az MCP3201 IFA áramkörhöz (2)*

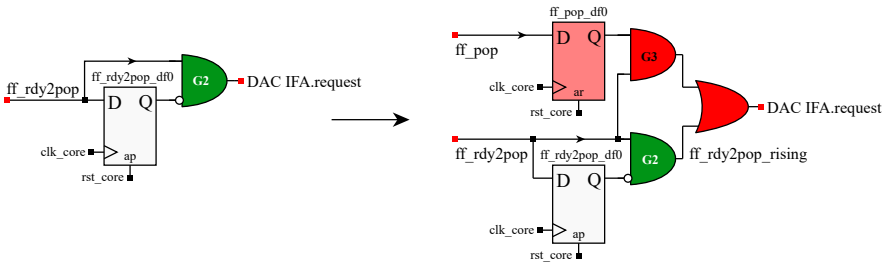


4-153. ábra. *FIFO-alapú órajel-szinkronizáló burkoló az MCP4921 IFA áramkörhöz (1)*

A D/A-átalakítónak küldendő bináris vektorokat a felhasználó a CDC-funkciót ellátó *dualclock_fifo_hswrp* IP-mag egy példányába írja be. A FIFO kiolvasásának és az interfészadapter aktiválásának ütemét egy, az A/D IFA burkolónál láthatóval megegyező működésű számlálóstruktúra határozza meg. A *fifo_pop_delay_cnt_max* futásidejű paraméterben rögzített számú órajelciklus után a *fifo_pop_delay_cnt* számláló nullázódik. A maximális értéket dekódoló komparátor kimenetén megjelenő negatív impulzus a *G1 AND* kapun keresztül a FIFO *pop* bementére jut, feltéve, hogy a FIFO nem üres és készen áll a kiolvasás műveletre (*empty* kimenete alacsony, *rdy2pop* kimenete pedig magas szintű).[148] A FIFO kiolvasásának végét és a FIFO adatkimenetének érvényességét a *rdy2pop* kimeneten megjelenő felfutóél jelzi, az erre a kimenetre kapcsolt felfutóél-detektor pedig ennek észlelésekor a kimenetén megjelenő pozitív impulzussal utasítja az interfészadaptert a FIFO-ból kiolvasott adat elküldésére.

[148] Ezúttal nem tekintjük hibának, ha a kiolvasás pillanatában a FIFO még nem áll készen, ennek következménye csupán az, hogy a D/A-átalakító analóg kimenetén mindaddig az előző értéket látjuk, amíg új adat nem érkezik a FIFO-ba.

A 4-153. ábrán látható kapcsolás némi kiegészítésre szorul, ha a *dualclock_fifo_hswrp* IP-mag nulla késleltetésű változatát példányosítjuk, ekkor ti. a *rdy2pop* jelzés folyamatosan magas szintű, azon felfutóél sosem keletkezik (lásd 15. ábra). Ugyanezzel a problémával találkoztunk az UART-adó FIFO-alapú CDC-burkolómoduljánál is, az ott bemutatott megoldás ezúttal is használható. Az IFA *request* jelét meghajtó illesztőlogika a FIFO kiolvasó interfészét megfigyelve megállapíthatja, hogy nulla késleltetésű FIFO-val van dolga; ha az egyetlen órajelciklussal megkésleltetett *pop* impulzus magas szintje mellett a FIFO *rdy2pop* kimenete magas szintű, az egyértelműen azt jelzi, hogy a FIFO nulla késleltetésű, adatkimenetén az új érvényes adat már a *pop* impulzus lefutóélével egyidőben megjelenik, az IFAtranzakció tehát kezdeményezhető. A CDC-burkolómodul a 4-154. ábrán látható kiegészítéssel tehető kompatibilissé a FIFO nulla késleltetésű variánsával.



4-154. ábra. *FIFO-alapú órajel-szinkronizáló burkoló az MCP4921 IFA áramkörhöz (2)*

4.5. Aritmetika és jelfeldolgozás

A digitális áramkörök jelentős részét összetett számítási feladatok elvégzésére tervezzük. Ezek a számítási feladatok matematikai műveletek egymás után és/vagy egymással párhuzamosan végrehajtott kombinációi, amelyek önmaguk is egyszerűbb matematikai műveletekből állnak. Gondolhatunk például a bonyolult titkosítási eljárásokra, amelyekben fontos szerepe van a prímtényezőkre bontás műveletének. A prímtényezős felbontás legegyszerűbb algoritmusa prímszámokkal végzett maradékos osztások sora, a maradékos osztás különféle algoritmusai pedig a legelemibb kivonás és összehasonlítás műveletekre épülnek. Ez a hierarchikus egymásba ágyazottság ugyancsak alátámasztja az újrafelhasználhatóságra vonatkozó tervezési elvet; parametrizált formában leírt, többféle alkalmazási környezet igényei szerint testreszabható RTL-modellekből a bonyolult, sok műveletet igénylő adatfeldolgozási feladatok végrehajtására képes áramköröket gyorsan felépíthetjük. Ebben az alfejezetben a legalapvetőbb aritmetikai funkciókat megvalósító általános célú modulokon felül néhány összetettebb, specifikusabb feladatot ellátó adatfeldolgozó IP-magot is részletesen bemutatunk. Ezek működésének könnyebb megértése érdekében érdemes átolvasni a bináris fix- és lebegőpontos számábrázolás alapelveit összefoglaló 5.3. Függeléket.

4.5.1. Fixpontos aritmetikai egység

Áttanulmányozásra javasolt alfejezetek:

5.3.1. *Bináris számábrázolás – Fixpontos számok*

Ebben a pontban egy kompakt, általános célú, fixpontos aritmetikai modult mutatunk be. Kompaktsága abban rejlik, hogy egyazon tervezési egység (egy VHDL csomagban definiált eljárás) három fixpontos alapműveletet – az összeadást, a kivonást és a szorzást – valamint három, az eredményvektorok méretezésével kapcsolatos segédműveletet ír le. A modul sajátossága, hogy a túlcsordulást és a szaturációt is kezeli, így beépíthető részegységként használva lehetővé teszi, hogy a konkrét számítási feladatra koncentráljunk a számábrázolás ügyes-bajos dolgai helyett. A VHDL eljárás *(procedure)* deklarációját a 4-155. ábra mutatja be.

VHDL

```
procedure fxp_arith_procedure (
  operation:          in  std_logic_vector (2 downto 0); -- Az elvégzendő művelet kódja.
  twos_complement_on: in  boolean;                       -- Kettes-komplemens kódolás bekapcsolása.
  saturation_on:      in  boolean;                       -- Szaturáció bekapcsolása.
  resize_on:          in  boolean;                       -- Eredmény átméretezése az operandusok méretére.
  iipw:               in  integer range 1 to 64; -- "input integer part width", az egészrész-bitek száma.
  ifpw:               in  integer range 0 to 63; -- "input fractional part width", a törtrész-bitek száma.
  opa:                in  std_logic_vector;              -- A operandus.
  opb:                in  std_logic_vector;              -- B operandus.
  result:             out std_logic_vector;              -- Eredmény.
  overflow:           out std_logic                      -- A túlcsordulást jelző kimenet.
);
```

4-155. ábra. *A fixpontos aritmetikai eljárás deklarációja*

Az eljárás öt paramétert bocsát a felhasználó rendelkezésére. A *twos_complement_on* paraméterrel fejezhetjük ki, hogy az operandusokat nemnegatív vagy kettes-komplemens kódolású bináris számként kell-e értelmezni, a *saturation_on* paramétert pedig a telítődéses működés be- vagy kikapcsolására használhatjuk. N-bites operandusok esetén az összeadás és kivonás műveletek (N+1), a szorzás pedig (2×N)-bites eredményre vezet. A *resize_on* paraméter segítségével meghatározhatjuk, hogy a modul az aritmetikailag garantáltan helyes, megnövekedett méretű eredményt állítsa elő, vagy a kimenet méretét az operandusok méretére redukálja. Az *iipw (input integer part width)* és *ifpw (input fractional part width)* paraméterekkel állítható be az operandusok formátuma az egészrészt és a törtrészt ábrázoló bitek számának egzakt meghatározásával. A fixpontos aritmetikai eljárás használatakor az *operation* bemenettel jelölhetjük ki az elvégzendő műveletet az alábbiak szerint:

- 000_2: Az eredmény *(result)* az A operandus *(opa)* és a B operandus *(opb)* összege.
- 001_2: Az eredmény A és B különbsége.
- 010_2: Az eredmény A és B szorzata.
- 011_2: Az összegként értelmezett A operandus méretének redukálása.
- 100_2: A különbségként értelmezett A operandus méretének redukálása.
- 101_2: A szorzatként értelmezett A operandus méretének redukálása.

A három méretcsökkentő üzemmódra azért van szükség, mert az aritmetikai alapműveletek csak az operandusok méretével megegyező *(resize_on = true)* vagy az ahhoz képest megnövelt méretű *(resize_on = false)* kimenetet tudják előállítani. Ha egy adatfeldolgozó rendszerben használjuk majd az aritmetikai modult, és a teljes számítási lánc végén az eredeti operandusok számábrázolási formátumára szeretnénk visszatérni, akkor a belső változóinkat vissza kell méreteznünk a kimenetek kívánt mérete szerint. A potenciálisan túlcsordulást okozó méretcsökkentésnél fontos az, hogy a legutolsó méretnövekedést milyen művelet okozta, hiszen ez határozza meg, hogy a méretcsökkentés során melyik biteket kell levágnunk (összeg vagy különbség esetén csak a legfelső bitet, szorzat esetén a legfelső és a legalsó bitek egy részét). Továbbá a szaturációs értékek meghatározásánál az összeg és különbség típusú bemenet közötti különbségtétel is számít. Ezeknek a műveleteknek a részleteit később kifejtjük.

A fixpontos aritmetikai modul tehát egy VHDL eljárás, amelynek vázlatos szerkezetét a 4-156. ábra szemlélteti. A 3–7. sorokban segédkonstansokat hoztunk létre. Az *iw* az operandusok méretét leíró egész szám, amit a felhasználó által megadott, az egészrész és a törtrész ábrázolására használt bitek számának összegeként képezünk. A *zeros_a/b* és a *ones_a/b* vektorok megfelelően méretezett, csak nullákat/egyeseket tartalmazó logikai vektorok. A 9–11. sorokban ideiglenes változókat deklaráltunk, amelyek mindig az operandusok aritmetikailag garantáltan helyes (nem fordulhat elő túlcsordulás), teljes méretű összegét *(tmp_sum)*, különbségét *(tmp_dif)* és szorzatát *(tmp_product)* fogják tárolni.[149]

VHDL

```
1  procedure fxp_arith_procedure ([...]) is
2
3  | constant iw:          integer                                  := iipw + ifpw;
4  | constant zeros_a:     std_logic_vector (iipw downto 0)         := (others => '0');
5  | constant ones_a:      std_logic_vector (iipw downto 0)         := (others => '1');
6  | constant zeros_b:     std_logic_vector (iipw/2 downto 0)       := (others => '0');
7  | constant ones_b:      std_logic_vector (iipw/2 downto 0)       := (others => '1');
8  |
9  | variable tmp_product: std_logic_vector (2*(iipw+ifpw)-1 downto 0);
10 | variable tmp_sum:     std_logic_vector (iipw+ifpw downto 0);
11 | variable tmp_dif:     std_logic_vector (iipw+ifpw downto 0);
12 |
13 begin
14 |
15 | -- default outputs... ---------------------------
16 | result := (result'range => '0'); overflow := '0';
17 |
18 | case operation is
19 | |
20 | | when "000" => [...] -- ADDITION                  result = opa + opb
21 | | when "001" => [...] -- SUBTRACTION               result = opa - opb
22 | | when "010" => [...] -- MULTIPLICATION            result = opa * opb
23 | | when "011" => [...] -- DOWNSIZE OPA AS SUM       result = downsize(opa)
24 | | when "100" => [...] -- DOWNSIZE OPA AS DIFFERENCE result = downsize(opa)
25 | | when "101" => [...] -- DOWNSIZE OPA AS PRODUCT   result = downsize(opa)
26 | |
27 | | when others => assert false severity failure; report "Invalid operation code!";
28 | |
29 | end case;
30 |
31 end procedure fxp_arith_procedure;
```

4-156. ábra. *A fixpontos aritmetikai modult leíró VHDL eljárás vázlata*

[149] Itt a „tárolás” kifejezés csak azt jelenti, hogy a változókat ezeknek az értékeknek a reprezentálására használjuk. Mivel ez egy VHDL eljárás, amelyet fizikailag egy kombinációs hálózat valósít majd meg, ezért a leírásban lévő változók áramköri értelemben nem tárolók, csupán vezetékek. Arról, hogy szükség esetén ezeknek a vezetékeknek az értéke D flip-flopokban vagy RAM-cellákban legyen eltárolva, a modul felhasználójának kell gondoskodnia (pl. úgy, hogy az eljárást egy szinkron folyamatban használja, lásd később).

A VHDL eljárás egy összetett kombinációs hálózat viselkedését igyekszik kifejezni. Ahhoz, hogy a leírásból az áramkörszintézis során valóban egy – kimeneti latch-et nem tartalmazó – tisztán kombinációs hálózat keletkezzen, gondoskodnunk kell róla, hogy a modell a bemenetek bármilyen kombinációjához a kimeneteknek valamilyen konkrét értékét rendelje (ahogy azt egy kombinációs hálózattól is elvárjuk).[150] A 16. sorban lévő alapértelmezett értékadások ennek garantálását szolgálják. A 18-29. sorok között definiált *case* szerkezet törzsében lévő utasítások az alapértelmezett kiértékelések eredményét felülírják az eljárás törzsének, mint VHDL utasításblokknak a sorrendi jellege miatt.[151] A továbbiakban az egyes műveletekhez tartozó utasításokat – vagyis a *case* szerkezet ágainak belső működését – egy-egy külön ábrán mutatjuk be. A 4-157. ábra az összeadás művelethez tartozó kódrészletet szemlélteti.

VHDL

```
1  -- ADDITION --
2  when "000" => if ( twos_complement_on = true ) then
3                  tmp_sum := std_logic_vector(signed(opa(opa'left) & opa) + signed(opb(opb'left) & opb));
4                else tmp_sum := std_logic_vector(unsigned('0' & opa) + unsigned('0' & opb));
5                end if;
6
7            -- túlcsordulás detektálása és szaturáció...
8            if ( resize_on = true ) then result := tmp_sum(tmp_sum'left-1 downto 0);
9
10           | if ( twos_complement_on = true ) then
11           | |
12           | | if ( tmp_sum(tmp_sum'left) /= tmp_sum(tmp_sum'left-1) ) then overflow := '1';
13           | | |
14           | | | if ( saturation_on = true ) then
15           | | | |
16           | | | | if ( tmp_sum(tmp_sum'left) = '1' ) then
17           | | | |   result(tmp_sum'left-1) := '1'; result(tmp_sum'left-2 downto 0) := (others => '0');
18           | | | | else
19           | | | |   result(tmp_sum'left-1) := '0'; result(tmp_sum'left-2 downto 0) := (others => '1');
20           | | | | end if;
21           | | | |
22           | | | end if;
23           | | |
24           | | end if;
25           | |
26           | else -- nemnegatív eset
27           | |
28           | | if ( tmp_sum(tmp_sum'left) = '1' ) then overflow := '1';
29           | |   if ( saturation_on = true ) then result := (result'range => '1'); end if;
30           | | end if;
31           | |
32           | end if;
33           |
34           else result := tmp_sum; -- az eredményt nem méretezzük át
35           end if;
```

4-157. ábra. *Az összeadás művelet megvalósítása a fixpontos aritmetikai modulban*

Az összeadás művelet – a kivonáshoz és a szorzáshoz hasonlóan – az aritmetikailag garantáltan helyes, megnövelt méretű eredményvektor előállításával kezdődik (4-157. ábra: 2–5. sor), amelyet a túlcsordulás és a szaturáció kezelésére vonatkozó logika leírása követ (4-157. ábra: 8–35. sor). A megnövelt méretű összeg *(tmp_sum)* előállításához az *ieee* könyvtárbeli *numeric_std* csomag *signed* és *unsigned* adattípusaira definiált összeadás operátort használjuk. Fontos részlet, hogy az

[150] Lásd még DRMSz: 4-43. ábra.
[151] Vö. DRMSz: 4-45. ábra: A folyamatok sorrendi jellegének kihasználása prioritás megfogalmazásához.

összeadást az előjelkiterjesztett[152] operandusokon hajtjuk végre, mert a *numeric_std* csomag operátorai N-bites operandusok esetén ugyancsak N-bites eredménnyel térnek vissza, és semmilyen módon nem adják tudtunkra, hogy a művelet során keletkezett-e átvitel, az összeg legfelső bitjét egyszerűen levágják.

Ha a híváskor a felhasználó nem kérte az eredmény visszaméretezését az operandusok méretére *(resize_on = false)*, vagyis közvetlenül a megnövelt méretű eredményvektorra kíváncsi, akkor a helyzet egyszerű, a kimenetre a *tmp_sum* belső változó értékét kell írni (4-157. ábra: 34. sor). Ellenkező esetben az eredményről le kell választani az összeadáskor keletkezett átvitelbitet, vagyis a *tmp_sum* vektor legfelső bitjét (4-157. ábra: 8. sor). A gond az, hogy a legfelső helyiértéken lévő bit aritmetikai szempontból igencsak értékes, vagyis levágásával információt veszítünk, tehát túlcsordulás fordulhat elő. Fixpontos aritmetikai modulunkat pontosan azért hozzuk létre, hogy ezt az információvesztést „ne hagyja annyiban"; még ha a kimenet kívánt mérete nem is teszi lehetővé a pontos eredmény előállítását, a túlcsordulás tényét jelezzük, továbbá túlcsorduláskor – igény esetén – magát az eredményt is másképpen, szaturációs viselkedést megvalósítva állítjuk elő. A túlcsordulás detektálásának és a szaturáció megvalósításának részletei nemnegatív és kettes-komplemens kódolás esetén eltérnek, ezért logikánk a *twos_complement_on* paraméter értéke szerint kettéágazik. A nemnegatív eset egyszerűbb (4-157. ábra: 28–30. sor). Ilyenkor, ha a *tmp_sum* változó, tehát a megnövelt méretű eredményvektor felső bitje 1, az túlcsordulást jelent (4-157. ábra: 28. sor). Ha a nemnegatív operandusok összeadása során túlcsordulás keletkezik és a felhasználó szaturációs viselkedést határozott meg *(saturation_on = true)*, akkor a kimenetre a rendelkezésre álló eredménybitek számának megfelelő legnagyobb értéket kell juttatni, vagyis a kimenet minden bitjét 1-re kell állítani (4-157. ábra: 29. sor). Kettes-komplemens kódolás esetén (tehát ha az operandusvektorok nemnegatív és negatív számokat is képviselhetnek), akkor a megnövelt méretű eredményvektor legmagasabb és az annál eggyel alacsonyabb helyiértékű bitjeinek különbözősége jelzi a túlcsordulást (4-157. ábra: 12. sor). A túlcsordulás negatív és pozitív irányú is lehet; ha a megnövelt méretű eredményvektor legfelső helyiértékű bitje 1, akkor negatív, ellenkező esetben pozitív túlcsordulásról van szó. A kimenet bitszámának megfelelő, lehető legnagyobb abszolútértékű számot negatív esetben az $1000..._2$ (4-157. ábra: 17. sor), pozitív esetben pedig a $0111..._2$ bináris kombináció fejezi ki (4-157. ábra: 19. sor).

A kivonás művelet megvalósítása (lásd 4-158. ábra) koncepcionálisan megegyezik az összeadáséval, ezért ennek sajátosságait az összeadással összevetésben tárgyaljuk. A megnövelt méretű eredményvektor *(tmp_dif)* előállítására az összeadásnál látottakhoz hasonlóan a *numeric_std* csomag megfelelő operátorát használjuk. A *tmp_dif* változó beállítása után a túlcsordulás és a szaturáció kezelése csaknem teljesen megegyezik az összeadásnál látottakkal (4-158. ábra: 10–24. sor). Különbség a két művelet között két helyen jelentkezik. Ha a felhasználó nem kívánja redukálni az eredményvektor méretét, akkor összeadásnál a helyzet egyszerű, mert az összeg nem eshet kívül a megnövelt számábrázolási tartományon. Nemnegatív számok kivonásánál azonban előfordulhat, hogy az eredmény negatív, ami túlcsordulást jelent még megnövelt méretű kimeneti vektor

[152] Az előjelkiterjesztés nemnegatív esetben csak annyit jelent, hogy az operandushoz „balról", tehát a legmagasabbnál eggyel magasabb helyiértékre egy nullát illesztünk (ez annak felel meg, mintha a tízes számrendszerbeli 13,28 helyett azt írnánk, hogy 013,28), kettes-komplemens kódolás esetén pedig az operandus legmagasabb helyiértékű bitjét másoljuk a kiterjesztett vektor legmagasabb helyiértékére, így érve el, hogy a bitek száma eggyel nagyobb legyen, az ábrázolt érték viszont ne változzon.

VHDL

```
1  -- SUBTRACTION --
2  when "001" => if ( twos_complement_on = true ) then
3                  tmp_dif := std_logic_vector(signed(opa(opa'left) & opa) - signed(opb(opb'left) & opb));
4                else tmp_dif := std_logic_vector(unsigned('0' & opa) - unsigned('0' & opb));
5                end if;
6
7            -- túlcsordulás detektálása és szaturáció...
8            if ( resize_on = true ) then result := tmp_dif(tmp_dif'left-1 downto 0);
9
10           if ( twos_complement_on = true ) then
11
12             if ( tmp_dif(tmp_dif'left) /= tmp_dif(tmp_dif'left-1) ) then overflow := '1';
13
14               if ( saturation_on = true ) then
15
16                 if ( tmp_dif(tmp_dif'left) = '1' ) then
17                   result(tmp_dif'left-1) := '1'; result(tmp_dif'left-2 downto 0) := (others => '0');
18                 else
19                   result(tmp_dif'left-1) := '0'; result(tmp_dif'left-2 downto 0) := (others => '1');
20                 end if;
21
22               end if;
23
24             end if;
25
26           else -- nemnegatív eset
27
28             if ( tmp_dif(tmp_dif'left) = '1' ) then overflow := '1';
29               if ( saturation_on = true ) then result := (result'range => '0'); end if;
30             end if;
31
32           end if;
33
34           else result := tmp_dif; -- az eredményt nem méretezzük át
35
36           if ( twos_complement_on = false and tmp_dif(tmp_dif'left) = '1' ) then overflow := '1';
37             if ( saturation_on = true ) then result := (result'range => '0'); end if;
38           end if;
39
40           end if;
```

4-158. ábra. *A kivonás művelet megvalósítása a fixpontos aritmetikai modulban*

esetén is (4-158. ábra: 36. sor).[153] Ha a felhasználó szaturációs viselkedést írt elő, akkor az abszolút hiba minimalizálása céljából ilyenkor a lehető legkisebb nemnegatív számot, vagyis a nullát adjuk eredményül (4-158. ábra: 37. sor). A másik különbség az összeadás és a kivonás között a redukált méretű nemnegatív eredményvektor szaturációs előállításánál tapasztalható; ha a méretredukció túlcsordulást okoz, akkor összeadás esetén a kimenet a lehető legnagyobb abszolútértékű nemnegatív érték kell hogy legyen, vagyis a kimenet minden bitje 1, kivonásnál viszont a szaturált eredménynek nullát kell adnia, vagyis a kimenet minden bitjét 0-ra kell állítani (4-158. ábra: 29. sor).

A 4-159. ábrán bemutatott szorzásművelet részletei némileg eltérnek az összeadásnál és kivonásnál látottaktól, de az alapstruktúra ugyanaz; előbb előállítjuk a megnövelt méretű – ezúttal az operandusok méreténél kétszer több bitből álló – szorzatot (*tmp_product*, 4-159. ábra: 2–5. sor), majd kezeljük a visszaméretezést az esetleges túlcsordulás detektálásával és a szaturáció megvalósításával (4-159. ábra: 8–37. sor). A *numeric_std* csomag nemcsak összeadás és kivonás, hanem szorzás operátort is rendelkezésünkre bocsát, a megnövelt méretű *tmp_product* vektor kiszámításában tehát nincs újdonság. Ha a felhasználó nem kéri az eredmény méretének redukálását, akkor a helyzet az összeadáshoz hasonlóan problémamentes, a *tmp_product* vektor maga a visszaadandó

[153] A „megnövelt méretű, aritmetikailag **garantáltan** helyes eredmény” szófordulat ettől az esettől eltekintve azért mégiscsak helyes.

VHDL

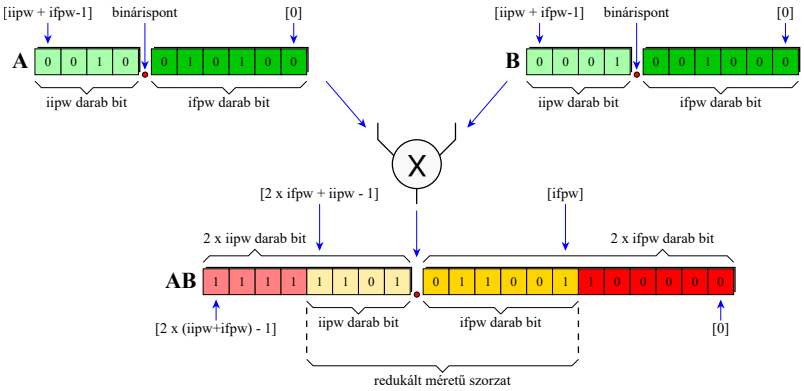
```
1  -- MULTIPLICATION --
2  when "010" => if ( twos_complement_on = true ) then
3                  tmp_product := std_logic_vector(signed(opa) * signed(opb));
4                else tmp_product := std_logic_vector(unsigned(opa) * unsigned(opb));
5                end if;
6
7            -- túlcsordulás detektálása és szaturáció...
8            if ( resize_on = true ) then result := tmp_product(2*ifpw+iipw-1 downto ifpw);
9
10           if ( twos_complement_on = true ) then
11
12             if ( (tmp_product(2*iw-1 downto 2*ifpw+iipw-1) /= zeros_a) and
13                  (tmp_product(2*iw-1 downto 2*ifpw+iipw-1) /= ones_a) ) then overflow := '1';
14
15             if ( saturation_on = true ) then
16
17               if ( tmp_product(tmp_product'left) = '1' ) then
18                 result := (result'range => '0'); result(result'left) := '1';
19               else
20                 result := (result'range => '1'); result(result'left) := '0';
21               end if;
22
23             end if;
24
25           end if;
26
27           else -- nemnegatív eset
28
29           if (tmp_product(2*iw-1 downto 2*ifpw+iipw) /= zeros_a(zeros_a'left downto 1)) then
30             overflow := '1';
31             if ( saturation_on = true ) then result := (result'range => '1'); end if;
32           end if;
33
34           end if;
35
36           else result := tmp_product; -- az eredményt nem méretezzük át
37           end if;
```

4-159. ábra. *A szorzás művelet megvalósítása a fixpontos aritmetikai modulban*

eredmény (4-159. ábra: 36. sor). A kimenetnek az operandusok méretére való átméretezése némileg bonyolultabb, mint az összeadás és a kivonás esetén. Az operandusok *iipw* darab egészrész és *ifpw* darab törtrész bitet tartalmaznak. Ennek megfelelően a megnövelt méretű, tehát aritmetikailag biztosan helyes eredmény úgy áll elő, hogy mind az egészrész, mind a törtrész kétszer annyi bitet tartalmaz, mint az operandusok. A *tmp_product* változó tehát egy *2 × (iipw + ifpw)* méretű vektor, amelynek felső *(2 × iipw)* bitje a szorzat egészrészét, alsó *(2 × ifpw)* bitje pedig annak törtrészét ábrázolja. A formátum, vagyis a binárispont relatív pozíciójának megváltoztatása nélkül a méretredukció úgy valósítható meg, hogy levágjuk az egészrész felső *iipw*, illetve a törtrész alsó *ifpw* bitjét (lásd 4-160. ábra, szögletes zárójelben az egyes vektorok bitjeinek indexeit tüntettük fel).

Ezt a műveletet fejezi ki a 4-159. ábra kódrészletének 8. sora. Az alsó bitek elvesztése „csupán" pontosságcsökkenést jelent, hiszen ezek a bitek 2 negatív hatványainak járulékait képviselik az ábrázolt szám értékében, a túlcsordulásban tehát nincs szerepük.[154] A túlcsordulás detektálása nemnegatív és kettes-komplemens kódolás esetén ezúttal is eltér, és ismét a nemnegatív eset az egyszerűbb. Ebben az esetben akkor van túlcsordulás, ha a levágott bitek között bármelyik 1 értékű. Ezt fejezik ki – a VHDL-re általában is jellemző bosszantó körülményességgel – a 4-159. ábra kódrészletének 29–30. sorai. Ilyenkor a szorzat nagyobb, mint amit a redukált méretű kimeneti

[154] A 4-160. ábra példájában -2,3125-öt szoroztunk össze 1,125-tel, ami -2,6015625. A szorzat ábrázolása 12 törtrész-bittel nem is gond, de ha az eredményt visszaméretezzük az operandusok formátumának megfelelően 6 törtrész-bitre, akkor a 2 mínusz hetedik hatványát képviselő 1-et elveszítjük, vagyis a redukált méretű eredmény nem pontos: -2,609375 ≠ -2,6015625. A hiba abszolútértéke $2^{(-7)}$.



4-160. ábra. *Redukált méretű szorzat előállítása*

vektorral ábrázolhatnánk, tehát ha a felhasználó szaturációs viselkedést írt elő, akkor a kimenet minden bitjét 1-re állítjuk. Kettes-komplemens kódolás esetén túlcsordulás akkor lép fel, ha az egészrészről levágott bitek nem egyeznek meg egytől egyig a megmaradó rész legfelső helyiértékű bitjével. Fogalmazhatunk úgy is, hogy akkor nincs túlcsordulás, ha a levágott részeknek kizárólag az előjel kiterjesztésében van szerepük (4-159. ábra: 12–13. sor). Ha szaturációs viselkedést vár el a felhasználó, akkor a kimeneti vektor generálása az összeadásnál és kivonásnál látható séma szerint alakul; negatív túlcsorduláskor a kimenet értéke a legkisebb lehetséges érték, tehát $1000..._2$, pozitív túlcsorduláskor pedig a legnagyobb, vagyis $0111..._2$ lesz (4-159. ábra: 15–23. sor).

Az összeadás vagy kivonás műveletek eredményeként előálló vektorok méretét az eredeti művelet operandusainak méretére csökkentő funkció leírását a 4-161. ábra mutatja be. Észrevehetjük, hogy ezt a problémát valójában már megoldottuk. Az összeadás és kivonás műveletek maguk is képesek eredményük méretének redukálására, ha a felhasználó a *resize_on* paraméter segítségével ezt kéri. Ebből adódik, hogy a 4-161. ábra kódrészlete csaknem teljesen megegyezik a 4-157. és 4-158. ábrák leírásainak 7. sor alatti részleteivel. A különbség az, hogy ebben az esetben nem egy belül előállított összegre vagy különbségre *(tmp_sum, tmp_dif)*, hanem az A operandusra *(opa)* vonatkozóan végezzük el az átméretezést és a szaturációkezelést. A 4-161. ábra 2. sorában az is jól látható, hogy az *opa* bemenetet összegnek és különbségnek tekintő funkciókat összevontuk, mert a két eset közötti eltérés kimerül abban, hogy nemnegatív operandus túlcsordulása esetén összegként értelmezett bemenetnél a kimenetet $111..._2$, különbségként értelmezett bemenet esetén pedig $000..._2$ értékre kell állítani (4-161. ábra: 27-28. sor).

A szorzatok eredményének méretét csökkentő, a 4-162. ábrán bemutatott funkció ugyancsak megegyezik azzal, amit a szorzás művelet eredményének redukálásakor már megvalósítottunk (4-159. ábra: 10–34.sor). Figyeljük meg, hogy a két leírás között csupán az a különbség, hogy míg a 4-159. ábra kódrészletében a szorzás művelet eredményén, vagyis a *tmp_product* változón dolgoztunk, addig a 4-162. ábra kódrészletében az átméretezendő változó maga az *opa* bemenet. További különbség figyelhető meg az *iipw, ifpw* és az ezekből származtatott *iw* konstansok használatában. Ennek oka, hogy ezek a konstansok mindkét esetben a bemenet formátumára vonatkoznak, viszont míg a szorzás művelet esetén az operandusok mérete kisebb, mint a teljes szorzaté, addig az átméretező funkció esetén a helyzet épp fordított, hiszen itt egy megnövelt méretű szorzat a bemenet. Ez a körülmény azonban a méretredukció logikájára nincs hatással.

VHDL

```vhdl
-- DOWNSIZE SUM / DIFFERENCE --
when "011" | "100" => result := opa(opa'left-1 downto 0);

                  -- túlcsordulás detektálása és szaturáció...
                  if ( twos_complement_on = true ) then

                   if ( opa(opa'left) /= opa(opa'left-1) ) then overflow := '1';

                    if ( saturation_on = true ) then

                     if ( opa(opa'left) = '1' ) then
                       result(opa'left-1) := '1'; result(opa'left-2 downto 0) := (others => '0');
                     else
                       result(opa'left-1) := '0'; result(opa'left-2 downto 0) := (others => '1');
                     end if;

                    end if;

                   end if;

                  else -- nemnegatív eset

                   if ( opa(opa'left) = '1' ) then overflow := '1';

                    if ( saturation_on = true ) then

                     if ( operation = "011" ) then result := (result'range => '1'); -- SUM
                     else result := (result'range => '0');                          -- DIF
                     end if;

                    end if;

                   end if;

                  end if;
```

4-161. ábra. *Az összegek/különbségek méretének redukálása a fixpontos aritmetikai modulban*

VHDL

```vhdl
-- DOWNSIZE PRODUCT --
when "101" => result := opa(ifpw+iipw/2-1 downto ifpw/2);

      -- túlcsordulás detektálása és szaturáció...
      if ( twos_complement_on = true ) then

       if ( (opa(iw-1 downto ifpw+iipw/2-1) /= zeros_b) and
            (opa(iw-1 downto ifpw+iipw/2-1) /= ones_b) ) then overflow := '1';

        if ( saturation_on = true ) then

         if ( opa(opa'left) = '1' ) then result := (result'range => '0'); result(result'left) := '1';
         else result := (result'range => '1'); result(result'left) := '0';
         end if;

        end if;

       end if;

      else -- nemnegatív eset

       if (opa(iw-1 downto ifpw+iipw/2) /= zeros_b(zeros_b'left downto 1)) then
         overflow := '1';
         if ( saturation_on = true ) then result := (result'range => '1'); end if;
       end if;

      end if;
```

4-162. ábra. *A szorzatok méretének redukálása a fixpontos aritmetikai modulban*

A fixpontos aritmetikai eljárás használata ◆ A fixpontos aritmetikai modul egy eljárásként van megfogalmazva, amelyet többféleképpen is beépíthetünk RTL-modelljeinkbe. A 4-163. ábra egy one-shot módszerrel leírt, adatfeldolgozó feladatot ellátó (PI szabályozó) állapotgép egy részletét mutatja be, ahol a végrehajtandó algoritmus aritmetikai műveletei magának az állapotgépnek a HDL-modelljébe vannak beágyazva (lásd még DRMSz: 4.8. fejezet, 4-90. ábra). Áramköri szempontból ilyenkor a fixpontos aritmetikai egységből létrejövő kombinációs hálózat tulajdonképpen az állapotgép kimeneti logikájának részévé válik.

VHDL

```
1  L_MAIN_FSM: process ( clk, rst )
2    [...]
3  begin
4    if ( rst = '1' ) then [...]
5    elsif ( rising_edge(clk) ) then
6
7      case state is
8
9        [...]
10
11       when s1 =>
12         -- P = Kp * error
13         fxp_arith_procedure("010", true, true, true, 8, 16, Kp, error, v_P, v_OF_P);
14         P <= v_P; overflow_P <= v_OF_P;
15
16         -- I = Ki * error_sum
17         fxp_arith_procedure("010", true, true, true, 8, 16, Ki, error_sum, v_I, v_OF_I);
18         I <= v_I; overflow_I <= v_OF_I;
19
20         state <= s2;
21
22       when s2 =>
23         -- actuator = P + I
24         fxp_arith_procedure("000", true, true, true, 8, 16, P, I, v_PI, v_OF_PI);
25         actuator_fxp <= v_PI; overflow_PI <= v_OF_PI;
26         [...]
27
28       [...]
29     end case;
30
31   end if;
32 end process;
```

4-163. ábra. *A fixpontos aritmetikai eljárás használata one-shot állapotgépmodellbe ágyazva*

Ez sok esetben kényelmes megoldás (lásd még 4-256. ábra: 36–39. sor), de ha adatfeldolgozó rendszerünket az összetettebb esetekben praktikusabb „vezérlőegység + műveletvégző egység” séma szerint szeretnénk felépíteni, akkor célszerű volna, ha a fixpontos aritmetikai egységet is egy önálló almodulként példányosíthatnánk egy magasabb hierarchiaszintű modul architektúrájának törzsében. A 4-164. ábra az aritmetikai eljárásnak egy afféle burkolómodulját mutatja be, amely a procedurálisan megfogalmazott logikát egy példányosítható almodullá alakítja.

A 4-164. ábrán az eljárás bizonyos bemeneteit (a műveletvégzés „beállításait”) szintézis-paraméterekké *(generic)*, más bemeneteit (az operandusokat) pedig a portlista részévé tettük. Ennek a különbségtételnek az áramkörszintézis szempontjából nagy jelentősége van. A 4-164. ábrán vázolt modell esetén az aritmetikai modul egy konkrét példánya csak úgy hozható létre, hogy az általa elvégzendő művelet *(operation)*, illetve annak beállításai is előre ismertek, a működő áramkör esetén már nem változnak. Ez azt jelenti, hogy a szintézis során valójában sosem jöhet létre egy olyan áramkör, amely az aritmetikai eljárás minden funkcióját megvalósítja. Mivel az elvégzendő műveletet kiválasztó *operation* bemenet egy szintézis-paraméter, valójában a 4-157–162. ábrákon bemutatott funkciókból – a szintézis folyamatának részét képező, ún. constant propagation

VHDL

```
entity fxp_arith_unit is
  generic (
    operation: std_logic_vector (2 downto 0);                  -- 000: ADDITION
    twos_complement_on, saturation_on, resize_on: boolean;     -- 001: SUBTRACTION
    iipw: integer range 1 to 64;                               -- 010: MULTIPLICATION
    ifpw: integer range 0 to 63                                -- 011: DOWNSIZE OPA AS SUM
  );                                                           -- 100: DOWNSIZE OPA AS DIFFERENCE
                                                               -- 101: DOWNSIZE OPA AS PRODUCT
  port (
    opa, opb:           in  std_logic_vector;
    result:             out std_logic_vector;
    overflow:           out std_logic
  );
end entity fxp_arith_unit;

architecture rtl of fxp_arith_unit is
begin

  process ( opa, opb )
    variable v_result:      std_logic_vector (result'left downto 0);
    variable v_overflow:    std_logic;
  begin

    fxp_arith_procedure(
      operation, twos_complement_on, saturation_on, resize_on, iipw, ifpw,
      opa, opb, v_result, v_overflow
    );

    result <= v_result;
    overflow <= v_overflow;

  end process;

end architecture rtl;
```

4-164. ábra. *A fixpontos aritmetikai eljárás átalakítása példányosítható modullá*

eljárásnak köszönhetően (lásd még DRMSz 8-32. ábra) – mindig csak az egyik valósul meg fizikailag. Továbbá, a kiválasztott műveletet leíró többszintű *if-else* szerkezet maga is jócskán leegyszerűsödik, mivel a feltétel mezőkben szereplő *twos_complement_on* és *saturation_on* kifejezések is meghatározott értékűek.[155]

A „beállítás vs. operandus” kategorizálás valójában önkényes, nincs elvi akadálya, hogy az eljárás minden bemenete a portlista részét képezze. Ebben az esetben azonban nem szabad elfelejtenünk, hogy a fixpontos aritmetikai modul meglehetősen összetett funkciót ír le ahhoz képest, hogy a HDL kód szerkezetéből adódóan csak kombinációs hálózatként valósítható meg. Ha a műveletvégzés beállításait is futásidőben szeretnénk változtatni – például azért, mert a fixpontos aritmetikai modult egy programot futtató gép általános célú aritmetikai egységeként használjuk, amelyben az elvégzendő műveletet a folyton változó utasításkód határozza meg –, akkor számítanunk kell rá, hogy az aritmetikai modul egy jókora kombinációs hálózat lesz, amelynek időzítési optimalizációja a lokális adatutak jelentős késleltetése miatt komoly fejfájást okozhat.[156]

[155] Vö. DRMSz: 4.6. alfejezet, 6-67. ábra: A szintézis- és futásidejű paraméterek hatása az áramköri szerkezetre.

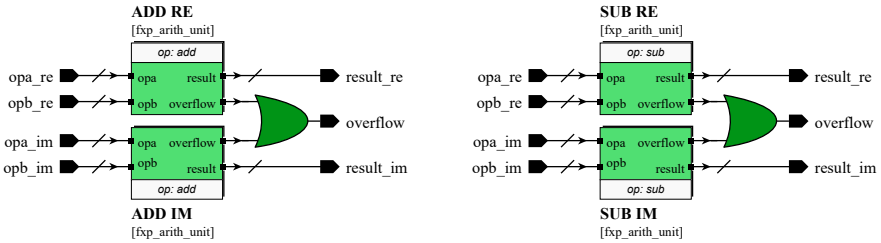
[156] Megjegyzendő, hogy az *operation* paraméter okozta *case*-szerkezetből és a túlcsordulás-detektálást és szaturációs logikát megvalósító, látszólag bonyolult feltételes szerkezetekből szintetizált áramköri részletek járuléka a teljes, sokszintű kombinációs hálózat késleltetéséhez viszonyítva nem számottevő. A nagy késleltetés inkább az aritmetikai műveleteket megvalósító operátorokból szintetizált kapuknak köszönhető. Összeadás esetén az FPGA áramkörök alapcelláinak carry-logikája némileg mérsékli ezt a problémát (lásd még DRMSz: 8-41. ábra), a szorzást pedig ideális esetben a szintézis-szoftverek időzítési szempontból optimalizált blokk-szorzókkal oldják meg (lásd még DRMSz: 7-12. ábra). Ha tehát a kiválasztott FPGA-ban ezek az erőforrások megtalálhatók, akkor a helyzet könnyen kezelhető. Ha olyan FPGA-val dolgozunk, amelyben nincsenek meg ezek az értékes összetevők, akkor nem várhatunk el gyors működést.

4.5.1.1. Komplex aritmetika

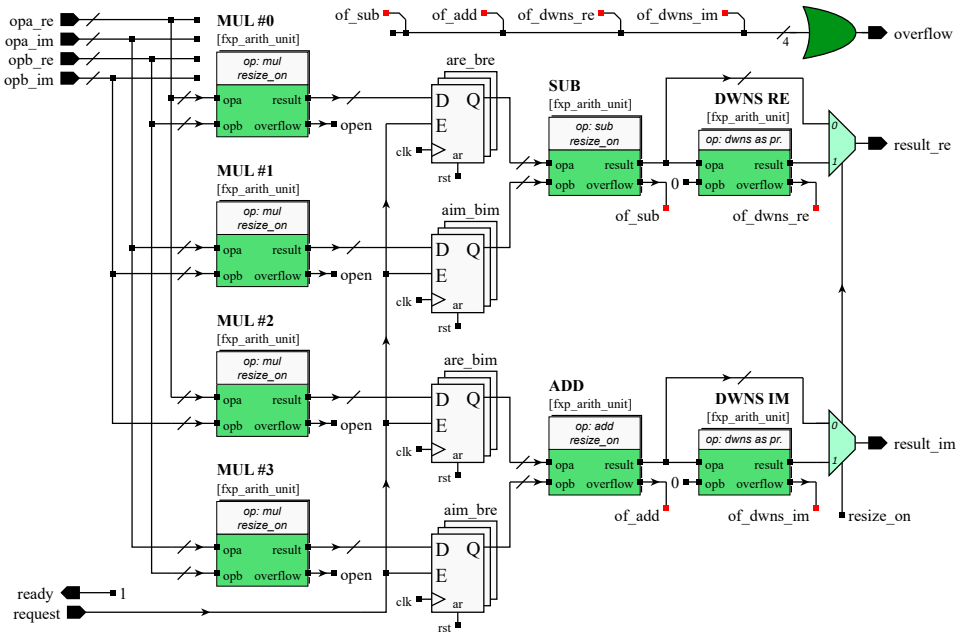
A fixpontos aritmetikai modul felhasználásával könnyen készíthetünk komplex számok kényelmes kezelésére alkalmas aritmetikai egységeket is. Az $(A_{Re} + jA_{Im})$ és $(B_{Re} + jB_{Im})$ alakban ábrázolt komplex számok összege és különbsége a következőképpen képezhető:

$$S = (A_{Re} \pm B_{Re}) + j(A_{Im} \pm B_{Im})$$

Az összeg vagy különbség valós és képzetes része tehát rendre az operandusok valós és képzetes részeinek összege vagy különbsége lesz. A komplex összeadó/kivonó egység tehát két-két fixpontos aritmetikai egységből létrehozható a 4-165. ábrán szemléltetett módon.



4-165. ábra. *Komplex összeadó és kivonó megvalósítása a fixpontos aritmetikai egység felhasználásával*



4-166. ábra. *Komplex szorzó megvalósítása a fixpontos aritmetikai egység felhasználásával*

Az $(A_{Re} + jA_{Im})$ és $(B_{Re} + jB_{Im})$ alakban ábrázolt komplex számok szorzata a következőképpen fejezhető ki:

$$P = (A_{Re}B_{Re} - A_{Im}B_{Im}) + j(A_{Re}B_{Im} + A_{Im}B_{Re})$$

A szorzat valós és képzetes részének kiszámításához tehát két-két valós (fixpontos) szorzás és egy-egy összeadás/kivonás szükséges. Elvileg nincs akadálya annak, hogy a komplex szorzót az összeadóhoz és a kivonóhoz hasonlóan kombinációs hálózatként valósítsuk meg, ekkor azonban fixpontos szorzó és összeadó/kivonó modulokat kell sorbakapcsolnunk, ami jelentős terjedési késleltetéssel, ezen keresztül pedig várhatóan alacsony maximális órajel-frekvenciával járna a beágyazó rendszer szempontjából. Célszerű tehát a szorzás és összeadás/kivonás műveletek közé egy regiszterréteget beiktatni a 4-166. ábrán látható módon.

A komplex szorzat kiszámításához létrehoztunk négy regisztert az $(A_{Re} \times B_{Re})$, $(A_{Im} \times B_{Im})$, $(A_{Re} \times B_{Im})$ és $(A_{Im} \times B_{Re})$ valós szorzatok tárolására. E regiszterekbe a fixpontos szorzók kimenetein az

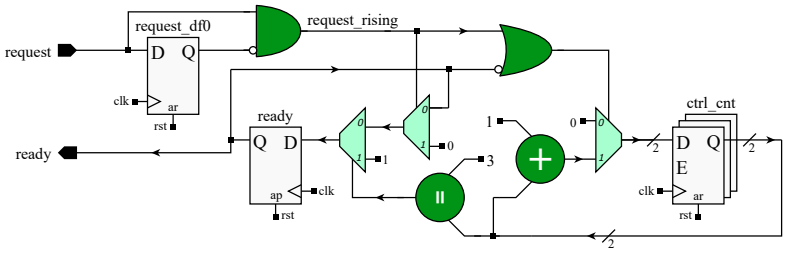
VHDL – szinkron kézfogásos interfész

```
1  L_HS_IF: process ( clk, rst ) begin
2    if ( rst = '1' ) then
3      request_df0 <= '0'; ready <= '1';
4    elsif ( rising_edge(clk) ) then
5
6      request_df0 <= request;
7
8      if ( ctrl_cnt = 3 ) then ready <= '1';
9      elsif ( request_rising = '1' ) then
10       ready <= '0';
11     end if;
12
13   end if;
14 end process;
15
16 request_rising <= request and not request_df0;
```

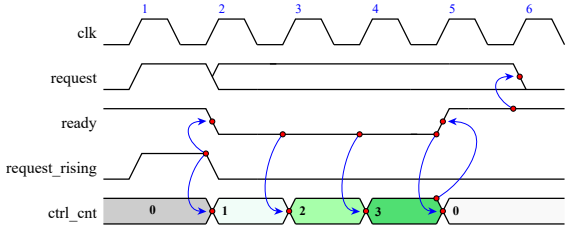
VHDL – számláló

```
1  L_CTRL_CNT: process ( clk, rst ) begin
2    if ( rst = '1' ) then
3      ctrl_cnt <= "00";
4    elsif ( rising_edge(clk) ) then
5
6      if ( request_rising = '1' or ready = '0' ) then
7        ctrl_cnt <= ctrl_cnt + 1;
8      else
9        ctrl_cnt <= "00";
10     end if;
11
12   end if;
13 end process;
```

Áramköri struktúra – számlálós vezérlő



Hullámforma



4-167. ábra. *A számlálós vezérlő megvalósítása a komplex szorzó modulban*

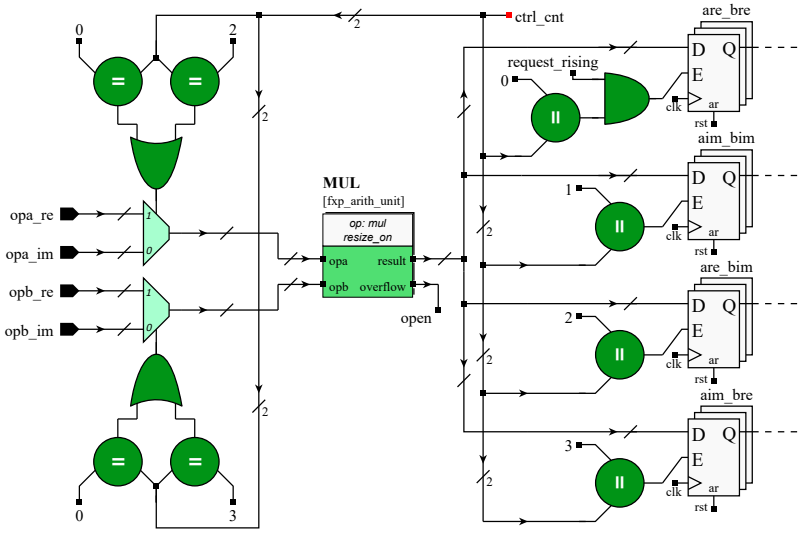
VHDL – operandusok multiplexálása

```
1  mux_mul_a <= opa_re when (
2    ctrl_cnt = 0 or ctrl_cnt = 2
3  ) else opa_im;
4
5  mux_mul_b <= opb_re when (
6    ctrl_cnt = 0 or ctrl_cnt = 3
7  ) else opb_im;
8
9  L_MUL: entity work.fxp_arith_unit(rtl)
10         generic map ( [...] )
11         port map (
12           opa      => mux_mul_a,
13           opb      => mux_mul_b,
14           result   => mul_result,
15           overflow => open
16         );
17
```

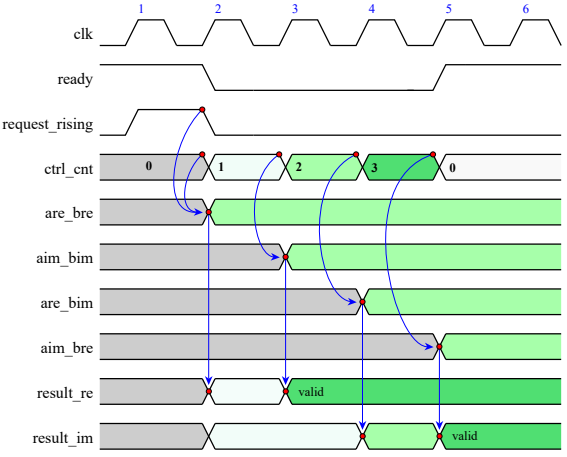
VHDL (folyt.) – valós szorzatok betöltése

```
18 L_PRODUCT_REGS: process ( clk, rst )
19 begin
20   if ( rst = '1' ) then [...]
21   elsif ( rising_edge(clk) ) then
22     if ( request_rising = '1' and ctrl_cnt = 0 ) then
23       are_bre <= mul_result;
24     end if;
25     if ( ctrl_cnt = 1 ) then aim_bim <= mul_result; end if;
26     if ( ctrl_cnt = 2 ) then are_bim <= mul_result; end if;
27     if ( ctrl_cnt = 3 ) then aim_bre <= mul_result; end if;
28   end if;
29 end process;
```

Áramköri struktúra



Hullámforma



4-168. ábra. *Műveletvégző elemek időmultiplexált felhasználása a számlálós vezérlővel*

operandusok megváltozásakor azonnal (aszinkron módon) előálló szorzatokat a *request* jel magas szintre állításával tudjuk betölteni. A szorzatpárokat összeadó/kivonó, valamint az eredményeket az operandusok méretére visszaskálázó aritmetikai egységek ugyancsak aszinkron logikák, tehát a regiszterek értékének megváltozása egyúttal a kimenet megváltozását is jelenti. A felhasználói interfész viselkedése így a nulla késleltetésű szinkron kézfogásos interfész viselkedésével egyenértékű (vö. 1-5. ábra).

A 4-166. ábrán látható komplex szorzó négy, párhuzamosan működő fixpontos szorzót igényel. Az erőforrások vezérelt újrafelhasználásával készíthetünk olyan megoldást is, amely a négy szorzást egymás után hajtja végre, így mindössze egyetlen fixpontos szorzóra van szükség. Ebben az esetben ugyanazt a számítást csak több órajelciklus alatt tudjuk elvégezni. Az erőforrások vezérléséhez használhatunk egyszerű számlálós vezérlőt a 4-167. ábrán szemléltetett módon. A *request* jel felfutóél-detektorának kimenete *(request_rising)* – a szinkron kézfogásos interfész-specifikáció elvárásainak megfelelően – nullázza a *ready* kimenetet, és egyúttal engedélyezi a számlálást az órajel második felfutóélénél (lásd 4-167. ábra jobb oldal: 6–7. sor). A számlálás engedélyezését az órajel harmadik felfutóélétől a *ready* jel alacsony állapota veszi át, majd amikor a számláló eléri a 3 értéket, akkor a következő órajel-felfutóélnél a *ready* jel ismét magas szintű lesz (lásd 4-167. ábra bal oldal: 8. sor). Ugyanebben a pillanatban a számláló átfordul, de mivel az órajel hatodik felfutóélénél az éldetektor nem jelez, a *ready* kimenet pedig magas szintű, ezért a számlálás megáll. A 0 → 1 → 2 → 3 → 0 számlálási ciklus tehát csak a *request* bemeneten előállított újabb felfutóéllel indítható el. A műveletvégző egységek időmultiplexált használatához immár rendelkezésünkre áll egy „állapotregiszter” *(ctrl_cnt)*, ami lehetővé teszi a számítás fázisai közötti különbségtételt. Ezúttal egyetlen szorzó modult használunk, amelynek bemeneteire a számláló különböző értékeinél más-más operandusokat juttatunk, így állítva elő a négy belső valós szorzatot egymás után, órajelciklusonként egyet-egyet (lásd 4-168. ábra).

Az áramkör többi része a 4-166. ábrán látottakkal megegyezik, tehát a szorzatok összegét és különbségét, valamint ezek redukált méretű változatát kiszámító aritmetikai egységek ezúttal is aszinkron módon állítják elő kimenetüket. A szorzatok megváltozásakor így az eredményvektorok is változnak, értékük – a szinkron kézfogásos interfész specifikációjával összhangban – a *ready* jel felfutásával egyidőben tekinthető véglegesnek.

4.5.2. Iteratív fixpontos szorzó

> ***Áttanulmányozásra javasolt alfejezetek:***
> ***5.3.1.*** *Bináris számábrázolás – Fixpontos számok*

Ahogy az on-chip memória-IP-k esetén, úgy az aritmetikai funkciókat megvalósító moduloknál is nehézségekbe ütközhetünk, ha a 2. fejezet elején rögzített alapelveknek megfelelő RTL-terveket szeretnénk készíteni. Széles körben újrafelhasználható, technológiafüggetlen IP-k tervezésekor rendszerint kompromisszumokra kényszerülünk, mert nem támaszkodhatunk a különféle gyártók ezernyi FPGA architektúrájának sajátosságaira, hanem olyan megoldásokra kell törekednünk, amelyek a lehető legtöbb gyártó eszközeivel megvalósíthatók. A 4.5.1. pontban bemutatott, fixpontos aritmetikai egységgel kapcsolatban elmondható, hogy gyakorlatilag bármilyen eszközre szintetizálható, de fontos hozzátenni, hogy a szorzás funkció esetén az ebből a HDL-leírásból

szintetizált áramkör mérete és számítási teljesítménye erősen függ attól, hogy az adott FPGA tartalmaz-e DSP-blokkokat. A DSP-blokkok – amellett, hogy rengetegféle eltérő felépítésű és képességű típus van belőlük – legalapvetőbb elemi alkatrésze a méret és időzítési tulajdonságok (számítási teljesítmény) szempontjából is optimalizált blokk-szorzó. A fixpontos aritmetikai egységünk ennél többet nem is igényel ahhoz, hogy hatékony adatfeldolgozó rendszert építhessünk belőle, ebből a szempontból tehát az IP-mag technológiafüggetlennek mondható. Ha azonban a céleszköz egyáltalán nem tartalmaz semmilyen DSP-blokkot, akkor a fixpontos aritmetikai egységünkből szintetizált áramkört a szintézis-szoftver kénytelen az FPGA általános célú alkatrészeiből (LUT-ok, vezetékezés és kapcsolóelemek) felépíteni. Az így létrehozott áramkörök mérete (és késleltetése) az operandusok ábrázolására szánt bitek számának növekedésével igen gyorsan nő, ami egy bizonyos operandus-méret felett gyakorlatilag használhatatlanná teszi ezt a megoldást.

Ebben a pontban egy olyan szorzó IP-magot készítünk, ami nem egy lépésben, hanem iteratív úton, több órajelciklus alatt, összeadás és léptetés műveletek sorozatával számítja ki két operandus szorzatát. E megoldás számítási teljesítmény szempontjából nyilván nem veszi fel a versenyt a blokk-szorzós változattal, de az ebből a leírásból szintetizált áramkör erőforrás-felhasználása csekély, a számábrázolásra szánt bitek számának növekedésével alig nő, továbbá nem igényel semmiféle DSP-blokkot.

Az iteratív szorzó alapjául szolgáló algoritmus működésének megértéséhez tekintsünk egy egyszerű példát. Keressük az $A = 6$ és $B = 11$, nemnegatív operandusok $P = A \times B$ szorzatát. Írjuk fel a P-t úgy, hogy a B operandust a bináris helyiértékes számábrázolás szerint kettő hatványainak súlyozott összegeként fejezzük ki:

$$P = A \times B = 6 \times [(\mathbf{1} \times 2^3) + (\mathbf{0} \times 2^2) + (\mathbf{1} \times 2^1) + (\mathbf{1} \times 2^0)]$$

A B operandust képviselő összeg minden tagját szorozzuk be az A operandussal:

$$P = 6 \times (\mathbf{1} \times 2^3) + 6 \times (\mathbf{0} \times 2^2) + 6 \times (\mathbf{1} \times 2^1) + 6 \times (\mathbf{1} \times 2^0) =$$

$$= \mathbf{1} \times (6 \times 2^3) + \mathbf{0} \times (6 \times 2^2) + \mathbf{1} \times (6 \times 2^1) + \mathbf{1} \times (6 \times 2^0) = \sum_{i=0}^{3} \boldsymbol{b_{B,i}} \times (6 \times 2^i)$$

A és B szorzata tehát kifejezhető elemi részszorzatok összegeként. A teljes szorzatot iteratív úton, a B operandus bitjein végiglépkedve számíthatjuk ki úgy, hogy ha az i-edik lépésben, az i-edik helyiértéken B bináris alakjában 1-es szerepel, akkor A-t kettő i-edik hatványával beszorozva hozzáadjuk az eggyel korábbi lépésben kapott részösszeghez. Ez a módszer látszólag nem oldotta meg az eredeti problémát, mely szerint a szorzás műveletet – megfelelő hardverelemek, DSP-blokkok híján – nem tudjuk hatékonyan megvalósítani, azonban vegyük figyelembe, hogy ebben az esetben mindig csak kettő valamely egész kitevőjű hatványával kell szoroznunk az A operandust, ami A bináris reprezentációjának balra léptetésével is megoldható. Az algoritmus tehát léptetés és összeadás műveletek sorozatából áll, ezért ezt a módszert shift&add szorzásnak hívjuk.[157] A fenti példában egész számokat használtunk az algoritmus működésének demonstrálásához, de – mivel a fixpontos számábrázolásban az egész és törtszámok ábrázolása csupán az egyes helyiértékeken

[157] A szemfüles olvasó alighanem észrevette, hogy a shift&add módszer valójában nem más, mint az általános iskolában elsajátított „írásbeli szorzás” kettes számrendszerbeli megfelelője.

lévő bináris számjegyek értelmezésében tér el (lásd 5.3. Függelék), ezért a módszer fixpontos törtszámok esetén is alkalmazható.

4.5.2.1. Viselkedési modell

Ahhoz, hogy a fenti szorzás algoritmust megvalósító áramkör RTL-modelljét elkészítsük, ezúttal is célszerű előbb egy viselkedési modellt alkotnunk. A 4-169. ábra pszeudokódja a shift&add algoritmus magjának egy lehetséges megfogalmazását mutatja be.

Pseudo – viselkedési modell

```
tmp_product <= 0
i <= 0

loop {
  if ( opb[i] == 1 ) {
    tmp_product <= tmp_product + opa
    if ( i < iw-1 ) {                  ; iw: input width, az operandusok bitszélessége
      i <= i + 1
      opa <= opa << 1
    } else break
  }
}

result <= tmp_product
```

4-169. ábra. *A shift&add szorzás algoritmus magja*

Ez az algoritmus remekül működik nemnegatív operandusok esetén, de a regularitás érdekében olyan IP-t szeretnénk készíteni, amely ugyanazokat a szolgáltatásokat nyújtja, mint a 4.5.1. pontban bemutatott, blokk-szorzókra épülő megoldás. A fenti algoritmust tehát ki kell egészítenünk

Pseudo – viselkedési modell

```
IDLE: wait until request
      ready <= 0; i <= 0; tmp_product <= 0; overflow <= 0

      opa_buf[2*iw-1:iw] <= 0 ; iw: input width, twc: two's complement
      if ( twos_complement_on == 1 and opa < 0 ) opa_buf[iw-1:0] <= twc.result(opa)
      else opa_buf[iw-1:0] <= opa

      if ( twos_complement_on == 1 and opb < 0 ) opb_buf <= twc.result(opb)
      else opb_buf <= opb

CORE: loop {
  if ( opb_buf[i] == 1 ) {
    tmp_product <= tmp_product + opa_buf
    if ( i < iw-1 ) {
      i <= i + 1
      opa_buf <= opa_buf << 1
    } else break
  }
}

SIGN_HANDLING: if ( twos_complement_on == 1 and opa[iw-1] /= opb[iw-1] )
                 tmp_product <= twc.result(tmp_product)

RESIZE: if ( resize_on == 1 ) {      ; iipw: input integer part width, ifpw: input fractional part width
          (result, overflow) <= resize_logic(tmp_product, iipw, ifpw, twos_complement_on, saturation_on)
        } else {
          result <= tmp_product; overflow <= 0
        }

ready <= 1; goto IDLE
```

4-170. ábra. *A kibővített shift&add algoritmus*

úgy, hogy negatív, kettes-komplemens kóddal ábrázolt operandusokkal is boldoguljon, illetve képes legyen a kimeneti vektor méretének redukálására és szaturáció megvalósítására. Az e képességeket is magában foglaló algoritmust a 4-170. ábrán láthatjuk.

A negatív operandusok kezelését az előjelek elő- és utófeldolgozásával oldjuk meg; a műveletvégzés elején az *opa* és *opb* operandusok abszolútértékét – amit negatív számok esetén a *twc* függvénnyel számítunk ki – rendre az *opa_buf* és *opb_buf* változókba töltjük be (4-170. ábra: 4–9. sor). Figyeljük meg, hogy az *opa_buf* változó mérete kétszerese az *opb_buf* változóénak, mivel ennek értékét fogjuk az algoritmus magjának végrehajtása során balra léptetni, és a léptetések során a felső helyiértékeken lévő bitek nem veszhetnek el (4-170. ábra: 4. sor). Az *opa_buf* és *opb_buf* változókon ezután végrehajtjuk az eredeti, nemnegatív operandusok feldolgozására alkalmas algoritmust (4-170. ábra: 11–19. sor), majd – ha a két eredeti operandus előjele különböző – az operandusok abszolútértékének szorzatát tároló változónak képezzük a kettes-komplemensét, vagyis a mínusz egyszeresét (4-170. ábra: 21–22. sor). Ezen a ponton tehát a *tmp_product* változó a két eredeti operandus *(opa, opb)* előjelhelyes szorzatát tartalmazza. Ezek után a *resize_on* paraméter értékétől függően végrehajtjuk a méretredukciót, amelynek eredménye a redukált méretű *result* szorzat és az esetleges túlcsordulást jelző *overflow* flag (4-170. ábra: 25. sor). A viselkedési modell a méretredukció és szaturációs viselkedés pontos megvalósítását diszkréten a *resize_logic* függvény mögé rejti, aminek az oka az, hogy ezt a műveletet pontosan ugyanazzal a kombinációs logikával fogjuk megvalósítani, amivel a fixpontos aritmetikai egység esetén tettük, így tehát a viselkedési modellezés szintjén szükségtelen ezt részletesen definiálnunk. Természetesen, ha a felhasználó nem igényli a szorzat méretének redukálását, akkor a *result* kimenet egyszerűen a *tmp_product* változó értékét kapja, az *overflow* flag értéke pedig nulla lesz (4-170. ábra: 27. sor).

4.5.2.2. RTL-modell

A shift&add szorzás algoritmusának birtokában nekifoghatunk a hardverleíró nyelvű RTL-modell elkészítésének. A teljes IP-katalógusra vonatkozóan regularitásra törekszünk, tehát a hasonló funkciót ellátó almodulok interfészét is igyekszünk hasonlóan kialakítani. Az iteratív fixpontos szorzó felhasználói interfészét a fixpontos aritmetikai eljárást tartalmazó, példányosítható modul felhasználói interfészéhez hasonlóan hozzuk létre (lásd 4-171. ábra, vö. 4-164. ábra).

VHDL

```
1  entity shift_and_add is
2  |  generic (
3  |  |  twos_complement_on, saturation_on, resize_on: boolean;
4  |  |  iipw: integer range 1 to 64;                          ; input integer part width
5  |  |  ifpw: integer range 0 to 63                           ; input fractional part width
6  |  );
7  |
8  |  port (
9  |  |  clk, rst: in  std_logic;
10 |  |  request:  in  std_logic;
11 |  |  ready:    out std_logic;
12 |  |  opa, opb: in  std_logic_vector;
13 |  |  result:   out std_logic_vector;
14 |  |  overflow: out std_logic
15 |  );
16 end entity shift_and_add;
```

4-171. ábra. *A shift&add szorzó modul felhasználói interfésze*

Az algoritmus viszonylag egyszerű, ezért a műveletvégző egység és a vezérlőegység HDL-modellbeli különválasztása helyett a teljes funkciót egyetlen, one-shot típusú állapotgéppel írjuk le, amelynek vezérlési állapotai magukban foglalják a műveletvégzésre vonatkozó, aritmetikai jellegű értékadásokat is (lásd még DRMSz 135. oldal). A HDL-modell vázlatát a 4-172. ábra szemlélteti.

VHDL

```
1  L_REQUEST_RISING_EDGE_DETECTOR: process ( clk, rst ) begin
2  | if ( rst = '1' ) then request_df0 <= '0';
3  | elsif ( rising_edge(clk) ) then request_df0 <= request;
4  | end if;
5  end process;
6  request_rising <= request and not request_df0;
7
8  L_MAIN_FSM: process ( clk, rst ) begin
9  | if ( rst = '1' ) then
10 | | state <= idle; ready <= '1';
11 | | opa_buf <= (others => '0'); opb_buf <= (others => '0');
12 | | tmp_product <= (others => '0'); result <= (result'range => '0'); overflow <= '0';
13 | elsif ( rising_edge(clk) ) then
14 | |
15 | | case state is
16 | | |
17 | | | when idle => if ( request_rising = '1' ) then [...]
18 | | |
19 | | | [...]
20 | | |
21 | | | when others => null;
22 | | |
23 | | end case;
24 | |
25 | end if;
26 end process;
```

4-172. ábra. *A shift&add szorzó modult leíró VHDL-modell vázlata*

A 1–6. sorokban leírt, a *request* jelre ültetett éldetektor biztosítja a szinkron kézfogásos interfész 14. ábrán látható specifikációja szerinti működést; a felhasználó az operandusok elhelyezése után – vagy azzal egyidejűleg – magas szintre húzza a *request* jelet, aminek hatására a vezérlő állapotgép megkezdi a szorzás algoritmusának végrehajtását.

