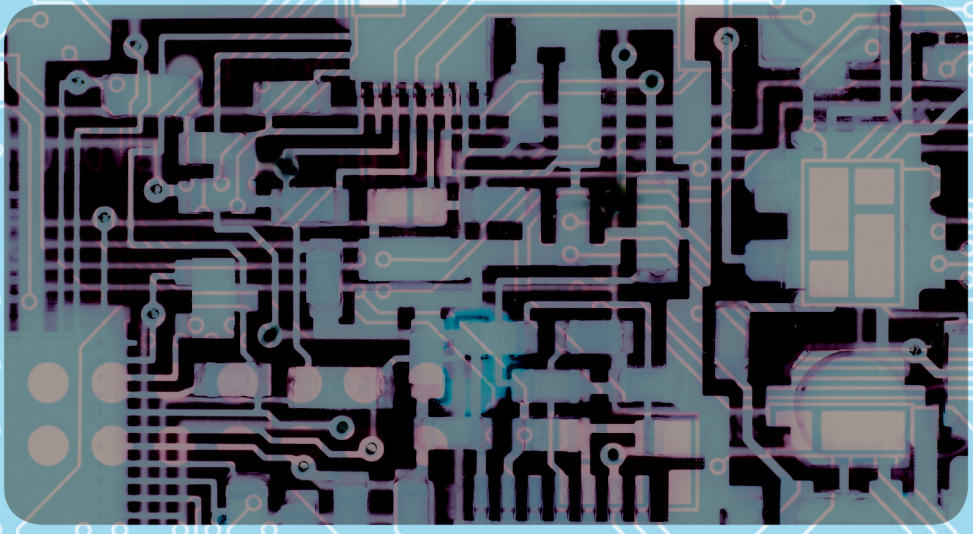
SAPIENTIA
ERDÉLYI MAGYAR
TUDOMÁNYEGYETEM
MAROSVÁSÁRHELYI KAR

jegyzet



BRASSAI SÁNDOR TIHAMÉR

ÚJRAKONFIGURÁLHATÓ DIGITÁLIS ÁRAMKÖRÖK TERVEZÉSI ÉS TESZTELÉSI MÓDSZEREI

Scientia Kiadó | 2018

BRASSAI SÁNDOR TIHAMÉR

ÚJRAKONFIGURÁLHATÓ DIGITÁLIS ÁRAMKÖRÖK TERVEZÉSI ÉS TESZTELÉSI MÓDSZEREI

SAPIENTIA ERDÉLYI MAGYAR TUDOMÁNYEGYETEM
MAROSVÁSÁRHELYI KAR
VILLAMOSMÉRNÖKI TANSZÉK

BRASSAI SÁNDOR TIHAMÉR

ÚJRAKONFIGURÁLHATÓ DIGITÁLIS ÁRAMKÖRÖK TERVEZÉSI ÉS TESZTELÉSI MÓDSZEREI

Scientia Kiadó
Kolozsvár·2018

Lektor:
Vásárhelyi József (Miskolc)

Sorozatborító
Tipotéka Kft.

Descrierea CIP a Bibliotecii Naţionale a României
BRASSAI, SÁNDOR-TIHAMÉR
Újrakonfigurálható digitális áramkörök tervezési és tesztelési módszerei /
Brassai Sándor Tihamér. – Cluj-Napoca : Scientia, 2018
Conţine bibliografie
ISBN 978-606-975-020-9

62

TARTALOMJEGYZÉK

CONTENTS

CUPRINS

ELŐSZÓ

A digitális fényképezőgépek, tévékészülékek, számítógépek, adathordozók, személyes digitális eszközök, fogyasztói termékek alapvető alkotóelemeit, amelyek digitális feldolgozást végeznek, a digitális elektronikus áramkörök képezik.

A mérnökjelöltek számára kiemelkedő fontosságú a digitális áramkörök tervezési és tesztelési módszereinek az elsajátítása és gyakorlatban való alkalmazása.

A jegyzet bemutatja a digitális áramkörök alapvető tervezési és tesztelési módszereit és a témához kapcsolódó alapvető fogalmakat.

Az áramkörtervezésben fontos szerepe van a hardverleíró nyelveknek (VHDL, Verilog). A jelenlegi tervezőeszközök a HDL alapú tervezés mellett mára már lehetővé teszik az áramkör működésének a specifikálását különböző magas szintű programozási nyelven is (C, C++, System C, Matlab).

Az első fejezet a digitális rendszerek tervezési módszereit, a digitális áramkörök különböző tartományokban történő leírását mutatja be. Részletezi a lépéseket, amelyeket követve, kiindulva a követelmények megfogalmazásából, elérünk egy digitális áramkör fizikai megvalósításáig. Továbbá bevezet a VHDL hardverleíró nyelv alapú tervezésbe. Rávilágít a VHDL-hez kapcsolódó alapvető szabványokra. Ismerteti a VHDL nyelv nyelvtani elemeit, sajátosságait, felépítését, könyvtárak használatát, a tervezett hardver és a VHDL kapcsolatát.

A második rész a VHDL-ben alkalmazható fontosabb szekvenciális kifejezéseket – folyamat, feltételes értékadás, ciklusok, jelek és változók – tekinti át. Az ismeretek áttekintését és a példák begyakorlását követően az olvasó képes lesz kombinációs és szekvenciális áramkörök VHDL-ben való modellezésére és szimulációjára.

A harmadik fejezetben az olvasó elsajátítja a konkurens VHDL utasítások alkalmazását. Összehasonlítva a szekvenciális utasításokkal, több áramkör működése egyaránt szekvenciális és konkurens utasítással is leírható.

Alapvető áramkörök, szekvenciális és konkurens utasításokkal való megvalósítására összehasonlító példákat tárgyal a fejezet. A példaprogramok tanulmányozása során elsajátítható a fontosabb utasítások működése és különböző gyakorlati alkalmazása.

A bonyolult áramkörök tervezésében fontos szerepet játszik az áramkörök paraméterezése, általánosítása, egy adott áramkör paraméterek szerinti kialakítása, részekre való felosztása, az egyes programrészek megosztása, újrafelhasználása. A generikus (GENERIC) paraméterek, valamint a FOR GENERATE és IF GENERATE utasítások alkalmazása lehetővé teszi a bonyolult áramkörök egyszerű módon való moduláris felépítését.

A következő fejezet a tervezett digitális áramkör különböző lehetséges tesztelési lehetőségeit mutatja be. A fejezet első fele a tervezett áramkör szimulációval történő ellenőrzését, a vizsgálathoz szükséges tesztelő áramkör létrehozását szemlélteti. A fejezet második felében az FPGA-n elkészített áramkör FPGA-ba integrált logikai analizátor alkalmazásával a rendszer működés közbeni tesztelését részletezi.

Az ötödik fejezet az adatutat is tartalmazó komplex állapotgép VHDL hardverleíró nyelven való megvalósítását tárgyalja. Felfrissíti az olvasó számára a Moore és Mealy véges állapotú állapotgépek felépítését és működési elvét, ugyanis az említett automaták képezik a komplex állapotgépek alapjait. A komplex állapotgép az egyik alapvető tervezési módszer, amelyet egyaránt alkalmaznak egyszerű és bonyolult műveletvégző áramköri tervezésben. A fejezet tárgyalja a komplex állapotgépek VHDL alapú megvalósítását és azt példával szemlélteti.

A hatodik fejezet egy komplex állapotgépnek Vivado környezetben való tervezését és szimulációját részletezi. Az utolsó két fejezetben magas szintű szintézisre (HLS) épülő áramkörtervezés van bemutatva. A hetedik fejezetben a System Generator alapú rendszertervezést és hardver-szoftver-koszimulációt részletezzük. A System Generator alapú környezet alkalmazása egyszerűsíti a hardvertervezést és -tesztelést.

Míg a VHDL alapú tervezés alapos hardveres ismereteket igényel, a HLS technikák és egyben a System Generator alapú környezet alkalmazásával minimális hardverismeretek mellett is könnyedén megvalósítható egy alkalmazás. A fejezetben az impulzusszélesség modulált jelgenerátor System Generator környezetben való tervezését és hardver-szoftver-koszimulációval történő működés közbeni tesztelését mutatjuk be.

Az utolsó téma keretében a Vivado HLS (High Level Synthesis) magas szintű szintetizáló eszközt alkalmazva valósítjuk meg néhány egyszerű áramkör: összeadó vagy szorzó áramkör tömbelemei összegének a kiszámítását. A fejezet bevezeti a HLS módszerhez kapcsolódó alapvető fogalmakat, mint interfészek, protokollok, portjelek, és a korlátokat, amelyek az áramkör tervezése során alkalmazhatók az interfészek, portok, memóriák, ciklusok specifikálására. A fejezet tárgyalja a tervezés során a tervezési korlátok meghatározására alkalmazható parancsokat, a modulok összekapcsolására szolgáló vezérlőjelek (interfész és portjelek) függvényargumentumokból való létrehozását. A mintafeladatok gyakorlását követően az olvasó elsajátítja a magas szintű szintézisen alapuló áramkörtervezés alapjait. A jegyzet egyszerű példák szemléltetésével nyújt betekintést a digitálisáramkör-tervezés és -tesztelés világába.

Marosvásárhely, 2018. november 14. Brassai Sándor Tihamér

1. fejezet

VHDL alapismeretek

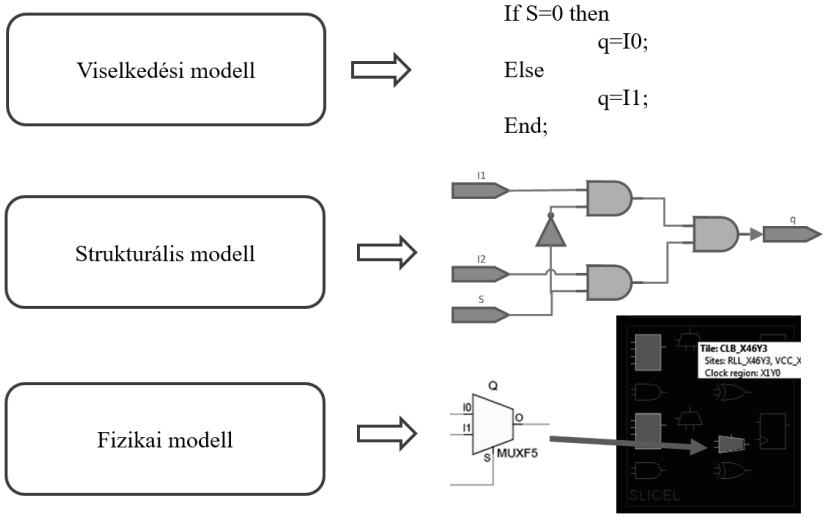
A fejezet bemutatja a VHDL (VHSIC – Very High Speed Integrated Circuits Hardware Description Language) nyelv nyelvtani elemeit, sajátosságait, felépítését, könyvtárak használatát, a tervezett hardver és a VHDL kapcsolatát.

1.1. Digitális rendszerek tervezése

A digitális rendszertervezés során, kiindulva a rendszerrel szembeni követelmények megfogalmazásából, több lépésen keresztül eljutunk a rendszer fizikai megvalósításáig. A technológia fejlődése egyre bonyolultabb integrált áramkörök megvalósítását teszi lehetővé. Az összetett tervek kezelésére két módszer alakult ki [1]:

- áramkörök leírása viselkedésük alapján,
- tervezési folyamat számítógép alapú automatizálása.

Egy áramkört tekinthetünk egy fekete doboznak, amelynek nem ismerjük a belső felépítését, szerkezetét, de ismerjük a viselkedését, a bemenetek és kimenetek közötti összefüggést. Egy másik megközelítés alapján ismerjük részletes felépítését, hogy modulárisan milyen alegységeket, modulokat, komponenseket tartalmaz, és a komponensek közötti kapcsolatokat. Egy áramkörnek a tervezése során meghatározhatjuk a viselkedését vagy felépíthetjük olyan áramköri elemekből, amelyek ugyanazt a viselkedést eredményezik (1.1. ábra).



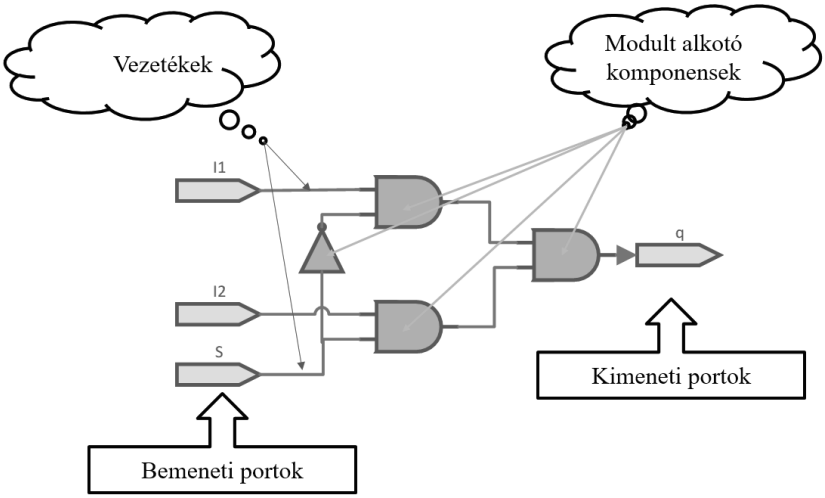
1.1. ábra. Tervezési modellek szemléltetése

A viselkedési modell a bemeneti és kimeneti adatok közötti összefüggések leírására irányul. A viselkedési modell úgy tekinti a rendszert, mint egy fekete dobozt, eltekint a belső felépítésétől, leírja a rendszer vagy alrendszer viselkedését. $Q = I_0\overline{S}_1\overline{S}_0 + I_1\overline{S}_1S_0 + I_2S_1\overline{S}_0 + I_3S_1S_01$.

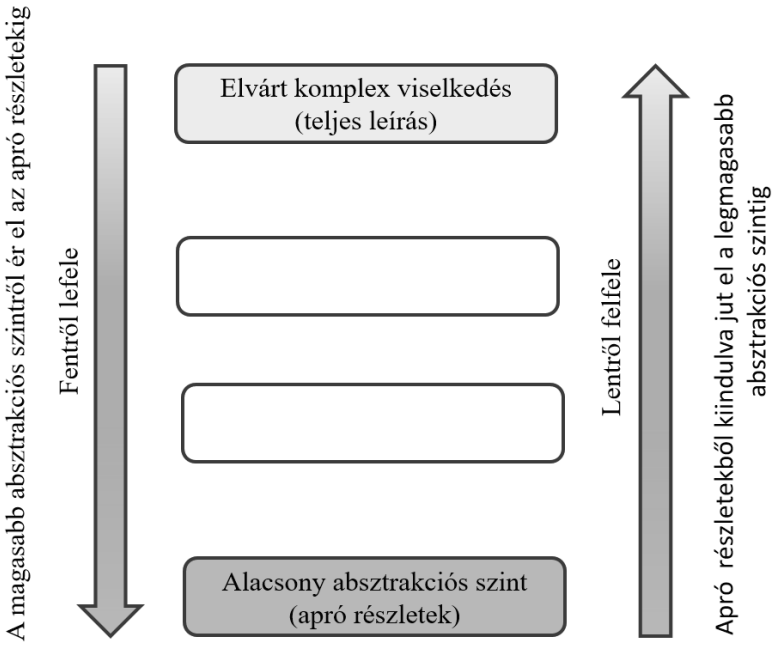
A strukturális modell leírja a rendszer felépítését, meghatározza, milyen áramköri elemeket tartalmaz, valamint az áramköri elemek hogyan vannak egymással összekapcsolva (1.2. ábra). A modulok belső felépítését hierarchikus szinteken keresztül adja meg. A példa szerint egy multiplexer áramkört logikai kapukból építünk fel. A modulok port jeleken keresztül kommunikálnak a rendszerben megtalálható modullal.

A digitális rendszerek tervezésére két tervezési módszert különböztetünk meg (1.3. ábra):

– fentről lefele (top down): egy magasabb absztrakciós szintről apró lépésekkel ér el a részletekig,
– lentről felfele: az apró részletekből kiindulva jut el a magasabb absztrakciós szintig.

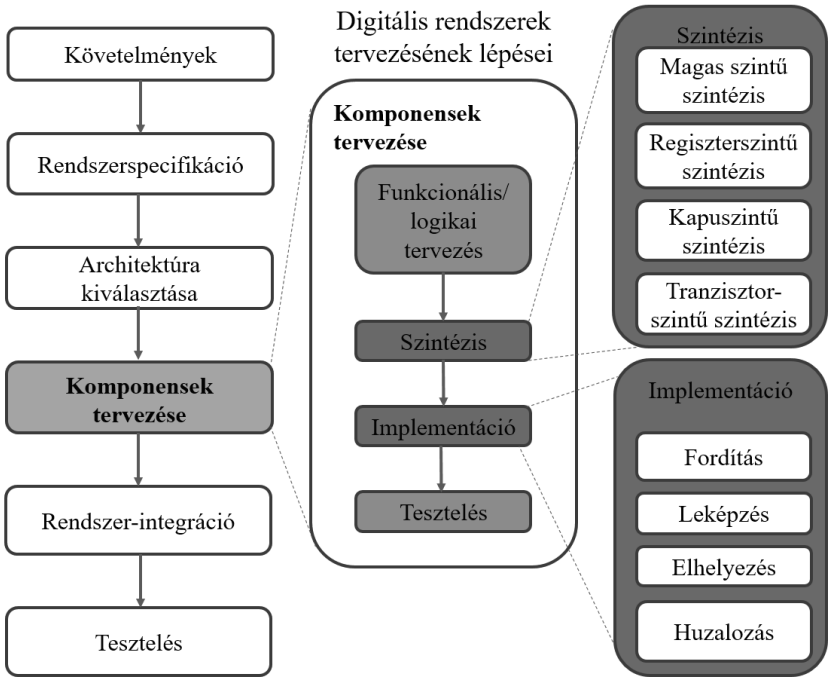


1.2. ábra. Strukturális tervezési modell



1.3. ábra. Tervezési módszerek: fentről le és lentről fel

1.1.1. Digitális rendszerek tervezésének lépései



1.4. ábra. Digitális rendszerek tervezési lépései

A digitális rendszerek tervezési lépései, melyeket az 1.4. ábra szemléltet, a következők:

1. Rendszerkövetelmények. A tervezési folyamat első lépése a rendszerrel szembeni követelmények: működési sebesség, késleltetési idők, csatlakozási pontok, disszipált teljesítmény meghatározása.

2. Rendszer-specifikáció. A specifikáció összefoglalja a rendszert leíró szabályokat, az áramkör viselkedését leíró algoritmust. A rendszer-specifikáció a követelmények alapján készül el. A specifikációban apró részletességgel megadjuk, hogyan fog működni a tervezett rendszer:
   - különböző szinteken milyen műveleteket végez el az áramkör,
   - hogyan épül fel az áramkör,

   - meghatározzuk a belső részegységeket, modulokat,
   - meghatározzuk, milyen formában állnak rendelkezésre a bemeneti adatok, és milyen formában várjuk az eredményt.

3. A következő lépés az architektúra kiválasztása. A tervezésnek ebben a fázisában meg kell határozni, hogy milyen architektúrát követ a rendszer: processzor alapú megvalósítás CISC vagy RISC architektúrával, csővezetékes struktúra kialakítása, hogyan történik az adatok tárolása.

4. Modulok, alegységek tervezése. A tervben meghatározott alegységek, modulok fizikai megvalósítása több lépésben történik: az egységek, alegységek funkcionális és logikai tervezése, az áramköri szintézis, a fizikai megvalósítás, majd utolsó lépésben az áramkör tesztelése.
   - A funkcionális tervezés során azonosítjuk a tervben megtalálható funkcionális egységeket, amelyek alkalmazhatóak egy adott célfeladat megoldására. Ezt a folyamatot tömbszintű (block level) tervezésnek is szokás nevezni. A tömbök az adatok tárolására szolgáló regiszterekből és aritmetikai és logikai műveletvégző egységekből épülnek fel. A modulok működésének leírása regiszterátviteli szinten történik, és a megoldásban regiszterek, memóriák, aritmetikai egységek, állapotgépek vesznek részt.
   - A logikai tervezés során megvalósítjuk a regiszterátviteli szinten leírt elemeket. A logikai tervezést kapuszintű tervezésnek is nevezik, mivel ezen a szinten logikai kapukat, multiplexer áramköröket, bistabil áramköröket, vagyis úgymond kapuszintű elemeket alkalmaznak a megvalósításra. A kapuszinten alkalmazott elemek funkcionalitása teljesen meghatározott. A logikatervezés során a kapuszintű elemeket megpróbáljuk összekapcsolni, úgy, hogy egy adott funkcionális tömböt valósítson meg.
   - A logikai tervezést követi az áramkörtervezés, majd a fizikai megvalósítással (implementálással) jön létre a digitális áramkör.

5. A logikai szintézis során egy magasabb absztrakciós szinten általában hardverleíró nyelven specifikált feladat, egy alacsonyabb absztrakciós szinten, általában kapuszinten valósul meg. A feladat leírása általában regiszterátviteli szinten (RTL) történik. A szintézis folyamatának az elvégzésére egy szintetizáló eszköz van alkalmazva.

6. A fizikai megvalósítással (implementálással) jön létre a digitális áramkör.

7. Rendszerintegráció. Az egyes alegységek tervezését, majd tesztelését követően a modulokat integrálják a rendszerbe.

8. A tervezési fázis utolsó lépéseként elvégzik a rendszer egységes tesztelését.

Egy digitális hardver különböző absztrakciós szinten írható le. A tervezendő hardverrendszer specifikációjából kiindulva, általában egy magasabb hierarchikus szintről alacsonyabb hierarchikus szintekre lépegetve jutunk az áramkör hardverszintű megvalósításáig.

A hardver kialakítható különböző specifikációs modellekből kiindulva, ismerve a hierarchikus szinteket és megértve az áramkör-leírási modellek (strukturális, viselkedési, fizikai) közötti összefüggéseket.

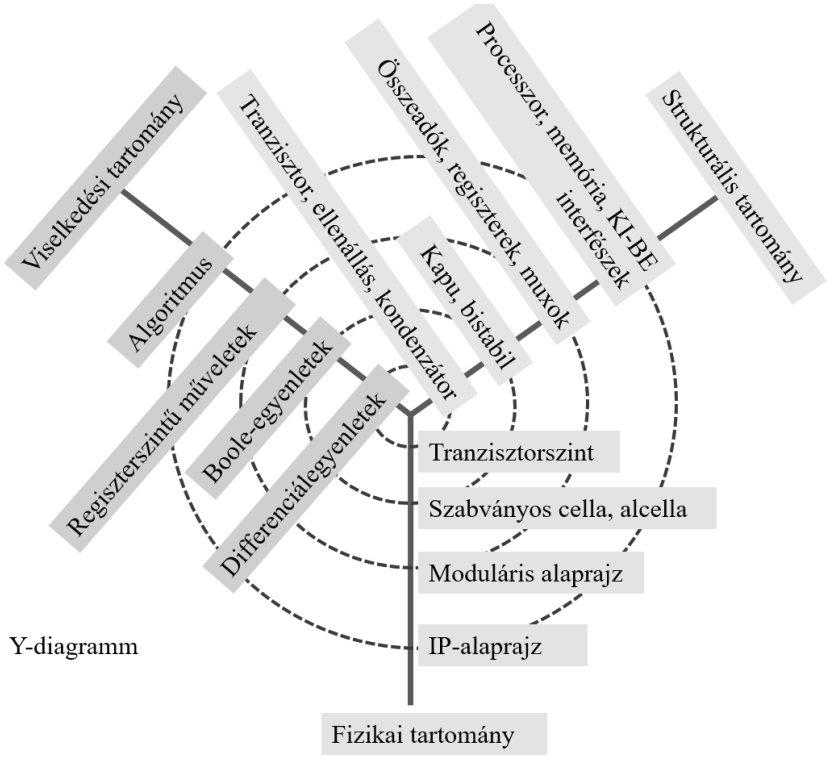
A digitális rendszerek különböző absztrakciós szinteken és különböző tervezési modellek szerint írhatók le. A digitális rendszerek leírására alkalmazott absztrakciós szintek:

- tranzisztorszint,
- kapuszint,
- regiszterszint,
- processzorszint.

Minden egyes absztrakciós szinten az áramkörök három különböző tartományban írhatók le:

- viselkedési,
- strukturális,
- fizikai.

A hardver absztrakciós szintek és a tartományok közötti összefüggéseket az Y diagram szemlélteti (1.5. ábra). Egy adott áramkör minden egyes absztrakciós szinten értelmezhető mindhárom tartomány szerint. Az Y diagramon az ágakon kívülről befele történő leírást finomításnak, a bentről kifele való leírást absztrakciónak nevezzük [1].



1.5. ábra. Y diagram

Absztrakciós szintek:

– Tranzisztor absztrakciós szinten alkalmazott alap- építőelemek: tranzisztorok, ellenállások, kondenzátorok, tekercsek. Ezen a szinten a digitális áramköröket analóg áramkörként tekintik. A digitális rendszerek tervezésekor nem foglalkoznak ezzel a szinttel, mivel ez az áramkörgyártó feladata. A viselkedési tartományban a bemenet és kimenet közötti összefüggéseket differenciálegyenletekkel vagy áramfeszültség diagramokkal írják le.
– Kapu absztrakciós szinten az alap építőelemek egyszerű logikai kapuk, alapmemória elemek, mint például a bistabil áramkörök, multiplexerek. Viselkedési tartományban a bemeneti-kimeneti kapcsolatot Boole-egyenletekkel határozzák meg. A jeleket ’1’ vagy ’0’ logikai szinteknek tekintik. Strukturális tartományban logikai kapuk vannak alkalmazva az áramkör megvalósítására. Az időzítéssel

kapcsolatos információk is egyszerűsítve vannak a kapuk terjedési késleltetésére korlátozva (propagation delay).

– A regiszterátviteli szint (RTL) egy absztrakciós tervezési szint egy rendszer digitális részének meghatározásához. Regiszterátviteli szinten az alkotóelemek kapukból épített modulok, összeadók, komparátorok, tárak, regiszterek. A regiszterátviteli szint a kapuszintű absztrakcióra épül. Ezen a szinten a több bites jelek sínekbe vannak csoportosítva. Ezen az absztrakciós szinten az áramkör leírására véges állapot automatát alkalmaznak. Viselkedési nézetben a jelek speciális adatként vannak értelmezve: előjeles vagy előjel nélküli egész, lebegőpontos. Strukturális tartományban például kettes komplemensként értelmezett bitsorozat, míg fizikai tartományban több bites vezetékként értelmezhető.
  Az RTL absztrakció az alap elvonatkoztatási szintnek számít az elektronikus rendszerek meghatározására, és gyakran arany modellként tekinthető a digitális rendszerek tervezési és ellenőrzési folyamatában.
  A regiszterátviteli szint szinkron logikán alapul, és három elsődleges részt tartalmaz:
  – regisztereket, amelyek az állapotok tárolására szolgálnak,
  – a kombinációs logikát a következő állapot meghatározására az aktuális állapot és bemeneti jelek függvényében,
  – valamint az órajelet, amely szinkronizálja az állapotok közötti váltást.
– Processzorabsztrakciós szinten alkalmazott strukturális építőelemek a processzorok, a memóriák, a sínrendszerek, az aritmetikai és logikai műveletvégző egységek. Viselkedési tartományban a leírás a program a számolás lépéseivel és a kommunikációs folyamatokkal. A fizikai tartományban a processzorabsztrakciós szintet IP-alaprajzként értelmezik, amely tartalmazza a makrocellákat a köztük levő összeköttetésekkel. A processzorszintű absztrakció az RTL szintű absztrakcióra épül.

A Gajski-Kuhn Y diagram a tervezők számára összefoglalja és átláthatóvá teszi a különböző nézetekben, különböző absztrakciós szinteken specifikált tervet vagy tervnek a részét.

Annak függvényében, hogy a feladat mely absztrakciós szintű elemekre épül, a szintézis lehet:

– Magas szintű szintézis: egy algoritmust RT szintű leírásra alakít át. Magas szintű szintézissel magas szintű programozási nyelven C,

C++, SystemC specifikált modell alapján is szintetizálható egy áramkör.

– Regiszterszintű szintézis. RT szintű viselkedési leírás analízise alapján strukturálisan felépíti az áramkört RT szintű elemeket alkalmazva. A regiszterszintű szintézis az RTL szinten megtalálható funkcionális egységek alkalmazásával alakítja ki az áramkört. A különböző alegységek vezérlésére egy állapotautomatát alkalmaz.
– A kapuszintű szintézis kapuszintű elemeket alkalmaz az áramkör kialakítására, a strukturális megvalósítás kapuszintű elemekre épül. Általában egy többszintes optimalizálási eljárást alkalmaznak az eredő áramkör méretének minimalizálására.