A VHDL-leírás a viselkedési modell egyes fázisaiban megfogalmazott műveleteket egy-egy vezérlési állapothoz rendeli, kihasználva, hogy az egyes fázisokon belül a különböző belső változókra vonatkozó értékadások nem függenek egymástól, tehát egymással párhuzamosan, egyazon órajel-felfutóélnél elvégezhetők. A 4-173. ábra alsó részén a viselkedési modell 1-10. soraiban megfogalmazott működést végrehajtó vezérlési állapot látható.

A *request* jel felfutásának hatására tehát az állapotregiszterbe az órajel következő felfutóélénél a *core* érték íródik be (4-173. ábra VHDL-modell: 16. sor), miközben az *opa_buf* és *opb_buf* regiszterekben eltárolódnak az operandusok abszolútértékei. Az ezeken az abszolútértékeken elvégzett, előjel nélküli szorzás algoritmust a *core* vezérlési állapot önmagában megvalósítja (lásd 4-174. ábra). Az órajel következő *iw* db felfutóélénél az *opb_buf* regiszter i-edik bitjének értékétől függően a *tmp_product* regisztert megnöveljük az *opa_buf* regiszter aktuális értékével (4-174. ábra VHDL-modell: 19. sor). Ez utóbbi regiszter értékét ugyancsak minden órajel-felfutóélnél egy-egy balra léptetéssel frissítjük, így érve el, hogy az bármely pillanatban az eredeti operandus abszolútértékének 2^i-szerese legyen.

Miután a *core* állapotban végiglépkedtünk az *opb_buf* regiszter minden bitjén, a *tmp_product* regiszter az eredeti operandusok abszolútértékének szorzatát tartalmazza. Ekkor átlépünk a *sign_handling* vezérlési állapotba, ahol az eredeti, kettes-komplemens kóddal ábrázolt operandusok *(opa, opb)* legfelső helyiértékű bitjeinek különbözősége – vagyis eltérő előjelű operandusok – esetén

Pseudo – viselkedési modell

```
1  IDLE: wait until request
2        ready <= 0; i <= 0; tmp_product <= 0; overflow <= 0
3
4        opa_buf[2*iw-1:iw] <= 0 ; iw: input width, twc: two's complement
5        if ( twos_complement_on == 1 and opa < 0 ) opa_buf[iw-1:0] <= twc.result(opa)
6        else opa_buf[iw-1:0] <= opa
7
8        if ( twos_complement_on == 1 and opb < 0 )
9           opb_buf <= twc.result(opb)
10       else opb_buf <= opb
11 [...]
```

VHDL

```
1  when idle =>
2    if ( request_rising = '1' ) then
3    | ready <= '0'; i <= 0; tmp_product <= (others => '0'); overflow <= '0';
4    |
5    | opa_buf(opa_buf'left downto iw) <= (others => '0');
6    | if ( twos_complement_on = true and opa(opa'left) = '1' ) then
7    |   opa_buf(iw-1 downto 0) <= unsigned(not opa) + to_unsigned(1, iw);
8    | else opa_buf(iw-1 downto 0) <= unsigned(opa);
9    | end if;
10   |
11   | if ( twos_complement_on = true and opb(opb'left) = '1' ) then
12   |   opb_buf <= unsigned(not opb) + to_unsigned(1, iw);
13   | else opb_buf <= unsigned(opb);
14   | end if;
15   |
16   | state <= core;
17   end if;
18 [...]
```

4-173. ábra. *Az operandusok előjelének kezelése a számítás legelső lépésében*

Pseudo – viselkedési modell

```
12 CORE: loop {
13 | if ( opb_buf[i] == 1 ) {
14 | | tmp_product <= tmp_product + opa_buf
15 | | if ( i < iw-1 ) {                    ; iw: input width
16 | | |  i <= i + 1
17 | | |  opa_buf <= opa_buf << 1
18 | | } else break
19 | }
20 }
21
22 SIGN_HANDLING: if ( twos_complement_on == 1 and opa[iw-1] /= opb[iw-1] )
23                  tmp_product <= twc.result(tmp_product)
24 [...]
```

VHDL

```
19 when core => if ( opb_buf(i) = '1' ) then tmp_product <= tmp_product + opa_buf; end if;
20
21             if ( i < iw-1) ) then
22               i <= i + 1;
23               opa_buf <= shift_left(opa_buf, 1);
24             else state <= sign_handling;
25             end if;
26
27 when sign_handling => if ( twos_complement_on = true and opa(opa'left) /= opb(opb'left) ) then
28                       tmp_product <= (not tmp_product) + 1;
29                     end if;
30                     state <= resize;
31 [...]
```

4-174. ábra. *A nemnegatív operandusok szorzását és az előjel kezelését elvégző vezérlési állapotok*

a *tmp_product* regiszterbe betöltjük annak mínusz egyszeresét (4-174. ábra VHDL-modell: 28. sor), így előállítva az immár előjelhelyes, az operandusokéhoz képest kétszeres bitszélességű szorzatot. Az eredményvektor átméretezését és a szaturáció megvalósítását – ha azt a felhasználó a *resize_on* és

Pseudo – viselkedési modell

```
RESIZE: if ( resize_on == 1 ) {      ; iipw: input integer part width, ifpw: input fractional part width
          (result, overflow) <= resize_logic(tmp_product, iipw, ifpw, twos_complement_on, saturation_on)
        } else {
          result <= tmp_product; overflow <= 0
        }

        ready <= 1; goto IDLE
```

VHDL

```
when resize => if ( resize_on = true ) then result <= tmp_product(2*ifpw+iipw-1 downto ifpw);
               |
               | if ( twos_complement_on = true ) then
               | |
               | | if ( (tmp_product(2*iw-1 downto 2*ifpw+iipw-1) /= zeros) and
               | |    | (tmp_product(2*iw-1 downto 2*ifpw+iipw-1) /= ones) ) then overflow <= '1';
               | | |
               | | | if ( saturation_on = true ) then
               | | | |
               | | | | if ( tmp_product(tmp_product'left) = '1' ) then
               | | | |   result <= (result'range => '0'); result(result'left) <= '1';
               | | | | else
               | | | |   result <= (result'range => '1'); result(result'left) <= '0';
               | | | | end if;
               | | | |
               | | | end if;
               | | |
               | | end if;
               | |
               | else -- nemnegatív eset
               | |
               | | if (tmp_product(2*iw-1 downto 2*ifpw+iipw) /= zeros(zeros'left downto 1)) then
               | |   overflow <= '1';
               | |   if ( saturation_on = true ) then result <= (result'range => '1'); end if;
               | | end if;
               | |
               | end if;
               |
               else result <= tmp_product; -- az eredményt nem méretezzük át
               end if;
               ready <= '1'; state <= resize;
```

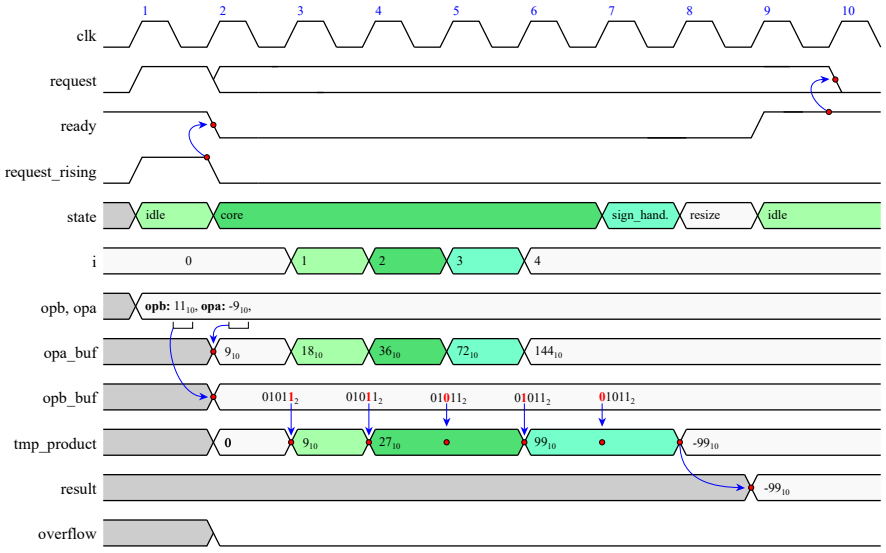
4-175. ábra. *Az eredményvektor méretredukcióját elvégző vezérlési állapot*

a *saturation_on* paraméterek beállításával kérte – a *resize* vezérlési állapotban végezzük el (lásd 4-175. ábra).

A méretredukció megvalósítása – tekintve, hogy ezúttal is nemnegatív vagy kettes-komplemens kódolású, negatív fixpontos számábrázolást használunk – teljesen megegyezik azzal, amit a kombinációs hálózatként megvalósított aritmetikai egység esetén már láthattunk (lásd 4-159. és 4-160. ábra).[158]

A 4-176. ábra egy egyszerű példán keresztül szemlélteti a shift&add szorzó modul működését (ötbites operandusok esetén; *iw = 5*). Ebben a példában egész számokat használunk a részműveletek ütemezésének szemléletes bemutatására, de fontos kiemelni, hogy az IP-mag az *iipw* és *ifpw* szintézis-paraméterekkel definiált formátumú törtszámok esetén is használható.

[158] Az eljárásként megvalósított aritmetikai egység VHDL-leírásában a szorzatot képviselő *tmp_product* egy változó *(variable)*, míg a shift&add szorzóban egy jel *(signal)*. Ezért a vonatkozó értékadások annyiban mégis különböznek, hogy előbbi esetben a :=, utóbbiban pedig a <= operátort használjuk, ennek azonban a funkció szempontjából nincs jelentősége.



4-176. ábra. *A shift&add algoritmus részműveleteinek ütemezése*

4.5.2.3. Órajel-szinkronizálás

A shift&add szorzó modul felhasználói interfészének működése megfelel az 1.2.1.1. pontban bemutatott szinkron kézfogásos interfész-specifikációnak, vagyis a 2-2. ábra órajel-szinkronizáló fokozata ebben az esetben is használható. Az általánosított $A_{i,S}$ és $B_{k,S}$ vektorok a 4-10. táblázat szerint feleltethetők meg a shift&add szorzó modul interfészjeleinek.

Általános CDC-logika	Shift&add szorzó modul
$A_{i,S}$	*opa, opb*
$B_{k,S}$	*result, overflow*

4-10. táblázat. *A shift&add szorzó modul és az általánosított CDC-logika jeleinek megfeleltetése*

4.5.3. Lebegőpontos aritmetikai egység

Áttanulmányozásra javasolt alfejezetek:

5.3.2. *Bináris számábrázolás – Lebegőpontos számok*

Bár a digitális jelfeldolgozásban sokkal gyakrabban használunk fixpontos számábrázolást, mint lebegőpontosat, akadhat olyan alkalmazás – rendszerint az ábrázolandó számtartomány mérete miatt –, amelyben kénytelenek vagyunk ehhez a megoldáshoz folyamodni. Ebben az alfejezetben egy, a fixpontos aritmetikai modullal azonos szolgáltatásokat (összeadás, kivonás, szorzás) nyújtó lebegőpontos aritmetikai egységet mutatunk be, amelynek működése – és emiatt RTL-modellje is – a fixpontos változaténál lényegesen összetettebb. Ez az összetettség magában a számábrázolási

módszerben, és az abból adódó, viszonylag bonyolult aritmetikai műveletekben gyökerezik. Tekintsük adottnak az *A* és *B*, lebegőpontos formátumban ábrázolt operandusokat:

$$A = \pm M_A \times 2^{E_A} \qquad B = \pm M_B \times 2^{E_B}$$

ahol M_A és M_B az egészre normalizált mantisszák, E_A és E_B pedig az eltolt karakterisztikák (lásd 5-53. táblázat). A két operandus *S* összegét/különbségét közös karakterisztikára hozás után annak kiemelésével képezhetjük:

$$A = \pm M'_A \times 2^{E_K} \qquad B = \pm M'_B \times 2^{E_K}$$

$$S = \pm(M'_A \pm M'_B) \times 2^{E_K}$$

Az M'_A és M'_B mantisszák az eredeti M_A és M_B mantisszák kettővel való szorzásával vagy osztásával képezhetők attól függően, hogy a közös karakterisztika az eredeti karakterisztikánál kisebb vagy nagyobb. A két operandus *P* szorzatát a mantisszák összeszorzásával és a karakterisztikák összeadásával kapjuk meg:

$$P = (\pm M_A \times \pm M_B) \times 2^{E_A + E_B}$$

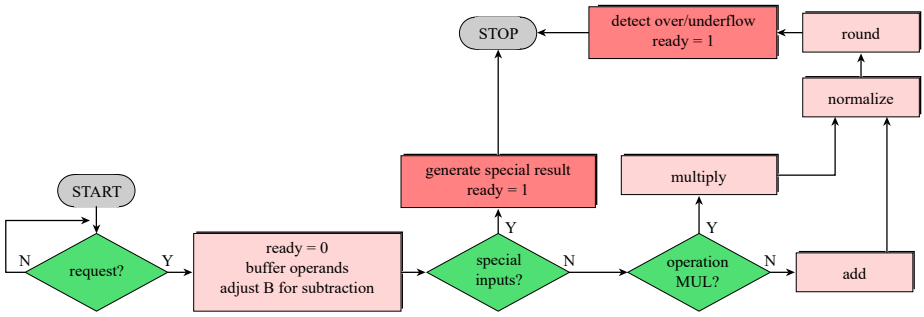
Az *S* összeg/különbség és a *P* szorzat kiszámítására képes IP-mag tervezésekor ezúttal is parametrikus megoldásra törekszünk, ami – a fixpontos aritmetikai egységnél látottakhoz hasonlóan – az egyes bitmezők méretének, vagyis a mantisszák és karakterisztikák ábrázolására szánt bitek számának testreszabhatóságát jelenti. A felhasználó az alkalmazási körülmények ismeretében hangolhatja be ezeket a paramétereket, így optimalizálva a leendő áramkör számábrázolási tartományát, pontosságát, méretét és fogyasztását.[159] A másik fontos RTL-tervezési alapelv, a regularitás jegyében az áramkört az 1.2.1.1. pontban részletezett, szinkron kézfogásos felhasználói interfésszel látjuk el.

4.5.3.1. Viselkedési modell

A mantisszák és karakterisztikák ábrázolására szánt bitek korlátozott száma sok apró részlet megfontolását teszi szükségessé, ezért az áramkör RTL-modelljének megtervezését egy részletes viselkedési modell elkészítésével kezdjük. A viselkedési modell lényegében az összeadás/kivonás és szorzás műveleteinek algoritmusa, amelynek részleteit pszeudokód segítségével írjuk le. Az áttekinthetőség kedvért fokozatokra osztott algoritmus vázlatát a 4-177. ábra mutatja be.

A viselkedési modell első fokozatának leírása a 4-178. ábrán látható. Bármelyik műveletről is legyen szó, az első lépés, hogy az operandusokat egy-egy belső tárolóba mentjük. A 4-178. ábra 5. sorában azt látjuk, hogy mivel az összeadás és a kivonás műveletek lényegében megegyeznek, valójában csak az összeadást valósítjuk meg, a kivonást pedig erre vezetjük vissza oly módon, hogy kivonás esetén a B operandus mínusz egyszeresét tároljuk el az *op_b_buf* belső változóba. Ettől

[159] Az itt bemutatott lebegőpontos aritmetikai egység tehát megfelelően felparaméterezve kompatibilis lesz az 5.3.2. pontban tárgyalt ANSI/IEEE Std 754-1985 szabvány által rögzített számábrázolási megoldással.



4-177. ábra. *A lebegőpontos aritmetikai egység által megvalósított algoritmus vázlata*

Pseudo – viselkedési modell

```
1  START: loop {
2    if ( request ) {
3      ready <= 0
4      op_a_buf <= op_a
5      if ( operation == sub ) op_b_buf <= -1 * op_b ; A kivonást összeadásra vezetjük vissza.
6      else op_b_buf <= op_b
7      break
8    }
9  }
```

4-178. ábra. *Az operandusok ideiglenes tárolókba helyezése és a B operandus előkészítése kivonásra*

a ponttól kezdve az *op_a_buf* és *op_b_buf* változókat tekintjük operandusoknak. Az algoritmus következő lépése a speciális bemeneti szimbólumok kezelése (lásd 4-179. ábra).

Pseudo – viselkedési modell

```
1  EXCEPTIONS: if (
2                op_a_buf == "+/-0" or op_b_buf == "+/-0" or
3                op_a_buf == "+/-Inf" or op_b_buf == "+/-Inf" or
4                op_a_buf == "NaN" or op_b_buf == "NaN"
5              ) {
6                result_sig <= flp_exception_codec.result[sig](ew, mw, operation, op_a_buf, op_b_buf)
7                result_exp <= flp_exception_codec.result[exp](ew, mw, operation, op_a_buf, op_b_buf)
8                result_man <= flp_exception_codec.result[man](ew, mw, operation, op_a_buf, op_b_buf)
9                goto STOP
10             } elsif ( operation == mul ) goto MUL
11             else goto ADD
```

4-179. ábra. *A speciális bemenetekre adott válaszokat egy dekóderrel állítjuk elő*
*Inf: **Inf**inity (végtelen), NaN: **N**ot **a** **N**umber (érvénytelen bináris kód)*

A lebegőpontos számábrázolás 5-54. táblázatban felsorolt speciális szimbólumait egy egyedi dekóderrel *(flp_exception_codec)* kezeljük.[160] Ezekkel a szimbólumokkal az aritmetikai műveletek elvégzésének nincs értelme, mert ezek csak konvencionális bináris reprezentációi bizonyos speciális értékeknek, mint a végtelen vagy a nulla. Az ANSI/IEEE Std 754-1985 szabvány táblázatos formában rögzíti, hogy a különböző műveleteknek milyen eredményt kell adnia e speciális bemenetek esetén (lásd 5-55–57. táblázatok). A viselkedési modellbeli *flp_exception_codec*

[160] A 4-179. ábra 2-4. soraiban lévő felsorolás szándékosan nem tartalmazza az 5-54. táblázatban speciális szimbólumnak tekintett, törtre normalizált, ún. „szubnormális" számokat. Az ilyen számokkal ui. a lebegőpontos IP-mag képes lesz műveleteket végezni.

függvény a karakterisztika és a mantissza mérete *(ew: exponent width, mw: mantissa width)*, a művelet *(operation)* és az operandusok ismeretében közvetlenül állítja elő az eredményvektor három komponensét; az előjelet *(sig)*, a karakterisztikát *(exp)* és a mantisszát *(man)*, természetesen a szabvány előírásainak megfelelően.[161] Ezeket az értékeket egy-egy belső változóban tároljuk el (4-179. ábra: 6–8. sor), amelyekből a műveletvégzés végén összefűzéssel képezzük a végeredményt. Az *flp_exception_codec* függvényt áramkörileg egy kombinációs hálózatként képzeljük el, amelynek HDL-modelljét az 5.3.2.1. Függelék tartalmazza. A 4-179. ábra kódrészletének 10–11. sorában látható, hogy az algoritmus ezen a ponton kettéágazik attól függően, hogy milyen műveletet hajtunk éppen végre. A kivonás itt már nem szerepel külön műveletként, hiszen a különbséget a B operandus mínusz eggyel való beszorzása után már az összeadás algoritmusa fogja előállítani (lásd 4-180. ábra).

Pseudo – viselkedési modell

```
1  ADD: exp_a <= ("00", op_a_buf[exp]) ; A nemnegatív karakterisztikákat
2       exp_b <= ("00", op_b_buf[exp]) ; 2 extra bittel kiegészítve tároljuk.
3
4       if ( exp_a >= exp_b ) {
5         exp_c <= exp_a + 2                ; A közös karakterisztika a nagyobbiknál 2-vel nagyobb.
6         m_offset_a <= 2                   ; A mantisszákat legalább 2-vel toljuk jobbra.
7         m_offset_b <= (exp_a - exp_b) + 2  ; Az eltolt mantisszák előjeles összegének előjele
8       } else {                            ; egyúttal a végeredmény előjele is.
9         exp_c <= exp_b + 2
10        m_offset_a <= (exp_b - exp_a) + 2
11        m_offset_b <= 2
12      }
13
14      if ( exp_a == 0 ) man_a <= ("0", op_a_buf[man], "00")  ; A mantisszák elé beszúrjuk a rejtett bitet.
15      else man_a <= ("1", op_a_buf[man], "00")               ; A törtre normalizált (szubnormális)
16                                                             ; számok esetén a rejtett bit értéke nulla.
17      if ( exp_b == 0 ) man_b <= ("0", op_b_buf[man], "00")  ; A mantisszákat az alsó helyiértékeken 2
18      else man_b <= ("1", op_b_buf[man], "00")               ; extra bittel (guard-bit) egészítjük ki
19                                                             ; a pontosság megőrzése érdekében.
20      man_a <= man_a >> m_offset_a
21      man_b <= man_b >> m_offset_b
22
23      if ( opa_buf[sig] == "1" ) man_a <= twc.result(man_a)  ; twc: two's complement
24      if ( opb_buf[sig] == "1" ) man_b <= twc.result(man_b)
25
26      man_c <= man_a + man_b
27
28      sig_c <= man_c[MSBI]                                   ; MSBI: most significant bit index
29      if ( man_c[MSBI] == "1" ) man_c <= twc.result(man_c)
30
31      goto NORMALIZE
```

4-180. ábra. *A lebegőpontos összeadás algoritmusa*

Az összeadást azzal kezdjük, hogy a két operandus karakterisztikáit olyan belső tárolókba mentjük, amelyek két-két extra bitet biztosítanak a legmagasabb helyiértékeken (4-180. ábra: 1–2. sor). E két bitre azért van szükség, mert az összeadás és szorzás végrehajtásakor magukon a nemnegatív karakterisztikákon is végzünk összeadást és kivonást, amelyek során keletkezhet átvitel. Ha csak az átvitel tényét szeretnénk detektálni, akkor elegendő lenne egyetlen extra bit. A másik extra bitre azért van szükség, hogy különbséget tudjunk tenni a karakterisztikák aritmetikájában

[161] A viselkedési modellben a vektorindexelést jelző *[...]* operátor argumentumában szereplő *sig*, *exp* és *man* kifejezések rendre a teljes lebegőpontos számot tartalmazó vektorok előjelre, karakterisztikára és mantisszára fenntartott mezőit jelölik (pl. a szabványos, szimpla pontosságú esetben a 32-bites vektor [31] bitje az előjel, [30:23] mezője az eltolt karakterisztika, [22:0] mezője pedig a rejtett bit nélküli mantissza; lásd 5-53. táblázat). Az MSBI (**M**ost **S**ignificant **B**it **I**ndex) kifejezés a vektorindexelés operátor argumentumában a legnagyobb helyiértékű bitre utal.

bekövetkező túl- és alulcsordulás között, ami az algoritmus záró lépésében a *+/-0* és *+/-Inf* eredmények közötti különbségtétel miatt lesz fontos. A 4-180. ábra 4–12. soraiban történik meg az összegképzéshez szükséges közös karakterisztika *(exp_c)* megállapítása. Látható, hogy a közös karakterisztika mindig kettővel nagyobb, mint a nagyobbik karakterisztika, aminek a magyarázata a következő:

- A közös karakterisztika miért mindig **nagyobb**, mint az operandusok karakterisztikái közül a nagyobb? – Ha a közös karakterisztika N-nel nagyobb, mint az adott operandus karakterisztikája, akkor ahhoz, hogy az operandus értéke ne változzon, a mantisszáját 2^N-nel osztani kell. A mantisszán a 2^N-nel való osztást N-nel való jobbra léptetéssel oldjuk meg (4-180. ábra: 20–21. sor). A jobbra léptetés miatt a mantissza legalsó helyiértékű bitjeit „veszítjük el", amivel – tekintve, hogy ezek a bitek kettőnek a legkisebb, negatív kitevőjű hatványainak együtthatóit képviselik – a pontosság romlik ugyan, de az érték maga alapvetően helyes marad. A pontosság javítása céljából a mantisszák belső tárolására használt változókban *(man_a, man_b)* a legalsó helyiértékeken 2 darab, ún. „guard-bitet" is használunk (4-180. ábra: 14–18. sor), amelyek két helyiértéknyit meg tudnak őrizni a mantissza alsó, kiléptetett végéből. Ha a közös karakterisztika kisebb lenne, mint az adott operandus karakterisztikája, akkor a mantisszát szükségszerűen balra léptetve a legmagasabb helyiértékű bitek vesznének el, amivel az ábrázolt értéknek immár semmi köze nem lenne az eredeti operandushoz.
- A közös karakterisztika miért épp **kettővel** nagyobb? – Az összeadás algoritmusban a rejtett bittel kiegészített és a közös karakterisztika szerint leosztott (jobbra léptetett) mantisszákon végezzük el az összeadás műveletet (4-180. ábra: 26. sor). Az operandusok a mantisszákat előjeles-abszolútértékes formában tartalmazzák, amelyek összegének tárolásakor egy extra bit mindenképpen szükséges az esetleges átvitel miatt. Ugyanakkor előjelhelyes eredményt csak akkor kapunk, ha a mantisszákat kettes-komplemens kódolásúvá alakítjuk át; pozitív operandusok esetén nem teszünk semmit, mivel a pozitív számok kettes-komplemens kódja önmaga, negatív operandusok esetén pedig a bitenként negált vektorhoz hozzáadunk egyet (4-180. ábra: 23–24. sor). A kettes-komplemens kódolás azonban adott bitszám esetén csak feleakkora tartomány lefedésére képes, ezért az információvesztés nélküli összeadáshoz egy további extra bitre van szükségünk. A két extra bit azt jelenti, hogy a mantisszák kettes-komplemens kódolásúvá alakítása előtt a *man_a* és *man_b* változók felső két bitjének mindenképp nullának kell lennie. Így biztosítható, hogy az előjeles összegképzés során nem veszik el információ, és hogy az előálló eredmény előjele egyúttal a végeredmény előjele is. Ez utóbbit a *sig_c* változóban tároljuk (4-180. ábra: 28. sor). A mantisszák első két nullája akkor biztosított, ha a közös karakterisztika nagysága legalább két jobbra léptetést kíván meg, vagyis a közös karakterisztika legalább kettővel nagyobb, mint a nagyobbik karakterisztika.

Az eltolt mantisszákon elvégezzük az előjelhelyes összeadást (4-180. ábra: 26. sor), azonban a feladat ezzel még nem ért véget. Az IP-magtól azt várjuk, hogy a végeredményt ugyanolyan formátumban szolgáltassa, ahogy az operandusokat kapta, vagyis végül egy egészre normalizált, előjeles-abszolútértékes mantisszát kell előállítania. Ezért, ha a *man_c* változó negatív (egyessel kezdődik), akkor a 4-180. ábra 29. sorában látható módon képezzük az abszolútértékét.

Az algoritmus utolsó lépései, vagyis a *man_c* és *exp_c* változókban tárolt eredmény normalizálása, valamint a kerekítés és a végeredményt képviselő bináris vektor *(result)* előállítása az összeadás és a szorzás esetén azonosak, ezért ezek bemutatása előtt vizsgáljuk meg, hogy a *sig_c*, az *exp_c* és a *man_c* változók értékeinek előállítása miképp zajlik le a szorzás művelet esetén. A viselkedési modellnek ezt a részletét a 4-181. ábrán láthatjuk.

Pseudo – viselkedési modell

```
1  MUL: sig_c <= op_a_buf[sig] xor op_b_buf[sig]
2
3       exp_a <= ("00", op_a_buf[exp])                      ; A két karakterisztika összege az eltolást
4       exp_b <= ("00", op_b_buf[exp])                      ; kétszer tartalmazza -> egy eltolásnyit
5       exp_c <= (exp_a + exp_b) - (2**(ew-1)-1)            ; le kell vonni az összegből.
6
7       if ( exp_a == 0 ) man_a <= ("0", op_a_buf[man], "00")  ; A mantisszák képzése az összeadásnál
8       else man_a <= ("1", op_a_buf[man], "00")            ; látottakhoz hasonló:
9                                                           ;   - rejtett bit hozzáfűzése elölről
10      if ( exp_b == 0 ) man_b <= ("0", op_b_buf[man], "00")  ;   - 2 guard-bit hozzáfűzése hátulról
11      else man_b <= ("1", op_b_buf[man], "00")
12
13      man_c <= (man_a[MSBI:2] * man_b[MSBI:2])[MSBI:mw-1];
14      exp_c <= exp_c + 1
15
16      goto NORMALIZE
```

4-181. ábra. *A lebegőpontos szorzás algoritmusa*

A szorzat előjelének kifejezése a lebegőpontos számformátum esetén nagyon egyszerű; képezzük a két operandus előjelbitjeinek *XOR* kapcsolatát, és ezt tároljuk el a *sig_c* változóban (4-181. ábra: 1. sor). Az operandusok karakterisztikáit tároló változókat az összeadásnál látottakkal megegyező módon kezeljük; a velük végzett aritmetikai műveletek információvesztés nélküli kezelése céljából két extra bitet használunk a legfelső helyiértékeken (4-181. ábra: 3–4. sor). A lebegőpontos szorzás definíciója szerint az eredmény karakterisztikáját a két operandus karakterisztikájának összege adja. Nem szabad azonban megfeledkeznünk arról, hogy az *exp_a* és *exp_b* változók valójában az ún. „eltolt" karakterisztikákat, vagyis az

$$A = \pm M_A \times 2^{E_A} \qquad B = \pm M_B \times 2^{E_B}$$

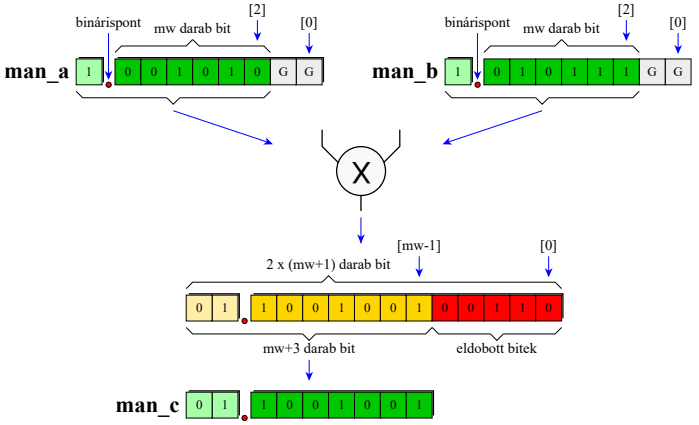
kifejezésekbeli E_A és E_B értékeknek az exponens-eltolással ($2^{(ew-1)}-1$-gyel) megnövelt értékeit tárolják, ahol *ew* a karakterisztikák tárolására szánt bitek száma. Ha tehát egyszerűen csak összeadjuk az *exp_a* és *exp_b* változókban tárolt értékeket, akkor *exp_c* ezt az eltolást kétszer fogja tartalmazni. Ahhoz, hogy a számábrázolás konzisztenciáját megtartsuk, és az *exp_c* változó az operandusokhoz hasonlóan a

$$P = (\pm M_A \times \pm M_B) \times 2^{E_A+E_B}$$

formula szerint az $(E_A + E_B)$ hatványkitevő eltolt értékét tárolja, az *(exp_a + exp_b)* összegből le kell vonnunk $2^{(ew-1)}-1$-et (4-181. ábra: 5. sor). Ezek után a mantisszák belső reprezentációjának *(man_a, man_b)* képzése az összeadásnál látottakhoz hasonlóan történik; az operandusok mantisszájához hozzáfűzzük a rejtett bitet a legmagasabb helyiértéken, a legalacsonyabb helyiértéken pedig bevezetjük a két guard-bitet (4-181. ábra: 7–11. sor).[162] A *man_a* és *man_b* változók szorzásakor a felhasznált erőforrások minimalizálása céljából a nulla értékű guard-biteket nem vesszük figyelembe

[162] A guard-bitek hozzáfűzésére szigorúan véve a szorzás miatt nem lenne szükség, hiszen a szorzat előállítása úgyis megduplázza a bitek számát, ezeket csak a *man_a* és *man_b* regiszterek kezelésének egységessége céljából hagytuk a leírásban.

(4-181. ábra: 13. sor). A szorzás két operandusa tehát egy-egy *(mw+1)*-bites, nemnegatív fixpontos számnak tekinthető, amelyek egészrészét 1, törtrészét pedig *mw* biten ábrázoltuk. Két ilyen formátumú szám összeszorzásakor egy *2×(mw+1)*-bites bináris vektort kapunk eredményül, amelyben az egészrész- és törtrész-bitek száma egyaránt duplája az operandusokra jellemzőnek (vö. 4-160. ábra). A *man_c* változó csupán *(mw+3)*-bites (rejtett bit + mantissza + 2 guard-bit), ezért a szorzat alsó bitjeit eldobjuk. A 4-182. ábrán az is jól látszik, hogy a *man_c* változó most két egészrész-bitet tartalmaz, tehát a *man_c* és *exp_c* változók együttese nem konzisztens; ti. az *exp_c* változónak azt az eltolt hatványkitevőt kell tartalmaznia, ami az egyetlen egészrész-bitet tartalmazó mantisszához tartozik. Ennek érdekében a *man_c* változó tartalmát „újraértelmezzük” úgy, hogy az *exp_c* változóhoz hozzáadunk egyet (4-181. ábra: 14. sor).



4-182. ábra. *A mantisszák belső reprezentációinak szorzása a lebegőpontos szorzás során*

Az összeadás és szorzás műveletek eddig bemutatott részműveletei vezethetnek olyan eredményre, amelyben a *man_c* változó felső bitjei nulla értékűek. Ez nem felel meg az elvárt formátumnak, ti. *man_c* legfelső bitje a rejtett bit, amely az egészre normalizált, elvárt formátum esetén mindig 1. A *man_c* változóban tárolt értéket tehát addig kell balra léptetnünk, ameddig a legfelső bit értéke 1 nem lesz. Ezt a folyamatot normalizálásnak nevezzük (lásd 4-183. ábra). A balra léptetés szükséges mértékét, vagyis a *man_c* változó felső helyiértékein lévő nullák számát a viselkedési modellben az *nlz_logic (**n**umber of **l**eading **z**eros)* névre keresztelt függvény állapítja meg, amit az *flp_exception_codec*-hez hasonlóan áramkörileg egy kombinációs hálózatként érdemes elképzelni.

Pseudo – viselkedési modell

```
1  NORMALIZE: nlz <= nlz_logic.result(man_c) ; number of leading zeros: a man_c regiszter felső
2             man_c <= man_c << nlz          ; helyiértékein lévő nullák száma
3             exp_c <= exp_c - nlz
4             goto ROUND
```

4-183. ábra. *A man_c és exp_c értékek normalizálása*

A léptetés mértékének megállapítása és *nlz* változóban való eltárolása (4-183. ábra: 1. sor) után tehát a *man_c* értékét balra léptetjük. Ne feledjük azonban, hogy ezzel a lépéssel egyúttal az *exp_c* és *man_c* változók által együttesen meghatározott értéket 2^{nlz}-szeresére növeljük, vagyis a karakterisztikát tároló *exp_c* változó értékét *nlz* értékével csökkentenünk kell (4-183. ábra: 3. sor).

Ezen a ponton a *sig_c*, *exp_c* és *man_c* változók a két operandus összegének vagy szorzatának megfelelő értékeket tárolják. A következő, az összeadás és a szorzás esetén egyformán végrehajtandó lépés a kerekítés, amikor a két guard-bitet felhasználva meghatározzuk a *man_c* változó pontosított, végleges értékét, egyúttal át is töltjük azt a végeredményt tároló *result_man* változóba (lásd 4-184. ábra).

Pseudo – viselkedési modell

```
ROUND: if ( man_c[2:0] == "110" or man_c[1:0] == "11" ) {
         result_man <= man_c[MSBI-1:2] + 1
         if ( result_man == 0 ) exp_c <= exp_c + 1
       } else result_man <= man_c[MSBI-1:2]
       goto OVER_UNDERFLOW
```

4-184. ábra. *A kerekítés művelete a lebegőpontos összeadás/szorzás algoritmusában*

Ha a guard-bitek értéke 00_2 vagy 01_2, akkor a *man_c* értékén nem változtatunk, ami a lefelé kerekítésnek felel meg. Ha a guard-bitek értéke 11_2, akkor mindig felfelé kerekítünk, vagyis a *man_c* guard-biteken felüli részéhez hozzáadunk egyet. Ha a guard-bitek értéke 10_2, akkor a kerekítés irányáról a guard-bitek feletti utolsó bit értéke alapján döntünk; egyes esetén felfelé, nulla esetén lefelé kerekítünk. Mivel a kerekítés irányát meghatározó bit sok-sok egymást követő számítás elvégzése során egyenlő arányban lesz nulla és egy, ezzel elkerülhető a kerekítési hibák felhalmozódása. A 4-184. ábra 3. sorában láthatjuk, hogy a felfelé kerekítéskor adódhat átvitel, ami a *man_c* változó átfordulását okozza. Ekkor – mivel az átvitel tényét a mantissza nem tudja rögzíteni – az *exp_c* értékét növeljük eggyel. Ennek megfelelően elvileg a *man_c* változót jobbra kellene léptetnünk, ami által az imént a bal oldalon „elvesző” átvitelbit visszakerülne a rejtett bit helyiértékére, ezt a lépést azonban elhagyjuk, hiszen a rejtett bit helyiértékén lévő bitet úgysem másoljuk a végeredményt tároló *result_man* változóba, azt mindig egynek tekintjük.

Akár összeadásról, akár szorzásról legyen szó, az algoritmus utolsó lépése az esetleges túl- vagy alulcsordulás detektálása. Az eddig elvégzett részműveletek során az eredmény karakterisztikáját tároló *exp_c* változót növeltük és csökkentettük is, de a végeredményben értékére ugyanazok a kötöttségek vonatkoznak, mint az operandusok esetén; teljesítenie kell a $1 \le exp_c \le 2^{ew}-2$ feltételt. Ha ez a feltétel nem teljesül, akkor – a megadott karakterisztikaméret esetén – az eredmény nem ábrázolható, túl- vagy alulcsordulás keletkezett, tehát a kimenetre (a *result_sig*, *result_exp* és *result_man* változókba) vagy a *+/-0*, vagy a *+/-Inf* szimbólumot kell írnunk (lásd 4-185. ábra).

Pseudo – viselkedési modell

```
OVER_UNDERFLOW: result_sig <= sig_c
                if ( signed(exp_c) > (2**ew)-2 ) { result_exp <= "111..." result_man <= "000..." }
                elsif ( signed(exp_c) < 1 ) { result_exp <= "000..." result_man <= "000..." }
                else result_exp <= exp_c[MSBI-2:0]

STOP: result <= (result_sig, result_exp, result_man)
      ready <= 1
      goto START
```

4-185. ábra. *Túl- vagy alulcsordulás detektálása az összeg/szorzat karakterisztikája alapján*

A túl- és alulcsordulás között a karakterisztikákat tároló belső változók felső helyiértékeire bevezetett két extra bit segítségével tudunk különbséget tenni; ha az *exp_c* értéke egy azt megnövelő művelet hatására vett fel az érvényes tartományon kívül eső értéket, akkor a két extra bit értéke 01_2 (mivel nincs olyan művelet, amely során annyival növelnénk *exp_c* értékét, hogy az a legfelső helyiértéken is átbillenést idézzen elő), ha azonban a tartományból való kiesést egy, az eltolt karakterisztikát nulla alá csökkentő lépés idézte elő, akkor a legfelső bit is egyes lesz. A legfelső bit

figyelembevételét úgy tudjuk egyszerűen kifejezni, hogy az *exp_c* változóban tárolt értéket előjeles, kettes-komplemens kódolásúnak tekintjük (4-185. ábra: 2. és 3. sor).

A 4-185. ábra 6. sorában végül a *result_sig*, *result_exp* és *result_man* változókból összefűzéssel képezzük a végeredményt *(result)*, amivel az összeadás vagy szorzás művelet befejezettnek tekinthető.

4.5.3.2. Műveletvégző egység

A lebegőpontos aritmetikai egység viselkedési modelljéből kiindulva az RTL-modellt ugyanazzal a szisztematikus módszerrel állítjuk elő, amit a 2.2. alfejezetben a *CPU* tervezésénél már alkalmaztunk. Ennek az eljárásnak az első lépése egy RTL-netlista, vagyis a viselkedési modell értékadásai által meghatározott, tároló és műveletvégző elemek között megvalósítandó adatkapcsolatok listájának létrehozása.

1. lépés - RTL-netlista létrehozása a viselkedési modellből ◆ Az RTL-netlistát alkotó adatkapcsolatokat azok balértékei alapján csoportokra bontjuk a jobb áttekinthetőség érdekében. Az egyes csoportokat külön-külön vizsgáljuk tovább, és készítjük el belőlük a műveletvégző egység vonatkozó darabkáját. A 4-186. ábra az RTL-netlista első csoportját mutatja be, amelyben az *op_a_buf*, az *op_b_buf*, az *exp_a*, valamint az *exp_b* változókra vonatkozó értékadásokat gyűjtöttük egybe.

Értékadások a viselkedési modellben

```
1  op_a_buf <= op_a
2
3  op_b_buf <= -1 * op_b
4  op_b_buf <= op_b
5
6  exp_a <= ("00", op_a_buf[exp])
7
8  exp_b <= ("00", op_b_buf[exp])
```

RTL-netlista

```
1  op_a_buf <= op_a
2
3  op_b_buf <= (not op_b[MSBI], op_b[MSBI-1:0])
4  op_b_buf <= op_b
5
6  exp_a <= ("00", op_a_buf[exp])
7
8  exp_b <= ("00", op_b_buf[exp])
```

4-186. ábra. *A lebegőpontos aritmetikai egység RTL-netlistája (1)*

Az *op_a_buf* változó a viselkedési modellben csupán egyetlen helyen szerepel balértékként, feladata az *op_a* bemenet eltárolása, vagyis a rá vonatkozó értékadást változtatás nélkül másoljuk át az RTL-netlistába. Az *op_b_buf* változó szerepe hasonló, de a kivonás művelet összeadásra való visszavezetése miatt nemcsak az *op_b* bemenetet, hanem annak mínusz egyszeresét is képesnek kell lennie eltárolni. A 4-186. ábra RTL-netlistájának 3. sorában azt a ritka esetet láthatjuk, amikor a lebegőpontos számábrázolás nem bonyolítja, hanem egyszerűsíti az életünket; egy ilyen formában adott szám mínusz egyszeresét egyszerűen az előjelbit negálásával képezhetjük. Az *exp_a* és *exp_b* változók az operandusok karakterisztikáit tárolják a legfelső helyiértékekre bevezetett két extra bittel kiegészítve.

Az *exp_c* változó központi szerepet tölt be a lebegőpontos műveletek végrehajtásában, ami a rá vonatkozó értékadások számából is jól látható (lásd 4-187. ábra). Az *exp_c* változóba az önmagán, valamint az *exp_a* és *exp_b* változókon végzett aritmetikai műveletek eredményét töltjük az összeadó/szorzó algoritmus különböző pontjain.

A tervezésnek ezen a pontján döntjük el, hogy az aritmetikai műveleteket milyen típusú erőforrások (aritmetikai áramkörök, multiplexerek) milyen kombinálásával akarjuk kezelni (vö. DRMSz: 4.1. fejezet: Erőforrás-allokáció). Többféle optimalizációs célt is kitűzhetünk; tervezhetünk minimális számú aritmetikai erőforrással vagy igyekezhetünk minimalizálni az ezek

Értékadások a viselkedési modellben

```
1  exp_c <= exp_a + 2
2  exp_c <= exp_b + 2
3  exp_c <= exp_c - nlz
4  exp_c <= (exp_a + exp_b) - (2**(ew-1)-1)
5  exp_c <= exp_c + 1
6  exp_c <= exp_c - 1
```

RTL-netlista

```
1  exp_add.opa <= exp_a
2  exp_add.opa <= exp_b
3  exp_add.opa <= exp_c
4  exp_add.op_b <= 2
5  exp_add.op_b <= 1
6  exp_add.op_b <= exp_b
7
8  exp_sub.op_a <= exp_c
9  exp_sub.op_a <= exp_add.result
10 exp_sub.op_b <= nlz
11 exp_sub.op_b <= (2**(ew-1)-1)
12 exp_sub.op_b <= 1
13
14 exp_c <= exp_add.result
15 exp_c <= exp_sub.result
```

4-187. ábra. *A lebegőpontos aritmetikai egység RTL-netlistája (2)*

bemeneteire kapcsolandó multiplexerek számát. Nemcsak a felhasznált erőforrások mennyiségének, hanem bizonyos időzítési tulajdonságoknak az optimalizálására is törekedhetünk. A végrehajtáshoz szükséges órajelciklusok számát csökkenthetjük több, párhuzamosan működtetett aritmetikai áramkör beépítésével, vagy a kritikus út várható késleltetését is minimalizálhatjuk akár több, kisebb méretű és pipeline regiszterekkel „felszeletelt" aritmetikai áramkör létrehozásával. A különféle optimalizációs szempontok egymásnak persze ellentmondanak, így a feladat a megfelelő kompromisszum megtalálása.[163] A 4-187. ábra RTL-netlistájában egy olyan megoldást láthatunk, amely a viselkedési modell bármely értékadását egyetlen órajelciklus alatt képes végrehajtani, és ehhez egy összeadó és egy kivonó áramkört használ. Ez nem kevesebb, mint öt multiplexer beépítését igényli majd, ráadásul – ahogy az az RTL-netlista 9. sorából is jól látható – a két aritmetikai áramkör sorba lesz kapcsolva. A ciklusszám, vagyis az adott művelet elvégzéséhez szükséges órajelciklusok száma tehát kedvező, ugyanakkor az ezeket az értékadásokat végrehajtó részáramkör kritikus útja három multiplexert és két aritmetikai áramkört is tartalmaz majd (részletesen lásd 4-199. ábra), ami nagy késleltetést, ezáltal alacsony maximális órajel-frekvenciát okozhat. Vegyük azonban figyelembe, hogy a műveletvégzés a viszonylag kevés biten ábrázolt karakterisztikákon történik, vagyis ezek az aritmetikai áramkörök aránylag kicsik lesznek. Fogalmazhatunk úgy is, hogy várhatóan lesz ennél nagyobb késleltetésű adatút is, hiszen a mantisszákon is kell majd műveleteket végezni és azokat tipikusan sokkal több biten ábrázoljuk, így a karakterisztikák aritmetikai áramköreit nem érdemes a lokális adatutak késleltetésére optimalizálni.[164]

A 4-188. ábra az összeadás algoritmusban a közöskarakterisztikára-hozáshoz szükséges eltolások mértékét számontartó *m_offset_a* és *m_offset_b* változók értékadásait és az ezeket megvalósító RTL-netlistát mutatja be.

Az eltolás mértékét ugyancsak ki kell számítanunk a két operandus karakterisztikái alapján, vagyis ezúttal is aritmetikai áramkörökre lesz szükség. A 4-188. ábra RTL-netlistájából látható, hogy nem használjuk újra a karakterisztikáknál bevezetett aritmetikai egységeket – bár ennek

[163] Újrafelhasználható IP-magok esetén a helyzet nem egyszerű, mivel nem tudhatjuk előre, hogy a leendő felhasználó mire akarja majd optimalizálni a saját rendszerét, amelybe a mi modulunkat beépíti. Ha az adott architekturális döntés jelentősen befolyásolja az IP-magnak valamely, RTL-tervezési szempontból fontos tulajdonságát (számítási teljesítmény, erőforrás-felhasználás, kritikus út késleltetése), akkor az adott IP különféle változatait készítjük el. Így születnek az egyazon IP-nak a teljesítményre, órajel-frekvenciára, fogyasztásra stb. optimalizált variánsai, amelyek az itt tárgyalt, ún. mikroarchitekturális sajátosságokban térnek el egymástól.

[164] A lebegőpontos aritmetikai egységek működését tárgyaló források sokszor a „kis ALU" és „nagy ALU" kifejezésekkel hivatkoznak a karakterisztikák és a mantisszák aritmetikai műveleteit végző áramkörökre.

Értékadások a viselkedési modellben

```
1  m_offset_a <= 2
2  m_offset_a <= (exp_b - exp_a) + 2
3
4  m_offset_b <= (exp_a - exp_b) + 2
5  m_offset_b <= 2
```

RTL-netlista

```
1  m_offset_sub.op_a <= exp_b
2  m_offset_sub.op_a <= exp_a
3  m_offset_sub.op_b <= exp_a
4  m_offset_sub.op_b <= exp_b
5
6  m_offset_add.op_a <= m_offset_sub.result
7  m_offset_add.op_b <= 2
8
9  m_offset_a <= 2
10 m_offset_a <= m_offset_add.result
11
12 m_offset_b <= 2
13 m_offset_b <= m_offset_add.result
```

4-188. ábra. *A lebegőpontos aritmetikai egység RTL-netlistája (3)*

nem lenne semmi akadálya –, hanem újakat hozunk létre, amivel lehetővé válik a közös karakterisztika és az eltolások egyidejű kiszámítása.

A 4-189. ábrán azt látjuk, hogy a két – a rejtett bittel és a legalacsonyabb helyiértékekre bevezetett két extra bittel kiegészített – mantisszát a viselkedési modellben megfogalmazottak alapján ugyanazoknak a műveleteknek vetjük alá.

Értékadások a viselkedési modellben

```
1  man_a <= ("0", op_a_buf[man], "00")
2  man_a <= ("1", op_a_buf[man], "00")
3  man_a <= man_a >> m_offset_a
4  man_a <= twc.result(man_a)
5
6  man_b <= ("0", op_b_buf[man], "00")
7  man_b <= ("1", op_b_buf[man], "00")
8  man_b <= man_b >> m_offset_b
9  man_b <= twc.result(man_b)
```

RTL-netlista

```
1  man_a_shr.data <= man_a
2  man_a_shr.steps <= m_offset_a
3
4  man_a_twc.op <= man_a
5
6  man_a <= ("0", op_a_buf[man], "00")
7  man_a <= ("1", op_a_buf[man], "00")
8  man_a <= man_a_shr.result
9  man_a <= man_a_twc.result
10
11 man_b_shr.data <= man_b
12 man_b_shr.steps <= m_offset_b
13
14 man_b_twc.op <= man_b
15
16 man_b <= ("0", op_b_buf[man], "00")
17 man_b <= ("1", op_b_buf[man], "00")
18 man_b <= man_b_shr.result
19 man_b <= man_b_twc.result
```

4-189. ábra. *A lebegőpontos aritmetikai egység RTL-netlistája (4)*

A két elvégzendő aritmetikai művelettípus a jobbra léptetés és a ketteskomplemens-képzés. Az RTL-netlistában látható, hogy két-két külön aritmetikai áramkört használunk ezek magvalósítására, vagyis a két mantissza párhuzamosan lesz kezelhető, ami nyilván kétszer annyi erőforrást igényel, ugyanakkor egyszerűbb vezérlést és gyorsabb műveletvégzést tesz majd lehetővé.

A 4-190. ábrán az összeg/szorzat mantisszáját *(man_c)* és előjelét *(sig_c)* tároló változókra, valamint a normalizálás lépésben a *man_c* változó legmagasabb helyiértékein álló nullák számát tároló változóra *(nlz)* vonatkozó értékadásokat és az azokat lehetővé tévő RTL-netlistát láthatjuk.

Az ártalmatlannak tűnő ketteskomplemens-képzés és összeadás, valamint – különösképpen – a szorzás külön figyelmet érdemel, mert ezeknek az aritmetikai áramköröknek az operandusai már a jellemzően sok biten ábrázolt mantisszák. Ezeket az aritmetikai funkciókat egyszerű kombinációs hálózatokként megvalósítva számíthatunk arra, hogy a teljes lebegőpontos aritmetikai egységre értelmezett kritikus út itt alakul majd ki, ami azt jelenti, hogy ha célunk a magas órajel-frekvencia elérése – akár a megnövekedett ciklusszám árán is –, akkor érdemes lehet ezeket a műveleteket

Értékadások a viselkedési modellben

```
1  man_c <= man_a + man_b
2  man_c <= twc.result(man_c)
3  man_c <= man_c << nlz
4  man_c <= (man_a[MSBI:2] * man_b[MSBI:2])[MSBI:mw-1]
5
6  sig_c <= man_c[MSBI]
7  sig_c <= op_a_buf[sig] xor op_b_buf[sig]
8
9  nlz <= nlz_logic.result(man_c)
```

RTL-netlista

```
1  man_c_add.op_a <= man_a
2  man_c_add.op_b <= man_b
3
4  man_c_twc.op <= man_c
5
6  man_c_shl.data <= man_c
7  man_c_shl.steps <= nlz
8
9  man_c_mul.op_a <= man_a[MSBI:2]
10 man_c_mul.op_b <= man_b[MSBI:2]
11
12 man_c <= man_c_add.result
13 man_c <= man_c_twc.result
14 man_c <= man_c_shl.result
15 man_c <= man_c_mul.result[MSBI:mw-1]
16
17 sig_c <= man_c[MSBI]
18 sig_c <= op_a_buf[sig] xor op_b_buf[sig]
19
20 nlz_logic.op <= man_c
21 nlz <= nlz_logic.result
```

4-190. ábra. *A lebegőpontos aritmetikai egység RTL-netlistája (5)*

részösszegek/részszorzatok előállításával és az azokat tároló, később regiszterekként megvalósított belső változók közbeiktatásával több fokozatra bontani. Az itt alkalmazott, a műveletvégző egység szisztematikus előállítását célzó tervezési eljárás során azonban ehhez nem szükségképpen kell a 4-190. ábra RTL-netlistáján bármit is módosítanunk. Ezek a leírások egyelőre csak afféle segédeszközök a műveletvégző egység részletes HDL-modelljének kidolgozásához. A bennük található függvényszerű összetevők (mint a *man_c_add*, *man_c_mul* és *man_c_twc*) aritmetikai műveleteket végző egységek absztrakt szimbólumai csupán, amelyek bármilyen konkrét HDL megvalósítást takarhatnak, akár olyat is, amelyben az adott művelet végrehajtását több órajelciklus alatt, pipeline regiszterek közbeiktatásával oldjuk meg. Ezeknek a részleteknek a kidolgozása a tervezési folyamat következő fázisának a feladata.

A 4-191. ábra a lebegőpontos összeadás/szorzás végeredményének három összetevőjét külön-külön tároló változókra vonatkozó értékadásokat és RTL-netlistát mutatja. Figyeljük meg, hogy a *man_c* változó értékének inkrementálására – amelynek végrehajtása a kerekítés során esetleg előforduló túlcsordulás miatt válhat szükségessé (lásd 4-184. ábra: 2. sor) – újabb összeadó egységet vezetünk be, annak ellenére, hogy erre a célra a *man_c_add* összeadót is használhatnánk. Láthatóan tehát nem spórolunk az aritmetikai elemekkel, aminek két oka van; egyrészt az IP-magot FPGA technológián tervezzük megvalósítani, ahol az összeadó „olcsó” erőforrás abban az értelemben, hogy sok van belőle, másrészt a *man_c_add* összeadó megosztott használata bemeneteinek multiplexálását igényelné, ami tovább növelné a *man_c* bemenetét meghajtó, amúgy is méretes kombinációs hálózat késleltetését.

2. lépés – Multiplexerek beiktatása az RTL-netlistába ◆ Az RTL-netlistában sok, egymástól független, azonos balértékű bejegyzés van. RTL-terveinkben – az FPGA technológia sajátosságai miatt[165] kerüljük a háromállapotú buszok használatát, ezért a többszörös hozzárendeléseket multiplexerek beiktatásával kell feloldanunk. A 4-192–197. ábrák a multiplexerekkel kiegészített RTL-netlistát mutatják be.

[165] Az FPGA áramkörök csak I/O buffereikben tartalmaznak háromállapotú (tri-state) meghajtókat, a belső logikák kialakításakor ezt a megoldást nem használjuk.

Értékadások a viselkedési modellben

```
result_sig <= flp_exception_codec.result[sig](ew, mw, operation, op_a_buf, op_b_buf)
result_sig <= sig_c

result_exp <= flp_exception_codec.result[exp](ew, mw, operation, op_a_buf, op_b_buf)
result_exp <= "111..."
result_exp <= "000..."
result_exp <= exp_c[MSBI-2:0]

result_man <= flp_exception_codec.result[man](ew, mw, operation, op_a_buf, op_b_buf)
result_man <= man_c[MSBI-1:2] + 1
result_man <= man_c[MSBI-1:2]
result_man <= "000..."

result <= (result_sig, result_exp, result_man)
```

RTL-netlista

```
flp_exception_codec.ew <= ew; flp_exception_codec.mw <= mw; flp_exception_codec.operation <= operation
flp_exception_codec.op_a <= op_a_buf; flp_exception_codec.op_b <= op_b_buf

result_sig <= flp_exception_codec.result[sig]; result_sig <= sig_c

result_exp <= flp_exception_codec.result[exp]; result_exp <= exp_c[MSBI-2:0]
result_exp <= "111..."; result_exp <= "000..."

resman_add.op_a <= man_c[MSBI-1:2]; resman_add.op_b = 1

result_man <= flp_exception_codec.result[man]; result_man <= resman_add.result
result_man <= man_c[MSBI-1:2]; result_man <= "000..."

result <= (result_sig, result_exp, result_man)
```

4-191. ábra. *A lebegőpontos aritmetikai egység RTL-netlistája (6)*

RTL-netlista

```
op_a_buf <= op_a

op_b_buf <= (not op_b[MSBI], op_b[MSBI-1:0])
op_b_buf <= op_b

exp_a <= ("00", op_a_buf[exp])

exp_b <= ("00", op_b_buf[exp])
```

RTL-netlista multiplexerekkel

```
op_a_buf <= op_a

mux_op_b_buf.din0 <= (not op_b[MSBI], op_b[MSBI-1:0])
mux_op_b_buf.din1 <= op_b
op_b_buf <= mux_op_b_buf.dout

exp_a <= ("00", op_a_buf[exp])

exp_b <= ("00", op_b_buf[exp])
```

4-192. ábra. *Multiplexerek beiktatása a lebegőpontos aritmetikai egység műveletvégző egységébe (1)*

RTL-netlista

```
exp_c <= exp_a + 2
exp_c <= exp_b + 2
exp_c <= exp_c - nlz
exp_c <= (exp_a + exp_b) - (2**(ew-1)-1)
exp_c <= exp_c + 1
exp_c <= exp_c - 1
```

RTL-netlista multiplexerekkel

```
mux_exp_add_op_a.din0 <= exp_a
mux_exp_add_op_a.din1 <= exp_b
mux_exp_add_op_a.din2 <= exp_c
exp_add.op_a <= mux_exp_add_op_a.dout
mux_exp_add_op_b.din0 <= 2
mux_exp_add_op_b.din1 <= 1
mux_exp_add_op_b.din2 <= exp_b
exp_add.op_b <= mux_exp_add_op_b.dout

mux_exp_sub_op_a.din0 <= exp_c
mux_exp_sub_op_a.din1 <= exp_add.result
exp_sub.op_a <= mux_exp_sub_op_a.dout
mux_exp_sub_op_b.din0 <= nlz
mux_exp_sub_op_b.din1 <= (2**(ew-1)-1)
mux_exp_sub_op_b.din2 <= 1
exp_sub.op_b <= mux_exp_sub_op_b.dout

mux_exp_c.din0 <= exp_add.result
mux_exp_c.din1 <= exp_sub.result
exp_c <= mux_exp_c.dout
```

4-193. ábra. *Multiplexerek beiktatása a lebegőpontos aritmetikai egység műveletvégző egységébe (2)*

RTL-netlista

```
m_offset_a <= 2
m_offset_a <= (exp_b - exp_a) + 2

m_offset_b <= (exp_a - exp_b) + 2
m_offset_b <= 2
```

RTL-netlista multiplexerekkel

```
mux_m_offset_sub_op_a.din0 <= exp_b
mux_m_offset_sub_op_a.din1 <= exp_a
m_offset_sub.op_a <= mux_m_offset_sub_op_a.dout
mux_m_offset_sub_op_b.din0 <= exp_a
mux_m_offset_sub_op_b.din1 <= exp_b
m_offset_sub.op_b <= mux_m_offset_sub_op_b.dout

m_offset_add.op_a <= m_offset_sub.result
m_offset_add.op_b <= 2

mux_m_offset_a.din0 <= 2
mux_m_offset_a.din1 <= m_offset_add.result
m_offset_a <= mux_m_offset_a.dout

mux_m_offset_b.din0 <= 2
mux_m_offset_b.din1 <= m_offset_add.result
m_offset_b <= mux_m_offset_b.dout
```

4-194. ábra. *Multiplexerek beiktatása a lebegőpontos aritmetikai egység műveletvégző egységébe (3)*

RTL-netlista

```
man_a <= ("0", op_a_buf[man], "00")
man_a <= ("1", op_a_buf[man], "00")
man_a <= man_a >> m_offset_a
man_a <= twc.result(man_a)

man_b <= ("0", op_b_buf[man], "00")
man_b <= ("1", op_b_buf[man], "00")
man_b <= man_b >> m_offset_b
man_b <= twc.result(man_b)
```

RTL-netlista multiplexerekkel

```
man_a_shr.data <= man_a
man_a_shr.steps <= m_offset_a

man_a_twc.op <= man_a

mux_man_a.din0 <= ("0", op_a_buf[man], "00")
mux_man_a.din1 <= ("1", op_a_buf[man], "00")
mux_man_a.din2 <= man_a_shr.result
mux_man_a.din3 <= man_a_twc.result
man_a <= mux_man_a.dout

man_b_shr.data <= man_b
man_b_shr.steps <= m_offset_b

man_b_twc.op <= man_b

mux_man_b.din0 <= ("0", op_b_buf[man], "00")
mux_man_b.din1 <= ("1", op_b_buf[man], "00")
mux_man_b.din2 <= man_b_shr.result
mux_man_b.din3 <= man_b_twc.result
man_b <= mux_man_b.dout
```

4-195. ábra. *Multiplexerek beiktatása a lebegőpontos aritmetikai egység műveletvégző egységébe (4)*

RTL-netlista

```
man_c <= man_a + man_b
man_c <= twc.result(man_c)
man_c <= man_c << nlz
man_c <= (man_a[MSBI:2] * man_b[MSBI:2])[MSBI:mw-1]

sig_c <= man_c[MSBI]
sig_c <= op_a_buf[sig] xor op_b_buf[sig]

nlz <= nlz_logic.result(man_c)
```

RTL-netlista multiplexerekkel

```
man_c_add.op_a <= man_a
man_c_add.op_b <= man_b

man_c_twc.op <= man_c

man_c_shl.data <= man_c
man_c_shl.steps <= nlz

man_c_mul.op_a <= man_a[MSBI:2]
man_c_mul.op_b <= man_b[MSBI:2]

mux_man_c.din0 <= man_c_add.result
mux_man_c.din1 <= man_c_twc.result
mux_man_c.din2 <= man_c_shl.result
mux_man_c.din3 <= man_c_mul.result[MSBI:mw-1]
man_c <= mux_man_c.dout

mux_sig_c.din0 <= man_c[MSBI]
mux_sig_c_din1 <= op_a_buf[sig] xor
                  op_b_buf[sig]
sig_c <= mux_sig_c.dout

nlz_logic.op <= man_c
nlz <= nlz_logic.result
```

4-196. ábra. *Multiplexerek beiktatása a lebegőpontos aritmetikai egység műveletvégző egységébe (5)*

RTL-netlista

```
1  flp_exception_codec.ew <= ew; flp_exception_codec.mw <= mw; flp_exception_codec.operation <= operation
2  flp_exception_codec.op_a <= op_a_buf; flp_exception_codec.op_b <= op_b_buf
3
4  result_sig <= flp_exception_codec.result[sig]; result_sig <= sig_c
5
6  result_exp <= flp_exception_codec.result[exp]; result_exp <= exp_c[MSBI-2:0]
7  result_exp <= "111..."; result_exp <= "000..."
8
9  resman_add.op_a <= man_c[MSBI-1:2]; resman_add.op_b = 1
10
11 result_man <= flp_exception_codec.result[man]; result_man <= resman_add.result
12 result_man <= man_c[MSBI-1:2]; result_man <= "000..."
13
14 result <= (result_sig, result_exp, result_man)
```

RTL-netlista multiplexerekkel

```
1  flp_exception_codec.ew <= ew; flp_exception_codec.mw <= mw; flp_exception_codec.operation <= operation
2  flp_exception_codec.op_a <= op_a_buf; flp_exception_codec.op_b <= op_b_buf
3
4  mux_result_sig.din0 <= flp_exception_codec.result[sig]; mux_result_sig.din1 <= sig_c
5  result_sig <= mux_result_sig.dout
6
7  mux_result_exp.din0 <= flp_exception_codec.result[exp]; mux_result_exp.din1 <= "111..."
8  mux_result_exp.din2 <= "000..."; mux_result_exp.din3 <= exp_c[MSBI-2:0]
9  result_exp <= mux_result_exp.dout
10
11 resman_add.op_a <= man_c[MSBI-1:2]; resman_add.op_b = 1
12
13 mux_result_man.din0 <= flp_exception_codec.result[man]; mux_result_man.din1 <= resman_add.result
14 mux_result_man.din2 <= man_c[MSBI-1:2]; mux_result_man.din3 <= "000..."
15 result_man <= mux_result_man.dout
16
17 result <= (result_sig, result_exp, result_man)
```

4-197. ábra. *Multiplexerek beiktatása a lebegőpontos aritmetikai egység műveletvégző egységébe (6)*

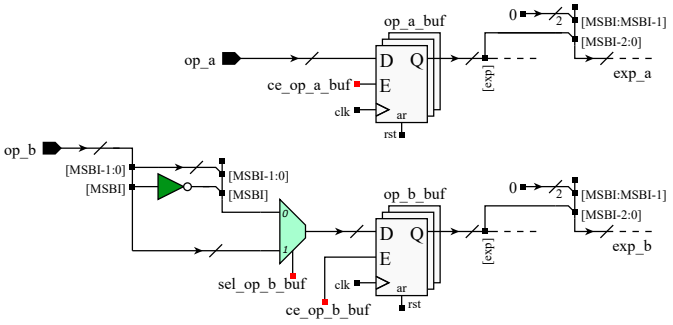
RTL-netlista

```
1  op_a_buf <= op_a
2
3  op_b_buf <= (not op_b[MSBI], op_b[MSBI-1:0])
4  op_b_buf <= op_b
5
6  exp_a <= ("00", op_a_buf[exp])
7
8  exp_b <= ("00", op_b_buf[exp])
```

Verilog – műveletvégző egység

```
1  always @ ( posedge clk, posedge rst ) begin
2  | if ( rst == 1 ) begin op_a_buf <= 0; op_b_buf <= 0;
3  | end else begin
4  | |
5  | | if ( ce_op_a_buf == 1 ) op_a_buf <= op_a;
6  | | if ( ce_op_b_buf == 1 )
7  | | | if ( sel_op_b_buf == 0 ) op_b_buf <=
8  | | |   { ~op_b[$left(op_b)], op_b[$left(op_b)-1:0] };
9  | | | else op_b_buf <= op_b;
10 | | |
11 | | end
12 | |
13 end
14
15 assign exp_a = { 2'b00, op_a_buf[$left(op_a_buf)-1:mw] };
16 assign exp_b = { 2'b00, op_b_buf[$left(op_b_buf)-1:mw] };
```

Áramköri struktúra – műveletvégző egység



4-198. ábra. *A lebegőpontos aritmetikai modul műveletvégző egységének Verilog modellje (1)*

3. lépés – Hardverleíró nyelvű modell létrehozása ◆ A multiplexereket is magában foglaló, immár teljesnek tekinthető RTL-netlista alapján a műveletvégző egység HDL-modelljének elkészítése már gyerekjáték, a hardverleíró nyelvek szintaxisának és szemantikájának ismeretében szinte gépiesen végrehajtható. Az RTL-netlista egyes csoportjainak Verilog-leírását a 4-198–204. ábrák szemléltetik.