A szintetizáló program a HDL leírást átalakítja kapuszintű netlist állománnyá, az átalakítás során alkalmazva technológiai könyvtárakat. A szintetizáló program futásának eredménye a gyártó függvényében a hardver leíró nyelven specifikált terv valamilyen EDIF (Electronic Design Interchange Format) formátumra való fordítása. Az EDIF formátumot az automatizált elektronikus tervezésben használják különböző formátumok közötti adatcserére. A szintézis során két- vagy többszintes optimalizálási eljárást alkalmaznak az eredő áramkör méretének minimalizálására.

Technológiai leképzés (Technology mapping). A célrendszer előre elkészített kapuszintű primitív elemeket tartalmaz, ami lehet egy logikai cella egy standard célkönyvtárból vagy egy primitív cella például egy FPGA rendszerből. Ahhoz, hogy a kapuszintű áramkört fizikailag megvalósítsuk egy partikuláris célrendszeren, a kapuszintű áramköri elemeket le kell képezni a kiválasztott célrendszerben megtalálható áramköri elemekre. Ezt a folyamatot nevezik technológiai leképzésnek (technology mapping).

A fizikai tervezés során több finomítási lépést alkalmazva a szintetizált áramkörből létrejön a fizikai áramkör.

A fizikai megvalósítás lépései:

– A szintetizált áramkör fordítása: akár több különböző hardverleíró nyelven elkészített tervezői állomány egyesítésével létrehoz egyetlen netlist (EDIF) állományt. A fordítási fázis során a fordítóeszköz kombinálja az összes bemeneti NGC (Native Generic Circuit) netlistállományokat és korlátokat tartalmazó állományokat (User Constraints File) és létrehoz egy logikai tervállományt (NGD Native Generic Database) egy fordítóprogram alkalmazásával. A netlistállomány egy listát tartalmaz, felsorolja az áramkörben található komponenseket, az áramköri elemek összekapcsolására alkalmazott vezetékeket és csomópontokat, valamint attribútumokat, a leíró nyelv

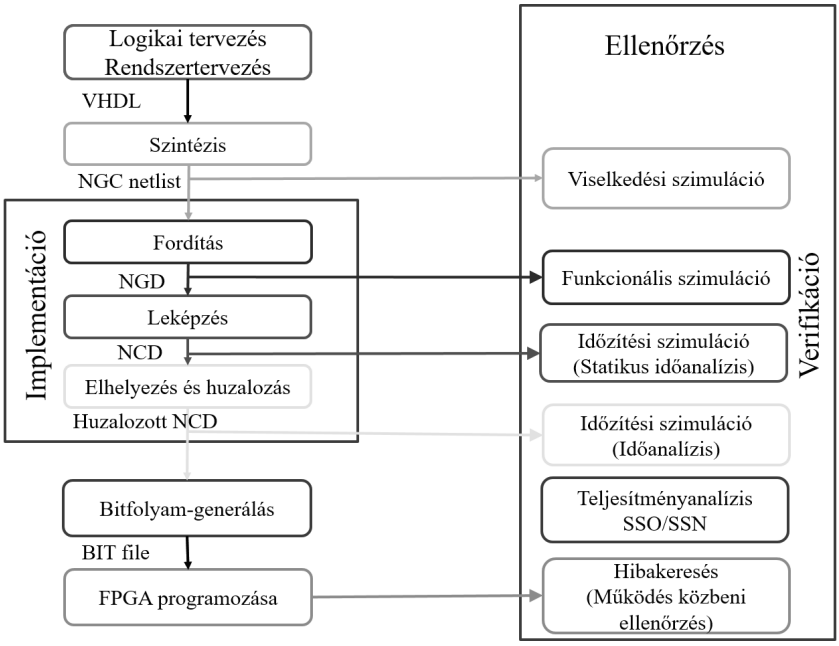
és annak sajátosságai függvényében az alkatrészhez hozzárendelt tulajdonságokat.

- Leképezés során a logikai elemeket tartalmazó áramkört (NGD) leképezi a céláramkörben elérhető alegységekre, úgy, hogy ezek elhelyezhetők legyenek. Például egy FPGA alapú célrendszerre az NGD állományban leírt logikát leképezi az FPGA áramkör megfelelő elemeire (konfigurálható logikai tömbök, keresőtáblázatok, bistabil áramkörök, BlockRAM memóriák) és generál egy NCD (Native Circuit Description) állományt, amely fizikailag leírja az FPGA áramkör elemeit. A leképzést is egy célprogram végzi el. Ebben a fázisban tehát pontosan ismertek az áramkört alkotó elemek, viszont még nem ismert, hogy a sok áramkör közül melyiket alkalmazzuk a végső kialakításban. Fizikailag a céláramkörben több ÉS kapu található, a fordítási fázis eredményeként egy ÉS kaput alkalmazunk, de még nincs meghatározva, hogy pontosan melyiket.
- Elhelyezés során az előző fázis eredményeként kapott alegységeket elhelyezi (rögzíti) a céláramkörben, figyelembe véve a különböző tervezési korlátokat.
- A huzalozás lépésben összeköti a tervben található alegységeket. Az elhelyezésre és huzalozásra rendelkezésre áll egy célprogram. Az elhelyezési és huzalozási folyamat figyelembe veszi (feloldja) a korlátok közötti ellentmondásokat. Sikertelen huzalozás esetén vagy a terv optimalizálása céljából a célprogram ciklikusan újra megpróbálja elhelyezni és huzalozni az alkatrészeket. A huzalozás eredménye egy huzalozott NCD állomány.

A huzalozás eredményeként kapott NCD állomány különbözőképpen van alkalmazva a céláramkör megvalósítására. FPGA áramkörök alkalmazásakor a szintetizáló eszköz, utolsó lépésben, létrehozza a huzalozott NCD állományból a konfigurációs állományt. ASIC áramkörgyártás esetében az NCD állomány alapján az alkalmazott technológia függvényében elkészülnek az áramkör gyártásához szükséges maszkok. Az 1.6. ábra a tervezés lépéseit, valamint a tervezési folyamattal párhuzamosan a fontosabb ellenőrzési lehetőségeket szemlélteti. Az 1.6. ábrán egyes tervezési fázisok FPGA alapú megvalósításának megfelelően fel van tüntetve a létrejött állományok típusa, amelyben el van tárolva az adott tervezési fázis eredménye.

A tervezés különböző fázisaiban alkalmazható ellenőrzési lehetőségek:

- Viselkedési szimuláció: a szintézis eredményeként létrehozott áramkör kimondottan csak viselkedésszintű ellenőrzésére alkalmazzák. Az



1.6. ábra. A tervezés és ellenőrzés lépései

ellenőrzés ezen fázisa nem tartalmazza az időzítési korlátok ellenőrzését.

– Funkcionális szimuláció: az áramkör ellenőrzése a fordítási fázis után. A funkcionális szimuláció egyszerűen teszteli az áramkör „funkcionális" működésének logikáját. Nem veszi figyelembe a belső logikából, valamint a huzalozásból származó késleltetéseket.
– Statikus időanalízis. A leképzés fázist követően ismertek a célrendszer megvalósítására alkalmazott áramkörök, tehát kiszámíthatók az áramkörökön fellépő késleltetések. Az ebben a fázisban elvégzett ellenőrzést statikus időanalízisnek nevezzük.
– Időanalízis. A huzalozási fázist követően a fizikai megvalósítással kapcsolatos részletek ismertek. Ebben a fázisban ismertek egyaránt az áramkörökön fellépő késések, a huzalozásból a vezetékeken fellépő, valamint a huzalozásból származó beiktatott áramkörök késleltetései is. Az időanalízis alapján ellenőrizhető, hogy a tervezés során sikerült-e teljesíteni az áramkörrel szemben megfogalmazott időkorlátokat, működési sebességeket.

– Az elhelyezési és huzalozási fázist követően elvégezhető a rendszer disszipációs analízise, ellenőrizhető, hogy teljesülnek-e a teljes áramkörre, almodul szinten vagy kimeneti portjel szinten az áramkörrel szemben megfogalmazott áramkorlátok és teljesítménydisszipáció.
– Működés közbeni ellenőrzésre több lehetőség van: külső méréstechnikai eszközök alkalmazása (oszcilloszkóp, logikai analizátor) vagy az FPGA áramkörbe integrált logikai analizátor. A XILINX típusú FPGA-án megvalósított hardver működés közbeni tesztelésére alkalmazható a hardver-koszimulációs módszer. A hardver-koszimulációval működés közben az FPGA áramkörből a jelek megjeleníthetők a számítógép képernyőjén. A hardveres koszimuláció konkrét alkalmazását a 7. fejezetben részletezzük.

1.2. HDL alapú tervezés

A digitális rendszerek tervezésére számos módszer alkalmazható, mint a hardverleíró nyelv alapú, kapcsolási raj alapú, modellalapú tervezés (Matlab Simulink) vagy akár magas szintű programozási nyelvek (C, C++, SystemC). A hardverleíró nyelvek kiemelkedő szerepet kapnak a hardvertervezés során.

A hardverleíró nyelv (HDL) egy speciális számítógépes programozási nyelv, amelyet elektronikus áramkörök, a leggyakrabban digitális áramkörök struktúrájának és viselkedésének a leírására alkalmaznak.

Egy hardverleíró nyelv lehetővé teszi a pontos, formális leírását egy elektronikus áramkörnek, és egy megfelelő eszköztár alkalmazásával a HDL leírás alapján elvégezhető az elektronikus áramkör automatizált elemzése, szimulációja és fizikai megvalósítása.

A VHDL alkalmas nagy komplexitású VHSIC elektronikus rendszerek működésének specifikálására. Különösen alkalmas FPGA és ASIC alkalmazás specifikus integrált áramkörökkel megvalósított digitális elektronikus tervek strukturális is viselkedési leírására.

A VHDL rövidítés a **V**HSIC-**HDL** rövidítésekből származik, vagyis nagyon nagy sebességű integrált áramkörök hardverleíró nyelve (**V**ery **H**igh **S**peed **I**ntegrated **C**ircuit **H**ardware **D**escription **L**anguage) [2], [3].

A VHDL egy szabványba foglalt modellező nyelv, amelyet pontosan és teljesen meghatároz a Nyelvi Referencia Kézikönyv (Language Reference Manual – LRM).

A VHDL nyelv tulajdonságai:
- újrahasználható;
- technológiafüggetlen:
  - nem kötődik egyetlen szimulátorhoz vagy adatbázishoz sem,
  - nem erőltet rá a tervezőre egyetlen tervezési módszertant sem,
  - független a berendezési és IC-gyártási eljárástól;
- könnyen használható tervek módosítására vagy kiegészítésére.

A VHDL hatékony alkalmazásához szükség van egy tervezési módszertanra és egy eszköztárra. A szimulációs és szintetizáló eszközök a két legfontosabb eszköz, amelyek a VHDL hardverleíró nyelv alapján működnek.

A Nyelvi Referencia kézikönyv nem határoz meg egy szimulátort, viszont meghatározza minden egyes szimulátornak, hogy mit tegyen a nyelvi leírás minden egyes részletével.

A VHDL nem korlátozza a felhasználót egyetlen leírási módra sem, lehetővé teszi a tervek fentről le, lentről fel vagy vegyes leírását bármelyik absztrakciós szinten. VHDL-ben egy terv specifikálható kapu, regiszterátviteli vagy akár processzorabsztrakciós szinten.

Sikeres magas szintű tervezés során szükség van egy hardverleíró nyelvre, a megfelelő tervezési eszközökre (szimulátor, szintetizáló), valamint a megfelelő tervezési módszer alkalmazására.

A tervező dönti el az alkalmazott hardverleíró nyelvet, a szintézisre és szimulációra alkalmazott eszköztárat, valamint a tervezési módszert. Hardverleíró nyelvként alkalmazható a VHDL vagy egyéb hardverleíró nyelvek.

Az eszköztárat általában az FPGA áramkört gyártó cég szolgáltatja. Összefoglalva: a VHDL nyelv leírja egy elektronikus áramkör vagy rendszer viselkedését, struktúráját, amely alapján elemezhető vagy megvalósítható a fizikai áramkör. A VHDL mint szabványosított hardverleíró nyelv a következő esetekben alkalmazható:
- szintézisre, áramköri leírásra,
- FPGA vagy CPLD áramköri tervezésre,
- áramköri szimulációra.

1.2.1. VHDL szabványok

A VHDL hardverleíró nyelvet az amerikai Védelmi Minisztérium (DoD – Department of Defence) megbízásából fejlesztették ki. A fejlesztés során sikerült egy programozási nyelvet tervezni és megvalósítani, amely teljes

mértékben megfelel a tervezés szempontjainak, sőt messzemenően túlteljesíti azokat. A hardverleíró nyelvek akár bonyolult rendszerek modellezésére alkalmazhatók. Egyik legfontosabb erősségük a párhuzamos folyamatok kezelése.

A VHDL hardverleíró nyelvet a Villamosmérnökök Nemzetközi Szervezete, az IEEE (Institute of Electrical and Electronics Engineers) szabványosította 1987-ben [3].

A VHDL hardverleíró nyelvhez kapcsolódó fontosabb VHDL szabványok, szabványmódosítások, továbbfejlesztések [4], [5], [6], [7]:

- IEEE Std. 1076-1987, *IEEE 1076-1987* – IEEE Standard VHDL Language Reference Manual, első változata a szabványnak
- IEEE Std. 1076-1993, *IEEE 1076-1993* – IEEE Standard VHDL Language Reference Manual
- IEEE Std. 1164-1993, *IEEE 1164-1993* – IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164)
- IEEE Std. 1076.6-1999, *IEEE 1076.6-1999* – IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis
- IEEE Std. 1076-2000, *IEEE 1076-2000* – IEEE Standard VHDL Language Reference Manual (IEEE Std 1076, 2000 Edition – tartalmazza az IEEE Std 1076-1993 és IEEE Std 1076a-2000 szabványokat), VHDL 2000 kiadása bevezeti a védett típusokat
- IEEE Std. 1076-2002, *IEEE 1076-2002* – IEEE Standard VHDL Language Reference Manual, felülvizsgálata az IEEE Std 1076-2000 kiadásnak
- IEEE Std. 1076.6-2004, *IEEE 1076.6-2004* – IEEE Standard for VHDL Register Transfer Level (RTL), IEEE Std 1076-1993, IEEE Std 1076-2002 felülvizsgálata
- IEEE Std. 1076-2008, *IEEE 1076-2008* – IEEE Standard VHDL Language Reference Manual- VHDL-2008.
- IEEE Std. 1029.1-1998, *IEEE 1029.1-1998* – IEEE Standard For VHDL Waveform and Vector Exchange (Waves) to Support Design and Test Verification

1.3. VHDL szintaktika és VHDL elemek

A VHDL hardverleíró nyelv bevezetése szempontjából kiemelném a VHDL strukturális felépítését (entitás, architektúra) és a nyelvi elemeket: lexikális szabályok, objektumok, adattípusok és operátorok.

A VHDL hardverleíró nyelv strukturálisan a következő részekbe van szervezve:

- Könyvtárak és csomagok (library and package)
- Entitásdeklaráció (entity declaration)
- Entitásleírás
  - Strukturális
  - Viselkedési
  - Szimulációs állomány (szimuláció során a bemeneti jelek generálása)

1.3.1. Lexikális elemek

A VHDL hardverleíró nyelv keretében előforduló lexikális elemek:

- Megjegyzések "- -"
- Azonosítók
  - Alkalmazható karakterek
    - Latin ABC betűi
    - Számok
    - Aláhúzás
  - Operátorok: +, -, &, *, /, ., <, =, >, |, /=, :=, >=, <=, <>
  - Tömbök
- Kulcsszavak
- Számok
  - Egész
  - Lebegőpontos
  - Valós
- Karakterek, karakterláncok

A VHDL nem tesz különbséget a kis- és nagybetű között.

A VHDL hardverleíró nyelven meghatározott kulcsszavak:

abs, access, after, alias, and, architecture, array, assert, attribute, begin, block, body, buffer, bus, case, component, configuration, constant, disconnect, downto, else, elsif, end, entity, exit, file, for, function, generate, generic, guarded, if, impure, in, inertial, inout, is, label, library, linkage, literal, loop, map, mod, nand, new, next, nor, not, null, of, on, open, or, others, package, port, postponed, procedure, process, pure, range, record, register, eject, rem, report, return, rol, ror, select, severity, shared, signals, sra, srl, subtype, then, to, transport, type unaffected, units, until, use, variable, wait, when, while, with, xnor, xor

A kulcsszavak nem alkalmazhatóak változók, konstansok, azonosítók megnevezésére.

1.3.2. Objektumok VHDL-ben

Az alegységek, komponensek összekapcsolása jelekkel valósul meg. Különböző típusú jeleket különböztetünk meg: signal, port és variable. A különböző jelek között különbség a láthatóságban van. A portjelek globálisak, és az entitás bemeneti és kimeneti jeleinek deklarálására alkalmazzuk. A signalok a csomagban és az architektúra részben érhetőek el. A változók csak lokálisan a folyamatokban alkalmazhatóak.

A jel és a változó pontos alkalmazására visszatérünk a *process* kifejezés részletes tárgyalásakor.

Hasonlóan a többi programozási nyelvhez, lehetőség van konstansok (*constant*) deklarálására.

Egy speciális objektum a VHDL keretében az állományok kezelésére szolgáló $file$. Az állományok hasznosak vektorok tárolására, amelyeket szimuláció során az egyes bemenetek stimulálására vagy meghajtására lehet felhasználni. Működésük hasonlít általában a programozási nyelvek keretében alkalmazott állományműveletekre, mint például C-ben. Fontos megjegyezni, hogy a *file* utasítás nem szintetizálható, csak szimuláció során alkalmazható bemeneti jelek állományból való beolvasására és a szimuláció eredményének kiírására.

Számok ábrázolása különböző számrendszerekben

A számok ábrázolására a következő két lehetőséget említjük meg:
- N-alapú számok
- Bit-stringek

N-alapú számok

$Based\_literal ::= base\#basedinteger[.basedinteger]\#[exponent]$
A 196-ot binárisan, hexadecimálisan és négyes számrendszerben a következő példák szemléltetik:

1. Binárisan megadott egész szám: 2#1100_0100#
2. Hexadecimálisan megadott egész szám: $16\#C4\#$
3. Négyes számrendszerben megadott egész szám: $4\#301\#E2$

A 4095.0 valós szám ábrázolása különböző számrendszerekben a következőképpen alakul:

1. Binárisan megadott valós szám: $2\#1.1111\_1111\_111\#E+11$
2. Hexadecimálisan megadott valós szám: $16\#FF.F\#E+1$

Bit-stringek

$Bit\_string_literal$ ::= $base\_specifier$"bit_value"
$base\_speifier$ ::= $B|O|X$
bit_value ::= $extended\_digit[underline]extended\_digit$
A számok bit-strinként való ábrázolása különböző számrendszerekben:

1. Binárisan megadott szám:
   B"1010110" 8 bit
2. Nyolcas számrendszerben meghatározott szám:
   O"126" 3x3bit → B"001_010_110"
3. Hexadecimálisan megadott szám:
   X"56" 2x4 bit → B"0101_0110"

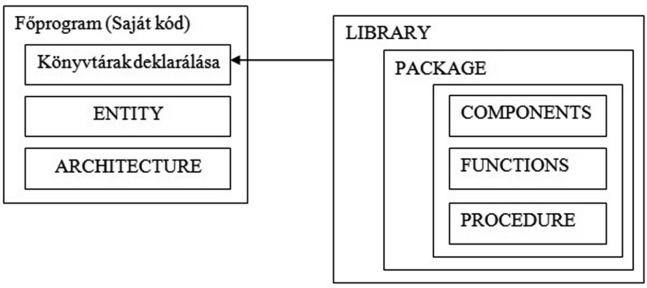
1.4. A VHDL modell szerkezete

A VHDL programrész első soraiban a használt könyvtárakat (*LIBRARY*) és könyvtáron belüli csomagokat jelöljük meg. A következő részben (*ENTITY*) a tervezési egységek portjeleit és paramétereit határozzuk meg. Az utolsó részben következik a megvalósítási egység (*ARCHITECTURE*), amely szintén két alrészre bontható: deklarációs, illetve megvalósítás részre.

A VHDL szerkezetét a következő ábra 1.7 mutatja [2]:

Amint az 1.7. ábra szemlélteti, egy VHDL programban három különböző részt különíthetünk el:

- *LIBRARY* – könyvtárhasználat,
- *ENTITY* – tervezési egység [8] (egyed [9]) meghatározása,
- *ARCHITECTURE* – megvalósítási egység (építmény [9]).



1.7. ábra. A VHDL szerkezeti felépítése

1.4.1. Könyvtárak deklarálása

A könyvtárakban az elemeket csomagokba (*PACKAGE*) szervezték. A *LIBRARY* kulcsszóval bejelentjük, hogy melyik könyvtárban találhatók az elemek, amelyeket szeretnénk alkalmazni, valamint a USE utasítással kijelöljük, hogy a könyvtáron belül melyik csomagot, és azon belül pontosan mely elemet szeretnénk alkalmazni.

A könyvtárak felhasználásához két utasítás alkalmazására van szükség, a LIBRARY, illetve USE a következő példa szerint:

- *A LIBRARY konyvtar_neve*
- *USE konyvtar_neve.csomag_neve.csomag_resze*

Egy terv elkészítésénél minimálisan három könyvtárra van szükség:
- *ieee.std_logic_1164* (IEEE könyvtárból)
- *std* standard könyvtárból
- *work* saját munkakönyvtárból

Fontosabb IEEE könyvtárcsomagok

A tervezés során általában az IEEE könyvtárcsomagból a következőket alkalmazzák gyakrabban [3]:
- **Std_logic_1164** csomag tartalmazza a két leggyakrabban használt adattípust *std_logic* és *std_ulogic*. Az *std_logic* típus kilenc lehetséges értéket vehet fel.
- **Std_logic_arith** csomag a *signed* és *unsigned* adattípusok és erre vonatkozó aritmetikai, összehasonlító és adatkonverzióra használt műveleteket *conv_integer(p)*, *conv_unsigned(p,b)*, *conv_signed(p,b)*, *conv_std_logic_vector(p,b)* tartalmazza.

– **Std_logic_signed** csomag az *STD_LOGIC_VECTOR* adaton értelmezett függvényeket tartalmaz ha az adat *SIGNED* típusú.
  A csomag támogatja az inkrementálás, dekrementálás műveleteknek a használatát *std_logic_vector* típusokon.
– **Std_logic_unsigned** csomag az *STD_LOGIC_VECTOR* adaton értelmezett függvényeket tartalmaz, ha az adat *UNSIGNED* típusú.
– **Numeric_std** csomagban *signed* és *unsigned* adattípusok és erre vonatkozó aritmetikai, összehasonlító és adatkonverzióra használt műveletek vannak értelmezve.

Az *STD_LOGIC_ARITH*, *Numeric_std*, valamint az *STD_LOGIC_SIGNED* és *STD_LOGIC_UNSIGNED* csomagokban értelmezett műveleteket a 1.1., 1.2., 1.3. táblázatok foglalják össze.

1.1. táblázat. STD_LOGIC_ARITH fontosabb elemei [2], [3], [10]

Művelettípus	Műveletek
Adattípusok	unsigned, signed
Aritmetikai műveletek	+, -, *
Összehasonlító műveletek	<, <=, >, >=, =, /=
Bitmozgatás	shl, shr
Konverzió	conv_integer, conv_unsigned, conv_signed

1.2. táblázat. NUMERIC_STD csomag fontosabb elemei [2], [3], [10]

Művelettípus	Műveletek
Aritmetikai műveletek	+, - , *, /, rem, mod
Összehasonlító műveletek	>, <, <=, >=, =, /=
Forgatási és bitmozgatási műveletek	sll, srl, rol, ror
Konverziós műveletek	TO_INTEGER, TO_UNSIGNED, TO_SIGNED
Logikai műveletek	not, and, or, nand, nor, xor, xnor

Az *STD_LOGIC_SIGNED* és *STD_LOGIC_UNSIGNED* kiterjeszti az *std_logic_arith* könyvtárakat úgy, hogy az *std_logic_vector* típusokat előjeles vagy előjel nélküli egészként kezelje. Az *std_logic_arith* könyvtárhoz hasonlóan az 1.3. táblázatban definiálva vannak a fontosabb műveletek:

1.3. táblázat. A NUMERIC_STD csomag fontosabb elemei [3], [10]

Művelettípus	Műveletek
Aritmetikai műveletek	+, -, *
Összehasonlító műveletek	<, <=, >, >=, =, /=
Bitmozgatás	shl, shr
Konverzió	conv_integer

A kiterjesztésben az *std_logic_vector* értékeket argumentumként határozzuk meg, és előjeles vagy előjel nélküli egészként vannak értelmezve.

1.4.2. Az ENTITY részletezése

Röviden úgy lehetne összefoglalni, hogy az *ENTITY* rész tartalmazza és leírja a modul jeleit, amelyek segítségével kapcsolódhat egyéb modulokhoz. A tervezés során az *ENTITY* részben két dolgot határozunk meg:

- *GENERIC* – paraméterek, áramköri jellemzők, pl. frekvencia, késleltetés, sínszélesség,
- *PORT* jelek – fizikai kapcsolat a külvilággal.

A *GENERIC* változókra a következőkben generikus változóként hivatkozunk [10]. A generikus változók segítségével parametrizálható, általánosítható egy áramkör. Például bistabil áramkörökből szeretnénk egy léptetőregisztert felépíteni. A generikus paraméter segítségével általánosítható, hogy 2, 4, 8 bites regisztert szeretnénk kialakítani. A példányosítás során eldönthető, hogy egy tervben éppen melyik változatot szeretnénk alkalmazni. A regiszter portjelei természetesen kapcsolódnak egyes jelekhez. A generikus változókkal egységesen megvalósítható, hogy a regiszter méretének függvényében 2, 4, 8 bites jelekkel kapcsoljuk össze a modulokat.

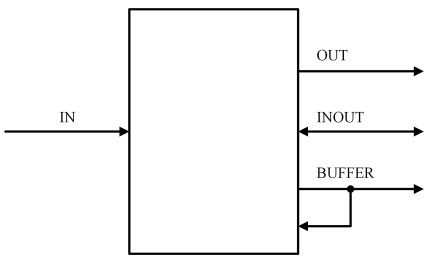
A *PORT* jelek meghatározzák, hogy a modulhoz milyen bemeneteken, kimeneteken szeretnénk kapcsolódni. Egy listát tartalmaz az áramkör összes be- és kimenetével, meghatározva a jel irányát, típusát, jelméretét [3].

```
entity entitas_neve is
generic
--Generikus parameterek
(parameter:parameter_tipusa:=parameter_erteke);
port (
   jel1_neve : jel_iránya jel_modjel1_tipus [:= kezdő érték];

   -- opcionális a kezdőérték adás
   jel2_neve : jel_iránya jel_modjel2_tipus [:= kezdő érték];
   -- ...
   jeln_neve : jel_iránya jel_modjeln_tipus [:= kezdő érték]
   -- az utolsó sorban a sor végén nincs ;
);
end entitas_neve;

-- példa portjelek meghatározására
port(
   jel_1: in std_logic := '0';  -- opcionális a kezdőérték
       adás!
   ...
   jel_n: out std_logic_vector(0 to 7)
);
```

A PORT jelek a következő típusúak lehetnek:

- *in*: bemeneti jelek,
- *out*: kimeneti jelek,
- *inout*: ki- és bemeneti jelek,
- *buffer*: kimenet, amely vissza van csatolva bemenetként a modulba [11].

A porttípusokat az 1.8. ábrán szemléltetjük.

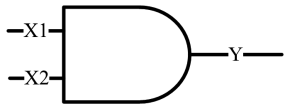


1.8. ábra. Portjelek típusainak ábrázolása

A bemeneti portokat csak olvasni lehet, a kimenetit csak írni és a be-kimenetit írni-olvasni, a *buffer* pedig visszacsatolt kimenet. Ritkábban

használt porttípus a *linkage*, ugyancsak csökkentett hatáskörű ki-bemenet, viszont szintetizálható tervekben nem használt.

A bemeneti jeleken visszük be az információt a modulba, a kimeneti jeleken pedig a modul válaszát, amelyek kapcsolódhatnak más modulok bemeneteire vagy a külvilágra. Az *inout* egyaránt lehet ki- és bemenet és kétirányú sínrendszerek megvalósítására alkalmazható. Több modul kimeneteit sínrendként összekötve, egyik modul kimeneti portját leválaszthatjuk, ha a port jeleit nagy impedanciás állapotba állítjuk. A kétirányú illesztés a *WHEN ELSE* kifejezéssel valósítható meg. A *buffer* típusú kimenet bemenetként vissza van csatolva a modulba.