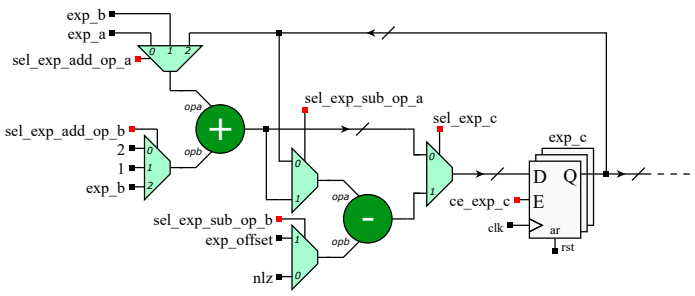
RTL-netlista

```
mux_exp_add_op_a.din0 <= exp_a
mux_exp_add_op_a.din1 <= exp_b
mux_exp_add_op_a.din2 <= exp_c
exp_add.op_a <= mux_exp_add_op_a.dout
mux_exp_add_op_b.din0 <= 2
mux_exp_add_op_b.din1 <= 1
mux_exp_add_op_b.din2 <= exp_b
exp_add.op_b <= mux_exp_add_op_b.dout

mux_exp_sub_op_a.din0 <= exp_c
mux_exp_sub_op_a.din1 <= exp_add.result
exp_sub.op_a <= mux_exp_sub_op_a.dout
mux_exp_sub_op_b.din0 <= nlz
mux_exp_sub_op_b.din1 <= (2**(ew-1)-1)
mux_exp_sub_op_b.din2 <= 1
exp_sub.op_b <= mux_exp_sub_op_b.dout

mux_exp_c.din0 <= exp_add.result
mux_exp_c.din1 <= exp_sub.result
exp_c <= mux_exp_c.dout
```

Verilog – műveletvégző egység

```
always @ (*) begin
  case ( sel_exp_add_op_a )
    2'b00: mux_exp_add_op_a <= exp_a;
    2'b01: mux_exp_add_op_a <= exp_b;
    2'b10: mux_exp_add_op_a <= exp_c;
    default: ;
  endcase
end

always @ (*) begin
  case ( sel_exp_add_op_b )
    2'b00: mux_exp_add_op_b <= 2;
    2'b01: mux_exp_add_op_b <= 1;
    2'b10: mux_exp_add_op_b <= exp_b;
    default: ;
  endcase
end

always @ (*) begin
  case ( sel_exp_sub_op_a )
    1'b0: mux_exp_sub_op_a <= exp_c;
    1'b1: mux_exp_sub_op_a <= exp_add_result;
    default: ;
  endcase
end

always @ (*) begin
  case ( sel_exp_sub_op_b )
    1'b0: mux_exp_sub_op_b <= nlz;
    1'b1: mux_exp_sub_op_b <= (2**(ew-1))-1;
    default: ;
  endcase
end

assign exp_add_result = mux_exp_add_op_a +
                        mux_exp_add_op_b;
assign exp_sub_result = mux_exp_sub_op_a -
                        mux_exp_sub_op_b;

always @ ( posedge clk, posedge rst ) begin
  if ( rst == 1 ) exp_c <= 0;
  else
    if ( ce_exp_c == 1 )
      if ( sel_exp_c == 0 ) exp_c <= exp_add_result;
      else exp_c <= exp_sub_result;
end
```

Áramköri struktúra – műveletvégző egység



4-199. ábra. *A lebegőpontos aritmetikai modul műveletvégző egységének Verilog modellje (2)*

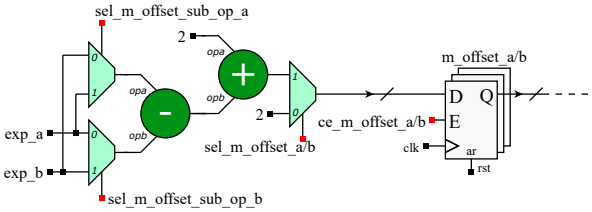
RTL-netlista

```
1  mux_m_offset_sub_op_a.din0 <= exp_b
2  mux_m_offset_sub_op_a.din1 <= exp_a
3  m_offset_sub.op_a <= mux_m_offset_sub_op_a.dout
4  mux_m_offset_sub_op_b.din0 <= exp_a
5  mux_m_offset_sub_op_b.din1 <= exp_b
6  m_offset_sub.op_b <= mux_m_offset_sub_op_b.dout
7
8  m_offset_add.op_a <= m_offset_sub.result
9  m_offset_add.op_b <= 2
10
11 mux_m_offset_a.din0 <= 2
12 mux_m_offset_a.din1 <= m_offset_add.result
13 m_offset_a <= mux_m_offset_a.dout
14
15 mux_m_offset_b.din0 <= 2
16 mux_m_offset_b.din1 <= m_offset_add.result
17 m_offset_b <= mux_m_offset_b.dout
```

Verilog – műveletvégző egység

```
1  always @ (*) begin
2    case ( sel_m_offset_sub_op_a )
3      1'b0: mux_m_offset_sub_op_a <= exp_b;
4      1'b1: mux_m_offset_sub_op_a <= exp_a;
5      default: ;
6    endcase
7  end
8
9  always @ (*) begin
10   case ( sel_m_offset_sub_op_b )
11     1'b0: mux_m_offset_sub_op_b <= exp_a;
12     1'b1: mux_m_offset_sub_op_b <= exp_b;
13     default: ;
14   endcase
15 end
16
17 assign m_offset_sub_result = mux_m_offset_sub_op_a -
18                               mux_m_offset_sub_op_b;
19
20 assign m_offset_add_result = 2+m_offset_sub_result;
21
22 always @ ( posedge clk, posedge rst ) begin
23   if ( rst == 1 ) begin
24     m_offset_a <= 0; m_offset_b <= 0;
25   end else begin
26     if ( ce_m_offset_a == 1 )
27       if ( sel_m_offset_a == 0 ) m_offset_a <= 2;
28       else m_offset_a <= m_offset_add_result;
29
30     if ( ce_m_offset_b == 1 )
31       if ( sel_m_offset_b == 0 ) m_offset_b <= 2;
32       else m_offset_b <= m_offset_add_result;
33
34   end
35 end
```

Áramköri struktúra – műveletvégző egység



4-200. ábra. *A lebegőpontos aritmetikai modul műveletvégző egységének Verilog modellje (3)*

A viselkedési modell tárgyalásánál és az RTL-netlista részleteinek kidolgozása során a balértékekre mint „változókra” hivatkoztunk, a „regiszter” kifejezést szándékosan kerülve. Ennek oka, hogy sem a pszeudokóddal megfogalmazott viselkedési modell, sem az RTL-netlista nem rögzíti azt, hogy az egyes változók tárolására valóban tárolóelemeket, vagyis regisztereket hozunk létre, vagy ezeknek a változóknak az értékét egyszerű vezetékek hordozzák majd. Ezt a döntést csak a HDL-modell létrehozásakor kell meghoznunk. Jó példa erre a 4-198. ábrán látható Verilog modell, amelyben az *op_a_buf* és *op_b_buf* változókat ténylegesen regiszterekként valósítottuk meg, ugyanakkor az *exp_a* és *exp_b* változók egyszerű konkurens értékadásokkal leírt buszok, tehát önmagukban nem tároló elemek (4-198. ábra jobb oldal: 15–16. sor). Az áramköri séma alapján belátható, hogy miért; az *op_a/b_buf* regiszterekből az *exp_a/b* értékek egyszerű hozzáfűzéssel és bitkiválasztással képződnek, tehát nincs semmilyen kombinációs hálózat, amelynek a késleltetése miatt érdemes lenne újabb

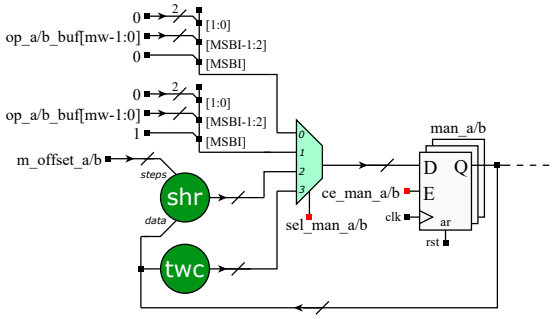
RTL-netlista

```
1  man_a_shr.data <= man_a
2  man_a_shr.steps <= m_offset_a
3
4  man_a_twc.op <= man_a
5
6  mux_man_a.din0 <= ("0",op_a_buf[man],"00")
7  mux_man_a.din1 <= ("1",op_a_buf[man],"00")
8  mux_man_a.din2 <= man_a_shr.result
9  mux_man_a.din3 <= man_a_twc.result
10 man_a <= mux_man_a.dout
11
12 man_b_shr.data <= man_b
13 man_b_shr.steps <= m_offset_b
14
15 man_b_twc.op <= man_b
16
17 mux_man_b.din0 <= ("0",op_b_buf[man],"00")
18 mux_man_b.din1 <= ("1",op_b_buf[man],"00")
19 mux_man_b.din2 <= man_b_shr.result
20 mux_man_b.din3 <= man_b_twc.result
21 man_b <= mux_man_b.dout
```

Verilog – műveletvégző egység

```
1  always @ ( posedge clk, posedge rst ) begin
2    if ( rst == 1 ) begin man_a <= 0; man_b <= 0;
3    end else begin
4      if ( ce_man_a == 1 )
5        case ( sel_man_a )
6          0: man_a <= { 1'b0, op_a_buf[mw-1:0], 2'b00 };
7          1: man_a <= { 1'b1, op_a_buf[mw-1:0], 2'b00 };
8          2: man_a <= man_a >> m_offset_a;
9          3: man_a <= (~man_a) + 1;
10         default: ;
11       endcase
12
13     if ( ce_man_b == 1 )
14       case ( sel_man_b )
15         0: man_b <= { 1'b0, op_b_buf[mw-1:0], 2'b00 };
16         1: man_b <= { 1'b1, op_b_buf[mw-1:0], 2'b00 };
17         2: man_b <= man_b >> m_offset_b;
18         3: man_b <= (~man_b) + 1;
19         default: ;
20       endcase
21   end
22 end
```

Áramköri struktúra – műveletvégző egység



4-201. ábra. *A lebegőpontos aritmetikai modul műveletvégző egységének Verilog modellje (4)*

regiszterréteget bevezetnünk. A vezérlőegység tervezésekor úgy kell számolnunk, hogy az *op_a_buf* és az *op_b_buf* regiszterekbe való betöltéssel egyidőben az *exp_a* és *exp_b* értékek is előállnak.

A 4-199. ábra az *exp_c* változó adatkapcsolatait, valamint az azokból készült Verilog modellt és RTL kapcsolási sémát mutatja be. Látható, hogy az *exp_c* változót regiszterként valósítottuk meg, és ezen a reprezentáción már egyértelműen megfigyelhetők a korábban említett multiplexerek is, amelyeket az *exp_c* változón végzett sokféle művelet tett szükségessé.

A közös karakterisztikára hozáshoz szükséges eltolások mértékét tároló *m_offset_a* és *m_offset_b* változók esetén a helyzet hasonló; ezek kiszámításához több aritmetikai műveletet is el kell végeznünk, ráadásul az összeadás/szorzás műveletek operandusaitól függő módon, így a műveletvégző egység vonatkozó részlete (lásd 4-200. ábra) ezúttal is regiszterek bemeneteit meghajtó aritmetikai elemek és multiplexerek megfelelő kombinációjaként írható le.

A HDL-leírásokban érdemes megfigyelni, hogy a multiplexálás funkciót többféleképpen fejezhetjük ki; leírhatjuk egy-egy különálló utasítással (4-200. ábra: 1–7. sor), de be is ágyazhatjuk a meghajtott regisztert leíró szinkron *always*-blokkba (4-200. ábra: 27–28. és 31–32. sor). Áramköri szempontból a két megoldás között nincs különbség, de a kód további optimalizálása során fontos lehet az, hogy az utóbbi esetben – mivel a szinkron *always*-blokk nemcsak magát

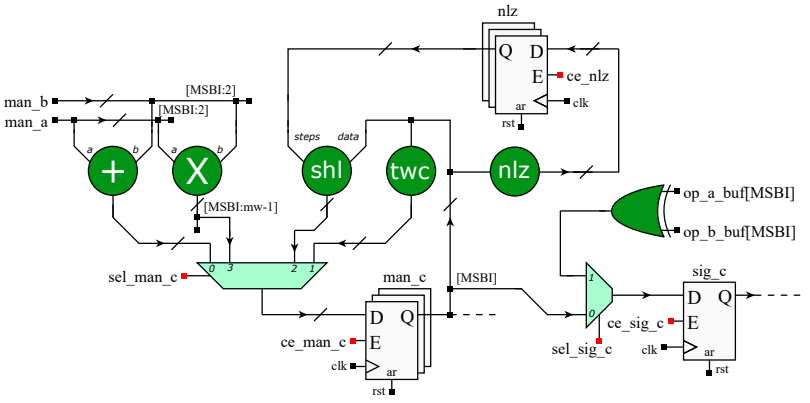
RTL-netlista

```
man_c_add.op_a <= man_a
man_c_add.op_b <= man_b

man_c_twc.op <= man_c

man_c_shl.data <= man_c
man_c_shl.steps <= nlz

man_c_mul.op_a <= man_a[MSBI:2]
man_c_mul.op_b <= man_b[MSBI:2]

mux_man_c.din0 <= man_c_add.result
mux_man_c.din1 <= man_c_twc.result
mux_man_c.din2 <= man_c_shl.result
mux_man_c.din3 <= man_c_mul.result[MSBI:mw-1]
man_c <= mux_man_c.dout

mux_sig_c.din0 <= man_c[MSBI]
mux_sig_c_din1 <= op_a_buf[sig] xor
                  op_b_buf[sig]
sig_c <= mux_sig_c.dout

nlz_logic.op <= man_c
nlz <= nlz_logic.result
```

Verilog – műveletvégző egység

```
assign man_c_mul_result = man_a[$left(man_a):2] *
                          man_b[$left(man_b):2];

always @ ( posedge clk, posedge rst ) begin
  if ( rst == 1 ) begin
    sig_c <= 0; man_c <= 0; nlz <= 0;
  end else begin
    if ( ce_sig_c == 1 )
      if ( sel_sig_c == 0 ) sig_c <=
                              man_c[$left(man_c)];
      else sig_c <= op_a_buf[$left(op_a_buf)] ^
                    op_b_buf[$left(op_b_buf)];

    if ( ce_man_c == 1 )
      case ( sel_man_c )
        0: man_c <= man_a + man_b;
        1: man_c <= (~man_c) + 1;
        2: man_c <= man_c << nlz;
        3: man_c <= man_c_mul_result[
                      $left(man_c_mul_result):mw-1
                    ];
        default: ;
      endcase

    if ( ce_nlz == 1 ) begin
      tmp = { man_c, 1'b1 }; first_1_pos = 0;
      for ( int i = 0; i <= (mw+3); i = i+1 ) begin
        if ( tmp[i] == 1'b1 ) first_1_pos = i;
      end
      nlz <= (mw+3) - first_1_pos;
    end
  end
end
```

Áramköri struktúra – műveletvégző egység



4-202. ábra. *A lebegőpontos aritmetikai modul műveletvégző egységének Verilog modellje (5)*

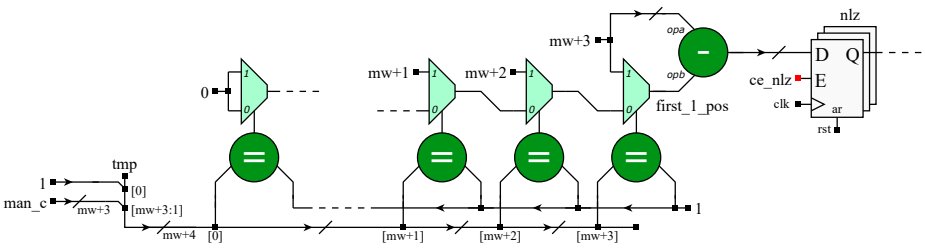
a regisztert, hanem a regiszter és az azt meghajtó kombinációs hálózat (ez esetben egy multiplexer) együttesét írja le[166] – „nem nevezzük nevén” a multiplexer kimenetét, így a kódban másutt hivatkozni sem tudunk rá. Ezzel ellentétben, ha a multiplexert egy saját kombinációs *always*-blokkal modellezzük, akkor annak kimenetére bárhol máshol is utalhatunk (pl. 4-200. ábra: 17–18. sor: a *mux_m_offset_sub_op_a/b* kifejezések a kivonó egység bemeneteit meghajtó multiplexerek

[166] Lásd még DRMSz: 4.5.1. alfejezet.

kimenetét jelentik). Ha az RTL-modellben olyan módosítást kellene végrehajtanunk, amelyben a regiszterek mindenkori bemenetére, vagyis a *sel_m_offset_a/b* kiválasztójelekkel vezérelt multiplexer kimenetére kellene hivatkoznunk, kénytelenek lennénk a modellt teljesen átfogalmazni. Az összevont megfogalmazás tehát tömör és könnyebben olvasható, a szétbontott leírás viszont ebből a szempontból rugalmasabbnak mondható.

A 4-201. ábrán, a *man_a/b* változókra vonatkozó HDL-modellrészletben ugyancsak a multiplexálás és a szinkron tároló funkció összevont leírására láthatunk példát. Ezúttal, mivel a *man_a/b* regiszterbe négy különböző értéket kell betöltenünk az algoritmusok végrehajtása során, négybemenetű multiplexert kell használnunk, amit hardverleíró nyelven a *case* szerkezettel tudunk hatékonyan kifejezni. Ebben a leírásban azt is megfigyelhetjük, hogy a szinkron *always*-blokk „hatásköre" a regiszterek bemenetét közvetlenül meghajtó multiplexeren is túlmutathat; a 4-201. ábra Verilog-leírásának 17-18. soraiban (tehát a szinkron *always*-blokk belsejében) fejezzük ki a jobbra léptetés és a ketteskomplemens-képzés műveletét is, vagyis a multiplexer 2. és 3. bemeneteit meghajtó, ugyancsak kombinációs hálózatként megvalósított aritmetikai műveleteket. A kompakt, jól áttekinthető megoldás hátránya ebben az esetben is érvényesül; ezeknek az operátoroknak a kimenetei nincsenek expliciten megnevezve a kódban, így máshonnan hivatkozni nem tudunk rájuk.

A 4-202. ábra a *man_c*, a *sig_c*, valamint az *nlz* változókon végzett műveleteket lehetővé tévő áramkörök HDL-modelljét és szerkezetét mutatja be. A Verilog kód felépítése hasonló a korábban látottakéhoz, az *nlz* regiszterbe betöltendő érték előállítását kifejező 26–30. sorok azonban külön figyelmet érdemelnek. Az *nlz* regiszterbe a normalizálás során a *man_c*, tehát egy *mw* méretű, tetszőleges tartalmú regiszter legmagasabb helyiértékein lévő nullák számát kell betöltenünk. Ebben a modellben ezt a feladatot egy *for* ciklussal megvalósított kombinációs hálózat látja el. Ehhez létrehozunk egy ideiglenes változót *(tmp)*, amelybe a 26. sorban blokkoló értékadással betöltjük a *man_c* regiszter értékét, kiegészítve egy egyessel a legalsó helyiértéken. Ugyancsak a 26. sorban, szintén blokkoló értékadással nullára inicializáljuk a *first_1_pos* változót, ami a legmagasabb helyiértéken fellelt egyes bit vektorbeli indexét tárolja majd. Ezután egy *for* ciklus segítségével a legalacsonyabb helyiértéktől kezdve végiglépkedünk a *tmp* változó bitjein, és ha az i-edik pozíción egyest találunk, akkor a *first_1_pos* változóba betöltjük a vizsgált bit vektorbeli indexét. A ciklus végén a legmagasabb helyiértékeken lévő nullák számát a 30. sorban lévő, ezúttal már nem-blokkoló értékadással töltjük be az *nlz* regiszterbe. A 26. és a 29. sor között a *tmp* és a *first_1_pos* változók kizárólag blokkoló értékadással kapnak értéket, ami azt jelenti, hogy a 26. sor értékadásainak és a ciklustörzs összes utasításának kumulált eredménye egyazon szimulációs ciklus végén íródik be a *first_1_pos* változóba. Az *always*-blokk sorrendi jellege miatt az egyazon szimulációs cikluson belül történő értékadások – ezúttal az egymást követő iterációkban, egyazon változóra vonatkozó



4-203. ábra. *A man_c regiszter legmagasabb helyiértékein lévő nullák számát megállapító kombinációs hálózat*

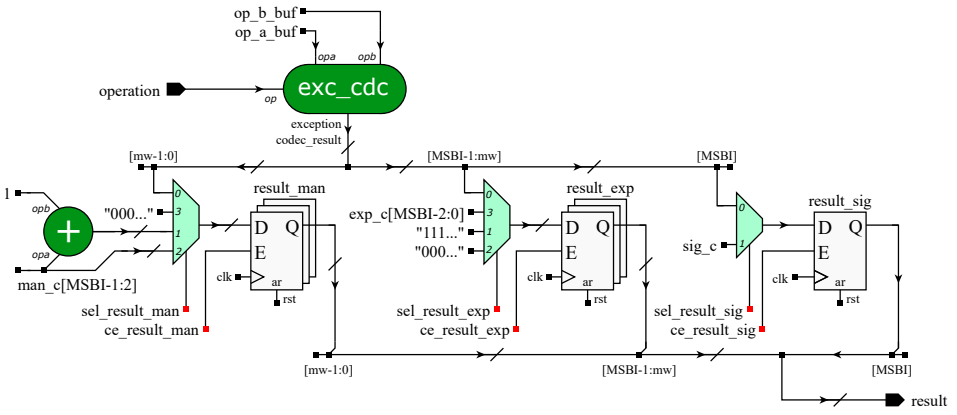
RTL-netlista

```
flp_exception_codec.ew <= ew; flp_exception_codec.mw <= mw; flp_exception_codec.operation <= operation
flp_exception_codec.op_a <= op_a_buf; flp_exception_codec.op_b <= op_b_buf

mux_result_sig.din0 <= flp_exception_codec.result[sig]; mux_result_sig.din1 <= sig_c
result_sig <= mux_result_sig.dout

mux_result_exp.din0 <= flp_exception_codec.result[exp]; mux_result_exp.din1 <= "111..."
mux_result_exp.din2 <= "000..."; mux_result_exp.din3 <= exp_c[MSBI-2:0]
result_exp <= mux_result_exp.dout

resman_add.op_a <= man_c[MSBI-1:2]; resman_add.op_b = 1

mux_result_man.din0 <= flp_exception_codec.result[man]; mux_result_man.din1 <= resman_add.result
mux_result_man.din2 <= man_c[MSBI-1:2]; mux_result_man.din3 <= "000..."
result_man <= mux_result_man.dout

result <= (result_sig, result_exp, result_man)
```

Verilog – műveletvégző egység

```
flp_exception_codec #(ew, mw) exc_cdc (operation, op_a_buf, op_b_buf, exception_codec_result);

always @ ( posedge clk, posedge rst ) begin
  if ( rst == 1 ) begin result_sig <= 0; result_exp <= 0; result_man <= 0;
  end else begin
      if ( ce_result_sig == 1 )
        if ( sel_result_sig == 0 ) result_sig <= exception_codec_result[ew+mw]; else result_sig <= sig_c;

      if ( ce_result_exp == 1 ) case ( sel_result_exp )
                                  0: result_exp <= exception_codec_result[ew+mw-1:mw];
                                  1: result_exp <= { ew{1'b1} };
                                  2: result_exp <= 0;
                                  3: result_exp <= exp_c[$left(exp_c)-2:0];
                                  default: ;
                                endcase

      if ( ce_result_man == 1 ) case ( sel_result_man )
                                  0: result_man <= exception_codec_result[mw-1:0];
                                  1: result_man <= man_c[$left(man_c)-1:2] + 1;
                                  2: result_man <= man_c[$left(man_c)-1:2];
                                  3: result_man <= 0;
                                  default: ;
                                endcase
  end
end

assign result = { result_sig, result_exp, result_man };
```

Áramköri struktúra – műveletvégző egység



4-204. ábra. *A lebegőpontos aritmetikai modul műveletvégző egységének Verilog modellje (6)*

értékadások – közötti hierarchia is érvényesül; a később „végrehajtott” kiértékelés eredménye felülírja a korábbiakat. Áramköri szempontból az ilyen jellegű, ciklussal kifejezett leírások szintetizálhatóak; az eredményül kapott áramköri szerkezet egy kaszkád-multiplexer struktúra, aminek a mérete a ciklusszám függvénye (lásd 4-203. ábra, vö. DRMSz: 81–84. oldal).

A kaszkád-multiplexer struktúra egy sokszintű kombinációs hálózat, amelynek terjedési késleltetése a sok sorba kapcsolt áramköri elem miatt nagy lehet, könnyen alakulhat ki itt a kritikus út. Ezért az ilyen megoldásokat nagy körültekintéssel kell elkészíteni, és a szintézis eredményét mindig érdemes ellenőrizni![167]

A 4-204. ábra az összeadás/szorzás műveletek végeredményének komponenseit tároló *result_sig*, *result_exp* és *result_man* regisztereknek és az azokat meghajtó kombinációs logikáknak a HDL-modelljét és RTL kapcsolási sémáját mutatja. A viselkedési modell tárgyalásánál már említettük, hogy a *+/-0*, a *+/-Inf* és a *NaN* bemeneteket mint egyedi eljárást igénylő, speciális eseteket egy különálló kódoló-dekódoló kombinációs hálózattal kezeljük. A 4-204. ábra Verilog-leírásában ezt az áramkört *exc_cdc (exception codec)* példánynévvel tüntettük fel. A belső működését tekintve egyszerű, ugyanakkor rengeteg *if-else* kifejezésből álló, ezáltal terjedelmes HDL-leírás az 5.3.2.1. Függelékben megtalálható. Ezt a kombinációs hálózatot a fixpontos aritmetikai egységnél látottakhoz hasonlóan VHDL nyelven, egy sorrendi utasításblokkban, de ezúttal eljárás *(procedure)* helyett függvényként *(function)* valósítottuk meg, amelyhez egy példányosítható burkolómodult is létrehoztunk. Ennek a burkolómodulnak a példányosítása látható a 4-204. ábra Verilog-leírásának 1. sorában.[168]

4.5.3.3. Vezérlőegység

A lebegőpontos aritmetikai modul műveletvégző egysége ezzel elkészült. A tervezés következő lépése az ehhez a műveletvégző egységhez tartozó vezérlőszerkezet elkészítése. Erre a célra – a *CPU*-nál látottakhoz hasonlóan – a megvalósított algoritmusok összetettsége miatt egy állapotgépet érdemes tervezni. Az állapotgépet ezúttal egy, az állapotregiszter és a kimeneti/következőállapot-logika működését egy-egy külön *always*-blokkban megfogalmazó Verilog modellel írjuk le, amelynek vázlatát a 4-205. ábra szemlélteti.

A szinkron kézfogásos interfész 1-4. ábrán látható specifikációja szerinti működést biztosító éldetektort (4-205. ábra: 1–5. sor) – a két *always*-blokkot alkalmazó állapotgép-leírási módszernek megfelelően – az állapotregiszter működését megfogalmazó szinkron *always*-blokk (4-205. ábra: 7–10. sor), majd a következőállapot-logika és a kimeneti logika együttesét leíró kombinációs *always*-blokk követi (4-205. ábra: 12–29. sor). Ez utóbbiban fontos szerepe van a hálózat kimeneteit az aktuális állapot alapján meghatározó *case*-szerkezetet megelőző alapértelmezett értékadásoknak (4-205. ábra: 15–17. sor). Mivel kombinációs *always*-blokkról van szó, ezért a tároló funkció akaratlan megfogalmazásának – és általa latch szintézisének[169] – elkerülése céljából elvileg a *case*-szerkezet minden ágában nyilatkoznunk kellene minden vezérlőjelről, amely a kódot átláthatatlanná tenné.

---

[167] Az *nlz* funkció megvalósítására az itt bemutatott módszernél időzítési szempontból hatékonyabb, fa-struktúrájú megoldások is léteznek.

[168] A függvény és az ahhoz tartozó burkolómodul VHDL nyelven van megfogalmazva. A VHDL-modul Verilog modellbeli példányosításának (vagy fordítva) elvi akadálya ugyan nincs, de bizonyos szimulációs és szintézis-szoftverek a hozzájuk vásárolt licensz típusától tehetik függővé, hogy ezt megtehetjük-e vagy sem.

[169] Lásd még: DRMSz 4-43. ábra, 4-45. ábra.

Verilog – műveletvégző egység

```
1  always @ ( posedge clk, posedge rst ) begin        // REQUEST RISING EDGE DETECTOR
2    if ( rst == 1'b1 ) request_df0 <= 1'b0;
3    else request_df0 <= request;
4  end
5  assign request_rising = request & ~request_df0;
6
7  always @ ( posedge clk, posedge rst ) begin        // CONTROL FSM: state register
8    if ( rst == 1'b1 ) present_state <= idle;
9    else present_state <= next_state;
10 end
11
12 always @ (*) begin                                 // CONTROL FSM: next-state logic & ouput logic
13
14   // defaults...
15   next_state <= present_state; ready <= 0;
16   ce_op_a_buf <= 0; ce_op_b_buf <= 0;       [...] // reset registers' clock enable signals...
17   sel_op_b_buf <= 0; sel_exp_add_op_a <= 0; [...] // reset multiplexers' select signals...
18
19   case ( present_state )
20
21     idle:        if ( request_rising == 1 ) begin [...]
22
23     [...]
24
25     default:     ;
26
27   endcase
28
29 end
```

4-205. ábra. *A lebegőpontos aritmetikai egység vezérlő állapotgépét leíró Verilog modul vázlata*

Ehelyett kihasználjuk az *always*-blokknak mint sorrendi utasításblokknak azt a tulajdonságát, hogy a törzsében foglalt utasítások a nyelv szimulációs szemantikája alapján egymás után értékelődnek ki. Ez egyúttal azt is jelenti, hogy ha egyazon jelnek több helyen is értéket adunk, akkor a később kiértékelt értékadás felülírja a korábbi kiértékelések eredményét. A 15–17. sorokban látható alapértelmezett értékadások tehát gondoskodnak arról, hogy minden vezérlőjel kapjon valamilyen értéket az *always*-blokk minden kiértékelése során, függetlenül attól, hogy az egyes vezérlési állapotokhoz tartozó *case*-ágakban melyik jeleknek adtunk értéket, így a modell jól áttekinthető marad.[170]

Az első vezérlési állapotban (lásd 4-206. ábra) az *op_a_buf* és *op_b_buf* operandusregiszterek betöltésére kerül sor. Mivel a két operandus egymástól függetlenül kezelhető, ezért a viselkedési modell utasításainak végrehajtása egyetlen órajelciklusba sűrítve történik meg. A 4-207. ábrán egy lebegőpontos kivonás[171] elvégzésének első lépését követhetjük nyomon, amely során az *op_a_buf* és *op_b_buf* regiszterekbe betöltődnek az operandusok.

Figyeljük meg, hogy az *exp_a* és *exp_b* változók értéke a műveletvégző egységben – mivel ezek reprezentálására nem hoztunk létre külön regisztereket – az *op_a_buf* és az *op_b_buf* regiszterek kimenetének megváltozásával egyidőben előáll.

[170] A * szimbólum az érzékenységi listában azt fejezi ki, hogy az *always*-blokk az annak törzsében található minden jobbérték-kifejezés kiértékelése után maga is kiértékelődik, modellezve azt, hogy a leírni szándékozott áramköri struktúra (jelesül egy kombinációs hálózat) minden bemenetére azonnal reagál. Fogalmazhatunk tehát úgy is, hogy az érzékenységi lista az alapértelmezett értékadásokkal együtt használva biztosítja, hogy a megfogalmazott viselkedés valóban egy kombinációs hálózat viselkedésével azonos. Fontos, hogy ez a módszer csak akkor működik, ha az alapértelmezett értékadások között a kombinációs hálózat összes kimenetét felsoroljuk, beleértve a vezérlő állapotgép kimeneteit és a következő állapotot meghatározó változót.

[171] A szemléltető hullámformákon 4-bites karakterisztikájú és 8-bites mantisszájú lebegőpontos formátumban ábrázolt számokat használunk.

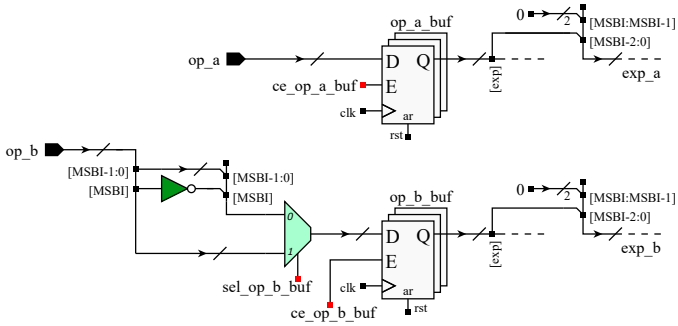
Pseudo – viselkedési modell

```
1  START: loop {
2    if ( request ) {
3      ready <= 0
4      op_a_buf <= op_a
5      if ( operation == sub )
6        op_b_buf <= -1 * op_b
7      else op_b_buf <= op_b
8      break
9    }
10 }
11
12 EXCEPTIONS: ...
```

Verilog – vezérlőegység

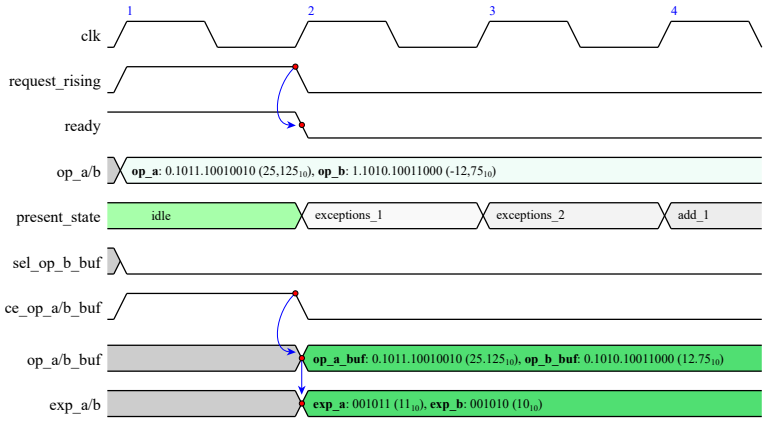
```
1  idle: begin
2    ready <= 1;
3    if ( request_rising == 1 ) begin
4      ce_op_a_buf <= 1;
5      if ( operation == 1 ) sel_op_b_buf <= 0;
6      else sel_op_b_buf <= 1;
7      ce_op_b_buf <= 1;
8      next_state <= exceptions_1;
9    end
10 end
11
12 exceptions_1: [...]
```

Áramköri struktúra – műveletvégző egység



4-206. ábra. *A lebegőpontos aritmetikai egység vezérlő állapotgépének részlete (1)*
Az operation bemenet kódolása: összeadás: 0, kivonás: 1, szorzás: 2

Az operandusregiszterek betöltése után azok vizsgálatára kerül sor az *exceptions_2* állapotban (lásd 4-208. ábra). Az *exceptions_1* állapotra csupán az állapotgép működésének egy órajelciklussal való késleltetése miatt van szükség, ui. az operandusregiszterek az órajelnek a 4-207. ábra számozása szerinti második felfutóéle hatására kapnak új értéket, vagyis csak az órajel harmadik felfutóélénél, tehát az *exceptions_2* állapotban mintavételezhetők. Ekkor kerülhet sor a 4-208. ábra Verilog-leírásának 4–7. soraiban lévő feltételes kifejezések kiértékelésére.



4-207. ábra. *Az operandusok betöltése egy kivonás művelet esetén*

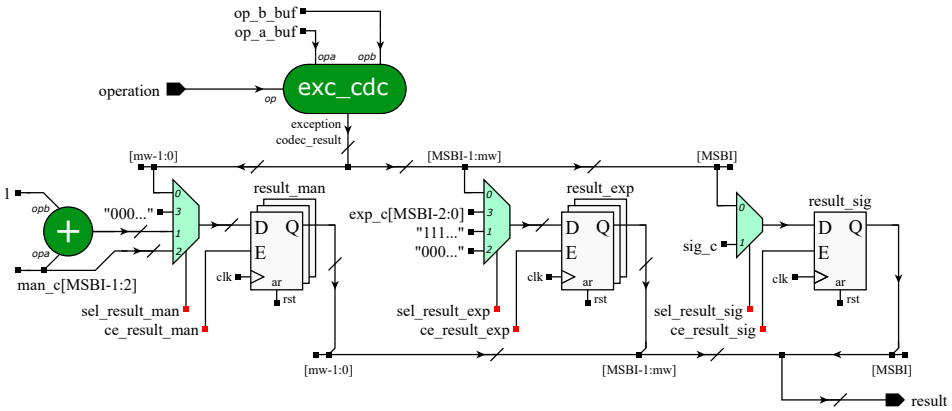
Pseudo – viselkedési modell

```
1  EXCEPTIONS:
2    if ( op_a_buf == "+/-0" or op_b_buf == "+/-0" or op_a_buf == "+/-Inf" or op_b_buf == "+/-Inf" or
3         op_a_buf == "NaN" or op_b_buf == "NaN" ) {
4    | result_sig <= flp_exception_codec.result[sig](ew, mw, operation, op_a_buf, op_b_buf)
5    | result_exp <= flp_exception_codec.result[exp](ew, mw, operation, op_a_buf, op_b_buf)
6    | result_man <= flp_exception_codec.result[man](ew, mw, operation, op_a_buf, op_b_buf)
7    | goto STOP
8    } elsif ( operation == mul ) goto MUL else goto ADD
```

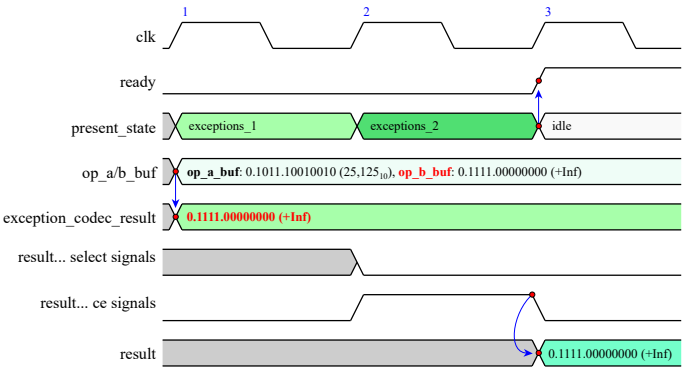
Verilog – vezérlőegység

```
1  exceptions_1: next_state <= exceptions_2;
2  exceptions_2:
3    if (
4    | (op_a_buf[$left(op_a_buf)-1:mw] == 0 && op_a_buf[mw-1:0] == 0) ||          // op_a = +/- 0
5    | (op_b_buf[$left(op_b_buf)-1:mw] == 0 && op_b_buf[mw-1:0] == 0) ||          // op_b = +/- 0
6    | (op_a_buf[$left(op_a_buf)-1:mw] == (2**ew-1) && op_a_buf[mw-1:0] == 0) || // op_a = +/- Inf or NaN
7    | (op_b_buf[$left(op_b_buf)-1:mw] == (2**ew-1) && op_b_buf[mw-1:0] == 0)    // op_b = +/- Inf or NaN
8    ) begin
9    | sel_result_sig <= 0; sel_result_exp <= 0; sel_result_man <= 0;
10   | ce_result_sig <= 1; ce_result_exp <= 1; ce_result_man <= 1;
11   | next_state <= idle;
12   end else if ( operation == 2 ) next_state <= mul_1; else next_state <= add_1;
```

Áramköri struktúra – műveletvégző egység



4-208. ábra. *A lebegőpontos aritmetikai egység vezérlő állapotgépének részlete (2)*



4-209. ábra. *Az exceptions_1/2 vezérlési állapotok működése*

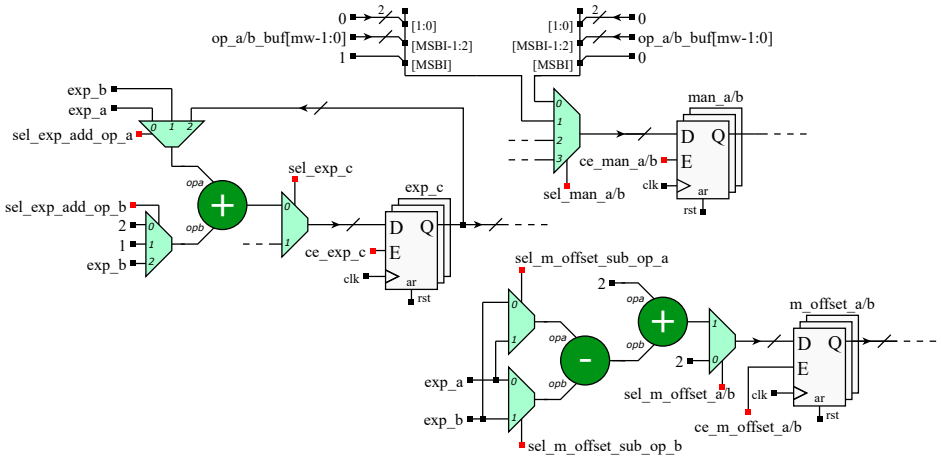
Pseudo – viselkedési modell

```
1  ADD: exp_a <= ("00", op_a_buf[exp]) ; A nemnegatív karakterisztikákat
2       exp_b <= ("00", op_b_buf[exp]) ; 2 extra bittel kiegészítve tároljuk.
3
4       if ( exp_a >= exp_b ) {
5       | exp_c <= exp_a + 2                  ; A közös karakterisztika a nagyobbiknál 2-vel nagyobb.
6       | m_offset_a <= 2                     ; A mantisszákat legalább 2-vel toljuk jobbra.
7       | m_offset_b <= (exp_a - exp_b) + 2   ; Az eltolt mantisszák előjeles összegének előjele
8       } else {                              ; egyúttal a végeredmény előjele is.
9       | exp_c <= exp_b + 2
10      | m_offset_a <= (exp_b - exp_a) + 2
11      | m_offset_b <= 2
12      }
13
14      if ( exp_a == 0 ) man_a <= ("0", op_a_buf[man], "00")  ; A mantisszák elé beszúrjuk a rejtett bitet.
15      else man_a <= ("1", op_a_buf[man], "00")               ; A törtre normalizált (szubnormális)
16                                                             ; számok esetén a rejtett bit értéke nulla.
17      if ( exp_b == 0 ) man_b <= ("0", op_b_buf[man], "00")  ; A mantisszákat az alsó helyiértékeken 2
18      else man_b <= ("1", op_b_buf[man], "00")               ; extra bittel (guard-bit) egészítjük ki
19      [...]                                                  ; a pontosság megőrzése érdekében.
```

Verilog – vezérlőegység

```
1  add_1: begin
2  |
3  | if ( exp_a >= exp_b ) begin
4  | | sel_exp_c <= 0; sel_exp_add_op_a <= 0; sel_exp_add_op_b <= 0; ce_exp_c <= 1;
5  | | sel_m_offset_a <= 0; ce_m_offset_a <= 1;
6  | | sel_m_offset_b <= 1; sel_m_offset_sub_op_a <= 1; sel_m_offset_sub_op_b <= 1; ce_m_offset_b <= 1;
7  | end else begin
8  | | sel_exp_c <= 0; sel_exp_add_op_a <= 1; sel_exp_add_op_b <= 0; ce_exp_c <= 1;
9  | | sel_m_offset_a <= 1; sel_m_offset_sub_op_a <= 0; sel_m_offset_sub_op_b <= 0; ce_m_offset_a <= 1;
10 | | sel_m_offset_b <= 0; ce_m_offset_b <= 1;
11 | end
12 |
13 | if ( exp_a == 0 ) sel_man_a <= 0; else sel_man_a <= 1; ce_man_a <= 1;
14 | if ( exp_b == 0 ) sel_man_b <= 0; else sel_man_b <= 1; ce_man_b <= 1;
15 |
16 | next_state <= add_2;
17 |
18 end
```

Áramköri struktúra – műveletvégző egység



4-210. ábra. *A lebegőpontos aritmetikai egység vezérlő állapotgépének részlete (3)*

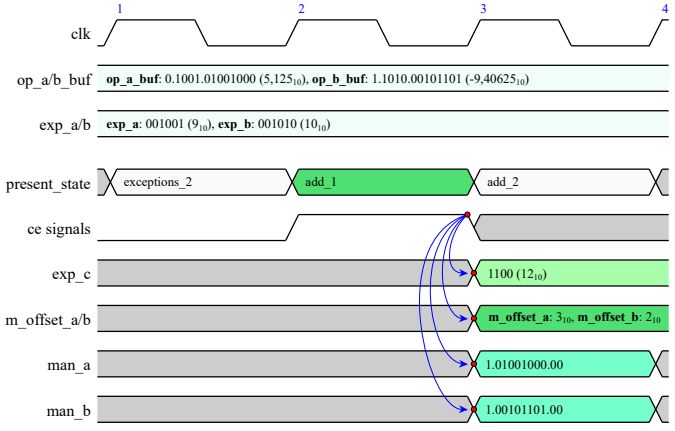
A 4-209. ábra hullámformáján az *exceptions_1/2* állapotok működését figyelhetjük meg egy olyan esetben, amikor az *op_b_buf* operandusregiszterbe a *+Inf* érték kerül. Az órajel első felfutóélénél

betöltődnek az operandusokat tároló *op_a/b_buf* regiszterek. Az *exc_cdc* dekóder kimenetén azonnal megjelenik az ennek megfelelő eredményvektor *(+Inf)*. A vezérlő állapotgép az operandusregiszterek értéke alapján dönti el (lásd 4-208. ábra Verilog: 4–7. sor), hogy valamilyen speciális operandusról van-e szó, tehát fel kell-e használnia az *exc_cdc* dekóder kimenetét vagy sem. Az *op_b_buf* tartalma miatt a 7. sor feltételvizsgálata igazra értékelődik ki, vagyis az állapotgép észlelte a speciális operandust, aminek következtében a *result_sig*, *result_exp* és *result_man* regiszterekhez tartozó vezérlőjeleket úgy állítja be, hogy azokba az *exc_cdc* dekóder által már előkészített eredmény töltődjön be. Mivel a lebegőpontos aritmetikai egység *result* kimenete e regiszterekből egyszerűen azok összefűzésével képződik, ezért a regiszterek kimenetének átíródásakor egyúttal maga a *result* kimenet is frissül. Mivel ugyanebben a pillanatban az állapotgép az *idle* állapotba lép át, a *ready* kimenet is magas szintet vesz fel, vagyis a számítás ezzel véget is ért.

Az *exceptions_2* állapotban nemcsak az operandusregisztereket, hanem az elvégzendő műveletet kijelölő *operation* paramétert is vizsgáljuk (lásd 4-208. ábra Verilog: 12. sor). Figyeljük meg, hogy az állapotgép – amennyiben nem detektál speciális operandust – az *operation* paraméter 0 és 1 értéke esetén egyaránt az összeadás művelet első vezérlési állapotán folytatja a végrehajtást, hiszen ezen a ponton kivonás esetén az *op_b_buf* regiszterbe már az *op_b* bemenet mínusz egyszerese van betöltve az *idle* állapot 5–6. sorában lévő feltételvizsgálatnak köszönhetően (lásd 4-206. ábra).

Az összeadás algoritmusának első vezérlési állapotában a közös *exp_c* karakterisztika és az operandus-mantisszák *m_offset_a/b* eltolásainak kiszámítása, valamint a rejtett bittel és a guard-bitekkel kibővített mantisszák *man_a* és *man_b* regiszterekbe töltése egyidejűleg történik meg. A 4-210. ábra áramköri sémáján jól látható, hogy a párhuzamosításnak nincs akadálya, hiszen ezeknek a műveleteknek a végrehajtására dedikált aritmetikai elemeket vezettünk be a műveletvégző egység tervezése során.

Az *add_1* vezérlési állapot működését a 4-211. ábra hullámformája egy összeadás művelet példáján keresztül szemlélteti. Az első órajel-felfutóélnél az *op_a/b_buf* operandusregiszterek, általuk pedig már az *exp_a/b* buszok értéke is stabil. Jól megfigyelhető az órajel második felfutóélénél, vagyis az *add_1* állapotban beállított vezérlőjelek hatásának érvényesülése az órajel harmadik



4-211. ábra. *Az add_1 vezérlési állapot működése*

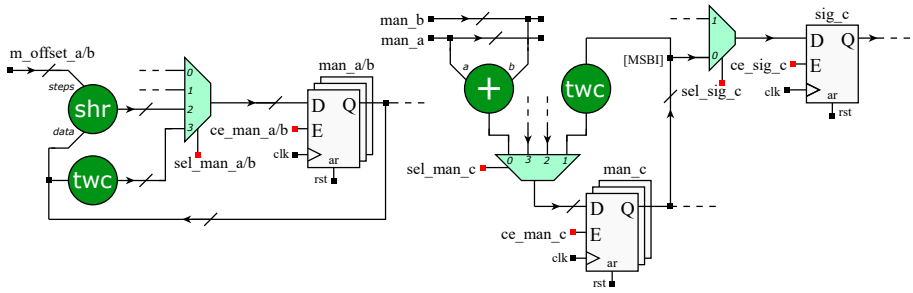
Pseudo – viselkedési modell

```
1  ADD: [...]
2
3       man_a <= man_a >> m_offset_a
4       man_b <= man_b >> m_offset_b
5       if ( op_a_buf[sig] == "1" ) man_a <= twc.result(man_a)  ; twc: two's complement
6       if ( op_b_buf[sig] == "1" ) man_b <= twc.result(man_b)
7
8       man_c <= man_a + man_b
9
10      sig_c <= man_c[MSBI]                                    ; MSBI: most significant bit index
11      if ( man_c[MSBI] == "1" ) man_c <= twc.result(man_c)
12
13      goto NORMALIZE
```

Verilog – vezérlőegység

```
1  add_2: begin sel_man_a <= 2; ce_man_a <= 1; sel_man_b <= 2; ce_man_b <= 1; next_state <= add_3; end
2
3  add_3: begin
4    if ( op_a_buf[$left(op_a_buf)] == 1 ) begin sel_man_a <= 3; ce_man_a <= 1; end
5    if ( op_b_buf[$left(op_b_buf)] == 1 ) begin sel_man_b <= 3; ce_man_b <= 1; end
6    next_state <= add_4;
7  end
8
9  add_4: begin sel_man_c <= 0; ce_man_c <= 1; next_state <= add_5; end
10
11 add_5: begin
12   sel_sig_c <= 0; ce_sig_c <= 1;
13   if ( man_c[$left(man_c)] == 1 ) begin sel_man_c <= 1; ce_man_c <= 1; end
14   next_state <= normalize_1;
15 end
```

Áramköri struktúra – műveletvégző egység



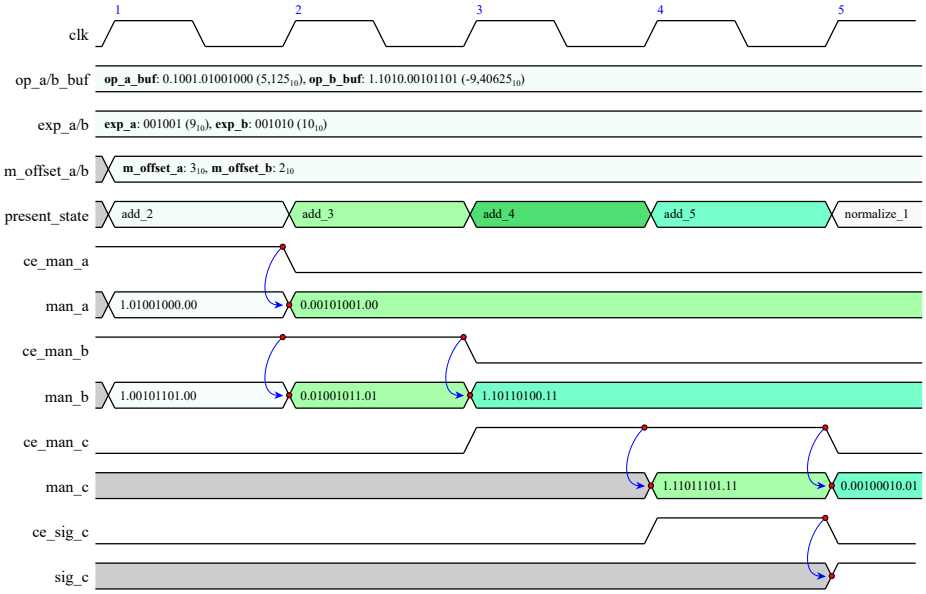
4-212. ábra. *A lebegőpontos aritmetikai egység vezérlő állapotgépének részlete (4)*

felfutóélénél; a közös karakterisztikát tároló *exp_c* regiszter, a két mantissza eltolását tároló *m_offset_a/b* regiszterek, illetve a rejtett bittel és a két guard-bittel kiegészített mantisszákat tároló *man_a/b* regiszterek egyaránt frissülnek.

Az összeadás algoritmusának következő lépései az operandus-mantisszák jobbra léptetése és – negatív előjel esetén – a jobbra léptetés eredményéül kapott vektor ketteskomplemens-kódolása. Az e műveletekhez szükséges vezérlőjeleket a 4-212. ábra Verilog kódrészletének 1–7. soraiban megfogalmazott *add_2* és *add_3* vezérlési állapotok állítják elő.

Az ily módon átalakított operandus-mantisszáknak a 4-212. ábra viselkedési modelljének 8. sora szerinti előjeles összeadása az *add_4* állapotban kiadott vezérlőjelek hatására történik meg. Az eredményül kapott, *man_c* regiszterben eltárolt vektor legfelső bitje már a végeredmény előjelét képviseli, így az *add_5* állapotban kiadott vezérlőjelek segítségével azt már el is tárolhatjuk a *sig_c* regiszterben (4-212. ábra Verilog: 12. sor). Ugyancsak az *add_5* állapot feladata, hogy a *man_c* regiszterben

előállt, kettes-komplemens kódolású összeget visszaalakítsa abszolútértékes formátumba (4-212. ábra Verilog: 13. sor). A 4-213. ábra hullámformája az *add_2 ... add_5* vezérlési állapotok működését szemlélteti.



4-213. ábra. *A mantisszák előjeles összegének előállítása az add_2 ... add_5 vezérlési állapotokban*

A példában 5,125-hez adunk hozzá -9,40625-et. Az *op_a/b_buf* operandusregiszterek, valamint az *exp_a/b* buszok már korábban, a műveletvégzés legelső állapotában kiadott vezérlőjelek hatására stabilizálódtak. Az órajelnek a 4-213. ábra számozása szerinti első felfutóélénél a mantisszák eltolásait tároló *m_offset_a/b* regiszterek értéke is előáll. Az *add_2* állapotban kiadott vezérlőjelek hatására a *man_a/b* regiszterekben tárolt értékeket az eltolások mértékének megfelelően jobbra léptetjük. A *man_b* regiszter esetén megfigyelhetjük a guard-bitek „pontosságmegőrző” működését; ezek nélkül az eredeti mantissza legalsó helyiértékén lévő egyes (0010110**1**) a jobbra léptetés áldozatául esett volna, így azonban hatása továbbra is érvényesül a számítás későbbi fázisaiban. Mivel az *op_b* operandus negatív előjelű, ezért az *add_3* állapotban kiadott vezérlőjelek hatására az órajel harmadik felfutóélénél a *man_b* regiszterbe a korábbi érték kettes-komplemense íródik. Az órajel negyedik felfutóélénél a *man_c* regiszterbe beíródik a két mantissza előjeles összege, amely ugyancsak negatív lett, ezért a következő vezérlési ciklusban, az órajel ötödik felfutóélénél visszaalakítjuk abszolútértékes formába, miközben a negatív előjelre utaló, legfelső helyiértéken lévő egyest átírjuk a *sig_c* regiszterbe. Az összeadás algoritmusának további lépései, a normalizálás és a kerekítés a szorzás algoritmusával közös vezérlési állapotok által valósulnak meg, ezért ezek részletes bemutatása előtt a szorzás műveletéhez tartozó vezérlést tekintjük át.

A lebegőpontos szorzás a számábrázolás sajátosságaiból adódóan némileg egyszerűbben megvalósítható, mint az összeadás. A mantissza és a karakterisztika normalizálás előtti állapotának eléréséhez – tehát ugyanaddig az állapotig, ameddig az összeadás algoritmusát is tárgyaltuk – mindössze két újabb vezérlési állapotra lesz szükségünk (lásd 4-214. ábra). A *mul_1* állapotban

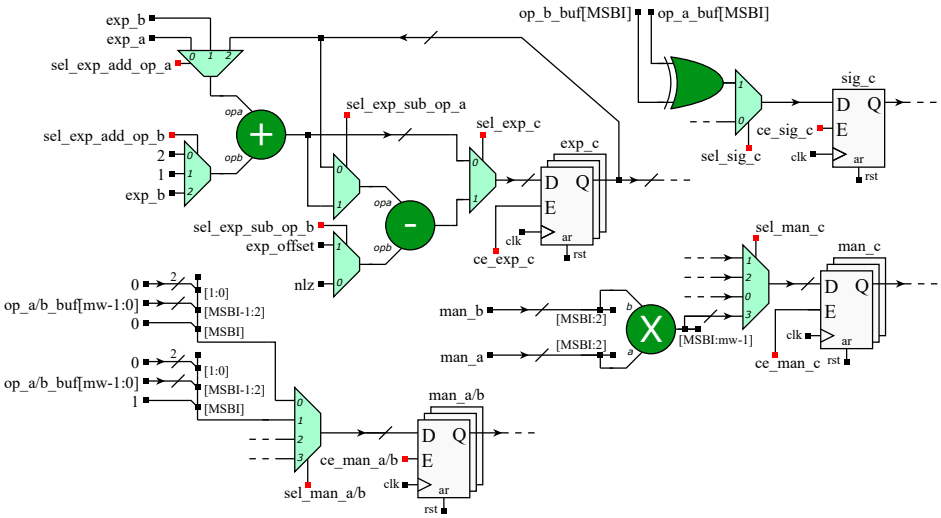
Pseudo – viselkedési modell

```
1  MUL: sig_c <= op_a_buf[sig] xor op_b_buf[sig]
2
3       exp_a <= ("00", op_a_buf[exp])                        ; A két karakterisztika összege az eltolást
4       exp_b <= ("00", op_b_buf[exp])                        ; kétszer tartalmazza -> egy eltolásnyit
5       exp_c <= (exp_a + exp_b) - (2**(ew-1)-1)              ; le kell vonni az összegből.
6
7       if ( exp_a == 0 ) man_a <= ("0", op_a_buf[man], "00")  ; A mantisszák képzése az összeadásnál
8       else man_a <= ("1", op_a_buf[man], "00")               ; látottakhoz hasonló:
9                                                             ;   - rejtett bit hozzáfűzése elölről
10      if ( exp_b == 0 ) man_b <= ("0", op_b_buf[man], "00")  ;   - 2 guard-bit hozzáfűzése hátulról
11      else man_b <= ("1", op_b_buf[man], "00")
12
13      man_c <= (man_a[MSBI:2] * man_b[MSBI:2])[MSBI:mw-1];
14      exp_c <= exp_c + 1
15
16      goto NORMALIZE
```

Verilog – vezérlőegység

```
1  mul_1: begin
2    sel_sig_c <= 1; ce_sig_c <= 1;
3    sel_exp_c <= 1; sel_exp_sub_op_a <= 1; sel_exp_sub_op_b <= 1;
4    sel_exp_add_op_a <= 0; sel_exp_add_op_b <= 2; ce_exp_c <= 1;
5
6    if ( exp_a == 0 ) sel_man_a <= 0; else sel_man_a <= 1;
7    if ( exp_b == 0 ) sel_man_b <= 0; else sel_man_b <= 1;
8    ce_man_a <= 1; ce_man_b <= 1;
9    next_state <= mul_2;
10 end
11
12 mul_2: begin
13   sel_man_c <= 3; ce_man_c <= 1;
14   sel_exp_c <= 0; sel_exp_add_op_a <= 2; sel_exp_add_op_b <= 1; ce_exp_c <= 1;
15   next_state <= normalize_1;
16 end
```

Áramköri struktúra – műveletvégző egység

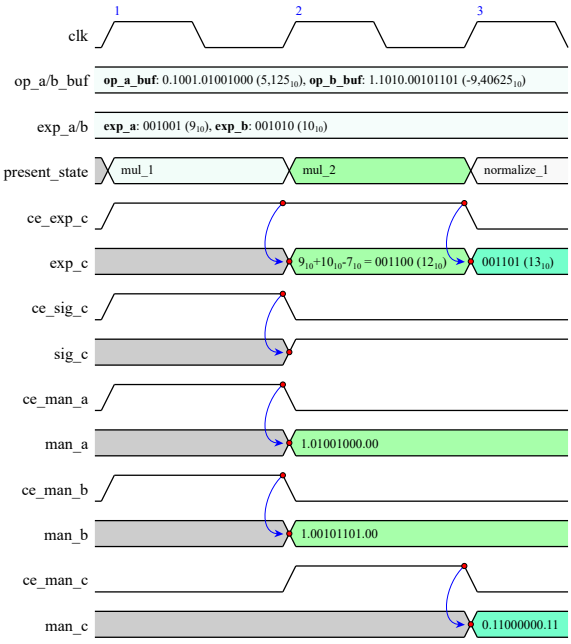


4-214. ábra. *A lebegőpontos aritmetikai egység vezérlő állapotgépének részlete (5)*

kiadott vezérlőjelek segítségével az operandusregiszterek legfelső helyiértékű bitjeinek *XOR* kapcsolatának értékét eltároljuk a *sig_c* regiszterben, előállítva ezáltal a szorzat előjelét a viselkedési modell első sorának megfelelően. Ugyancsak a *mul_1* állapotban generált vezérlőjelek segítségével

töltjük be a viselkedési modell ötödik sorában lévő értékadás jobbértékét a szorzat karakterisztikáját tároló *exp_c* regiszterbe, illetve – az összeadásnál látottakkal megegyező módon – az operandusok mantisszáinak rejtett bittel és guard-bitekkel kiegészített értékét a *man_a/b* regiszterekbe az órajelnek a 4-215. ábra számozása szerinti második felfutóélénél.

Ezek után a *mul_2* állapotban előállított vezérléssel a *man_a/b* regiszterek szorzatát az órajel harmadik felfutóélénél betöltjük a *man_c* regiszterbe, illetve inkrementáljuk az *exp_c* karakterisztika értékét az egészre normalizált formátum konzisztenciájának megtartása érdekében. A *mul_2* állapot tehát a viselkedési modell 13. és 14. soraiban rögzített műveleteket valósítja meg. Ezt a folyamatot a 4-215. ábra hullámformája egy példán keresztül szemlélteti.



4-215. ábra. *A szorzás magját megvalósító mul_1 és mul_2 vezérlési állapotok működése*

Akár összeadást, akár szorzást hajtunk végre, az eddig elvégzett műveletek eredményeként a *man_c* regiszterben előállhat olyan érték, amelynek legfelső helyiértékű bitje nulla, ami azt jelenti, hogy ebben a pillanatban az *exp_c* karakterisztika és a *man_c* mantissza együttese nem egészre normalizált alakú, hiszen a kettő nulladik hatványát képviselő bit értéke nulla. A 4-216. ábrán szemléltetett vezérlés segítségével végrehajtjuk a normalizálást, vagyis a *man_c* regisztert addig léptetjük balra, amíg annak legfelső bitje egyes nem lesz, miközben a karakterisztika értékét minden léptetésnél eggyel csökkentjük. Ezt az iteratív stílusban megfogalmazott eljárást a korábban megtervezett műveletvégző egységünk – megfelelő vezérléssel – valójában egyetlen órajelciklus alatt képes végrehajtani a 4-203. ábrán bemutatott *nlz_logic* és a *man_c_shl* logikák segítségével. A szükséges balra léptetések számát az *nlz_logic* nevű aszinkron aritmetikai elem (kombinációs hálózat) állapítja meg a *man_c* regiszter legfelső helyiértékein lévő nullák megszámlálásával, magát a léptetést pedig az ugyancsak aszinkron *man_c_shl* operátor hajtja végre. A vezérlő állapotgépnek a feladata tehát csupán az, hogy a *normalize_1* állapotban frissítse az *nlz* regiszter tartalmát annak *ce* jele segítségével,

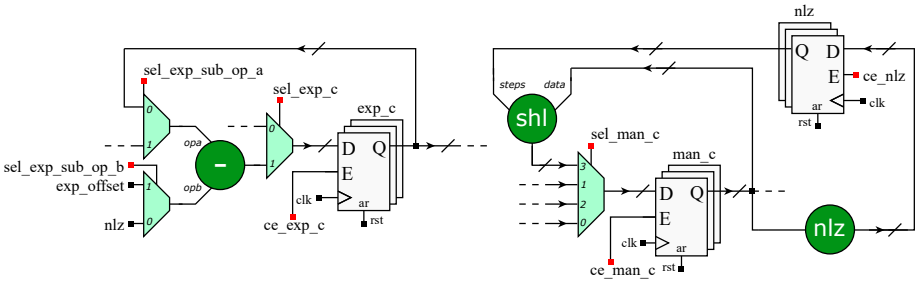
Pseudo – viselkedési modell

```
NORMALIZE: nlz <= nlz_logic.result(man_c) ; number of leading zeros: a man_c regiszter felső
           man_c <= man_c << nlz          ; helyiértékein lévő nullák száma
           exp_c <= exp_c - nlz
           goto ROUND
```

Verilog – vezérlőegység

```
normalize_1: begin ce_nlz <= 1; next_state <= normalize_2; end
normalize_2: begin
  sel_man_c <= 2; ce_man_c <= 1;
  sel_exp_c <= 1; sel_exp_sub_op_a <= 0; sel_exp_sub_op_b <= 0; ce_exp_c <= 1;
  next_state <= round_1;
end
```

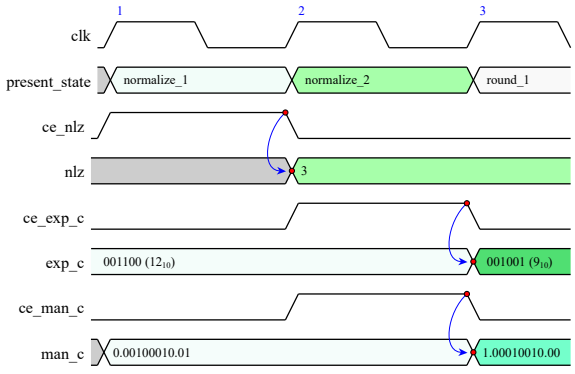
Áramköri struktúra – műveletvégző egység



4-216. ábra. *A lebegőpontos aritmetikai egység vezérlő állapotgépének részlete (6)*

majd a *normalize_2* állapotban a *man_c* regiszterbe betöltse a léptető logika kimenetét. A műveletvégző egység vonatkozó részletére tekintve láthatjuk, hogy az *op_a_buf* regiszterhez hasonlóan az *nlz* regiszter bemenetét meghajtó kombinációs hálózat sem tartalmaz multiplexereket, vagyis az *nlz* regiszter értékének frissítése az állapotgép részéről csupán a *ce* jelen előállított egyetlen pozitív impulzust igényel.

A normalizáláshoz tartozó vezérlési állapotok működését A 4-217. ábra hullámformája szemlélteti.



4-217. ábra. *A normalizálást megvalósító vezérlési állapotok működése*

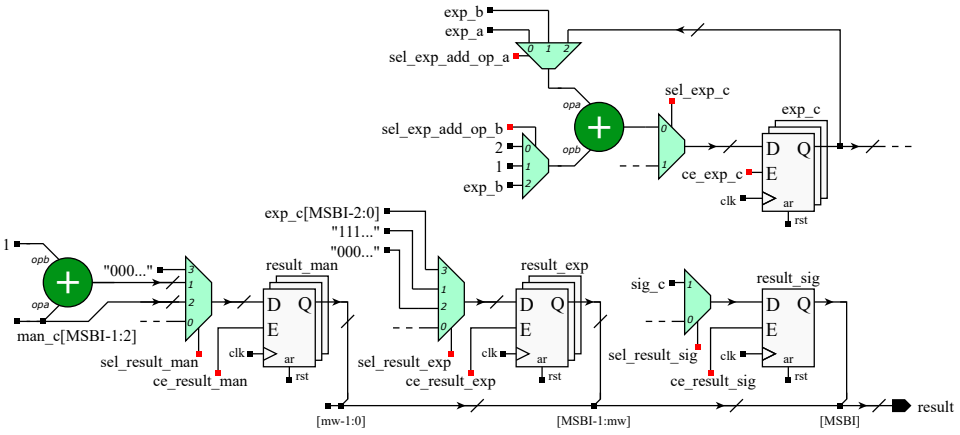
Pseudo – viselkedési modell

```
ROUND: if ( man_c[2:0] == "110" or man_c[1:0] == "11" ) {
         result_man <= man_c[MSBI-1:2] + 1
         if ( result_man == 0 ) exp_c <= exp_c + 1
       } else result_man <= man_c[MSBI-1:2]

OVER_UNDERFLOW: result_sig <= sig_c
                if ( signed(exp_c) > (2**ew)-2 ) { result_exp <= "111..." result_man <= "000..." }
                elsif ( signed(exp_c) < 1 ) { result_exp <= "000..." result_man <= "000..." }
                else result_exp <= exp_c[MSBI-2:0]

STOP: result <= (result_sig, result_exp, result_man)
      ready <= 1
      goto START
```

Verilog – vezérlőegység

```
round_1: next_state <= round_2;
round_2: begin
  if ( man_c[2:0] == 3'b110 || man_c[1:0] == 2'b11 ) begin
    sel_result_man <= 1; ce_result_man <= 1; next_state <= round_3;
  end else begin sel_result_man <= 2; ce_result_man <= 1; next_state <= over_underflow_1; end
end

round_3: next_state <= round_4;
round_4: begin
  if ( result_man == 0 ) begin
    sel_exp_c <= 0; sel_exp_add_op_a <= 2; sel_exp_add_op_b <= 1; ce_exp_c <= 1;
  end
  next_state <= over_underflow_1;
end

over_underflow_1: begin sel_result_sig <= 1; ce_result_sig <= 1; next_state <= over_underflow_2; end
over_underflow_2: begin
  if ( $signed(exp_c) > (2**ew)-2 ) begin
    sel_result_exp <= 1; ce_result_exp <= 1;
    sel_result_man <= 3; ce_result_man <= 1;
  end else if ( $signed(exp_c) < 1 ) begin
    sel_result_exp <= 2; ce_result_exp <= 1;
    sel_result_man <= 3; ce_result_man <= 1;
  end else begin sel_result_exp <= 3; ce_result_exp <= 1; end
  next_state <= idle;
end
```

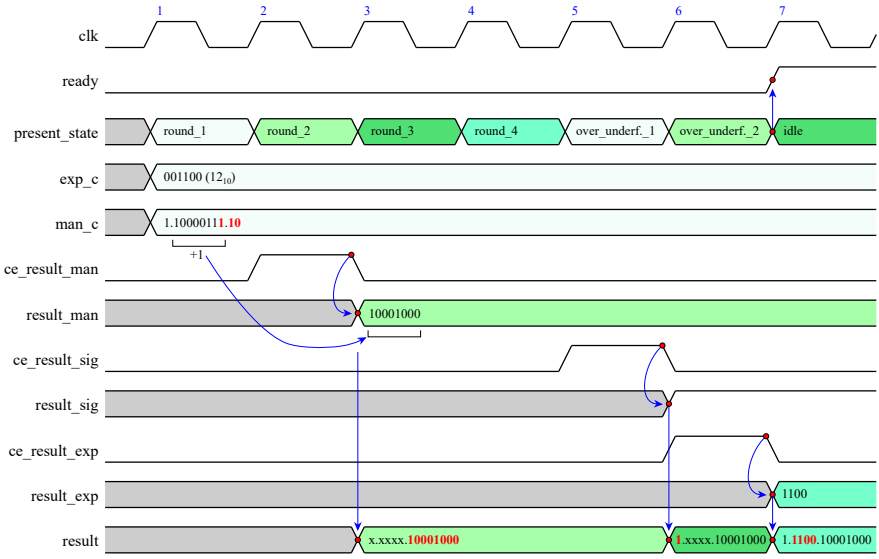
Áramköri struktúra – műveletvégző egység



4-218. ábra. *A lebegőpontos aritmetikai egység vezérlő állapotgépének részlete (7)*

A normalizálást a kerekítés és az esetleges túl- vagy alulcsordulás detektálásának lépései követik (lásd 4-218. ábra). Az ezekben a lépésekben kiszámított eredményeket már az IP-mag kimeneteit közvetlenül meghajtó *result_sig*, *result_exp* és *result_man* regiszterekbe töltjük át. Figyeljük meg, hogy a viselkedési modell első sorában foglaltaknak megfelelő *round_2*[172] állapotban a vezérlés kettéágazik; ha a kerekítés miatt a mantissza értékének megnövelésére kényszerülünk, akkor kénytelenek vagyunk annak eredményét is megvizsgálni, és az ott bekövetkező esetleges túlcsordulást a karakterisztika megnövelésével kompenzálni. Ebben az esetben a *round_2* állapotot a *round_3* és *round_4* állapotok követik (lásd 4-218. ábra Verilog: 3–4. sor).

Ha azonban a guard-bitek értéke nem teszi szükségessé a mantissza értékének megnövelését, akkor a kerekítés egyetlen órajelciklus alatt végbemegy azáltal, hogy a guard-biteket egyszerűen figyelmen kívül hagyjuk, és a *man_c* regiszter többi bitjét – a rejtett bitet ugyancsak elhagyva – áttöltjük a *result_man* regiszterbe, a végrehajtás pedig az *over_underflow_1* állapotra ugrik (lásd 4-218. ábra Verilog: 5. sor). Mindezek után az összeg/szorzat karakterisztikájának és mantisszájának végleges értéke a karakterisztika belső reprezentációjának *(exp_c)* vizsgálata alapján képezhető. Ha az eddigi számítások eredményeképp az *exp_c* regiszter értéke kívül esik a számábrázolás által lehetővé tett tartományon (lásd 4-218. ábra Verilog: 18. és 21. sor), akkor a *result_exp* és *result_man* kimeneti regiszterekbe az azok bemeneteit hajtó multiplexerek kiválasztójeleinek alkalmas vezérlésével a megfelelő speciális szimbólumokat képviselő bináris kombinációkat töltjük. Ha a karakterisztika értéke megfelelő, akkor már csak a *result_exp* regiszterről kell gondoskodnunk, ami az *exp_c* regiszter értékét kapja a felső két extra bit leválasztása után. A kerekítés és túl- vagy alulcsordulás-detektálás vezérlésének lépéseit a 4-219. ábra hullámformáján követhetjük nyomon.



4-219. ábra. *A kerekítést és a túl- vagy alulcsordulás detektálását végző vezérlési állapotok működése*

[172] A *round_1* állapot ezúttal is csak az állapotgép működésének késleltetését szolgálja, mert a kerekítéskor szükségünk van a *man_c* regisztrernek a normalizálás során balra léptetett értékére (lásd 4-218. ábra Verilog: 3. sor), ami a *round_1* állapotban még nem vizsgálható.

Ebben a példában azt látjuk, hogy a guard-bitek értéke alapján felfelé kerekítést hajtunk végre, amelynek során a mantisszában nem keletkezik túlcsordulás, ezért az *exp_c* regiszter értékének inkrementálására nem kerül sor. A *result_man*, *result_sig* és *result_exp* regiszterek rendre az órajel harmadik, hatodik és hetedik felfutóélénél veszik fel végleges értéküket, és e pillanatokban a *result* kimenet is frissül. A *result* kimenet tehát a hetedik órajel-felfutóélnél veszi fel végleges értékét, amit a *ready* kimenet magas szintre húzásával a felhasználó felé is jelzünk.

4.5.3.4. Órajel-szinkronizálás

A lebegőpontos aritmetikai egység felhasználói interfészének működése megfelel az 1.2.1.1. pontban bemutatott szinkron kézfogásos interfész-specifikációnak, vagyis a 2-2. ábra órajel-szinkronizáló fokozata ebben az esetben is használható. Az általánosított $A_{i,S}$ és $B_{k,S}$ vektorok a 4-11. táblázat szerint feleltethetők meg a lebegőpontos aritmetikai egység interfészjeleinek.

Általános CDC-logika	Lebegőpontos egység
$A_{i,S}$	*op_a, op_b*
$B_{k,S}$	*result*

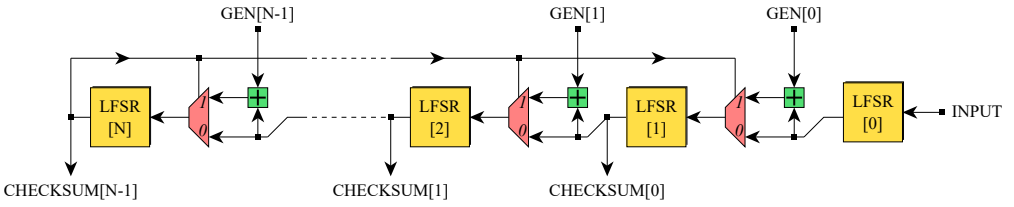
4-11. táblázat. *A lebegőpontos aritmetikai egység és az általánosított CDC-logika jeleinek megfeleltetése*

4.5.4. CRC-kódgenerálás visszacsatolt léptetőregiszterrel

Áttanulmányozásra javasolt alfejezetek:

5.4. *A ciklikus redundanciaellenőrzés elve*

A nagyméretű, összefüggő adathalmazok integritásának vizsgálatára használt ciklikus redundanciaellenőrzés egy összetett matematikai művelet, amely bináris együtthatójú polinomok osztására vezethető vissza (lásd 5.4. Függelék). Az algoritmus a vizsgálandó adathalmaz bitjeit egy polinom együtthatóinak tekinti, amelyet egy állandó értékű, ugyancsak bináris együtthatókkal leírt, ún. generátor-polinommal oszt le. A CRC-kód nem más, mint az osztás maradékaként előálló polinom bináris együtthatóiból álló bitsorozat. Összetettsége ellenére az algoritmus magja meglepően könnyen megvalósítható digitális áramkörökkel, egészen pontosan egy, a 4-220. ábrán szemléltetett szerkezetű, ún. visszacsatolt léptetőregiszter-struktúrával.



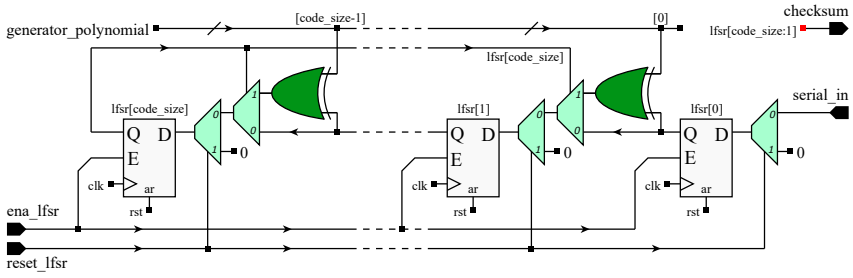
4-220. ábra. *Visszacsatolt léptetőregiszter szerkezeti vázlata (vö. 5-26. ábra). N a generált CRC-kód mérete*

E vázlatos szerkezeti modell alapján RTL-modellt készíteni igazán egyszerű, mivel a benne található funkcionális elemek gyakorlatilag megegyeznek az RTL-modelleket általában is felépítő részegységekkel; az egybites tárolóelemek szerepét flip-flopok tölthetik be, az egybites kivonás/összeadás funkciót *XOR* kapuk valósíthatják meg, a multiplexerek pedig regiszter-transzfer szinten is multiplexerek. Ez a modell bizonyos szempontból mégiscsak elnagyolt; nem utal rá, hogy a léptetés minek a hatására történik meg, illetve az a – funkcionális szempontból egyébként nagyon fontos – részlet sem olvasható ki belőle, hogy két független adathalmazra vonatkozó CRC-kódgenerálás között a léptetőregiszter tárolóit feltétlenül nullázni kell. Ha tehát rögzítjük, hogy e két szolgáltatás, a léptetés és a nullázás engedélyezésére egy-egy külön vezérlőbemenetet hozunk létre, akkor a visszacsatolt léptetőregiszter RTL-modellje már valóban elkészíthető (lásd 4-221. ábra).

VHDL

```
1  L_LFSR: process ( clk, rst ) begin
2   if ( rst = '1' ) then lfsr <= (others => '0');
3   elsif ( rising_edge(clk) ) then
4   |
5   |  if ( ena_lfsr = '1' ) then
6   |  |
7   |  |  if ( reset_lfsr = '1' ) then lfsr <= (others => '0');
8   |  |  else
9   |  |  |
10  |  |  |  lfsr(0) <= serial_in;
11  |  |  |  for i in 1 to code_size loop
12  |  |  |  |  if ( lfsr(code_size) = '1' ) then lfsr(i) <= generator_polynomial(i-1) xor lfsr(i-1);
13  |  |  |  |  else lfsr(i) <= lfsr(i-1);
14  |  |  |  |  end if;
15  |  |  |  end loop;
16  |  |  |
17  |  |  end if;
18  |  |
19  |  end if;
20  |
21   end if;
22 end process;
23
24 checksum <= lfsr(code_size downto 1);
```

Áramköri struktúra



4-221. ábra. *Visszacsatolt léptetőregiszter RTL-modellje*

CRC-kódok előállítására ez az áramkör már önmagában is alkalmas, de használatát nagyban megkönnyíthetjük, ha – az I²C master modulnál látottakhoz hasonlóan (vö. 4.3.3. pont) – olyan burkolómodult hozunk létre hozzá, amelynek felhasználói interfésze egyrészt kompatibilis a szinkron kézfogásos interfész-specifikációval, másrészt jobban illeszkedik a tipikus felhasználási

körülményekhez. A forrásadat, amely alapján a CRC-kódot ki kell számítani, jellemzően „darabokban” válik hozzáférhetővé. Ha egy soros-párhuzamos átalakítótól, pl. egy UART-vevőtől kapjuk a kódolandó adatokat, akkor e darabok az UART adatkereteinek, egy memóriában tárolt adathalmazon végzett ellenőrzés esetén pedig a memóriából egy lépésben – egy olvasástranzakcióval – kiolvasható bejegyzéseknek felelnek meg. A 4-222. ábrán egy olyan burkolómodul *(crc_chunk_feed_wrp)* RTL-modelljét láthatjuk, amely egy szinkron kézfogásos interfészen vesz át egy *chunk_size* méretű adattöredéket a felhasználótól, és annak minden bitjét automatikusan belépteti a 4-221. ábrán látható léptetőregiszter egy példányába.

A burkolómodul működése a következő; a *request* bemenet felfutása hatására a *ready* jelet nullázza, és egy belső lefelé számlálót *(bit_counter) chunk_size-1* értékre inicializál (4-222. ábra: 26–27. sor). A *chunk_in* bemeneten átvett adattöredéket a számlálóval megcímezve (4-222. ábra: 34. sor) órajelciklusonként előállítja a léptetőregiszter soron következő bemenetét. Ha a *bit_counter*

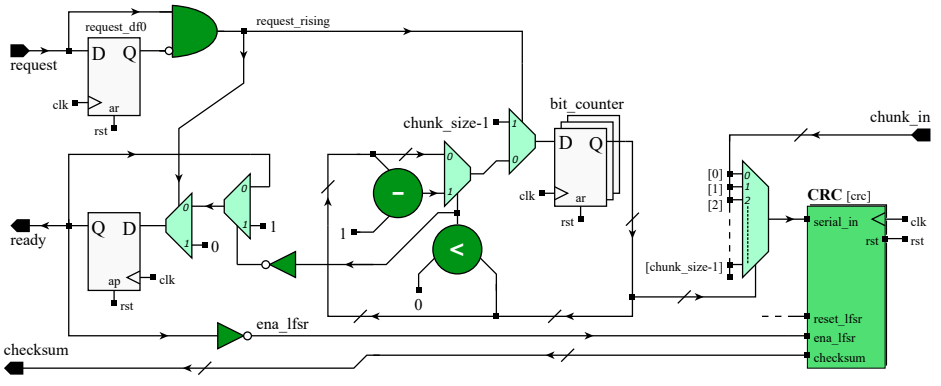
VHDL

```
1  L_REQUEST_EDGE_DETECTOR: process ( clk, rst )
2  begin
3  | if ( rst = '1' ) then
4  |   request_df0 <= '0';
5  |   ready <= '1';
6  | elsif ( rising_edge(clk) ) then
7  | |
8  | | request_df0 <= request;
9  | |
10 | | if ( request_rising = '1' ) then
11 | |   ready <= '0';
12 | | elsif ( bit_counter = 0 ) then
13 | |   ready <= '1';
14 | | end if;
15 | |
16 | end if;
17 end process;
18
19 request_rising <= request and not request_df0;
20
21 ena_lfsr <= not ready;
22
```

VHDL (folyt.)

```
23 L_BIT_COUNTER: process ( clk, rst ) begin
24 | if ( rst = '1' ) then bit_counter <= 0;
25 | elsif ( rising_edge(clk) ) then
26 | | if ( request_rising = '1' ) then
27 | | | bit_counter <= chunk_size-1;
28 | | elsif ( bit_counter > 0 ) then
29 | | | bit_counter <= bit_counter - 1;
30 | | end if;
31 | end if;
32 end process;
33
34 crc_serial_in <= chunk_in(bit_counter);
35
36 L_CRC: entity work.crc(rtl)
37 generic map (
38 |  code_size            => code_size,
39 |  generator_polynomial => generator_polynomial
40 )
41
42 port map (
43 | clk        => clk,
44 | rst        => rst,
45 | ena_lfsr   => ena_lfsr,
46 | reset_lfsr => [...],
47 | serial_in  => crc_serial_in,
48 | checksum   => checksum
49 );
```

Áramköri struktúra



4-222. ábra. *Burkolómodul a visszacsatolt léptetőregiszterhez (1)*

számláló eléri a nullát, akkor a *ready* jelet magas szintre húzza, jelezve, hogy az adattöredék beléptetése megtörtént (4-222. ábra: 12–13. sor). Egy-egy adattöredék bitjeinek beléptetése során az *lfsr* regiszter *chunk_size* alkalommal vesz fel újabb és újabb értékeket, amelyek közül az utolsót a következő adattöredék érkezéséig, vagyis a következő felhasználói kézfogásos tranzakcióig meg is őrzi.

A CRC-kód által „lefedett" adatfolyam méretéről eddig nem esett szó, mivel arra nincs elvi felső korlát. A 4-222. ábrán bemutatott burkolómodul *chunk_size* méretű adattöredékeket vár a felhasználótól, amelyek száma tulajdonképpen akármennyi lehet. Az utolsó töredéknek speciális szerepe lesz a végső CRC-kód kiszámításában. E speciális szerep nem a CRC-kód kiszámításának elvéből, hanem gyakorlati megfontolásokból következik az alábbiak szerint.

Egy CRC-kóddal ellátott adatfolyamban rendszerint be tudjuk azonosítani a CRC-kódot, mert annak pozíciója magához az üzenethez képest kötött. Ilyenkor előállíthatjuk helyben is az üzenet CRC-kódját, amit aztán összevethetünk a kapott CRC-kóddal. Ha a kettő egyezik, akkor az üzenet sértetlen, ellenkező esetben valami hiba történt az átvitel során. Az ellenőrzésnek azonban van egy másik, elterjedten alkalmazott formája is. Ha a küldő fél a CRC-kód kiszámításakor az eredeti üzenet végére egy, a CRC-kód méretével megegyező hosszúságú, nullákból álló bitsorozatot fűz, akkor értelemszerűen egy másik CRC-kód generálódik.[173] Ennek az alternatív CRC-kódnak az a sajátossága, hogy ha a fogadó fél az eredeti (nullákkal nem kiegészített) üzenet után ezt is belépteti a saját CRC-léptetőregiszterébe, akkor hibátlan átvitel esetén nullát kell kapnia eredményül. E megoldás előnye, hogy az ellenőrzés nem igényel a kódmérettel megegyező méretű komparátort.[174]

Az alternatív kódot CRC-modulunk esetén úgy állíthatjuk elő, hogy az utolsó adattöredék beléptetése után további *code_size+1* darab nullát léptetünk be.[175] A nullák számolgatását nem bízzuk a felhasználóra, helyette bevezetünk egy új, egybites vezérlőbemenetet *(last_chunk)*, amelynek segítségével a felhasználó jelezheti, hogy az éppen átadott adattöredék az utolsó lesz a sorban.[176] A burkolómodulon elvégzendő módosításokat a 4-223. ábra mutatja be.

Ha a *request* jel felfutásakor a *last_chunk* bemenet magas szintű, akkor a beléptetendő bitek számát meghatározó lefelé számláló értékét *chunk_size-1* helyett *chunk_size+code_size* értékűre inicializáljuk (4-223. ábra: 30. sor), így mire a számláló eléri a nullát, addigra pontosan a megfelelő számú léptetés történik meg ahhoz, hogy a léptetőregiszter végértéke éppen a teljes bitfolyamra vonatkozó CRC-kód legyen. Ne feledkezzünk meg arról sem, hogy a léptetőregiszter soros bemenetét a hasznos adatbitek után – vagyis a visszafelé számláló *code_size*-nál kisebb értékei mellett – nullákkal kell kiegészítenünk (4-223. ábra: 41. sor).

További fontos, és a burkolómodul jelenlegi formájában még elhanyagolt követelmény, hogy egy-egy bitfolyam feldolgozása után a felhasználónak képesnek kell lennie nullázni a léptetőregisztert, mielőtt újabb bitfolyamot kezdene el kódolni. A nullázást egy újabb vezérlőbemenet *(reset_lfsr)* teszi lehetővé. A szükséges módosítások a 4-224. ábrán láthatók.

---

[173] Ez azt jelenti, hogy az egyedinek tekintett eredeti üzenetet kiegészítjük, amitől magának az üzenetnek a tartalma nem változik meg. Az 5.4. Függelék terminológiájában úgy is fogalmazhatunk, hogy az üzenetpolinomot megszorozzuk 2^N-nel, ahol N a befűzött nullák száma.

[174] A csupa nullából álló CRC-kód akár magának az üzenet végének az azonosítására is alkalmas lehet, ha maga a protokoll azt nem jelöli ki egyértelműen valami más módszerrel. Ezt CRC-alapú keretezésnek nevezzük.

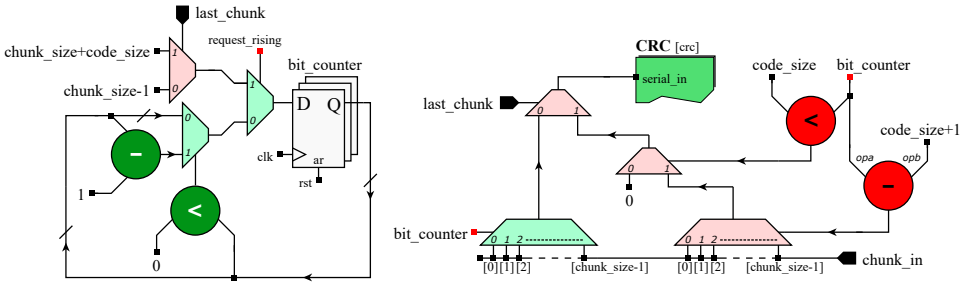
[175] A +1 léptetésre azért van szükség, mert a 4-221. ábra léptetőregiszterének legkisebb helyiértékű bitet képviselő flip-flopja csak egy buffer, ami a kód kiszámításában nem játszik szerepet. Ahhoz, hogy a legutolsó nulla is érvényre jusson, azt át kell léptetni a CRC-kód 1. helyiértékű bitjét képviselő flip-flopba.

[176] Ezt a megoldást az előre nem ismert hosszúságú adatfolyamok végének azonosítására a szabványos kommunikációs buszrendszerek is alkalmazzák (lásd pl. AMBA AXI-Stream protokoll).

VHDL

```
23 L_BIT_COUNTER: process ( clk, rst ) begin
24   if ( rst = '1' ) then bit_counter <= 0;
25   elsif ( rising_edge(clk) ) then
26
27     if ( request_rising = '1' ) then
28
29       if ( last_chunk = '0' ) then bit_counter <= chunk_size-1;
30       else bit_counter <= chunk_size+code_size;
31       end if;
32
33     elsif ( bit_counter > 0 ) then bit_counter <= bit_counter - 1;
34     end if;
35
36   end if;
37 end process;
38
39 crc_serial_in <= chunk_in(bit_counter) when last_chunk = '0' else
40                  chunk_in(bit_counter-(code_size+1)) when bit_counter > code_size else
41                  '0';
```

Áramköri struktúra

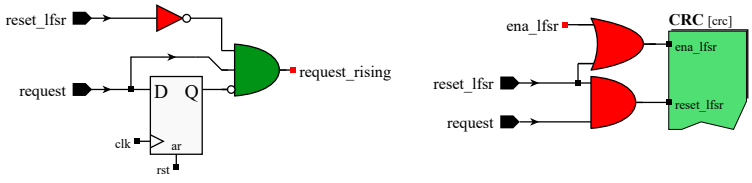


4-223. ábra. *Burkolómodul a visszacsatolt léptetőregiszterhez (2)*

VHDL

```
1  request_rising <= request and not request_df0 and not reset_lfsr;
2
3  L_CRC: entity work.crc(rtl)
4  generic map (
5    code_size            => code_size,
6    generator_polynomial => generator_polynomial
7  )
8
9  port map (
10   clk        => clk,
11   rst        => rst,
12   ena_lfsr   => ena_lfsr or reset_lfsr,
13   reset_lfsr => request and reset_lfsr,
14   serial_in  => crc_serial_in,
15   checksum   => checksum
16 );
```

Áramköri struktúra



4-224. ábra. *Burkolómodul a visszacsatolt léptetőregiszterhez (3)*

Ha a *request* bemenet felfutásakor a *reset_lfsr* bemenet magas szintű, akkor a *request_rising* impulzus generálását letiltjuk (4-224. ábra: 1. sor). Így a számláló nem „szerez tudomást” a felhasználó kéréséről, tehát nem aktiválódik. Ugyanakkor a 12. sorban megfogalmazott *OR* kapu segítségével magát a léptetőregisztert ekkor is engedélyezzük úgy, hogy közben a saját *reset_lfsr* bemenetét magas szintre húzva a léptetés helyett az nullázás funkciót aktiváljuk (4-224. ábra: 13. sor). A burkolómodul ebben az esetben a *ready* jelet egyáltalán nem húzza nullába, ami a szinkron kézfogásos interfész-specifikáció nulla késleltetésű változatának megfelelő viselkedés.

4.5.4.1. Órajel-szinkronizálás

A visszacsatolt léptetőregiszter köré tervezett burkolómodul felhasználói interfészének működése megfelel az 1.2.1.1. pontban bemutatott szinkron kézfogásos interfész-specifikációnak, vagyis a 2-2. ábra órajel-szinkronizáló fokozata ebben az esetben is használható. Az általánosított $A_{i,S}$ és $B_{k,S}$ vektorok a 4-12. táblázat szerint feleltethetők meg a *CRC* modul interfészjeleinek.

Általános CDC-logika	CRC modul
$A_{i,S}$	*chunk_in, last_chunk, reset_lfsr*
$B_{k,S}$	*checksum*

4-12. táblázat. *A CRC modul és az általánosított CDC-logika jeleinek megfeleltetése*

4.5.5. FFT

Áttanulmányozásra javasolt alfejezetek:

5.6. *Az FFT-algoritmus*
4.2. *On-chip adattárolás*
4.5.1. *Aritmetika és jelfeldolgozás – Fixpontos aritmetikai egység*

4.5.5.1. Viselkedési modell

Ebben az alfejezetben egy olyan IP-mag RTL-tervét mutatjuk be, amely képes egy komplex számokból álló mintasor diszkrét Fourier-transzformáltjának FFT-algoritmussal való kiszámítására. A szintetizálható RTL-modell elkészítéséhez az 5.6. Függelék alapján kidolgozott, Python nyelven megfogalmazott viselkedési leírás szolgál alapul (lásd 4-225. ábra).

Az RTL-tervezés specifikációjának tekinthető viselkedési modellben az FFT-algoritmust egy függvény írja le. A függvény paraméterei a bemeneti – és egyúttal a transzformált – vektor elemszáma *(N)*, a bemeneti vektor elemeinek valós és képzetes részét tartalmazó tömbök *(x_re, x_im)*, valamint a komplex egységvektor (az 5.6. Függelékben *W*) *N/2* db hatványát, vagyis a forgatási együtthatókat tartalmazó ún. cosinus-tábla. A függvény a transzformált vektor elemeinek valós és képzetes részét egy-egy tömbben adja vissza *(ram_re, ram_im)*. A cosinus-tábla mint paraméter némi kifejtést érdemel. A *W* komplex egységvektor definíciója a következő:

Python – viselkedési modell

```
def fft(N, x_re, x_im, costable):

  ram_re = N * [0]; ram_im = N * [0]

  q = 1           # A bemeneti résztranszformáltak elemszáma.
  r = 0           # A bemeneti résztranszformáltakon belüli futóindex.
  aa = 0          # A butterfly-művelet "a" operandusának címe.
  ab = 1          # A butterfly-művelet "b" operandusának címe.
  aw = 0          # A forgatási együttható cosinus-táblabeli címe (egyúttal W hatványkitevője).
  tfps = int(N/2) # "twiddle-factor power step": A forgatási együttható hatványkitevő-növelésének mértéke.

  for i in range(N):                              # A bemeneti vektor átrendezése
    reva = bit_reverse(i, math.log2(N))           # (bit-reverse re-sorting).
    ram_re[i] = x_re[reva]; ram_im[i] = x_im[reva];

  for i in range(int(math.log2(N))):

    for j in range(int(N/2)):

      op_a_re = ram_re[aa]; op_a_im = ram_im[aa]; op_b_re = ram_re[ab]; op_b_im = ram_im[ab]

      w_re = costable[aw]
      if ( aw < int(N/4) ): w_im = costable[aw+int(N/4)]
      else: w_im = -1*costable[aw-int(N/4)]

      (bfres_a, bfres_b) = butterfly((op_a_re, op_a_im), (op_b_re, op_b_im), (w_re, w_im))

      ram_re[aa] = bfres_a[0]; ram_im[aa] = bfres_a[1]; ram_re[ab] = bfres_b[0]; ram_im[ab] = bfres_b[1]

      if ( r == q-1 ): r = 0; aa = aa + q + 1; aw = 0
      else: r = r + 1; aa = aa + 1; aw = aw + tfps
      ab = aa + q

    q = q * 2; aa = 0; aw = 0; tfps = int(tfps / 2);
    ab = q

  return (ram_re, ram_im)
```

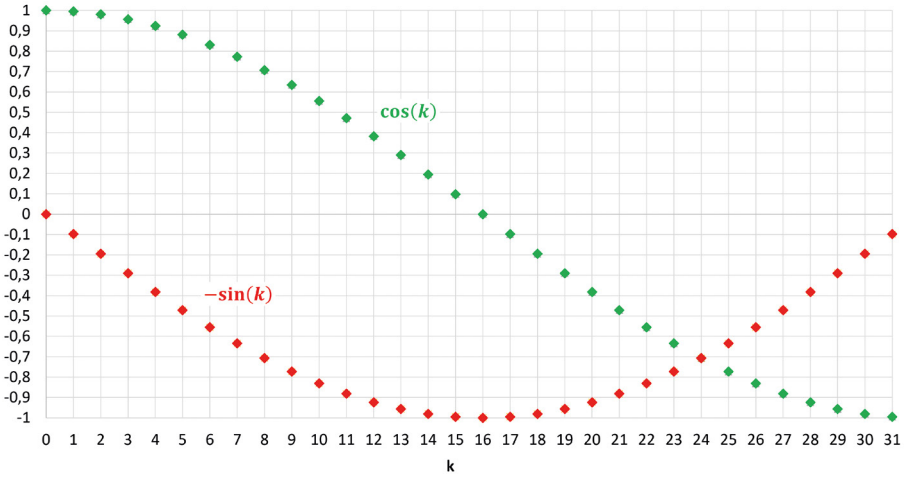
4-225. ábra. *Az FFT-algoritmus Python implementációja*

$$W = e^{-j\frac{2\pi}{N}}$$

Ahogy az az FFT-algoritmus működését grafikusan szemléltető 5-33. ábrán jól látható, ennek a komplex egységvektornak a $0 \ldots \left(\frac{N}{2} - 1\right)$-edik hatványaira lesz szükségünk. W különböző hatványai ugyancsak egységnyi abszolútértékű komplex számok, amelyek algebrai alakban az Euler-formula segítségével fejezhetők ki:

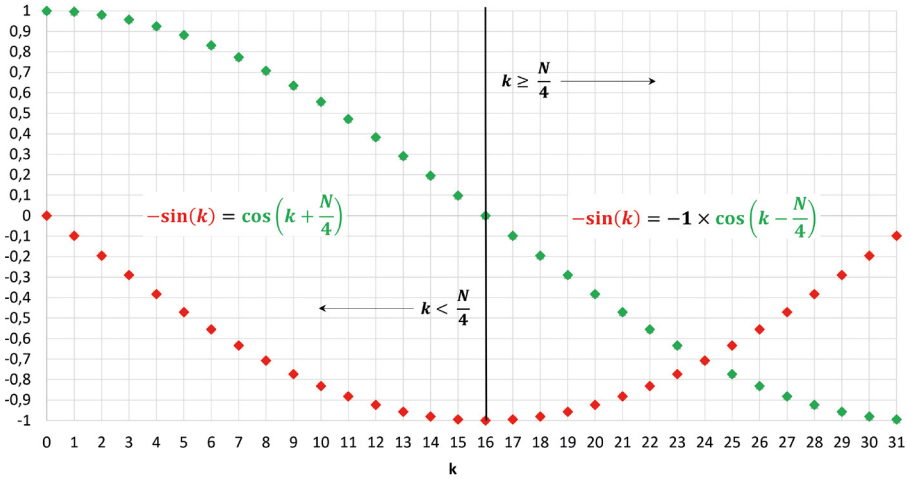
$$e^{-j\frac{2\pi}{N}k} = \cos\left(-\frac{2\pi}{N}k\right) + j\sin\left(-\frac{2\pi}{N}k\right) = \cos\left(\frac{2\pi}{N}k\right) + j\left[-\sin\left(\frac{2\pi}{N}k\right)\right], \qquad k = 0 \ldots \frac{N}{2} - 1$$

A zölddel kiemelt kifejezés a komplex egységvektor k-adik hatványainak valós részét, a pirossal kiemelt kifejezés pedig azok képzetes részét képviseli. Látható tehát, hogy az algoritmus végrehajtásához a cos és -sin függvények k szerinti helyettesítési értékeire lesz szükségünk (lásd 4-226. ábra).



4-226. ábra. *A cos(k) és -sin(k) függvények FFT-algoritmusban felhasznált helyettesítési értékei (N=64 esetén)*

A trigonometrikus függvények helyettesítési értékei többféle módon kiszámíthatók (pl. az ún. CORDIC algoritmussal). Ebben a megvalósításban azonban a függvények értékeit – mivel azokra csak aránylag kisszámú diszkrét pontban van szükségünk – nem számítjuk ki, hanem egy memóriában tároljuk. A 4-226. ábrán is jól látható, hogy a cos és -sin függvények felfoghatók egyazon függvény két, egymáshoz képest fázisban $\frac{\pi}{4}$ radiánnal eltolt változataiként, vagyis diszkrét pontokban értelmezett értékkészletük ugyanazokból a számokból áll. Ennek következtében tehát elegendő az egyik függvény értékeit eltárolni, a másik függvény értékei azokból minimális aritmetikával származtathatók (lásd 4-227. ábra).



4-227. ábra. *A -sin(k) függvény értékeinek származtatása a cos(k) függvény értékeiből*

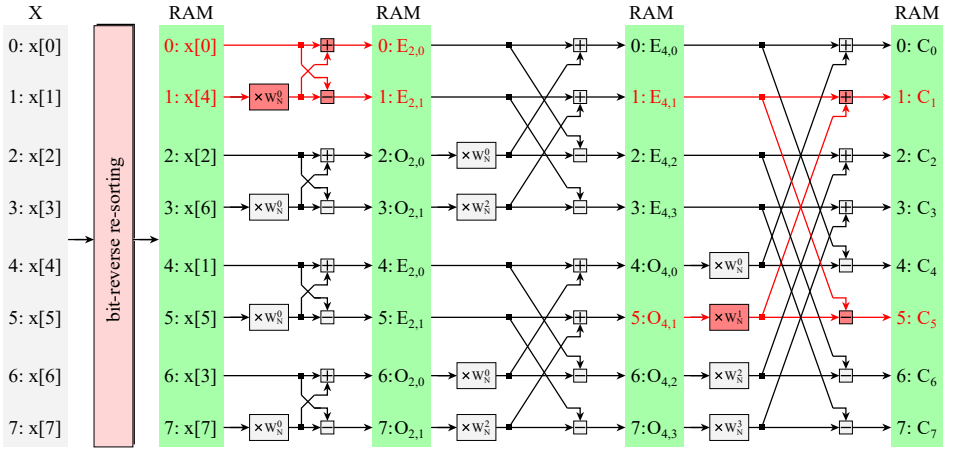
A cos függvény $\frac{2\pi}{N}k$ helyen vett értékeit eltárolva a komplex egységvektor különböző hatványainak valós részét a cosinus-tábla *k*-adik bejegyzése adja, a képzetes részt pedig a következőképpen kaphatjuk meg:

$$Im\{W^k\} = costable\left[k + \frac{N}{4}\right], \qquad k < \frac{N}{4}$$

$$Im\{W^k\} = -1 \times costable\left[k - \frac{N}{4}\right], \qquad k \geq \frac{N}{4}$$

A viselkedési modellben a cosinus-táblát az FFT függvény paraméterként veszi át, érzékeltetve, hogy a cos függvény helyettesítési értékeit a leendő IP-magba is kívülről kell majd betölteni.

Az algoritmus végrehajtása a viselkedési modell szerint a bemeneti mintasornak *(x_re, x_im)* egy belső memóriába *(ram_re, ram_im)* való áttöltésével kezdődik (4-225. ábra: 12–14. sor). Az *x* tömb *i*-edik eleme azonban nem a *ram* tömb *i*-edik rekeszébe kerül, mivel az *x* tömb olvasási indexe *(i)* és a *ram* tömb beírási indexe *(reva)* között egy transzformációt hajtunk végre, amely a tömbindex bináris számjegyeinek sorrendjét megfordítja (*bit_reverse*; pl. $100_2 \rightarrow 001_2$). Ennek az az eredménye, hogy a *ram* tömbbe az *x* tömb elemei átrendezve íródnak be. Erre az átrendezésre az algoritmust megvalósító áramkörhöz felhasznált belső memória mennyiségének minimalizálása céljából van szükség. A 4-228. ábra a 8 elemű mintasort feldolgozó FFT-algoritmust szemlélteti, kiemelve, hogy a végrehajtás közben az egyes résztranszformáltak elemei a *ram* tömb melyik rekeszében foglalnak helyet.



4-228. ábra. *A bemeneti mintaátrendezés következtében az FFT-algoritmus „helyben transzformál”*

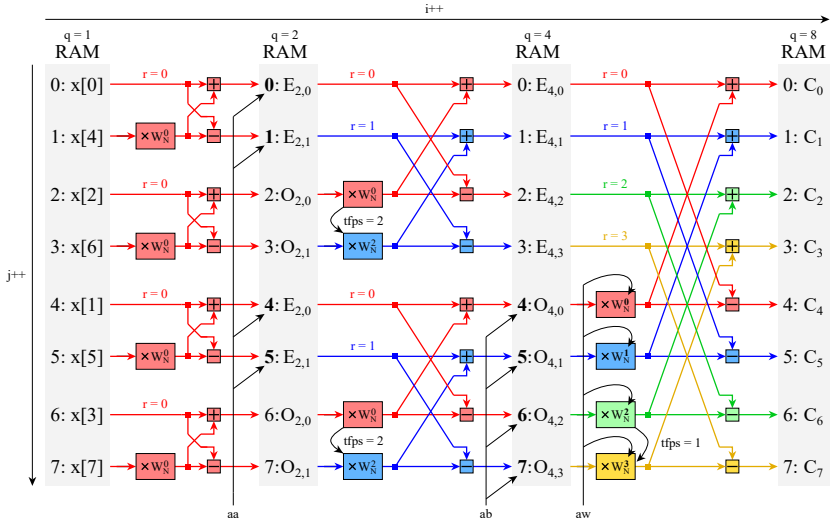
A mintaátrendezés következtében bármely lepkeműveletet is tekintjük a teljes algoritmus végrehajtása során, azt találjuk, hogy annak eredményei mindig a *ram* tömbnek ugyanazon rekeszeibe íródnak be, amely rekeszekből az operandusait kiolvastuk. Ez azt jelenti, hogy a résztranszformáltakat fizikailag mindvégig egyazon *N* kapacitású memóriában tárolhatjuk, mert bármelyik elemi művelet során csak olyan részeredményeket írunk felül, amelyekre a továbbiakban már nem lesz szükségünk, az algoritmus tehát a mintaátrendezés következtében „helyben transzformál”.

Az 5.6. Függelékben részletezetteknek megfelelően az FFT-algoritmus két, egymásba ágyazott ciklussal írható le. A belső ciklus a *q*-elemű résztranszformáltakból – *N/2* db lepkeművelet végrehajtásával – *2×q* elemű résztranszformáltakat állít elő, és ezt a műveletsort a külső ciklussal log_2N-szer megismételve kapjuk meg az *N*-elemű transzformált vektort, vagyis a végeredményt. A viselkedési modell tehát az FFT-algoritmusnak egy iteratív implementációja, amelyben a számítás folyamatának mindenkori állapotát hat segédváltozóval tartjuk számon (lásd 4-13. táblázat).

Változó neve	Változó szerepe
q	A bemeneti résztranszformáltak elemszáma. A bemeneti mintasort *N* db egyelemű résztranszformáltnak tekintjük, ezért *q* kezdeti értéke 1.
r	Futó változó a bemeneti résztranszformáltakon belüli iterációhoz.
aa	A lepkeműveletek *a* és *b* operandusainak indexei a résztranszformáltakat tároló memórián belül. Kezdeti értékek: *aa:* 0, *ab:* 1.
ab	
aw	A forgatási együttható valós részének cosinus-táblán belüli indexe, amelyből a képzetes rész indexe a 4-227. ábrán látható módon származtatható. Kezdeti értéke 0.
tfps	A forgatási együttható hatványkitevőjét a belső ciklus minden iterációjában növelni kell. E növelés mértéke attól függ, hogy a külső ciklus hányadik iterációjánál tartunk. Kezdeti értéke *N/2*.

4-13. táblázat. *Az FFT-algoritmus 4-225. ábrán látható megvalósításában használt segédváltozók*

A segédváltozók jelentése és változása az algoritmus végrehajtása során jól nyomon követhető a 8 elemű mintasoron végzett FFT-t szemléltető 4-229. ábrán.



4-229. ábra. *A segédváltozók szerepe az FFT-algoritmus végrehajtása során*

Az alábbiakban egy, az algoritmusnak ezt az iteratív kivitelét közvetlenül megvalósító IP-magot mutatunk be. Adatfeldolgozó rendszerről lévén szó áramkörünk RTL-modelljét ezúttal is egy vezérlőegység és egy műveletvégző egység alkotja.

4.5.5.2. Műveletvégző egység

A műveletvégző egység erőforrásainak meghatározásához ezúttal is a specifikációként szolgáló viselkedési modellből indulunk ki. A Python nyelven megfogalmazott függvény paramétereit és visszatérési értékét vizsgálva a következő rendszerkomponenseket azonosíthatjuk:

Be- és kimeneti FIFO-k ◆ Digitális jelfeldolgozó rendszerekben szokásos megoldás, hogy az adatfeldolgozási lánc elemei – azok átbocsátóképességének különbségeit kompenzálandó – FIFO-tárolókon keresztül kapják bemeneti adataikat, és eredményeiket is FIFO-tárolókba töltik be.[177] Az FFT IP-mag esetén is ezt a megoldást alkalmazzuk; az algoritmus bemenetét képező komplex mintasor elemeit a felhasználónak egy *N* bejegyzés tárolására képes FIFO-ba kell betöltenie, a transzformált vektort pedig egy ugyanilyen FIFO-ból kell kiolvasnia. Ahogy az összes újrafelhasználható IP esetén, úgy az FFT magnál is törekszünk arra, hogy az IP belső órajele könnyen függetleníthető legyen a beágyazó környezet órajelétől, ezért a be- és kimeneti FIFO-kat a 4.2.5. pontban bemutatott *dualclock_fifo_hswrp* modullal valósítjuk meg. Mivel az FFT mag komplex számokon végez műveleteket, ezért a FIFO-k adatbuszának méretét úgy állítjuk be, hogy egyetlen bejegyzés egy komplex szám valós és képzetes részét egyszerre képviselhesse; az *L* bit széles adatbusz felső *L/2* bitjét tekintjük a tárolt komplex szám valós, alsó *L/2* bitjét pedig ugyanazon komplex szám képzetes részének. *L* szintézis-paraméter, így a felhasználó az adott alkalmazás igényei szerint hangolhatja be a leendő áramkör pontosságát és erőforrásigényét.

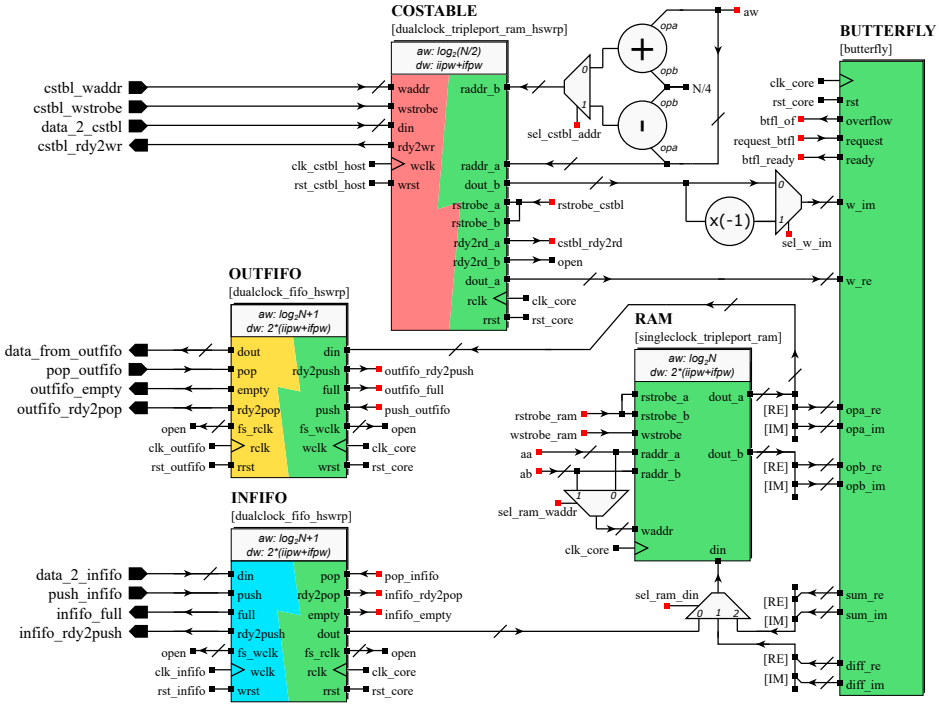
Cosinus-tábla ◆ Ahogy a viselkedési modellből is kiolvasható, a leendő áramkörnek a transzformáció végrehajtásához a cos és -sin függvények helyettesítési értékeire van szüksége. Ezeket az értékeket a felhasználónak az IP használata előtt (egyszer) be kell töltenie egy belső memóriamodulba, a cosinus-táblába. A cosinus-tábla betöltésére szolgáló interfész a be- és kimeneti FIFO-khoz hasonlóan egy külső interfész, amelynek használójáról feltételezzük, hogy külön órajeltartományban működik, ezért itt is olyan memóriamodult építünk be, amely beíró és kiolvasó interfészéhez külön órajelbemenetet biztosít. A 4.2.5. pontban tárgyalt *dualclock_tripleport_ram_hswrp* modul megfelelő választás; egyrészt biztosítja a két órajelbemenetet, másrészt egyszerre két tárolt értéket tudunk kiolvasni belőle, vagyis a forgatási együttható valós és képzetes része egyszerre érhető el úgy, hogy az adatbusz szélességét nem kell a be- és kimeneti FIFO-knál látottakhoz hasonló módon megduplázni, amivel jelentős tárkapacitást spórolhatunk.

RAM ◆ A Python függvény törzsét vizsgálva megállapíthatjuk, hogy a bemeneti (átrendezett) mintasor, majd – az algoritmus végrehajtásának előrehaladtával – a résztranszformáltak tárolására egy *N* db komplex szám tárolására képes belső memóriára is szükségünk lesz (4-225. ábra: 3. sor). A be- és kimeneti FIFO-khoz hasonlóan ebben a RAM modulban is egyetlen, *L* bitszélességű bejegyzés képviseli egy komplex szám *L/2* biten ábrázolt valós és ugyancsak *L/2* biten ábrázolt képzetes részét. Mivel az algoritmus minden elemi lépésében (a lepkeműveletben) két komplex operandusra és a forgatási együtthatóra van szükségünk, célszerű a 4.2.1. pontban bemutatott *singleclock_tripleport_ram* modult használni, így a két komplex operandust egyszerre tudjuk kiolvasni a memóriából.

[177] Lásd még 4.5.6. pont: FIR-szűrő, 4.5.7.1. pont: RC aluláteresztő szűrő FIFO-alapú órajel-szinkronizáló burkolója.

Segédváltozók ◆ A Python függvény 5–10. soraiban deklarált, a 4-13. táblázatban részletezett funkciójú segédváltozókat a vezérlőegység részének tekintjük, így a műveletvégző egységben nem jelennek meg külön regiszterekként. A viselkedési modell a lepkeművelet operandusait és az adott lépéshez tartozó forgatási együtthatót külön változókban tárolja el a lepkeműveletet megvalósító függvény meghívása előtt (4-225. ábra: 20. sor). Ezeknek az operandusoknak a tárolására a műveletvégző egységben – erőforrás-spórolás céljából – nem hozunk létre regisztereket. A viselkedési modellbeli *op_a_re*, *op_a_im*, *op_b_re* és *op_b_im* változókba betöltött értékek az RTL-modellben a két külön interfészen olvasható RAM modul kimeneti adatbuszain jelennek meg a címbemenetek *(aa, ab)* beállítása után következő első órajel-felfutóélnél. Így tulajdonképpen a viselkedési modell változóit a memóriamodul kimeneteivel azonosíthatjuk az RTL-modellben. Hasonló a helyzet a forgatási együttható valós és képzetes részét tároló *w_re* és *w_im* változókkal is. Megfelelő vezérlés esetén az RTL-modellben mindkét érték a cosinus-tábla adatkimenetein válik elérhetővé az *aw* segédváltozó beállítása és a kézfogásos memóriaolvasás végrehajtása után. Ha kihasználjuk a *dualclock_tripleport_ram_hswrp* modulnak mindkét kiolvasó interfészét, és a forgatási együttható képzetes részének címét egy egyszerű kombinációs hálózattal állítjuk elő az *aw* segédváltozóból – a 4-227. ábrán és a viselkedési modell 23–24. sorában definiált módon –, akkor a viselkedési modell *w_re* és *w_im* változóiba töltendő értékek az RTL-modellben egyazon órajel-felfutóélnél jelennek meg a cosinus-tábla adatkimenetein.

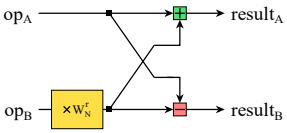
Az eddigi megfontolásaink alapján az FFT-algoritmust megvalósító IP-mag műveletvégző egységének RTL kapcsolási sémája a 4-230. ábrán látható módon képzelhető el.



4-230. ábra. *Az FFT IP-mag műveletvégző egysége*

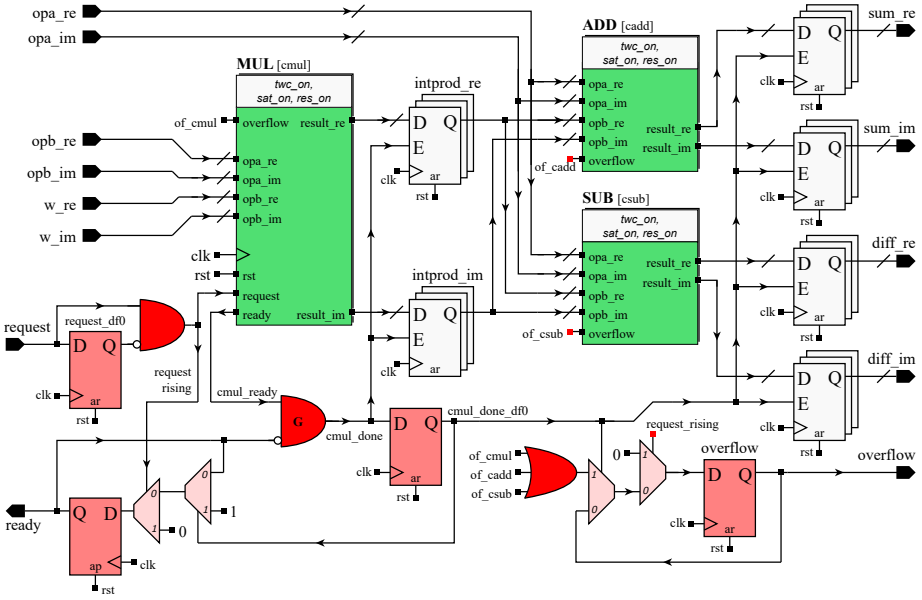
Butterfly-almodul ◆ A viselkedési modellben magát a lepkeműveletet egy függvényhívás „takarja el”. Ennek RTL megfelelője egy önálló almodul lesz, amelyet a műveletvégző egység belsejében példányosítunk. A lepkeművelet bemenetét három komplex operandus képezi, eredményül pedig két komplex számot állít elő a 4-231. ábrán látható módon (részletesen lásd 5.6. Függelék).

$$result_A = op_A + W_N^r \times op_B$$
$$result_B = op_A - W_N^r \times op_B$$



4-231. ábra. *A lepkeművelet definíciója*

A definíció szerint tehát egy olyan aritmetikai egységre van szükségünk, amely kiszámítja az op_B operandusnak és a forgatási együtthatónak a szorzatát, majd ezt a szorzatot az op_A operandushoz hozzáadva és abból kivonva állít elő két eredményt. Felhasználva a 4.5.1.1. pontban bemutatott, fixpontos komplex aritmetikai egységeket, a lepkeműveletet végrehajtó modul a 4-232. ábrán látható kapcsolási séma szerint építhető fel.

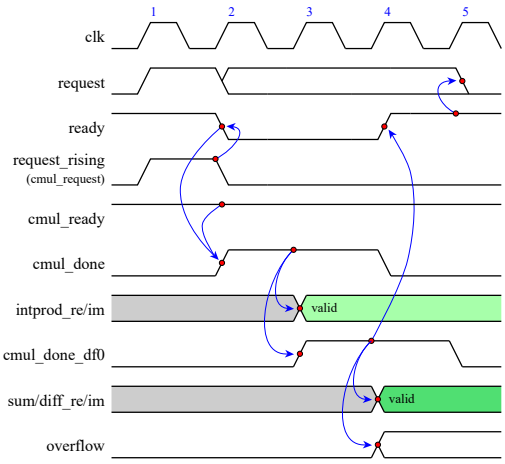


4-232. ábra. *A lepkeműveletet végrehajtó aritmetikai egység (butterfly) RTL kapcsolási sémája*

Mivel a szorzás és összeadás/kivonás műveleteket egymás után kell elvégezni, ezért a szorzatokat *(intprod_re/im)*, valamint a végeredményeket tároló regiszterek órajel-engedélyező bemeneteit megfelelően ütemezett vezérlőjelekkel kell meghajtani. Tovább bonyolítja a helyzetet, hogy a komplex szorzó modul két, eltérő erőforrásigényű és számítási teljesítményű belső architektúrával áll rendelkezésre, amelyek a szinkron kézfogásos interfész két, kismértékben különböző variánsát (nulla és nemnulla késleltetés) valósítják meg. A lepkeműveletet végző aritmetikai egységhez olyan

vezérlőlogikát érdemes konstruálnunk, amely a komplex szorzó modul mindkét változatával kompatibilis, ily módon a belső szorzó architektúra jellege (párhuzamos: 4-166. ábra vagy egyetlen szorzót tartalmazó, iteratív változat: 4-167. és 4-168. ábra) az FFT IP-magnak egy paraméterévé tehető. Így a felhasználónak lehetősége lesz választani a minimális erőforrás-felhasználású de lassabb, valamint a több szorzót igénylő, de gyorsabb változat közül.

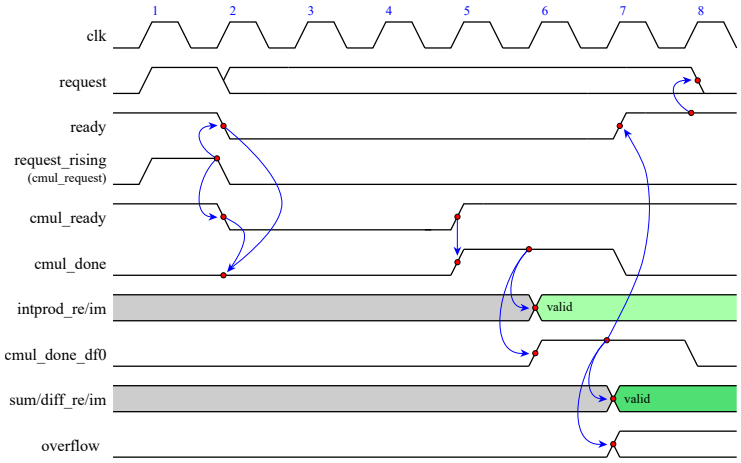
A 4-232. ábrán piros színnel jelölt vezérlőlogika működése a nulla késleltetésű komplex szorzó modul esetén a 4-233. ábrán követhető nyomon.



4-233. ábra. *A nulla késleltetésű komplex szorzó modul viselkedése a butterfly IP-magban*

Mivel a nulla késleltetésű esetben a komplex szorzó kombinációs hálózatként viselkedik, ezért – a szinkron kézfogásos interfész nulla késleltetésű verziójának 1-5. ábrán látható specifikációjának megfelelően – a *cmul_ready* jel mindvégig magas szintű marad. Így a *G* logikai kapu által előállított *cmul_done* jel a *ready* kimenet lefutóélének pillanatában (az órajel második felfutóélénél) magas szintű lesz. Mivel ezt a jelet használjuk a belső szorzatokat tároló *intprod_re/im* regiszterek órajel-engedélyező bemeneteinek meghajtására, ezért a komplex szorzó kimenetei az órajel harmadik felfutóélénél eltárolódnak. A belső szorzatokból a végeredményeket előállító komplex összeadó és kivonó áramkörök kombinációs hálózatok, ezért kimeneteiken a végeredmények az *intprod_re/im* regiszterek megváltozásával együtt előállnak. Az ezeket eltároló kimeneti regiszterek órajel-engedélyező bemenetei tehát a *cmul_done* jel egyetlen órajelciklussal késleltetett változatával, a *cmul_done_df0* jellel meghajthatóak. Ezt a vezérlőjelet egyúttal a *ready* kimenet egybe húzására, valamint a komplex aritmetikai modulok kimenetein esetleg előálló túlcsordulás-jelzéseket egy *OR* kapuval „összegyűjtő” *overflow* állapotjelző értékének beállítására is felhasználjuk.

A 4-234. ábrán látható hullámforma a nemnulla késleltetésű komplex szorzó modul esetét szemlélteti. Az egyetlen különbség a nulla késleltetésű esethez képest az, hogy ezúttal az órajel második felfutóélénél a komplex szorzó a *ready* kimenetét alacsony szintre húzza, emiatt ugyanebben a pillanatban a *cmul_done* jel nulla marad. A vezérlőáramkörben mindaddig nem történik semmi, amíg a komplex szorzó modul *ready* kimenete újra magas szintű nem lesz, vagyis amíg a komplex szorzatok elő nem állnak a szorzó modul belsejében végbemenő iterációk végén (vö.



4-234. ábra. *A nemnulla késleltetésű komplex szorzó modul viselkedése a butterfly IP-magban*

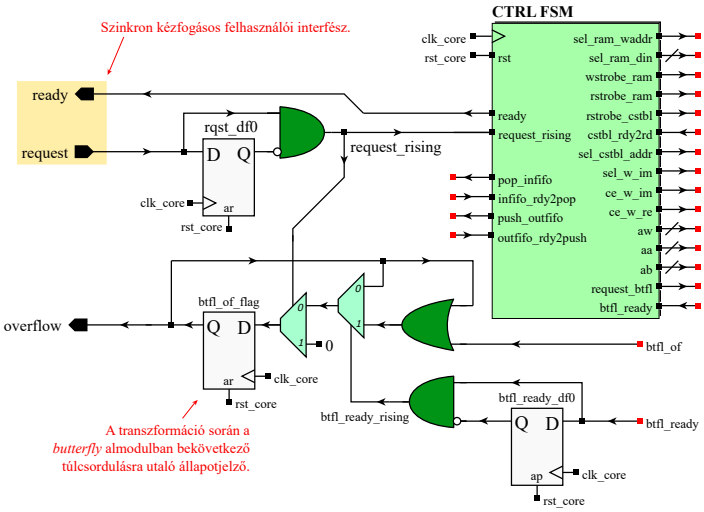
4-168. ábra). A szorzó *ready* jelének magas szintre állása után, az órajel 6-7-8. felfutóéleinél pontosan ugyanazok az események zajlanak le, amelyeket a 4-233. ábrán az órajel 3-4-5. felfutóéleinél már láthattunk.

A felhasználó – ebben az esetben maga az FFT IP-mag – a lepkeműveletet végrehajtó aritmetikai modul külső interfészén a fentiekből csupán annyit tapasztal, hogy az eredmények előállítása az iteratív komplex szorzót tartalmazó esetben három órajelciklussal több időt vesz igénybe. A kézfogásos interfész helyes használata esetén, tehát ha a számítás eredményének kiolvasását nem egy rögzített ciklusszám kivárása, hanem – a szükséges várakozás mértékétől függetlenül – mindig a *ready* jel felfutása után végezzük el, akkor a két eltérő késleltetésű és erőforrásigényű aritmetikai modul a beágyazó környezet bármiféle módosítása nélkül kicserélhető (vö. 2. fejezet bevezető: modularitás, lokalitás).

4.5.5.3. Vezérlőegység

A 4-230. ábra műveletvégző egysége tartalmazza mindazokat az aritmetikai és tároló elemeket, amelyek az FFT-algoritmus 4-225. ábrán látható viselkedési modell szerinti megvalósítását lehetővé teszik. Az adatfeldolgozó elemek megfelelően összekapcsolt hálózata azonban még nem elegendő; azok vezérlő bemeneteit pontosan megtervezett ütemezés szerint viselkedő vezérlőjelekkel kell meghajtanunk ahhoz, hogy az elvégzett műveletsor valóban az FFT-algoritmus legyen. E vezérlőjelek előállítását egy állapotgéppel oldjuk meg. Az FFT IP-mag vezérléséhez tartozik továbbá a szinkron kézfogásos interfészjeleket, valamint a számítás során a komplex aritmetikai egységekben esetleg előforduló túlcsordulásokra utaló állapotjelzőket kezelő logika is. A teljes vezérlőegység RTL kapcsolási sémáját a 4-235. ábra mutatja be.

A 2.2.3. pontban a *CPU* vagy a 4.5.3.3. pontban a lebegőpontos aritmetikai egység esetén olyan vezérlőszerkezeteket láthattunk, amelyeknek belsejében semmiféle aritmetikai művelet



4-235. ábra. *Az FFT IP-mag vezérlőegysége*

elvégzésére nem került sor.[178] Ezek az állapotgépek létrehozhatók egy, két vagy három VHDL *process*-t / Verilog *always*-blokkot tartalmazó kivitelben is,[179] és ezek a variánsok gépiesen át is alakíthatók egymásba, ami az erőforrás-optimalizáció során nagy előnyt jelenthet. A 4.5.2. pontban bemutatott, shift&add szorzást megvalósító IP-mag esetén pedig arra láthatunk példát, hogy az egy VHDL *process*-t / Verilog *always*-blokkot alkalmazó, ún. one-shot állapotgép-leírási módszerrel akár azt is megtehetjük, hogy az adatfeldolgozó rendszer műveletvégző egységét és vezérlőegységét egyetlen, tömör HDL-leírásba sűrítjük azáltal, hogy az állapotgép leírásába a műveletvégzés elemi lépéseit leíró aritmetikai értékadásokat is beágyazzuk (lásd pl. 4-174. ábra VHDL-modell: 19., 22., 23., 28. sorok).

Az FFT IP-mag esetén a 4-230. ábrán látható műveletvégző elemek vezérlőjeleit egy olyan one-shot típusú állapotgép szolgáltatja, amely az állapotgépek különféle megvalósítási lehetőségeinek egyfajta keveréke; a műveletvégző elemek vezérlőjeleit a *CPU* állapotgépéhez hasonlóan állítja elő, ugyanakkor az iteratív FFT-algoritmus segédváltozóit és az azokon végzett aritmetikai műveleteket az iteratív fixpontos szorzó HDL-modelljénél látottakhoz hasonlóan foglalja magában. Ennek a kevert jellegű állapotgépnek a vázlatát a 4-236. ábrán láthatjuk.

Az állapotgép vázlatában a 4. sorban aszinkron reset hatására alapállapotba állított jelek a viselkedési modell 5–10. soraiban deklarált változóknak felelnek meg. Mivel ezeknek a jeleknek a Verilog modellben – a one-shot állapotgép-leírási módszernek megfelelően – egy szinkron

[178] Az egyes vezérlési állapotokban elvégzett feltételvizsgálatokat áramkörileg komparátorok valósítják meg, amelyeket tekinthetünk aritmetikai elemeknek, ilyen értelemben tehát a fenti kijelentés nem teljesen pontos. Ugyanakkor ezek az összehasonlító áramkörök létrehozhatóak a műveletvégző egységben is, ebben az esetben azonban kimeneteiket vissza kell csatolnunk a vezérlő állapotgépbe, hogy az említett feltételvizsgálatokban hivatkozhassunk rájuk. Ez a HDL-modell áttekinthetőségét eléggé lerontja, úgyhogy ezt a módszert általában kerüljük. Kivételt jelentenek azok az esetek, amikor az adott komparátor kimenetét a műveletvégző egység egyéb elemei is felhasználják.

[179] Lásd még DRMSz: 4.5.5. alfejezet.

Verilog – vezérlőegység

```
1  always @ ( posedge clk_core, posedge rst_core ) begin
2  | if ( rst_core == 1 ) begin
3  | | state <= idle; ready <= 1;
4  | | q <= 1; r <= 0; aa <= 0; ab <= 1; aw <= 0; tfps <= N/2; i <= 0; j <= 0;  // internal registers
5  | | pop_infifo <= 0; rstrobe_cstbl <= 0; rstrobe_ram <= 0; wstrobe_ram <= 0; // datapath control signals
6  | | push_outfifo <= 0; request_btfl <= 0;
7  | | sel_cstbl_addr <= 0; sel_ram_din <= 0; sel_ram_waddr <= 0; sel_w_im <= 0;
8  | end else begin
9  | |
10 | | // default values of the pulse-like control signals...
11 | | pop_infifo <= 0; rstrobe_cstbl <= 0; rstrobe_ram <= 0; wstrobe_ram <= 0;
12 | | push_outfifo <= 0; request_btfl <= 0;
13 | |
14 | | case ( state )
15 | | |
16 | | | idle:    [...]
17 | | |
18 | | | [...]
19 | | |
20 | | | default: ;
21 | | |
22 | | endcase
23 | |
24 | end
25 end
```

4-236. ábra. *Az FFT IP-mag vezérlő állapotgépének vázlata*

always-blokkban adunk értéket, ezért áramkörileg egy-egy több-bites regiszterként valósulnak majd meg a szintézis során. Ezek a regiszterek tulajdonképpen az állapotgép kimeneti buffereinek tekinthetők, az ezekre vonatkozó aritmetikai műveleteket megvalósító kombinációs hálózatok pedig az állapotgép kimeneti logikájának részét képezik.[180]

Az algoritmus végrehajtása a bemeneti mintasor átrendezésével kezdődik a viselkedési modell 12–14. soraiban definiált ciklus segítségével. Az RTL-modellben az ehhez a folyamathoz tartozó vezérlési állapotok feladata, hogy kiolvassák a bemeneti FIFO-ba természetes sorrendben betöltött mintákat, majd – meghatározva az éppen kiolvasott minta bit-fordított címét – beírják azokat a belső RAM megfelelő rekeszébe. A vezérlési állapotok Verilog-leírását és a folyamat időzítését a 4-237. ábra szemlélteti.

A vezérlőegység – miután meggyőződött róla, hogy a kiolvasandó minta már rendelkezésre áll (4-237. ábra Verilog modell: 12. sor) – végrehajt egy *pop* műveletet a bemeneti FIFO-n. A *pop_infifo* jelen előállított pozitív impulzus hatására a bemeneti FIFO *rdy2pop* jele az órajel harmadik felfutóélénél vált nullába – jelezve, hogy a FIFO modul megkezdte a kiolvasás műveletet – ezért a *pop* művelet befejezését jelző felfutóélt legkorábban a *resort_3* állapotban várhatjuk.[181] Az *N* db

[180] Vö. DRMSz: 4-90. ábra: Iteratív akkumulátor összevont állapotgépes modellje.