1.9. ábra. Az ÉS kapu portjelei

Az 1.9. ábra szerint az ÉS kapu jelei a következőképpen vannak leírva két különböző módszerrel:

```
entity es_kapu is
--port jelek deklarálása
   port (
      X1: in bit;  --bemeneti jel
      X2: in bit;  --bemeneti jel
      Y: out bit;) --kimeneti jel
end es_kapu


entity es_kapu is
--port jelek deklarálása
port (
   X: in std_logic_vector(1 downto 0); -- két bites bemeneti
       jel
   Y: out std_logic); --kimeneti jel
end es_kapu;
```

A második példában az ÉS kapu bemenete egy vektor, első esetben a kapu bemenetei pedig egybites jelek.

1.4.3. Az ARCHITECTURE leírása

A VHDL-ben az *ARCHITECTURE* részben a modulnak a működését írjuk le. Az *ARCHITECTURE* egységen belül két külön rész különíthető

el. Az *ARCHITECTURE* és a *BEGIN* közötti rész úgynevezett deklarációs rész. Ebben a részben soroljuk fel azon elemeket, amelyeket szeretnénk alkalmazni a megvalósítandó modul leírásában:

- *COMPONENT* modulon belüli alkatrészek, *FUNCTION* függvények, *PROCEDURE* eljárások, amelyeket felhasználunk az új modul kialakításában,
- az egyes elemek összeköttetésére alkalmazott jelek,
- *CONSTANT* konstansok, *TYPE* új típusok létrehozása [3].

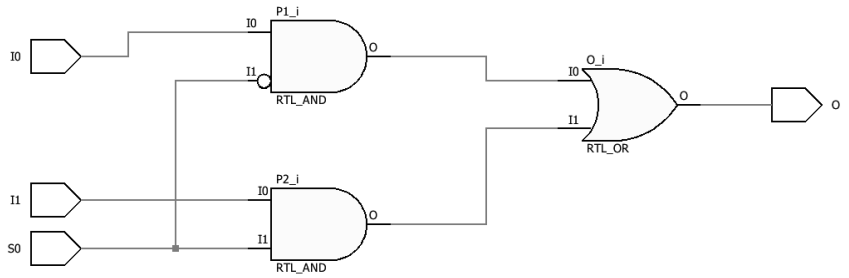
A *BEGIN* és az *END* rész közötti rész tartalmazza a modul működésének leírását vagy szerkezeti felépítését. Egy adott modul működésének a leírására két lehetőség van:

- viselkedési szinten – algoritmusszerűen történik a rendszer működése,
- strukturálisan – az alegységek közötti kapcsolatok leírásával jellemezzük a rendszert.

Az alábbi példa egy egyszerű multiplexer áramkör VHDL-ben való leírását szemlélteti:

```
ENTITY mux
--port jelek deklarálása
Port(
   I0: IN bit; -- adatbement
   I1: IN bit; -- adatbemenet
   S0: IN bit; -- kiválasztó bemenet
   O: OUT bit); --multiplexer kimenete
END mux;


ARCHITECTURE arch1 OF mux IS
BEGIN
-- ha S0 akkor I1 egyébként meg I0
   O<= ((not S0) and I0) or (S0 and I1);
END arch1

ARCHITECTURE arch2 OF mux IS
   Signal p1:bit; --belső jel daklarálás
   Signal p2:bit; --belső jel daklarálás
BEGIN
--egyszerűbb függvényekre felbontott változat
   p1<=(not S0)and I0; -- ha nem S0 akkor I0
   P2<= S0 and I1;     -- ha S0 akkor I1
   O <= p1 OR p0;      -- p1 vagy p0
END arch2;
```

A példa alapján a modul működését a multiplexert leíró logikai függvénnyel határoztuk meg. A második változatban a logikai függvény fel van bontva egyszerűbb függvényre. A kapuszintű áramköri rajz az 1.10. ábrán látható.



1.10. ábra. Multiplexer áramkör kapcsolási rajza

1.4.4. Szabványos adattípusok

A VHDL-ben alapértelmezetten a következő szabványos adattípusok vannak meghatározva:

- *Integer* – egész típus [-2 147 483 647, 2 147 483 647]
- *Boolean* – Boole típus true, false
- *bit* típus: '0', '1' logikai '0' vagy logikai '1' értéket vehet fel
- *bit_vector* típus – bit típusú elemekből álló tömb
- real

A *bit*, *bit_vector*: hátránya az, hogy a jelhez csupán kétállapotú jelszinteket lehet rendelni, magas impedanciás állapot jelkonfliktust eredményez. A konfliktust az *IEEE Std. 1076-1993* szabványban bevezetett *STD_LOGIC* és *STD_LOGIC_VECTOR* típusokkal oldották fel.
Az *STD_LOGIC* típus a következő logikai szinteket értelmezi : 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-', amelyek az 1.4 táblázatban vannak értelmezve.

Két *STD_LOGIC* típusú jel csatlakozásakor az 1.11. ábrán szemléltetett igazságtáblázat érvényesül [11]:

1.4. táblázat. STD_LOGIC típusban értelmezett logikai szintek

Szimbólum	Értelmezés
'U'	Nem inicializált jel értéke
'X'	Határozatlan értékű jel
'W'	Gyenge határozatlan logikai állapot, egy jel olyan értéket vesz fel, amit nem lehet sem logikai 0-nak, sem logikai 1-nek értelmezni.
'0'	Határozott logikai '0', akkor veszi fel egy jel, ha szabványos meghajtó áramkör vezérli
'1'	Határozott logikai '1', akkor veszi fel egy jel, ha szabványos meghajtó áramkör vezérli
'Z'	Nagy impedanciás érték, háromállapotú meghajtó áramkör
'L'	Gyenge logikai '0', a meghajtó vezérlő áramkör gyenge meghajtó áramot szolgáltat
'H'	Gyenge logikai '1', a meghajtó vezérlő áramkör gyenge meghajtó áramot szolgáltat
'-'	Redundáns érték

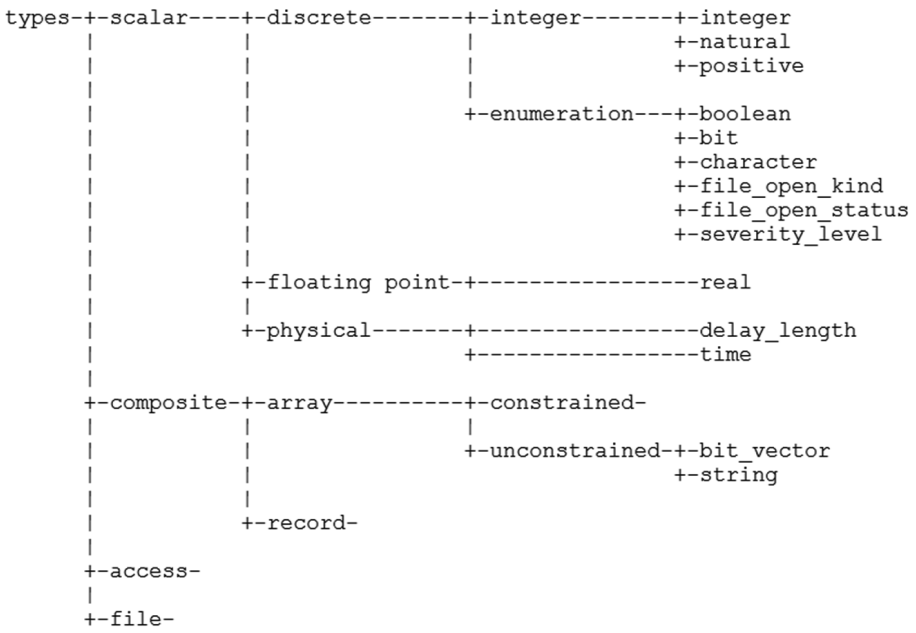
```
CONSTANT and_table : stdlogic_table := (
--      ----------------------------------------------------
--      |  U    X    0    1    Z    W    L    H    -        |   |
--      ----------------------------------------------------
        ( 'U', 'U', '0', 'U', 'U', 'U', '0', 'U', 'U' ),  -- | U |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | X |
        ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ),  -- | 0 |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | 1 |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | Z |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | W |
        ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ),  -- | L |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | H |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' )   -- | - |
);
```

1.11. ábra. Két STD_LOGIC típus jel csatlakozásának igazságtáblázata

VHDL-ben alkalmazott adattípusok:
- Egész típusok
- Lebegőpontos típusok
- Fizikai mennyiség
- Felsorolás (enumeration)
- Vektorok (Arrays)
- Rekordok

Egy részletes összefoglaló a VHDL-ben alkalmazható típusokról az 1.12. ábrán látható.

A VHDL-ben elérhető típusokhoz újabb típusok definiálhatóak.



1.12. ábra. VHDL típus-összefoglaló

Egész típus definíció

A VHDL nyelv lehetővé teszi különböző bitszélességű előjel nélküli és előjeles számok alkalmazását. Az intervallumot a *to* vagy *downto* kifejezés határozza meg. Különböző egész típusok létrehozását az alábbi példák szemléltetik.

```
type egesz_tipus range  egyszeru_kifejezes [ to | downto]
    egyszeru_kifejezes
--előjel nélküli egész típus  [0 - 255] intervallumon
Type byte_int is range 0 to 255;
--előjeles egész -32768 tól 32767-ig, vagyis előjeles, 16
    bites számnak megfelelő egész
Type signed_word_int is range -32768 to 32767;
--egész típus 31-től 0-ig
```

```
Type bit_index is range 31 downto 0;
--egész típus 0-tól 31-ig a hónap napjainak kódolására
Type honap_napjai is range 0 to 31;
--változó deklarálása a létrehozott típus alapján
Variable mai_nap : honap_napjai:=3;
--konstans deklarálása és inicializálása
Constant konstans_azonosito : integer :=32;

ARCHITECTURE az_architektura_neve OF entitas_neve
--Deklarációs rész

   --A deklarációs részben deklarálhatók a következő elemek:
   [jelek SIGNAL]
   [alkatrészek COMPONENT]
   [függveny FUNCTION]
   [konstansok CONSTANT]
   [típusok TYPE]

BEGIN
   Programkod resz;
END az_architektura_neve;
```

Egész típuson végrehajtható műveletek:

+	összeadás
−	kivonás
∗	szorzás
/	osztás
mod	modulo
rem	egésszel való osztás maradéka
abs	abszolút érték
∗∗	hatványozás

Lebegőpontos számábrázolás

Lehetőség van valós adattípus alkalmazására. A típus létrehozásakor meg kell határozni az intervallumot, valamint pontosságot. A pontosságot a tizedespont utáni számjegyek (általában nullások) száma határozza meg. A valós számok nem alkalmazhatóak szintézisre, csak szimulációra.

```
-- valós típus [-10.00 10.00] intervallumon
Type jelszint is range -10.00 to 10.00;
-- valós típus [0.0 1.0] intervallumon
Type valoszinuseg is range 0.0 to 1.0;
```

Előre definiált valós számtípus: Real -1E38-tól 1E38-ig

Felsorolástípus

Hasonlóan más programnyelvekhez, a VHDL-ben is létezik egy felsorolástípus. A felsorolástípus egy speciális adattípus, amely rendezett értékkészlettel rendelkezik, tartalmaz egy azonosítót és a felsorolásban részt vevő elemeket. A típus alapján létrehozott jel vagy változó a felsorolt elemekből vehet fel értéket. Minden felsorolt elem rendezett, és mindegyikhez van egy numerikus érték rendelve. Az első elemhez rendelt érték nullás, a következő elemekhez rendelt érték eggyel növekszik. A felsorolástípus egyik lehetséges alkalmazása a véges állapotautomatákban az állapotok kódolása.

A VHDL-ben alapértelmezetten definiált felsorolástípusok:

```
type CHARACTER is ( {full ISO 8859-1: 1987(3) karakter
    szett} );
-- bit felsorolástípus deklarálása
type BIT is ( '0', '1' );
-- boole felsorolástípus deklarálása
type BOOLEAN is (FALSE, TRUE);
-- különböző hibakezelési szintekre felsorolás típus
-- létrehozása
type SEVERITY_LEVEL is ( NOTE, WARNING, ERROR, FAILURE );
```

Az alábbiakban egy pár példa a felsorolástípus alkalmazására és a típusok leírására:

```
--típus deklarálása hibaszintek kezelésére
Type logikai_szintek  is (unknown, low, undriven, high);
-- felsorolás típus deklarálása lehetséges aritmetikai
-- műveletek kódolására
Type alu_muveletek is (add, substract, multiply, divide);
-- felsorolástípus alkalmazása nyolcas számrendszer
-- szimbolumainak ábrázolására
Type octal_digit is ('0', '1', '2', '3','4', '5', '6', '7');
```

Vektorok, mátrixok

A tömbök összetett objektumok, amelyek minden egyes eleme ugyanolyan altípusú. Minden egyes elem egy vagy több index szerint címezhető, amelyek a definíció szerinti típushoz tartoznak. Az indexeknek a száma megegyezik a dimenzióknak a számával. Az indexek sorrendje követi a típusdeklarációban meghatározott dimenzió sorrendjét. A tömb lehet korlátos

vagy általános. A nem korlátos tömbök mérete előre ismeretlen. A tömb mérete korlátozható diszkrét adattípus alkalmazásával vagy a range kifejezés megadásával. Mindkét esetben a tömb pontos mérete fordítás előtt ismert.

Korlátozott tömbtípus deklarálása:

```
-- típus létrehozása 32 elemű tömbre, amelynek minden eleme
-- egy-egy bitet tárol
Type word is array(31 downto 0)  of bit;
-- tömb típus létrehozása, minden egyes elem 16 bites
-- std_logic_vector típusú értéket képes tárolni
Type memory is array(address) of std_logic_vector(15 downto
    0);
-- típus létrehozása kétdimenziós tömbre, mindkét dimenzió
-- szerint 1-től 4-ig címezhető a tömb és minden cellája
-- real típusokat tárol
Type transform is array(1 to 4, 1 to 4) of real;
-- 32 elemű tömbre típus létrehozása, minden cella 1 bit
-- típusú adatot tárol
type WORD_32 is array (31 downto 0) of BIT;
-- típus létrehozása egy tömbre, a tömb 0-tól 127-ig
-- címezhető, minden tömbelem egy egész (integer)
Type register_bank is array(byte range 0 to 127) of integer;
```

Nem korlátozott tömbtípus deklarálása:

```
-- nem korlátozott tömbre típus létrehozása, amely, real
    típusú adatokat tárol
Type vektor is array(integer range <>) of real;
-- nem korlátozott tömb, amely természetes számokkal
    címezhető, és bit típusokat tárol
Type bit_vector is array(natural range <>) of bit;
--tömb elemeinek címzése
A(1)='1'  vagy  b(1,1)='1';

A(8 to 15)=x;  --ahol x egy 8 bitet tartalmazó vektor
```

Rekordok deklarálása és alkalmazása

```
--record típus létrehozása utasítás kódolására
Type utasitas is
Record
utasitas_kod : processzor_utasitas;
cimzesi_mod: cimzesi_mod_tipus;
operandus_1 : integer range 0 to 15;
operandus_2: integer range 0 to 31;
End record utasitas;
```

```
-- record típus létrehozása dátum kódolására
type DATUM is record
HONAP : STRING (0 to 20); --honap elnevezésée
NAP : INTEGER range 1 to 31;  --nap
EV : INTEGER range 1800 to 2050;  --év
end record DATUM;

--record típus létrehozása lakcím kódolására
Type CIM is record
UTCA : STRING (0 to 50);
IRANYITOSZAM : INTEGER range 0 to 99999;
VAROS : STRING (0 to 50);
ORSZAG : STRING (0 to 40);
end record ADDRESS;
```

Altípusok alkalmazása

Az altípus (subtype) lehetővé teszi, hogy az objektum által felvehető értékeket korlátozzuk az alaptípus csak egy bizonyos részére (intervallumára).

```
subtype kimenetek_szama is integer range  0 t0 400;
subtype digits is character range '0' to '9';
subtype MEM_ADR is INTEGER range 0 to 1023;
subtype BUS_VAL is INTEGER range 0 to 65535;
```

Konstansok

Konstansok alkalmazásakor a konstans létrehozáskor kap egy kezdeti értéket, és végig megmarad a kezdeti értéke. Példák konstansok deklarálására:

```
--e valós konstansként való deklarálása
constant  e: real :=2.71828;
--5 ns késleltetés konstansként való deklarálása
constant delay: Time :=5ns;
-- konstans deklarálása
constant max_size : natural:=16;

-- álnév hozzárendelése az intr-hez, amelyet bit_vector
-- típusként értelmezünk
alias op_code : bit_vector(7 downto 0) is intr(31 downto 24);
```

Fizikai mennyiségek

A fizikai típus egy numerikus skalár, amely bizonyos mennyiséget képvisel. A fizikai típus bármely mértékegysége az elsődleges mértékegység egész számú többszöröse.

```
type tipus_neve is range also_hatar to felso_hatar
units
   elsodleges_mertekegyseg_neve;
   masodlagos_mertekegyseg_neve = tobbszoros;
       elsodleges_mertekegyseg_neve
   masodlagos_mertekegyseg_neve = tobbszoros;
       elsodleges_mertekegyseg_neve
   . . .
end units tipus_neve;
```

Az alábbi példákban egy pár mértékegységtípus leírása van szemléltetve: távolság, idő [12], kapacitás.

```
--távolságra mértékegység létrehozása
type distance is range 0 to 1E16
units Ang; -- angstrom, elsődleges mértékegység
   nm = 10 Ang; -- nanometer
   um = 1000 nm; -- micrometer (micron)
   mm = 1000 um; -- millimeter
   cm = 10 mm; -- centimeter
   dm = 100 mm; -- decameter
   m = 1000 mm; -- meter
   km = 1000 m; -- kilometer
   mil = 254000 Ang; -- mil (1/1000 inch)
   inch = 1000 mil; -- inch
   ft = 12 inch; -- foot
   yd = 3 ft; -- yard
   fthn = 6 ft; -- fathom
   frlg = 660 ft; -- furlong
   mi = 5280 ft; -- mile
   lg = 3 mi; -- league
end units;

--idő mérésére mértékegység létrehozása
type Time is range
units fs; -- femto szekundum
   ps = 1000 fs; -- pico-szekundum
   ns = 1000 ps; -- nano-szekundum
   us = 1000 ns; -- mikroszekundum
   ms = 1000 us; -- milliszekundum
```

```
   sec = 1000 ms; -- szekundum
   min = 60 sec; -- perc
   hr = 60 min; -- óra
end units;

--mértékegység típus kapacitás leírására
type CAPACITANCE is range 0 to 1000000000 -- intervallum
-- meghatározása
units pf ; -- alapmértékegység picofarad
   nf = 1000 pf; -- nanofarad,
   uf = 1000 pf; -- mikrofarad
   mf = 1000 uf; -- millifarad
   f = 1000 mf; -- farad
end units CAPACITANCE ;
```

Attribútumok

Az attribútumok további kiegészítő információkat nyújtanak egy-egy elemről: jelekről, változókról, tömbökről, típusról vagy alkatrészről [13].

Az attribútumok négy kategóriába sorolhatók:

- típushoz kötődő attribútum,
- jelattribútum,
- skalárhoz kapcsolódó attribútum,
- tömbre jellemző attribútum.

Az attribútumra a következőképpen hivatkozunk: *objektum_neve'attribútum*.

Entitásokra (1.8. táblázat) típusokra (1.5. táblázat), tömbökre (1.6. táblázat), jelekre (1.7. táblázat) számos előre meghatározott attribútum áll a tervező rendelkezésére [13], [11].

Saját attribútumok létrehozása:

Egy saját attribútum létrehozása két lépésben valósítható meg:

- az attribútum deklarálása,
- az attribútum specifikálása.

Attribútum deklarálása:

```
ATTRIBUTE attributum_neve : attributum_tipusa
```

Attribútum specifikálása:

```
ATTRIBUTE  attributum_neve: OF  cel_elem_neve : osztaly IS
    ertek;
```

Attribútum típusa: bármely adattípus BIT, INTEGER, STD_LOGIC, STD_LOGIC_VECTOR, STRING stb.

osztaly: architecture, component, configuration, constant, entity, file, function, group, label, literal, package, procedure, signal, subtype, type, variable, units

ertek: az osztály által felvehető érték mint például ”11111”, 27, ”WRITE_FIRST”, ”READ_FIRST”.

1.5. táblázat. Típusattribútumok

Attribútum neve	Attribútum leírása
T’BASE	a T típus alaptípusa
T’LEFT	T legbaloldalibb értéke
T’RIGHT	T legjobboldalibb értéke
T’HIGH	T-ből a legnagyobb érték
T’LOW	T-ből a legkisebb érték
T’ASCENDING	logikailag igaz, ha a T tartománya növekszik, másként hamis
T’VALUE(X)	X karakterláncból átalakított T típusú érték
T’IMAGE(X)	T-ben tárolt érték karakterláncként ábrázolva
T’POS(X)	X egész számú pozíciója a T-ben
T’VAL(X)	X-edik elem értéke a T-ből
T’SUCC(X)	X utáni elem értéke a T-ből
T’PRED(X)	X előtti elem értéke a T-ből
T’LEFTOF(X)	X-től balra eső elem értéke a T-ből
T’RIGHTOF(X)	X-től jobbra eső értéke a T-ből

Jelek és változók értékadása

Jelek deklarálása és értékadása

```
--jelek deklarálása
--négy bites std_logic_vector típusú jel deklarálása
signal s1: std_logic_vector(3 downto 0);
signal s2: std_logic_vector(3 downto 0);
--nyolc bites std_logic_vector típusú jel deklarálása
signal w:  std_logic_vector(0 to 7);
.....
--jel  értékadása:  <=

s1 <= s2; -- az s1 std_logic_vector típusú, 4 bites jel
    felveszi az s2 értékét
```

```
s1 <= v2; -- az s1 std_logic_vector típusú, 4 bites jel
    felveszi az v2 értékét
w <= "10000000"; --a w std_logic_vector típusú nyol bites
    jel felveszi a bináris "10000000" értéket
w <= (0=>'1', others=>'0') --a w std_logic_vector típusú, 8
    bites jel nuladik bitje '1' -es értéket vesz fel, a többi
    pedig '0'-'t.
```

1.6. táblázat. Tömbattribútumok

Attribútum neve	Attribútum leírása
A'LEFT	az A tömb (vagy korlátos méretű tömb) bal szélső indexe
A'LEFT(N)	az A tömb N-edik dimenzió (vagy korlátos méretű tömb) bal szélső indexe
A'RIGHT	az A tömb (vagy korlátos méretű tömb) jobb szélső indexe
A'RIGHT(N)	az A tömb N-edik dimenzió (vagy korlátos méretű tömb) jobb szélső indexe
A'HIGH	az A tömb indextartományának felső értéke
A'HIGH(N)	az A tömb N-edik dimenzió indextartományának felső értéke
A'LOW	az A tömb indextartományának alsó értéke
A'LOW(N)	az A tömb N-edik dimenzió indextartományának alsó értéke
A'RANGE	A tömb indextartománya
A'RANGE(N)	A tömb N-edik dimenziójának indextartománya
A'REVERSE_RANGE	A tömb fordított indextartománya
A'REVERSE_RANGE(N)	A tömb N-edik dimenziójának a fordított indextartománya
A'LENGTH	A tömb elemeinek a száma
A'LENGTH(N)	A tömb N-edik dimenziója elemeinek a száma
A'ASCENDING	igaz, ha az A tömb indextartománya növekvő
A'ASCENDING(N)	igaz, ha az A tömb N-edik dimenziójának az indextartománya növekvő

Változók deklarálása és értékadása

```
--négy bites std_logic_vector típusú változó deklarálása és
    inicializálása
variable v1: std_logic_vector(3 downto 0):="0000";
variable v2: std_logic_vector(3 downto 0):="1010";
variable y: std_logic_vector(3 downto 0):="0000";
.....
```

```
--változó értékadása

v1 := s2;
v2 := v2;
y:="1010";
```

1.7. táblázat. Jelattribútumok

Attribútum neve	Attribútum leírása
S'DELAYED(t)	S jel időben t egységnyit késleltetve
S'STABLE	igaz, ha nem történt esemény az S jelen
S'STABLE(t)	igaz, ha az utóbbi t egységnyi idő alatt nem történt esemény az S jelen
S'QUIET	igaz, ha nem történt esemény az S jelen (ebben a szimulációs ciklusban)
S'QUIET(t)	igaz, ha az S jel t-időegység alatt nem változott
S'TRANSACTION	egy bit típusú jel, amely mindig vált, amikor az S jelen megjelenik egy tranzakció. A tranzakciót egy jel értékadása váltja ki. A tranzakció hatására, ha a jel értéket vált, akkor eseményről van szó
S'EVENT	igaz, ha esemény történt az S jelen a szimulációs ciklus alatt
S'ACTIVE	igaz, ha egy tranzakció történt az S jelen a szimulációs ciklus alatt
S'LAST_EVENT	az utolsó eseménytől eltelt idő
S'LAST_ACTIVE	az utolsó tranzakciótól eltelt idő
S'LAST_VALUE	az S jel utolsó esemény előtti értéke
S'DRIVING	igaz, ha a folyamat vezérli az S jelet vagy az S bármelyik elemét
S'DRIVING_VALUE	az S jelet meghajtó aktuális értéke abban a folyamatban, amely tartalmazza az S értékadását

1.8. táblázat. Entitásattribútumok

Attribútum neve	Attribútum leírása
E'SIMPLE_NAME	E nevét tartalmazó karakterlánc
E'INSTANCE_NAME	az E elérési útvonala, beleértve a példányosított entitás nevét is
E'PATH_NAME	az E elérési útvonala viszont nem tartalmazza a példányosított entitás nevét

A változókat csak szekvenciális részekben, *process*en belül lehet alkalmazni, viszont az értékadás azonnal megtörténik. A változókat, hasonlóan a programozási nyelvekhez, értékek tárolására alkalmazzuk. Egy megvalósított áramkörben egy változó például egy regiszter tartalmának a tárolására szolgál.

A jeleket modulok összekapcsolására alkalmazzuk. A jelek alkalmazhatóak egyaránt konkurens és szekvenciális utasításokban. Szekvenciális utasításokban való alkalmazásuk során (process utasításban) az értékadás a processből való kilépéskor valósul meg.

A változók értékadásához hasonlóan a változók inicializálására, konstansok és generikus változók inicializálására is a ':=' szimbólumot használjuk.

A vektor elemeinek értékadása

A vektor elemeinek az értékadására két lehetséges módot különböztetünk meg:

- sorrendi hozzárendelés,
- megnevezett hozzárendelés.

A sorrendi hozzárendelés során egymást követően felsoroljuk a vektor inicializálására szánt elemeket. A sorrend, vagyis annak alapján, ahogyan következnek egymás után az elemek, kerülnek be a vektorba.

A=('f','b','o','d')

A megnevezett értékadás során megadjuk, hogy a vektor melyik elemét szeretnénk inicializálni. Az alábbi minta szerint a vektor harmadik elemét 'f'-fel, a 4-edik elemét 'd'-vel és az összes többi elemet '0'-val inicializáljuk.

A=(3=>'f', 4=>'d', others=>'0');

Főleg akkor hasznos a megnevezett hozzárendelés alkalmazása, ha a vektor elemeinek a számát generikusan határoztuk meg. Mindegy, hogy 4, 8 vagy 16 eleme van a vektornak, mert az *X =(others => '0')* mindenik elemét inicializálja '0'-val.

2. fejezet

VHDL szekvenciális kifejezések

Ebben a részben a VHDL-ben alkalmazható fontosabb szekvenciális kifejezéseket: a folyamatot, feltételes értékadást, ciklusokat, jeleket és változókat tekintjük át. Az olvasó képes lesz az ismeretek áttekintését és begyakorlását követően kombinációs és szekvenciális áramkörök VHDL-ben való modellezésére és szimulációjára.

2.1. VHDL utasítások

Egy modul VHDL-ben való megvalósítása során két külön részt különböztetünk meg:

- szekvenciális rész,
- konkurens programrész.

A szekvenciális kifejezésekkel a folyamat-, illetve alprogram részekben találkozunk. A szekvenciális programrészek abban a sorrendben vannak végrehajtva, ahogy megjelennek a *process* vagy az alprogram *function* részekben.

A konkurens részek párhuzamosan vannak kiértékelve. A konkurens rész az architecture által van képviselve, amely tartalmaz:

- folyamatot,
- konkurens eljáráshívást (procedure),
- konkurens jelhozzárendelést,
- komponenspéldányosítást.