[181] A bemeneti FIFO megvalósítására a *dualclock_fifo_hswrp* IP-t használjuk, amelynek késleltetése a 4-20. ábrán bemutatott kézfogásos logika nyomán paraméterként beállítható. Ha a késleltetést nullára állítjuk, akkor a *rdy2pop* jelen nem jelenik meg a 4-237. ábrán az órajel harmadik felfutóélénél látható lefutóél, ennek ellenére a fenti állapotgép helyesen működik. A modularitás alapelve megkívánja, hogy az együttműködő alrendszerek figyelembe vegyék egymás különféle működési módjait, és azokhoz megfelelően alkalmazkodjanak. Ha ez az alkalmazkodás túl költséges – akár fejlesztési idő, akár az erőforrás-felhasználás megnövekedése miatt – akkor célszerű gondoskodni róla, hogy a felhasználónak ne legyen módja egymással nem kompatibilis üzemmódba állítani a két együttműködő alrendszert. A fenti példában megtehetjük, hogy a bemeneti FIFO késleltetés paraméterét nem „ajánljuk ki" a felhasználónak az FFT IP toplevel moduljának paraméterlistáján, hanem azt fixen nullára állítjuk. Ebben az esetben az állapotgépet nem kell felkészíteni a nullánál nagyobb késleltetésű működésre, ami kevesebb vezérlési állapotot és gyorsabb működést jelent. Ennek az egyszerűsítésnek az ára az, hogy a felhasználónak nem lesz módja az FFT IP számítási teljesítményének és a FIFO késleltetésének összefüggését vizsgálni. Az FFT IP tervezőjeként pedig a saját dolgunkat is megnehezítjük azáltal, hogy így nagyobb késleltetésű FIFO-t csak az állapotgép módosításával tudunk később beépíteni a modellünkbe.

Python – viselkedési modell

```
12    for i in range(N):
13      reva = bit_reverse(i, math.log2(N))
14      ram_re[i] = x_re[reva]; ram_im[i] = x_im[reva];
```

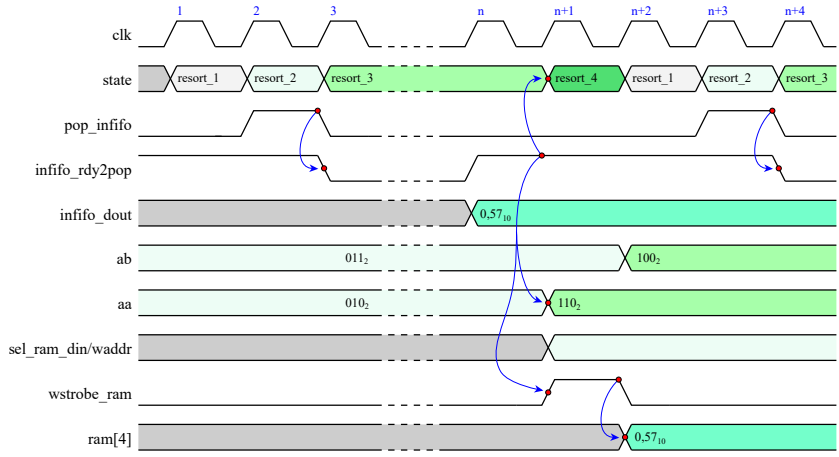
Verilog – vezérlőegység

```
1  idle: begin
2  |  q <= 1; r <= 0;
3  |  aa <= 0; ab <= 0; aw <= 0;
4  |  tfps <= N/2; i <= 0; j <= 0;
5  |  if ( request_rising == 1 ) begin
6  |  |  ready <= 0;
7  |  |  state <= resort_1;
8  |  end
9  end
10
11 resort_1: begin
12 |  if ( infifo_empty == 0 ) begin
13 |  |  pop_infifo <= 1; state <= resort_2;
14 |  end
15 end
16
17 resort_2: state <= resort_3;
18
```

Verilog – vezérlőegység (folyt.)

```
19 resort_3: begin
20 |  if ( infifo_rdy2pop == 1 ) begin
21 |  |  aa <= reverse_bits(ab);
22 |  |  sel_ram_din <= 0; sel_ram_waddr <= 0;
23 |  |  wstrobe_ram <= 1; state <= resort_4;
24 |  end
25 end
26
27 resort_4: begin
28 |  if ( ab < N-1 ) begin
29 |  |  ab <= ab + 1; state <= resort_1;
30 |  end else begin
31 |  |  aa <= 0; ab <= 1; state <= calc_fft_1;
32 |  end
33 end
```

Hullámforma



4-237. ábra. *A bemeneti FIFO kiolvasását és a mintasor elemeinek átrendezését végző vezérlési állapotok*

minta bemeneti FIFO-ból való kiolvasása és RAM-ba írása egy *N* iterációt igénylő folyamat, amit az *aa* és *ab* regiszterekkel követünk nyomon; a viselkedési modell *i* ciklusváltozójának szerepét az RTL-modellben az *ab* regiszter tölti be, ezen végezzük a *0...(N-1)* számlálást, az *aa* regiszter szerepe pedig a RAM-ba íráshoz szükséges írási cím, vagyis az *ab* ciklusváltozó bit-fordított értékének eltárolása lesz.

Miután a bemeneti minta előállt a FIFO kimenetén (4-237. ábra Verilog modell: 20. sor), elvégezzük az e mintának megfelelő index *(ab)* bitsorrendjének megfordítását a 4-238. ábrán látható függvénnyel.[182] A RAM-ba íráshoz szükséges cím *(aa)* meghatározása, valamint a RAM modul

[182] Az állapotgép leírásába ágyazott függvényhívás áramkörileg a fixpontos aritmetikai eljárásnál látottakhoz hasonlóan (vö. 4-163. ábra) egy kombinációs hálózatként valósul meg, ebben az esetben azonban ez a „kombinációs hálózat” valójában nem tartalmaz logikai kapukat, hiszen csak a megfelelő vezetékek szimbolikus átnevezéséről van szó.

Verilog

```
function [$clog2(N)-1:0] reverse_bits;

  input [$clog2(N)-1:0] address_in;

  begin
    for ( int i = 0; i < $clog2(N); i = i + 1 ) begin
      reverse_bits[i] = address_in[$clog2(N)-1-i];
    end
  end

endfunction
```

4-238. ábra. *A bitsorrendet megfordító Verilog függvény*

adat- és címbemenetét meghajtó multiplexerek kiválasztójelének beállítása után (4-237. ábra Verilog modell: 22–23. sor) végrehajtjuk a RAM-írás műveletet az órajel (n+2)-edik felfutóélénél előállított pozitív impulzus segítségével. Figyeljük meg, hogy a RAM-ba írás a FIFO olvasásával ellentétben csak egyetlen órajelciklust igényel, hiszen a belső memória megvalósítására a kézfogás nélküli *singleclock_tripleport_ram* IP-t használtuk. A RAM-ba írással egyidőben, a *resort_4* állapotban megvizsgáljuk az *ab* ciklusváltozó értékét. Ha még nem olvastunk be *N* db mintát, akkor *ab* értékét növeljük, és a mintabeolvasási lépéssort a *resort_1* vezérlési állapottól újrakezdjük, ellenkező esetben az *aa* és *ab* regisztereket a viselkedési modellben meghatározott kiindulási értékekre (4-225. ábra: 7–8. sor) állítva a műveletvégzést a *calc_fft_1* állapotban folytatjuk.

Az FFT-algoritmus magját két, egymásba ágyazott ciklus képezi, amelyeket a viselkedési modellben az *i* és *j* ciklusváltozókkal azonosíthatunk. Ezeknek a ciklusváltozóknak az RTL-modellbeli megfelelőik az ugyancsak az *i* és *j* azonosítókkal hivatkozott regiszterek lesznek (4-236. ábra: 4. sor). A két ciklust megvalósító vezérlési állapotok működését a 4-239., 4-240. és 4-241. ábrákon követhetjük. Az RTL-modellben mindkét ciklust „hátultesztelő" módon fogalmaztuk meg úgy, hogy a *calc_fft_1...8* vezérlési állapotok a belső ciklus műveleteit (4-225. ábra Python modell: 20–32. sor), a *calc_fft_9...10* állapotok pedig a belső ciklus végrehajtása utáni, de még a külső ciklus törzsének részét képező műveleteket (4-225. ábra Python modell: 34–35. sor) valósítják meg. A belső és a külső ciklus kilépési feltételeit tehát rendre a *calc_fft_8* és a *calc_fft_10* állapotokban vizsgáljuk.

A belső ciklus törzsében a viselkedési modell szerint a következő lépésekre kerül sor:

1. Az *op_a_re*, *op_a_im*, *op_b_re* és *op_b_im* változókba betöltjük a végrehajtandó lepkeművelet operandusait (4-239. ábra Python modell: 20. sor).
2. A *w_re* és *w_im* változókba betöltjük a megfelelő forgatási együtthatót (4-239. ábra Python modell: 22–24. sor).
3. Az előkészített operandusokon végrehajtjuk a lepkeműveletet, és annak eredményét a *bfres_a* és *bfres_b* nevű ideiglenes változókba töltjük (4-239. ábra Python modell: 26. sor).
4. A *bfres_a* és *bfres_b* ideiglenes változók értékét átírjuk a *ram_re/im* tömb *aa* és *ab* változók által címzett rekeszeibe (4-240. ábra Python modell: 28. sor).
5. A számítás állapotát számontartó segédváltozókat frissítjük (4-240. ábra Python modell: 30–32. sor).

A 4-239. ábra e műveletsor első három lépését, vagyis a lepkeművelet aktuális operandusainak előkészítését mutatja be.

Python – viselkedési modell

```
16   for i in range(int(math.log2(N))):
17   |
18   | for j in range(int(N/2)):
19   | |
20   | | op_a_re = ram_re[aa]; op_a_im = ram_im[aa]; op_b_re = ram_re[ab]; op_b_im = ram_im[ab]
21   | |
22   | | w_re = costable[aw]
23   | | if ( aw < int(N/4) ): w_im = costable[aw+int(N/4)]
24   | | else: w_im = -1*costable[aw-int(N/4)]
25   | |
26   | | (bfres_a, bfres_b) = butterfly((op_a_re, op_a_im), (op_b_re, op_b_im), (w_re, w_im))
27   | |
..   | | [...]
```

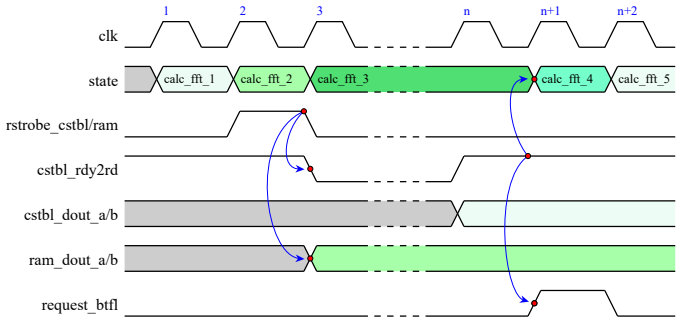
Verilog – vezérlőegység

```
35 calc_fft_1: begin
36   if ( aw < N/4 ) begin
37     sel_cstbl_addr <= 0; sel_w_im <= 0;
38   end else begin
39     sel_cstbl_addr <= 1; sel_w_im <= 1;
40   end
41   rstrobe_cstbl <= 1; rstrobe_ram <= 1;
42   state <= calc_fft_2;
43 end
44
```

Verilog – vezérlőegység (folyt.)

```
45 calc_fft_2: state <= calc_fft_3;
46
47 calc_fft_3: if ( cstbl_rdy2rd == 1 ) begin
48   request_btfl <= 1; state <= calc_fft_4;
49 end
50
51 calc_fft_4: state <= calc_fft_5;
52
53 calc_fft_5: [...]
54
```

Hullámforma



4-239. ábra. *Az FFT-algoritmus egymásba ágyazott ciklusait megvalósító vezérlési állapotok (1)*

A viselkedési modell – a formális megfogalmazására használt nyelv jellegéből és a feladat egyszerűségéből adódóan – nem fejezi ki az egyes alapműveletek közötti lehetséges párhuzamosításokat, azonban a leendő áramkörünk számítási teljesítményének maximalizálása érdekében az RTL-modellben célszerű az egymással párhuzamosan végrehajtható műveleteket egyidőben vagy legalább átlapoltan elvégezni.[183] Ezért a viselkedési modell 20–24. soraiban megfogalmazott memóriaolvasásokat egyidőben kezdeményezzük a *calc_fft_1* állapotban. A két megszólított memóriamodul a belső RAM, amelyből a lepkeművelet operandusait olvassuk ki, illetve a cosinus-tábla, amely a forgatási együtthatókat tartalmazza. E két memóriamodul eltérő viselkedésű; míg a RAM kimenetén a *rstrobe* jelen előállított impulzus lefutóélével egyidőben megjelenik

[183] A Python nyelv – és a programozási nyelvek általában – alkalmasak ugyan párhuzamosság kifejezésére (pl. szálak létrehozásával), ezek használata azonban a viselkedési modellt e konkrét példa esetén aránytalan mértékben megbonyolítaná, ezért ezt a megoldást itt nem használjuk. Összetettebb funkciók esetén azonban akár ilyen módszerek bevetése is célszerű lehet, ha már a viselkedési modellben expliciten érzékeltetni szeretnénk az egyes részműveletek egymáshoz képesti ütemezésének finom részleteit.

Python – viselkedési modell

```
16  for i in range(int(math.log2(N))):
17  |
18  | for j in range(int(N/2)):
..  | |
..  | | [...]
..  | |
26  | | (bfres_a, bfres_b) = butterfly((op_a_re, op_a_im), (op_b_re, op_b_im), (w_re, w_im))
27  | |
28  | | ram_re[aa] = bfres_a[0]; ram_im[aa] = bfres_a[1]; ram_re[ab] = bfres_b[0]; ram_im[ab] = bfres_b[1]
29  | |
30  | | if ( r == q-1 ): r = 0; aa = aa + q + 1; aw = 0
31  | | else: r = r + 1; aa = aa + 1; aw = aw + tfps
32  | | ab = aa + q
..  | |
..  | [...]
```

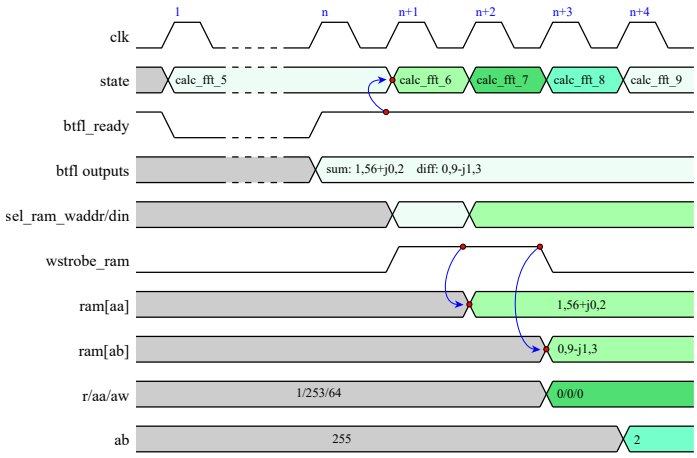
Verilog – vezérlőegység

```
53  calc_fft_5: if ( btfl_ready == 1 ) begin
54    sel_ram_waddr <= 0; sel_ram_din <= 2;
55    wstrobe_ram <= 1; state <= calc_fft_6;
56  end
57
58  calc_fft_6: begin
59   sel_ram_waddr <= 1; sel_ram_din <= 1;
60   wstrobe_ram <= 1; state <= calc_fft_7;
61  end
62
```

Verilog – vezérlőegység (folyt.)

```
63 calc_fft_7: begin
64 | if ( r == q-1 ) begin
65 |   r <= 0; aa <= aa + q + 1; aw <= 0;
66 | end else begin
67 |   r <= r + 1; aa <= aa + 1; aw <= aw + tfps;
68 | end
69 | state <= calc_fft_8;
70 end
71
72 calc_fft_8: begin
73 | ab <= aa + q;
74 | if ( j < (N/2)-1 ) begin
75 |   j <= j + 1; state <= calc_fft_1;
76 | end else begin
77 |   j <= 0; state <= calc_fft_9;
78 | end
79 end
80
```

Hullámforma



4-240. ábra. *Az FFT-algoritmus egymásba ágyazott ciklusait megvalósító vezérlési állapotok (2)*

a kiolvasott adat (4-239. ábra: 3. órajel-felfutóél), addig a cosinus-tábla kézfogásos interfésze miatt a forgatási együtthatóra a vezérlő állapotgépnek várnia kell a *cstbl_rdy2rd* jel felfutóélére (4-239. ábra: n-edik órajel-felfutóél). A műveletvégző egység részleteinek kidolgozásánál meghozott döntés nyomán a kiolvasott bejegyzések tárolására nem állnak rendelkezésre külön regiszterek, hanem

a memóriamodulok adatkimenetei közvetlenül a lepkeműveletet végrehajtó almodul bemenetire vannak kapcsolva. Így a 4-239. ábra szerinti (n+1)-edik órajel-felfutóélnél a viselkedési modell 20–24. soraiban definiált előkészítő lépések lezárultak, tehát kezdeményezhetjük a lepkeművelet végrehajtását a *request_btfl* jelen előállított pozitív impulzussal. A lepkeművelet végrehajtásának kezdeményezését és az eredmények RAM-ba való visszaírását végző vezérlési állapotokat a 4-240. ábra mutatja be.

A lepkeműveletet végrehajtó aritmetikai egység egy több órajelciklust igénylő, összetett számítási feladatot végez el, ezért a *calc_fft_5* állapotban kénytelenek vagyunk kivárni ennek a végét. Az eredmények tárolására a viselkedési modellben létrehozott *bfres_a/b* ideiglenes változóknak az RTL-modellben szintén nincs külön regisztermegfelelőjük, a lepkeműveletet végző egység kimenetei egy multiplexeren keresztül a RAM modul adatbemeneteire vannak kötve. Ez azt jelenti, hogy a viselkedési modell 26. sorában leírt művelet az RTL-modellben a 4-240. ábra szerinti n-edik órajel-felfutóélnél már elvégzettnek tekinthető, tehát az (n+1)-edik órajel-felfutóélnél megkezdhető az eredmények RAM-ba való visszaírása. A RAM modulnak csupán egy beíró interfésze van, a lepkeművelet pedig két eredményt állít elő, így két beírást kell végrehajtanunk. Figyeljük meg, hogy a RAM *wstrobe* jelét a *calc_fft_5* és *calc_fft_6* állapotokban egyaránt magas szintre állítjuk, miközben a memóriamodul cím- és adatbemenetét meghajtó multiplexerek kiválasztójelét átbillentjük. Így – a *singleclock_tripleport_ram* modul folyamatos írás üzemmódját (vö. 4-15. ábra) kihasználva – a két szükséges írásművelet az órajel (n+2)-edik és (n+3)-adik felfutóélénél, közvetlenül egymás után végbemegy. A második visszaírással egyidőben – a *calc_fft_7* állapotban – az *r*, *aa* és *aw* segédváltozókat képviselő regisztereknek a viselkedési modell 30–31. soraiban definiált frissítésére is sor kerül. Mivel az *r*, *aa* és *aw* regiszterek frissítése egymástól független – egyik regiszter sem szerepel egy másik regiszterre vonatkozó értékadás jobbértékeként –, ezért alkalmazhatunk párhuzamosítást. Azonban az *ab* regiszter új értéke a már felülírt *aa* regiszter értékétől függ, ezért annak frissítését csak egy órajelciklus késleltetéssel, az órajel (n+4)-edik felfutóélénél végezhetjük el (4-240. ábra Verilog modell: 73. sor). A belső ciklus kilépési feltételének vizsgálatára a *calc_fft_8* állapotban kerül sor. Ha a belső ciklus még nem ért véget, akkor a *j* regiszter értékének inkrementálása után a belső ciklustörzs újabb elvégzését kezdeményezzük a *calc_fft_1* vezérlési állapotra való ugrással, ellenkező esetben a *calc_fft_9* állapotban folytatjuk a végrehajtást.

A belső ciklus végrehajtása után a *q*, *aa*, *aw* és *tfps* segédváltozókat képviselő regisztereket frissítjük, ezúttal a viselkedési modell 34. sorában definiáltak szerint (lásd 4-241. ábra). Az *ab* regiszter frissítésére a *q* regiszter értékétől való függés miatt ezúttal is csak egy órajelciklussal később, a *calc_fft_10* állapotban kerülhet sor, a külső ciklus kilépési feltételének vizsgálatával egyidőben. Ha a külső ciklus még nem ért véget, akkor ismét a két egymásba ágyazott ciklus közös belépési pontját képviselő *calc_fft_1* kezdőállapotba ugrunk, ellenkező esetben megkezdhető az immár a bemeneti mintasor diszkrét Fourier-transzformáltját képviselő memóriatartalom kimeneti FIFO-ba írása. Ezt a folyamatot, amely tulajdonképpen a viselkedési modell utolsó sorának RTL-modellbeli megfelelőjeként is felfogható, a 4-242. ábrán látható vezérlési állapotok vezénylik le.

A RAM kiolvasása és a kiolvasott értékek kimeneti FIFO-ba írása ismét egy *N* iterációt igénylő folyamat, ahol a ciklusváltozó szerepét az *aa* regiszter tölti be. Az *N* db bejegyzés kiírásának végén a vezérlő állapotgép a *ready* kimenetét magas szintre húzza, jelezve, hogy a transzformáció véget ért, a kimeneti FIFO-ból a transzformált vektor kiolvasható.

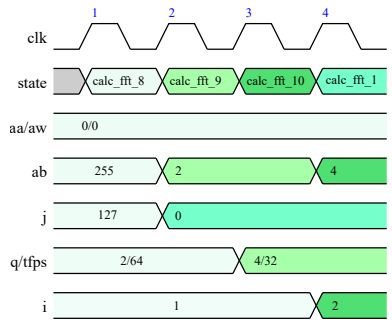
Python – viselkedési modell

```
16   for i in range(int(math.log2(N))):
17   |
18   |  for j in range(int(N/2)):
..   |  |
..   |  |  [...]
..   |  |
32   |  |  ab = aa + q
33   |
34   |  q = q * 2; aa = 0; aw = 0; tfps = int(tfps / 2);
35   |  ab = q
36
37   [...]
```

Verilog – vezérlőegység

```
81 calc_fft_9: begin
82   q <= q << 1; aa <= 0; aw <= 0; tfps <= tfps >>  1;
83   state <= calc_fft_10;
84 end
85
86 calc_fft_10: begin
87 | ab <= q;
88 | if ( i < $clog2(N)-1 ) begin
89 |   i <= i + 1;
90 |   state <= calc_fft_1;
91 | end else begin
92 |   i <= 0;
93 |   aa <= 0; rstrobe_ram <= 1;
94 |   state <= push_outfifo_1;
95 | end
96 end
97
```

Hullámforma



4-241. ábra. *Az FFT-algoritmus egymásba ágyazott ciklusait megvalósító vezérlési állapotok (3)*

Python – viselkedési modell

```
36   [...]
37   return (ram_re, ram_im)
```

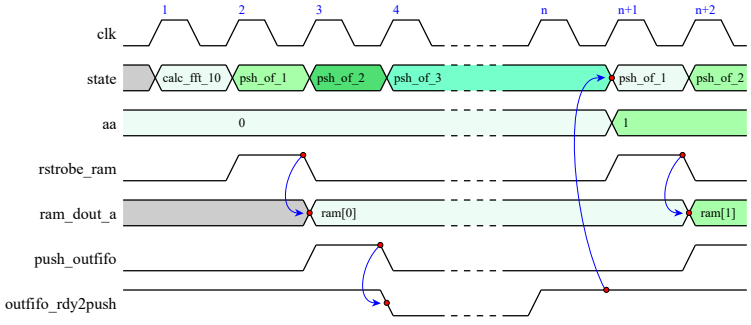
Verilog – vezérlőegység

```
98   push_outfifo_1: begin
99   | if ( outfifo_full == 0 ) begin
100  |   push_outfifo <= 1; state <= push_outfifo_2;
101  | end
102  end
103
104  push_outfifo_2: state <= push_outfifo_3;
105
```

Verilog – vezérlőegység (folyt.)

```
106  push_outfifo_3: if ( outfifo_rdy2push == 1 )
107  | if ( aa < N-1 ) begin
108  |   aa <= aa + 1; rstrobe_ram <= 1;
109  |   state <= push_outfifo_1;
110  | end else begin
111  |   aa <= 0; ready <= 1; state <= idle;
112  | end
113
```

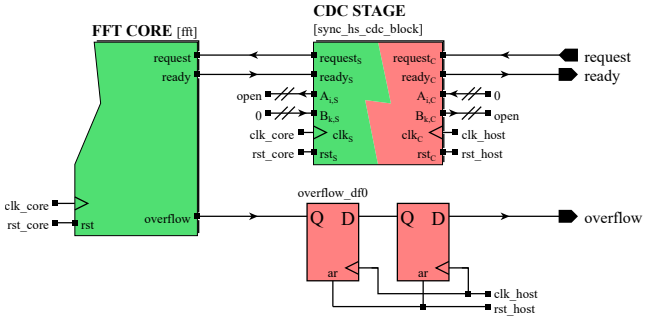
Hullámforma



4-242. ábra. *A transzformált vektor kimeneti FIFO-ba írását végző vezérlési állapotok*

4.5.5.4. Órajel-szinkronizálás

A 4-230. ábrán jól látható, hogy az FFT modul műveletvégző egységében elhelyezett cosinus-tábla, valamint a be- és kimeneti FIFO-k önmagukban is gondoskodnak róla, hogy az IP egy saját, környezetéhez képest aszinkron órajeltartományban működhessen; mindhárom tárolófunkciót egy-egy dual-clock tulajdonságú IP-maggal valósítottuk meg. A vezérlőegység esetén ugyan van némi teendőnk, de az egységesített interfészeknek köszönhetően a helyzet itt is egyszerű. Mivel az FFT IP-mag *request-ready* jelpárosa a szinkron kézfogásos interfész-specifikáció szerinti működésű, ezért beépíthetjük a 2-2. ábrán látható általános vektorszinkronizáló fokozatot úgy, hogy annak adatvektorait nem használjuk ki (lásd 4-243. ábra).



4-243. ábra. *Az FFT modul vezérlőegységének órajel-szinkronizáló fokozatai*

Az FFT modul *overflow* kimenete egy egyszerű statikus jel, így annak szinkronizálása a felhasználó órajeltartományához egy egyszerű double-flop fokozattal megoldható.

4.5.6. FIR-szűrő

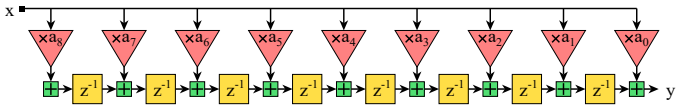
Áttanulmányozásra javasolt alfejezetek:

***4.2**. On-chip adattárolás*
***4.5.1**. Aritmetika és jelfeldolgozás – Fixpontos aritmetikai egység*

Ebben a pontban a digitális jelfeldolgozás egyik legalapvetőbb funkcionális egységével, az ún. véges impulzusválaszú (FIR, **F**inite **I**mpulse **R**esponse) szűrővel foglalkozunk. A FIR-szűrő egy jel diszkrét mintáiból álló jelfolyamon végez transzformációt oly módon, hogy minden új bemeneti minta *(x[n])* feldolgozásakor az aktuális kimeneti minta az

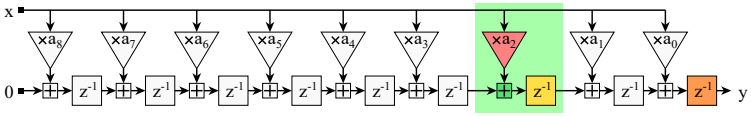
$$y[n] = a_0x[n] + a_1x[n-1] + \ldots + a_nx[n-N] = \sum_{i=0}^{N} a_ix[n-i] \qquad (4.5\text{-}1)$$

összefüggés szerint az aktuális és az előző *N* lépésben beérkezett bemeneti minta súlyozott összegeként áll elő. Az *N* állandót a szűrő fokszámának, az a_i súlyokat pedig a szűrő együtthatóinak nevezzük. A 4-244. ábra egy nyolcadfokú FIR-szűrőnek egy lehetséges megvalósítását mutatja be.



4-244. ábra. *A FIR-szűrő felépítése (1)*

Az ábrán szándékosan nem az RTL kapcsolási sémáknál használt összeadás és szorzás szimbólumokat használjuk, érzékeltetve, hogy ez a reprezentáció csupán vázlata a következőkben részletesen megtervezett RTL-modellnek. Az ábrán a z^{-1} szimbólumok a diszkrét jelfolyam egy ütemmel való késleltetését fejezik ki. A 4-245. ábra azt hangsúlyozza, hogy a FIR-szűrő tulajdonképpen egy azonos felépítésű és funkciójú, a fokszámtól függő mennyiségben példányosított elemi műveletvégző egység sorbakapcsolásával építhető fel.



4-245. ábra. *A FIR-szűrő felépítése (2)*[184]

Ez az alapelem egy összetett aritmetikai egység – amely két operandus szorzatához ad hozzá egy harmadik operandust –, a digitális szűrők legelemibb összetevője, az ún. szorzó-akkumulátor (**M**ultiply & **Ac**cumulate, MAC). Olyannyira alapvető, hogy az FPGA áramkörökben a blokk-RAM-okhoz hasonló módon beépítve megtalálhatjuk azokat az architekturális komponenseket (a DSP-blokkokat), amelyek kifejezetten ennek az áramköri részletnek a minél hatékonyabb megvalósítását célozzák.[185] Természetesen a DSP-blokkok a különféle gyártók eszközeiben nem teljesen egyformák, ezért technológiafüggetlen IP-magok fejlesztésénél – a blokk-RAM-okhoz hasonlóan – közvetlenül nem példányosíthatók. Ehelyett olyan újrafelhasználható szorzó-akkumulátor modellt készítünk, amely a lehető legáltalánosabb formában írja le a szükséges funkciókat, így bármely gyártó eszközeit használjuk is a fizikai megvalósításra, biztosak lehetünk benne, hogy a szintézis-szoftver képes lesz azokat leképezni a konkrét eszköz DSP-blokkjaira, így számítási teljesítmény szempontjából is igen kedvező megoldást kapunk. Egy ilyen technológiafüggetlen szorzó-akkumulátor RTL-modelljét mutatja be a 4-246. ábra.

A szorzó-akkumulátorban felhasználjuk a 4.5.1. pontban tárgyalt fixpontos aritmetikai egységet *(fxp_arith_unit)*, ami a szorzás és az összeadás funkciókat áramkörileg egyaránt egy-egy kombinációs hálózattal valósítja meg. Szigorúan funkcionális szempontból ugyan nem indokolt, de az e leírásból szintetizált áramkörök időzítési tulajdonságainak javítása – a minél nagyobb órajel-frekvencia elérése – érdekében a szorzó és az összeadó közé egy pipeline regisztert *(product)* iktatunk be. Ez azt jelenti, hogy egy operandus-hármasnak a bemenetre kapcsolása után legalább két órajelciklus szükséges az *acc* kimenet előállításához, de az *ena_product* és *ena_acc* bemenetek magas szinten tartása esetén – és a bemenetek órajel-periódusonkénti változtatásával – minden órajelciklusban

[184] A struktúra kimenetére az ismétlődő alapelem részeként beiktattunk egy újabb késleltető elemet, ami a kimeneti jelfolyam minden elemét egy ciklussal késlelteti anélkül, hogy a kimeneti értékeken változtatna, így funkcionális szempontból nincs jelentősége.

[185] Lásd még DRMSz: 12-7. ábra.

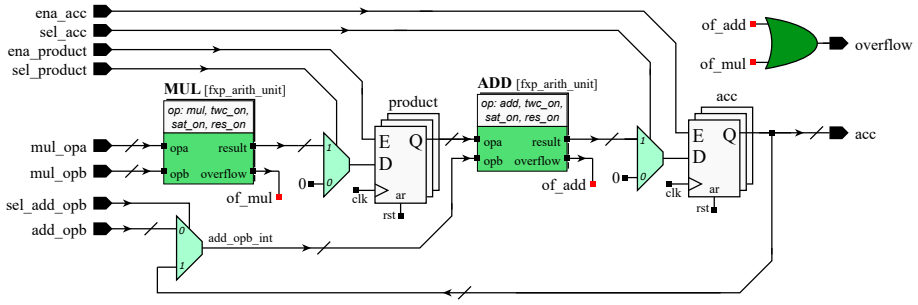
VHDL

```
1  L_MUL: entity work.fxp_arith_unit(rtl)
2    generic map (
3      "010", true, true, true, iipw, ifpw
4    )
5    port map (
6      mul_opa, mul_opb, mul_result, of_mul
7    );
8
9  L_PRODUCT: process ( clk, rst ) begin
10   if ( rst = '1' ) then
11     product <= (others => '0');
12   elsif ( rising_edge(clk) ) then
13     if ( ena_product = '1' ) then
14       if ( sel_product = '0' ) then
15         product <= (others => '0');
16       else product <= mul_result;
17       end if;
18     end if;
19   end if;
20 end process;
21
22 add_opb_int <= add_opb when sel_add_opb = '0'
23                else acc;
24
```

VHDL (folyt.)

```
25 L_ADD: entity work.fxp_arith_unit(rtl)
26   generic map ("000", true, true, true, iipw, ifpw)
27   port map (
28     product, add_opb_int, add_result, of_add
29   );
30
31 L_ACC: process ( clk, rst ) begin
32   if ( rst = '1' ) then
33     acc <= (others => '0');
34   elsif ( rising_edge(clk) ) then
35     if ( ena_acc = '1' ) then
36       if ( sel_acc = '0' ) then
37         acc <= (others => '0');
38       else acc <= add_result;
39       end if;
40     end if;
41   end if;
42 end process;
43
44 overflow <= of_add or of_mul;
```

Áramköri struktúra



4-246. ábra. *A szorzó-akkumulátor RTL-modellje*

egy-egy új eredmény állítható elő.[186] A *product* és *acc* regiszterek a *sel_product* és *sel_acc* vezérlőjelek segítségével nullázhatók. A struktúra tartalmaz egy visszacsatoló ágat; az *acc* regiszter értéke felhasználható az összeadás bemeneteként is a *sel_add_opb* vezérlőjel magas szintre állításával. E funkció segítségével a szorzó-akkumulátor tulajdonképpen egy integrátor szerepét töltheti be.[187] A két aritmetikai modult – a digitális jelfeldolgozásban előnyös – szaturációs üzemmódban használjuk, amelyben a túlcsordulásjelzések akkor aktiválódnak, ha az előállított eredmény szaturált, tehát – bár matematikailag „kevésbé rossz", mint a szaturációs logika nélkül előállított eset (átfordulás) – nem teljesen pontos.[188] A szorzó-akkumulátor a belső szorzó és összeadó almodulok túl-

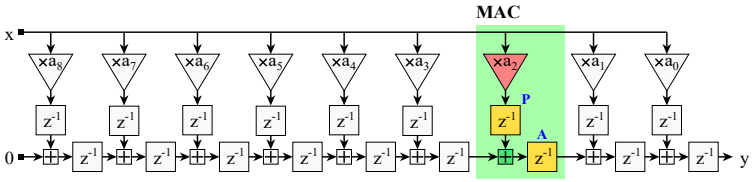
[186] Az *ena_product* és *ena_acc* jelek megfelelő ütemezésű vezérlésével a műveletvégzés sebessége korlátozható, a szorzó-akkumulátor működése az adatfeldolgozási lánc többi eleméhez hangolható.

[187] Az e pontban bemutatott pipeline (párhuzamos) FIR-szűrő esetén a visszacsatoló ágnak nincs szerepe. Készíthető azonban olyan megvalósítás is, amelyben a (4.5-1) kifejezésben definiált súlyozott összeget iteratív úton, egyetlen szorzó-akkumulátor felhasználásával állítjuk elő. Ebben az esetben a visszacsatoló ág biztosítja az iterációnkénti automatikus összegzést.

[188] Mivel véges számú bitet használunk a számok ábrázolására, ezért az eredmény általában nem tekinthető matematikai értelemben teljesen pontosnak. Szaturáció esetén azonban nem erről a „pontatlanságról", vagyis a számábrázolás felbontásbeli korlátozottságáról van szó, hanem arról, hogy adott operandusok esetén a művelet eredményének abszolútértéke kívül esik az adott számábrázolási módszer és a felhasznált bitek száma által meghatározott ábrázolási tartományon.

csordulást jelző kimeneteit egyszerűen egy *OR* kapun keresztül juttatja a kimentre, így e jelzés minden órajelciklusban frissül. Ha a túlcsordulás tényét regisztrálni vagy az előfordulások számát rögzíteni szeretnénk, akkor az ehhez szükséges logikát (pl. egy állapotjelzőt vagy számlálót) a szorzó-akkumulátoron kívül kell létrehoznunk (lásd pl. a FIR-szűrő vezérlőegységében: 4-253. ábra).

A szorzó-akkumulátor modul 4-246. ábrán bemutatott belső struktúráját figyelembe véve a FIR-szűrő modelljét a 4-247. ábra szerint finomíthatjuk. Figyeljük meg, hogy a szorzó-akkumulátorok *product* regiszterei egy-egy újabb késleltetőelem beiktatását jelentik – ami kedvező hatással lesz az áramkör időzítési tulajdonságaira – de mivel ezt a késleltetést az összes, párhuzamosan működő jelút tartalmazza, a megvalósított funkció nem változik meg.



4-247. ábra. *A bemutatott szorzó-akkumulátorral megvalósított FIR-szűrő felépítése P, A: a szorzó-akkumulátor product és acc regisztereit képviselő késleltető elemek*

A FIR-szűrő architekturális specifikációja és az elemi aritmetikai egység, a szorzó-akkumulátor birtokában immár minden adott ahhoz, hogy megtervezzük a teljes IP RTL-modelljét, amit – az adatfeldolgozó rendszerekre általánosan jellemző módon – egy műveletvégző egységből és egy, a műveletvégző elemek összehangolt működtetését végző vezérlőegységből állítunk össze.

4.5.6.1. Műveletvégző egység

A FIR-szűrőket alkalmazási szempontból alapvetően meghatározzák szűrőegyütthatóik és fokszámuk. Az általános felhasználhatóság érdekében az RTL-modell tervezése során e két sajátosságot paraméterként kezeljük; a szűrő fokszáma szintézis-, együtthatói pedig futásidejű paraméterek lesznek. A 4-245. ábrán látható, hogy az N-edfokú szűrő tulajdonképpen a szorzó-akkumulátor fokozat (N+1)-szeri példányosításával hozható létre, ami a VHDL *for ... generate* utasításával könnyen megvalósítható parametrikus formában is. A szűrőegyütthatók futásidejű módosíthatósága azonban nem ennyire magától értetődő. A 4-247. ábrán szemléltetett szerkezeten jól látható, hogy a megvalósítandó struktúrában a szorzó-akkumulátor modulok mindegyike egyszerre, párhuzamosan dolgozik minden mintafeldolgozási ciklusban. Ez azt jelenti, hogy bármely pillanatban az összes szűrőegyütthatónak egyszerre hozzáférhetőnek kell lennie, rákapcsolva az egyes szorzó-akkumulátor modulok megfelelő bemenetére. Az együtthatók futásidejű módosíthatóságát úgy érhetjük el, hogy mindegyik együtthatót egy-egy különálló, az adott szorzó-akkumulátor fokozathoz hozzárendelt regiszterben tároljuk, és valami módon lehetővé tesszük, hogy ezeknek a regisztereknek a tartalma módosítható legyen. Mindezek alapján a szűrő magját a korábban bemutatott szorzó-akkumulátor modul felhasználásával a 4-248. ábrán vázolt módon hozhatjuk létre.

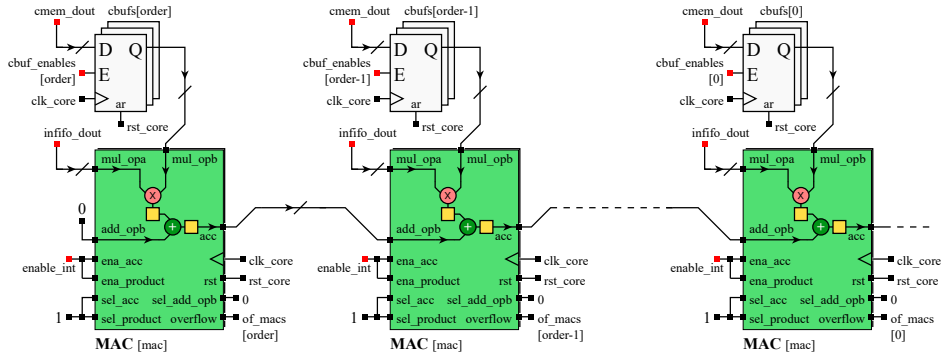
VHDL

```
1  mac_outputs(order+1) <= (others => '0');
2
3  L_FILTER_BLOCK: for i in 0 to order generate
4
5    L_CBUFS: process ( clk_core, rst_core )
6    begin
7      if ( rst_core = '1' ) then cbufs(i) <= (others => '0');
8      elsif ( rising_edge(clk_core) ) then
9        if ( cbuf_enables(i) = '1' ) then
10         cbufs(i) <= cmem_dout;
11       end if;
12     end if;
13   end process;
14
```

VHDL (folyt.)

```
15   L_MAC: entity work.mac(rtl)
16     generic map (iipw, ifpw)
17
18     port map (
19       clk         => clk_core,
20       rst         => rst_core,
21       mul_opa     => infifo_dout,
22       mul_opb     => cbufs(i),
23       add_opb     => mac_outputs(i+1),
24       ena_product => enable_int,
25       ena_acc     => enable_int,
26       sel_product => '1',
27       sel_acc     => '1',
28       sel_add_opb => '0',
29       acc         => mac_outputs(i),
30       overflow    => of_macs(i)
31   );
32
33 end generate;
```

Áramköri struktúra

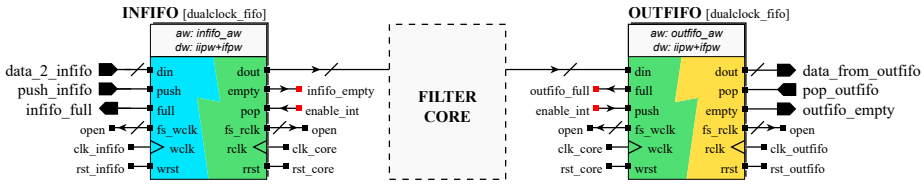


4-248. ábra. *A FIR-szűrő magjának RTL kapcsolási sémája*

Ebben a megvalósításban a szűrőegyütthatókat tehát regiszterekben *(cbufs)* tároljuk, értéküket pedig a *cbuf_enables* vektor megfelelő bitjének magas szintre húzásával meg is változtathatjuk (4-248. ábra: 9–11. sor). Ilyenkor a regiszterekbe a *cmem_dout* vektor értéke töltődik, amely – ahogy arra elnevezése is utal – egy memóriamodul *(coefficient memory, cmem)* kimeneti adatbusza lesz. E memóriamodul használatának részleteire a vezérlőegység tárgyalásánál térünk vissza.

Egy adatfolyam-feldolgozó áramkörben a rendszerórajel frekvenciája és a mintavételi frekvencia nem feltétlenül azonos, tehát az órajel-periódust és a mintafeldolgozási periódust érdemes valami módon áramkörileg különválasztani. Úgy is fogalmazhatunk, hogy általános esetben az adatfolyam-feldolgozó áramkör N órajelciklus alatt dolgoz fel egy bemeneti mintát és generál egy kimeneti mintát. Szélsőséges esetben N = 1, vagyis maximális feldolgozási sebesség esetén az áramkör az órajelének minden felfutóélénél egy-egy új eredményt állít elő. A 4-248. ábrán látható szűrőmag pontosan így működik. Annak érdekében, hogy a belső működés sebességét függetlenítsük a beágyazó rendszer sebességétől – közvetetten egyúttal a mintavételi frekvenciától –, a bemeneti mintasort egy FIFO-n keresztül fogadjuk, a kimeneti mintasort pedig ugyancsak egy FIFO-ba töltjük be. E két FIFO-tárolót a 4.2.5. pontban bemutatott *dualclock_fifo* IP-val valósítjuk meg, így a bemenetet generáló, a kimenetet feldolgozó, valamint a FIR-szűrő magját képező áramköri részletek mind egy-egy különálló, a többihez képest aszinkron órajelről járhatnak (lásd 4-249. ábra).

A FIR-szűrőben végrehajtott adatfeldolgozás sebessége tehát független a rendszer egyéb komponenseinek feldolgozási sebességétől; a bemeneti FIFO tetszőleges ütemben tölthető, a kimeneti FIFO pedig ugyancsak tetszőleges ütemben üríthető.[189]



4-249. ábra. *FIFO-tárak a szűrő magjának bemenetén és kimenetén*

4.5.6.2. Vezérlőegység

A FIR-szűrő műveletvégző egysége tehát egy többfokozatú szűrőmagból, valamint egy bemeneti és egy kimeneti FIFO-ból áll. Ehhez a műveletvégző egységhez olyan vezérlést kell terveznünk, amely eleget tesz az alábbi követelményeknek:

1. Lehetővé teszi, hogy a felhasználó ki/bekapcsolhassa az áramkört. Bekapcsolt állapotban a szűrő a bementi FIFO-jából folyamatosan olvassa ki a bemeneti mintasort, kimeneti FIFO-jába pedig ugyancsak folyamatosan tölti be a feldolgozott mintasort.
2. Ha a bemeneti FIFO kiürül vagy a kimeneti FIFO telítődik, akkor az áramkör vezérlése automatikusan szünetelteti a feldolgozást a belső állapot (a szorzóakkumulátor-fokozatok belső regisztereinek értékei) megőrzése mellett.
3. Lehetővé teszi a felhasználó számára a szűrőegységek együtthatók futásidőben való konfigurálását. Az együtthatók frissítése közben a vezérlés a szűrő működését szünetelteti, a konfigurációs folyamat állapotát pedig jelzi a felhasználó felé.[190]

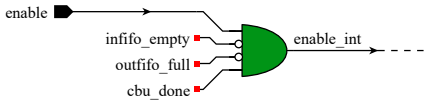
A feldolgozás autonóm engedélyezése/tiltása ◆ A 4-248. és 4-249. ábrákat vizsgálva látható, hogy a bemeneti FIFO olvasását és a kimeneti FIFO írását, valamint a szorzó-akkumulátorok működését egyazon vezérlőjel *(enable_int)* engedélyezi vagy tiltja. A belső működés a specifikáció értelmében akkor engedélyezhető, ha

[189] A digitális jelfeldolgozó rendszerekben a mintavételi frekvencia természetesen központi jelentőségű; elég csak azt felidéznünk, hogy egy digitális szűrő együtthatóit a kívánt átviteli tulajdonságok és a mintavételi frekvencia együtt határozzák meg. A 4-249. ábrán látható szerkezeti felépítés nagy rugalmasságot biztosít a különböző részáramkörök saját órajel-frekvenciájának optimalizálásában, ugyanakkor fontos kiemelni, hogy a mintavételi frekvenciát nem ezeknek az almoduloknak az órajel-frekvenciája határozza meg, hanem az, hogy a feldolgozott analóg jelen a rendszer bemenetén lévő A/D-átalakító milyen gyakran végzi el az analóg-digitális konverziót. Ez lehet magának az A/D-átalakítóhoz tartozó interfészadapter áramkörnek (lásd pl. 4.4.9. pont: FIFO-alapú órajel-szinkronizáló burkolók), vagy az azt üzemeltető ütemező/vezérlőrendszernek a tulajdonsága. A FIR-szűrő magjának – sőt, valójában az adatfeldolgozó rendszer bármely komponensének – maximális órajel-frekvenciája természetesen egy felső korlátot jelent a teljes rendszer mintafeldolgozási ütemére vonatkozóan.

[190] A „futásidőben” kifejezés tehát nem arra utal, hogy a szűrő működése közben változtatjuk az együtthatókat. A futásidejű paraméterek az áramkör azon konfigurációs beállításai, amelyeket a szintézis után – technológiától függően a gyártás vagy az FPGA felkonfigurálása után – is tudunk változtatni (vö. szintézis-paraméterek).

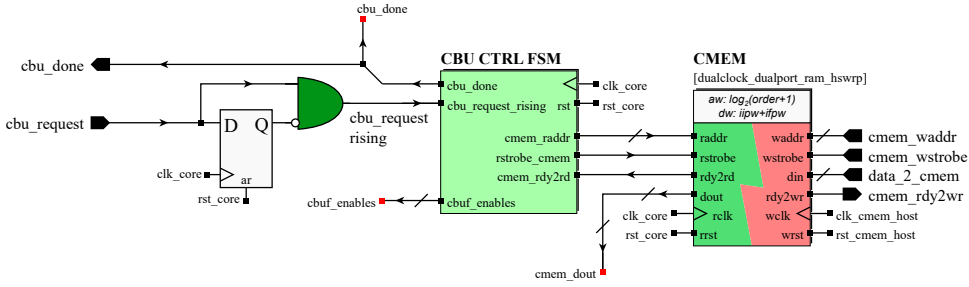
- a felhasználó nem tiltja azt,
- a bemeneti FIFO-ban rendelkezésre áll a következő feldolgozandó minta,
- a kimeneti FIFO-ban van hely a következő feldolgozott mintának,
- az együtthatók újrakonfigurálásának mechanizmusa inaktív.

Ezek alapján az *enable_int* vezérlőjelet a 4-250. ábrán látható módon állíthatjuk elő.



4-250. ábra. *A belső feldolgozást engedélyező/tiltó vezérlőjel előállítása (**c**oefficient **b**uffers **u**pdate, cbu)*

A szűrőegyütthatók újrakonfigurálása ◆ A szűrő üzemeltetése közben az aktuális szűrőegyütthatókat az egyes szorzó-akkumulátor fokozatok bemenetét meghajtó regiszterekben (4-248. ábra: *cbufs[0] ... cbufs[order]*) tároljuk. Ezek újrakonfigurálásának lehetőségét úgy biztosítjuk, hogy létrehozunk egy ún. együttható-memóriát *(cmem)*, amelynek beíró interfészét kiajánljuk a felhasználónak. A memória megvalósítására a *dualclock_dualport_ram_hswrp* IP-magot használjuk fel, így függetlenítve a beírást végrehajtó felhasználó órajelét a FIR modul órajelétől. Miután az új együtthatók értékét a felhasználó a memóriába beírta, a *cbufs* regiszterek frissítését a *cbu_request* bemeneten keresztül kezdeményezheti. Ekkor a FIR IP-mag a *cbu_done* kimenet alacsony szintre húzásával jelzi a felhasználó felé, hogy az újrakonfigurálást megkezdte, egyúttal ideiglenesen letiltva a szűrő működését. Az új együtthatókat egy egyszerű állapotgép olvassa ki a memóriából és helyezi el a megfelelő *cbufs* regiszterben, majd a folyamat végén újra magas szintre húzza a *cbu_done* jelet. Az együtthatók újrakonfigurálásának mechanizmusát megvalósító részáramkörök RTL kapcsolási sémája a 4-251., az együtthatók frissítését végző állapotgép VHDL-modellje pedig a 4-252. ábrán látható.



4-251. ábra. *A szűrőegyütthatók újrakonfigurálását végző áramköri elemek RTL-modellje*

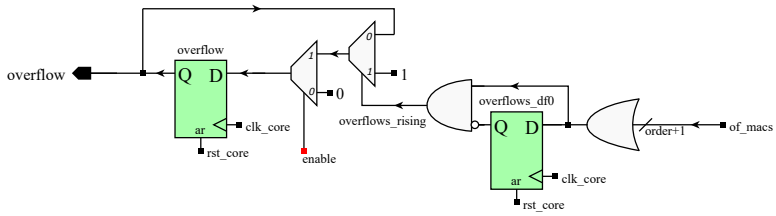
A FIR IP-mag vezérlőegységét a műveletvégzés során a szorzó-akkumulátorokban esetleg keletkező túlcsordulás[191] kezeléséért felelős áramköri részletek teszik teljessé (lásd 4-253. ábra). Mivel maguk a szorzó-akkumulátor modulok minden órajelciklusban frissítik túlcsordulásjelző kimenetüket, ezért az eseményt permanensen regisztráló állapotjelzőt a FIR IP vezérlőegységében kell megvalósítanunk.

[191] A szorzó-akkumulátorok beállítása miatt a túlcsordulásjelzés valójában szaturációt jelent.

VHDL

```
L_CBU_CTRL_FSM: process ( clk_core, rst_core ) begin
 if ( rst_core = '1' ) then
   state <= idle; cbu_done <= '1'; cbuf_enables <= (others => '0');
   cmem_raddr <= (others => '0'); coeff_pointer <= 0; cmem_rstrobe <= '0';
 elsif ( rising_edge(clk_core) ) then
   case state is

     when idle => if ( cbu_request_rising = '1' ) then
                    cbu_done <= '0'; state <= update_cbufs_1;
                  end if;

     when update_cbufs_1 => cbuf_enables <= (others => '0');
                            cmem_raddr <= std_logic_vector (
                                            to_unsigned(coeff_pointer,integer(ceil(log2(real(order+1)))))
                                          );
                            cmem_rstrobe <= '1';
                            state <= update_cbufs_2;

     when update_cbufs_2 => cmem_rstrobe <= '0'; state <= update_cbufs_3;

     when update_cbufs_3 => if ( cmem_rdy2rd = '1' ) then
                              cbuf_enables(coeff_pointer) <= '1';

                              if ( coeff_pointer < order ) then
                                coeff_pointer <= coeff_pointer + 1; state <= update_cbufs_1;
                              else coeff_pointer <= 0; cbu_done <= '1'; state <= idle;
                              end if;

                            end if;

     when others => null;

   end case;
 end if;
end process;
```

4-252. ábra. *A szűrőegyütthatókat tartalmazó cbufs regiszterek frissítését végző állapotgép*



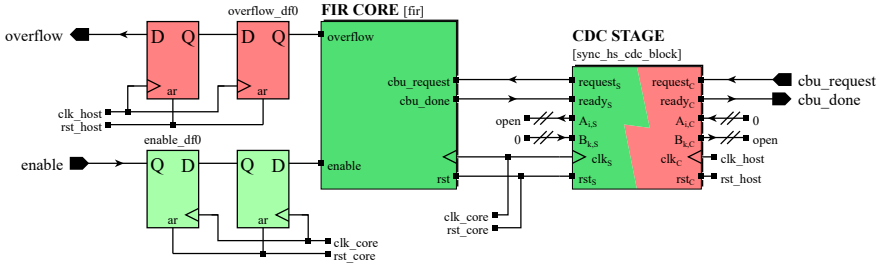
4-253. ábra. *A szorzó-akkumulátorok túlcsordulás-jelzéseit regisztráló állapotjelző*

Ezzel a megoldással, ha a szűrő aktiválása után bármikor, bármelyik szorzó-akkumulátorban túlcsordulás következik be, akkor az *overflow* kimenet magas szintre áll, és úgy is marad mindaddig, amíg a felhasználó a műveletvégzést tiltó *enable* bemenetet alacsony szintre nem állítja, visszaigazolva, hogy a túlcsordulásjelzést tudomásul vette.

4.5.6.3. Órajel-szinkronizálás

A FIR modul órajel-szinkronizáló burkolómodulja az FFT IP-magnál látottakhoz hasonló módon készíthető el. A szűrőegyütthatókat tároló memóriamodul, valamint a be- és kimeneti FIFO-tárolók ebben az esetben is dual-clock képességű, tehát önmagukban is órajel-szinkronizáló fokozatként működő IP-magok példányai, ezekkel tehát nincs tennivalónk. A szűrőegyütthatók frissítését végző állapotgép egy szinkron kézfogásos interfészen keresztül hozható működésbe,

ami az általános vektorszinkronizáló fokozat egy példányával illeszthető a felhasználónak a FIR IP magórajeléhez képest aszinkron órajeltartományához (lásd 4-254. ábra).



4-254. ábra. *A FIR modul vezérlőegységének órajel-szinkronizáló fokozatai*

Statikus jelek lévén a szűrőmag működésének ideiglenes felfüggesztését lehetővé tévő *enable* bemenet, valamint az *overflow* állapotjelző kimenet egy-egy double-flop szinkronizáló fokozattal kezelhető.

4.5.7. RC aluláteresztő szűrő

Áttanulmányozásra javasolt alfejezetek:

***5.7**. Az elsőfokú RC aluláteresztő szűrő diszkretizálása*
***4.2**. On-chip adattárolás*
***4.5.1**. Aritmetika és jelfeldolgozás – Fixpontos aritmetikai egység*
***4.5.2**. Aritmetika és jelfeldolgozás – Iteratív fixpontos szorzó*

A 4.5.6. pontban bemutatott FIR-szűrő egy nagyon rugalmas megoldás, átviteli tulajdonságai a fokszám és a szűrőegyütthatók módosításával széles skálán változtathatók. A 4-244. ábrán látható konkrét architektúra további fontos sajátossága, hogy nagyon gyors működésre képes; a mintavételi frekvencia elérheti az órajel frekvenciáját, vagyis minden órajel-felfutóélnél egy-egy újabb feldolgozott minta állhat elő. Ennek a számítási teljesítménynek azonban ára van; az N-edfokú szűrő megvalósításához N+1 db szorzó-akkumulátorra van szükség, amelyek mindegyike egy saját szorzó és egy saját összeadó fokozatot tartalmaz. Bizonyos technológiák (kisebb méretű FPGA-k) esetén a blokk-szorzó elemek száma korlátozott lehet, ilyen értelemben tehát a nagymértékben párhuzamosított architektúra igen költségesnek számíthat. Ráadásul az, hogy szükség van-e a nagy mintavételi frekvenciára, ugyancsak alkalmazásfüggő; felesleges MHz nagyságrendű mintavételi frekvenciával üzemeltetni a rendszert, ha a feldolgozandó jel legmagasabb frekvenciájú spektrális összetevője is egy nagyságrenddel kisebb frekvenciájú (pl. audio-jelek), hiszen így szükségtelenül növelnénk rendszerünk – órajelének frekvenciájával arányosan növekvő – fogyasztását. Ebben a pontban egy, a FIR-szűrőnél korlátozottabb képességű, elsőfokú RC aluláteresztő szűrő erőforrás-felhasználásra optimalizált kivitelét mutatjuk be.

Egy aluláteresztő RC-tag diszkretizálásával az alábbi rendszeregyenletet kapjuk (részletesen lásd 5.7. Függelék):

$$y[n] = Ax[n] + By[n-1] \tag{4.5-2}$$

ahol az

$$A = \frac{T_S}{RC + T_S}, \qquad B = 1 - A \tag{4.5-3}$$

együtthatók a szűrő pontos átviteli sajátosságait meghatározó állandók; *R* és *C* a diszkretizált RC-tag rezisztív és kapacitív összetevőinek értékei, T_s pedig a mintavételi idő.[192] A 4.5-2 összefüggésből látható, hogy a szűrő az aktuális kimenetének értékét az aktuális bemenet és az előző lépésben kiszámított kimenet súlyozott összegeként állítja elő. Tehát minden számítási ciklusban két szorzást és egy összeadást kell elvégeznünk. Az erőforrás-felhasználás minimalizálása érdekében meghozott első döntésünk legyen az, hogy a két szorzást egyazon szorzó modullal, egymás után végezzük el.[193] A szorzás és összeadás műveletek kombinált elvégzésére a FIR-szűrőben a 4-246. ábrán látható szorzó-akkumulátort használtuk. Ez az újrafelhasználható modul ebben az esetben is megállná a helyét, de a leendő áramkör költségének csökkentése érdekében ezúttal más megoldást választunk. Olyan tervet készítünk, amely – a *MAC* egységgel ellentétben – egyáltalán nem tartalmaz kombinációs hálózatként leírt szorzókat, így lehetővé válhat a szűrő létrehozása olyan eszközökben is, amelyekben egyáltalán nincsenek blokk-szorzó modulok.[194] A fixpontos szorzás(oka)t tehát nem – a kombinációs hálózatokra jellemző módon – egyetlen lépésben, hanem iteratív úton végezzük el, a 4.5.2. pontban tárgyalt iteratív fixpontos szorzó IP-mag segítségével. A szűrő RTL-modelljét az iteratív fixpontos szorzó és a 4.5-2 összefüggést megvalósító algoritmus összes többi ideiglenes adattárolási és aritmetikai műveletét egyetlen one-shot leírásba tömörítő állapotgép kombinációjaként építjük fel. Az RTL-modell szerkezetét és az állapotgép vázlatát a 4-255., az állapotgép vezérlési állapotainak részleteit pedig a 4-256. ábra szemlélteti.

Az állapotgép működését a *request* jelet fogadó felfutóél-detektor kimenetén megjelenő pozitív impulzus indítja el. Ennek hatására az állapotgépben alacsony szintre állítjuk a szorzó modul bemeneteit meghajtó multiplexerek kiválasztóbemeneteit, majd egy szinkron kézfogásos tranzakciót kezdeményezünk a szorzó felé (4-256. ábra: 4–5. sor). Ennek végén a szorzó *result* kimenetén előáll az *Ax* szorzat, amelyet az állapotgép *tmp* nevű regiszterébe töltünk (4-256. ábra: 14. sor). A második szorzás kezdeményezésének pillanatában a multiplexerek kiválasztójeleit magas szintre húzzuk (4-256. ábra: 25. sor), így a második szorzás végén a szorzó modul *result* kimenetén a *By* szorzat jelenik meg. Az órajelnek az állapotgép *s5* állapotában megérkező felfutóéle pillanatában tehát az *Ax* és a *By* szorzatok rendre az állapotgép *tmp* kimeneti regiszterében és a szorzó modul *result* kimenetén állnak rendelkezésre.

E két fixpontos értéket a 4.5.1. pontban bemutatott, többfunkciós fixpontos aritmetikai egység magját képező eljárás *(fxp_arith_procedure)* segítségével adjuk össze (4-256. ábra: 36–39. sor), az

[192] Ez a szűrő valóban kevésbé rugalmas, mint a korábban bemutatott FIR-szűrő, hiszen együtthatóinak változtatásával aluláteresztő jellege nem, csupán határfrekvenciája változtatható meg.

[193] Ez a kisméretű – egyúttal óhatatlanul kisebb számítási teljesítményű – áramkörök fejlesztése során alkalmazott egyik leghatékonyabb módszer, amit az erőforrások vezérelt újrafelhasználásának nevezünk (vö. 4.5.1.1. pont).

[194] Érdemes megjegyezni, hogy a blokk-szorzókat nem tartalmazó eszközökben is létrehozható a *MAC* egységben is használt, kombinációs hálózatként modellezett szorzó, de ha ezt a szintézis-szoftvernek az adott eszköz általános célú logikai elemeiből (LUT-okból) kell felépítenie, akkor a kész áramkör időzítési és erőforrás-felhasználási szempontból is nagyon kedvezőtlen lehet.

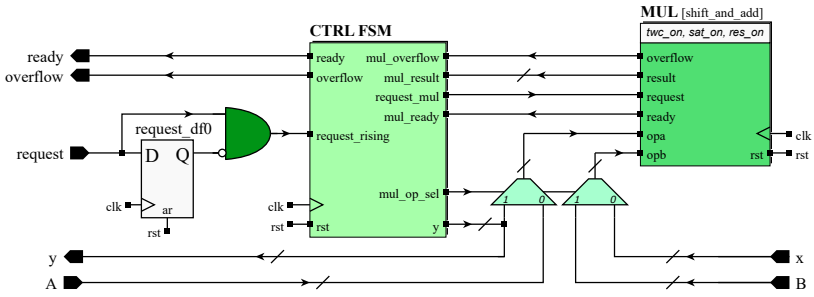
VHDL

```
1  L_REQUEST_DETECTOR: process ( clk, rst ) begin
2    if ( rst = '1' ) then request_df0 <= '0';
3    elsif ( rising_edge(clk) ) then
4      request_df0 <= request;
5    end if;
6  end process;
7  request_rising <= request and not request_df0;
8
9  L_CTRL_FSM: process ( clk, rst )
10
11   variable v_add_result: std_logic_vector
12                          (iipw+ifpw-1 downto 0);
13   variable v_of_add:     std_logic;
14
15 begin
16   if ( rst = '1' ) then
17     ready <= '1'; state <= idle;
18     request_mul <= '0'; mul_op_sel <= '0';
19     y <= (others => '0'); tmp <= (others => '0');
20     overflow <= '0';
21   elsif ( rising_edge(clk) ) then
22
23     case state is
24
25       when idle => [...]
26
27       [...]
28
29       when others => null;
30
31     end case;
32
33   end if;
34 end process;
```

VHDL

```
1  mul_op_a <= A when mul_op_sel = '0' else y;
2  mul_op_b <= x when mul_op_sel = '0' else B;
3
4  L_MUL: entity work.shift_and_add(rtl)
5    generic map (
6      twos_complement_on  => true,
7      saturation_on       => true,
8      resize_on           => true,
9      iipw                => iipw,
10     ifpw                => ifpw
11   )
12
13   port map (
14     clk      => clk,
15     rst      => rst,
16     request  => request_mul,
17     ready    => mul_ready,
18     opa      => mul_op_a,
19     opb      => mul_op_b,
20     result   => mul_result,
21     overflow => of_mul
22   );
```

Áramköri struktúra



4-255. ábra. *Az RC szűrő szerkezete*

összeadás eredményét pedig az *y* kimeneti regiszterbe töltjük.[195] Figyeljük meg, hogy az *y* regiszter értékét a műveletvégzés során nem, csak annak végén változtatjuk meg, így a második szorzás elvégzésekor, amelynek egyik operandusa az állapotgép *y* kimeneti regisztere, az még valóban az előző ciklusban kiszámított érték, ahogy azt a specifikáció (4.5-2) előírja.

[195] A fixpontos műveletvégző elem one-shot típusú állapotgépbe való beágyazásával azt érjük el, hogy az összeadóként felparaméterezett eljárásból szintetizálódó kombinációs hálózat az állapotgép kimeneti logikájának lesz a része. Fontos kiemelni, hogy a fixpontos aritmetikai eljárást ebben az esetben összeadóként, az *operation* bemenetét fix értékre (000_2) állítva használjuk. A szintézis során – a constant propagation eljárás nyomán (lásd még: DRMSz 8-32. ábra) – a többi műveletet megvalósító áramköri részletek nem kerülnek be a végső netlistába. E megoldás tehát nem igényel szorzó modulokat, dacára annak, hogy a HDL-modellben jelen van a szorzás művelet.

VHDL

```
1  when idle =>
2    if ( request_rising = '1' ) then
3      ready <= '0'; overflow <= '0';
4      mul_op_sel <= '0';
5      request_mul <= '1';
6      state <= s1;
7    end if;
8
9  when s1 => state <= s2;
10
11 when s2 =>
12   if ( mul_ready = '1' ) then
13
14     request_mul <= '0'; tmp <= mul_result;
15
16     if ( of_mul = '1' ) then
17       overflow <= '1';
18     end if;
19
20     state <= s3;
21
22   end if;
23
```

VHDL (folyt.)

```
24 when s3 =>
25   mul_op_sel <= '1'; request_mul <= '1';
26   state <= s4;
27
28 when s4 => state <= s5;
29
30 when s5 =>
31   if ( mul_ready = '1' ) then
32
33     request_mul <= '0';
34     if ( of_mul = '1' ) then overflow <= '1'; end if;
35
36     fxp_arith_procedure("000", true, true, true,
37                         iipw, ifpw,
38                         tmp, mul_result,
39                         v_add_result, v_of_add);
40     y <= v_add_result;
41
42     if ( v_of_add = '1' ) then overflow <= '1'; end if;
43
44     ready <= '1';
45     state <= idle;
46
47   end if;
```

4-256. ábra. *Az RC szűrő állapotgépének vezérlési állapotai*

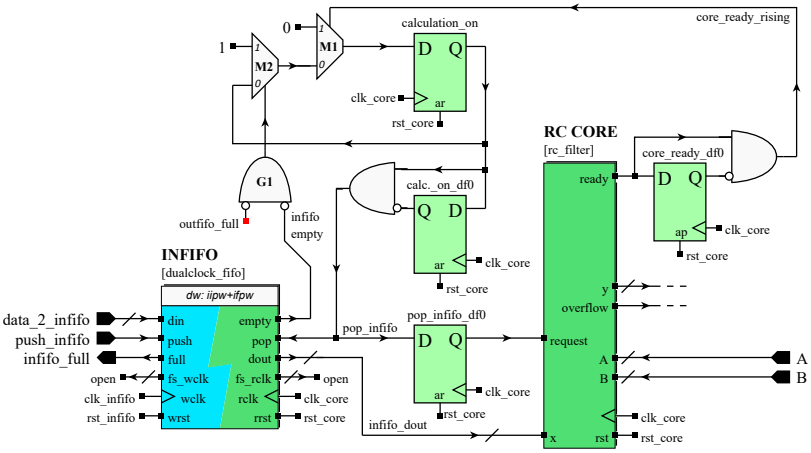
Az esetleges túlcsordulást a többi adatfeldolgozó modulnál látottakhoz hasonlóan kezeljük; az állapotgép minden ciklus elején nullázza az *overflow* kimenetet (4-256. ábra: 3. sor), majd a két szorzás és az összeadás elvégzésének végén az éppen megszólított aritmetikai egység túlcsordulásjelzése alapján frissíti azt (4-256. ábra: 16–18., 34. és 42 sor). Így a szűrő felhasználói interfészén kezdeményezett szinkron kézfogásos tranzakciók végén ellenőrizhető, hogy az adott számítási ciklusban bekövetkezett-e túlcsordulás.

4.5.7.1. Órajel-szinkronizálás

Ha a bemeneti FIFO-jába kellően gyors ütemben töltünk adatot és kimeneti FIFO-ját is kellően gyakran olvassuk ki, akkor a FIR-szűrő magjában az *enable_int* jel folyamatosan magas szintű lesz, a belső logika tehát az órajel minden felfutóélénél egy új kimenetet állít elő, az áramkör kihasználtsága így maximális. Az RC szűrő – az erőforrások vezérelt újrafelhasználásán alapuló belső architektúra okán – ilyen gyors működésre nem képes, szinkron kézfogásos interfészén az egyetlen kimeneti mintát előállító tranzakció több órajelciklust igényel. Egy jelfolyam feldolgozása tehát a kézfogásciklusok periodikus ismétlésével lehetséges, és e ciklusok időtartamának mindenképpen kisebbnek kell lennie a mintavételi időnél. Minél jobban megközelíti a kézfogás-periódus időtartama a mintavételi időt, annál jobb az áramkör kihasználtsága, energiahatékonysága. A felhasználó feladata, hogy e hatékonyság maximalizálása érdekében – az adott alkalmazás sajátosságainak, igényeinek ismeretében – összehangolja a mintavételi időt és a szűrőmag órajelének frekvenciáját.

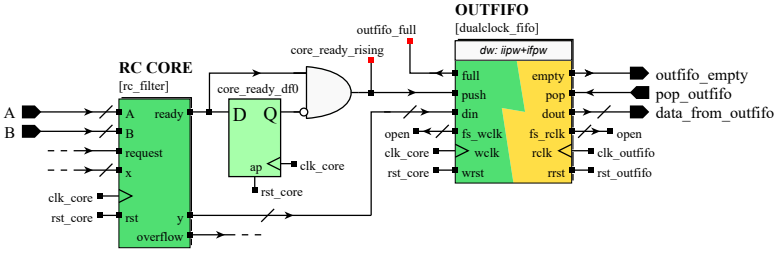
A szűrőmaghoz ezúttal is praktikus egy FIFO-alapú CDC-burkolómodult kialakítani, amely a bemenő adatfolyamot egy FIFO-ból olvassa ki, a szűrőmagot automatikusan indítja, és a számítási ciklus végén annak eredményét ugyancsak automatikusan írja be egy kimeneti FIFO-ba. Egy ilyen burkolómodul RTL kapcsolási sémáját mutatja be a 4-257. és 4-258. ábra. A CDC-burkolómodul bemeneti FIFO-jából az adatokat a szűrőmag bemenetére juttató logika akkor lép működésbe, amikor a bemeneti FIFO nem üres és a kimeneti FIFO nincs tele. Ekkor a *G1 AND* kapu az *M2* multiplexeren keresztül magas szintre húzza a *calculation_on* állapotjelzőt. Az ennek

kimenetén lévő felfutóél-detektor egy pozitív impulzust állít elő a bemeneti FIFO *pop* bemenetén *(pop_infifo)*, aminek hatására az adat az órajel következő felfutóélénél megjelenik az *infifo_dout* buszon (a FIFO-k a FIR-szűrőben lévőkkel megegyező módon nem kézfogásos működésűek).



4-257. ábra. *Az RC szűrő FIFO-alapú CDC-burkolómoduljának szerkezete (1)*

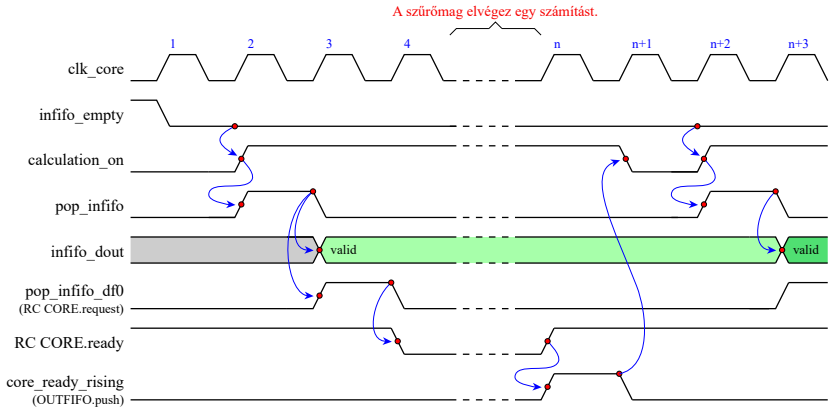
Mivel a FIFO azonnal szolgáltatja az adatot, ezért a *pop_infifo* jelen megjelenő pozitív impulzust egyetlen órajelciklussal megkésleltető *pop_infifo_df0* flip-flop kimenetét felhasználhatjuk arra, hogy vele kezdeményezzük a szűrőmagban a számítási ciklus indítását. A számítási ciklus végén a szűrőmag *ready* jelzését fogadó felfutóél-detektor az *M1* multiplexeren keresztül nullázza a *calculation_on* állapotjelzőt. Az *M1* és *M2* multiplexerek közötti prioritásnak fontos szerepe van; a *calculation_on* állapotjelző így legalább egy órajelciklus időtartamra nullázódik akkor is, ha a bemeneti FIFO *empty* jelzése mindvégig alacsony szintű, a *pop* bemenetet meghajtó felfutóél-detektor tehát újra jelezni fog.[196]



4-258. ábra. *Az RC szűrő FIFO-alapú CDC-burkolómoduljának szerkezete (2)*

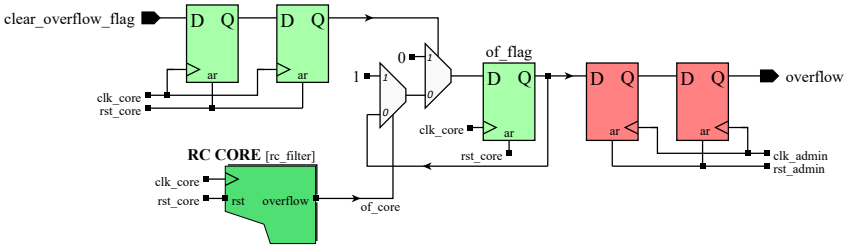
A szűrőmag *ready* jelzésének felfutóélét jelző éldetektor kimenetét nemcsak a *calculation_on* állapotjelző nullázására, hanem a kimeneti FIFO *push* bemenetének meghajtására, vagyis a számítás eredményének kimeneti FIFO-ba írására is felhasználjuk. A CDC-burkolómodul működését a 4-259. ábrán látható hullámforma szemlélteti.

[196] Vö. 4-42. ábra: *M1* és *M2* multiplexerek szerepe az UART-adó FIFO-alapú CDC-burkolójában.



4-259. ábra. *A CDC-burkolómodul működése*

A szűrőmag *overflow* jelzése a burkolómodul autonóm viselkedéséből adódóan dinamikusan változik; minden számítási ciklus elején nullázódik, azok végén pedig potenciálisan magas szintű lesz. E dinamikus működés miatt a CDC-burkolómodulban létre kell hoznunk egy egyszerű logikát, amely regisztrálja a túlcsordulás bekövetkezését, és e jelzést mindaddig fenntartja, ameddig a felhasználó nem törli azt. A 4-260. ábrán látható kapcsolási sémában a szűrőmag *overflow* kimenetének *(of_core)* bármikori magas szintje az *of_flag* állapotjelző egybe billenését idézi elő. Az *of_flag* értékét egy egyszerű double-flop szinkronizáló fokozattal visszük át a szűrő magórajeléhez és a FIFO-k órajeleihez képest is aszinkronnak tekintett *clk_admin* órajeltartományba.



4-260. ábra. *Az overflow jelzés kezelése a CDC-burkolómodulban*

Az *of_flag* állapotjelző törléséhez bevezetjük a *clear_overflow_flag* bemenetet, amelyről ugyancsak feltételezzük, hogy meghajtója a szűrő magórajeléhez képest aszinkron órajeltartományban működik, ezért ezt a jelet is egy double-flop szinkronizáló fokozattal fogadjuk.

5

Függelék

5.1. Kellékek a CPU assembly nyelvű programozásához

5.1.1. Utasításkészlet

5.1.1.1. Natív (hardver) utasítások

i2rfl

Utasítás	Utasítás szerkezete: az utasításregiszter (Instruction Register, IR) bitjei															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
i2rfl	0	0	0	0	d	d	d	d	i	i	i	i	i	i	i	i

5-1. táblázat. *Az i2rfl utasítás szerkezete*

Utasítás	Utasítás funkciója	Módosított flag-ek
i2rfl	RF[IR[11:8]][15:8] = 0 RF[IR[11:8]][7:0] = IR[7:0]	Z

5-2. táblázat. *Az i2rfl utasítás funkcionális definíciója*

Az *i2rfl (immediate to register file – low)* utasítás segítségével lehetőség van konstansok betöltésére a regisztertömbbe. Mivel az utasításregiszter és a regisztertömb egyes rekeszei egyaránt 16-bitesek, nem lehet tetszőleges, 16-bites konstanst betölteni egyetlen gépi utasítás segítségével. Az *i2rfl* utasításban 8-bites konstanst adhatunk meg, amely a megcímzett regiszter alsó bájtjába töltődik be. A megcímzett regiszter felső bájtja a 0x00 értéket kapja. 16-bites konstans betöltéséhez több gépi utasítást kell használnunk (lásd pl. 5-49. táblázat: *i2rf* pszeudoutasítás). Az 5-3. táblázat néhány példát mutat az *i2rfl* utasítás viselkedésére vonatkozóan.

ASM	Utasításregiszter	Utasítás eredménye
`i2rfl $4, 0x23`	`0x0423 (0000_0100_0010_0011)`	`RF[4]: 0x0023, Z: 0, N: 0`
`i2rfl $1, 0x00`	`0x0100 (0000_0001_0000_0000)`	`RF[1]: 0x0000, Z: 1, N: 0`
`i2rfl $15, 0xFF`	`0x0FFF (0000_1111_1111_1111)`	`RF[15]: 0x00FF, Z: 0, N: 0`
`i2rfl $13, 0x83`	`0x0D83 (0000_1101_1000_0011)`	`RF[13]: 0x0083, Z: 0, N: 0`

5-3. táblázat. *Példák az i2rfl utasítás használatára*

A fenti táblázatban látható, hogy az *i2rfl* utasítás nem végez előjel-kiterjesztést, az utasításban megadott konstanst tehát előjel nélküli egészként kezeli. Negatív számok betöltéséhez több gépi utasítást kell használnunk!

load

Utasítás	Az utasításregiszter bitjei															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
load	0	0	0	1	d	d	d	d	x	x	x	x	y	y	y	y

5-4. táblázat. *A load utasítás szerkezete*

Utasítás	Utasítás funkciója	Módosított flag-ek
load	RF[IR[11:8]] = PERIPH[RF[IR[7:4]] + RF[IR[3:0]]]	Z, N

5-5. táblázat. *A load utasítás funkcionális definíciója*

A *load* utasítás segítségével a periféria-áramkörökből adatot olvashatunk ki és tölthetünk be a processzor regisztertömbjébe. Az olvasási cím két általános célú regiszter értékének összegzésével áll elő. Ez az ún. indirekt indexelt regiszter címzés a tömbök hatékony kezelését teszi lehetővé.[197] A *load* utasítás az adatbuszról beolvasott 16-bites érték alapján állítja be az állapotjelzők értékét. Az 5-6. táblázat néhány példát mutat a *load* utasítás viselkedésére vonatkozóan.

ASM	Utasításregiszter	Előzmények	Utasítás eredménye

`i2rfl $2, 0x03`
`i2rfl $3, 0x04`
`load $15, $2, $3`
	`0x0203 (0000_0010_0000_0011)`
	`0x0304 (0000_0011_0000_0100)`
	`0x1F23 (0001_1111_0010_0011)`
		`DMEM[7]: 0x4567`	`RF[2]: 0x0003, Z: 0, N: 0`
			`RF[3]: 0x0004, Z: 0, N: 0`
			`RF[15]: 0x4567, Z: 0, N: 0`

`i2rfl $5, 0xF0`
`i2rfl $4, 0x04`
`load $15, $5, $4`
	`0x05F0 (0000_0101_1111_0000)`
	`0x0404 (0000_0100_0000_0100)`
	`0x1F54 (0001_1111_0101_0100)`
		`DMEM[244]: 0x8211`	`RF[5]: 0x00F0, Z: 0, N: 1`
			`RF[4]: 0x0004, Z: 0, N: 0`
			`RF[15]: 0x8211, Z: 0, N: 1`

5-6. táblázat. *Példák a load utasítás használatára*

store

Utasítás	Az utasításregiszter bitjei															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
store	0	0	1	0	s	s	s	s	x	x	x	x	y	y	y	y

5-7. táblázat. *A store utasítás szerkezete*

Utasítás	Utasítás funkciója	Módosított flag-ek
store	PERIPH[RF[IR[7:4]] + RF[IR[3:0]]] = RF[IR[11:8]]	-

5-8. táblázat. *A store utasítás funkcionális definíciója*

A *store* utasítás segítségével a periféria-áramkörökbe adatot írhatunk ki a processzor regisztertömbjéből. Az írási cím két általános célú regiszter értékének összegzésével áll elő. Ez az ún. indirekt indexelt regiszter címzés a tömbök hatékony kezelését teszi lehetővé. Mivel a *store* utasítás

[197] Az egyik regiszter tárolja a tömb kezdőcímét, a másik pedig az eltolást, vagyis a kiolvasandó elem tömbbeli indexét.

nem ír be adatot a regisztertömbbe, ezért az állapotjelzők értékét sem módosítja. Az 5-9. táblázat néhány példát mutat a *store* utasítás viselkedésére vonatkozóan.

ASM	Utasításregiszter	Előzmények	Utasítás eredménye

`i2rfl $2, 0x03`
`i2rfl $3, 0x04`
`store $15, $2, $3`
	`0x0203 (0000_0010_0000_0011)`
	`0x0304 (0000_0011_0000_0100)`
	`0x2F23 (0010_1111_0010_0011)`
		`RF[15]: 0xDEAD`	`RF[2]: 0x0003, Z: 0, N: 0`
			`RF[3]: 0x0004, Z: 0, N: 0`
			`DMEM[7]: 0xDEAD`

`i2rfl $6, 0xEA` `store $15, $6, $1`	`0x06EA (0000_0110_1110_1010)` `0x2F61 (0010_1111_0110_0001)`	`RF[1]: 0xFFFF` `RF[15]: 0xBEEF`	`RF[6]: 0x00EA, Z: 0, N: 1` `DMEM[233]: 0xBEEF`

5-9. táblázat. *Példák a store utasítás használatára*

Az utasítások többsége esetén az utasításregiszter [11:8] mezője az utasítás célregiszterének indexét tárolja. A *store* utasítás esetén nem beszélhetünk célregiszterről, hiszen az utasítás a regisztertömbből ír ki adatot az adatbuszra. Ennek megfelelően az utasításregiszter [11:8] mezője a kiírandó adatot tároló regiszter indexe.

add

Utasítás	Az utasításregiszter bitjei															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
add	0	0	1	1	d	d	d	d	x	x	x	x	y	y	y	y

5-10. táblázat. *Az add utasítás szerkezete*

Utasítás	Utasítás funkciója	Módosított flag-ek
add	RF[IR[11:8]] = (RF[IR[7:4]] + RF[IR[3:0]]) % 2^{16}	Z, N

5-11. táblázat. *Az add utasítás funkcionális definíciója*

Az *add* utasítás két általános célú regiszter tartalmát összegzi, az összeadás eredményét a célregiszterben tárolja. Az utasítás a túl- és alulcsordulást nem kezeli explicit módon. Ha szükséges, azok az operandusok és az eredmény előjelének vizsgálatával detektálhatók. Az 5-12. táblázat néhány példát mutat az *add* utasítás viselkedésére vonatkozóan.

ASM	Utasításregiszter	Előzmények	Utasítás eredménye

`i2rfl $2, 0x03`
`i2rfl $3, 0x04`
`add $4, $2, $3`
	`0x0203 (0000_0010_0000_0011)`
	`0x0304 (0000_0011_0000_0100)`
	`0x3423 (0011_0100_0010_0011)`
		-	`RF[2]: 0x0003, Z: 0, N: 0`
			`RF[3]: 0x0004, Z: 0, N: 0`
			`RF[4]: 0x0007, Z: 0, N: 0`

`add $3, $1, $2`	`0x3312 (0011_0011_0001_0010)`	`RF[1]: 0xFFFF` `RF[2]: 0x0001`	`RF[3]: 0x0000, Z: 1, N: 0`
`add $15, $7, $8`	`0x3F78 (0011_1111_0111_1000)`	`RF[7]: 0x7FFF` `RF[8]: 0x0001`	`RF[15]: 0x8000, Z: 0, N: 1`
`add $10, $5, $6`	`0x3A56 (0011_1010_0101_0110)`	`RF[5]: 0x8000` `RF[6]: 0xFFFF`	`RF[10]: 0x7FFF, Z: 0, N: 0`

5-12. táblázat. *Példák az add utasítás használatára*

sub

Utasítás	Az utasításregiszter bitjei															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sub	0	1	0	0	d	d	d	d	x	x	x	x	y	y	y	y

5-13. táblázat. *A sub utasítás szerkezete*

Utasítás	Utasítás funkciója	Módosított flag-ek
sub	RF[IR[11:8]] = (RF[IR[7:4]] - RF[IR[3:0]]) % 2^{16}	Z, N

5-14. táblázat. *A sub utasítás funkcionális definíciója*

A *sub* utasítás két általános célú regiszter tartalmát vonja ki egymásból, a kivonás eredményét a célregiszterben tárolja. Az utasítás a túl- és alulcsordulást nem kezeli explicit módon. Ha szükséges, azok az operandusok és az eredmény előjelének vizsgálatával detektálhatók. Az 5-15. táblázat néhány példát mutat a *sub* utasítás viselkedésére vonatkozóan.

ASM	Utasításregiszter	Előzmények	Utasítás eredménye

`i2rfl $2, 0x03`
`i2rfl $3, 0x04`
`sub $4, $2, $3`
	`0x0203 (0000_0010_0000_0011)`
	`0x0304 (0000_0011_0000_0100)`
	`0x4423 (0100_0100_0010_0011)`
		-	`RF[2]: 0x0003, Z: 0, N: 0`
			`RF[3]: 0x0004, Z: 0, N: 0`
			`RF[4]: 0xFFFF, Z: 0, N: 1`

`sub $3, $1, $2`	`0x4312 (0100_0011_0001_0010)`	`RF[1]: 0xFFFF` `RF[2]: 0xFFFF`	`RF[3]: 0x0000, Z: 1, N: 0`
`sub $15, $7, $8`	`0x4F78 (0100_1111_0111_1000)`	`RF[7]: 0x8000` `RF[8]: 0x0001`	`RF[15]: 0x7FFF, Z: 0, N: 0`
`sub $10, $5, $6`	`0x4A56 (0100_1010_0101_0110)`	`RF[5]: 0x8000` `RF[6]: 0x8001`	`RF[10]: 0xFFFF, Z: 0, N: 1`

5-15. táblázat. *Példák a sub utasítás használatára*

and

Utasítás	Az utasításregiszter bitjei															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
and	0	1	0	1	d	d	d	d	x	x	x	x	y	y	y	y

5-16. táblázat. *Az and utasítás szerkezete*

Utasítás	Utasítás funkciója	Módosított flag-ek
and	RF[IR[11:8]] = RF[IR[7:4]] AND RF[IR[3:0]]	Z, N

5-17. táblázat. *Az and utasítás funkcionális definíciója*

Az *and* utasítás két általános célú regiszter között végez bitenkénti *AND* műveletet, amelynek eredményét a célregiszterben tárolja. Az 5-18. táblázat néhány példát mutat az *and* utasítás viselkedésére vonatkozóan.

ASM	Utasításregiszter	Előzmények	Utasítás eredménye

`i2rfl $2, 0xF0`
`i2rfl $3, 0x92`
`and $4, $2, $3`
	`0x02F0 (0000_0010_1111_0000)`
	`0x0392 (0000_0011_1001_0010)`
	`0x5423 (0101_0100_0010_0011)`
		-	`RF[2]: 0x00F0, Z: 0, N: 0`
			`RF[3]: 0x0092, Z: 0, N: 0`
			`RF[4]: 0x0090, Z: 0, N: 0`

`and $3, $1, $2`	`0x5312 (0101_0011_0001_0010)`	`RF[1]: 0x0FFF` `RF[2]: 0xFFF0`	`RF[3]: 0x0FF0, Z: 0, N: 0`
`and $15, $7, $8`	`0x5F78 (0101_1111_0111_1000)`	`RF[7]: 0x8000` `RF[8]: 0x0001`	`RF[15]: 0x0000, Z: 1, N: 0`
`and $10, $5, $6`	`0x5A56 (0101_1010_0101_0110)`	`RF[5]: 0x8000` `RF[6]: 0x8001`	`RF[10]: 0x8000, Z: 0, N: 1`

5-18. táblázat. *Példák az and utasítás használatára*

or

Utasítás	Az utasításregiszter bitjei															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
or	0	1	1	0	d	d	d	d	x	x	x	x	y	y	y	y

5-19. táblázat. *Az or utasítás szerkezete*

Utasítás	Utasítás funkciója	Módosított flag-ek
or	RF[IR[11:8]] = RF[IR[7:4]] OR RF[IR[3:0]]	Z, N

5-20. táblázat. *Az or utasítás funkcionális definíciója*

Az *or* utasítás két általános célú regiszter között végez bitenkénti *OR* műveletet, amelynek eredményét a célregiszterben tárolja. Az 5-21. táblázat néhány példát mutat az *or* utasítás viselkedésére vonatkozóan.

ASM	Utasításregiszter	Előzmények	Utasítás eredménye

`i2rfl $2, 0xF0`
`i2rfl $3, 0x92`
`or $4, $2, $3`
	`0x02F0 (0000_0010_1111_0000)`
	`0x0392 (0000_0011_1001_0010)`
	`0x6423 (0110_0100_0010_0011)`
		-	`RF[2]: 0x00F0, Z: 0, N: 0`
			`RF[3]: 0x0092, Z: 0, N: 0`
			`RF[4]: 0x00F2, Z: 0, N: 0`

`or $3, $1, $2`	`0x6312 (0110_0011_0001_0010)`	`RF[1]: 0x0FFF` `RF[2]: 0xFFF0`	`RF[3]: 0xFFFF, Z: 0, N: 1`
`or $15, $7, $8`	`0x6F78 (0110_1111_0111_1000)`	`RF[7]: 0x8000` `RF[8]: 0x0001`	`RF[15]: 0x8001, Z: 0, N: 1`
`or $10, $5, $6`	`0x6A56 (0110_1010_0101_0110)`	`RF[5]: 0x8030` `RF[6]: 0x8001`	`RF[10]: 0x8031, Z: 0, N: 1`

5-21. táblázat. *Példák az or utasítás használatára*

xor

Utasítás	Az utasításregiszter bitjei															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xor	0	1	1	1	d	d	d	d	x	x	x	x	y	y	y	y

5-22. táblázat. *Az xor utasítás szerkezete*

Utasítás	Utasítás funkciója	Módosított flag-ek
xor	RF[IR[11:8]] = RF[IR[7:4]] XOR RF[IR[3:0]]	Z, N

5-23. táblázat. *Az xor utasítás funkcionális definíciója*

Az *xor* utasítás két általános célú regiszter között végez bitenkénti *XOR* műveletet, amelynek eredményét a célregiszterben tárolja. Az 5-24. táblázat néhány példát mutat az *xor* utasítás viselkedésére vonatkozóan.

ASM	Utasításregiszter	Előzmények	Utasítás eredménye

`i2rfl $2, 0xF0`
`i2rfl $3, 0x92`
`xor $4, $2, $3`
	`0x02F0 (0000_0010_1111_0000)`
	`0x0392 (0000_0011_1001_0010)`
	`0x7423 (0111_0100_0010_0011)`
		`-`	`RF[2]: 0x00F0, Z: 0, N: 0`
			`RF[3]: 0x0092, Z: 0, N: 0`
			`RF[4]: 0x0062, Z: 0, N: 0`

`xor $3, $1, $2`	`0x7312 (0111_0011_0001_0010)`	`RF[1]: 0x0FFF` `RF[2]: 0xFFF0`	`RF[3]: 0xF00F, Z: 0, N: 1`
`xor $15, $7, $8`	`0x7F78 (0111_1111_0111_1000)`	`RF[7]: 0x8000` `RF[8]: 0x0001`	`RF[15]: 0x8001, Z: 0, N: 1`
`xor $10, $5, $6`	`0x7A56 (0111_1010_0101_0110)`	`RF[5]: 0x8030` `RF[6]: 0x8001`	`RF[10]: 0x0031, Z: 0, N: 0`

5-24. táblázat. *Példák az xor utasítás használatára*

shl

Utasítás	Az utasításregiszter bitjei															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
shl	1	0	0	0	d	d	d	d	x	x	x	x	y	y	y	y

5-25. táblázat. *Az shl utasítás szerkezete*

Utasítás	Utasítás funkciója	Módosított flag-ek
shl	RF[IR[11:8]] = RF[IR[7:4]] SHL unsigned(RF[IR[3:0]])	Z, N

5-26. táblázat. *Az shl utasítás funkcionális definíciója*

Az *shl (shift left)* utasítás az *X* operandus értékét az *Y* operandus értékének (előjel nélküli egész) megfelelő mértékben balra lépteti és az eredményt a célregiszterben tárolja. A művelet az eredményregiszter alsó helyiértékeit 0 bitekkel tölti fel. Az 5-27. táblázat néhány példát mutat az *shl* utasítás viselkedésére vonatkozóan.

ASM	Utasításregiszter	Előzmények	Utasítás eredménye

`i2rfl $2, 0xF0`
`i2rfl $3, 0x02`
`shl $4, $2, $3`
	`0x02F0 (0000_0010_1111_0000)`
	`0x0392 (0000_0011_0000_0010)`
	`0x8423 (1000_0100_0010_0011)`
		`-`	`RF[2]: 0x00F0, Z: 0, N: 0`
			`RF[3]: 0x0002, Z: 0, N: 0`
			`RF[4]: 0x03C0, Z: 0, N: 0`

`shl $3, $1, $2`	`0x8312 (1000_0011_0001_0010)`	`RF[1]: 0x0FF7` `RF[2]: 0x0003`	`RF[3]: 0x7FB8, Z: 0, N: 0`
`shl $15, $7, $8`	`0x8F78 (1000_1111_0111_1000)`	`RF[7]: 0x8000` `RF[8]: 0x0001`	`RF[15]: 0x0000, Z: 1, N: 0`
`shl $10, $5, $6`	`0x8A56 (1000_1010_0101_0110)`	`RF[5]: 0x8031` `RF[6]: 0x000A`	`RF[10]: 0xC400, Z: 0, N: 1`

5-27. táblázat. *Példák az shl utasítás használatára*

shr

Utasítás	Az utasításregiszter bitjei															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
shr	1	0	0	1	d	d	d	d	x	x	x	x	y	y	y	y

5-28. táblázat. *Az shr utasítás szerkezete*

Utasítás	Utasítás funkciója	Módosított flag-ek
shr	RF[IR[11:8]] = RF[IR[7:4]] SHR unsigned(RF[IR[3:0]])	Z, N

5-29. táblázat. *Az shr utasítás funkcionális definíciója*

Az *shr (shift right)* utasítás az *X* operandus értékét az *Y* operandus értékének (előjel nélküli egész) megfelelő mértékben jobbra lépteti és az eredményt a célregiszterben tárolja. A művelet az eredményregiszter felső helyiértékeit 0 bitekkel tölti fel. Az 5-30. táblázat néhány példát mutat az *shr* utasítás viselkedésére vonatkozóan.

ASM	Utasításregiszter	Előzmények	Utasítás eredménye

`i2rfl $2, 0xF0`
`i2rfl $3, 0x02`
`shr $4, $2, $3`
	`0x02F0 (0000_0010_1111_0000)`
	`0x0392 (0000_0011_0000_0010)`
	`0x9423 (1001_0100_0010_0011)`
		-	`RF[2]: 0x00F0, Z: 0, N: 0`
			`RF[3]: 0x0002, Z: 0, N: 0`
			`RF[4]: 0x003C, Z: 0, N: 0`

`shr $3, $1, $2`	`0x9312 (1001_0011_0001_0010)`	`RF[1]: 0x0FF7` `RF[2]: 0x0003`	`RF[3]: 0x01FE, Z: 0, N: 0`
`shr $15, $7, $8`	`0x9F78 (1001_1111_0111_1000)`	`RF[7]: 0x8000` `RF[8]: 0x0001`	`RF[15]: 0x4000, Z: 0, N: 0`
`shr $10, $5, $6`	`0x9A56 (1001_1010_0101_0110)`	`RF[5]: 0x8031` `RF[6]: 0x000A`	`RF[10]: 0x0020, Z: 0, N: 0`

5-30. táblázat. *Példák az shr utasítás használatára*

bnz

Utasítás	Az utasításregiszter bitjei															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
bnz	1	0	1	0	i	i	i	i	i	i	i	i	i	i	i	i

5-31. táblázat. *A bnz utasítás szerkezete*

Utasítás	Utasítás funkciója	Módosított flag-ek
bnz	if (Z = 0) then IP = IP + signed(IR[11:0]) else IP = IP+1	-

5-32. táblázat. *A bnz utasítás funkcionális definíciója*

A *bnz (branch not zero)* utasítás a *Z* állapotjelző értékétől függő feltételes ugró utasítás. Ha a *Z* állapotjelző értéke a *bnz* utasítás végrehajtásakor logikai alacsony, vagyis az ugró utasítás előtt végrehajtott utasítás eredménye NEM nulla volt, akkor a *bnz* utasítás az utasításmutató regiszter *(IP)* értékét az utasításban megadott konstanssal növeli. Az utasításban megadott 12-bites konstanst a processzor kettes-komplemens kódolású, előjeles egészként értelmezi (az *IP*-vel való összegzés előtt 16 bitre kiterjeszti a konstans előjelét), így az ugrás mértéke negatív irányban (visszaugrás) 2048, pozitív irányban (előre ugrás) 2047 lehet. Ha nagyobb ugrásra van szükség, az ugrótáblák

implementálásával lehetséges[198]. Ha a *Z* állapotjelző értéke a *bnz* utasítás végrehajtásakor logikai magas, akkor az utasításmutató értéke eggyel nő. Az 5-33. táblázat néhány példát mutat a *bnz* utasítás viselkedésére vonatkozóan.

ASM	Utasításregiszter	Előzmények	Utasítás eredménye
`shr $3, $1, $2` `@0x0011: bnz 3`	`0x9312 (1001_0011_0001_0010)` `0xA003 (1010_0000_0000_0011)`	`RF[1]: 0x0FF7` `RF[2]: 0x0003`	`RF[3]: 0x01FE, Z: 0, N: 0` `IP: 0x0014`
`shr $3, $1, $2` `@0x0011: bnz -2`	`0x9312 (1001_0011_0001_0010)` `0xAFFE (1010_1111_1111_1110)`	`RF[1]: 0x0FF7` `RF[2]: 0x0003`	`RF[3]: 0x01FE, Z: 0, N: 0` `IP: 0x000F`
`shr $3, $1, $2` `@0x0011: bnz 3`	`0x9312 (1001_0011_0001_0010)` `0xA003 (1010_0000_0000_0011)`	`RF[1]: 0x0FF7` `RF[2]: 0x0020`	`RF[3]: 0x0000, Z: 1, N: 0` `IP: 0x0012`

5-33. táblázat. *Példák a bnz utasítás használatára*

bnn

Utasítás	Az utasításregiszter bitjei															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
bnn	1	0	1	1	i	i	i	i	i	i	i	i	i	i	i	i

5-34. táblázat. *A bnn utasítás szerkezete*

Utasítás	Utasítás funkciója	Módosított flag-ek
bnn	if (N = 0) then IP = IP + signed(IR[11:0]) else IP = IP+1	-

5-35. táblázat. *A bnn utasítás funkcionális definíciója*

A *bnn (branch not negative)* utasítás az *N* állapotjelző értékétől függő feltételes ugró utasítás. Ha az *N* állapotjelző értéke a *bnn* utasítás végrehajtásakor logikai alacsony, vagyis az ugró utasítás előtt végrehajtott utasítás eredménye NEM negatív volt, akkor a *bnn* utasítás az utasításmutató regiszter *(IP)* értékét az utasításban megadott konstanssal növeli. Az utasításban megadott 12-bites konstanst a mikroprocesszor kettes-komplemens kódolású, előjeles egészként értelmezi (az *IP*-vel való összegzés előtt 16 bitre kiterjeszti a konstans előjelét), így az ugrás mértéke negatív irányban (visszaugrás) 2048, pozitív irányban (előre ugrás) 2047 lehet. Ha nagyobb ugrásra van szükség, az ugrótáblák implementálásával lehetséges. Ha az *N* állapotjelző értéke a *bnn* utasítás végrehajtásakor logikai magas, akkor az utasításmutató értéke eggyel nő. Az 5-36. táblázat néhány példát mutat a *bnn* utasítás viselkedésére vonatkozóan.

ASM	Utasításregiszter	Előzmények	Utasítás eredménye
`shl $3, $1, $2` `@0x0011: bnn 3`	`0x8312 (1000_0011_0001_0010)` `0xB003 (1011_0000_0000_0011)`	`RF[1]: 0x0FF7` `RF[2]: 0x0004`	`RF[3]: 0xFF70, Z: 0, N: 1` `IP: 0x0012`
`shl $3, $1, $2` `@0x0011: bnn 3`	`0x8312 (1000_0011_0001_0010)` `0xB003 (1011_0000_0000_0011)`	`RF[1]: 0x0FF7` `RF[2]: 0x0003`	`RF[3]: 0x7FB8, Z: 0, N: 0` `IP: 0x0014`
`shl $3, $1, $2` `@0x0011: bnn -2`	`0x8312 (1000_0011_0001_0010)` `0xBFFE (1011_1111_1111_1110)`	`RF[1]: 0x0FF7` `RF[2]: 0x0003`	`RF[3]: 0x7FB8, Z: 0, N: 0` `IP: 0x000F`

5-36. táblázat. *Példák a bnn utasítás használatára*

[198] Az ugró utasítás egy olyan kódrészletre ugrik, ahol további ugró utasítások találhatók.

jmp

Utasítás	Az utasításregiszter bitjei															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
jmp	1	1	0	0	i	i	i	i	i	i	i	i	i	i	i	i

5-37. táblázat. *A jmp utasítás szerkezete*

Utasítás	Utasítás funkciója	Módosított flag-ek
jmp	IP = IP + signed(IR[11:0])	-

5-38. táblázat. *A jmp utasítás funkcionális definíciója*

A *jmp (jump)* utasítás feltétel nélküli ugrást valósít meg. A *jmp* utasítás az utasításmutató regiszter *(IP)* értékét az utasításban megadott konstanssal növeli. Az utasításban megadott 12-bites konstanst a mikroprocesszor kettes-komplemens kódolású, előjeles egészként értelmezi (az *IP*-vel való összegzés előtt 16 bitre kiterjeszti a konstans előjelét), így az ugrás mértéke negatív irányban (visszaugrás) 2048, pozitív irányban (előre ugrás) 2047 lehet. Ha nagyobb ugrásra van szükség, az ugrótáblák implementálásával lehetséges. Az 5-39. táblázat néhány példát mutat a *jmp* utasítás viselkedésére vonatkozóan.

ASM	Utasításregiszter	Előzmények	Utasítás eredménye
`@0x0011: jmp 3`	`0xC003 (1100_0000_0000_0011)`	`-`	`IP: 0x0014`
`@0x0011: jmp -2`	`0xCFFE (1100_1111_1111_1110)`	`-`	`IP: 0x000F`

5-39. táblázat. *Példák a jmp utasítás használatára*

call

Utasítás	Az utasításregiszter bitjei															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
call	1	1	0	1	i	i	i	i	i	i	i	i	i	i	i	i

5-40. táblázat. *A call utasítás szerkezete*

Utasítás	Utasítás funkciója	Módosított flag-ek
call	STACK[SP] = EI, Z, N, IP+1 SP = SP+1 EI = 0 IP = IP + signed(IR[11:0])	EI

5-41. táblázat. *A call utasítás funkcionális definíciója*

A *call* utasítás olyan, feltétel nélküli ugró utasítás, amely az utasításmutató regiszter *(IP)* ugrási címre való beállítása előtt a mikroprocesszor mag belső állapotát (az állapotjelzők értékét és az utasításmutató értékét) egy belső veremtárba menti. Ezt a folyamatot kontextusváltásnak nevezzük. Ily módon a processzor tulajdonképpen „megjegyzi" az ugrás előtti állapotát, amely később a *ret* utasítással visszaállítható. Az 5-42. táblázat néhány példát mutat a *call* utasítás viselkedésére vonatkozóan.

ASM	Utasításregiszter	Előzmények	Utasítás eredménye

`@0x0034: call 21`	`0xD015 (1101_0000_0001_0101)`	`EI: 1`
		`Z: 1`
		`N: 0`
		`SP: 2`
			`STACK[2]: 0x60035 (EI, Z, N, IP+1)`
			`SP: 3`
			`EI: 0`
			`IP: 21`

`@0x0012: call 45`	`0xD02D (1101_0000_0010_1101)`	`EI: 0`
		`Z: 1`
		`N: 0`
		`SP: 1`
			`STACK[1]: 0x20013 (EI, Z, N, IP+1)`
			`SP: 2`
			`EI: 0`
			`IP: 45`

5-42. táblázat. *Példák a call utasítás használatára*

ret

Utasítás	Az utasításregiszter bitjei															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ret	1	1	1	0	-	-	-	-	-	-	-	-	-	-	-	-

5-43. táblázat. *A ret utasítás szerkezete*

Utasítás	Utasítás funkciója	Módosított flag-ek
ret	SP = SP-1 EI, Z, N, IP = STACK[SP]	EI, Z, N

5-44. táblázat. *A ret utasítás funkcionális definíciója*

A *ret (return)* utasítással a mikroprocesszor legutóbb elmentett állapota állítható vissza. Az utasítás végrehajtásakor a processzor kiolvassa a belső állapotveremtár legutóbb elmentett bejegyzését, az ott tárolt információ alapján visszaállítja az állapotjelzők értékét, majd az utasításmutatót is a veremben tárolt értékre állítja. Ha a legutóbbi kontextusváltást egy *call* utasítás meghívása váltotta ki, akkor a végrehajtás a *call* utasítást követő gépi utasítással folytatódik. Ha a legutóbbi kontextusváltás egy megszakításkérés hatására történt, akkor a végrehajtás azzal a gépi utasítással folytatódik, amelynek végrehajtása a megszakítás kiszolgálása miatt nem kezdődhetett el. Az 5-45. táblázat néhány példát mutat a *ret* utasítás viselkedésére vonatkozóan.

ASM	Utasításregiszter	Előzmények	Utasítás eredménye

`@0x0011: ret`	`0xE000 (1110_0000_0000_0000)`	`EI: 0`
		`Z: 1`
		`N: 0`
		`SP: 2`
		`STACK[1]: 0x60035`
			`SP: 1`
			`EI: 1`
			`Z: 1`
			`N: 0`
			`IP: 0x0035`

`@0x0022: ret`	`0xE000 (1110_0000_0000_0000)`	`EI: 1`
		`Z: 0`
		`N: 1`
		`SP: 4`
		`STACK[3]: 0x20042`
			`SP: 3`
			`EI: 0`
			`Z: 1`
			`N: 0`
			`IP: 0x0042`

5-45. táblázat. *Példák a ret utasítás használatára*

ei

Utasítás	Az utasításregiszter bitjei															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ei	1	1	1	1	-	-	-	-	-	-	-	-	-	-	-	i

5-46. táblázat. *Az ei utasítás szerkezete*

Utasítás	Utasítás funkciója	Módosított flag-ek
ei	EI = IR[0]	EI

5-47. táblázat. *Az ei utasítás funkcionális definíciója*

Az *ei (enable interrupt)* utasítás segítségével állítható be a megszakításokat engedélyező/tiltó *EI* állapotjelző értéke. Az 5-48. táblázat néhány példát mutat az *ei* utasítás viselkedésére vonatkozóan.

ASM	Utasításregiszter	Előzmények	Utasítás eredménye
`ei 1`	`0xF001 (1111_0000_0000_0001)`	`EI: 0`	`EI: 1`
`ei 0`	`0xF000 (1111_0000_0000_0000)`	`EI: 1`	`EI: 0`

5-48. táblázat. *Példák az ei utasítás használatára*

5.1.1.2. Pszeudoutasítások

A pszeudoutasítások a *CPU*-hoz tartozó assembler fordítóprogram által definiált, absztrakt utasítások, amelyek a natív utasítások egyedileg felparaméterezett változatai, vagy azok sorozatai. Az assembly nyelvű forráskód elemzésének első lépéseként ezeket az utasításokat az assembler a nekik megfelelő natív utasításokkal vagy azok sorozatával helyettesíti, majd az így kapott, immár kizárólag a *CPU* utasításdekódere által is ismert utasításokból álló programból állítja elő a programtárba betölthető bájtkódot. Az 5-49. táblázat 32 pszeudoutasítást mutat be.

Szintaxis	Implementáció	Funkció
`inc $D, $R`	`i2rfl $15, 1` `add $D, $R, $15`	`$D = $R + 1`
`dec $D, $R`	`i2rfl $15, 1` `sub $D, $R, $15`	`$D = $R - 1`

`i2rfh $D, 0xIJ`	`i2rfl $D, 0xIJ`
	`i2rfl $15, 8`
	`shl $D, $D, $15`
		`$D = 0xIJ00`

`i2rf $D, 0xIJ, 0xKL`	`i2rfl $D, 0xIJ`
	`i2rfl $15, 8`
	`shl $D, $D, $15`
	`i2rfl $15, 0xKL`
	`or $D, $D, $15`
		`$D = 0xIJKL`

`nop`	`jmp 1`	`no operation`

`addi $R, $IJ, $KL`	`i2rfl $15, 0xIJ`
	`i2rfl $14, 8`
	`shl $15, $15, $14`
	`i2rfl $14, 0xKL`
	`or $15, $15, $14`
	`add $R, $R, $15`
		`$R = $R + 0xIJKL`

Szintaxis	Implementáció	Funkció

`subi $R, 0xIJ, 0xKL`	`i2rfl $15, 0xIJ`
	`i2rfl $14, 8`
	`shl $15, $15, $14`
	`i2rfl $14, 0xKL`
	`or $15, $15, $14`
	`sub $R, $R, $15`
		`$R = $R – 0xIJKL`

`andi $R, 0xIJ, 0xKL`	`i2rfl $15, 0xIJ`
	`i2rfl $14, 8`
	`shl $15, $15, $14`
	`i2rfl $14, 0xKL`
	`or $15, $15, $14`
	`and $R, $R, $15`
		`$R = $R AND 0xIJKL`

`ori $R, 0xIJ, 0xKL`	`i2rfl $15, 0xIJ`
	`i2rfl $14, 8`
	`shl $15, $15, $14`
	`i2rfl $14, 0xKL`
	`or $15, $15, $14`
	`or $R, $R, $15`
		`$R = $R OR 0xIJKL`

`xori $R, 0xIJ, 0xKL`	`i2rfl $15, 0xIJ`
	`i2rfl $14, 8`
	`shl $15, $15, $14`
	`i2rfl $14, 0xKL`
	`or $15, $15, $14`
	`xor $R, $R, $15`
		`$R = $R XOR 0xIJKL`

`shli $R, 0xIJ`	`i2rfl $15, 0xIJ` `shl $R, $R, $15`	`$R = $R << unsigned(0xIJ) (logikai shift)`
`shri $R, 0xIJ`	`i2rfl $15, 0xIJ` `shr $R, $R, $15`	`$R = $R >> unsigned(0xIJ) (logikai shift)`

`shra $D, $R1, $R2`	`i2rfl $15, 0x80`
	`i2rfl $14, 8`
	`shl $15, $15, $14`
	`and $15, $15, $R1`
	`bnz 3`
	`shr $D, $R1, $R2`
	`jmp 8`
	`i2rfl $15, 0xFF`
	`shl $15, $15, $14`
	`i2rfl $14, 0xFF`
	`or $15, $15, $14`
	`xor $14, $R1, $15`
	`shr $D, $14, $R2`
	`xor $D, $D, $15`
		`$D = $R1 >> unsigned($R2) (aritmetikai shift)`

`shrai $D, $R1, 0xIJ`	`i2rfl $15, 0x80`
	`i2rfl $14, 8`
	`shl $15, $15, $14`
	`and $15, $15, $R1`
	`bnz 4`
	`i2rfl $15, 0xIJ`
	`shr $D, $R1, $15`
	`jmp 9`
	`i2rfl $15, 0xFF`
	`shl $15, $15, $14`
	`i2rfl $14, 0xFF`
	`or $15, $15, $14`
	`xor $14, $R1, $15`
	`i2rfl $D, 0xIJ`
	`shr $D, $14, $D`
	`xor $D, $D, $15`
		`$D = $R1 >> unsigned(0xIJ) (aritmetikai shift)`

`bz LABEL`	`bnz 2` `jmp LABEL`	`if ( Z = 1 ) then jump to LABEL` `else jump to next`
`bn LABEL`	`bnn 2` `jmp LABEL`	`if ( N = 1 ) then jump to LABEL` `else jump to next`

Szintaxis	Implementáció	Funkció

`beq $R1, $R2, LABEL`	`sub $15, $R1, $R2`
	`bnz 2`
	`jmp LABEL`
		`if ( $R1 = $R2 ) then jump to LABEL`
		`else jump to next`

`bne $R1, $R2, LABEL`	`sub $15, $R1, $R2` `bnz LABEL`	`if ( $R1 != $R2 ) then jump to LABEL` `else jump to next`

`beqi $R, 0xIJ, LABEL`	`i2rfl $15, 0xIJ`
	`sub $14, $R, $15`
	`bnz 2`
	`jmp LABEL`
		`if ( $R = 0xIJ ) then jump to LABEL`
		`else jump to next`

`bnei $R, 0xIJ, LABEL`	`i2rfl $15, 0xIJ`
	`sub $14, $R, $15`
	`bnz LABEL`
		`if ( $R1 != 0xIJ ) then jump to LABEL`
		`else jump to next`

`not $R`	`i2rfl $15, 0xFF`
	`i2rfl $14, 8`
	`shl $15, $15, $14`
	`i2rfl $14, 0xFF`
	`or $15, $15, $14`
	`xor $R, $R, $15`
		`$R = NOT $R`

`cpy $D, $R`	`i2rfl $15, 0` `add $D, $R, $15`	`$D = $R`
`ldd $D, $R`	`i2rfl $15, 0` `load $D, $R, $15`	`$D = DMEM[$R]`
`std $S, $R`	`i2rfl $15, 0` `store $S, $R, $15`	`DMEM[$R] = $S`

`calleq $R1, $R2, LABEL`	`sub $15, $R1, $R2`
	`bnz 2`
	`call LABEL`
		`if ( $R1 = $R2 ) then call LABEL`
		`else jump to next`

`callne $R1, $R2, LABEL`	`sub $15, $R1, $R2`
	`bnz 2`
	`jmp 2`
	`call LABEL`
		`if ( $R1 != $R2 ) then call LABEL`
		`else jump to next`

`calleqi $R, 0xIJ, LABEL`	`i2rfl $15, 0xIJ`
	`sub $14, $R, $15`
	`bnz 2`
	`call LABEL`
		`if ( $R1 = $IJ ) then call LABEL`
		`else jump to next`

`callnei $R, 0xIJ, LABEL`	`i2rfl $15, 0xIJ`
	`sub $14, $R, $15`
	`bnz 2`
	`jmp 2`
	`call LABEL`
		`if ( $R1 != 0xIJ ) then call LABEL`
		`else jump to next`

`push $R, $SP`	`i2rfl $15, 0`
	`store $R, $SP, $15`
	`i2rfl $15, 0x01`
	`add $SP, $SP, $15`
		`DMEM[$SP] = $R`
		`$SP++`

`pop $R, $SP`	`i2rfl $15, 1`
	`sub $SP, $SP, $15`
	`i2rfl $15, 0`
	`load $R, $SP, $15`
		`$SP--`
		`$R = DMEM[$SP]`

`delay 0xIJ, 0xKL`	`i2rfl $14, 0xIJ`
	`i2rfl $15, 8`
	`shl $14, $14, $15`
	`i2rfl $15, 0xKL`
	`or $14, $14, $15`
	`i2rfl $15, 1`
	`sub $14, $14, $15`
	`bnz -1`
		`Késleltetés #1: 0xIJKL dekrementálása,`
		`amíg az érték 0 nem lesz.`

`delayr $R`	`i2rfl $15, 1`
	`sub $R, $R, $15`
	`bnz -1`
		`Késleltetés #2: $R dekrementálása,`
		`amíg az érték 0 nem lesz.`

5-49. táblázat. *Pszeudoutasítások*

5.1.2. Javasolt regiszterhasználati konvenció

Az assembly nyelvű programok elkészítését jelentősen megkönnyítheti, ha a *CPU* hardverszinten egyenértékű, általános célú regisztereit a különféle assembly nyelvű programokban mindig ugyanarra a célra használjuk. Az egyes regiszterek „szokásos funkcióját” regiszterhasználati konvenciónak hívjuk. Ennek egy lehetséges változatát az 5-50. táblázat szemlélteti.

Regiszter index	Regiszter álneve	Funkció
0...4	*tmp_1* … *tmp_5*	Ideiglenes változók, szerepüket a konkrét alkalmazás határozza meg. Tekinthetők az operatív tár legalsó szintjének.
5	*intr_served*	Állapotjelző, amit a központi megszakításkiszolgáló rutin használ annak számontartására, hogy a legutóbbi megszakításkérést kiszolgálta-e már valamelyik perifériaspecifikus megszakításkiszolgáló rutin (a részleteket lásd 5.1.3. pont).
6	*param_1*	Szubrutinok paramétereinek átadására használt regiszterek. A hívó ezekbe a regiszterekbe helyezi el a szubrutinok paramétereit, a szubrutinok pedig innen olvassák ki azokat (érték szerinti paraméterátadás). Ha egy szubrutinnak több paramétere van, és/vagy azok mérete túl nagy, akkor ezekben a regiszterekben adhatók át a paraméterek memóriacímei, magukat a paramétereket pedig egy külső memóriában kell elhelyeznünk (cím szerinti paraméterátadás).
7	*param_2*	
8	*param_3*	
9	*param_4*	
10	*retvalue*	A szubrutinok visszatérési értékét tároló regiszter. Ha a visszatérési érték nem egyetlen érték, akkor e regiszterben adható vissza a visszatérési érték memóriacíme, a visszatérési értéket pedig a szubrutin egy külső memóriában helyezi el.

11	*par*	**p**eripheral **a**ddress **r**egister: Minden perifériahozzáférés előtt ebbe a regiszterbe töltjük azt az MMIO-címet, ahonnan adatot akarunk olvasni vagy ahová adatot akarunk írni.
		peripheral **d**ata **r**egister: Perifériaírás esetén ebbe a regiszterbe töltjük a kiírandó adatot, perifériaolvasáskor pedig ebben a regiszterben tároljuk el a beolvasott adatot. E két álnevet deklarálva a perifériavezérlőkhöz való hozzáférés a következőképpen nézhet ki:
		`; olvasás művelet:`
		`L_PERIPH_READ: i2rf par, <address H>, <address L>`
		`ldd pdr, par`
		`; írás művelet:`
		`L_PERIPH_WRITE: i2rf par, <address H>, <address L>`
		`i2rf pdr, <data H>, <data L>`
		`std pdr, par`
		ahol
		<address H>: a perifériavezérlő MMIO-címének felső bájtja
		<address L>: a perifériavezérlő MMIO-címének alsó bájtja
		<data H>: a kiírandó adat felső bájtja
		<data L>: a kiírandó adat alsó bájtja

12	*pdr*	
13	*sp*	**s**tack **p**ointer: Ha a processzoros rendszer tartalmaz MMIO memóriamodult (lásd pl. 2.6.1. pont), akkor lehetőség van adat-veremtár létrehozására.[199] Ebben az esetben a veremtárba való írás és az onnan való olvasás könnyen megvalósítható a *push* és *pop* pszeudó-utasítások segítségével. Mindkét utasításnak szüksége van egy regiszterre, ami tulajdonképpen nem más, mint egy, a veremtár tetejére mutató pointer. Az *sp* regiszter kezdeti értékét mindig a használni kívánt memóriaterület elejére kell állítani.
14	*pseudo_1*	Pszeudoutasítások megvalósításához használt regiszterek (lásd 5-1. ábra).
15	*pseudo_2*	

5-50. táblázat. *Javasolt regiszterhasználati konvenció*

[199] Az így létrehozott adatveremtár nem összetévesztendő a *CPU* belső, az utasításkészlet specifikációjában *STACK* néven hivatkozott állapotveremtárral. Az adatveremtárba a felhasználó bármilyen információt elmenthet, míg az állapotveremtárhoz nincs közvetlen hozzáférése, azt a *CPU* autonóm módon használja a szubrutinhívások végrehajtása során, állapotának elmentésére és visszaállítására.

A regiszterhasználati konvenciók lehetővé teszik a regularitás alapelvének kiterjesztését az assembly programok elkészítésére. A konvenciók használatával az assembly programok gyakran előforduló részletei (pl. perifériavezérlő áramkör írása/olvasása vagy szubrutinhívás) egységes formában valósíthatók meg. Ennek akkor van különösen nagy jelentősége, ha az assembly programot nem kézzel, hanem egy magas szintű programozási nyelven (pl. C/C++ stb.) megfogalmazott specifikáció alapján gépi úton, ún. fordítóprogrammal állítjuk elő. A fordítóprogramok a magas szintű programozási nyelvek szemantikai konstrukcióit (változók kezelése, függvényhívás stb.) azok assembly nyelvű sablonjaiból kiindulva generálják, ami csak úgy lehetséges, ha ezek a sablonok egyazon célra mindig ugyanazt az általános célú regisztert használják.

5.1.3. Központi megszakításkiszolgáló rutin

A *CPU* a 2.5. pontban tárgyalt megszakításvezérlővel együtt válik alkalmassá több megszakítási forrás kezelésére. Ilyenkor a központi megszakításkiszolgáló rutin feladata a megszakításkérés forrásának azonosítása, és az annak megfelelő perifériaspecifikus megszakításkiszolgáló rutin hívása. E kötött feladatot végrehajtó assembly kódrészlet egy lehetséges kivitelét mutatja be az 5-1. ábra.

Python – viselkedési modell

```
1  @400: L_MAIN_ISR:  i2rfl    intr_served, 0x00
2                     store    pseudo_1, sp, intr_served
3                     i2rfl    pseudo_1, 1
4                     add      sp, sp, pseudo_1
5                     store    pseudo_2, sp, intr_served
6                     add      sp, sp, pseudo_1
7
8                     push     par, sp
9                     push     pdr, sp
10
11                    i2rf     par, addr_intr_ctrl_src_reg_H, addr_intr_ctrl_src_reg_L
12                    ldd      pdr, par
13
14                    calleqi  pdr, 0x00, L_ISR_UART
15                    beqi     intr_served, 0x01, L_RET_MAIN_ISR
16
17                    calleqi  pdr, 0x01, L_ISR_SPI
18                    beqi     intr_served, 0x01, L_RET_MAIN_ISR
19
20                    calleqi  pdr, 0x02, L_ISR_FIFO
21                    beqi     intr_served, 0x01, L_RET_MAIN_ISR
22
23                    L_RET_MAIN_ISR: i2rfl  intr_served, 0x00
24
25                                    pop    pdr, sp
26                                    pop    par, sp
27
28                                    i2rfl  pseudo_1, 1
29                                    sub    sp, sp, pseudo_1
30                                    load   pseudo_2, sp, intr_served
31                                    sub    sp, sp, pseudo_1
32                                    load   pseudo_1, sp, intr_served
33                                    ret
```

5-1. ábra. *A központi megszakításkiszolgáló rutin megvalósítására alkalmas sablon*

A példaként bemutatott kódrészlet értelmezésekor a következőket kell figyelembe vennünk:

- A program az 5-50. táblázatban rögzített regiszterhasználati konvenciót követi.
- A megszakításvezérlő megszakítási forrást azonosító regiszterének *(source_index)* memóriacíme az *addr_intr_ctrl_src_reg_H* és *addr_intr_ctrl_src_reg_L* álneveken keresztül érhető el. Az *_H* utótag a magasabb, az *_L* pedig az alacsonyabb helyiértékű bájtra utal.

- A rendszer tartalmaz egy MMIO memóriamodult (1-13. ábra: *BRAM* RTL-modell: 2.6.1. pont), amelynek egy részét veremtárként használjuk az 5-50. táblázatban rögzített regisze terhasználati konvenciónak megfelelően. Az *sp* regisztert a főprogram ennek megfelelően inicializálja.
- A rendszerben 3 megszakítási forrás van:
  - UART adóvevő a megszakításvezérlő nullás indexű bemenetére kapcsolva. A hozzá tartozó perifériaspecifikus megszakításkiszolgáló rutin belépési pontjának címkéje: *L_ISR_UART*.
  - SPI master egység a megszakításvezérlő egyes indexű bementére kapcsolva. A hozzá tartozó perifériaspecifikus megszakításkiszolgáló rutin belépési pontjának címkéje: *L_ISR_SPI*.
  - FIFO modul a megszakításvezérlő kettes indexű bemenetére kapcsolva. A hozzá tartozó perifériaspecifikus megszakításkiszolgáló rutin belépési pontjának címkéje: *L_ISR_FIFO*.

Az algoritmus működése a következő:

- A szubrutin által használt változók elmentése a veremtárba.
  - 2–6. sor: Mivel előfordulhat, hogy a megszakítás épp egy főprogrambeli, több natív gépi utasításból álló, a *pseudo_1* és *pseudo_2* regisztereket használó pszeudoutasítást „szakított félbe", ezért e két regisztert el kell mentenünk a veremtárba. Mivel maguk a megszakítási rutinok is használhatnak pszeudoutasításokat, amelyek megváltoztatják a *pseudo_1* és *pseudo_2* regiszterek értékét, ezért elsőként e két regisztert kell elmentenünk. Figyeljük meg, hogy e két regiszter veremtárba másolásához nem használható a *push* utasítás, hiszen az maga is pszeudoutasítás, amely használja a két szóban forgó regisztert. Az *intr_served* regiszter használata ebben a kontextusban elsőre furcsának tűnhet, hiszen az ideiglenes regiszterek veremtárba másolásának ehhez a regiszterhez semmi köze nincs. Ebben a lépésben azt használjuk ki, hogy a műveletvégzésnek ebben a pillanatában az *intr_served* regiszter tartalma mindenképpen nulla, vagyis a *store* utasítás által előállított memóriacím éppen az *sp* regiszter tartalmával egyenlő.
  - 8–9. sor: A *par* és a *pdr* regisztereket ugyancsak használja a központi megszakításkiszolgáló rutin, ezért ezeket is a veremtárba kell menteni. E két regiszter esetén már használható a *push* pszeudoutasítás.
- 11–12. sor: A megszakításvezérlő *source_index* regiszterét megvizsgálva megállapítható, hogy a 16 lehetséges megszakítási forrás közül melyik volt az, amelyik kiváltotta a központi megszakításkiszolgáló rutin hívását.
- 14–21. sor: A fenti kódrészletben a *source_index* regiszter értékét a *calleqi* utasítás vizsgálja meg. A vizsgálat kimenetele kétféle lehet:
  - A *calleqi* utasítás meghívja a perifériaspecifikus megszakításkiszolgáló rutint, amely – miután ellátta feladatát – az *intr_served* regiszter értékét 1-re állítja. A perifériaspecifikus megszakításkiszolgáló rutin visszatérésekor a *beqi* utasítás az *intr_served* regisztert megvizsgálva ugrást idéz elő, így a végrehajtás az *L_RET_MAIN_ISR* címkén lévő utasítással folytatódik.

- A *calleqi* utasítás nem hívja meg a perifériaspecifikus megszakításkiszolgáló rutint, így az *intr_served* regiszter értéke 0 marad. A *beqi* utasítás ezért nem fog ugrást előidézni, a megszakításvezérlő *source_index* regiszterének vizsgálata folytatódik a következő *calleqi* utasítással.

Minden megszakítási forráshoz tartozik egy *calleqi-beqi* utasításpár. Ha új megszakítási rutint definiálunk, akkor egy ilyen utasításpár hozzáadásával kell kiegészítenünk a központi megszakításkiszolgáló rutint. A kód többi részét nem kell módosítani.

- 23–33. sor: Az *intr_served* regiszter értékét visszaállítjuk 0-ra, majd a korábban a veremtárba másolt regiszterek értékét visszaállítjuk. Mindezek után a végrehajtás visszatér a főprogramba.

A fenti algoritmus biztosítja, hogy minden megszakításkéréskor csak a megszakítást kiváltó forráshoz tartozó perifériaspecifikus megszakításkiszolgáló rutin fusson le, továbbá, ha valami hiba folytán olyan megszakítási forrás vált ki megszakítást, amelyikhez nem definiáltunk kiszolgáló rutint, akkor egyszerűen ne történjen semmi; a központi megszakításkiszolgáló rutin lefut, de nem hív meg perifériaspecifikus megszakításkiszolgáló rutint.[200]

5.2. Egyszerű soros kommunikációs interfészek

Az 1.2.1. pontban a teljes funkcióbővítő rendszeren átívelő megoldásokat láthatunk arra, hogy miként tud két belső almodul információt cserélni, kommunikálni egymással. Ezek az interfészek fizikailag az FPGA-n belül helyezkednek el, az őket alkotó jelek sokaságát a chipen belüli huzalozási erőforrások (vezetékek és kapcsolóelemek) továbbítják (lásd DRMSz: 7. fejezet). Ez a megoldás – amellett, hogy könnyen tervezhető, egyszerű – az interfész sebessége szempontjából is előnyös, hiszen az egyetlen órajelciklus alatt átvitt bitek számára szinte nincs felső korlát. Egy folytonos üzemben használt, nulla késleltetésű szinkron kézfogásos interfészen, amelyben az általánosított A_i vektor 32-bites, a kommunikációban részt vevő felek órajelének frekvenciája pedig 50 MHz, az interfész átviteli sebessége közel 1,5 Gb/s.

Az egyazon nyomtatott huzalozású lemezre ültetett integrált áramkörök – pl. a funkcióbővítő rendszert tartalmazó FPGA és az azt üzemeltető mikrovezérlő – közötti kommunikáció megvalósításakor sem a viszonylag magas órajel-frekvencia, sem a sok, párhuzamosan futó vezeték nem praktikus. Amellett, hogy a rengeteg vezeték sok helyet foglal, a több tíz MHz ütemű változások a vezetékek késleltetésének egyformaságát is megkövetelik, ami fizikailag nehezen kivitelezhető. Ezek a problémák még hangsúlyosabban jelentkeznek, ha a kommunikáló felek fizikailag egymástól távol, még csak nem is egyazon hordozón helyezkednek el, így összekapcsolásukra kénytelenek vagyunk kábelezést használni.

A fenti problémák okán gyakran használunk ún. soros kommunikációs interfészeket egymástól távol elhelyezkedő alrendszerek között. A „soros” kifejezés arra utal, hogy az információ bitjeit nem páhuzamosan, sok különálló vezetéken, hanem egyazon vezetéken, időben egymás után visszük át. A különféle soros átviteli szabványok ennek az alapötletnek a megvalósítására sokféle

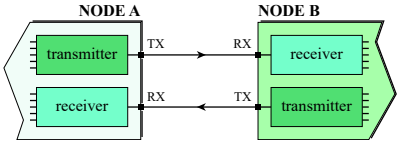
[200] Az 5-1. ábrán látható algoritmusban megvalósítható egy további vizsgálat: ha az *L_RET_MAIN_ISR* címkével ellátott utasítás elérésekor az *intr_served* regiszter értéke nulla, akkor olyan megszakítási forrás küldött megszakításkérést, amelyikhez nem definiáltunk kiszolgáló rutint. Szükség esetén a 23. sor elé beszúrható a hiba kezeléséért felelős kódrészlet.

megoldást kínálnak. Ezek közül e fejezetben a beágyazott rendszerekben leggyakrabban alkalmazott, kissebességű módszereket, az UART-ot (**U**niversal **A**synchronous **R**eceiver/**T**ransmitter), az SPI-t (**S**erial **P**eripheral **I**nterface) és az I²C-t (**I**nter-**I**ntegrated **C**ircuit) mutatjuk be a hozzájuk tartozó, 4.3. fejezetbeli példaáramkörök működésének – vagyis a soros/párhuzamos átalakításnak – megértését lehetővé tévő mélységig.

A sokat sejtető „kissebességű" jelző egyrészt arra utal, hogy az UART/SPI/I²C módszerek tipikusan nem biztosítanak nagy átviteli sebességet,[201] másrészt pedig arra, hogy a soros adatátviteli technikáknak van egy nagysebességű területe is. Ezeket a soros/párhuzamos átalakítást végző, tehát a 4.3. fejezetben tárgyaltakhoz hasonló funkciót ellátó áramköröket összefoglaló néven SER-DES (**Ser**ializer-**Des**erializer) áramköröknek hívjuk.

5.2.1. UART

Az UART (**U**niversal **A**synchronous **R**eceiver/**T**ransmitter) protokoll két vezetéket definiál két eszköz kétirányú, pont-pont kapcsolatának megvalósításához (lásd 5-2. ábra).



5-2. ábra. *Kétirányú UART-kapcsolat*

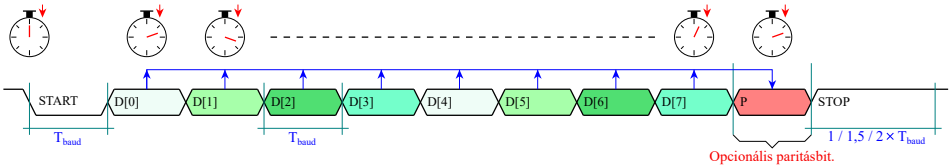
E kapcsolatban a két fél egyenrangú, a két irány egymástól független, bármelyik fél küldhet adatot a másiknak.[202] A két vezeték közül az egyik az egyik átviteli irányt, a másik pedig a másikat valósítja meg. A megoldás tehát teljesen szimmetrikus, az „adó" áramkört (transmitter, párhuzamos-soros átalakító) és a „vevő" áramkört (receiver, soros-párhuzamos átalakító) is mindkét fél tartalmazza. Mindegyik soros kommunikációs megoldásra igaz, hogy az átviendő információ (pl. egy 8-bites adatkeret) bitjeit az adó valamilyen ütemezési szabályrendszer szerint, egymás után kapcsolja rá egy kétállapotú fizikai vonalra. A vevő feladata az, hogy – az ütemezési szabályrendszer ismeretében – a megfelelő pillanatokban mintavételezze ezt a vonalat, és az egymást követő, egy-egy bitet reprezentáló mintákból visszaállítsa a küldött bájtot. A különböző soros interfész-protokollok közötti legjelentősebb eltérés a mintavételezés pillanatának megállapításában, az ún. bitszinkronizálásban rejlik. A **szinkron** soros átviteli módszerek – mint amilyen az SPI és az I²C – a mintavétel

[201] Az elérhető legnagyobb adatátviteli sebességet e három megoldás esetén elsősorban a fizikai réteg kialakítása határozza meg. Egy UART-kapcsolat akár több 10 Mb/s sebességű is lehet differenciális (pl. RS-485) vonali meghajtókkal kombinálva, az SPI pedig akár nem-differenciális átvitel esetén is elérheti ezt a sebességet szinkron jellegének köszönhetően. A „kissebességű" kifejezés tehát kifejezetten a nagysebességű átviteli módszerek kontextusában értelmezendő, amelyek tipikus sebessége a 100 Mb/s – 10 Gb/s tartományba esik.