Az egyes alegységek, modulok jelekkel vannak összekapcsolva.

2.1.1. Szekvenciális kifejezések

A fontosabb szekvenciálisan végrehajtott utasítások, amelyeket gyakrabban alkalmaznak a VHDL alapú tervezés során: *PROCESS FUNCTION*, *PROCEDURES*, *IF*, *WAIT*, *CASE LOOP*.

A szekvenciális kódrészben lehetőség van akár változóknak, akár jeleknek a használatára. A jelek (signal) globálisan, míg a változók (variable) lokálisan az adott *process*, *function* vagy *procedures* részekben alkalmazhatóak.

A szekvenciális kifejezések a következő osztályokba sorolhatóak:

- Szekvenciális értékadás: a szekvenciális programrészen belül a változókhoz és jelekhez való érték-hozzárendelés
- Folyamatirányító kifejezések: (utasítások)
  - *IF*, *CASE* – feltételes végrehajtás
  - *LOOP FOR*, *LOOP WHILE*, *LOOP UNTIL* – ciklikus végrehajtás
  - *NEXT*, *EXIT* – ugrás
  - *WAIT* – folyamat felfüggesztésére szolgál és esemény bekövetkezésére vár a folyamat
  - *NULL* – nem történik vezérlés
- Alprogramrészek – ismételten végrehajtott algoritmusok (*FUNCTION*, *PROCEDURES*)

2.1.2. A folyamat szerkezete

A folyamatnak két külön szerepet betöltő része különíthető el: egy deklarációs és egy implementációs rész:

- a deklarációs rész a *PROCESS* és *BEGIN* között található kifejezések. A deklarációs részben deklaráljuk az implementáláshoz szükséges erőforrásokat: alprogramrészeket, típusokat, altípusokat, konstansokat, változókat, attribútumokat, use kifejezéseket;
- implementáció: a *BEGIN* és *END* közötti rész. Az implementációs részben szekvenciális kifejezéseket alkalmazva leírjuk az áramkör működését.

A folyamat szerkezete a következő programrészben van szemléltetve.

```
[cimke]:PROCESS (erzekenysegi lista)
   [tipus deklaralas]
   [konstans deklaralas]
   [valtoza deklaralas]
```

```
[VARIABLE valtozo_neve tipus [intervallum] [:=kezdeti_ertek]
   [alprogram deklaralas]
BEGIN
   ...
   -- szekvenciális kifejezések
   -- IF, WAIT, CASE, LOOP
   ...
END PROCESS [cimke];
```

A folyamat a legegyszerűbb megoldás szekvenciális áramkörök tervezésére. A folyamat a terv többi részével a folyamaton kívül deklarált jeleken és portokon kommunikál.

A folyamatra jellemző:

- szekvenciális kifejezés,
- minden folyamat egyetlen kifejezés,
- az architecture-ben minden folyamat egyszerre (konkurensen) hajtódik végre.

A *PROCESS* kifejezés egy szekvenciális algoritmust implementál. A folyamatban minden sor szekvenciális végrehajtást feltételez.

Egy folyamat akkor aktiválódik, ha a *PROCESS* kifejezést követő érzékenységi listából vagy a *WAIT* kifejezésből a jelek értéket váltanak.

A *PROCESS* implementációs részben a folyamatirányító kifejezések (utasítások) meghatározzák, hogy a jelekre mely értékadás fog végrehajtódni. A *PROCESS* utasítás alkalmazható élesítő jelekkel vagy élesítő jelek nélkül. Az élesítő jeleket alkalmazó folyamat akkor aktiválódik, amikor az élesítő listából bármely jel állapotot vált. Az élesítő listát tartalmazó folyamat nem tartalmazhat az implementációs részben WAIT kifejezést. A folyamatban az utolsó utasítást követően történik meg az értékadás. Az élesítőlista-nélküli folyamatnak tartalmaznia kell legalább egy *WAIT* kifejezést. Egyes tervezőeszközök esetében a *WAIT* kifejezés rögtön a *BEGIN* után kell következzen. A WAIT kifejezést követik azok a jelek, amelyek változását figyeljük. A *WAIT* felfüggeszti a folyamatot és a *WAIT*-ben megadott esemény bekövetkezésére vár. Egy élesítési lista nélküli folyamat a következő *WAIT* kifejezésig fut. Egyes tervezőeszközök folyamatonként több *WAIT* kifejezés használatát is támogatják.

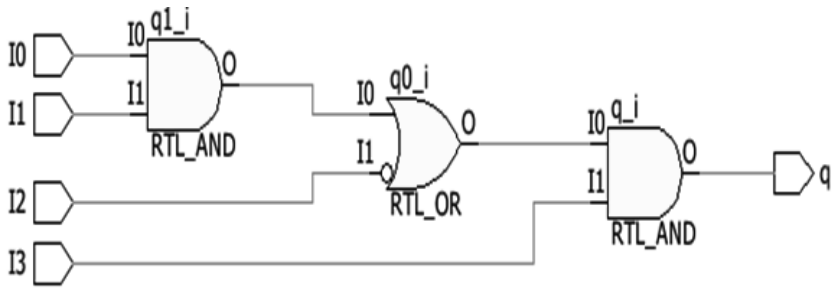
A már egyszer aktiválódott folyamat kiértékelése a legutolsó felfüggesztéstől kezdődik, a folyamat fentről lefele hajtódik végre. Ha a folyamat végéig nincs (még) egy *WAIT* (újabb *WAIT*), a kiértékelés visszaugrik a folyamat elejére és folytatódik a *WAIT* utasításig. A folyamatban alkalmazott jelek értékeire a folyamat indításakor aktuális jel értéke érvényes.

A folyamaton belül az összes jelértékadás csak lehetséges értékadás, a jelekre az utolsó értékadás érvényesül. A folyamat keretében a jel értékadása csak a folyamat kiértékelésének a végén válik véglegessé.

Kombinációs áramkört megvalósító folyamat

A következő VHDL kódrészlet a 2.1. ábrán szemléltetett logikai áramkört valósítja meg. A folyamat minden egyes alkalommal lefut, amikor a $I0$, $I1$, $I2$, $I3$ bemenetek közül bármelyik is változik. A példa helyes, viszont a példa szerint megvalósított folyamat kerülendő.

```
entity log_lut is
Port (
   I0 : in STD_LOGIC; --bemeneti port jel
   I1 : in STD_LOGIC; --bemeneti port jel
   I2 : in STD_LOGIC; --bemeneti port jel
   I3 : in STD_LOGIC; --bemeneti port jel
   q : out STD_LOGIC);  --kimeneti port jel
end log_lut;

architecture Behavioral of log_lut is
begin

process (I0, I1, I2, I3) --élesítő jelek
begin
   -- megvalósított logikai függvény
   q<=((I0 and I1) or (not I2)) and I3;
end process;
end Behavioral;
```



2.1. ábra. Kombinációs áramkört megvalósító process

Szekvenciális áramkört megvalósító folyamat

D tároló Reset jel nélkül

D típusú tároló VHDL-ben való megvalósítását a következő programrészben ismertetjük:

```
entity d_tarolo is
port (
   src_clk : in std_logic; --órajel bemenet
   d : in std_logic;  --adatbemenet
   q : out std_logic --d tároló kimenete
);
end d_tarolo;

architecture Behavioral of d_tarolo is
process (src_clk)  --a folyamat az órajelre élesül
begin
   Q <= '0'; -- kezdőérték hozzárendelése
   --lefutó óraél detektálása
   if src_clk'event and src_clk='0' then
      q <= d; -- felfutó óraélre a d jelnek a q kimenetre
            -- való kapcsolása
end if;
end process;
end Behavioral;
```

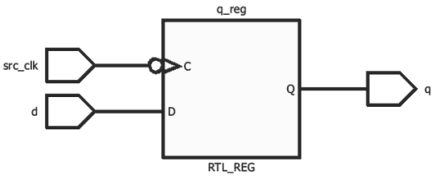
A tároló nem tartalmaz reset bemenetet. Minden egyes lefutó óraélre a d-adatvonalon található érték beíródik a tárolóba és elérhető a q kimeneten.

D tároló Reset jellel, szinkron reset

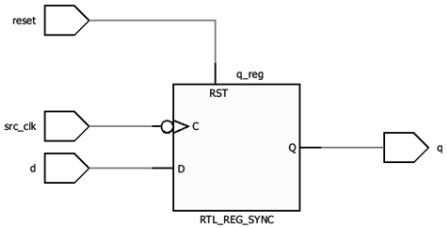
Szinkron resetet alkalmazó tároló esetében a tároló tartalma az *src_clk* órajellel szinkronban nullázódik.

```
entity d_tarolo is
port (
   src_clk : in std_logic;  --órajel bemenet
   reset : in std_logic;    --reset bemenet
   d : in std_logic;    --adatbemenet
   q : out std_logic    --kimenet
);
end d_tarolo;

process (src_clk, reset)
begin
```

```
   --felfutó óraél detektálása
   if src_clk'event and src_clk='0' then
      if reset='1' then --reset feltétel ellenőrzése
         q <= '0';  --d tároló inicializálása
      else
         q<=d;      --adat beírása a regiszterbe
      end if;
   end if;
end process;
end Behavioral
```



2.2. ábra. RTL szimbólum: D tároló reset jel nékül



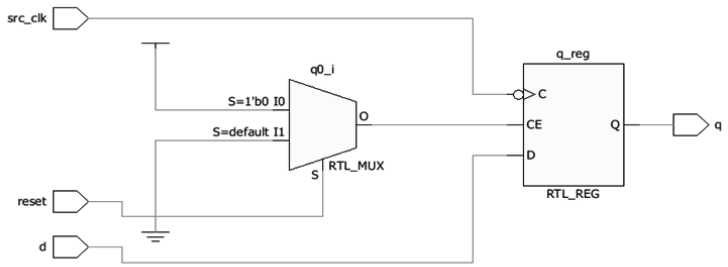
2.3. ábra. RTL szimbólum: D tároló reset jellel

D tároló Reset jellel, aszinkron reset

Aszinkron resetet alkalmazó tároló esetében az órajeltől függetlenül visszaállítható alaphelyzetébe a tároló [11].

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------
entity d_tarolo is
Port
   (src_clk: in std_logic;
   reset : in std_logic;
   d : in std_logic;
   q : out std_logic);
end d_tarolo;
-----------------------------------------
architecture Behavioral of d_tarolo is
begin
process (src_clk, reset) --élesítő lista
begin
   if reset='1' then --reset feltétel ellenőrzése
      q <= (others => '0');   -- kezdő értékadás
```

```
                        --felfutó óraél detektálása
   elsif src_clk'event and src_clk='0' then
      q <= d;           --adat beírása a regiszterbe
   end if;
end process;
end Behavioral;
```



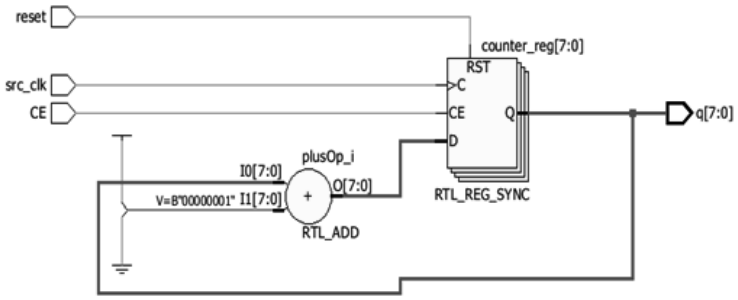
2.4. ábra. RTL kapcsolási rajz: D tároló szinkron reset jellel

A D-típusú tároló három különböző specifikálásának eredménye a 2.2., 2.3., 2.4. ábrákon van szemléltetve.

Az alábbi programrészben egy számláló van specifikálva. A *reset* jel logikai egyesre való kapcsolásakor a számláló visszaáll nullára. Az *src_clk* órajel minden egyes felfutó élére növeljük a számláló értékét (2.5. ábra).

```
entity szamlalo is
generic
 --generikus paraméter létrehozása és inicializálása
(BIT_SZAM : natural :=8)
port (
   src_clk : in std_logic; --órajel bemenet
   CE : in  std_logic; --órajel engedélyező
   reset : in  std_logic;  --reset bemenet
   --számláló kimenete
   q : out std_logic_vector((BIT_SZAM-1 downto 0)
);
end szamlalo;

architecture Behavioral of szamlalo is
process (src_clk, reset) --élesítő bemenetek
--számláló értékének tárolása változóban
variable counter : std_logic_vector(BIT_SZAM-1 downto 0);
begin
```

```
   if reset='1' then  --reset feltétel ellenőrzése
      counter := (others=>'0');  --kezdőérték beállítása
   -- felfutó óraél detektálása
   elsif src_clk'event and src_clk='1' and CE='1' then
      counter := counter+1; --számláló értékének
      -- inkrementálása
   end if;
   q<=counter; -- számláló értékét tartalmazó változó
            -- kivezetése a q kimenetre
end process;

end Behavioral;
```



2.5. ábra. RTL kapcsolási rajz: Szekvenciális áramkört megvalósító folyamat

2.1. táblázat. Jelek és változók rövid összehasonlítása

Jelek	Változók
SIGNAL – vezeték szerepe komponensek közötti összekapcsolásra A SIGNAL deklarálható: – PACKAGE-ben – ENTITY-ben (port típusú szignál) – ARCHITECTURE-ban Folyamaton belül a jel módosítása csak folyamat végén történik SIGNAL értékadás: $jel <= kifejezes$	VARIABLE (változó) csak folyamaton belül deklarálható, és tartalma folyamaton kívül nem elérhető Alkalmazható köztes eredmények tárolására. A VARIABLE módosítása azonnali. Változó értékadás $valtozo\_neve := kifejezes$

IF, ELSIF feltételes végrehajtás

A feltételes kifejezések ellenőrzik a többi szekvenciális utasítás végrehajtását. Az *IF* kifejezés tartalmaz legalább egy logikai feltételt az *IF* kulcsszót követően. A többi feltétel a következő *ELSIF* elágazásokban van meghatározva. A feltételek értékelése egyenként fentről lefele történik, amíg egyik feltétel igaz vagy nincs további feltétel. Ha egy elágazásban a feltétel igaz, akkor az adott elágazásban megtörténik a kifejezések sorozatának végrehajtása. Ha egyik feltétel sem teljesül, akkor a vezérlés az *IF* utasítás után a következő utasítással folytatódik.

Feltételes elágazás megvalósítható IF ELSIF vagy CASE WHEN kifejezésekkel [12].

```
IF feltetel THEN
   -- értékadás ha a feltétel igaz
ELSE
   -- értékadás ha a feltétel hamis
END IF;

IF (x<y) THEN temp:="11111111";
ELSIF (x=y and w='0') THEN
   temp:="11110000";
ELSE
   temp:=(OTHERS=>'0');
END IF;
```

Példa számláló megvalósítására

```
LIBRARY ieee;
USE ieee.std_logic_vector;
-----------------------------------------------
ENTITY counter IS
PORT (
   clk: IN STD_LOGIC;
   digit: OUT INTEGER RANGE 0 TO 9);
END counter
-----------------------------------------------
ARCHITECTURE counter OF counter IS
BEGIN
   count:PROCESS(clk)  --clk órajelre élesül a folyamat
   --változó számláló értékének tárolására
   VARIABLE temp : INTEGER RANGE 0 TO 10;
BEGIN
   --felfutó óraél detektálása
   IF (clk'EVENT AND clk='1') THEN
      temp:=temp+1; --számláló inkrementálása
```

```
      IF (temp=10) THEN --ha a számláló elérte a 10-et
         temp:=0;     -- értékének visszaállítása nullára
      END IF;
   END IF;
   digit <=temp;  --számláló értékének kivezetése a
   -- kimeneti jelre
   END PROCESS count;
END COUNTER
```

CASE elágazás

Ha egy folyamatban több lehetséges elágazás van, akkor célszerű a *CASE* kifejezést alkalmazni [3], [12]. Fontos megemlíteni, hogy a *CASE* struktúra csak folyamatokban használható. Az egyik leggyakoribb alkalmazása a *CASE WHEN* utasításnak multiplexer áramkörök specifikálása.

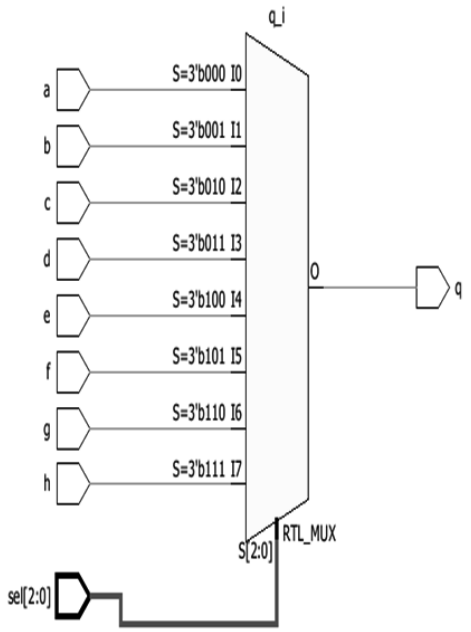
```
[ cimke: ] case kifejezés is
   when K1 => szekvenciális kifejezések;
   when K2 => szekvenciális kifejezések;
   when others => szekvenciális kifejezések;
end case [ cimke ];
```

Multiplexer áramkör megvalósítása

```
--sel a multiplexer áramkör szelekciós bemenete
--a,b,...,h a multiplexer bemenetei
--q multiplexer kimenete
process (sel)
begin
   case (sel) is
      when "000" =>
         q<=a;
      when "001" =>
         q<=b;
      when "010" =>
         q<=c;
      when "011" =>
         q<=d;
      when "100" =>
         q<=e;
      when "101" =>
         q<=f;
      when "110" =>
         q<=g;
      when "111" =>
         q<=h;
```

```
      when others =>
         q<='X';
   end case;
end process;
end Behavioral;
```

A CASE kifejezéssel specifikált multiplexer áramkör jelei a 2.6. ábrán tekinthetők át.



2.6. ábra. RTL kapcsolási rajz: multiplexer áramkör CASE utasítással való megvalósítása

2.1.3. A WAIT kifejezés

Ha a folyamaton belül alkalmazzuk a WAIT parancsot, a folyamatnak nem lehet érzékenységi paramétere. Élesítő lista nélküli folyamat esetében:

- A program végrehajtódik a következő *WAIT*-ig.
- A program végrehajtása felfüggesztődik, amíg a *WAIT* feltétele teljesül.

A WAIT kifejezés lehetséges alkalmazási formái [12]:

– *WAIT UNTIL* signal_condition;
– *WAIT ON* signal1 [, signal2, ...] ;
– *WAIT FOR* time;

WAIT UNTIL

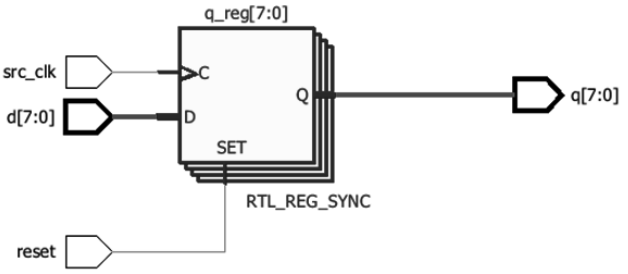
Mivel a folyamatnak nincs érzékenységi listája, a *WAIT UNTIL* parancs kell az első kifejezés legyen a folyamaton belül. A folyamat minden alkalommal élesül, végrehajtódik, amikor a feltétel teljesül.

```
PROCESS  --nincs érzékenységi lista!
--felfutó él detektálása a wait until kifejezéssel
   WAIT UNTIL (clk'EVENT AND clk='1');
   IF (rst='1') THEN --szinkron reset
      output <='00000000';
   ELSIF (clk'EVENT AND clk='1') THEN
      output <=input; --felfutó élre az adat beírása a
      -- regiszterbe és rávezetése a kimenetre
   END IF;
END PROCESS;
```

Regiszter megvalósítása *WAIT UNTIL* kifejezéssel

```
entity pelda_8d is
   --generikus változó deklarálása
GENERIC (BIT_SZAM : natural :=8);
--port jelek deklarálása
Port (
   src_clk : in std_logic; -- bemeneti órajel
   reset : in std_logic;   -- bemeneti reset jel
   d : in std_logic_vector(BIT_SZAM-1 downto 0); --bemeneti
       adatsín
   q : out std_logic_vector(BIT_SZAM-1 downto 0) --kimenet
);
end pelda_8d;

architecture Behavioral of pelda_8d is

begin
PROCESS  --nincs élesítő jel
begin
   -- felfutó óraél detektálása
   WAIT UNTIL src_clk'EVENT AND src_clk='1';
   IF (reset='1') THEN --reset feltétel ellenőrzése
      q <=(others=>'1'); --kezdőérték egyesre állítása
   ELSE
```

```
      q <=d;  --adat beírása a regiszterbe
   END IF;
END PROCESS;
end Behavioral;
```



2.7. ábra. RTL kapcsolási rajz: regiszter WAIT UNTIL kifejezéssel megvalósítva

WAIT ON

A *WAIT ON* utasításon várakozik a folyamat mindaddig, amíg valamelyik jel változik. A folyamatnak ebben az esetben nincs érzékenységi listája.

```
PROCESS
BEGIN
-- élesítő jelek meghatározása a wait on kifejezést követően
   WAIT ON clk, rst
   IF (rst='1') THEN --reset feltétel ellenőrzése
      output <="00000000"; --kezdőérték beállítása
   -- felfutó óraél detektálása
   ELSIF (clk'EVENT AND clk='1') THEN
      output <=input; --adat beírása a regiszterbe
   END IF;
END PROCESS
```

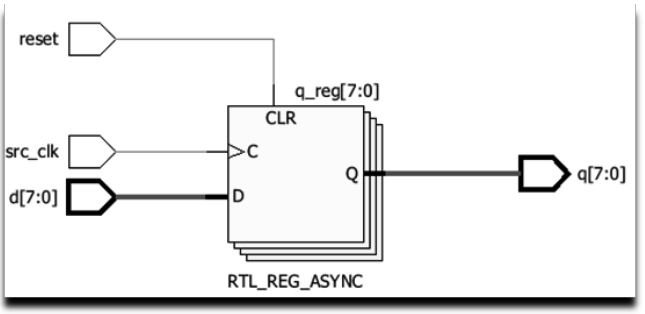
A *CASE* és *WAIT* kifejezések alkalmazása regiszter megvalósítására:

```
entity pelda_8 is
   -- generikus paraméter deklarálása és inicializálása
GENERIC (BIT_SZAM : natural :=8);
-- port jelek deklarálása
Port (
```

```
   src_clk : in std_logic;  -- órajel bemenet
   reset : in std_logic;    -- reset bemenet
   d : in std_logic_vector(BIT_SZAM-1 downto 0);
       --adatbemenet
   q : out std_logic_vector(BIT_SZAM-1 downto 0) -- q kimenet
);
end pelda_8;

architecture Behavioral of pelda_8 is
begin
PROCESS
BEGIN
   WAIT ON src_clk, reset;  --élesítő jelek meghatározása
   IF (reset='1') THEN  --reset feltétel ellenőrzése
      q <="00000000"; --kezdőérték beállítása
   --felfutó óraél detektálása
   ELSIF (src_clk'EVENT AND src_clk='1') THEN
      q <=d; -- bemeneti adat beírása a regiszterbe
   END IF;
END PROCESS;
END BEhavioral;

entity pelda_8c is
-- generikus paraméter deklarálása és inicializálása
GENERIC (BIT_SZAM : natural :=8);
-- port jelek deklarálása
Port (
   src_clk : in std_logic;  -- órajel bemenet
   reset : in std_logic;   -- reset bemenet
   d : in std_logic_vector(BIT_SZAM-1 downto 0); -- bemeneti
       adatsín
   q : out std_logic_vector(BIT_SZAM-1 downto 0) -- kimenet
);
end pelda_8c;

architecture Behavioral of pelda_8c is
begin
PROCESS
BEGIN
   WAIT ON src_clk, reset;  -- élesítő jelek
   CASE reset IS  -- elágazás a reset jel alapján
      WHEN '1' => q<=(others=>'0');  -- reset esetén
          kezdőértékadás
      WHEN '0' =>
         -- ha nincs reset felfutó él detektálása
         IF (src_clk'EVENT AND src_clk='1') THEN
            q<=d; -- bementi adat beírása a regiszterbe
```

```
         END IF;
      WHEN OTHERS => NULL;
   END CASE;
END PROCESS;
END Behavioral;
```

Mindkét példában megadott programrészlet ugyanazt az eredményt szolgáltatja (2.8. ábra).



2.8. ábra. RTL kapcsolási rajz: regiszter megvalósítása CASE és WAIT kifejezésekkel

WAIT FOR

A *WAIT FOR* parancsot csak szimulációra alkalmazzuk. Várakozik, hogy elteljen a *WAIT FOR* után meghatározott idő, majd a folyamat tovább fut a következő *WAIT* kifejezésig.

```
wait for idő;
```

Az alábbi programrészben szimulációban alkalmazott órajel generálása van megvalósítva wait for kifejezések alkalmazásával. Szekvenciális áramkörök szimulációjához az órajel a mintaprogrammal hozható létre.