[202] Azt persze valami módon biztosítani kell, hogy az adat elküldésekor a fogadó fél készen álljon annak fogadására, különben adatvesztés lépne fel. Azt a mechanizmust, amely gondoskodik az adatvesztés elkerüléséről, átvitelvezérlésnek hívjuk. Az UART protokoll tartalmazza az ún. hardver-átvitelvezérlés fogalmát, amely az adó adási szándékát és a vevő fogadókészségét egy-egy külön vezetékkel jelzi (**R**equest **T**o **S**end: RTS, **C**lear **T**o **S**end: CTS), a legegyszerűbbként aposztrofált kivitel azonban ezeket a vezetékeket nem tartalmazza. Az adatvesztés elkerülésének biztosítása hardver-átvitelvezérlés híján az UART feletti protokollrétegek feladata, ami az esetek többségében könnyen kivitelezhető a két irány logikai összekapcsolásával vagy megfelelő méretű FIFO-tárak beépítésével, ezért a plusz két vezetéket igénylő hardver-átvitelvezérlést aránylag ritkán használjuk.

megfelelő pillanatát egy külön erre a célra fenntartott vezetéken előállított állapotváltozás segítségével jelzik (lásd később). Az UART protokoll nevében az **aszinkron** jelző arra utal, hogy a mintavételezés pillanatát a vevőnek időméréssel kell megállapítania.

A fizikai vonal alapállapotban magas szintű. Amikor az adó egy adatkeretet akar küldeni, akkor a vonalat alacsony szintre húzza, és egy, az adó és a vevő által is ismert időtartamig (bitidő, T_{baud}) alacsony szinten tartja. Ez az ún. start-bit, amelynek lefutóélét detektálva a vevő megkezdi a bitszinkronizáláshoz szükséges időmérést (lásd 5-3. ábra).



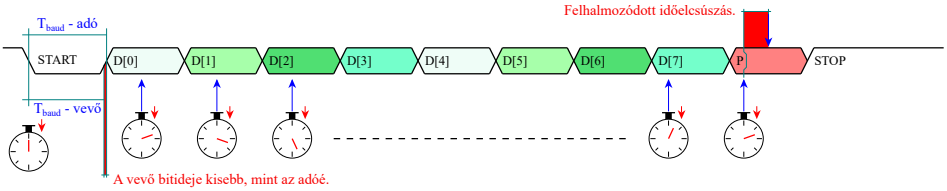
5-3. ábra. *Az UART adatkeret időzítése*

A start-bit után az adó az átküldendő adatkeret egyes bitjeit – a legalacsonyabbtól a legmagasabb helyiértékűig (**L**east **S**ignificant **B**it, LSB-first) – sorban egymás után rákapcsolja a fizikai vonalra úgy, hogy az egyes bitek mindig pontosan T_{baud} ideig állnak rendelkezésre. A vevő feladata, hogy a bitidő ismeretében – a start-bit lefutó élétől kezdve az időmérést – megállapítsa a mintavételezés optimális pillanatát. Az adatkeret bitjeinek átküldését egy opcionális, az adatbitekből képzett paritásbit követi, majd a tranzakciót az ún. stop-bit zárja le, ami a vonal egy, másfél vagy két T_{baud} időtartamú magas szinten tartását jelenti.

Az UART esetén tehát a bitszinkronizáláshoz elengedhetetlen a pontos időmérés, ami az adó és a vevő oldalon egyaránt egy-egy időzítő-számlálóval oldható meg.[203] Az időmérés pontossága így a lokális órajelforrások precizitásán múlik. Az órajelforrás FPGA-s megvalósítás esetén jellemzően külső kristályoszcillátor vagy belső (az FPGA chipen belül kialakított) RC-oszcillátor. Ezek az UART-ra jellemző 2-200 kb/s adatátviteli sebességek (T_{baud} = 5-500 μs) mellett gyakorlatilag tökéletesen pontosnak tekinthetők.

A fizikai réteg megfelelő kialakításával, pl. differenciális meghajtók használatával az UART adatátviteli sebessége akár több tíz Mb/s-ra (T_{baud} < 100 ns) is növelhető, ebben az esetben azonban már feltétlenül vizsgálnunk kell az adó és vevő áramkörök lokális időmérésének pontosságát. Az adó- és vevőáramkörök órajelforrásainak pontossága óhatatlanul eltér, teljesen „egyformán járó órákat" – észszerű költség mellett – lehetetlen készíteni, ami korlátozza az adott technológián, adott eszközökkel (mikrovezérlő, FPGA stb.) létrehozott UART-kapcsolat maximális adatátviteli sebességét. Ha az adó és a vevő által mért bitidők eltérnek, akkor a vevő oldalon a mintavételezés pillanata minden átvitt bittel egy kissé elcsúszik az optimális pillanathoz képest. Az 5-4. ábra példájában megfigyelhető, hogy a két fél által mért bitidők apró eltérése miatt a paritásbit vevőoldali mintavételezésének pillanata veszélyesen megközelíti az adatvonal adó általi beállításának pillanatát.

[203] Ez az oka annak, hogy katalógus-IC-kben nem jellemző az UART használata. A pontos időmérés analóg megvalósítása pontosan beállított értékű, sok helyet foglaló, külső RC-tagokat, digitális kivitele pedig egy lokális órajelforrást és egy számláló áramkört igényel. Mindkét megoldás bonyolultabb és költségesebb, mint a következő pontokban tárgyalt szinkron soros átviteli módszerekhez szükséges áramkörök, ezért a katalógus-IC-ket (szenzorok, A/D- és D/A-átalakítók stb.) tipikusan inkább ezekkel szerelik fel.



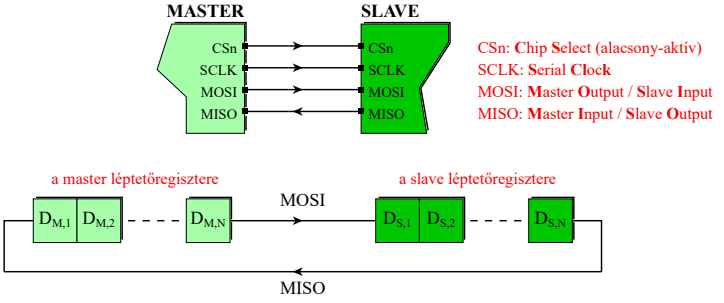
5-4. ábra. *A mintavétel pillanatának elcsúszása az optimális pillanathoz képest*

Minél több bitet viszünk át egy adatkereten belül, annál több elcsúszás halmozódik fel a vevőoldali mintavételezés során. Ha a felhalmozott időelcsúszás eléri vagy meghaladja a bitidő felét, akkor a vevő instabil adatvonalat mintavételez, ami funkcionális hibához vezet. A mintavétel pillanatának az adatvonal változásaihoz képesti eltolódása csak az adatkereten belül érvényesül, hiszen az időmérést az adatkeretek elején lévő start-bit újraindítja, így az elcsúszásból adódó hiba valószínűsége a keretméret csökkentésével mérsékelhető. Ugyanakkor az adatkeret méretének csökkentése azt is jelenti, hogy a start-bit, paritásbit, stop-bit hármas arányaiban egyre több időt igényel a hasznos adatbitekhez képest, ami csökkenti a nettó adatátviteli sebességet, ezért nyolcbitesnél kisebb adatkeretek a gyakorlatban nem jellemzők.

Tovább nehezíti a helyzetet, ha a fizikai vonal zajjal terhelt, ekkor ui. nem elegendő bitenként egyetlen, jól ütemezett mintavétel az adott bit időablakának közepén. A zajos vonalon való kommunikáció megbízhatósága túlmintavételezéssel növelhető, vagyis ha az adott bit időablakában több mintát veszünk, majd e mintákból becsüljük meg az adó eredeti szándékát. Ennek a módszernek a legegyszerűbb kivitele a háromszoros mintavétel, amelyben a bitenkénti három mintából „kétharmados szavazással” állítjuk vissza a bit valószínűsített értékét. A túlmintavételezéses megoldásokban az adott bit időablakában egyenletesen elosztva célszerű mintákat venni a fizikai vonalból, és a mintavételi pillanatoknak az órajelforrások eltérő pontosságából adódó elcsúszását ugyancsak körültekintően vizsgálni kell, különösen nagy átviteli sebességek esetén.

5.2.2. SPI

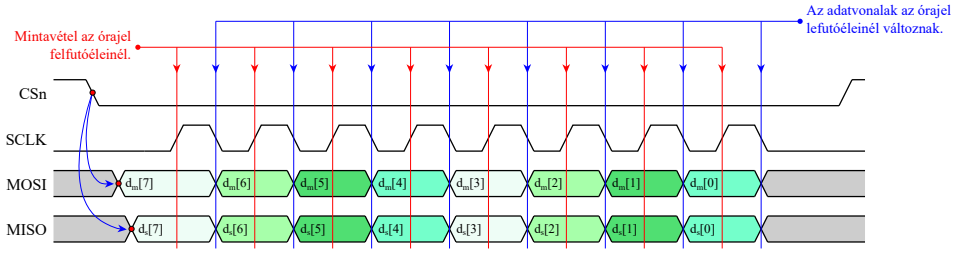
Az SPI (**S**erial **P**eripheral **I**nterface) protokoll egy két eszköz (master és slave) közötti kétirányú, pont-pont kapcsolat megvalósításához négy vezetéket definiál. Ezt a négy vezetéket felhasználva egy SPI-tranzakcióban N darab adatbitet viszünk át a master-ből a slave-be, és másik N darabot



5-5. ábra. *Az SPI-interfész jelei*

a slave-ből a master-be, amit szemléletesen úgy is elképzelhetünk, mint két, gyűrűbe kapcsolt léptetőregiszterben tárolt bitsorozat kicserélését (lásd 5-5. ábra).

A két kommunikáló fél szerepköre nem azonos; az adatátvitelt mindig a master kezdeményezi. A tranzakció első lépése, hogy a master alacsony szintre húzza a *CSn* (***C**hip **S**elect*, az *n* a jel alacsony-aktív jellegére utal) jelet, ezzel mintegy felébresztve a slave oldalt.[204] Az SPI esetén a bitszinkronizálásért felelős vezeték az *SCLK* (***S**erial **Cl**ock*), amelyen a master a *CSn* jel lehúzása után N darab órajelimpulzust állít elő 50%-os kitöltési tényezővel. Az *SCLK* jel felfutóéleinél a master a slave által meghajtott *MISO* (***M**aster **I**nput / **S**lave **O**utput*), a slave pedig a master által meghajtott *MOSI* (***M**aster **O**utput / **S**lave **I**nput*) jelet mintavételezi. A *MOSI* és *MISO* jelek az *SCLK* lefutóéleinél veszik fel a következő átviendő bit értékét. Az átvitel általában a legnagyobb helyiértékű bittel kezdődik (**M**ost **S**ignificant **B**it, MSB-first), de sok eszköz ezt a tulajdonságot paraméterként kezeli. Az SPI-tranzakció időzítését az 5-6. ábra szemlélteti.



5-6. ábra. *Egy nyolcbites SPI-tranzakció időzítése*[205]

Kialakítható olyan SPI-interfész is, amelyben egyetlen master több slave modullal kommunikál részben megosztott fizikai vezetékeken keresztül, ezt SPI busznak nevezzük. Az ún. multi-drop kialakítást az 5-7. ábra mutatja be. Ebben a megvalósításban minden slave egységnek saját *CSn* bemenete van, az *SCLK*, *MISO* és *MOSI* jeleket pedig mindegyik slave-be bekötjük.[206]

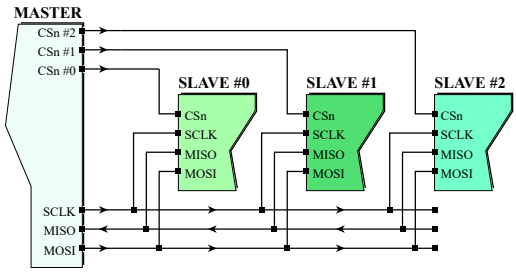
Az SPI busz-kialakítás fontos követelményt támaszt a slave egységekkel szemben; a *MISO* kimenetüknek háromállapotú meghajtókat kell tartalmaznia. Amikor egy slave egység *CSn* jele magas értékű – vagyis a master nem az adott slave egységet választotta ki –, akkor a *MISO* kimenetét nagyimpedanciás (high-Z) állapotban kell tartania az éppen kiválasztott, a *MISO* jelet aktívan használó egységgel való összehajtás elkerülése érdekében.

---

[204] A „felébreszt” kifejezés használata sok esetben kifejezetten pontosan írja le, hogy mi történik; nagyon kis fogyasztású katalógus-áramkörökben a *CSn* jel valóban a chip belső részegységeinek bekapcsolására szolgál. Amíg a *CSn* jel magas szintű, addig ezeknek az áramköröknek a fogyasztása jellemzően a µA tartományba esik. Sok gyártó ezt a működést „sleep” üzemmódnak hívja.

[205] Az SPI protokoll az *SCLK* jel polaritása (CPOL, **C**lock **Pol**arity) és az adatvonalak változásainak *SCLK*-hoz viszonyított fázisa (CPHA, **C**lock **Pha**se) szerint négy üzemmódot definiál (MODE0...3). Az itt bemutatott időzítési modell a MODE0 üzemmódnak felel meg.

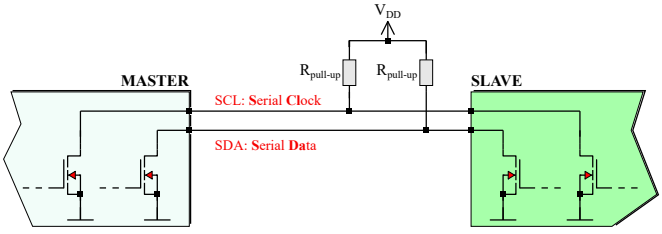
[206] SPI busz kialakítható oly módon is, hogy az egyes slave egységek az egyetlen *CSn* jelet is megosztva használják. Ebben a struktúrában a master *MOSI* kimenete az első slave *MOSI* bemenetét hajtja. Az n-edik slave *MISO* kimenetét az (n+1)-edik slave *MOSI* bemenetére kötjük, és a legutolsó slave *MISO* kimenetével hajtjuk meg a master *MISO* bemenetét. Így tulajdonképpen az összes résztvevő léptetőregisztereiből egyetlen nagy gyűrűt képezünk (daisy-chain). A master által generált *SCLK* impulzusok száma határozza meg, hogy a master léptetőregiszterének tartalma melyik slave egységig „jut el”, egyúttal melyik slave léptetőregiszterének tartalma íródik be a master egységbe. Ez a megoldás kevesebb vezetéket igényel, ugyanakkor a kommunikáció lelassul, ráadásul az adatcsere időigénye függ attól, hogy melyik slave egységgel szeretnénk kommunikálni.



5-7. ábra. *Több slave egység egy SPI buszon*

5.2.3. I²C

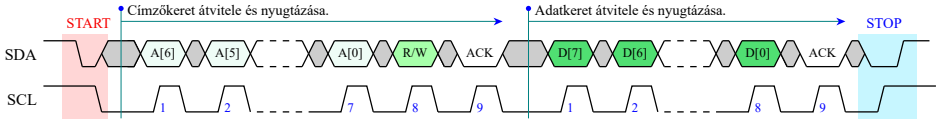
Az I²C (**I**nter-**I**ntegrated **C**ircuit) protokoll az SPI-hoz több szempontból hasonló megoldás; szinkron soros adatátviteli módszer, amelynek legegyszerűbb, pont-pont jellegű kiépítésében egy master és egy slave egység kommunikál egymással, és az adatcserét ebben az esetben is mindig a master kezdeményezi. A fizikai kialakítás ugyanakkor jócskán eltér az SPI-tól; az I²C mindössze két vezetéket használ a kétirányú kommunikáció megvalósítására. Az *SCL (**S**erial **Cl**ock)* vezeték az adatátvitel ütemezéséért (bitszinkronizálás) felelős, míg az *SDA (**S**erial **D**ata)* vezeték magát az átvinendő adatot hordozza. Az I²C sajátossága, hogy e két vezetéket soha egyik fél sem húzza aktívan magas szintre, a master és a slave kimenetei is nyitott kollektoros/nyelős jellegűek (open-collector vagy open-drain az eszközben használt kapcsolástechnikától függően). Ebből az következik, hogy az *SCL* és *SDA* vezetékeket egy-egy felhúzó ellenálláson keresztül a tápfeszültségre kell kapcsolni ahhoz, hogy a logikai magas szint is előállhasson rajtuk (lásd 5-8. ábra).



5-8. ábra. *Pont–pont I²C-kapcsolat külső felhúzó ellenállásokkal, nyitott nyelős (open-drain) végpontokkal*

A logikai alacsony szint az adott vezetéken tehát akkor áll elő, ha akár a master, akár a slave a kimeneti fokozatán keresztül aktívan lehúzza azt, a magas szint pedig úgy, hogy mindkét fél nagyimpedanciás állapotba kapcsolja az adott kimenetét („elengedi a vezetéket"), így a gyenge felhúzást biztosító ellenálláson keresztül a vezeték potenciálja a tápfeszültségre áll be. Az adatátvitel során az *SDA* vonalat a master és a slave is az *SCL* lefutóéle után változtatja meg, és az *SCL* felfutóéle után mintavételezi. Úgy is fogalmazhatunk, hogy **alapesetben** az *SDA* vonal értéke csak alacsony *SCL* mellett változtatható meg, és a kommunikáló felek mindig az *SCL* magas értékénél vizsgálják az adatvonalat.

A legegyszerűbb I²C-tranzakció az alábbi lépésekből áll (lásd 5-9. ábra):



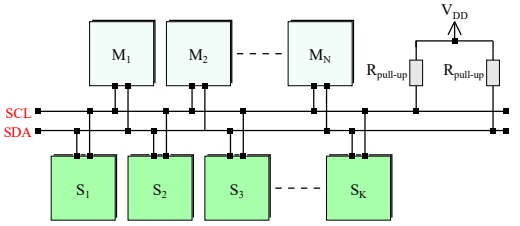
5-9. ábra. *Egy I²C-tranzakció időzítési modellje*

1. A fent említett, általános időzítési szabályt megsértve a master magas *SCL* mellett lehúzza az *SDA* vonalat, majd némi késleltetés után az *SCL*-t is. Ez a „szabálysértés” az ún. startfeltétel (start-condition), amit érzékelve a slave egység felkészül a tranzakcióra (vö. SPI *CSn* lefutóél).
2. A master egy nyolcbites, ún. címzőkeretet küld a slave-nek. A címzőkeret felső hét bitje a slave kiválasztását szolgálja, a legalsó helyiértékű bit (R/W) pedig azt kódolja, hogy a megkezdett tranzakció írás (alacsony szint: a master küld adatot a slave-nek) vagy olvasás (magas szint: a slave küld adatot a master-nek) lesz-e. A címzőkeret átvitele a következőképpen megy végbe; a master az *SDA* értékét a címzőkeret n-edik bitjének megfelelő értékre állítja, majd – miután az *SDA* vonal értéke stabilizálódott – felengedi az *SCL*-t, jelezve a slave-nek, hogy az *SDA* vonalat mintavételezheti. A mintavételhez szükséges késleltetés beiktatása után az *SCL* vonalat ismét alacsony szintre húzza. E lépéseket nyolcszor elismételve a teljes címzőkeret átvitele megtörténik.
3. Ha a slave a master által küldött hétbites címet sajátjaként azonosította, akkor az *SCL* vonalnak az R/W bit mintavételezését követő lefutóéle után alacsony szintre húzza az *SDA* vonalat, nyugtázva a master kérését (acknowledge, ACK). A master fogadja a nyugtázást a következőképpen; felengedi az *SCL* jelet, mintavételezi az *SDA* vonalat, majd újra lehúzza az *SCL* vonalat.
4. Ezek után egy vagy több, ugyancsak nyolcbites adatkeret átvitelére kerül sor. A folyamat a címzőkeret átviteléhez hasonló módon zajlik, a különbség csupán az, hogy az *SDA* vonal meghajtásáért ebben a ciklusban írástranzakció esetén (R/W = 0) a master, olvasástranzakció esetén (R/W = 1) a slave felel. A nyolcbites adatkeret átvitelét újabb nyugtázás követi, amellyel a fogadó fél az átvitel sikerességét jelzi.[207] Az R/W bit értéke a teljes tranzakcióra vonatkozik, függetlenül attól, hogy abban hány adatkeret átvitelére kerül sor. Az adatáramlás iránya csak újabb startfeltétel generálásával és újabb címzőkeret (egyúttal újabb R/W bit) elküldésével fordítható meg.
5. A tranzakció utolsó lépése, hogy a master lehúzza az *SDA* vonalat, felengedi az *SCL*-t, majd némi késleltetés után – ismét megsértve az időzítési alapszabályt – felengedi az *SDA* vonalat is. Ez az ún. stopfeltétel (stop-condition), ami jelzi a slave számára, hogy a tranzakció befejeződött (vö. SPI *CSn* felfutóél).

[207] Az I²C-protokoll nem rögzíti, hogy egymás után hány darab adatkeret átvitelére kerülhet sor egyazon tranzakción belül. Sok I²C slave interfésszel ellátott katalógus-IC az adatkeretekben kapott adatot egy korábbi tranzakció segítségével kiválasztott regisztercímre írja (vagy fordított irányú átvitel esetén a master-nek küldendő adatot onnan olvassa be). Ha az adott tranzakcióban több adatkeret átvitelére kerül sor, akkor tipikusan a célregiszter címe a slave egységen belül automatikusan inkrementálódik. Így hatékonyan valósítható meg blokkos adatátvitel egy master és egy slave között úgy, hogy azt másik master nem szakítja meg.

Az 5-9. ábrán látható, hogy – bár az I²C-protokoll nem fogalmaz meg követelményt a kitöltési tényezőre vagy az egymást követő bitek átvitele közötti maximális késleltetésre vonatkozóan – az *SCL* jel kvázi-periodikus, az egyes bitek átvitele általában ugyanannyi időt vesz igénybe (kivéve az ún. órajelnyújtás esetét, lásd később). Az I²C-busz átviteli sebességét rendszerszinten legegyszerűbben az *SCL* jel maximális frekvenciájával szokás jellemezni. A legtöbb, I²C slave interfésszel ellátott katalógus-IC a „standard" és a „fast" sebességet támogatja, amelyeknél az *SCL* vonal frekvenciája rendre 100 kHz és 400 kHz. A protokollnak léteznek újabb változatai is (fast plus, high-speed, ultra-fast), ezek leggyorsabb verziója esetén az *SCL* vonal frekvenciája akár az 5 MHz-et is elérheti.

Multi-master / multi-slave topológia ◆ Az SPI protokollal ellentétben az I²C több master és több slave kommunikációját is lehetővé teszi egyazon fizikai vezetékpáron keresztül (lásd 5-10. ábra).



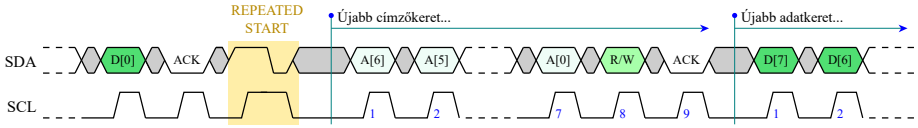
5-10. ábra. *Multi-master / multi-slave I²C-busz fizikai kialakítása*

A két vezetéket (a közös „buszt") használva bármely master egység cserélhet információt bármely slave egységgel. Az I²C-busz esetén a végpontok nyitott kollektoros kimenete lehetetlenné teszi az „összehajtást", vagyis hogy két végpont ellentétes logikai értéket kényszerítve a közös vezetékre közvetlen összeköttetést hozzon létre a tápsín és a föld között, ez a kialakítás tehát kizárja az eszközök fizikai sérülésének lehetőségét. Ugyanakkor nyilván egyszerre csak az egyik master által kezdeményezett tranzakció valósulhat meg hibátlanul, hiszen ha két egység egyidőben használja a buszt, akkor az *SCL* és *SDA* vezetékeken valójában a két különböző forrásból származó adatok logikai *AND* kapcsolata jelenik meg, ami funkcionális hibához vezet.

Minden megosztott erőforrás-használati séma fontos kérdése az ún. arbitráció, vagyis a közös erőforrások használati jogát meghatározó mechanizmus. Az I²C esetén a master egységek felelőssége a busz állapotának ellenőrzése annak használatba vétele előtt és közben:

- A master egység folyamatosan monitorozza a buszt a start- és stopfeltételekre jellemző mintázatokat keresve. Ha a buszon egy másik master már kezdeményezett egy tranzakciót és még nem zárta azt le, akkor újabb tranzakció értelemszerűen nem kezdeményezhető.
- Mivel a startfeltétel detektálása időt vesz igénybe, ezért előfordulhat, hogy két master egyazon pillanatban kezdeményez tranzakciót. Ez a helyzet úgy oldható fel, ha a master egységek a cím- és adatkeretek átvitelekor az *SDA* elengedése után mindig ellenőrzik, hogy a vonal valóban magas szintet vett-e fel. Ha egyszerre két master használja a buszt, akkor előbb-utóbb az egyikük azt fogja tapasztalni, hogy az *SDA* vonal alacsony szintű maradt. Ilyenkor az adott master az általa kezdeményezett tranzakciót sikertelennek minősítve elengedi a buszt. Ezt a módszert *SDA*-arbitrációnak nevezzük.

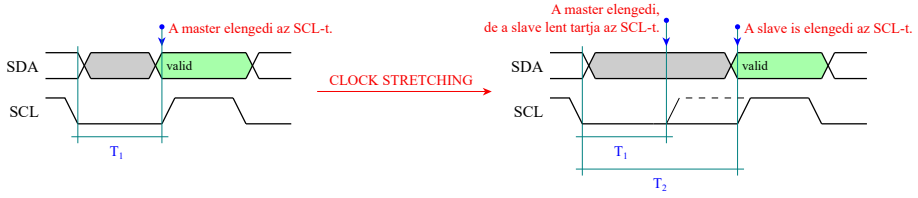
Megismételt startfeltétel (repeated start-condition) ◆ Ha több master használja egyszerre a buszt, akkor hasznos lehet, ha egy master két, teljesen eltérő címre szánt tranzakció között magánál tudja tartani a busz használatának jogát. Ennek megvalósítására definiálja az I²C-protokoll az ún. megismételt startfeltétel funkciót (lásd 5-11. ábra).



5-11. ábra. *Megismételt startfeltétel*

A megismételt startfeltétel generálásakor a master egy adatkeret nyugtázása után, alacsony *SCL* mellett felengedi az *SDA* vonalat, majd az *SCL*-t is. Ezután, az immár magas *SCL* mellett – tehát a korábban is említett általános szabálynak ismét ellentmondva – lehúzza az *SDA* vonalat, majd némi késleltetés után az *SCL*-t is. Így a buszt eddig is használó master úgy állított elő egy startfeltételt (vö. 5-9. ábra piros kiemelés és 5-11. ábra sárga kiemelés), hogy a busz használatának jogát mindvégig magánál tartotta, hiszen a folyamat során nem állította elő a stopfeltételnek megfelelő mintázatot a két vezetéken.

Órajelnyújtás (clock stretching) ◆ Ha megvizsgáljuk egyetlen bit slave-ből master-be való átvitelének folyamatát, akkor azt látjuk, hogy az *SCL* vonal lefutása után a slave egységnek adott időn belül – az *SCL* következő felfutása előtt – stabilizálnia kell az *SDA* vonal értékét ahhoz, hogy a master megfelelően mintavételezhesse azt (lásd 5-12. ábra).



5-12. ábra. *Órajelnyújtás*

Ha a slave egység (pl. egy lassú A/D-átalakító) nem képes T_1 idő alatt megfelelően beállítani az *SDA* értékét, akkor folyamodhat az órajelnyújtás (clock stretching) módszeréhez. Az eddigiek során azt láttuk, hogy az *SCL* jel állapotát mindig a master egység határozta meg, a slave-nek csak az *SDA* vonal írásával és olvasásával volt feladata. Az órajelnyújtás során azonban a slave az *SCL* vonal állapotának befolyásolásával éri el az adott bit átvitelére szánt időtartam megnövelését. Amikor a master az *SCL* lefutása után – T_1 idő elteltével – elengedi az *SCL* vonalat, akkor a slave annak alacsony szinten tartásával jelzi, hogy még nem sikerült beállítania az *SDA* vonal értékét. Amikor a slave végül megfelelően beállítja az *SDA*-t (az 5-12. ábrán T_2 idővel az *SCL* lefutása után), akkor felengedi az *SCL* vonalat, és az adatcsere a szokásos módon folytatódik a következő bit átvitelével.

5.3. Bináris számábrázolás

Az RTL-tervezés idealizált digitális világában bármi, aminek értéke van, az – egy-két speciális esettől eltekintve – logikai alacsony (nulla) vagy magas (egy) szintű. Gondolkodhatunk tehát úgy, hogy áramköreink minden adatot, információt bináris formában kezelnek. Egyetlen információhordozó elem – legyen az egy vezeték, egy flip-flop vagy egy RAM-cella – egy bit információ reprezentálására képes. Minél összetettebb a rögzítendő információ, annál több bitre van szükségünk, ezért létrehoztuk a logikai vektor fogalmát, ami valamilyen rendező elv szerint összetartozó, egyedien azonosított bitek halmaza. A 2.2. alfejezetben bemutatott CPU modell esetén például a 16 darab gépi utasítás megkülönböztetésére egy négybites logikai vektort, az ún. utasításkódot használjuk, amelyet fizikailag a processzoron belül egy 16 darab D flip-flopból álló regiszter, az utasításregiszter megegyezésszerűen kiválasztott 4 bitje képvisel. A logikai vektorokon belüli biteket egyszerű sorszámozással, indexeléssel azonosítjuk. Az utasításregiszter példájánál maradva azt mondjuk, hogy az utasítás típusát a [15:12] indexű bitek kódolják.

Mivel áramköreink jelentős részét azért hozzuk létre, hogy bonyolult számításokat végezzenek helyettünk, célszerű tisztában lennünk azzal, hogy milyen lehetőségeink vannak a számok bináris ábrázolására. Ebben a fejezetben áttekintjük a különféle számábrázolási megoldásokat, rámutatva azok sajátosságaira, valamint alkalmazásuk előnyeire és korlátaira.

5.3.1. Fixpontos számok

Az arab számokhoz és a tízes számrendszerhez szokott szemünknek természetesnek hat, hogy a számokat a 0, 1, 2 ... 9 szimbólumok, „számjegyek” sorozataként írjuk fel. Ritkán gondolunk bele, hogy ez az ún. „helyiértékes” számábrázolás[208] valójában azt mutatja meg, hogy a kifejezendő számot az alapszám (tíz) hatványainak súlyozott összegeként képezve az összeg melyik tagjának mi a súlya, vagyis, hogy hány darabra van belőle szükség:

$$235{,}12 = (\mathbf{2} \times 10^2) + (\mathbf{3} \times 10^1) + (\mathbf{5} \times 10^0) + (\mathbf{1} \times 10^{-1}) + (\mathbf{2} \times 10^{-2})$$

Nagy szerencsénkre a helyiértékes számábrázolás módszere más alapszámokra is általánosítható. A feltétel csupán annyi, hogy definiáljuk az alapszámnak megfelelő számú különböző szimbólumot az egyes hatványok mennyiségének kifejezésére. Ahogy korábban említettük, a digitális rendszerekben csak kétféle szimbólum jöhet szóba, így kénytelenek vagyunk a helyiértékes számábrázolásnak azt a változatát használni, amelynek alapszáma 2. Ezt a számrendszert hívjuk kettes, vagy bináris számrendszernek.

Természetes számok ábrázolása ◆ Elsőre talán túlságosan is korlátozottnak tűnhet a természetes (nemnegatív egész) számok halmaza, ha bonyolult számítási feladatokról van szó, de a digitális áramkörökben gyakran ez írja le legjobban az adott problémát (pl. memóriacímzés). Kettes számrendszerben a természetes számokat a 2, mint alapszám nemnegatív egész kitevőjű hatványainak súlyozott összegeként állítjuk elő. Az egyes hatványok együtthatója csak 1 vagy 0 lehet, így a hatványsor együtthatóit egymás mellé felsorakoztatva egy logikai vektort kapunk:

[208] A „helyiértékes” kifejezés arra utal, hogy az egyes számjegyeknek az értéke és a többi számjegyhez viszonyított pozíciója (helye) együttesen hordozza az információt, ellentétben a hieroglifikus számábrázolások szimbólumaival, amelyek pozíciójuktól függetlenül mindig ugyanazt az értéket képviselik (pl. római számok: V: 5, X: 10, L: 50. C: 100).

$$25_{10} = (\mathbf{1} \times 2^4) + (\mathbf{1} \times 2^3) + (\mathbf{0} \times 2^2) + (\mathbf{0} \times 2^1) + (\mathbf{1} \times 2^0) = \mathbf{11001_2}$$

Az általánosan megfogalmazott formula a természetes számok bináris helyiértékes kifejezésére a következő:

$$V = \sum_{i=0}^{i=(N-1)} b_i \times 2^i$$

ahol *N* az ábrázolásra szánt számjegyek (bitek[209]) száma, b_i pedig kettő *i*-edik hatványának együtthatója. A legnagyobb ábrázolható érték 2^N-1, a legkisebb pedig értelemszerűen nulla, hiszen természetes számokról van szó.

Nemnegatív törtszámok fixpontos ábrázolása ◆ A tízes számrendszerben a számok törtrészét tíz negatív egész kitevőjű hatványainak segítségével fejezzük ki. Ez a módszer is átvihető a kettes számrendszerbe; a törteket kettő negatív kitevőjű hatványaival írjuk fel:

$$2{,}625_{10} = (\mathbf{1} \times 2^1) + (\mathbf{0} \times 2^0) + (\mathbf{1} \times 2^{-1}) + (\mathbf{0} \times 2^{-2}) + (\mathbf{1} \times 2^{-3}) = \mathbf{10{,}101_2}$$

A tízes számrendszerbeli tizedespont helyét a „kettedes-” vagy „binárispont” veszi át, amelynek szerepe is ugyanaz; elválasztja az egészrész és a törtrész ábrázolására szánt számjegyeket.[210] A binárispont helyzete adott számábrázolási formátumban kötött; innen a módszer elnevezése: fixpontos számábrázolás. A nemnegatív törtszámok értéke bináris helyiértékes ábrázolás esetén a következőképpen fejezhető ki:

$$V = \sum_{i=-F}^{i=(I-1)} b_i \times 2^i$$

ahol *I* az egészrész, *F* a törtrész ábrázolására szánt bitek száma b_i pedig kettő *i*-edik hatványának együtthatója. Ebben a számábrázolási formában a legnagyobb ábrázolható érték $2^I - 2^{-F}$, a legkisebb pedig ezúttal is nulla.[211]

Negatív számok ábrázolása ◆ Tízes számrendszerben ún. előjeles-abszolútértékes ábrázolással kezeljük a negatív számokat. Ez annyit jelent, hogy negatív számok esetén maguk a számjegyek a szám abszolútértékét írják le, a negatív jelleget pedig egy újabb szimbólum, a negatív előjel (-) odabiggyesztésével fejezzük ki. A digitális világban újabb szimbólum bevezetésére nincs lehetőség, ezért a negatív számok ábrázolását másképp kell megoldanunk. Az ún. „kettes-komplemens” kódolás abban különbözik az egyszerű, bináris helyiértékes ábrázolási módtól, hogy a szám értékét

[209] bit: binary digit, bináris számjegy

[210] A törteket ábrázoló logikai vektorokkal műveleteket végző áramkörökben a binárispont pozícióját többféle módon is számontarthatjuk. A *fixed_pkg* VHDL csomag a törtszámot reprezentáló logikai vektor bitjeit úgy indexeli, hogy az adott bithez tartozó index megegyezik a bithez tartozó hatványkitevővel (negatív indexek is értelmezettek). Más megoldásokban (pl. a 4.5.1. pont aritmetikai egységében) minden logikai vektor esetén ismert az egészrész és a törtrész ábrázolására szánt bitek száma – pl. a vektorhoz szorosan kötődő, egész típusú adatként –, így a vektor által képviselt érték is egyértelműen meghatározható.

[211] A nemnegatív egész számok bináris helyiértékes ábrázolása is „fixpontos”, ahol a binárispont a legkisebb helyiértékű bit jobb oldalára képzelendő, tehát a törtrész ábrázolására szánt bitek száma *(F)* nulla.

adó, kettő hatványait tartalmazó súlyozott összegben a legnagyobb kitevőjű hatványt negatív előjellel vesszük figyelembe:

$$-5{,}25_{10} = -(\mathbf{1} \times 2^3) + (\mathbf{0} \times 2^2) + (\mathbf{1} \times 2^1) + (\mathbf{0} \times 2^0) + (\mathbf{1} \times 2^{-1}) + (\mathbf{1} \times 2^{-2}) = \mathbf{1010{,}11_2}$$

A kettes-komplemens kódolású bináris alakban megadott számok értéke tehát az alábbi módon fejezhető ki:

$$V = -b_{(I-1)} \times 2^{(I-1)} + \sum_{i=-F}^{i=(I-2)} b_i \times 2^i$$

ahol I az egészrész, F a törtrész ábrázolására szánt bitek száma, b_i pedig kettő i-edik hatványának együtthatója. Az 5-51. táblázat bemutatja, hogy ezzel az apró trükkel valójában mit is hajtottunk végre. A táblázat középső oszlopában egy négybites logikai vektor összes lehetséges értékét látjuk. A bal oldali oszlop tartalmazza a logikai vektor decimális értékét, ha a bináris alakot nemnegatív egész számként értelmezzük, a jobb oldali oszlop pedig azokat a decimális értékeket tartalmazza, amelyeket akkor kapunk, ha a bináris alakot kettes-komplemens kódolású egész számnak tekintjük.

Decimális érték (nemnegatív)	Bináris alak	Decimális érték (kettes-komplemens)
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	-8
9	1001	-7
10	1010	-6
11	1011	-5
12	1100	-4
13	1101	-3
14	1110	-2
15	1111	-1

5-51. táblázat. *A kettes-komplemens kódolás eltolja az ábrázolt tartományt a negatív számok irányába*

N darab bit felhasználásával a nemnegatív esethez hasonlóan továbbra is 2^N különböző számot tudunk leírni. Ami megváltozott, az az ábrázolt tartomány helyzete a számegyenesen; a (0 ... 15) tartomány helyett a kettes-komplemens kódolással a (-8 ... 7) tartományt fedjük le. Az előjel ábrázolásának ára tehát az ábrázolható számok abszolútértékének felére csökkenése. Fogalmazhatunk úgy is, hogy az 1-essel kezdődő bináris alakokat ezúttal a negatív tartomány lefedésére

használjuk.[212] A kettes-komplemens kódolású számábrázolás esetén a legnagyobb ábrázolható érték $2^{(l-1)} - 2^{(-F)}$, a legkisebb pedig $-2^{(l-1)}$. Az 5-51. táblázatban az is látható, hogy a nemnegatív számok kettes-komplemens kódja megegyezik a bináris helyiértékes alakkal. Az adott esetben fejszámolást sem mellőző tervezés során hasznos „trükk", ha a negatív számok kettes-komplemens kódját úgy képezzük, hogy a szám abszolútértékének megfelelő bináris vektor bitjeit negáljuk, majd az eredményhez hozzáadunk egyet (az összeadásnál keletkező átvitelbitet eldobjuk). Pl. a -4 érték esetén: 4: 0100 → 1011 → 1100: -4. A dolog természetesen visszafelé is működik; ha kíváncsiak vagyunk egy kettes-komplemens kódban ábrázolt negatív (1-essel kezdődő) szám abszolútértékére, akkor ugyanezeket a lépéseket végezzük el: -5: 1011 → 0100 → 0101: 5.

Figyeljük meg a kettes-komplemens kódnak azt a jellegzetességét, hogy a bináris alak által képviselt decimális számok ugyanolyan természetes sorrendben követik egymást, mint az előjel nélküli esetben. Ha az 1100 bináris alaknak megfelelő, előjel nélküli 12-höz hozzáadunk 1-et, akkor a táblázatban egy sorral lejjebb lépve a 13 decimális értékhez (12+1) és az 1101 bináris alakhoz jutunk, amely ugyancsak a 13 decimális számot képviseli. Ha ugyanezt a műveletet a kettes-komplemens kódolású esetben is elvégezzük, akkor azt látjuk, hogy az 1100 bináris alaknak megfelelő decimális érték ezúttal a -4, amelyhez 1-et hozzáadva – vagyis a táblázatban egy sorral lejjebb lépve – az 1101 bináris alakhoz jutunk, amely ezúttal a decimális -3 számot kódolja, ami ugyancsak 1-gyel nagyobb, mint a -4. Ez a tulajdonság lehetővé teszi azt, hogy a kettes-komplemens kódolású számokon ugyanazokkal a módszerekkel végezzünk aritmetikai műveleteket, mint amelyeket az egyszerű bináris helyiértékes nemnegatív számokon is használnánk. „Módszerek" alatt itt a helyiértékenkénti összeadás/kivonást értjük, ami számunkra azért különösen fontos, mert aritmetikai áramköreink lényegében ennek a megoldásnak a közvetlen fizikai megvalósításai (pl. ripple-carry összeadó struktúra, lásd DRMSz: 8-40. ábra). Ez azt jelenti, hogy ugyanazok az aritmetikai áramkörök használhatók előjel nélküli és kettes-komplemens kódolású előjeles számok kezelésére is; ha a bemeneteket kettes-komplemens kódolású számként értelmezzük, akkor a művelet eredményét is akképp értelmezve helyes megoldáshoz jutunk. Ez természetesen nem vonatkozik az összehasonlító műveletek áramköreire; ezek belső működése függ attól, hogy operandusaikat nemnegatív vagy kettes-komplemens kódolású számnak tekintjük. Az 5-51. táblázatban az átláthatóság érdekében egész számokat használtunk, de a kettes-komplemens kódolás kedvező tulajdonságai törtszámok esetén is érvényesülnek.

5.3.1.1. A fixpontos számábrázolás gyakorlati kérdései

Tartomány vs. pontosság ◆ A digitális jelfeldolgozást és/vagy az általánosabb értelmű adatfeldolgozást végző rendszerek többségében kettes-komplemens kódolású, fixpontos számábrázolást használunk. A fixpontos számábrázolással működő áramkörök tervezésekor a rendszertervezés részeként – a problémakör pontos ismeretében – az adott feladatra kell optimalizálnunk

[212] Találkozhatunk azzal a megfogalmazással is, hogy a kettes-komplemens kódolású bináris számok legfelső helyiértékű bitje az előjelbit, ez azonban félrevezető; a legfelső bit nem önmagában az előjelbit, hanem az összes többi bittel együtt kódol egy konkrét számot, ami negatív, ha a legfelső bit 1. A „legfelső bit az előjelbit" megfogalmazás inkább csak az ún. előjeles-abszolútértékes bináris alakra igaz, amely – a tízes számrendszerben megszokott módon – a legfelső szimbólumot az előjel kódolására használja, a többi szimbólum pedig az ábrázolandó szám abszolútértékét ábrázolja nemnegatív bináris helyiértékes alakban (pl. a -5 szám előjeles-abszolútértékes alakban 1101_2, kettes-komplemens kódolás esetén azonban 1011_2).

a számábrázolás formátumát, vagyis azt, hogy hány bitet szánunk az egészrész és a törtrész leírására. Előbbi az ábrázolható tartomány méretét, utóbbi pedig a pontosságot határozza meg, adott bitszám esetén tehát az ábrázolható tartományt csak a pontosság rovására növelhetjük, és fordítva. A bitek számának növelése ugyanakkor egyrészt nagyobb méretű és fogyasztású áramkörökhöz vezet, másrészt – a nagyobb aritmetikai kombinációs hálózatokból adódóan – csökkenti az elérhető legmagasabb órajel-frekvenciát.

A sok, akár önmagában is összetett műveletből álló számítási feladatok esetén szokásos megoldás az, hogy a bemenő adatokat egy adott formátumban várjuk, de a belső számításokat egy megnövelt bitszámú formátumban végezzük el. Ez különösen iteratív, a saját maga által előállított részeredményeket a további számítások során bemenetként is felhasználó algoritmusok esetén hasznos, mert így a sok-sok aritmetikai művelet elvégzése kisebb eséllyel vezet az egészrész ábrázolására szánt bitek korlátozott száma miatti túlcsorduláshoz, továbbá kisebb mértékű a törtrész-bitek véges számából adódó pontatlanság felhalmozódása. A belső számítások elvégzése után aztán az eredményeket visszaalakíthatjuk a bemenetek formátumára. A kimenet méretének redukálása egy kritikus művelet, mivel egyrészt potenciálisan túlcsordulást okoz, másrészt veszíthetünk az eredmények pontosságából is. Ezeket a sajátosságokat rendszerszinten mindenképpen figyelembe kell vennünk.

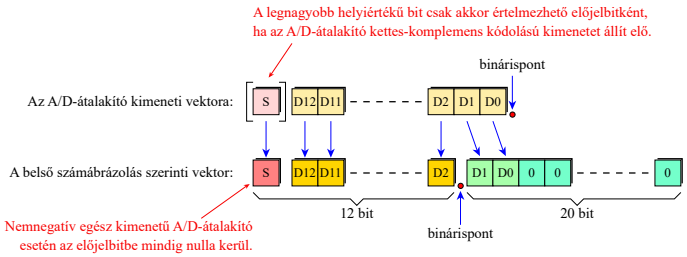
A/D-átalakítás ◆ Az adatfeldolgozó rendszerekben a forrásadatok sokszor mért és A/D-átalakítókkal digitalizált fizikai mennyiségek, a feldolgozott adatokat pedig D/A-átalakítókkal újra analóg mennyiségekké kell konvertálnunk. Az A/D-átalakítók egy analóg mennyiség digitális reprezentációját egy pontosan ismert és állandó értékű másik mennyiséghez, az ún. referenciafeszültséghez hasonlítva állítják elő, legegyszerűbb esetben a következőképpen:

$$D = 2^N \times \frac{V_A}{V_{REF}}$$

ahol V_A a digitalizálni kívánt analóg mennyiség, V_{REF} a referenciafeszültség, N az átalakító bitszáma, D pedig az átalakítás eredménye, egy N-bites bináris vektor. Az ún. unipoláris eszközök esetén V_A értéke mindig kisebb, mint V_{REF}, az átalakítás D eredménye pedig nemnegatív egészként értelmezve mindig egy 0 és 2^N-1 közötti egész szám. Bipoláris eszközöknél, vagyis ha V_A értéke V_{REF}-nél kisebb és nagyobb is lehet, akkor a helyzet kicsit bonyolultabb; itt a kimeneti számábrázolás jellege általában az adott A/D-átalakító egy beállítható paramétere. Kettes-komplemens kódolású kimenet esetén az A/D-átalakító a teljes $-V_{REF} \ldots +V_{REF}$ analóg bemeneti tartományát[213] a $-2^{(N-1)} \ldots +2^{(N-1)}-1$ digitalizált értékkészletre képezi le, nemnegatív egész kódolású kimenet esetén viszont ugyanazt a bipoláris bemeneti analóg tartományt a $0 \ldots +2^N-1$ értékkészlet képviseli. Ez utóbbi esetben tehát a nemnegatív egészként értelmezett bináris vektor $2^{(N-1)}$ értéke jelenti azt, hogy V_A éppen egyenlő V_{REF}-fel, az ennél kisebb számok a V_{REF}-nél kisebb, a nagyobbak pedig a V_{REF}-nél nagyobb V_A értékeket jelenítik meg. A nemnegatív egész kódolású bipoláris eszközök működése tehát felfogható úgy is, mintha egy egyszerű unipoláris eszköz a V_A bemeneti érték helyett a $V_A + V_{REF}$ mennyiséget alakítaná át nemnegatív egész számmá.

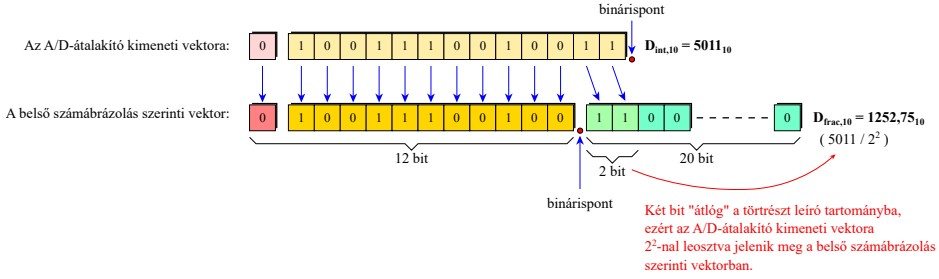
[213] Az eszköz pontos típusától függően az analóg bemeneti tartomány lehet $-V_{REF} \ldots +V_{REF}$ vagy $-V_{REF}/2 \ldots +V_{REF}/2$, ami nem elvi, csupán az adott eszköz jelölésrendszerétől függő formai különbség.

Az A/D-átalakító típusától függetlenül kijelenthető tehát, hogy a konverzió során egy fizikailag mért és korlátozott (V_{REF}) mennyiségből egy nemnegatív egész vagy kettes-komplemens kódolású egész szám keletkezik. Az A/D-átalakítók jellemző felbontása 10-16 bit, az adatfeldolgozó rendszerek a belső számítások pontosságának fenntartása érdekében azonban ennél jellemzően több bitet használnak, így felmerülhet a kérdés, hogy a digitalizált analóg minta bitjeiből hogyan képezzük az adatfeldolgozó rendszer belső számábrázolása szerinti fixpontos törtszámot. Az 5-13. ábra egy olyan példát mutat be, amelyben az A/D-átalakító 14-bites, kettes-komplemens kódolású kimenetet szolgáltat, a belső számábrázolás pedig 32-bites, kettes-komplemens kódolású fixpontos tört, ahol az egészrész és a törtrész ábrázolására szánt bitek száma pedig rendre 12 és 20.



5-13. ábra. *14-bites kettes-komplemens kódolású A/D-kimenet leképezése 32-bites belső számábrázolásra*

Az ábrán azt látjuk, hogy az A/D-átalakító kimeneti vektorát a belső számábrázolás szerint kódolt bináris vektor lehető legmagasabb helyiértékű tartományába helyezzük, az alsó helyiértékeket pedig nullákkal töltjük fel, így biztosítva a lehető legtöbb törtrészbitet a belső számítások pontosságának megőrzéséhez. Figyeljük meg, hogy a belső számábrázolásra való áttérés egyúttal egy skálázást is jelent(het), mivel az A/D-átalakító kimeneti vektorának végén lévő binárispont adatbitekhez viszonyított relatív pozíciója a belső számábrázolás szerint kódolt vektorban megváltozhat. Ez azt jelenti, hogy a belső számábrázolás szerint kódolt vektor nem ugyanazt a decimális számértéket jeleníti meg, mint amit az A/D-átalakító saját számábrázolása szerint kódolt kimeneti vektor (lásd 5-14. ábra). A skálázás az A/D-átalakító bitszámától és a belső számábrázolás formátumától függően egyaránt jelenthet leosztást és felszorzást, amit a további számítások elvégzésekor figyelembe kell vennünk.

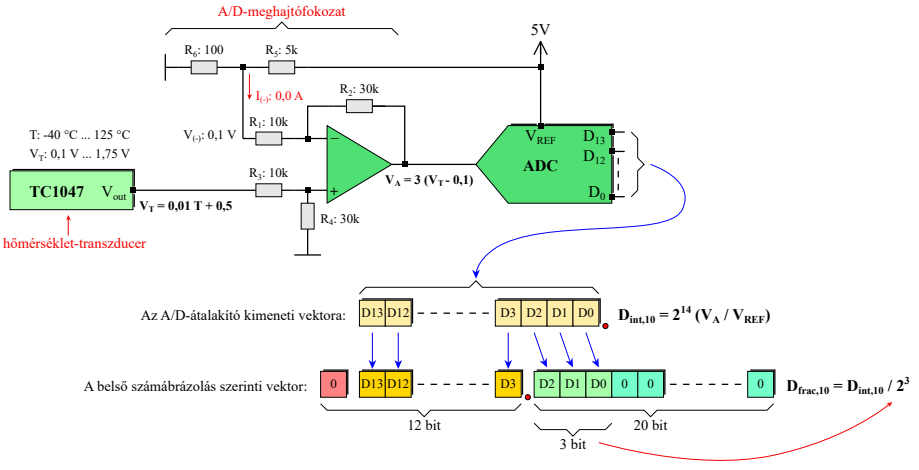


5-14. ábra. *A belső számábrázolásra való áttérés egyúttal skálázást is jelent(het)*

A binárispont relatív eltolódása nem az egyetlen tényező, amit egy analóg feszültség A/D-átalakítása és a digitalizált érték feldolgozása során figyelembe kell vennünk. Az A/D-átalakítók

bemeneti feszültségtartománya mindig kötött, amivel a mérendő feszültség lehetséges értékei nem feltétlenül vannak összhangban ezzel a tartománnyal. Az A/D-átalakító bemeneti tartományához képest nagyon kicsi mérendő tartomány azért problémás, mert ilyenkor nem tudjuk kihasználni az eszköz maximális pontosságát, a nagy pedig azért, mert az A/D-átalakító referenciafeszültségénél nagyobb bemeneti értékek az eszköz tönkremeneteléhez is vezethetnek. Ezért a mérendő feszültséget rendszerint valamilyen meghajtófokozaton keresztül kapcsoljuk az A/D-átalakító analóg bemenetére. A meghajtófokozat a mért feszültség és az A/D-átalakító bemenetére jutó feszültség között legegyszerűbb esetben egy lineáris transzformációt végez, vagyis a mért feszültséget skálázza és egy állandó értékkel el is tolja annak érdekében, hogy a mérendő tartományt az A/D-átalakító bementi tartományára képezze le.

További transzformációs tényezővel kell számolnunk akkor, ha a digitálisan megjelenítendő mennyiség nem egy feszültség, hanem egy más dimenziójú fizikai mennyiség, amit egy ún. transzducer alakít át az A/D-átalakító meghajtófokozatának bemenetén megjelenő feszültséggé. A különféle transzducer elemek legegyszerűbb esetben ugyancsak egy lineáris transzformációt hajtanak végre. Az 5-15. ábra egy olyan példát mutat be, amelyben egy TC1047 típusú, feszültség-kimenetű hőmérséklet-érzékelő kimeneti jelét digitalizáljuk egy egyszerű műveleti erősítős meghajtófokozattal ellátott 14-bites, nemnegatív egész kimenetű A/D-átalakítóval. Az A/D-átalakító kimeneti vektorát 32-bites, kettes-komplemens kódolású fixpontos vektorrá alakítjuk át, amelyben az egészrész ábrázolására 12, a törtrészére 20 bitet szánunk.



5-15. ábra. *Egyszerű hőmérséklet-érzékelő jelének digitalizálása*[214]

A hőmérséklet *(T)* és a belső számábrázolás formátumának megfelelően tárolt decimális érték *($D_{frac,10}$)* közötti összefüggés a következő:

[214] Az ábrán látható meghajtófokozat egy műveleti erősítővel megvalósított különbségképző áramkör, amely a hőmérséklet-érzékelő kimeneti jeléből 0,1 V-ot kivonva tolja el a mérendő tartományt úgy, hogy annak legkisebb szélsőértéke nulla legyen. Megjegyzendő, hogy a 0,1 V-os feszültséget előállító R_5/R_6 feszültségosztót terheletlennek feltételeztük ($I_{(-)} = 0$), ami csupán közelítőleg igaz, de ez a pontatlanság a példa szempontjából nem releváns.

$$D_{frac,10} = \frac{\boldsymbol{D_{int,10}}}{2^3}$$

$$D_{frac,10} = \frac{2^{14} \times \frac{\boldsymbol{V_A}}{V_{REF}}}{2^3}$$

$$D_{frac,10} = \frac{2^{14} \times \frac{3 \times (\boldsymbol{V_T} - 0{,}1)}{V_{REF}}}{2^3}$$

$$D_{frac,10} = \frac{2^{14} \times \frac{3 \times [(0{,}01 \times T + 0{,}5) - 0{,}1]}{V_{REF}}}{2^3}$$

Ez a bonyolultnak tűnő összefüggés lineáris függvények kompozíciója, amely az alábbi alakra egyszerűsíthető:

$$D_{frac,10} = 12{,}288 \times T + 491{,}52$$

Látható tehát, hogy a $D_{frac,10}$ vektor decimális értéke nem egyezik meg a hőmérséklet °C-ban kifejezett értékével, $D_{frac,10}$ tehát nem a hőmérsékletet, hanem egy azzal arányos mennyiséget ábrázol. A hőmérsékletet is bemenetként felhasználó további számítások során az A/D-átalakításnak ezt a sajátosságát is figyelembe kell vennünk.

Az analóg és digitális részegységek szoros együttműködésén alapuló adatfeldolgozó rendszerek (pl. szabályozó áramkörök, digitális jelfeldolgozás) tervezésekor jelentősen megkönnyítheti, átláthatóbbá teheti a munkát, ha minden digitalizált analóg mennyiséget egy konzisztens SI mértékegységrendszer szerint ábrázolunk, tehát ha a mért feszültségek belső számábrázolás szerint megjelenített értéke mindig voltban, az áramoké amperben, a teljesítményeké Wattban stb. értendő. A fenti gondolatmenet alapján ez úgy tehető meg, hogy a belső számábrázolás szerint képzett $D_{frac,10}$ vektort a további számítások elvégzése előtt egy újabb lineáris transzformációnak vetjük alá, amely a mérőrendszer minden hozzáadott lineáris transzformációjának (sőt ráadásként az A/D-átalakító erősítési és ofszethibájának) a hatását „kompenzálva” a kiválasztott konzisztens SI mértékegységrendszernek megfelelően fejezi ki az adott mennyiséget. Ha például az 5-15. ábra példájában mért hőmérséklettel közvetlenül °C-ban szeretnénk tovább számolni, akkor a $D_{frac,10}$ vektorban tárolt értéken az alábbi (vissza)transzformálást kell elvégezni:

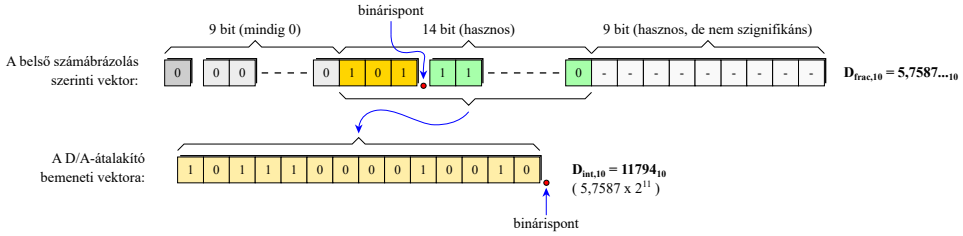
$$D'_{frac,10}[°C] = \frac{D_{frac,10} - 491{,}52}{12{,}288} = 0{,}08138 \times D_{frac,10} - 40$$

Ennek a transzformációnak – és természetesen az összes további számítási műveletnek – az elvégzéséhez fixpontos aritmetikai áramkörökre van szükségünk (lásd pl. 4.5.1. pont).

D/A-átalakítás ◆ Ahogy azt az A/D-átalakítással kapcsolatban említettük, az adatfeldolgozó rendszerek belső számábrázolása tipikusan több bitet használ, mint amennyit az A/D- és D/A-átalakítók kezelni képesek. A D/A-átalakítás esetén tehát egy analóg mennyiséggé konvertálandó digitális vektor bitjeit kénytelenek vagyunk egy kisebb méretű vektorrá alakítani, amelyet már közvetlenül a D/A-átalakító digitális bemenetére kapcsolhatunk. Erősen alkalmazásfüggő, hogy ezt a méretbeli redukciót miként érdemes végrehajtani, az A/D-átalakításnál bemutatott, a pontosságot maximalizáló általános sémához hasonló módszer nem igazán fogalmazható meg.

Az alkalmazástól való függés itt azt jelenti, hogy egy analóggá konvertálandó digitális értékről a konkrét adatfeldolgozó rendszer ismeretében állapítható csak meg, hogy – a belső számábrázolás által lehetővé tett – tág értékkészletének melyik tartománya az, amelyet a D/A-átalakító aránylag szűk bemeneti tartományára le kell képeznünk.

Példaként tekintsünk egy olyan rendszert, amelyben az analóggá alakítandó digitális vektor 32-bites, kettes-komplemens kódolású fixpontos formátumú, amelyben az egészrész ábrázolására 12, a törtrészére 20 bitet szánunk. Egy ilyen formátumban felírt értéket szeretnénk egy 14-bites, unipoláris D/A-átalakítóval analóg feszültséggé konvertálni. Ebben a formátumban az ábrázolható tartomány a -2048 ... (+2048 – 2^{-20}), de a rendszerről tudjuk, hogy az analóggá konvertálandó bináris vektor értéke mindig nagyobb nullánál, és a nyolcat sosem éri el. Ez azt jelenti, hogy a 32-bites vektor 12-bites egészrészének felső 9 bitje mindig nulla, ezek a bitek tehát nem hordoznak számunkra hasznos információt. A D/A-átalakítóra tehát az egészrész alsó három bitjét és a legmagasabb helyiértékű törtrész-biteket érdemes kapcsolnunk (lásd 5-16. ábra).



5-16. ábra. *32-bites, korlátozott értékkészletű bináris vektor leképezése egy 12-bites D/A-átalakító bemenetére*

A számunkra hasznos, a D/A-átalakítóra kapcsolandó bitek kiválasztása ezúttal is a binárispont relatív helyzetének megváltozásával jár, ami ebben az esetben is egy kettő valamely hatványával való szorzásnak vagy osztásnak felel meg. Ezt a skálázási faktort, amely pusztán a belső számábrázolás és a D/A-átalakító bitszélességének eltéréséből fakad, a rendszer megtervezése során figyelembe kell vennünk. Példánkban a D/A-átalakító analóg kimenetén előálló érték a következő:

$$V_A = \frac{D_{int,10}}{2^{14}} \times V_{REF}$$

$$V_A = \frac{D_{frac,10} \times 2^{11}}{2^{14}} \times V_{REF}$$

Telítődéses aritmetika ◆ Fixpontos számábrázolás esetén sokszor célszerű az ún. telítődéses (szaturációs) aritmetika megvalósítása is. Ez azt jelenti, hogy ha egy aritmetikai áramkör által végzett művelet eredménye kívül esne a számábrázolási tartományon, akkor a kimenetre a számformátum által lehetővé tett legnagyobb/legkisebb értéket kapcsoljuk. Ha például a nemnegatív, négybites formátumban ábrázolt 13-hoz (1101_2) hozzáadunk 5-öt (0101_2), akkor telítődéses aritmetika esetén a kimeneten 15 (1111_2) lesz ahelyett, hogy a számláló „átfordulna”, és 2-t (0010_2) adna eredményül. Az eredmény ettől persze nem lesz helyes, de a hiba abszolútértéke kisebb (ebben a konkrét esetben 3 a 16 helyett), ami bizonyos alkalmazásokban már elfogadható lehet.[215]

[215] Digitális jelfeldolgozási algoritmusok esetén az ilyen átfordulások a teljes rendszer jel/zaj-viszonyának ugrásszerű és nagymértékű csökkenését okozzák. Fogalmazhatunk úgy is, hogy a telítődéses aritmetika egy DSP rendszerben kisebb felharmonikus torzítást biztosít.

5.3.2. Lebegőpontos számok

A fixpontos számábrázolás nagy hátránya, hogy adott bitszám esetén az ábrázolható tartomány és a pontosság csak egymás rovására növelhető. Az 5-52. táblázat adott kettes-komplemens kódolású fixpontos formátum esetén az ábrázolható tartomány nagyságát és a pontosságot szemlélteti.

Fixpontos formátum		Legkisebb ábrázolható érték	Legnagyobb ábrázolható érték	Pontosság
I	F			
4	4	-8	7,9375	0,0625
4	8	-8	7,99609375	0,00390625
4	12	-8	7,999755859375	$2{,}44140625 \times 10^{-4}$
4	16	-8	7,99998474121094	$1{,}52587890625 \times 10^{-5}$
8	4	-128	127,9375	0,0625
8	8	-128	127,99609375	0,00390625
8	12	-128	127,999755859375	$2{,}44140625 \times 10^{-4}$
8	16	-128	127,999984741211	$1{,}52587890625 \times 10^{-5}$
12	4	-2048	2047,9375	0,0625
12	8	-2048	2047,99609375	0,00390625
12	12	-2048	2047,99975585937	$2{,}44140625 \times 10^{-4}$
12	16	-2048	2047,99998474121	$1{,}52587890625 \times 10^{-5}$
16	4	-32.768	32.767,9375	0,0625
16	8	-32.768	32.767,99609375	0,00390625
16	12	-32.768	32.767,9997558594	$2{,}44140625 \times 10^{-4}$
16	16	-32.768	32.767,9999847412	$1{,}52587890625 \times 10^{-5}$
16	24	-8.388.608	32.767,9999999404	$5{,}96046447753906 \times 10^{-8}$

5-52. táblázat. *Különböző formátumú fixpontos számábrázolások ábrázolási tartománya és pontossága*

Bizonyos alkalmazásokban nagyon nagy és nagyon kicsi abszolútértékű számokat is ábrázolnunk kell, miközben a pontosság sem elhanyagolható szempont. Ennek a problémának a feloldásában a lebegőpontos számábrázolás nyújthat segítséget, amely a számokat normalizált alakban fejezi ki:

$$V = \pm M \times 2^{E}$$

ahol M az egészre normalizált mantissza, E pedig a karakterisztika (exponens).[216] M és E bináris ábrázolásának pontos részletei sokfélék lehetnek, ebben a pontban az ANSI/IEEE Std 754-1985 szabványban rögzített, egyszeres pontosságú bináris lebegőpontos ábrázolás megoldásait mutatjuk be részletesen. E változatban a számokat 32 biten ábrázoljuk az 5-53. táblázatban részletezett módon.

[216] A mantissza kifejezés a latin „mantissa" szóból ered, amelynek jelentése: „toldalék". A karakterisztika szavunk pedig a görög eredetű „kharaktērízein" igéből származik, amelynek jelentése: „jellemzően ábrázol".

Bitmező indexei	Bitmező azonosítója	Bitmező tartalma
31	*s*	A mantissza *(M)* előjele. 1: *M* negatív, 0: *M* pozitív.
30 ... 23	*e*	A karakterisztika nemnegatív egészként kódolva a következők szerint (eltolt/ofszet karakterisztika): $e = E + 127$ Mivel a karakterisztika ábrázolására 8 bitet tartunk fenn, ezért az *E* hatványkitevő elvben a -127 ... 128 értékeket vehetné fel. Mivel azonban az $e = 0$ érték a ±0 szám és a törtre normalizált mantissza, az $e = 255$ érték pedig a *±Inf* és a *NaN* szimbólumok ábrázolására van lefoglalva, ezért az *E* hatványkitevő legkisebb értéke -126, a legnagyobb pedig 127 lehet.
22 ... 0	*m*	Az *M* mantissza egészre normalizált abszolútértéke. Mivel az egészre normalizált $(1 \leq M < 2)$ mantissza első bitje (a binárispont előtti egyetlen bit) definíció szerint mindig 1, ezért ezt nem tároljuk el (rejtett vagy implicit bit). A 23-bites *m* bitmező tehát a mantissza abszolútértékének csak a binárisponttól jobbra eső bitjeit tartalmazza.

5-53. táblázat. *A szimpla pontosságú lebegőpontos számok felépítése az ANSI/IEEE Std 754-1985 szabvány szerint*

A *V* szám decimális értéke az 5-53. táblázatban definiált *s*, *e* és *m* nemnegatív bináris számokkal a következőképpen fejezhető ki:

$$V = (-1)^s \times \left(1 + \sum_{i=0}^{22} m_i \times 2^{(i-23)}\right) \times 2^{(e-127)}$$

ahol m_i az egészre normalizált mantissza expliciten ábrázolt részének *(m)* i-edik bitje.