```
 -- Órajel generálása
<clock>_process :process
begin
   <clock> <= '0';  -- órajel nullásra állítása
wait for <clock>_period/2;  -- fél órajelperiódus  bevárása
   <clock> <= '1';  -- órajel egyesre kapcsolása
wait for <clock>_period/2; -- fél órajelperiódus  bevárása
end process;
```

A bemeneti jelek megadása az idő függvényében:

```
-- Szimulációs folyamat
stim_proc: process
begin
-- reset jel 100 ns-ig egyesbe való tartása.
   reset='1';
wait for 100 ns;
   reset='0';
wait for 10 ns;  -- várakozás 10 ns-ig
   start='1';   -- start jel logikai egyesre kapcsolása és a
       szimulálandó modul indítása
-- egyéb bemenetek meghatározása
wait;
end process
```

2.1.4. FOR LOOP, WHILE LOOP

VHDL-ben a ciklusok létrehozására két lehetőség van:

– *FOR / LOOP* – véges ciklusszám,

– *WHILE / LOOP* – a ciklus kezdetekor előre nem ismert ciklusszám.

A *NEXT* és *EXIT* kifejezésekkel lehetőség van a ciklus kezdetére lépni, vagy kiugrani a ciklusból [3].

A *NEXT* és *EXIT* lehetséges alkalmazási formái:

```
next;  -- következő ciklus elejére
next cimke; --ugrás a címkére
next when A>B;  --ugrás a címkére, ha a feltétel igaz
Next cimke when C=D or flag; -- ugrás a címkére C=D és  flag
-- igaz, ahol a flag egy Boole típusú változó
exit kifejezés
exit;
exit cimke;
exit when A>B;
exit cimke when C=D or done;
-- done -boole valtozo
```

Minta programrész a *FOR LOOP* és a *WHILE LOOP* szemléltetésére:

```
[cimke:] FOR azonosito IN intervallum LOOP
   (szekvencialis kifejezes)
END LOOP [cimke];

[cimke:] WHILE feltetel LOOP
   (szekvencialis kifejezes)
END LOOP [cimke];
```

```
[cimke:] EXIT [cimke_b] [WHEN feltetel];
[cimke:] NEXT [cimke_b] [WHEN feltetel];
```

```
FOR i IN 0 TO 5 LOOP
-- az x tömb elemeihez hozzárendeli a W(i+2)  tömb elem és
-- az enable jel közötti ÉS függvény eredményét
x(i)<=enable AND W(i+2);
-- az y kétdimenziós tömb 0-ik sorát feltölti a w tömb
-- elemeivel
y(0,i)<=w(i);
END LOOP;
```

```
-- amíg az i változó kisebb mint 10 figyeli a felfutó óraélet
WHILE (i<10) LOOP
   WAIT UNTIL clk'EVENT AND clk='1';
   -- other statements
END LOOP;
```

```
-- az alábbi programrész meghatározza, hogy a data jel hány
    bitje nulla
FOR I IN data'RANGE LOOP
   CASE data(i) IS
      WHEN '0' => count:=count+1;
      WHEN OTHERS =>EXIT;
   END CASE;
END LOOP;
```

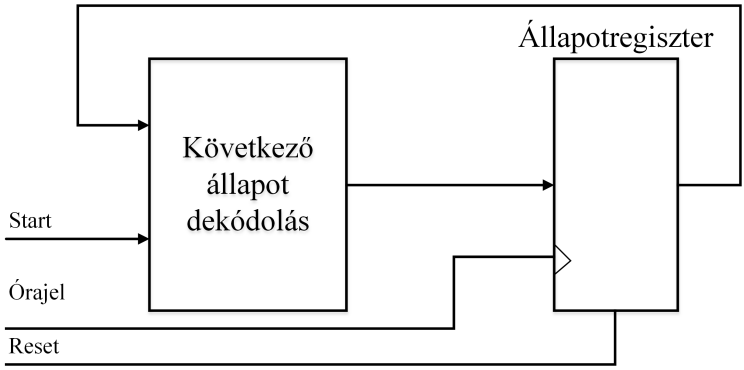
```
FOR I IN 0 TO 7  LOOP
   NEXT WHEN i=skip;
   ...
END LOOP;
```

Véges állapotú állapotgép VHDL alapú megvalósítása

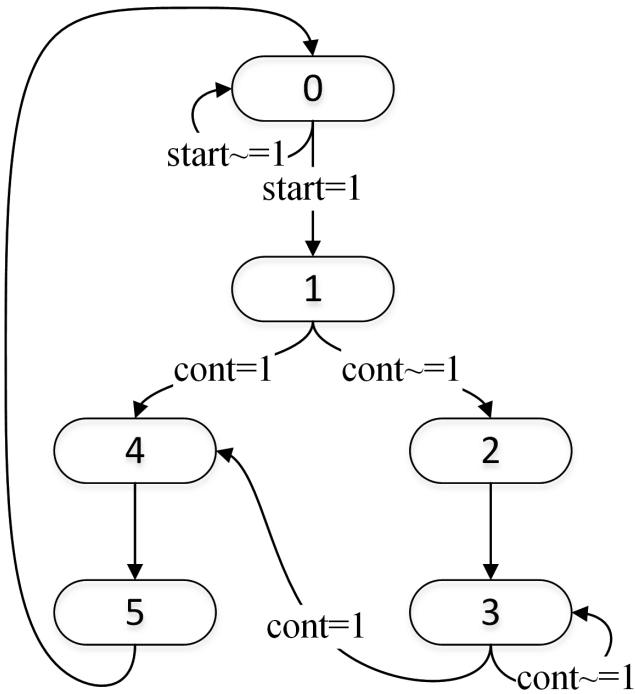
A véges állapotautomata az alapja a digitális rendszereknek, a különböző alegységek szinkronizálására, ütemezésére alkalmazható. A véges állapotautomata tartalmaz egy állapotregisztert, következő állapot meghatározására szolgáló logikát és kimeneti logikát.

Egy véges állapotautomata tömbvázlatát a 2.9. ábrán van szemléltetjük.

Következő állapotlogikát megvalósító programrészlet, amely *CASE WHEN* kifejezésekre épül. Az aktuális állapot (*AKT_ALL*) és a külső bemenetek függvényében meghatározza, hogy mely állapotba fog átlépni a rendszer. A VHDL kódrészletnek megfelelő állapotgráfot a 2.10. ábra szemlélteti.



2.9. ábra. Véges állapotautomata tömbvázlata



2.10. ábra. Állapotgráf példa

```
-- példa állapotautomata következő állapotlogikájának
    meghatározására
process (AKT_ALL_reg,start,cont)  -- élesítő jelek
    felsorolása
Begin
case (AKT_ALL) is  --az AKT_ALL jel alapján a megfelelő ág
    kiválasztása és végrehajtása
-- ha start='1' a következő állapot 1, egyébként 0
when 0 =>   if start = '1' then
   KOV_ALL <= 1;
else
   KOV_ALL <= 0;
end if;

when 1 =>
   if cont ='1' then
      --ha cont='1' a következő állapot 4, egyébként 2
      KOV_ALL <= 4;
   else
      KOV_ALL <= 2;
end if;

when 2 =>
   KOV_ALL <= 3;  --feltétel nélkül a következő állapot 3

when 3 =>
   if cont = '1' then
      --ha cont='1' a következő állapot 4, egyébként 3
      KOV_ALL <= 4;
   else
      KOV_ALL<=3; -- feltétel nélkül a következő állapot 3
end if;

when 4 =>
   KOV_ALL<= 5;  --feltétel nélkül a következő állapot 5

when 5 =>
   KOV_ALL<= 0;   --feltétel nélkül vissza 0-ás állapotba

when others =>
   KOV_ALL <= 0;   --bármely más esetben vissza a 0-ás á
       llapotba.
end case;
end process;
```

Az állapotregiszter implementálása megegyezik a D-tárolók kialakításával. Az állapotregiszter a *clk* órajelre van szinkronizálva, az órajel felfutó élére valósul meg a következő állapotba való átlépés.

```
-- példa állapotautomata állapotregiszterének meghatározására
process (clk,reset) -- élesítő jelek meghatározása
begin
   If reset='1' then
      AKT_ALL<=0;  -- reset jelre  kezdőállapot beállítása
   elsif clk'event and clk='1' then
   -- felfutó óraélre a következő állapot beírása az
   -- állapotregiszterbe, vagyis átlépés a következő
   -- állapotba
      AKT_ALL <= KOV_ALL;
   end if;
end process;
```

3. fejezet

VHDL konkurens kifejezések

Ebben a fejezetben az olvasó megismerheti a konkurens VHDL utasításokat. Összehasonlítva a szekvenciális utasításokkal, több áramkör szekvenciális és konkurens utasítással is megvalósítható. Példaprogramok tanulmányozása során elsajátítható a fontosabb utasítások működése és különböző gyakorlati alkalmazása. A generikus (GENERIC) paraméterek, valamint a FOR GENERATE és IF GENERATE utasítások alkalmazásával lehetőség van bonyolult áramkörök egyszerű módon való tervezésére.

3.1. Jelértékadás

A jelek értékadására több lehetőség is van. A 2. fejezetben említettük a folyamat keretében a jeleknek való értékadását. Ebben a részben, kibővítve, újabb lehetséges jel értékadási lehetőségeket mutatunk be. A jelek értékadása lehet:

- azonnali értékadás,
- feltételes jelértékadás,
- kiválasztott jelhozzárendelés.

Az azonnali értékadáskor a jobb oldali kifejezés kiértékelésének eredményét felveszi a bal oldali részben megnevezett jel.

```
x1 <= a and b;
```

Feltételes jelértékadás akkor következik be, amikor a jobb oldali kifejezésben megadott feltétel teljesül.

```
jel <= [kifejezes_1 when feltetel   else ...] kifejezes_2;
```

A szemléltetett példa alapján, ha a *feltetel* igaz, abban az esetben a *jel* felveszi a *kifejezes_1* kiértékelésének eredményét. Ellenkező esetben a jelre a *kifejezes_2* eredménye kerül.

A kiválasztott jelhozzárendelés alkalmazásával több kifejezés közül választhatunk a szelekciós bemenet függvényében.

```
with szelekcios_bemenet select
jel<= kifejezes_1 when választási_lehetőségek_1,
...
kifejezes_n when választási_lehetőségek_n,
[kifejezes when others];
```

A *WITH SELECT* alkalmazásakor a választási lehetőségek az összes lehetőséget le kell fedjék, ha nem, akkor kell alkalmazni az *others* kulcsszót [3]. A *WHEN* utasítást követően több lehetőség is van a szelekciós bemenet lehetséges értékeinek, értékkombinációinak a meghatározására.

```
WHEN value
WHEN value1 TO value2
WHEN value1 | value2 | ...
```

A következő mintaprogram szerint a folyamatok párhuzamosan hajtódnak végre.

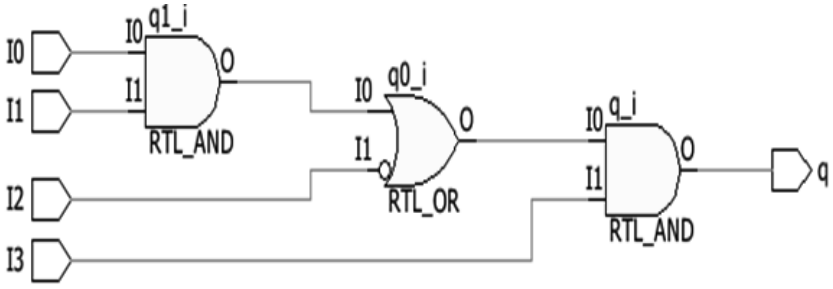
```
entity pelda is
Port (
   I0 : in STD_LOGIC; --bemeneti jel
   I1 : in STD_LOGIC; --bemeneti jel
   I2 : in STD_LOGIC; --bemeneti jel
   I3 : in STD_LOGIC; --bemeneti jel
   q : out STD_LOGIC); --kimeneti jel
end pelda;

architecture Behavioral of pelda is
Signal s0, s1 : std_logic;
Begin
RTL_AND_1: process(I0, I1) --élesítő jelek I0, I1
   Begin
      S0<= I0 and I1;   -- részeredmény az S0 jelre
   end process;

RTL_OR: process(S0, I2) --élesítő jelek S0, I2
   Begin
      S1<= S0 or (not I2); -- részeredmény az S1 jelre
   end process;

   RTL_AND_2: process(S1, I3) --élesítő jelek S1,I3
```

```
   Begin
      q<= S1 and  (not I2);  -eredmény a q kimenetre
   end process;
End Behavioral;
```

A folyamatra összefoglaljuk a fontosabb ismereteket, amelyekre figyelni kell a tervezés során. A folyamat:

- Szekvenciális kifejezéseket tartalmaz.
- Nem tartalmazhat konkurens kifejezéseket.
- Az architektúrában meghatároz egy régiót, amelyben az utasítások szekvenciálisan vannak végrehajtva.
- Lehetővé teszi a funkcionális leírást.
- Az architecture részen belül található folyamatok párhuzamosan hajtódnak végre.

A 3.1. ábrán szemléltetett áramkört eredményező példaprogramban minden egyes logikai kapu egy-egy folyamaton belül van megvalósítva.

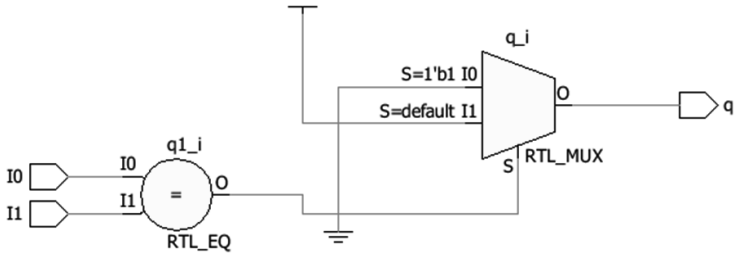


3.1. ábra. Példa folyamatok konkurens végrehajtására

A kizáró vagy logikai függvényre két megoldást szemléltetünk, amely ugyanazt a végleges megoldást eredményezi. Egy entitásra két megoldás látható: az $ar1\_xor2$ és $ar2\_xor2$ (3.2. ábra).

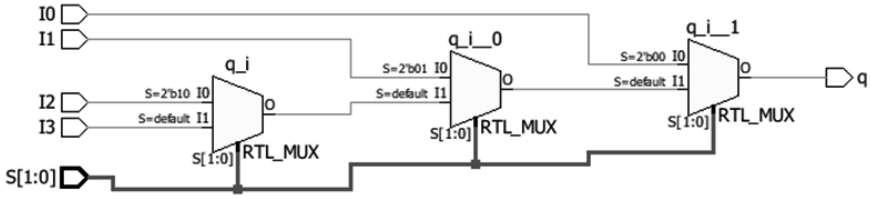
```
entity xor2 is
port (
   I0, I1: in std_logic;
   q: out std_logic);
end xor2;

architecture ar1_xor2 of xor2 is
begin
process (I0, I1) -- élesítő jelek I0, I1
begin
```

```
   if I0 = I1 then
      q <= '0';  -- ha a feltétel igaz
   else
      q <= '1';  -- ha a feltétel hamis
   end if;
end process;
end ar1_xor2;

architecture ar2_xor2 of xor2 is
begin
   -- feltételes értékadás
   q <= '0' when I0 = I1 else '1';
end ar2_xor2;
```



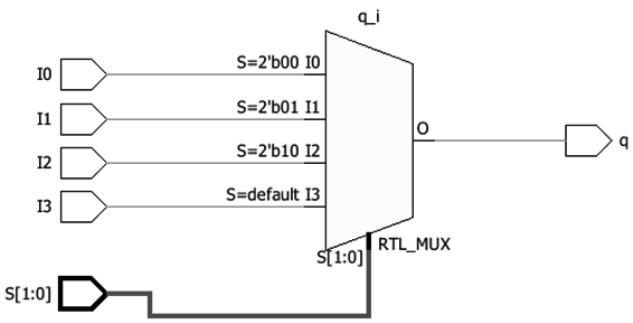
3.2. ábra. XOR logikai függvény RTL áramköri megvalósítása

A *WHEN ELSE* (3.3. ábra), illetve *WITH SELECT* (3.4. ábra) kifejezések vannak összehasonlítva egy 4 adatbemenettel rendelkező multiplexer áramkör megvalósításakor. A két megoldás az alábbi programrészben van szemléltetve [3]:



3.3. ábra. RTL kapcsolási rajz: multiplexer áramkör *WHEN ELSE* megvalósítással

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-- multiplexer áramkör
entity mux_2 is
Port (   S : in STD_LOGIC_vector(1 downto 0);
   I0, I1, I2, I3 : in STD_LOGIC; --bemeneti jelek
   q : out STD_LOGIC); -- kimeneti jel
end mux_2;

architecture ar1_mux_2 of mux_2 is
begin
--mux megvalósítása when else kifejezéssel
   q<=   I0 WHEN S="00" ELSE  -- I0 ha S= "00" egyébként
      I1 WHEN S="01" ELSE  -- I1 ha S= "01" egyébként
      I2 WHEN S="10" ELSE  -- I2 ha S= "10" egyébként
      I3;                  --I3
end Behavioral;

architecture ar2_mux_2 of mux_2 is

begin
--mux megvalósítása with select when kifejezéssel
--I0, I1, I2, I3 --adatbemenetek
with S select   --S -kiválasztó bemenet
   q<=   I0 WHEN "00",
      I1 WHEN "01",
      I2 WHEN "10",
      I3 when others;
end ar2_mux_2;
```



3.4. ábra. RTL kapcsolási rajz: multiplexer áramkör *WITH SELECT* megvalósítással

Összehasonlítva a két megoldást, a WHEN ELSE alapú megvalósítás két adatbemenetű multiplexer áramkörök kaszkádba való kapcsolásával alakítja ki az áramkört. A *WITH SELECT*-tel való implementáció egy 4 adatbemenetű multiplexer áramkört generál. A kaszkád alapú megvalósítás az egyes bemenetekre különböző késleltetést eredményez.

A *CONFIGURATION* parancs segítségével meghatározható, melyik megvalósítást fogjuk alkalmazni.

A *BLOCK* kifejezést az *ARCHITECTURE* részben található konkurens utasítások csoportosítására alkalmazzák. A *BLOCK* használatának két fő célja van:

- egyszerűsíteni a tervezett modul átláthatóságát,
- a guarded kifejezéssel bizonyos jelek feltételes alkalmazását biztosítani.

A *BLOCK* utasításnak csak szervezési jellegű szerepe van, használata nem befolyásolja közvetlenül a szimulációs modell végrehajtását.

A *PORT MAP* és a *GENERIC MAP* utasítások célja a blokkon kívül deklarált jeleknek és generikus változóknak a blokkok belsejébe deklarált portokra és általános paraméterekre való illesztése. A *BLOCK* utasítás csak minimális gyakorlati jelentőséggel bír, szimulációra általában hatékonyabb a folyamat alkalmazása. A *BLOCK* utasítást csak kritikus feladatok specifikálására javasolt alkalmazni.

BLOCK kifejezés:

- Nagyrészt azoknak a kifejezéseknek a használata, amelyek alkalmazhatóak az architecture részben
- Port és generikus változók deklarálása (*PORT MAP* és *GENERIC MAP*)
- Típus és altípus deklaráció
- Alprogramrészek deklarálása + megvalósítása
- Konstans, változó és jeldeklarálás
- Komponensdeklarálás
- File, attribútum és konfiguráció deklarálása

A következő VHDL programrészekben a *BLOCK* kifejezés alkalmazása van szemléltetve.

```
cimke: BLOCK
   -- [deklaracios resz]
BEGIN
   -- konkurens kifejezés
END BLOCK cimke;
```

A *BLOCK* utasítások egymásba ágyazhatóak, amint az alábbi példa szemlélteti [11].

```
cimke_1: BLOCK
   [deklaracios resz]
BEGIN
 -- [felső szintű blokk konkurens kifejezései]
   címke_2: BLOCK
   -- deklarációs rész alsó szintű blokk
   begin
   -- konkurens kifejezések;
   END BLOCK címke_2;
-- [felső szintű blokk konkurens utasítások]
END BLOCK címke_1;
```

Egy architektúrában több blokk alkalmazása is lehetséges az alábbi minta szerint

```
ARCHITECTURE pelda
BEGIN
   ...
   c_block1: BLOCK
      -- [deklarációs rész]
   BEGIN
      --  konkurens kifejezések
   END BLOCK c_block1

   c_block2: BLOCK
      --  [deklarációs rész]
   BEGIN
      --  konkurens kifejezések
END BLOCK c_block2
   ....
END pelda
```

Ha egy opcionális vezérlő feltétel van alkalmazva a *BLOCK* kifejezésben, abban az esetben úgymond „vezérelt” *BLOCK* típusú lesz. A vezérlő feltétel visszatérít egy logikai értéket, amely ellenőrzi a block keretében a vezérelt jeleket.

Ha a vezérlő feltétel hamisnak bizonyul, a *BLOCK*-on belül a vezérelt jelekre kikapcsolja a jelvezérlőt (*drivert*). Deklarálásakor a vezérelt jelek megjelölhetőek a jel neve után következő bus vagy regiszter kifejezések alkalmazásával.

A *bus* és a *register* típusú jelek közötti különbség, hogy ha egy *bus* típusú jelre az összes vezérlő ki van kapcsolva, szükség van egy feloldó funkcióra, amely értéket szolgáltat a jelnek. A *register* típusú jelek esetében a jel megtartja az utolsó értékét a vezérlők kikapcsolásakor.

A vezérelt *BLOCK*-ot abban az esetben alkalmazzák, ha a konkures kódrészben minimális szekvenciális viselkedést írnak le.

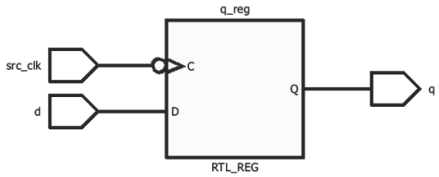
Az alábbi programrészek egy Latch és D-tároló block utasítással való megvalósítását szemléltetik [11]: Ha az órajel clk='1' értékre vált, vagyis a vezérlőfeltétel igaz, a q vezérelt jel felveszi a d jel értékét.

```
LIBRARY ieee
USE ieee.std_logic_1164.all;
ENTITY latch IS
PORT
   (clk: IN STD_LOGIC; -- bemeneti órajel
   d : IN STD_LOGIC;   -- adatbemenet
   q: OUT STD_LOGIC);   -- d-tároló kimenete
END latch;
-------------------------------------------------
ARCHITECTURE latch OF latch IS
BEGIN
b1: BLOCK (clk='1') -- vezérlő feltétel: clk='1'
BEGIN
   q<=GUARDED d; -- ha a vezérlő feltétel igaz, a d
   -- megvezérli a q jelet
   -- ha a vezérlő feltétel hamis, a d vezérlője ki van
       kapcsolva
END BLOCK b1;
END latch;
```

D típusú bistabil áramkör. Felfutó óraélre *src_clk'event and src_clk='1'* teljesül a vezérlőfeltétel, és a q felveszi a d értékét.

```
entity pelda_2_guarded is
Port (
   src_clk: in std_logic;
   d : in std_logic;
   q : out std_logic);
end pelda_2_guarded;

architecture Behavioral of pelda_2_guarded is
begin
b1: BLOCK (src_clk'event and src_clk='1') -- vezérlő
    feltétel, felfutó óraél
BEGIN
-- ha a vezérlő feltétel igaz, a d megvezérli a q jelet
```

```
-- ha a vezérlő feltétel hamis, a d vezérlője ki van
-- kapcsolva
q<=GUARDED d;
END BLOCK b1;
end Behavioral;
```



3.5. ábra. D tároló reset nélkül

D típusú tároló reset bemenettel. Az alábbi programrészben, ha a vezérlő feltétel teljesül, annak függvényében, hogy aktív-e a reset jel, a q felveszi a kezdeti '0' értéket vagy a d értékét.

```
LIBRARY ieee
USE ieee.std_logic_1164.all;
ENTITY dff IS
PORT
(d, clk, rst: IN STD_LOGIC;
q: OUT STD_LOGIC);
END dff;
--------------------------------------------------
ARCHITECTURE dff OF dff IS
BEGIN
   b1: BLOCK (clk'EVENT AND clk='1') --felfutó él detektálása
BEGIN
   q<=GUARDED  0 WHEN rst='1' ELSE d; --felfutó élre reset
   -- ellenőrzése, ha rst ='1',  tároló kezdőértékének
   -- inicializálása, egyébként a d adat beírása a tárolóba
   -- hamis vezérlőfeltételre a vezérlő kikapcsolása
END BLOCK b1;
END latch;
```

A Vivado nem tudja szintetizálni a guarded block kifejezéseket. A szekvenciális viselkedés védett blockként való megvalósítása helyett a szekvenciális kifejezéseket folyamatként javasolt megvalósítani.

Ötletek, amelyeket javasolt figyelembe venni a tervezés során:

- lehetővé tenni közös részek megosztását és újrahasznosítását;
- javasolt az alkalmazott elemeket könyvtárba (*LIBRARY*) helyezni;

- a tervből különböző részek megírhatóak:
  - komponensként (*COMPONENT*) integrálva,
  - függvényekben (*FUNCTIONS*),
  - eljárásokban (*PROCEDURES*).

3.1.1. FOR GENERATE, IF GENERATE

A *for generate* és *if generate* utasítások egy megoldást biztosítanak egy terv specifikálása során egyes részek iteratív vagy feltételes kialakítására [10]. A *for generate* utasítást többnyire alkatrészek többszöri példányosítására alkalmazzák. A *generate* utasításban szereplő paraméter használható az alkatrészek bemenetére kapcsolt mátrixszerű jelek indexelésére. A *generate* utasítások egyszerűsítik egy rendszer tervezési struktúrájának a leírását. Általában arra használják, hogy egy azonos összetevőkből álló alkatrész csoportot határozzon meg egy komponens specifikáció használatával, és ciklusszerűen ismételje az alkatrész példányosítását a generáló mechanizmus alkalmazásával. A *generate* kifejezés három fő részt tartalmaz:

- a generálási séma, amely lehet a ciklikus vagy feltételes változata az utasításnak,
- deklarációs rész, amely tartalmazhatja alprogramrészek, típusok, jelek, konstansok, komponensek, attribútumok, konfigurációk lokális deklarálását,
- konkurens kifejezéseket.

A generálási séma specifikálja, hogyan kell a konkurens szerkezetű kifejezéseket generálni, példányosítani.

Amint említettük, két generálási séma áll rendelkezésünkre: komponensek ciklikus elhelyezése a *for generate*, illetve feltételes elhelyezése az *if generate* utasításokkal. A for generate utasítás a szabályosan ismétlődő alegységek elhelyezésére szolgál. Ebben az esetben a *generate* utasításban megadott paraméter és értéktartománya hasonló módon jön létre, mint a szekvenciális ciklusok során. A szabályos kifejezések tartalmazhatnak szabálytalanságokat, mint például egy léptető regiszter esetében az első és utolsó elemre való jelek csatolása különbözik a közbenső bistabil áramköröktől. Míg a belső bistabilok szabályosan, ugyanúgy vannak láncszerűen összekapcsolva, addig az első elem kívülről kapja a bemeneti jeleket. Ezeknek a szabálytalanságoknak a megoldására szolgál az *if generate* kifejezés. Vagyis a *generate* utasítás lehetőséget biztosít komponensek több példányban való példányosítására. A parancsban paramétereként alkalmazott változó használható a vektor típusú signalok indexeléséhez.

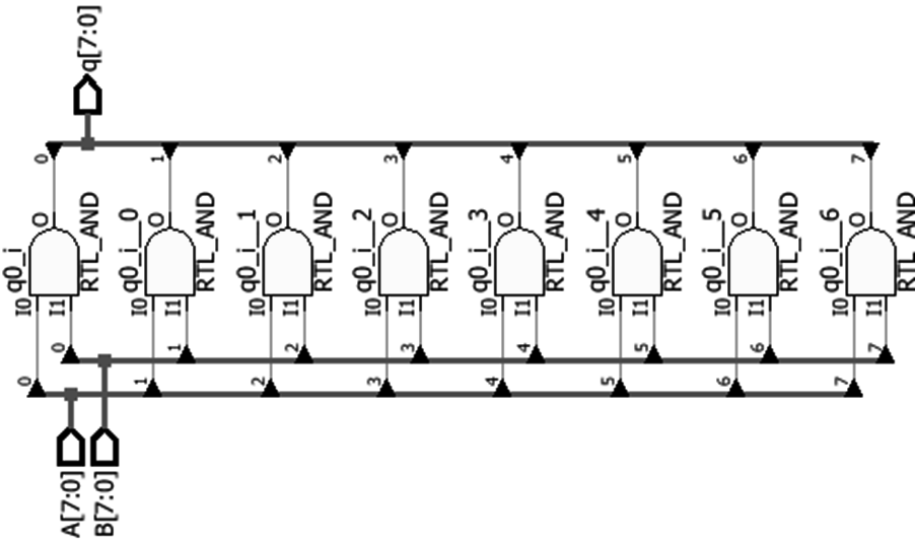
Generate kifejezés szerkezete az alkatrészek ciklikus példányosítására [11]:

```
Label1: FOR azonosító IN intervallum GENERATE
   (konkurens hozzárendelés)
END GENERATE;
```

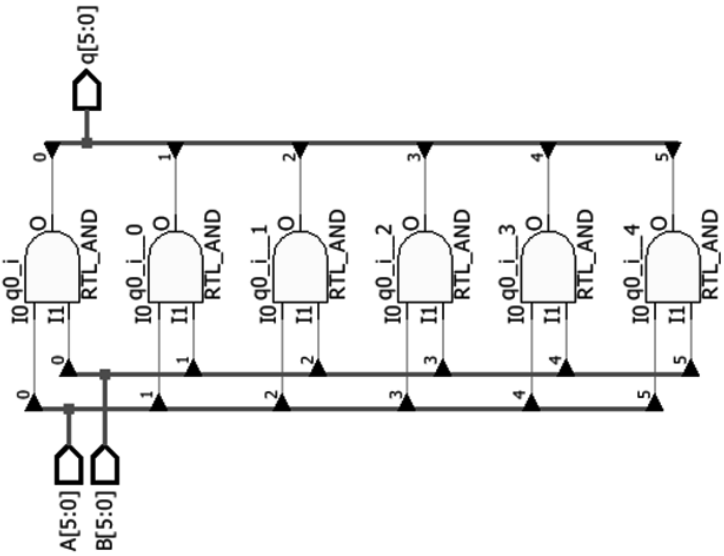
Az ***if generate*** kifejezés feltételes példányosításra [11]:

```
Label2: IF feltétel GENERATE
   (konkurens hozzárendelés)
END GENERATE;
```

A programrészekben a *for generate* és az *if generate* mintafeladatokban való alkalmazása van szemléltetve. Az ÉS kapu példányosítása és a bemeneti és kimeneti sínekre való csatolásának áramköri rajza a 3.6., 3.7. ábrán van szemléltetve. Ugyanazt a programot alkalmazva, csak megváltoztatva a generikus paraméter (N) értékét az egyik változatban 8, míg a másik változatban 6 ÉS kapu van példányosítva.



3.6. ábra. Generate alkalmazása 8 ÉS kaput tartalmazó logika kialakítására



3.7. ábra. Generate alkalmazása 6 ÉS kaput tartalmazó logika kialakítására

```
entity pelda_12_generate_and_kapu is
GENERIC ( N : natural :=8);  -- generikus paraméter
-- deklarálása
Port (
-- a bementi és kimeneti portjeleknek a generikus paraméter
-- alapján való meghatározása
   A: in STD_LOGIC_VECTOR(N-1 downto 0); -- bemeneti vektor
   B: in STD_LOGIC_VECTOR(N-1 downto 0); -- bemeneti vektor
   q : out STD_LOGIC_VECTOR(N-1 downto 0) -- kimeneti
       vektor);
end pelda_12_generate_and_kapu;

architecture Behavioral  of pelda_12_generate_and_kapu is
begi
--G1 címke, amelyel azonosítható a tervben a FOR GENERATE
-- kifejezéssel létrehozott áramkör
-- i ciklusváltozó az A bemenet sínszélessége alapján
G1: FOR i IN A'RANGE GENERATE
      q(i)<=A(i) and B(i);  -- minden egyes q(i) vektor
      -- elemét egy logikai ÉS függvénnyel származtatjuk az
      -- A(i), B(i) vektorelemek alapján
   END GENERATE;
end Behavioral;
```

Az alábbi programrészben az ÉS logika kialakítása a *WHEN* feltételes hozzárendeléssel van megvalósítva. Ugyancsak a 3.6. ábra szerinti megoldást eredményezi.

```
-- a FOR GENERATE kifejezés ciklusváltozója 0 -t=l 7-ig
OK: FOR i In 0 TO 7 GENERATE
-- az ÉS művelet helyett a WHEN ELSE kifejezés van alkalmazva
output(i)<='1' WHEN  (a(i) and b(i))='1' ELSE '0';
END GENERATE;
```

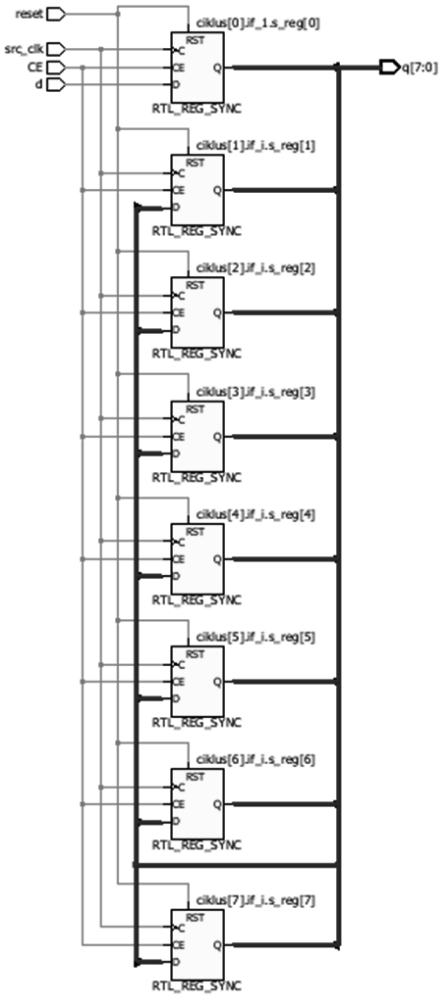
3.1.2. Regiszterlánc megvalósítása for generate és if generate utasítások alkalmazásával

A *for generate* a regiszterek többszöri példányosítására, míg az *if generate* az általános kapcsolástól való eltérések leírására szolgál. A *for generate* és *if generate* kifejezésekben egy-egy *process* kifejezést találunk a bistabil leírására. A bistabilok közötti kapcsolatok kialakítására egy jelvektor van alkalmazva. A ciklikus példányosítás során a jelek indexelésére az i ciklusváltozó van alkalmazva. A példányosítandó regisztereknek a száma egyszerűen változtatható az *RSZ* generikus változóval.

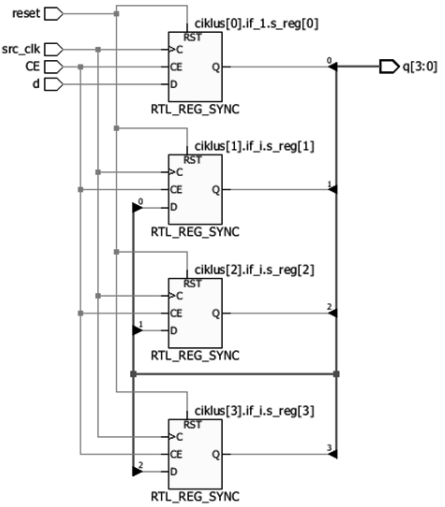
```
entity pelda_4_generate is
generic (RSZ : natural :=8);
Port (
   src_clk: in std_logic; -- bemeneti órajel
   reset : in std_logic;  -- reset
   CE : in std_logic;  --órajel engedélyező
   d : in std_logic;   -- bemeneti adat
   q : out std_logic_vector(RSZ-1 downto 0)); --regiszter
      lánc kimeneti vektora
end pelda_4_generate;

architecture Behavioral of pelda_4_generate is
signal s: std_logic_vector(RSZ-1 downto 0);
Begin
-- for generate  kifejezéssel az RSZ generikus paraméter
-- alapján többször példányosítjuk a folyamatban
-- megvalósított d tárolót
ciklus : for i in 0 to RSZ-1 generate
   -- az if generate alkalmazásával feltételesen a regiszter
   -- láncban az első  bistabil áramkör bemenetét a d
   -- bemeneti port szolgáltatja
         if_1: if i = 0 generate
         -- d tároló megvalósítása
```

```
            reg_0 : process(src_clk, reset)  --élesítő
                jelek
            begin
            -- felfutó óraél detektálása
               if src_clk'event and src_clk='1' then
                  if reset='1' then -- reset ellenőrzése
                     s(i)<= '0';   --tároló inicializálása
                         resetkor
                  elsif CE='1' then
                     s(i)<=d;    -- ha az órajel
                     -- engedélyezve, akkor az adat beírása
                     --a tárolóba
                  end if;
               end if;
            end process;
         end Generate;
      -- a többi tároló példántosítása
         if_i : if i>0 generate
            reg_i:process (src_clk, reset) -- élesítő jelek
            begin
            -- felfutó óraél detektálása
               if src_clk'event and src_clk='1' then
                  if reset='1' then
                     s(i)<= '0';
                  elsif CE='1' then
                     s(i)<=s(i-1); -- a következő  tárolók
                     -- a bemeneteiket az előző
                      -- tárolótól kapják
                  end if;
               end if;
            end process;
         end generate;
   q(i)<=s(i);  -- minden egyes tároló kimenetének a
   -- kivezetése a q(i) kimeneti portjelre
end generate;
end Behavioral;
```

A 3.8., 3.9. ábrákon egyetlen programnak az alkalmazásával két külön áramkör van generálva. Az egyik változatban az RSZ generikus változó értéke 8, míg a másik változatban 4.



3.8. ábra. 8 bites léptetőregiszter



3.9. ábra. 4 bites léptetőregiszter

3.1.3. Csomagok, függvények, eljárások

Gyakran használt kódrészek megírhatók komponensek (*COMPONENT*), függvények (*FUNCTIONS*) és eljárások formában, amelyeket csomagokban helyezünk el, és végül lefordítjuk egy célkönyvtárba (*LIBRARY*-ba). Ezen technikák előnye tehát, hogy lehetővé teszi a kód particionálását (felosztását), megosztását és újrafelhasználását.

Objektumok csomagokba való szervezése

A komponenseken (*COMPONENTS*), függvényeken (*FUNCTIONS*) és eljárásokon (*PROCEDURES*) kívül egy csomag tartalmazhat típusokat (*TYPE*) és konstansokat (*CONSTANT*) [11].

A csomag felépítése a következő programrészben van bevezetve:

```
PACKAGE csomag_neve IS
   -- [Deklarációs rész;]
END PACKAGE

[PACKAGE BODY csomag_neve IS
   -- Függvények és eljárások leírása;
END csomag_neve;]
```

Amint látható, egy csomag meghatározása két részből áll, a *PACKAGE* (a csomag deklarációs része) és a *PACKAGE BODY* (a csomag törzse). Az első rész kötelező és tartalmazza az összes deklarációt. A második rész csak abban az esetben kötelező, amikor az első rész tartalmaz függvényeket és eljárásokat. Ebben a részben meg kell határozni az alprogram leírását. A *PACKAGE* és *PACKAGE BODY* ugyanazt a nevet kell tartalmazza. A deklarációs rész a következő típusú elemeket tartalmazza: *COMPONENT*, *FUNCTION*, *PROCEDURE*, *TYPE*, *CONSTANT* stb.

Példák csomagok felépítésére.

```
LIBRARY ieee;
USE ieee. Std_logic_vektor.all;
PACKAGE pelda_csomag IS
   --típus deklarálása csomagban
   TYPE allapot   IS (st1, st2, st3, st4);
   TYPE szin   IS (red, green, blue, black);
   -- konstans deklarálása csomagban
   CONSTANT vector : STD_LOGIC_VECTOR (3 downto 0):="1111";
END példa_csomag;
```

Az előbbi példában megadott csomag esetében, mivel csak konstansokat és típusokat tartalmaz, törzsét (*PACKAGE BODY*) nem kell, nem szükséges megadni.

```
LIBRARY ieee;
USE  iee.std_logic_1164.all;
PACKAGE pelda_csomag IS
   --típus deklarálása csomagban
   TYPE állapot   IS (st1, st2, st3, st4);
   TYPE szín   IS (red, green, blue, black);
   -- konstans deklarálása csomagban
```

```
   CONSTANT vector : STD_LOGIC_VECTOR (3 downto 0):="1111";
   -- függvények deklarálása csomagban
   FUNCTION felfuto_el (SIGNAL s:STD_LOGIC_VECTOR) RETURN
       BOOLEAN;
   END példa_csomag;
```

Ebben az esetben most már szükség van a csomag törzsének is a meghatározására.

```
-- csomag törzse
PACKAGE BODY példa_csomag IS
-- függvény megvalósítása
FUNCTION felfutó_él (SIGNAL S:STD_LOGIC) RETURN BOOLEAN IS
BEGIN
   RETURN (S'EVENT and S='1'); -- felfutó él detektálása és
   -- BOOLEAN típusként való visszatérítése
END felfutó_él;
END példa_csomag;
```

Ha alkatrészként deklarálunk egy programrészt, akkor használható egy másik áramkörben és lehetővé teszi a hierarchikus tervezést. Gyakran használt áramkörök, mint például bistabilok, multiplexerek, összeadó áramkörök, kapuk, elhelyezhetők egy könyvtárban, és így egy másik tervben használhatjuk anélkül, hogy újraírnánk az elemet leíró kódot (programot). Egy alkatrésznek (*COMPONENT*) a használatához két programrészt kell meghatározni. Először kell deklarálni (deklarációs rész), másodszor – amikor használom – az alkatrészt kell csatolni.

Modulon belüli alkatrész deklarálása:

```
COMPONENT komp_nev IS
GENERIC
 -- paraméterek deklarálása
PORT(
   Port_neve : signal_mod signal_tipus;
   -- .....................);
END COMPONENT;
```

Modulon belüli alkatrész példányosítása:

```
Címke: komp_nev
GENERIC MAP
   -- paraméterek csatolása
PORT MAP
   -- portok csatolása;
```

A deklarációs rész szintaxisa megegyezik az *ENTITY* paranccsal. Ebben a részben meghatározzuk a ki/bemeneti jeleket és a jelek típusát,

valamint a ki/bemeneti módot (*IN*, *OUT*, *INOUT*, *BUFFER*). A példányosításhoz szükség van egy címkére, (azonosítóra), a modulon belüli alkatrész nevére és utána a *GENERIC MAP* és *PORT MAP* részekben meghatározzuk a paraméterek és a portjelek csatolását.

A komponens deklarálása a törzs (architecture) deklarációs részében, a példányosítás pedig a megvalósítási részben történik.

Példa az alkatrész deklarálására:

```
-------------COMPONENT DEKLARÁLÁS---------------------
COMPONENT inverter IS
PORT ( --modulon belüli alkatrész  portjeleinek a
       -- meghatározása
   a : IN STD_LOGIC;
   b: OUT STD_LOGIC);
END COMPONENT;


------------COMPONENT PÉLDÁNYOSÍTÁS-----------------

v1 : inverter
PORT MAP (x,y);
```

A példaprogramban:

- $v1$: címke, azonosító, a létrejött tervben a címke alapján azonosítható egy-egy áramkör,
- inverter: a komponens neve, amely példányosítva van,
- x és y a jelek vagy portjelek, amelyekre kapcsoljuk a komponensünket.

A komponensnek a jelekkel való összekapcsolására két lehetőség van:

- Sorrendi hozzárendelés (Positional mapping): Ebben az esetben csak egymás után fel kell sorolni a jeleket, a sorrend meghatározza, hogy melyik külső jel a komponens melyik belső jelére kapcsolódik.
- Névleges hozzárendelés: Ebben az esetben külön-külön minden jelre, a jel megnevezése alapján meghatározzuk, hogy melyik külső jelhez fog kapcsolódni.

Példa ugyanannak az elemnek a sorrendi, illetve névleges jelpárosítására:

```
COMPONENT inverter IS
PORT (
   a: IN STD_LOGIC; -- bemeneti jel
   b : OUT STD_LOGIC);    -- kimeneti jel
END COMPONENT;
```

```
-- inverter példányosítása
-- V1 első páldány címkéje

V1:inverter
-- inverter bemeneti és kimeneti jeleinek sorrend
-- szerinti illesztése
   PORT MAP (x,y);

-- inverter példányosítása
-- V2 második példány címkéje
-- baloldali jelek a példányosítandó komponens portjelei
-- joboldali jelek: bemeneti vagy kimeneti portjelek,
-- illetve belső jelek amelyek a modulok összekapcsolásra
-- szolgálnak

V2:inverter
   PORT MAP (
    -- inverter bemeneti és kimeneti jeleinek
    -- név szerinti illesztése
      a=>x,
      b=>y);
```

A generikus paramétereknek komponensekben való alkalmazásakor a *GENERIC* részt is kell deklarálni és példányosítani. Ebben az esetben a komponens deklarációs részében megadjuk a *GENERIC* paramétereket, a *GENERIC MAP* részen pedig felülírjuk az alapértelmezetten megadott generikus paramétereket.

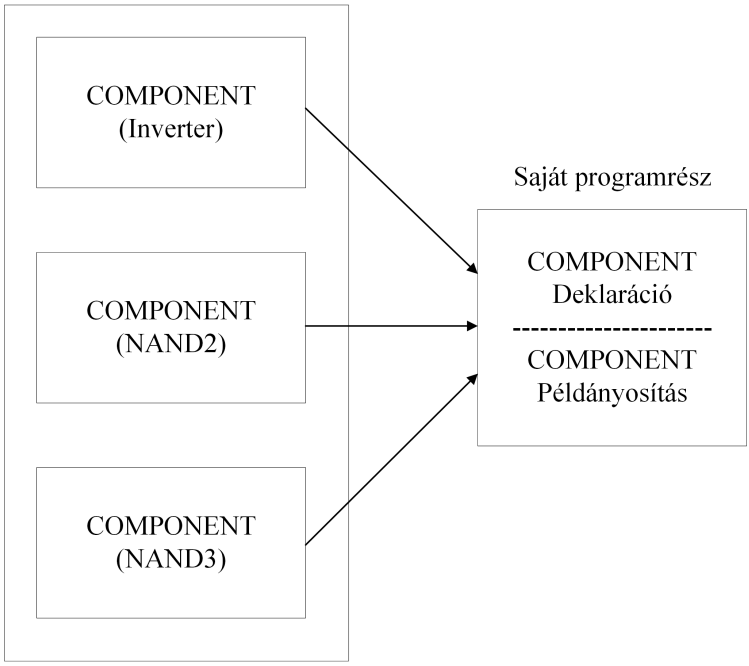
Kódrészlet generikus paramétert is tartalmazó komponens deklarálására:

```
COMPONENT komp_neve IS
GENERIC (
   -- generic paraméter lista, ugyanazok a
   -- paraméterek amit meghatározunk az ENTITY-ben
)
PORT (
   --ugyanazok a portok amit meghatározunk az ENTITY -ben
);
END COMPONENT;

Címke: komp_neve
GENERIC MAP
   -- (paraméter lista leképzése)
PORT MAP
   -- (port lista leképzése);
```

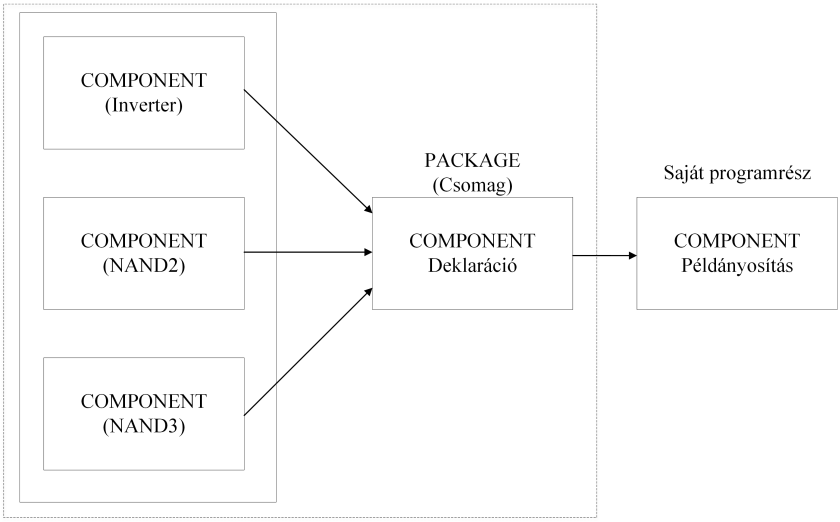
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY paritas IS
GENERIC (n: INTEGER :=7); -- alapértelmezetten megadott érték
PORT (
   be : IN STD_LOGIC_VECTOR (n downto 0); -- bemeneti jel
   ki :OUT STD_LOGIC_VECTYOR (n+1 downto 0)); -- kimeneti jel
end paritas;

ARCHITECTURE behavioral of paritas IS

BEGIN
PROCESS (be)
   VARIABLE temp1:STD_LOGIC;
   VARIABLE temp2 :STD_LOGIC_VECTOR(ki'RANGE);
BEGIN
Temp1:='0';
FOR I IN be'RANGE LOOP
-- I ciklusváltozóval  be jel i-edik elemének címzése
   Temp1:=Temp1 XOR be(i);  -- paritas ellenőrzése
   Temp2(i):=be(i); -- az i-dik elem beírása a Temp2
   -- változóba
END LOOP;
   Temp2(ki'HIGH):=temp1;  -- a paritásbit beírása a Temp2
   -- legnagyobb helyértékű bitjére
   Ki<=temp2; -- a Temp2 változó kiírása a kimeneti portra
END PROCESS;
END behavioral

LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY sajat_program IS
GENERIC (n:POSITIVE:=2); -- generikus paraméter
PORT (
   be : IN STD_LOGIC_VECTOR (n DOWNTO 0);
   ki : OUT STD_LOGIC_VECTOR (n+1 DOWNTO 0);
END sajat_program;

ARCHITECTURE sajat_architektura OF sajat_program IS
   -- paritas modulból komponens deklarálása
COMPONENT paritas IS
GENERIC (n:POSITIVE);  --generikus paraméter
PORT (
   be : IN STD_LOGIC_VECTOR (n DOWNTO 0);
   ki : OUT STD_LOGIC_VECTOR (n+1 DOWNTO 0);
END COMPONENT;
BEGIN
```

```
-- paritas modul példányosítása, első változat
C1: paritas
   GENERIC MAP (n)  --generikus paraméter sorrend szerinti
   -- leképzése
   PORT MAP (be,ki); --portjelek sorrend szerinti leképzése
-- paritas modul példányosítása, második változat
C2: paritas
-- generikus paraméterek név szerinti leképzése
   GENERIC MAP (n=>n)
   -- portjelek név szerinti leképzése
   PORT MAP (be=>be,ki=>ki);
END sajat_architektura;
```

Az alkatrész (COMPONENT) deklarálható a saját kódban (3.10. ábra) vagy elhelyezhető egy könyvtárban (3.11. ábra).



3.10. ábra. Modulon belüli alkatrész saját programrészben való deklarálása



3.11. ábra. Modulon belüli alkatrész csomagban való deklarálása

Függvények és eljárások

A függvényeket szekvenciális algoritmusok leírására alkalmazzák, amelyek egyetlen értéket térítenek vissza, mint például a típuskonverzió.

A *PROCEDURE* szerepe a viselkedési leírással tervezett elemek moduláris részekre való bontása, mint például egy processzoron belül az aritmetikai egység, memória, vezérlőegység stb. Az eljárás nem feltétlenül térít vissza értéket, viszont több értéket is visszatéríthet, ha az argumentumban out vagy inout típusú jelek vannak alkalmazva.

A függvényeket és eljárásokat főleg közösen használt programrészek megosztása és újrafelhasználása miatt a könyvtárakban célszerű tárolni. A függvények vagy eljárások az említettek ellenére deklarálhatók a saját programban is.

A szekvenciális programrész egy része függvényekben van implementálva. A cél új függvények létrehozása az általában jelen levő feladatokra, mint például: aritmetikai műveletek, adatkonverzió, logikai műveletek, új operátorok és attribútumok. A programrészeket függvényekbe szervezve megoszthatók és újrafelhasználhatók. Nagyrészt ugyanazok az utasítások alkalmazhatóak, mint a folyamaton belül, szekvenciális implementációra alkalmas utasítások, *IF*, *WAIT*, *CASE* és *LOOP*, kivételt képez a *WAIT*.

A legtöbb eszköz nem szintetizálja a *WAIT* kifejezést is tartalmazó alprogramrészeket. Ezenkívül egy függvény keretén belül nem használható a *SIGNAL* és *COMPONENT* deklarálás. Egy függvény használata során két részre van szükség: a függvény leírására és a függvényhívásra.

```
FUNCTION függvény_neve  [(parameter_lista)] RETURN
    adat_tipus IS
[deklarációs rész];
BEGIN
   -- (SZEKVENCIÁLIS KIFEJEZÉSEK)
END függvény_neve;
```

A paraméterlista tartalmazhat konstansokat (*CONSTANT*) és jeleket (*SIGNAL*), viszont a paraméterlistában nem engedélyezett a változók (*VARIABLE*) használata. A signalok típusa lehet bármilyen szintetizálható típus: *BOOLEAN*, *STD_LOGIC*, *INTEGER*. A paraméterlistában a vektorok és egészek esetében nem kell meghatározni a bitek számát, valamint az intervallumot.

Példa függvény meghatározására (törzse), amely felfutó élet detektál:

```
FUNCTION felfuto_el_detektor  (SIGNAL s: STD_LOGIC) RETURN
    BOOLEAN
BEGIN
   RETURN (s'event and s='1');
END felfuto_el_detektor;
```

A függvény meghívása egy programrészben a következőképpen alakul:

```
IF felfutó_el_detektor(s) THEN .....
```

A felfuto_el_detektor függvény paraméterként megkapja az s bemeneti jelet. Ha a szimulációs ciklus alatt az s jel értéket vált, és a jel értéke logikai '1'-es, vagyis egy felfutó él volt az s-jelen, abban az esetben a függvény logikai igazat térít vissza, ellenkezőleg pedig hamisat. Az if kifejezésben a függvény által visszatérített logikai érték alapján feltételesen lefut vagy nem a then kifejezést követő programrész.

Egészre való konverzióra egy függvény a következő példaprogram szerint valósítható meg:

```
FUNCTION conv_integer (SIGNAL vektor: STD_LOGIC_VECTOR)
RETURN INTEGER IS
--deklarációs rész

VARIABLE eredmeny : INTEGER  RANGE 0 TO 2* *vektor'LENGTH-1;
BEGIN
-- függvény implementációs része
```

```
-- legnagyobb helyértékű bit ellenőrzése
   IF vektor (vektor'HIGH) ='1' THEN
   eredmeny :=1;
Else
   eredmeny :=0;
END IF;
-- egyenként vesszük a biteket
FOR i IN vektor'HIGH-1 DOWNTO (vektor'LOW ) LOOP
   eredmeny:=eredmeny*2;  -- eredmény szorzása kettővel,
   -- vagyis balra való eltolás
   IF vektor(i) ='1' THEN   -- ha az ellenőrzött bit '1'-es,
   --  növeljük az eredmény változót
      eredmeny:= eredmeny+1;
   END IF;
END LOOP;
 -- eredmeny változó visszatérítése a függvényből.
RETURN eredmeny;
END conv_integer;
```

A függvényhívás ebben az esetben a következőképpen történik:

```
Y<=con_integer(a);
```

A függvény elhelyezhető csomagban vagy saját programrészben. Függvénynek saját programrészben való elhelyezését a következő VHDL kódrészlet szemlélteti:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY dff IS
PORT
(d, clk, rst : INT STD_LOGIC;  -- bemeneti jelek
   Q:    OUT STD_LOGIC);        -- kimeneti jel
END dff;

ARCHITECTURE sajat_dff of dff  IS
-- függvény létrehozása felfutó él detektálására
-- függvény deklarálása
FUNCTION felfuto_el_detektor (SIGNAL s: STD_LOGIC) RETURN
    BOOLEAN IS
BEGIN
   -- függvény megvalósítása
   RETURN S'EVENT and S='1';
END felfuto_el_detektor;

BEGIN
```

```
PROCESS (clk,rst) -- élesítő jelek
BEGIN
IF (rst='1') THEN   -- reset feltétel ellenőrzése
   q<='0';        -- tároló kezdőértékének inicializálása
ELSIF felfuto_el_detektor(clk) THEN  -- felfutó él
   -- detektálása
   q<=d; --bemeneti adat beírása a tárolóba
END IF;
END PROCESS:
END sajat_dff;
```

Csomagban tárolt függvény alkalmazásának szemléltetése: előnye a csomagban tárolt függvénynek, hogy újrafelhasználható és megosztható. Az alábbiakban két VHDL programot szemléltetünk, ahol a függvényt leírjuk, a másik pedig, amelyben meghívjuk. Ha a függvényt a csomagban deklaráltuk, akkor szükséges a csomag bejelentése.

```
LIBRARY ieee;
USE ieee.std_logic_all;

-- függvénynek csomagban való elhelyezése
PACKAGE sajat_csomag IS
-- függvény fejlécének leírása
FUNCTION felfuto_el_detektor (SIGNAL s:STD_LOGIC)
RETURN BOOLEAN;
END SAJÁT CSOMAG;
-- csomag törzsének definiálása
PACKAGE BODY sajat_csomag IS
   FUNCTION felfuto_el_detektor (SIGNAL s:STD_LOGIC)
RETURN BOOLEAN IS
BEGIN
-- függvény működésének leírása
RETURN S'EVENT AND S='1' ;
END felfuto_el_detektor;
END sajat_csomag;
```

A főprogramrész a függvény alkalmazására a következőképpen alakul:

```
LIBRARY ieee
USE ieee.std_logic_1164.all;
USE work.sajat_csomag.all;

ENTITY dff IS
PORT (
   d,clk,rst : IN STD_LOGIC; --bemeneti portjelek deklarálása
   Q: out STD_LOGIC);   -- kimeneti portjel deklarálása
END dff;
```

```
ARCHITECTURE sajat_dff OF dff  IS
BEGIN
-- tárolót megvalósító folyamat
PROCESS (Clk,Rst)   -- élesítő jelek
BEGIN
   IF rst='1' THEN   -- reset feltétel ellenőrzése
      q<='0';         --tároló kezdőértékének inicializálása
   ELSIF
      Felfuto_el_detektor(Clk) THEN q<=d; -- felfutó él
      -- detektálása, függvényként megvalósított változat
   END IF;
END PROCESS;
END sajat_dff;
```

Az eljárások (*PROCEDURE*) meghívhatók konkurensen vagy szekvenciálisan. A párhuzamos eljáráshívás végrehajtása akkor történik, ha bármelyik be- vagy kimenet paramétere megváltozik. Az eljárás szerkezete hasonló a függvényhez. A függvényhez hasonlóan két része van: az eljárás leírása és az eljárás meghívása.

```
PROCEDURE eljara_NEVE  (parameter_lista) IS
--    Deklarációs_rész;
BEGIN
-- Szekvenciális kifejezések
END eljárás_neve;
```

Ahol a paraméterlista lehet:

- konstans *CONSTANT* konstans_neve : mode, típus;
- jel *SIGNAL* signal_neve: mode, típus;
- változó *VARIABLE* változó_neve: mode, típus.

Egy eljárásnak lehet bármennyi *IN*, *OUT*, *INOUT* paramétere, amelyek, amint említettük lehetnek, konstansok, jelek vagy változók. A bemeneti paraméterek alapértelmezetten konstansok, a kimeneti paraméterek pedig alapértelmezetten változók. A *WAIT*, *SIGNAL* és *COMPONENT* nem szintetizálható, ha eljárásban használjuk. Kivétel lehet a signal, de ebben az esetben az eljárást kötelező a folyamaton belül használni.

A fejezet a konkurens végrehajtású VHDL kifejezéseket, ciklikus és feltételes alkatrész-példányosítást, kódrészeknek függvényekbe és eljárásokba való szervezését foglalja össze.

A fentről lefele vagy a lentről felfele való építkezés és a terv átláthatósága szempontjából meghatározó a részegységeknek alkatrészekbe (*COMPONENT*) való szervezése.

Komplex áramkörök tervezésekor kulcsszerepet játszik az egyes alkatrészek automatikus többszöri és feltételes példányosítása az if generate és for generate utasítások alkalmazása révén.