Példák

1.
*Ábrázoljuk a V_{10} = **-12,75** decimális számot szimpla pontosságú bináris lebegőpontos alakban!*
Az ábrázolandó szám negatív: s = 1
Ábrázoljuk binárisan a V szám abszolútértékét: $12{,}75_{10} = 1100{,}11_2$
Az egészre normalizált mantissza: $1{,}10011_2$, tehát m = 10011000000000000000000
A karakterisztika: E = 3, tehát $e = E + 127 = 130_{10} = 10000010_2$
A teljes, 32-bites bináris lebegőpontos alak tehát:
*V_2 = 1 10000010 10011000000000000000000 = **0xC14C0000***

2.
*Fejezzük ki a **0x42280000** szimpla pontosságú bináris lebegőpontos szám értékét decimálisan!*
A lebegőpontosan ábrázolt szám bináris alakja: 0 10000100 01010000000000000000000
Az s előjelbit értéke nulla, tehát az ábrázolt szám pozitív.
$e = E + 127 = 10000100_2 = 132_{10}$
A karakterisztika: $E = e - 127 = 132 - 127 = 5$
Az egészre normalizált mantissza: $1{,}0101000\ldots_2 = 1{,}3125_{10}$
*A decimális érték tehát: $V_{10} = 1{,}3125 \times 2^5 =$ **42**. Ne ess pánikba!*

Az itt bemutatott számformátum több speciális szimbólum kódolását is megkívánja, amelyek esetén a fenti formula nem érvényes. E speciális szimbólumokat az 5-54. táblázat foglalja össze.

s	*e*	*m*	Jelentés
0	0	0	+0
1	0	0	-0
0	255	0	*+Inf* (*+**Inf**inity*, plusz végtelen).
1	255	0	*-Inf* (*-**Inf**inity*, mínusz végtelen).
0/1	0	≠0	Törtre normalizált mantissza: a rejtett bit értéke 0 (szubnormális szám).
0/1	255	≠0	*NaN* (***N**ot **a N**umber*, érvénytelen bináris kód).

5-54. táblázat. *Speciális szimbólumok a lebegőpontos számábrázolásban*

Az ANSI/IEEE Std 754-1985 szabvány a speciális értékeken végzett műveletek eredményét is definiálja, amit egy lebegőpontos aritmetikai egység tervezésekor ugyancsak figyelembe kell venni. Ezeket a kivételeket az 5-55. - 5-57. táblázatok tartalmazzák. A *±Num* kifejezés a helyes formátumú, nem speciális operandusokat jelöli.

A→ **B↓**	**NaN**	**+Inf**	**-Inf**	**+0**	**-0**	**+Num**	**-Num**
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
+Inf	NaN	+Inf	NaN	+Inf	+Inf	+Inf	+Inf
-Inf	NaN	NaN	-Inf	-Inf	-Inf	-Inf	-Inf
+0	NaN	+Inf	-Inf	+0	+0	+Num	-Num
-0	NaN	+Inf	-Inf	+0	-0	+Num	-Num
+Num	NaN	+Inf	-Inf	+Num	+Num	+Num +Inf	±Num +0
-Num	NaN	+Inf	-Inf	-Num	-Num	±Num +0	-Num -Inf

5-55. táblázat. *A speciális szimbólumokon végzett műveletek eredményei összeadás esetén*

A→ **B ↓**	**NaN**	**+Inf**	**-Inf**	**+0**	**-0**	**+Num**	**-Num**
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
+Inf	NaN	NaN	-Inf	-Inf	-Inf	-Inf	-Inf
-Inf	NaN	+Inf	NaN	+Inf	+Inf	+Inf	+Inf
+0	NaN	+Inf	-Inf	+0	-0	+Num	-Num
-0	NaN	+Inf	-Inf	+0	+0	+Num	-Num
+Num	NaN	+Inf	-Inf	-Num	-Num	±Num +0	-Num -Inf
-Num	NaN	+Inf	-Inf	+Num	+Num	+Num +Inf	±Num +0

5-56. táblázat. *A speciális szimbólumokon végzett műveletek eredményei kivonás esetén*

A→ / B↓	NaN	+Inf	-Inf	+0	-0	+Num	-Num
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
+Inf	NaN	+Inf	-Inf	NaN	NaN	+Inf	-Inf
-Inf	NaN	-Inf	+Inf	NaN	NaN	-Inf	+Inf
+0	NaN	NaN	NaN	+0	-0	+0	-0
-0	NaN	NaN	NaN	-0	+0	-0	+0
+Num	NaN	+Inf	-Inf	+0	-0	+Num +Inf	-Num -Inf
-Num	NaN	+Inf	+Inf	-0	+0	-Num -Inf	+Num +Inf

5-57. táblázat. *A speciális szimbólumokon végzett műveletek eredményei szorzás esetén*

A 32-bites, szimpla pontosságú lebegőpontos számábrázolás esetén az ábrázolható tartomány:

$$\pm[(2 - 2^{-23}) \times 2^{127}] \approx \pm 3{,}403 \times 10^{38}$$

Az ábrázolás pontossága pedig . A lebegőpontos számábrázolás által lefedett tartomány tehát meglehetősen nagy, az alkalmazások többségében pedig pontossága is kielégítő. Használata nagyon kényelmes, ennek ellenére céláramkörök RTL-tervezésekor nagyon ritkán alkalmazzuk. Ennek magyarázata, hogy a lebegőpontos aritmetikai áramkörök (lásd pl. 4.5.3. pont) meglehetősen bonyolultak, sok erőforrást igényelnek és lassúak a fixpontos számábrázolást alkalmazó megfelelőikhez képest.

5.3.2.1. Az *flp_exception_codec* modul VHDL-modellje

VHDL

```
function flp_exception_codec_function (
  ew:        integer range 3 to 11;
  mw:        integer range 4 to 52;
  operation: std_logic_vector (1 downto 0);
  a:         std_logic_vector;
  b:         std_logic_vector
) return std_logic_vector is

  variable a_exp: unsigned (ew-1 downto 0);
  variable a_man: unsigned (mw-1 downto 0);
  variable b_exp: unsigned (ew-1 downto 0);
  variable b_man: unsigned (mw-1 downto 0);
  variable ret:   std_logic_vector (ew+mw downto 0);

begin

  a_exp := unsigned(a(a'left-1 downto mw)); a_man := unsigned(a(mw-1 downto 0));
  b_exp := unsigned(b(b'left-1 downto mw)); b_man := unsigned(b(mw-1 downto 0));
  ret := (others => '1'); -- alapértelmezett kimenet: NaN

  if ( operation = "00" or operation = "01" ) then
    -- összeadásra (és kivonásra) vonatkozó utasítások
  else
    -- szorzásra vonatkozó utasítások
  end if;

  return ret;

end flp_exception_codec_function;
```

5-17. ábra. *Az flp_exception_codec VHDL függvény (1)*

Az ANSI/IEEE Std 754-1985 szabvány rögzíti, hogy speciális bemeneti szimbólumok esetén egy lebegőpontos aritmetikai egységnek milyen kimeneteket kell előállítania (lásd 5-55–57. táblázatok). Ennek a funkciónak a megvalósítására egy kombinációs hálózatot hozunk létre, amelynek viselkedését egy VHDL függvény és az ahhoz tartozó, példányosítható burkolómodul segítségével írjuk le. A függvény csak az összeadás és szorzás műveletek esetére érvényes kombinációs logikát

VHDL

```
if (
  a_exp = 0 and a_man = 0 and  -- a = +/- 0
  b_exp = 0 and b_man = 0      -- b = +/- 0
) then
  ret(ret'left) := a(a'left) and b(b'left); ret(ret'left-1 downto mw) := (others => '0');
  ret(mw-1 downto 0) := (others => '0');
end if;

if (
  (a_exp = 0 and a_man = 0) and     -- a = +/- 0
  (b_exp = (2**ew-1) and b_man = 0) -- b = +/- Inf
) then
  ret(ret'left) := b(b'left); ret(ret'left-1 downto mw) := (others => '1');
  ret(mw-1 downto 0) := (others => '0');
end if;

if (
  (b_exp = 0 and b_man = 0) and     -- b = +/- 0
  (a_exp = (2**ew-1) and a_man = 0) -- a = +/- Inf
) then
  ret(ret'left) := a(a'left); ret(ret'left-1 downto mw) := (others => '1');
  ret(mw-1 downto 0) := (others => '0');
end if;

if (
  (a_exp = 0 and a_man = 0) and       -- a = +/- 0
  (b_exp /= 0 and b_exp /= (2**ew-1)) -- b = num
) then ret := b;
end if;

if (
  (b_exp = 0 and b_man = 0) and       -- b = +/- 0
  (a_exp /= 0 and a_exp /= (2**ew-1)) -- a = num
) then ret := a;
end if;

if (
  (a_exp = (2**ew-1) and a_man = 0) and -- a = +/- Inf
  (b_exp /= 0 and b_exp /= (2**ew-1))   -- b = num
) then
  ret(ret'left) := a(a'left); ret(ret'left-1 downto mw) := (others => '1');
  ret(mw-1 downto 0) := (others => '0');
end if;

if (
  (b_exp = (2**ew-1) and b_man = 0) and -- b = +/- Inf
  (a_exp /= 0 and a_exp /= (2**ew-1))   -- a = num
) then
  ret(ret'left) := b(b'left); ret(ret'left-1 downto mw) := (others => '1');
  ret(mw-1 downto 0) := (others => '0');
end if;

if (
  (a(a'left) = '0' and a_exp = (2**ew-1) and a_man = 0) and -- a = + Inf
  (b(b'left) = '0' and b_exp = (2**ew-1) and b_man = 0)     -- b = + Inf
) then ret := a;
end if;

if (
  (a(a'left) = '1' and a_exp = (2**ew-1) and a_man = 0) and    -- a = - Inf
  (b(b'left) = '1' and b_exp = (2**ew-1) and b_man = 0)        -- b = - Inf
) then ret := a;
end if;
```

5-18. ábra. *Az flp_exception_codec VHDL függvény (2)*

VHDL

```
1  if (
2    (a_exp = (2**ew-1) and a_man = 0) and -- a = +/- Inf
3    (b_exp = (2**ew-1) and b_man = 0)     -- b = +/- Inf
4  ) then
5    ret(ret'left) := a(a'left) xor b(b'left); ret(ret'left-1 downto mw) := (others => '1');
6    ret(mw-1 downto 0) := (others => '0');
7  end if;
8
9  if (
10   (a_exp = 0 and a_man = 0) and  -- a = +/- 0
11   (b_exp = 0 and b_man = 0)      -- b = +/- 0
12 ) then
13   ret(ret'left) := a(a'left) xor b(b'left); ret(ret'left-1 downto mw) := (others => '0');
14   ret(mw-1 downto 0) := (others => '0');
15 end if;
16
17 if (
18   (a_exp = 0 and a_man = 0) and       -- a = +/- 0
19   (b_exp /= 0 and b_exp /= (2**ew-1)) -- b = num
20 ) then
21   ret(ret'left) := a(a'left) xor b(b'left); ret(ret'left-1 downto mw) := (others => '0');
22   ret(mw-1 downto 0) := (others => '0');
23 end if;
24
25 if (
26   (b_exp = 0 and b_man = 0) and       -- b = +/- 0
27   (a_exp /= 0 and a_exp /= (2**ew-1)) -- a = num
28 ) then
29   ret(ret'left) := a(a'left) xor b(b'left);
30   ret(ret'left-1 downto mw) := (others => '0');
31   ret(mw-1 downto 0) := (others => '0');
32 end if;
33
34 if (
35   (a_exp = (2**ew-1) and a_man = 0) and    -- a = +/- Inf
36   (b_exp /= 0 and b_exp /= (2**ew-1))   -- b = num
37 ) then
38   ret(ret'left) := a(a'left) xor b(b'left);
39   ret(ret'left-1 downto mw) := (others => '1');
40   ret(mw-1 downto 0) := (others => '0');
41 end if;
42
43 if (
44   (b_exp = (2**ew-1) and b_man = 0) and -- b = +/- Inf
45   (a_exp /= 0 and a_exp /= (2**ew-1))   -- a = num
46 ) then
47   ret(ret'left) := a(a'left) xor b(b'left); ret(ret'left-1 downto mw) := (others => '1');
48   ret(mw-1 downto 0) := (others => '0');
49 end if;
```

5-19. ábra. *Az flp_exception_codec VHDL függvény (3)*

valósítja meg, mert a kivonás a lebegőpontos számábrázolásban nagyon egyszerűen vezethető vissza az összeadásra (lásd 4-178. ábra: 5. sor). A VHDL függvény vázlatát az 5-17. ábra mutatja be.

A függvény a karakterisztika és a mantissza mérete *(ew: **e**xponent **w**idth, mw: **m**antissa **w**idth)*, valamint az elvégzendő művelet *(operation)* alapján állapítja meg a kimenet értékét. Az operandusok karakterisztikájának és mantisszájának elkülönítésére változókat hozunk létre *(a_exp, a_man, b_exp, b_man)* a függvény törzsében megfogalmazott, összetett *if-else* szerkezetek jobb áttekinthetősége érdekében (5-17. ábra: 17–18. sor). Az 5-18. ábra kódrészlete az 5-17. ábra 21., az 5-19. ábra pedig az 5-17. ábra 23. sorába behelyettesítendő utasításokat szemlélteti. A kombinációs hálózatot leíró függvényt példányosítható almodulként is használhatóvá tévő burkolómodult az 5-20. ábra mutatja be.

VHDL

```vhdl
entity flp_exception_codec is
generic (
  ew:         integer range 3 to 11;
  mw:         integer range 4 to 52
);

port (
  operation: in  std_logic_vector (1 downto 0);
  a, b:      in  std_logic_vector(ew+mw downto 0);
  result:    out std_logic_vector(ew+mw downto 0)
);
end entity flp_exception_codec;

architecture rtl of flp_exception_codec is
begin
  process ( operation, a, b ) begin
    result <= flp_exception_codec_function(ew, mw, operation, a, b);
  end process;
end architecture rtl;
```

5-20. ábra. *Az flp_exception_codec VHDL-modul*

5.4. A ciklikus redundanciaellenőrzés elve

Az egyazon nyomtatott huzalozású lemezen vagy egymástól távol, vezetékes vagy vezetéknélküli technikákkal összekapcsolt integrált áramkörök almoduljai közötti kommunikációban a csatorna többnyire zajjal terhelt, az összeköttetés fizikai rétege tehát nem tekinthető tökéletesnek. Egy vevőáramkör sosem lehet benne teljesen biztos, hogy az adat, amit a fizikai közvetítő közegből kinyert, tökéletesen megegyezik azzal, amit az adó elküldött, az üzenet az átvitel során sajnos sérülhet. Hasonló a helyzet akkor is, ha nem az információ átviteléről, hanem annak tárolásáról van szó; egy memóriában tárolt bejegyzés visszaolvasásakor nem lehetünk benne biztosak, hogy az a beírás óta nem változott meg, az adattároló közeg (pl. egy SRAM, DRAM, flash-cella) ui. ugyancsak kitett lehet külső zavaroknak vagy meghibásodásnak.

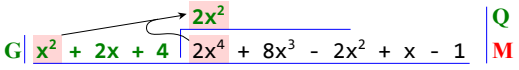
Legyen szó akár az adatok átviteléről akár azok tárolásáról, a feladat ugyanaz; a helyes működés érdekében egy összefüggő adathalmazról el kell döntenünk, hogy sérült vagy sem. A probléma kezelésére rengeteg módszer létezik, ezek képességekben (pl. hibadetektálás és/vagy javítás) és költségben (plusz hardver, számítási teljesítmény, effektív sávszélesség-csökkenés) nagyon eltérhetnek, de van egy közös vonásuk; mindegyik módszer alapja az adathalmaz redundanciája. Ez azt jelenti, hogy a hasznos forrásadatból valamilyen matematikai módszerrel előállítunk egy rá jellemző, tipikusan kisebb méretű másodlagos adathalmazt, egy ún. ellenőrzőösszeget (checksum), amit aztán a forrásadattal együtt viszünk át a kommunikációs csatornán vagy tárolunk el a memóriában. A forrásadat átvitel vagy tárolás utáni helyességéről az ellenőrzőösszeg újbóli kiszámításával győződhetünk meg; ha a potenciálisan sérült forrásadatból ugyanazzal a módszerrel ismét kiszámítjuk az ellenőrzőösszeget, és az nem egyezik meg az átvitt vagy eltárolt ellenőrzőösszeggel, akkor az adathalmazt sérültnek tekintjük. A redundáns információ előállításának a lehető legegyszerűbb – ezáltal legolcsóbb – módszerével az UART adatkereteknél (lásd 5-3. ábra) már találkoztunk, ez az ún. paritásbit, ami tulajdonképpen egy egybites, az adatkeret bitjeinek *XOR* kapcsolataként kiszámított ellenőrzőösszeg. Ez a megoldás képességeit tekintve nem túl kifinomult, páros számú hibás bit esetén például garantáltan kudarcot vall, ugyanakkor, ha feltételezzük, hogy egy

adatkereten belül csak egyetlen hiba keletkezhet, akkor a paritásbit kellő védelmet nyújt a hibás adatkeretek kiszűrésére.[217]

A nagyobb adathalmazokban keletkező, véletlenszerűen elhelyezkedő bithibák kiszűrésére az egyik legelterjedtebb módszer az ún. ciklikus redundanciaellenőrzés (**C**yclic **R**edundancy **C**heck, CRC). Ez a megoldás úgy számítja ki az ellenőrzőösszeget, hogy a forrásadat bitjeit egy polinom együtthatóinak tekinti. Ezt a polinomot egy másik polinommal, az ún. generátor-polinommal elosztva egy hányados- és egy maradék polinom keletkezik. A forrásadathoz fűzött ellenőrzőösszeg nem más, mint e maradék polinom együtthatóiból képezett bináris vektor, amit CRC-kódnak nevezünk. A CRC-kód előállításának paraméterei a generátor-polinom együtthatóinak sora, illetve a polinom mérete, ami egyúttal a CRC-kód méretét is meghatározza; a generátor-polinom együtthatóinak száma mindig eggyel nagyobb a generált CRC-kód bitjeinek számánál.

A CRC-kód tehát bináris együtthatójú polinomok osztásával számítható ki. Akár egy processzoron futó szoftver, akár dedikált digitális hardverelemek segítségével valósítjuk meg, a CRC-kód kiszámítása hatékonyan ún. visszacsatolt léptetőregiszter-struktúrával végezhető el. Ennek az állításnak az alátámasztására az alábbiakban először egy példán keresztül mutatjuk be két decimális együtthatójú polinom maradékos osztásának algoritmusát, majd az algoritmus bináris együtthatójú változatát is górcső alá vesszük, hogy lássuk, sajátosságai hogyan vezetnek el a visszacsatolt léptetőregiszterekhez. E rövid matematikai levezetés célja, hogy elkészítsük a CRC-kódgeneráló funkcionális egységnek egy olyan modelljét, amely alkalmas arra, hogy az RTL-tervezés alapjául szolgáljon. Az e modell alapján készült IP-mag részleteit a 4.5.4. pontban tárgyaljuk.

Decimális együtthatójú polinomok maradékos osztása ◆ Példánkban az M negyedfokú polinomot osztjuk el a G másodfokú polinommal. A maradékos osztás algoritmusának első lépése, hogy (1) M legmagasabb fokszámú tagját osztjuk G legmagasabb fokszámú tagjával (lásd 5-21. ábra).

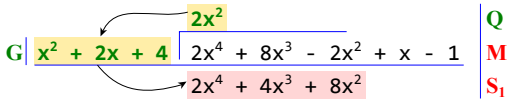


5-21. ábra. *Decimális együtthatójú polinomok maradékos osztása (1)*

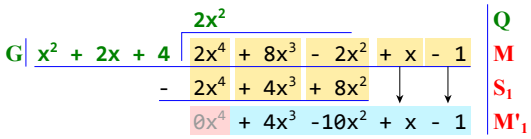
A két tag hányadosa $2x^2$, ez lesz a Q hányados polinom legmagasabb fokszámú tagja. Az algoritmus második lépéseként (2) a teljes G polinomot felszorozzuk a Q polinom legutóbb meghatározott tagjával, ez esetben $2x^2$-tel. E lépés eredményeként egy olyan S_1 polinomot kapunk, amelynek fokszáma, sőt legmagasabb fokszámú tagjának együtthatója is megegyezik M-ével, de a teljes S_1 polinom kisebb M-nél (lásd 5-22. ábra).

Ez alapján kimondhatjuk, hogy M-ben G biztosan megvan $2x^2$-szer. Állapítsuk meg, hogy M mennyivel nagyobb, mint G és Q szorzata; harmadik lépésként (3) vonjuk ki M-ből S_1-et (lásd 5-23. ábra).

[217] Azt a feltételezést, hogy egy adatkereten belül csupán egy bithiba keletkezhet, „hibamodellnek" nevezzük. Hibamodellt a kommunikációs interfész fizikai rétegének és/vagy a környezeti körülményeknek az ismeretében alkothatunk, az ellenőrzőösszeg képességeire vonatkozó elvárásainkat pedig e hibamodellel összhangban kell megfogalmaznunk. A paritásbit mérsékelten hatékony ugyan, de költsége csekély, és az alkalmazások jelentős részében a védelemnek ez a szintje megfelelő, sőt olykor még túlzás is (pl. egy asztali PC-t egy laborkörülmények között felépített mérésiadatgyűjtő-rendszerrel összekötő soros interfész esetén).

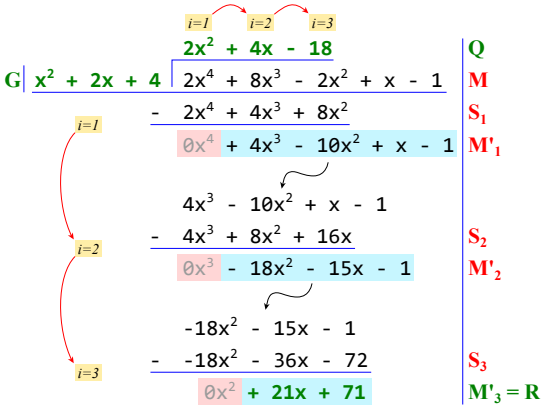


5-22. ábra. *Decimális együtthatójú polinomok maradékos osztása (2)*



5-23. ábra. *Decimális együtthatójú polinomok maradékos osztása (3)*

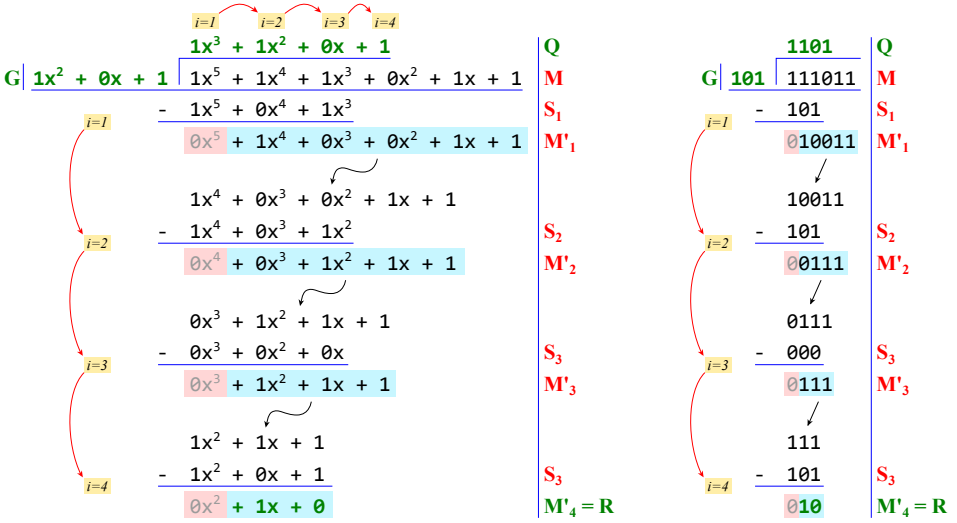
A kivonás eredményéül kapott M'_1 polinom legmagasabb fokszámú tagjának együtthatója mindig nulla lesz, tehát M'_1 fokszáma eggyel kisebb, mint M fokszáma. Ugyanakkor, ha M'_1 fokszáma még így is nagyobb, mint G fokszáma, akkor a Q hányados-polinomot további tagokkal kell bővítenünk. Ekkor az osztás algoritmusa az első három lépés ismétlésével folytatódik oly módon, hogy az i-edik lépésben kapott M'_i polinom az (i+1)-edik lépésben az osztandó szerepét tölti be, az új osztandó polinom legmagasabb fokszámú tagjának és G legmagasabb fokszámú tagjának hányadosát pedig minden lépésben hozzáadjuk a Q hányados polinom korábban meghatározott értékéhez (lásd 5-24. ábra).



5-24. ábra. *Decimális együtthatójú polinomok maradékos osztása (4)*

Az iteratív számítás akkor ér véget, amikor a különbségképzés eredményéül kapott M'_i polinom fokszáma kisebb lesz, mint G fokszáma, vagyis M'_i tovább már nem osztható G-vel. Ekkor Q (***Q**uotient*) az M és G polinomok osztásából származó hányados polinom, $M'_i = R$ (***R**emainder*) pedig a maradék polinom.

Bináris együtthatójú polinomok maradékos osztása ◆ A fenti osztás algoritmus kettes számrendszerbeli együtthatók esetén is változatlan formában használható, ezt szemlélteti az 5-25. ábrán látható példa.



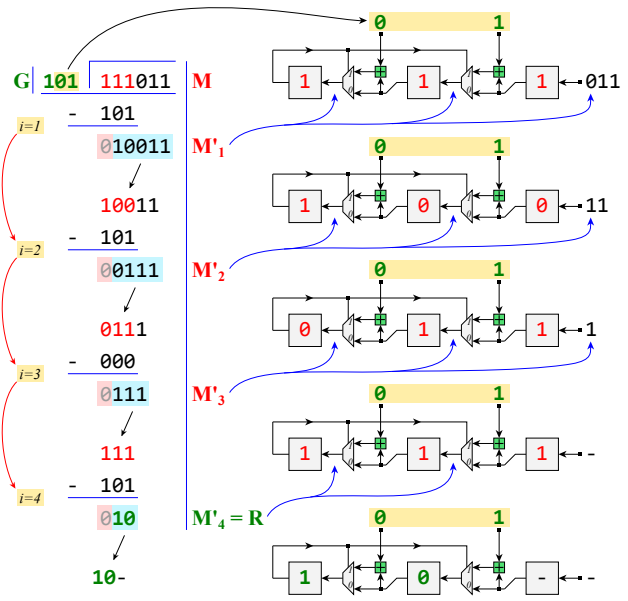
5-25. ábra. *Bináris együtthatójú polinomok maradékos osztása*

Két polinom kivonásakor az azonos kitevőjű tagok együtthatóit vonjuk ki egymásból, így kapjuk meg a különbség polinomnak az adott kitevőhöz tartozó együtthatóját. A kivonás műveletnek ez a definíciója számrendszertől független, ami speciálisan a bináris együtthatójú polinomok esetén azt jelenti, hogy a kivonás lépésben előálló különbség polinom együtthatói tulajdonképpen az M'_i polinom és az S_{i+1} polinom azonos kitevőjű tagjaihoz tartozó bináris együtthatóknak a logikai *XOR* kapcsolataként is felfoghatók.[218] Ezt jól szemlélteti az 5-25. ábra jobb oldalán látható leírás, ami a bal oldali polinomosztásnak egy olyan reprezentációja, amiben kizárólag a polinomok együtthatóit tüntettük fel, így a polinomokat tulajdonképpen egy-egy bináris vektor képviseli. Ennek a leírási módszernek a segítségével már közvetlenül megfigyelhető, hogy a bináris együtthatójú polinomosztás algoritmusa egy visszacsatolt léptetőregiszter-struktúrával (**L**inear **F**eedback **S**hift **R**egister, LFSR) igen hatékonyan valósítható meg (lásd 5-26. ábra).

Az osztandó polinomot *(M)* reprezentáló eredeti bitsorozatot a legfelső helyiértékű bittől kezdődően töltjük be a nullára inicializált léptetőregiszterbe. Amikor a léptetőregiszter legfelső helyiértékű bitje – az M'_i polinom legmagasabb kitevőjű tagjának együtthatója – 1 értékű, akkor a következő léptetésnél a multiplexereken keresztül az aktuális M'_i vektor és a *G* vektor *XOR* kapcsolatának eredménye, vagyis a két vektor által leírt polinomok különbség polinomjának együtthatói íródnak be a léptetőregiszterbe. Ha pedig a léptetőregiszter legfelső helyiértékű bitje nulla értékű, akkor a multiplexerek egyszerűen balra léptetik az M'_i vektor bitjeit. A generátor polinom fokszámát meghatározó, legmagasabb helyiértékű bit definíció szerint mindig 1 (ha 0 lenne, akkor a fokszám kisebb lenne), ami e bit tárolását feleslegessé teszi.[219] Ezért, amikor a generátor polinomot bináris vektor alakban adjuk meg, akkor a legfelső helyiértékű bitet nem „nevezzük nevén”, hiszen tudjuk, hogy az mindig 1; a nyolcbites *G = 0xD5* = $\mathbf{11010101}_2$ szám tehát valójában a nyolcadfokú

[218] A bináris együtthatók különbsége, összege és logikai *XOR* kapcsolata mind ugyanazt az eredményt adja. A logikai *XOR* kapcsolatot ezért sok forrás a „+” szimbólummal jelöli.

[219] Ez a polinomosztás algoritmusának abból a sajátosságából adódik, hogy a harmadik lépésben a kivonás eredményéül kapott M'_i polinom legmagasabb fokszámú tagjának együtthatója mindig nulla.



5-26. ábra. *Visszacsatolt léptetőregiszter a bináris együtthatójú polinomosztás megvalósítására*

($\mathbf{1}x^8 + \mathbf{1}x^7 + \mathbf{1}x^6 + \mathbf{0}x^5 + \mathbf{1}x^4 + \mathbf{0}x^3 + \mathbf{1}x^2 + \mathbf{0}x + \mathbf{1}$) generátor polinomot jelöli. Figyeljük meg, hogy ez a struktúra nem foglalkozik az osztásművelet hányadosával, hiszen a CRC-ellenőrzés alapelve szerint számunkra csak az osztás maradéka érdekes. Ezt a maradékot a bemenetre érkező bitek feldolgozása után a léptetőregiszter felső *N* db tárolójából olvashatjuk ki, ahol *N* mindig eggyel kisebb, mint a *G* vektor mérete.

Az 5-26. ábrán látható visszacsatolt léptetőregiszter-struktúra egy másodfokú, bináris együtthatójú generátor polinommal állít elő egy CRC-2 kódot, de a megoldás könnyen általánosítható tetszőleges fokszámú generátor polinomokra, ami egyúttal tetszőleges méretű CRC-kód előállításának a lehetőségét is magában hordozza. A visszacsatolt léptetőregiszterre épülő megoldást parametrizált formában megvalósító RTL-modellt a 4.5.4. pontban mutatjuk be.

5.5. Gray-enkóder és -dekóder

Az 5-27. ábra a Gray-enkóder és -dekóder áramkörök függvénnyel megvalósított, parametrikus Verilog modelljeit, valamint azok 4-bites kivitelének kapcsolási sémáit szemlélteti (lásd még DRMSz: 85–86. oldal).

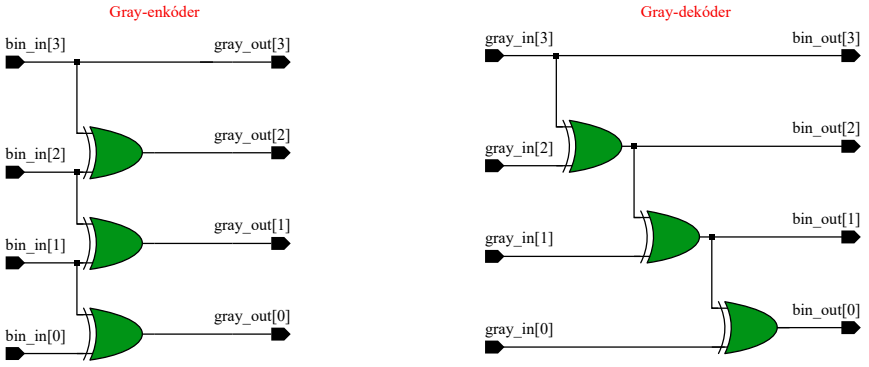
Verilog – Gray-enkóder

```
function [aw:0] gray_encoder;
  input [aw:0] bin_in;
  logic [aw:0] tmp;
begin
  tmp = 0;
  for ( int i = 0; i <= aw-1; i = i+1 )
    tmp[i] = bin_in[i] ^ bin_in[i+1];
    tmp[aw] = bin_in[aw];
  gray_encoder = tmp;
end

endfunction
```

Verilog – Gray-dekóder

```
function [aw:0] gray_decoder;
  input [aw:0] gray_in;
  logic [aw:0] tmp;
begin
  tmp = 0;
  tmp[aw] = gray_in[aw];
  for ( int i = aw-1; i >= 0; i = i-1 )
    tmp[i] = tmp[i+1] ^ gray_in[i];
  gray_decoder = tmp;
end

endfunction
```

Áramköri struktúra (4-bites bemenet esetén)



527. ábra. *Gray-enkóder és -dekóder Verilog-leírása*

5.6. Az FFT-algoritmus

A Fourier-analízis alapvetése, hogy a periodikusan változó jelek közelíthetők különböző amplitúdójú, frekvenciájú és fázisú szinuszos függvények szuperpozíciójával:

$$f(t) \approx A_0 + \sum_{k=1}^{K} A_k \cos\left(k\frac{2\pi}{T}t + \varphi_k\right) \tag{5.6-1}$$

Az összefüggés jobb oldalán szereplő kifejezést az *f(t)* függvény Fourier-összegének nevezzük. Számítástechnikai szempontból sokszor célszerűbb a Fourier-összegnek a komplex alakját használni, amelyben a szinuszos összetevőket egy-egy fazor írja le:

$$f(t) = \frac{1}{T}\sum_{k=-K}^{K} C_k e^{jk\frac{2\pi}{T}t} \tag{5.6-2}$$

Az összeget felépítő fazorok C_k komplex csúcsértékei, az ún. Fourier-együtthatók a

$$C_k = \int_0^T f(t)e^{-jk\frac{2\pi}{T}t}dt \tag{5.6-3}$$

integrállal számíthatók.

Ahhoz, hogy egy digitális rendszer a külvilágban mért bármilyen analóg mennyiséggel kapcsolatos számításokat tudjon végezni, a mennyiséget analóg-digitális átalakításnak kell alávetni. Az A/D-átalakítás eredménye a mennyiség értékének valamilyen bináris reprezentációja. Időben változó mennyiségek esetén a változás dinamikája sorozatos, rögzített időközönként végrehajtott mérések sokaságával vizsgálható. Az egymás után végrehajtott mérések és A/D-átalakítások eredménye egy sor bináris vektor, ezeket a mennyiség mintáinak, a minták összességét pedig diszkrét idejű jelnek nevezzük. Tegyük fel, hogy egy periodikusan változó jel egyetlen periódusából N db mintát veszünk; ebben az esetben a mintavett jelet egy N elemű diszkrét mintasor írja le. Az 5.6-2 összefüggéshez hasonlóan ez a periodikus jel ugyancsak felírható különböző amplitúdójú, frekvenciájú és fázisú diszkrét szinuszos jelek szuperpozíciójaként:

$$x[n] = \frac{1}{N}\sum_{k=0}^{N-1} C_k e^{jk\frac{2\pi}{N}n} \tag{5.6-4}$$

Az 5.6-4 összefüggésben a $C_k e^{jk\frac{2\pi}{N}n}$ fazor minden k-ra egy-egy $\left(\frac{2\pi}{N}\right)\times k$ körfrekvenciájú szinuszos komponenst ír le, aminek amplitúdóját és kezdőfázisát a C_k komplex csúcsérték abszolútértéke és szöge határozza meg. A komplex csúcsértékek a következőképpen írhatók fel:

$$C_k = \sum_{n=0}^{N-1} x[n]e^{-j\frac{2\pi}{N}nk} = \sum_{n=0}^{N-1} x[n]W_N^{nk}, \qquad k = 0\ldots(N-1) \tag{5.6-5}$$

A komplex csúcsértékeket Fourier-együtthatóknak, az ezeket tartalmazó N elemű vektort pedig a jel diszkrét Fourier-transzformáltjának (**D**iscrete **F**ourier **T**ransform, DFT) nevezzük. A diszkrét Fourier-transzformált 5.6-5 szerinti, közvetlen előállításához N^2 db komplex szorzást és összeadást kell elvégezni, ami meglehetősen nagy számítási igényt jelent. E számítási igény jelentősen csökkenthető az ún. FFT (**F**ast **F**ourier **T**ransform) algoritmus alkalmazásával, amennyiben N kettő egész kitevőjű hatványa.[220] A C vektor elemeinek 5.6-5 kifejezését bontsuk két részre x páros és páratlan indexű mintái szerint *($n = 2r$)*:

$$C_k = \sum_{n=0}^{N-1} x[n]W_N^{nk} = \sum_{r=0}^{N/2-1} x[2r]W_N^{2rk} + \sum_{r=0}^{N/2-1} x[2r+1]W_N^{(2r+1)k} \tag{5.6-6}$$

$$C_k = \sum_{r=0}^{N/2-1} x[2r](W_N^2)^{rk} + W_N^k \sum_{r=0}^{N/2-1} x[2r+1](W_N^2)^{rk} \tag{5.6-7}$$

Mivel W egy egységnyi abszolútértékű komplex szám, ezért

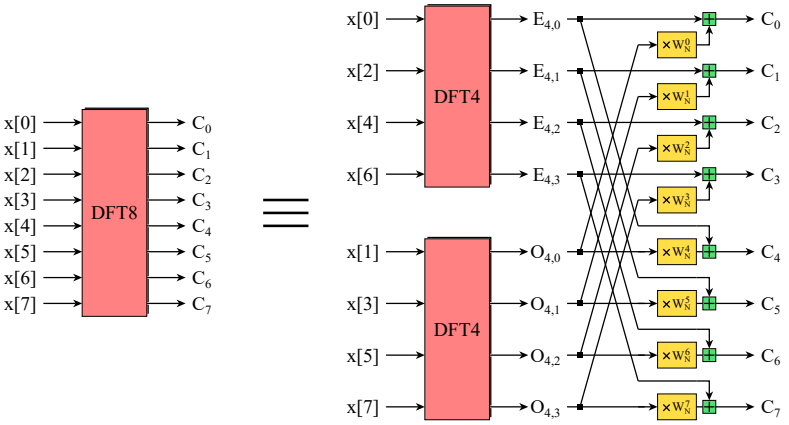
$$W_N^2 = W_{N/2} \tag{5.6-8}$$

tehát a C_k komplex együtthatók felírhatók a következő alakban:

[220] Ez a megkötés a gyakorlatban egyáltalán nem jelent nehézséget.

$$C_k = \sum_{r=0}^{N/2-1} x[2r] W_{N/2}^{rk} + W_N^k \sum_{r=0}^{N/2-1} x[2r+1] W_{N/2}^{rk} \tag{5.6-9}$$

Ha összehasonlítjuk az 5.6-9 és 5.6-5 összefüggéseket, észrevehetjük, hogy 5.6-9-ben a zölddel és kékkel kiemelt összegek tulajdonképpen rendre az *N* elemű mintasor páros és páratlan indexű elemeiből képzett, *N/2* elemű mintasorok diszkrét Fourier-együtthatói. Az *N* elemű DFT-t tehát *N/2* elemű, ún. résztranszformált vektorokkal fejeztük ki. Ezt szemlélteti az 5-28. ábra *N=8*-ra.



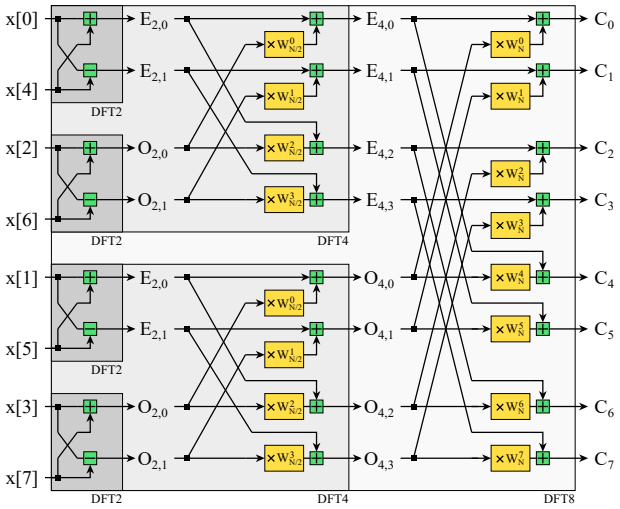
5-28. ábra. *A 8 elemű DFT kiszámítása a páros és páratlan indexű minták 4 elemű DFT-i alapján*

Az ábra jobb oldalán látható, hogy az *N* elemű transzformált vektor előállításához most két *N/2* elemű transzformáltat kell kiszámítanunk. Az *N* elemű transzformált vektor *N/2* elemű résztranszformáltakból való előállítása természetesen *N*-től függetlenül alkalmazható számítástechnikai trükk, amit rekurzívan mindaddig folytathatunk, ameddig az elvégzendő művelet egy kételemű mintasoron végzett DFT-vé nem egyszerűsödik:

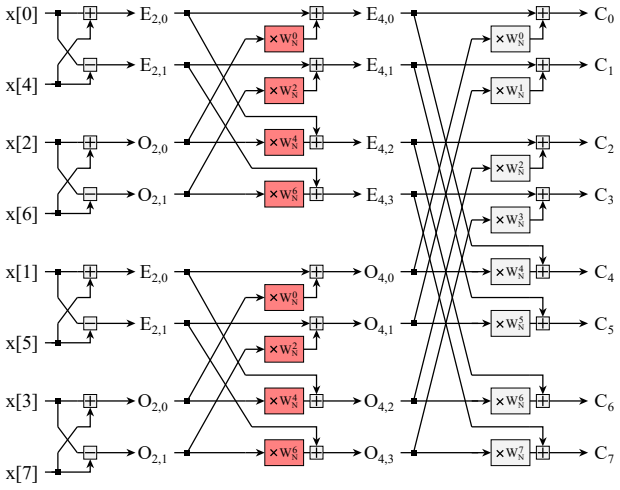
$$C_k = \sum_{n=0}^{1} x[n] W_2^{nk} \rightarrow C_0 = x[0] + x[1],\ C_1 = x[0] - x[1] \tag{5.6-10}$$

N=8 esetén a rekurzív eljárást az 5-29. ábra szemlélteti. A rekurzió alkalmazásával az *N* elemű (komplex) mintasor diszkrét Fourier-transzformáltjának kiszámításához szükséges lépésszám így csupán $(N \times \log_2 N)$-nel arányos, ami nagy elemszámok esetén igen jelentős számításiigény-csökkenés a definíció szerinti algoritmushoz képest, ahol ez a lépésszám N^2-tel arányos.

Az 5-29. ábrán látható, hogy W_N és $W_{N/2}$ hatványait egyaránt felhasználjuk a számítás során. Ismét kihasználjuk, hogy *W* egy egységnyi abszolútértékű komplex szám; mivel $W_{N/2} = W_N^2$, ezért további egyszerűsítésre van lehetőség (lásd 5-30. ábra).



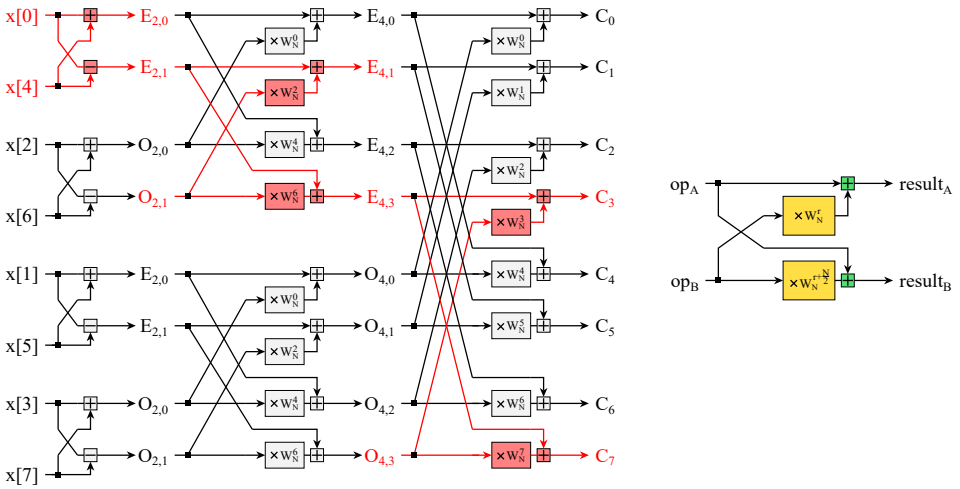
5-29. ábra. *N/2 elemű résztranszformáltak felhasználása N elemű DFT kiszámításához*



5-30. ábra. *A DFT kiszámításának további egyszerűsítése W tulajdonságainak kihasználásával*

A 8 elemű DFT kiszámítását szemléltető ábrákon jól megfigyelhető az FFT-algoritmusnak az a sajátos lépése, amikor két *N/2* elemű résztranszformált egy-egy eleméből előáll az *N* elemű résztranszformált két eleme (lásd 5-31. ábra).

Ezt az egyedi műveletet lepkeműveletnek (butterfly) hívjuk. Az ábra alapján könnyen belátható, hogy az FFT-algoritmussal a diszkrét Fourier-transzformált kiszámítása $(N/2) \times \log_2 N$ lepkeműveletet igényel, ami $N \times \log_2 N$ komplex szorzást és ugyanennyi komplex összeadást jelent. A komplex szorzások száma még tovább csökkenthető, ha ismét figyelembe vesszük a *W*



5-31. ábra. *A lepkeművelet (butterfly-operation)*

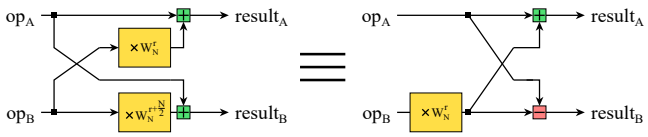
operandus hatványainak – amiket a lepkeművelet kontextusában forgatási együtthatóknak nevezünk[221] – sajátosságait:

$$W_N^{r+\frac{N}{2}} = W_N^r \times W_N^{\frac{N}{2}} = W_N^r \times e^{-j\frac{2\pi N}{N\,2}} = W_N^r \times e^{-j\pi} = -W_N^r \tag{5.6-11}$$

A lepkeművelet tehát a következő alakban írható fel:

$$\begin{aligned} result_A &= op_A + W_N^r \times op_B \\ result_B &= op_A - W_N^r \times op_B \end{aligned} \tag{5.6-12}$$

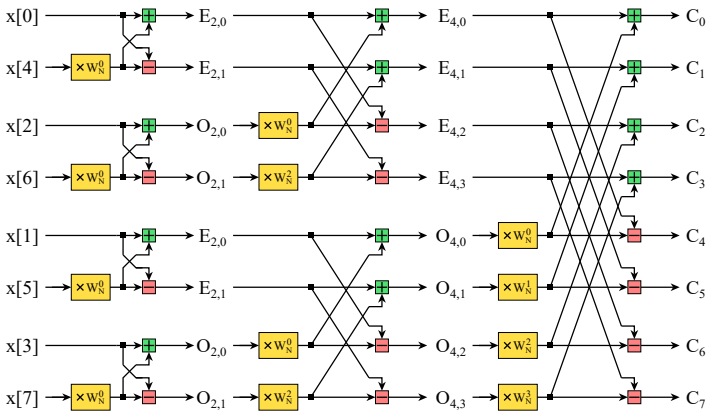
Az FFT-algoritmus magját képező műveletben tehát valójában csak egyetlen komplex szorzást kell elvégezni, ahogy azt az 5-32. ábra szemlélteti.



5-32. ábra. *A lepkeművelet optimalizálása*

Ezt az optimalizációs lehetőséget kihasználva a 8 elemű DFT kiszámítása az 5-33. ábrán látható alakot ölti.

[221] A W komplex szám definíció szerint egy egységvektor ($e^{-j\frac{2\pi}{N}}$), az ennek r-edik hatványaival való szorzás a szorzandó vektort $\left(\frac{2\pi}{N}r\right)$ radiánnal elforgatja.



5-33. ábra. *A DFT kiszámításának egyszerűsítése a lepkeművelet optimalizálásával*

Az FFT-algoritmus a diszkrét Fourier-transzformált előállításának egy igen hatékony módja. Ennek a hatékonyságnak a kulcsa – ahogy az algoritmus fenti levezetéséből is jól látszik – végső soron a harmonikus függvények szimmetriáiban rejlik. Ezeket a szimmetriákat továbbra is kihasználva megmutatjuk, hogy az FFT-algoritmus felhasználásával a diszkrét Fourier-transzformáltból maga a mintasor is visszaállítható, vagyis az inverz Fourier-transzformáció megvalósításához nem kell egy teljesen más eljárást kidolgoznunk. Induljunk ki az 5.6-4 összefüggésből, és vegyük az egyenlet mindkét oldalának a komplex konjugáltját:

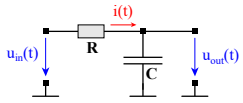
$$x[n] = \frac{1}{N}\sum_{k=0}^{N-1} C_k e^{jk\frac{2\pi}{N}n} \rightarrow (x[n])^* = \frac{1}{N}\sum_{k=0}^{N-1} C_k{}^* e^{-jk\frac{2\pi}{N}n} \tag{5.6-13}$$

Az 5.6-5 és 5.6-13 formulák összevetésével látható, hogy a piros színnel kiemelt kifejezés tulajdonképpen nem más, mint a vektor diszkrét Fourier-transzformáltja, amelyet természetesen az FFT-algoritmussal is előállíthatunk:

$$x[n] = \frac{1}{N} FFT(C_k^*)^* \tag{5.6-14}$$

5.7. Az elsőfokú RC aluláteresztő szűrő diszkretizálása

Az 5-34. ábra az elsőfokú RC aluláteresztő szűrő kapcsolási sémáját szemlélteti.



5-34. ábra. *Elsőfokú RC aluláteresztő szűrő*

Az $u_{in}(t)$ bemeneti feszültség hatására kialakuló $i(t)$ áram értéke a C kapacitással és a kimeneti feszültség megváltozásának sebességével arányos. Ez az áram az R ellenálláson $Ri(t)$ feszültséget ejt, így az RC-tag $u_{out}(t)$ kimeneti feszültsége a következőképpen fejezhető ki:

$$u_{out}(t) = u_{in}(t) - Ri(t) = u_{in}(t) - RC\frac{du_{out}}{dt}(t) \tag{5.7-1}$$

A folytonos idejű egyenlet diszkretizálásakor a visszalépő Euler-módszert használjuk, ami az $y(t)$ függvény t pontbeli deriváltját a

$$\frac{dy}{dt}(t) = \frac{y[n] - y[n-1]}{T_S} \tag{5.7-2}$$

összefüggés szerint határozza meg, ahol $y[n-1]$ és $y[n]$ rendre a függvény (n-1)-edik és n-edik időpillanatban vett mintája, T_S pedig a két mintavétel között eltelt idő, vagyis a mintavételi periódus. A kimeneti feszültséget ezek alapján a következőképpen írhatjuk fel:

$$u_{out}[n] = u_{in}[n] - RC\frac{u_{out}[n] - u_{out}[n-1]}{T_s} \tag{5.7-3}$$

Az egyenlet átrendezésével a következő alakhoz juthatunk:

$$u_{out}[n] = \frac{T_S}{T_S + RC}u_{in}[n] + \frac{RC}{T_S + RC}u_{out}[n-1] \tag{5.7-4}$$

A T_S mintavételi idő, valamint az R ellenállás és a C kapacitás a mintavételes rendszer jellemzői. Az ezekből képzett együtthatók jelölésére bevezetjük az A és B szimbólumokat, amelyekkel a kimeneti feszültségre felírt összefüggés az alábbi alakot ölti:

$$u_{out}[n] = Au_{in}[n] + Bu_{out}[n-1], \qquad B = 1 - A \tag{5.7-5}$$

Az 5.7-5 összefüggés szemléletesen mutatja, hogy a diszkrét idejű (mintavételes) elsőfokú RC aluláteresztő szűrő aktuális kimenetének értéke nem más, mint az aktuális bemenet és az előző mintavételi ciklusban kiszámított kimenet értékének súlyozott összege.

5.8. Feladatok

5.8.1. A bemutatott részegységek továbbfejlesztése

A funkcióbővítő rendszer komponenseinek a 2., 3. és 4. fejezetekben tárgyalt RTL-modelljei az adott funkciónak egy-egy viszonylag egyszerű megvalósítását mutatják be. Ez lehetőséget ad az olvasónak arra, hogy az RTL-tervezés gyakorlatát ezeknek az alapmodelleknek a mély megismerése után azok továbbfejlesztésével, új funkciók hozzáadásával sajátítsa el. Az alábbi pontban néhány ilyen továbbfejlesztési ötletet és ezekkel kapcsolatos rendszertechnikai kérdéseket gyűjtöttünk össze.

Jogosultságkezelés a memória-arbiterben ◆ Fejlessze tovább a 2.6.2. pontban bemutatott memória-arbiter modult a következőképpen:

- Tegye lehetővé, hogy a felhasználó (a *CPU*-n futó program készítője) – az arbiter inicializálásának részeként – az egyes kliensekhez egy-egy összefüggő címtartományt rendelhessen hozzá. Az adott kliens csak ezen a tartományon belül jogosult írás- és olvasáskérések kezdeményezésére. Az egyes kliensekhez rendelt memóriatartományok átfedhetnek.
- Ha egy kliens a hozzárendelt címtartományon kívül kezdeményez memóriaműveletet, akkor az arbiter – azon felül, hogy nem teljesíti a kérést – küldjön megszakításkérést a *CPU* felé.

Milyen rendszertechnikai előnyökkel és hátrányokkal jár az, hogy az egyes kliensekhez rendelt címtartományok átfedhetnek? Milyen regisztereket érdemes beépíteni a modulba annak érdekében, hogy megszakításkérés esetén minél több, a megszakításkiszolgáló rutinban feldolgozható információ álljon rendelkezésre?

Választható szűrőfokozat a DSP-bővítőmodulban ◆ Fejlessze tovább a 3.2. pontban bemutatott DSP-modult a következőképpen:

- A FIR-szűrő mellé építse be a 4.5.7. pontban bemutatott RC aluláteresztő szűrőfokozatot is a bővítőmodulba. Az RC szűrő együtthatóinak – a FIR-szűrő együtthatóihoz hasonlóan – futásidőben változtathatóknak kell lenniük.
- Tegye lehetővé a toplevel-integrátor számára, hogy a bővítőmodul példányosításakor egy szintézis-paraméter beállításával választhasson, hogy az A/D-átalakító interfészadapterétől érkező mintafolyam melyik szűrőfokozat bemenetére jusson el.

Futásidőben változtatható elemszámú FFT ◆ Fejlessze tovább a 4.5.5. pontban bemutatott FFT IP-magot úgy, hogy az *N* paraméter értéke – vagyis a transzformálandó mintasort tároló vektor és a Fourier-együtthatókat tartalmazó kimeneti vektor elemszáma – ne szintézis-, hanem futásidejű paraméter legyen.

- A 4.5.5. pontban tárgyalt megoldásban az *N* paraméter a belső memóriák (RAM és cosinus-tábla) méretét is meghatározza, így a modul szintézisekor csak annyi blokk-RAM

modult használunk fel, ami feltétlenül szükséges. Elérhető ez úgy is, ha az *N* paraméter futásidőben változik?

Állapotjelzők finomhangolása az SPI-FIFO adapterben ◆ A 2.8.3. pontban bemutatott SPI-FIFO útválasztó két, forgalomszabályozás megvalósítását segítő egybites kimenetet biztosít az üzemeltető rendszer számára; a *miso_fifo_empty* és a *mosi_fifo_full* jelzések rendre az SPI interfészadapter-áramkör kimeneti FIFO-jának kiürülését és bemeneti FIFO-jának telítettségét jelzik. Fejlessze tovább a modult oly módon, hogy e két állapotjelző ne csak a teljes telítettség és a teljes kiürülés tényét legyen képes jelezni, hanem az adott FIFO „csaknem kiürült" és „csaknem tele" állapotát is:

- A „csaknem tele" és „csaknem üres" állapotjelzők magas szintje jelentse azt, hogy az adott FIFO-ba még *N* darab bejegyzés fér. *N* értéke legyen futásidőben változtatható, a két FIFO-ra egyedileg meghatározható paraméter.
- Az üzemeltetői interfész vezetékeinek számán ne változtasson. A két állapotjelző kimenet viselkedése legyen futásidőben változtatható paraméter („tele" vs. „csaknem tele" és „üres" vs. „csaknem üres").

MMIO-tranzakciókat naplózó debugger logika ◆ A 2.9. pontban tárgyaltak mintájára készítsen hibakeresést segítő logikát a funkcióbővítő rendszer MMIO-tranzakcióinak nyomon követésére:

- A buszrendszer jeleinek megfigyelését egy triggerjel felfutása pillanatában kezdje meg és folytassa *T* időtartamig. *T* legyen szintézis-paraméter.
- A *T* időtartam alatt észlelt MMIO-tranzakciókról készítsen egy-egy „naplóbejegyzést", amelybe belekódolja a tranzakció típusát (írás vagy olvasás), a megszólított címet, valamint – a tranzakció típusától függően – az elküldött vagy visszaolvasott adatot.
- Minden naplóbejegyzéshez rendeljen hozzá egy *dT* mezőt, amelyből kiszámítható, hogy az adott tranzakcióban a *CPU* által elküldött kérés és a megszólított periféria-áramkör válasza között mennyi idő telt el. Nem szükséges nagy pontosságra törekedni, az eltelt időt mérheti a debugger logika saját órajel-periódusainak számlálásával.
- Ha egy MMIO-tranzakció időtartama elér egy *H* határértéket, akkor a megfigyelést szakítsa meg. *H* legyen szintézis-paraméter.
- A naplózott információkat egy szintézis-paraméterrel beállítható mélységű FIFO-ba töltse. A FIFO telítődésekor annak tartalmát az UART-kapcsolaton keresztül küldje el.

5.8.2. Új bővítőmodulok tervezése

Természetesen nemcsak meglévő részegységek módosításával, hanem teljesen új modellek létrehozásával is szerezhetünk tapasztalatot. Az alábbi pontban néhány ötlettel igyekszünk hozzájárulni az olvasó képzeletének beindításához.

MMIO CRC modul ◆ Tervezzen MMIO-kompatibilis CRC-modult, amely a 4.5.4. pontban bemutatott, szinkron kézfogásos interfésszel ellátott *crc_chunk_feed_wrp* burkolómodul szolgáltatásit képezi le MMIO-címtérképre.

Fixpontos aritmetikai bővítőmodul ◆ Tervezzen aritmetikai bővítőmodult, amely – a 4.5.3. pontban bemutatott lebegőpontos aritmetikai egységhez hasonlóan – összeadás-, kivonás- és szorzásműveleteket képes elvégezni fixpontos operandusokon. A megvalósítás során felhasználhatja a 4.5.1. pontban tárgyalt fixpontos aritmetikai modult és a 4.5.2. pontban bemutatott iteratív fixpontos szorzót.

- Az operandusok mérete legyen szintézis-paraméter. A legnagyobb megengedett operandusméret legyen 64 bit.
- Tegye lehetővé, hogy a felhasználó a bővítőmodul szolgáltatásainak igénybevétele után tájékozódhasson az elvégzett művelet során bekövetkezett túlcsordulásról.
- Túlcsordulás bekövetkezésekor a bővítőmodul küldjön megszakításkérést a *CPU* felé. A megszakításkérés küldését tegye futásidőben letilthatóvá.
- Tegye lehetővé a toplevel-integrátor számára, hogy a bővítőmodul példányosításakor egy szintézis-paraméter beállításával választhasson a számítási teljesítményre optimalizált (blokk-szorzó modulokat használó) architektúra és az erőforrás-felhasználásra optimalizált (blokk-szorzókat nem igénylő), iteratív szorzó almodult tartalmazó architektúra között.

SPI-UART protokoll-konverter ◆ Tervezzen bővítőmodult, amely az üzemeltetői SPI-interfész szolgáltatásait egy nagysebességű UART-interfészre képezi le. A megvalósítás során felhasználhatja a 4.3.1 pontban bemutatott UART-adót és a 4.3.4. pontban tárgyalt nagysebességű UART-vevő modult. A bővítőmodul UART-interfészén valósítson meg szabványos, szintézis-paraméter segítségével letiltható hardver-átvitelvezérlést.

Adatblokk-alapú memóriahozzáférés-vezérlő ◆ Tervezzen bővítőmodult, amely nagyméretű adatblokkok közvetlen átvitelét teszi lehetővé az üzemeltetői SPI-interfész és a külső operatív tár között. Az adatátvitel *N*-bájtos blokkok átvitelének, ún. tranzakcióknak a sorozataként valósuljon meg.

- Az egyetlen tranzakció során átvitt bájtok száma, a tranzakció iránya (SPI → operatív tár vagy operatív tár → SPI) és az átvitt adatblokk operatív tárbeli kezdőcíme legyen futásidőben változtatható paraméter.
- Az *N*-bájtos átvitel időtartama alatt a bővítőmodul nem blokkolhatja az MMIO-interfész működését. A tranzakció végét a bővítőmodul egy megszakításkérés küldésével jelezze.

PWM modul ◆ Tervezzen többcsatornás programozható frekvenciájú és kitöltési tényezőjű négyszögjel-generátort (impulzusszélesség-modulátor: **P**ulse **W**idth **M**odulation, PWM).

- Tegye lehetővé a toplevel-integrátor számára, hogy a kimeneti csatornák számát egy szintézis-paraméter segítségével állítsa be.
- Az egyes csatornák kitöltési tényezőre vonatkozó felbontása legyen 8 vagy 16 bit, a két lehetőség közül szintézis-paraméter(ek) segítségével lehessen választani.
- Az egyes csatornák frekvenciája és kitöltési tényezője egymástól függetlenül, futásidőben legyen módosítható.

- Az egyes csatornákon a négyszögjel-generálás legyen egymástól függetlenül letiltható, a kimenet letiltott állapotban felvett értéke pedig ugyancsak futásidőben konfigurálható.
- Minden csatornához hozzon létre két-két, futásidőben beállítható paramétert, amelyek az adott csatorna kitöltési tényezőjének alsó és felső korlátját definiálják. Ha a felhasználó az alsó korlátnál kisebb kitöltési tényezőt próbál beállítani, akkor a generált négyszögjel kitöltési tényezője vegye fel az alsó korlát értékét. Hasonló működést valósítson meg a felső korlát esetén is.

Veremtár ◆ Tervezzen verem szervezésű memóriamodult.

- A veremtár mérete legyen szintézis-paraméterrel beállítható.
- A modul által nyújtott szolgáltatások: *push* (adat beírása a verembe), *pop* (adat kiolvasása a veremből), *top* (legutóbb beírt adat visszaolvasása anélkül, hogy azt a veremből eltávolítanánk).
- Tegye lekérdezhetővé a verem telítettségére vonatkozó információt: tele *(full)* és üres *(empty)* állapotjelzők, valamint a szabad helyek száma.
- Ha a felhasználó a megtelt veremtárba próbál adatot írni, vagy az üres tárból próbál adatot olvasni, akkor a modul a hibát megszakításkéréssel jelezze a *CPU* felé.

Kimeneti FIFO ◆ Tervezzen bővítőmodult, amely a 4.2.5. pontban bemutatott *dualclock_fifo_hswrp* modul *push*-interfészének szolgáltatásait képezi le MMIO kérésekre.

- A modulban példányosított FIFO mérete legyen szintézis-paraméterrel beállítható.
- Tegye lekérdezhetővé a FIFO telítettségére vonatkozó információt: tele *(full)* és üres *(empty)* állapotjelzők, valamint a szabad helyek száma.
- A FIFO megtelését és kiürülését a modul egy-egy megszakításkéréssel jelezze a *CPU* felé.
- Ha a felhasználó a megtelt FIFO-ba próbál adatot írni, akkor a modul a hibát megszakításkéréssel jelezze a *CPU* felé.
- A FIFO *pop*-interfészét tegye a toplevel modul interfészének részévé.

Bemeneti FIFO ◆ Tervezzen bővítőmodult, amely a 4.2.5. pontban bemutatott *dualclock_fifo_hswrp* modul *pop*-interfészének szolgáltatásait képezi le MMIO kérésekre.

- A modulban példányosított FIFO mérete legyen szintézis-paraméterrel beállítható.
- Tegye lekérdezhetővé a FIFO telítettségére vonatkozó információt: tele *(full)* és üres *(empty)* állapotjelzők, valamint a szabad helyek száma.
- A FIFO megtelését és kiürülését a modul egy-egy megszakításkéréssel jelezze a *CPU* felé.
- Ha a felhasználó az üres FIFO-ból próbál adatot olvasni, akkor a modul a hibát megszakításkéréssel jelezze a *CPU* felé.
- A FIFO *push*-interfészét tegye a toplevel modul interfészének részévé.

Tömörítő ◆ Tervezzen FFT-alapú adattömörítő bővítőmodult, amely egy, a külső memóriában elérhető, változó méretű adatblokkon képes tömörítést és kicsomagolást végezni.

- A modul a forrásadatot a külső memóriából olvassa be, a feldolgozott (tömörített vagy kicsomagolt) adatot pedig ugyancsak a külső memóriába írja ki. A bemenő és kimenő adatblokkoknak a külső memóriabeli kezdőcímei legyenek futásidőben változtathatók.
- A modulban felhasználhatja a 4.5.5. pontban bemutatott FFT IP-magot.[222]

[222] A rendszertervezés során érdemes áttekinteni a 5.6. Függelékben foglaltakat, különös tekintettel az FFT-algoritmusnak az inverz Fourier-transzformált kiszámításában játszott szerepére.

Rövidítésjegyzék

BCD	Binary-Coded Decimal
CDC	Clock Domain Crossing
CPU	Central Processing Unit
CS(n)	Chip Select (SPI)
CRC	Cyclic Redundancy Check
CTS	Clear To Send
DDR	Dual Data Rate
DDS	Direct Digital Synthesis
DFT	Discrete Fourier Transform
DMEM	Data Memory
DRAM	Dynamic RAM
DSP	Digital Signal Processing
EEPROM	Electrically Erasable Read-Only Memory
EI	Enable Interrupt (állapotjelző)
FFT	Fast Fourier Transform
FIFO	First In First Out
FIR	Finite Impulse Response
GPIO	General-Purpose Input/Output
HDL	Hardware Description Language
HK	Housekeeping
IFA	Interface Adapter
I²C (IIC)	Inter-Integrated Circuit
IP	Instruction Pointer
IR	Instruction Register
ISR	Interrupt Service Routine
ksps	kiloSample(s) Per Sec
LED	Light Emitting Diode
LFSR	Linear Feedback Shift Register
LSB	Least Significant Bit/Byte
LSW	Least Significant Word

LVDS	Low Voltage Differential Signalling
MISO	Master Input Slave Output (SPI)
MISP	Minimal Instruction-Set Processor
MMIO	Memory-Mapped Input/Output
MOSI	Master Output Slave Input (SPI)
MSB	Most Significant Bit/Byte
MSBI	Most Significant Bit Index
MSW	Most Significant Word
MAC	Multiply & Accumulate
N	Negative (állapotjelző)
OSI	Open Systems Interconnection
PCB	Printed Circuit Board
PLC	Power Line Communication
PLL	Phase-Locked Loop
PWM	Pulse Width Modulation
RAM	Random Access Memory
RF	Register File
RTC	Real-Time Clock
RTS	Request To Send
SCL	Serial Clock (I^2C)
SCK/SCLK	Serial Clock (SPI)
SDA	Serial Data (I^2C)
SET	Single-Event Transient
SDRAM	Synchronous Dynamic Random Access Memory
SoC	System-on-Chip
SP	Stack Pointer
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
STA	Static Timing Analysis
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
WDT	Watchdog Timer
Z	Zero (állapotjelző)

L'Harmattan France
5-7 rue de l'Ecole Polytechnique
75005 Paris
T.: 33.1.40.46.79.20
Email: diffusion.harmattan@wanadoo.fr

L'Harmattan Italia SRL
Via Degli Artisti 15
10124 TORINO
Tél: (39) 011 817 13 88 / (39) 348 39 89 198
Email: harmattan.italia@agora.it

L'Harmattan Kiadó, Budapest
www.harmattan.hu

ISBN 978-963-646-444-8
ISBN 978-963-646-445-5 (pdf)
DOI: https://doi.org/10.56037/978-963-646-445-5

A kiadásért felel Gyenes Ádám, a kiadó vezetője.

A kiadó kötetei megrendelhetők,
illetve kedvezménnyel megvásárolhatók:
L'Harmattan Könyvesbolt
1053 Budapest, Kossuth L. u. 14–16.
Tel.: +36-70-554-3177

Olvasószerkesztő: Vigyázó Ágnes

Nyomdai előkészítés: Gembela Zsolt

Nyomdai kivitelezés: EPDB Nyomtatási Központ Zrt.
Felelős vezető: Gellér Ágoston József vezérigazgató