A parametrikus tervezés (*GENERIC*) lehetővé teszi egy létrehozott terv rugalmas méretezését. Egy parametrikusan elkészített terv a paraméterek megválasztásával egyszerűen méretezhető egy konkrét célfeladatnak megfelelően.

A tervek rendszerezése szempontjából fontos az egységeknek függvényekbe, eljárásokba való elhelyezése, valamint az egyszerű újrahasznosítás céljából ezeknek csomagokba való szervezése.

4. fejezet

FPGA-ban létrehozott áramkörök tesztelése

A laborgyakorlat célja az FPGA-n elkészített áramkörök tesztelése az FPGA-ba integrált analizátor alkalmazásával.

4.1. Bevezető

FPGA-ra készített áramkörök (kapcsolási rajzok) tesztelésére több lehetőség van:

- áramkör működésének szimulációja, valamely szimulációs környezetben;
- külső méréstechnikai eszközök (oszcilloszkóp, külső logikai analizátor) alkalmazása;
- az FPGA áramkörbe integrált logikai analizátor alkalmazása (áramkör működés közben való tesztelése).

4.1.1. VHDL-ben specifikált terv szimulációja

VHDL-ben specifikált áramkör szimulációjára több szimulációs környezet is alkalmazható. A tervezési fázis egyes lépései után (minden fázisnak megfelel egy szimulációs modell) különböző szintű szimuláció végezhető el (részletezve az 1.6. ábrán):

- viselkedési szimuláció szintezést követően,
- funkcionális szimuláció a fordítást követően,

– statikus időanalízis a leképzési fázis után,
– időanalízis - az elhelyezést és huzalozást követően,
– a szimuláción kívül elvégezhető az áramkör-teljesítmény analízise.

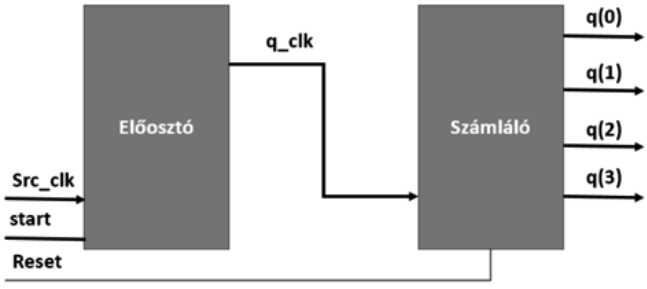
A régebbi FPGA rendszerekre: VIRTEX V, Spartan 6 VHDL-ben vagy Verilogban specifikált áramkörök szimulációval történő tesztelésére a XILNX ISE Design Suite tervezőkörnyezetben található ISIM modul alkalmazható. A régebbi verziókban az ISIM szimulátorban a gerjesztő jeleket egy grafikus felületen is meg lehetett határozni, rajzolni. Az újabb Vivado verziókban nincs meg ez a lehetőség a szimulációs sablon automatikus létrehozására.

A gerjesztő jeleket VHDL utasításokkal kell meghatározni. Az ISE környezetben a tervhez kell adni egy szimulációs modellt, amely egy minta VHDL állomány létrehozását eredményezi, amelyhez egyrészt hozzá van csatolva a szimulálandó modul, másrészt a gerjesztő jelek. Szekvenciális áramkör esetén az órajel generálásáért egy külön folyamat a felelős. A gerjesztő jeleknek az időbeli változását, alakulását a szimulációt végző felhasználónak kell elkészítenie. Egyes esetekben lehetőség van a tesztjelek automatikusan történő létrehozására.

Az egymást követő jelek időbeli ütemezéséhez a wait for utasítás alkalmazására van szükség. Meghatározzuk a gerjesztő jeleket, majd wait for utasítással bevárjuk a szimulációs idődiagramnak megfelelő időt.

A következő példa alapján szemléltetjük egy áramkör VHDL alapú, majd működés közben történő szimulációját.

A tesztelendő modul egy előosztóból és egy számlálóból van kialakítva a 4.1. ábrán szemléltetett tömbvázlat szerint.



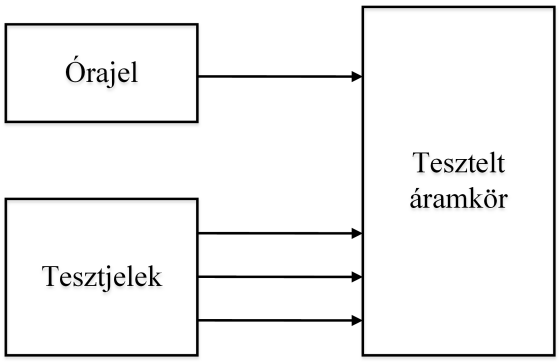
4.1. ábra. Tesztelendő áramkör

Az előosztót és a számláló modult egy-egy folyamat valósítja meg és a következő VHDL programrész szemlélteti:

```
Library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
-- use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if
    instantiating
-- any Xilinx leaf cells in this code.
-- library UNISIM;
-- use UNISIM.VComponents.all;



entity top_level_2 is
generic (
div_val : natural:=256); -- generikus paraméterek deklarálása
Port (
   src_clk : in STD_LOGIC; --bemeneti órajel
   reset : in STD_LOGIC;  -- bemeneti reset jel
   start : in STD_LOGIC;  -- modul indítására szolgáló jel
   q : out STD_LOGIC_VECTOR (3 downto 0)); --kimeneti jel
end top_level_2;

architecture Behavioral of top_level_2 is

signal q_clk: std_logic; -- belső órajel, a számláló modul
órajelbemenete
-- tesztelésre a jelek megjelölhetőek a mark_debug
-- attributummal
-- attribute mark_debug : string;
-- attribute mark_debug of reset: signal is "true";
-- attribute mark_debug of start: signal is "true";
-- attribute mark_debug of q: signal is "true";
-- attribute mark_debug of q_clk: signal is "true";
-- attribute mark_debug of  q_div : signal is "true";

begin

--számláló VHDL  programkódja
szamlalo: process(q_clk, reset) --q_clk, reset élesítő jelek
   -- számláló értékének tárolására szolgáló változó
```

```
   variable sz: std_logic_vector(3 downto 0);
begin
   if reset='1' then  --reset feltétel ellenőrzése
      sz:=(others=>'0'); --számláló kezdőértékének
      -- inicializálása
   -- felfutó él detektálása
   elsif q_clk'event and q_clk='1' and start='1' then
      sz:=sz+1; --számláló inkrementálása felfutó óraélre
   end if;
q<=sz; -- számláló értékének a rákapcsolása a kimeneti jelre
end process szamlalo;

-- előosztó VHDL programkódja
modulo:process(src_clk,reset) -- élesítő jelek
-- változó deklarálása az előosztóban szereplő
-- számláló számára
variable x: integer range 1023 downto 0 := 0;
-- q_div előosztóból kimenő leosztott órajel tárolására
-- szolgáló változó
variable q_div: STD_LOGIC:= '0';

begin
-- felfutó óraél detektálása
if src_clk'event and src_clk='1' then
   if x<div_val then   -- számláló felső határának
   -- ellenőrzése
      x:=x+1; --számláló inkrementálása
      q_div:=q_div;
   else
      x:=0; -- felsőkorlát túllépése esetén a számláló
      -- értékének nullázása
      q_div:=not(q_div);  -- órajel tagadása minden div_val
      -- lépést követően
   end if;
   q_clk<=q_div; --leosztott órajel rávezetése  a belső
   -- órajelre
end if;
end process modulo;
end Behavioral;
```

A VHDL alapú teszteléshez szükséges egy tesztelő áramkör létrehozása, amely tartalmazza a tesztelt áramkört, a tesztjeleket, valamint szekvenciális áramkör esetén az órajelet generáló modult (4.2. ábra). A tesztelő áramkör egy olyan VHDL állomány, aminek az entitása üres.



4.2. ábra. VHDL alapú tesztelő áramkör tömbvázlata

A tesztelő áramkört a következő VHDL program valósítja meg:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-- üres entitású tesztelő áramkör
entity szim is
generic (div_val : natural:=16);
--  Port ( );
end szim;

architecture Behavioral of szim is
--belső tesztjelek létrehozása
signal src_clk : STD_LOGIC; --belső órajel
signal reset : STD_LOGIC;  -- belső reset jel
signal start : STD_LOGIC;  -- belső start jel
signal  q: STD_LOGIC_VECTOR (3 downto 0);

-- teszt alkatrész deklarálása
component top_level_2 is
generic (
div_val : natural:=256); -- generikus paraméterek deklarálása
Port (
   src_clk : in STD_LOGIC; --bemeneti órajel
   reset : in STD_LOGIC;  -- bemeneti reset jel
   start : in STD_LOGIC;  -- modul indítására szolgáló jel
   q : out STD_LOGIC_VECTOR (3 downto 0)); --kimeneti jel
end component;

begin
-- órajelet generáló programrész
orajel:process
```

```
begin
   src_clk<='0';   -- órajelre '0'
   wait for 10ns;  -- várakozás egy ciklus ideig
   src_clk<='1';   -- órajelre '1'
   wait for 10ns;  -- várakozás egy ciklus ideig
end process orajel;

-- tesztelt áramkör példányosítása
tesztelt_aramkor:top_level_2
generic map (div_val=> div_val)
port map (
   src_clk => src_clk, --bemeneti órajel
   reset => reset,  -- bemeneti reset jel
   start => start,  -- modul indítására szolgáló jel
   q => q); --tesztelt áramkör kimeneti jele

-- tesztjelek létrehozása
tesztjelek:process
begin
   reset<='1';  -- áramkör kezdeti beállítása
   wait for 100ns;  -- várakozás 100 ns
   start<='0';
   reset<='0';
   wait for 10ns;
   start<='1';     -- áramkör indítása
   wait for 2500ns;   -- várakozás 250 óraciklusig
   wait;
end process;
end Behavioral;
```

4.1.2. Tesztelés külső méréstechnikai eszközök alkalmazásával

Sok esetben az FPGA-ra elkészített áramkör szimulációja során helyes működést tanúsít, azonban az FPGA áramkörbe való betöltést követően a tervezett áramkör hibás működését észleljük. A hibás modul (alegység) vagy hibásan működő rész felderítésében az FPGA-ba betöltött áramkör jeleinek működés közbeni analizálása vezet eredményre.

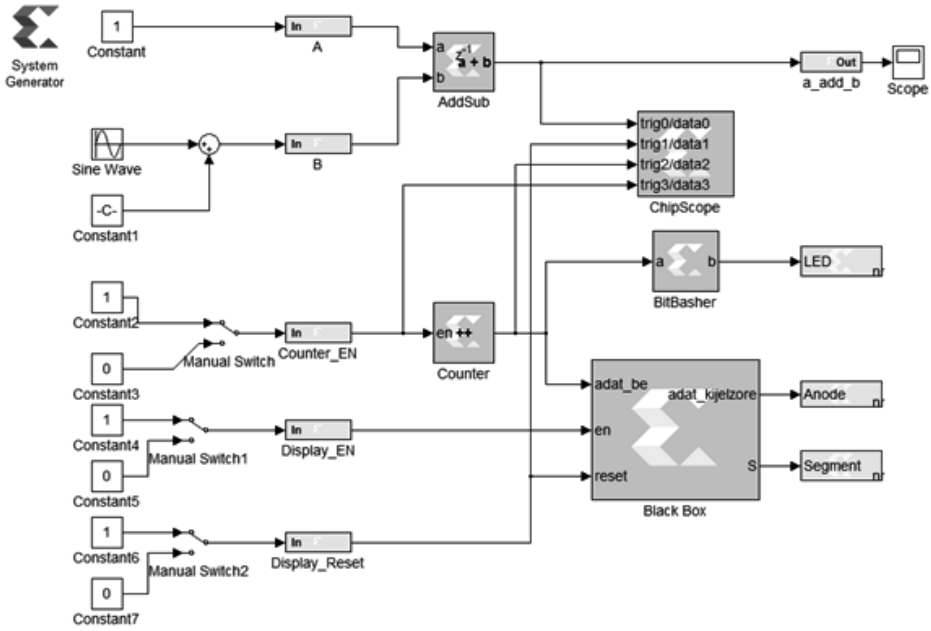
Az áramkör belső jeleinek a tanulmányozására, megjelenítésére használhatóak külső méréstechnikai eszközök, mint például oszcilloszkóp vagy külső logikai analizátor. Az oszcilloszkópon általában kevés számú jel (az oszcilloszkóp csatornáinak megfelelően) tanulmányozható, és általában a jelnek egy pillanatnyi szakaszában (kevés mintavétel), az oszcilloszkóp kijelzőjén megjeleníthető. Külső logikai analizátor alkalmazásával több csatornán,

hosszabb ideig történik a mérés, az adatok egy tampon memóriába kerülnek. A mérés leállítását követően a jelek vagy a dedikált logikai analizátor kijelzőjén, vagy a számítógép kijelzőjén grafikusan vannak ábrázolva. A felhasználó a mért jel különböző részeit analizálhatja, kereshet a jelformában, több jelet csoportosíthat, amelyeket egységesen kezel, különböző formátumokban jelenítheti meg a mérési eredményeket. A külső logikai analizátor alkalmazásakor a vizsgálandó jeleket ki kell vezetni az FPGA áramkörből a fejlesztőrendszer megfelelő kimeneteire. Az analizátor csatlakozóját rá kell kötni az FPGA áramkör megfelelő kivezetéseire.

4.2. FPGA áramkörbe integrált logikai analizátor

A következő és talán a leghatékonyabb lehetőség a XILINX tervezőkörnyezetben elérhető, hibaelhárításra szolgáló modulok alkalmazása és a számítógép képernyőjén e jelek megjelenítésére szolgáló eszközpár. Az ISE alapú tervezőeszközökben a *ChipScope Pro* segítségével volt lehetőség a jeleknek a számítógépen való megjelenítésére. Az újabb Xilinx tervezőeszközökben lényegében a *ChipScope Pro* modult a Vivado programba integrálták.

Ez a módszer egy *Chipscope IP* magnak az áramkörbe való integrálását jelenti. Ez esetben nem szükséges az analizálandó jeleket kivezetni az FPGA áramkörből, csak rá kell kapcsolni az analizátor modul bemeneteire. Az adatok az FPGA áramkörből a *Joint Test Action Group* (JTAG) teszt interfészen vannak kivezetve. A fejlesztőrendszertől függően a JTAG lánc általában USB protokollon kapcsolódik a számítógépre. A komplexebb FPGA fejlesztőpanelek esetében Ethernet csatlakozás is előfordul. A működés közbeni tesztelés során az analizátor a számítógépen grafikusan megjeleníti a tesztelt jeleket. Az ISE alapú *System Generator* programban lehetőség volt integrálni a tervbe *ChipScope Pro* analizátor modult, és az analizálandó, tesztelendő jeleket rávezetni a tesztegységre. ISE alapú *System Generator* 14.7-es változatában a 4.3 ábra szemlélteti a tesztelendő jeleknek a *ChipScope* modulra való csatolását.



4.3. ábra. Jelek ChipScope modulra való csatolása

4.3. Működés közbeni tesztelés Vivado környezetben

A Vivado tervezőeszközben több lehetőség van, több módon lehet beállítani, konfigurálni a tesztkörnyezetet egy FPGA-ba betöltött áramkör működés közben való tesztelésére [14].

- *Set Up Debug* varázsló alkalmazása a teszteléshez szükséges tesztmodul (Debug Cores) tervbe való integrálására;
- *XDC* utasítások alkalmazása a tesztelő modul (*Debug Core*) tervhez való kapcsolásához és konfigurálásához.

Az áramkörben működés közben való tesztelés több munkafázist tartalmaz:

1. Tesztelendő jelek kijelölése: az áramkörből a tesztelendő jelek kiválasztása, analizálása és tesztelésre való megjelölése.

   a) A tesztelendő jeleknek a HDL forráskódban való megjelölése

   b) A tesztelendő jeleknek a szintetizált áramkörben való megjelölése

c) A tesztelendő jeleknek tcl paranccsal való megjelölése

2. Implementációs fázis: a tesztelésre alkalmazott IP magokat és az IP magra csatlakoztatott tesztjeleket is tartalmazó terv implementációja és konfigurációs állomány létrehozása.

3. Analízis fázis: az FPGA áramkörbe betöltött terv teszt interfészére csatlakozva, működés közben kiolvasásra és megjelenítésre kerülnek a tesztelésre kijelölt jelek. Analizálva a jelformákat, ellenőrizzük az áramkör működését.

4.4. Tesztkörnyezet konfigurálása

4.4.1. Set Up Debug varázsló alkalmazása a teszteléshez szükséges tesztmodul (Debug Cores) tervbe való integrálására

Tesztelésre szánt jelek megjelölését követően a jeleket hozzá kell rendelni tesztelő IP modulhoz. A Vivado tervezőeszköz a *Set Up Debug* varázslóval egy egyszerű felületet biztosít a tesztelendő jeleknek a tesztelendő modulhoz való csatolására, automatikusan létrehozva a tesztelő magokat, hozzárendelve a tesztelendő jeleket a teszt IP mag bemenetére.

A teszt modul hozzáadás több fázisban történik [14]:

– a szintézist követően a tesztelésre szánt vezetékek kijelölése
  1. az *unassigned nets* listából a jelek kiválasztása
  2. a szintetizált áramkörben a jelek megjelölése (jobb klikk a vezetékre a *Sinthesized Design* nézetben, vagy a *Netlist* listából és a *Mark Debug* paranccsal megjelöljük tesztelésre)
– Set Up Debug parancs végrehajtása
  1. A *Flow navigator* ablakban, *Synthesis -> Synthesized Design* majd *Set Up Debug*
  2. *Tools* menüpontban *Set Up Design*
  3. *Window* menüpontból a *Debug* nézetben a tesztelendő jelek kijelölése az *Unassigned Debug Nets* listából és a *Set Up Debug* parancs kiadása (jobb klikk a kijelölt jelekre, majd *Set Up Debug*)
– Ha szükséges, újabb jeleket csatlakoztathatunk a tesztmodul bemenetéhez, vagy eltávolíthatunk a Find Nets to ADD-re kattintva

- Az órajel kiválasztása, amelyet majd alkalmaz a tesztmodul a jel mintavételezésére
- Speciális *trigger* és *capture* mód beállítása
- Ha úgy látjuk, hogy helyesen kijelöltük a tesztjeleket, a *Finish* gombra kattintva hozzáadjuk és csatoljuk az integrált logikai analizátor (ILA) modult a tervhez
- ILA maggal kapcsolatos paraméterek beállítása
  1. *C_DATA_DEPTH*
  2. *C_INPUT_PIPE_STAGES*
  3. *C_EN_STRG_QUAL*
  4. *C_ADV_TRIGGER*
- A tesztelésre szánt jelek (debug nets) a rajz szerint hozzá vannak csatolva a tesztmodulhoz

4.4.2. XDC utasítások alkalmazása a tesztelő modul (*Debug Core*) tervhez való kapcsolásához és konfigurálásához

TCL parancsokat alkalmazva a terv a következőképpen hozható létre:

```
set terv_nev teszt_7
set konyvtar C:/Munka/I_Felev/UKDA_2017/L6
start_gui
create_project ${terv_nev} ${konyvtar}/${terv_nev} -part
    xc7z010clg400-1

#VHDL hardver leíró nyelv beállítása
set_property target_language VHDL [current_project]

#VHDL forráskódnak a tervhez való csatolása
add_files -norecurse ${konyvtar}/top_level_2.vhd

#Megkötés állománynak a tervhez való csatolása
add_files -fileset constrs_1 -norecurse
    ${konyvtar}/system_4.xdc
update_compile_order -fileset sources_1
update_compile_order -fileset sim_1

#Szintézis futtatása
launch_runs synth_1
#Várakozás a szintézis befejezésére
```

```
wait_on_run synth_1

#Synthesized Design megnyitása
open_run synth_1

#Debug core magnak a tervhez való csatolása
create_debug_core u_ila_0 ila

#ILA modul konfigurálása
set_property C_DATA_DEPTH 65536 [get_debug_cores u_ila_0]
set_property C_TRIGIN_EN true [get_debug_cores u_ila_0]
set_property C_TRIGOUT_EN false [get_debug_cores u_ila_0]
set_property C_ADV_TRIGGER true [get_debug_cores u_ila_0]
    #enable advanced trigger mode
set_property C_INPUT_PIPE_STAGES 0 [get_debug_cores u_ila_0]
set_property C_EN_STRG_QUAL true [get_debug_cores u_ila_0]
    #enable Basic capture mode
set_property ALL_PROBE_SAME_MU true [get_debug_cores u_ila_0]
set_property ALL_PROBE_SAME_MU_CNT 1 [get_debug_cores u_ila_0]
set_property port_width 1 [get_debug_ports u_ila_0/clk]

#ILA modul órajelének meghatározása
connect_debug_port u_ila_0/clk [get_nets [list
    src_clk_IBUF_BUFG]]

#A Test modulon probe0 sínszélességének meghatározása
set_property port_width 3 [get_debug_ports u_ila_0/probe0]
#probe0 bemenetre a start, reset és q_div jelek csatolása.

#A forráskódban meghatározott jelek nem érhetőek el
    tesztelésre a szintézist követően, hanem a bemenetek
    esetében a start_IBUF reset_IBUF jeleket alkalmazzuk,
    kimenetek esetében pedig a q_OBUF jeleket. .
connect_debug_port u_ila_0/probe0 [get_nets [list start_IBUF
    reset_IBUF q_div]] jeleket kell alkalmazni

#Egy újabb bemenet (probe1) létrehozása a tesztmodul bemenetére
create_debug_port u_ila_0 probe
#Sínszélesség beállítása
set_property port_width 4 [get_debug_ports u_ila_0/probe1]
#q_OBUF[0], q_OBUF[1], q_OBUF[2], q_OBUF[3]  kimeneteknek a
    probe1 bemeneti portra való csatolása
connect_debug_port u_ila_0/probe1 [get_nets [list q_O*]]

#Megkötés állomány mentése
save_constraints_as system_${terv_nev}.xdc
```

```
set_property constrset system_teszt_6.xdc [get_runs synth_1]
set_property constrset system_teszt_6.xdc [get_runs impl_1]

#Implementáció lefuttatása
launch_runs impl_1
wait_on_run impl_1

#Konfigurációs állomány generálása
launch_runs impl_1 -to_step write_bitstream
wait_on_run impl_1

#Hardver megnyitása
open_hw

#Szerver indítása az FPGA áramkörre való kapcsolódás céljából
connect_hw_server
#Célhardver megnyitása
open_hw_target

#Konfigurációs .bit állomány és debug_net.ltx beállítása
set_property PROGRAM.FILE
    ${konyvtar}/${terv_nev}/${terv_nev}.runs/impl_1/
    top_level_2.bit [lindex [get_hw_devices] 1]
set_property PROBES.FILE
    ${konyvtar}/${terv_nev}/${terv_nev}.runs/impl_1/
    debug_nets.ltx [lindex [get_hw_devices] 1]

current_hw_device [lindex [get_hw_devices] 1]
#hardver frissítése
refresh_hw_device [lindex [get_hw_devices] 1]

#Hardver programozása
program_hw_devices [lindex [get_hw_devices] 1]

refresh_hw_device [lindex [get_hw_devices] 1]
```

A tcl szkript nyelven megadott programrészt lefuttatjuk az első fázistól az utolsóig, minden lépés automatikusan végrehajtódik, az FPGA áramkörre feltöltődik a konfigurációs állomány és csatlakozik a grafikus vizualizációs felülethez. A felprogramozott eszköz készen áll a tesztelés elvégzésére.

4.5. Áramkör működés közben való tesztelése és ennek lépései

A következő példa során egy egyszerű áramkör (mely két számlálót tartalmaz) működés közben való tesztelését vezetjük végig. A következő fontosabb lépések elvégzésére van szükség:

1. A top_level.vhd modul alapján egy terv létrehozása
   a) új terv létrehozása
   b) könyvtár kiválasztása
   c) *top_level.vhd* állománynak a tervhez való csatolása
   d) *system.xdc* megkötés állománynak a tervhez való csatolása
   e) újrakonfigurálható áramkör típusának kiválasztása (4.4. ábra)

4.4. ábra. FPGA áramkör típusának kiválasztása

2. A tesztelendő jelek kiválasztása, a debug interfésznek a tervbe való csatolása, a jeleknek a tesztmodulba való csatolása
   a) Kijelölt tesztelendő jelek
      i. *reset*
      ii. *start*
      iii. *q* – kimenet
      iv. *q_div* – előosztóban alkalmazott változó a leosztott jel rögzítésére
      v. *q_clk* – leosztott órajel
   b) a tesztelendő jelek bejelölése a VHDL forráskódban szintézis előtt
      – attribútumok alkalmazásával való bejelölés

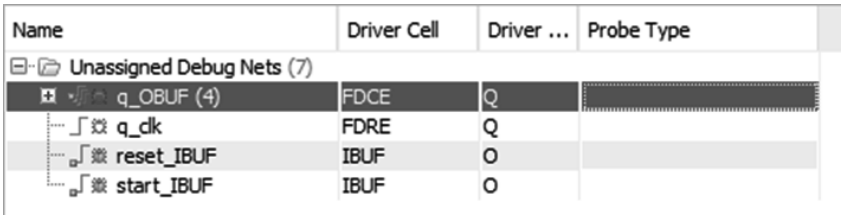
```
attribute mark_debug : string;
attribute mark_debug of reset: signal is "true";
attribute mark_debug of start: signal is "true";
attribute mark_debug of q: signal is "true";
attribute mark_debug of q_clk: signal is "true";
```

– a tesztelésre kijelölt jelekkel az áramkör szintézise (a példa szerint a Tcl parancssorból indítva)

```
launch_runs synth_1
wait_on_run synth_1
```

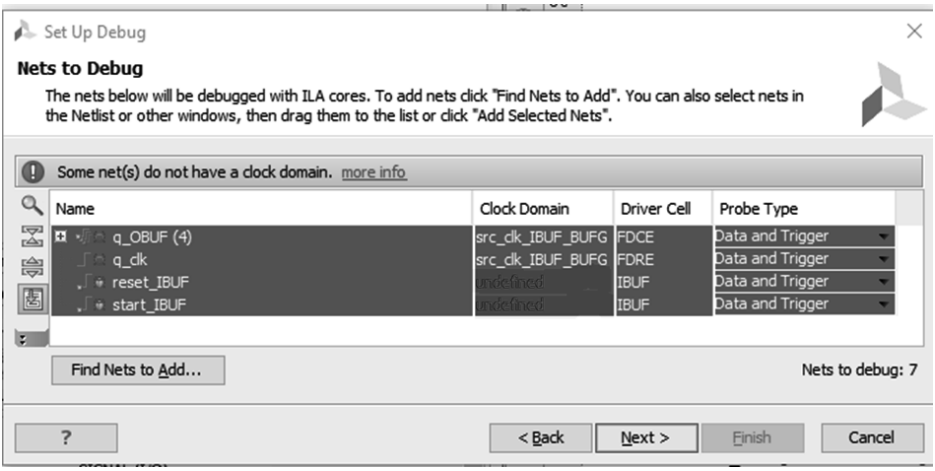
– Setup Debug parancs kiadása
  – A *Flow navigator* ablakban, *Synthesis*, *Synthesized Design*, majd *Set Up Design*
  – Window menüpontból a *Debug* nézetben a tesztelendő jelek kijelölése az *Unassigned Debug Nets* listából és a *Set Up Debug* parancs kiadása

c) *Set Up Debug* parancs kiadása a *Debug* nézetből
  – jobb kattintás az *Unassigned Debug Nets* listából a kijelölt tesztelendő jelekre (4.5. ábra)
  – *Set Up debug* parancs végrehajtása, majd első lépést követően ugrás a következő fázisra (*Next*)



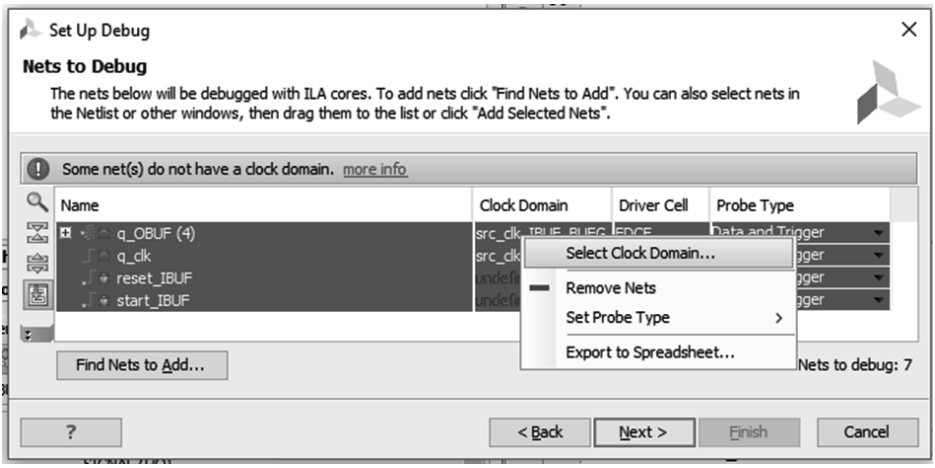
4.5. ábra. Teszteléshez még nem hozzárendelt jelek

3. Órajel-tartományok kiválasztása
Minden egyes tesztelendő jelre meg kell határozni, hogy melyik órajel-tartományhoz tartozik. Ebben az esetben a start és a reset jelekre automatikusan nem sikerült a tervezőeszköznek órajelet hozzárendelni, ezt a lépést most el kell végezni. Az órajel határozza meg, hogy milyen gyakran mintavételezzük a tesztelés során a jeleket.



4.6. ábra. Órajel-tartományok kiválasztása

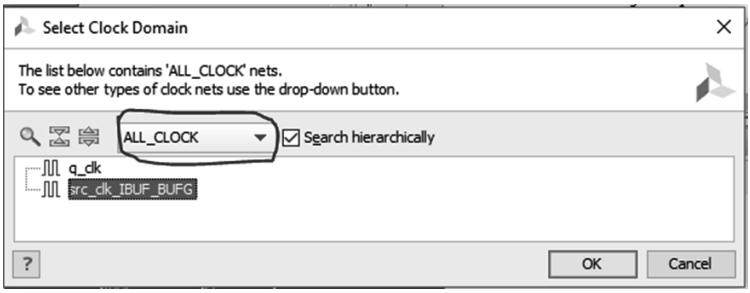
Az órajel a *Clock Domain* tulajdonságra való kattintást követően a *Select Clock Domain* gyorsmenüből választható ki és rendelhető a jelhez.



4.7. ábra. Órajel-tartományok kiválasztása

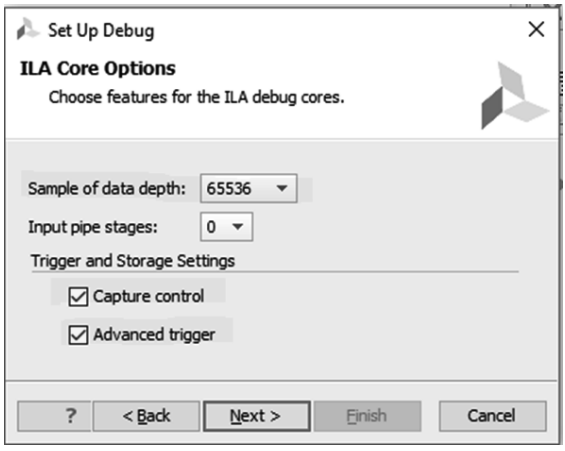
A *Select Clock Domain* ablakban a legördülő listából ki lehet választani, hogy milyen típusú órajelet szeretnénk alkalmazni (*GLOBAL_CLOCK*, *LOCAL_CLOCK*, *REGIONAL_CLOCK*,

ALL_CLOCK). A listában a *src_clk_IBUF_BUFG* a rendszer globális órajele, míg a *q_clk* a tesztelendő áramkör két alegységét összekötő lokális órajele.



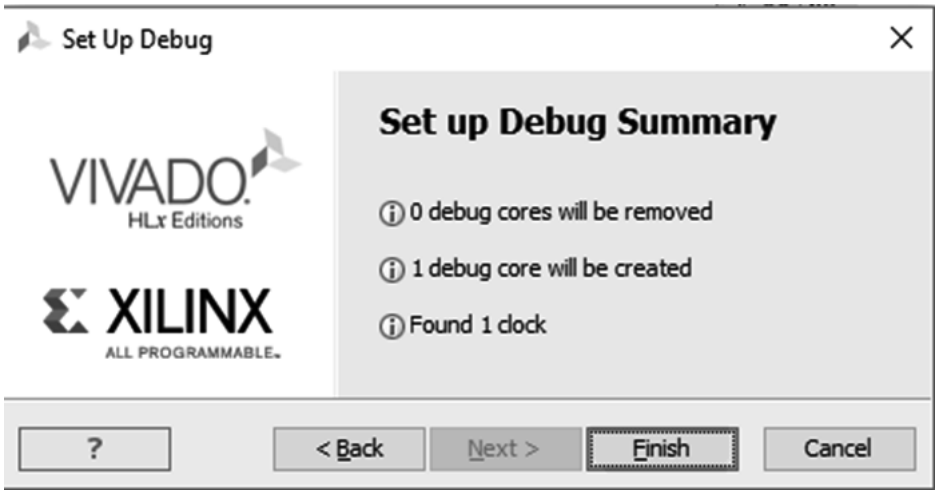
4.8. ábra. Órajel típusának kiválasztása

4. A tesztelő ILA IP modul paraméterezése
A következő ablakban (*ILA Core Options*) a tesztelésre alkalmazott ILA IP mag paraméterezhető (4.9. ábra):
   - egy mérés során beolvasható minták száma (*sample of data depth*)
   - *Input pipe stages* (bemeneti csővezetékek száma)
   - *Capture control* mintavételezés vezérlése
   - *Advanced trigger* (fejlett élesítő beállítások)



4.9. ábra. ILA integrált logikai analizátor paraméterezése

Az utolsó ablakban a terv jelenlegi állapotáról, a hozzáadott *debug core* modulokról, illetve az alkalmazott órajelek számáról jelenít meg információt. A *Finish* gombra kattintva befejezzük a *Set Up Debug* parancs konfigurálását (4.10. ábra).



4.10. ábra. Teszt konfigurálásának befejezése

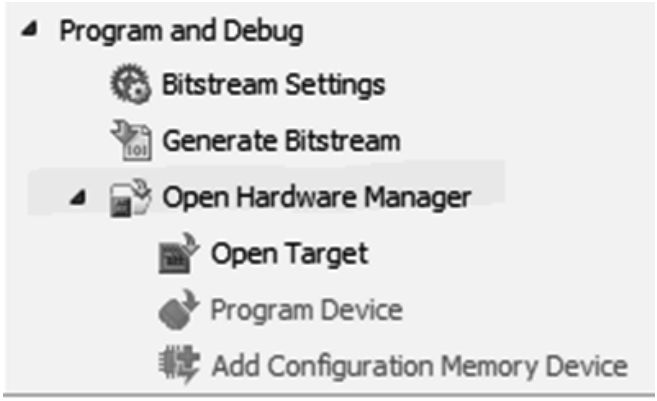
5. A terv implementálása
   Implementáció elindítása a tcl parancsablakból a következő utasítással:

```
launch_runs impl_1
```

6. Konfigurációs állomány létrehozása
   A konfigurációs állomány a tcl parancsablakból a következő utasítással hajtható végre:

```
launch_runs impl_1 -to_step write_bitstream
```

7. A Hardware Manager megnyitása. Ezen a felületen van lehetőség kapcsolódni a hardvereszközhöz. Hardware Manager megnyitása az open_hw vagy a *Flow Navigator* panelen az *Open Hardware Manager* menüponttal (4.11. ábra).



4.11. ábra. Hardware Manager megnyitása

A Logikai terv hardverben való tesztelése, alkalmazva a *Vivado Logic Analyzer* analizátort. A Vivado Analyzer eszköz kapcsolódik a tervben elhelyezett tesztelő modulokkal: ILA, virtuális ki-bemenet (VIO) és JTAG-to AXI. A logikai analizátor funkciók a *Flow Navigator* ablakon a *Program and Debug*, majd *Hardware Manager* menüpontra kattintva érhetők el.

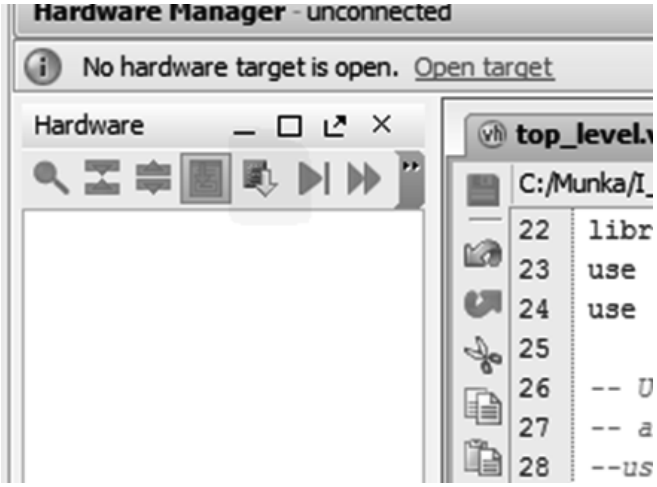
4.6. A tesztelés lebonyolítása

A fontosabb lépések, amelyek szükségesek a hardverteszt megvalósításához, a következő alfejezetekben vannak bemutatva.

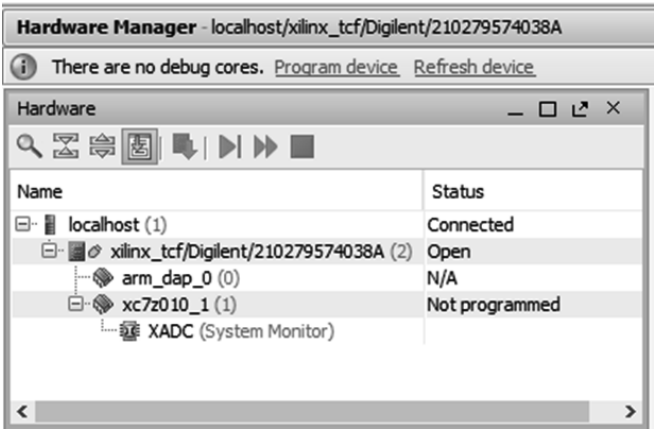
4.6.1. FPGA áramkörnek a rendszerhez való csatolása

Jelen esetben a *Zybo* fejlesztőkártyát az USB porton kapcsoljuk a számítógéphez, majd kapcsolódunk az FPGA áramkörre az alábbi rajzon megjelölt *Auto Connect* gombra kattintva (4.12. ábra).

Ha sikeres a kapcsolódás a 4.13. ábrának megfelelően, a rendszer felismeri a fejlesztőlapot, az FPGA áramkör típusát, valamint az FPGA áramkörön belül a *Xilinx Analog to Digital Converter* (XADC) analóg-digitális átalakítót, amelynek segítségével monitorizálni lehet például az áramkör hőmérsékletét.



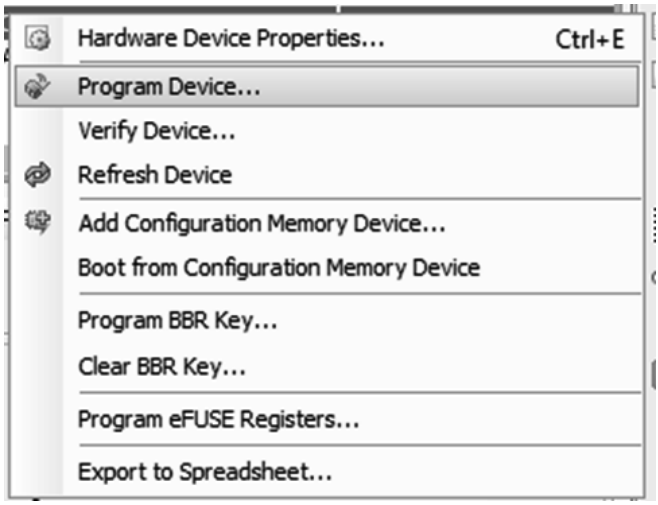
4.12. ábra. Hardware Manager, automatikus kapcsolódás az FPGA áramkörre



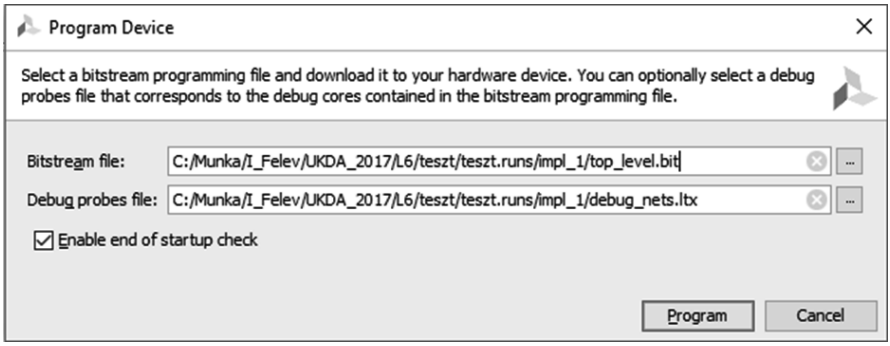
4.13. ábra. Hardware Manager sikeres kapcsolódása

A következő lépésben fel kell programozni az FPGA áramkört *Program Device* utasítással (4.14. ábra). Az FPGA áramkör típusára kattintva (jobb klikk) a gyorsmenüből is ki lehet választani a *Program Device* parancsot.

Programozáskor meg kell határozni a konfigurációs bit állományt, amelyet fel szeretnénk tölteni az FPGA áramkörre, valamint a tesztelésre kijelölt jeleket tartalmazó *ltx (debug_nets.ltx)* állományt (4.15. ábra).

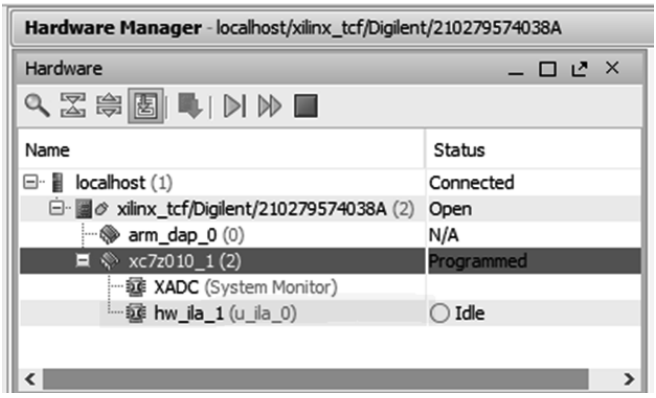


4.14. ábra. FPGA áramkör programozása



4.15. ábra. A konfigurációs állomány és a tesztelendő jeleket tartalmazó állományok kiválasztása

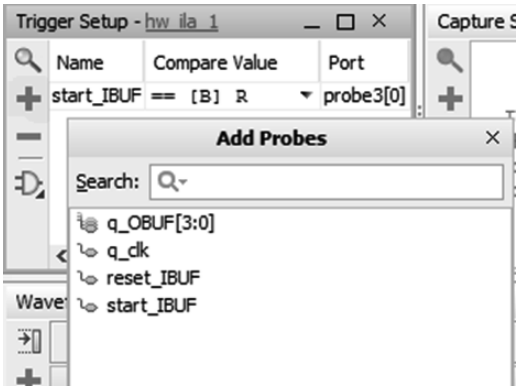
Az FPGA áramkör programozását követően, ha helyesen konfiguráltuk tesztelésre a tervet, az FPGA áramkör alatt elérhetők a tesztelendő/tesztelhető modulok (4.16. ábra).



4.16. ábra. Tesztelhető modulok megjelenítése

4.6.2. Az ILA tesztmodul trigger módjának beállítása, illetve a capture control beállításai

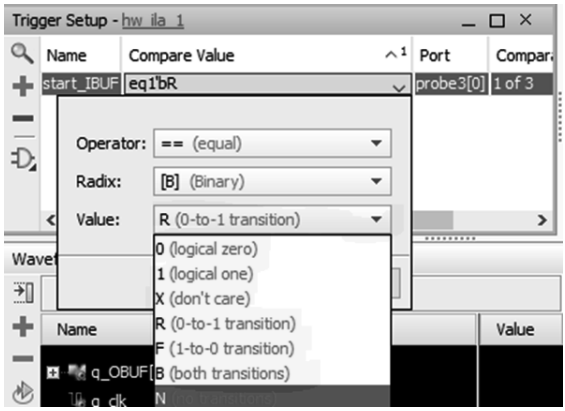
Ki kell választani, hogy mely események szeretnénk, hogy élesítsék, triggereljék az ILA tesztmodult, valamint milyen feltételek esetében történik mintavételezés. Az ILA modult aktiváló jelek a *Trigger Setup* panelen belül konfigurálhatók (4.17. ábra).



4.17. ábra. Élesítő jelek kiválasztása

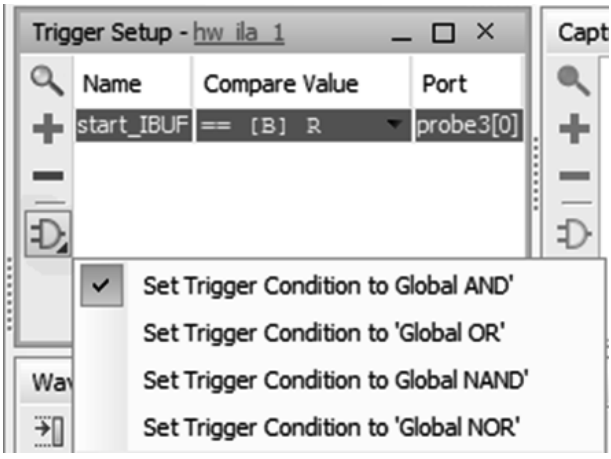
A tesztelésre szánt jelek kijelölésekor meg kell határozni, hogy milyen formában szeretnénk alkalmazni a tesztjelet: *Trigger, Data* vagy *Trigger and Data*. A *+ ADD Probes* gombra kattintva a megjelenített listából ki

lehet választani azokat a jeleket, amelyeket szeretnénk, hogy élesítsék az ILA modult, valamint milyen esetben történjen az élesítés: logikai 0 szint, logikai 1 szint, felfutó élre, lefutó élre vagy mindkét élre (lefutó, felfutó) (4.18. ábra).



4.18. ábra. Triggerfeltételek beállítása

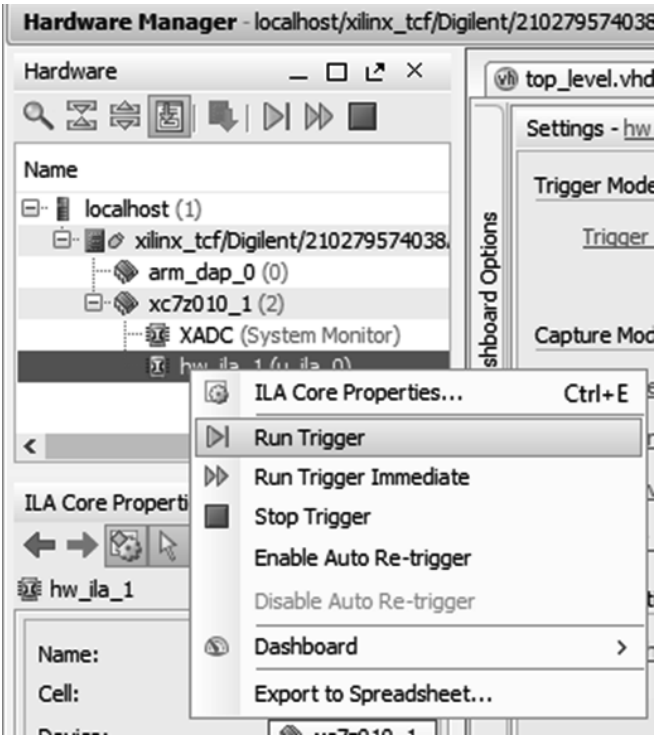
A *Set trigger* condition gombra kattintva, abban az esetben, ha több jel kombinációjával szeretnénk aktiválni az ILA modult, meg lehet határozni, hogy milyen logika szerint származtatjuk az aktiváló eseményt (4.19. ábra). A következő lépésben kezdődhet az adatgyűjtés.



4.19. ábra. Több triggerjel közti logika kiválasztása

4.6.3. A mérés elkezdése az ILA modulon IDLE állapotban

Az adatgyűjtési szakasz fázisai:
- A mérőmodul készenléti állapotban várakozik a mérés elkezdésére.
- *Wait for trigger*. A mérőmodul működik és várakozik, hogy az élesítő események bekövetkezzenek.
- *Post-trigger*. Teljesültek az élesítő feltételek és elkezdődik az adatgyűjtés.
- *Full*. Az adatok meg vannak jelenítve a *Waveform* ablakban.



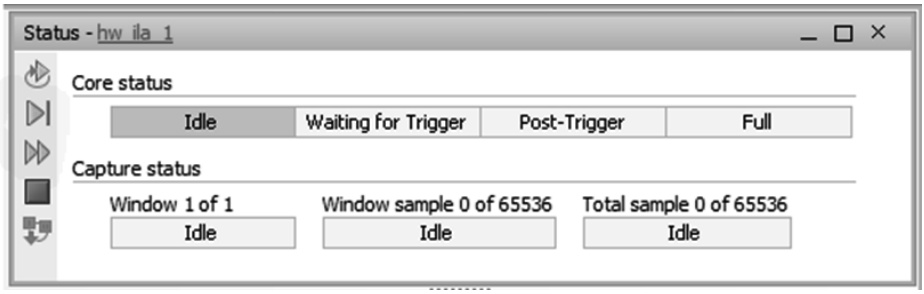
4.20. ábra. Mérési folyamat indítása

Az ILA modul *idle* készenléti állapotban várja az adatgyűjtés engedélyezését. A mérés indítására több lehetőség van:

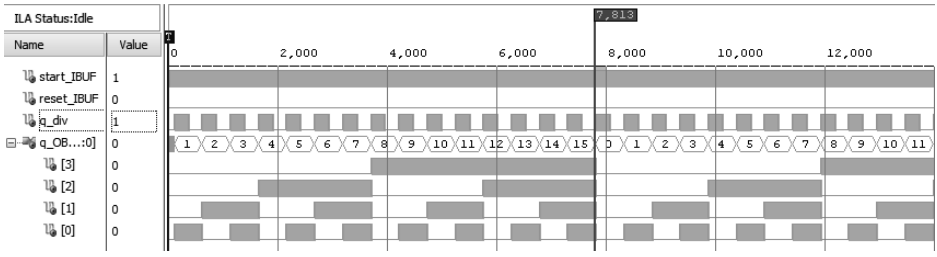
1. a hardware manager részben jobb klikk a *hw_ila_1 modulra* (4.20. ábra), majd *Run Trigger* vagy *Run Trigger Immediate*. Az *Enable*

Auto Re-trigger opció bekapcsolásával automatikusan újraindul az ILA modul.

2. *Status* vagy *waveform* panelekből a *Run Trigger* vagy *Run Trigger Immediate* parancsokra kattintva (4.21. ábra). A *Run Trigger Immediate* esetben azonnal megtörténik a mintavételezés, míg a *Run Trigger* esetben várja, hogy teljesüljön az élesítő feltétel. A *Status* (4.21. ábra) és *Waveform* (4.22. ábra) ablakokból *Toggle Auto Retrigger* üzemmód opcióval lehet engedélyezni az újraélesítést.



4.21. ábra. Mérés állapotának áttekintése



4.22. ábra. Adatok megjelenítése a tesztelés során

Az adatgyűjtést követően az adatok meg vannak jelenítve a *Waveform* ablakban. A puffer több kisebb méretű ablakra osztható, mindenik ablak a saját trigger markerével. Az alábbi példában 4 ablakban történik a mintavételezés (4.22. ábra).

5. fejezet

Véges állapotú automata adatúttal

Ebben a részben az adatutat is tartalmazó komplex állapotgép VHDL hardverleíró nyelven való megvalósítását tárgyaljuk. Az adatutas véges állapotú állapotgépek egyaránt alkalmazhatók egyszerű és komplex műveleteknek az elvégzésére. A véges állapotú állapotgép mellett a számításokat tartalmazó adatutakat is tartalmazza.

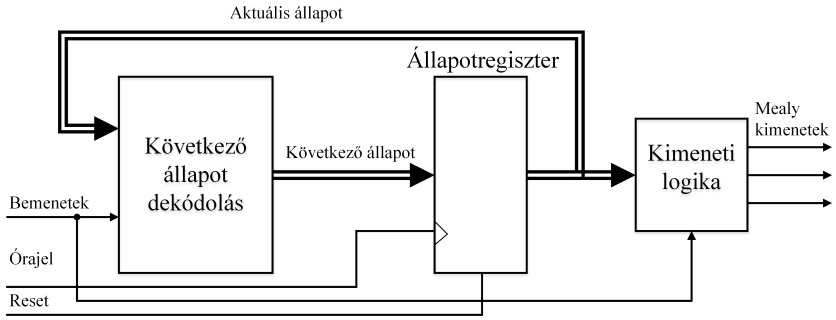
5.1. Bevezető

Az RTL alapú tervezés magába foglalja a tervezett rendszer viselkedésének a leírását, mint órajelre szinkronizált regiszterek közötti adatátvitel. Két alap véges állapotú állapotgépet különböztetünk meg:

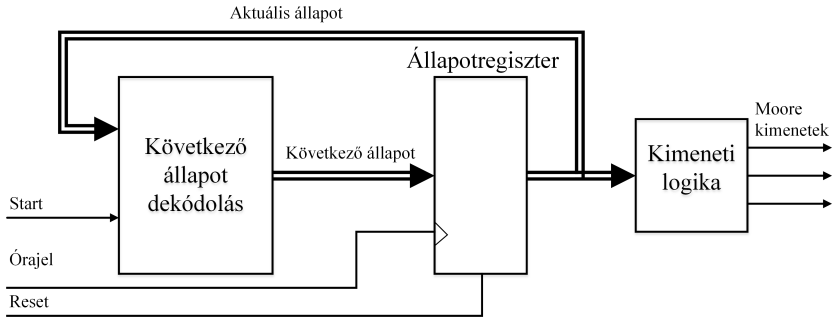
- Mealy: a kimeneteket az aktuális állapot és a bemenetek határozzák meg (5.1. ábra).
- Moore: a kimenetek csak az aktuális állapotoktól függnek (5.2. ábra).

A véges állapotú állapotgép a következő alrészeket tartalmazza:

- állapotregiszter: az aktuális állapot tárolására,
- a következő állapot dekodoló: az aktuális bemenet és állapot függvényében meghatározza a következő állapotot, amelybe a következő felfutó élre átlép az állapotgép,
- a kimeneti logika: az állapot (Moore automata), illetve az állapot és a bemenetek alapján (Mealy automata) meghatározza a kimenetet.



5.1. ábra. Mealy típusú állapotgép



5.2. ábra. Moore típusú állapotgép

A következőkben a Moore automatákra épülő adatutat is tartalmazó, komplex állapotgépet részletezzük.

5.2. Komplex állapotgép

A komplex állapotgép alkalmazása lehetővé teszi bonyolult feladatok egyszerű megvalósítását. Az adatutas véges állapotautomata két fő alegységre bontható:

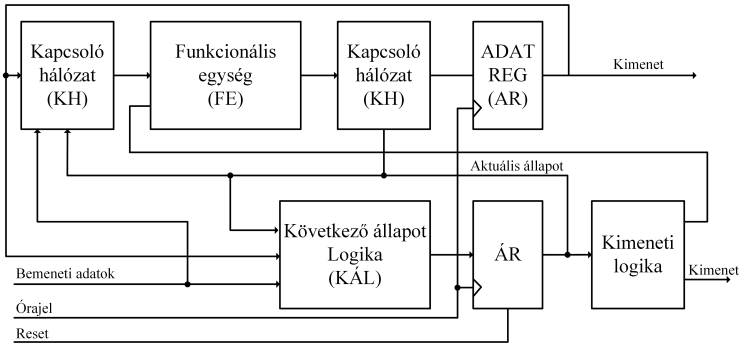
- tartalmaz egy sajátos vezérlőutat (control path) a regiszter műveletek megfelelő sorrendben való végrehajtására, amelyet egy véges állapotú állapotgép valósít meg,
- tartalmaz egy sajátos „adatutat” az összes elvárt regiszterműveletre (data path).

Adatút alegységei:
- adatregiszterek: az eredmények és részeredmények tárolására szolgálnak. A megoldandó algoritmusban minden egyes eredménynek és részeredménynek a tárolására egy-egy adatregiszter van alkalmazva,
- funkcionális egységek: a megvalósítandó algoritmusban azonosított műveletek vannak implementálva,
- kapcsoló hálózat (vagy hálózatok): az adatutak kiválasztásában játszik szerepet.

Lehetséges RT műveletek:
- $r <= 1$ konstans betöltése a regiszterbe
- $r <= r$ regiszter előbbi értékének megtartása
- $r <= r2 << 3$ eltolási művelet eredményének tárolása
- $r0 <= r1$ egy regiszter tartalmának egy célregiszterbe való átírása
- $n <= n - 1$ egy regiszter értékének dekrementálása vagy inkrementálása
- $y <= a + b + c + d$
- $s <= a2 + b2$ csak abban az esetben tekinthető regiszterműveletnek, ha a művelet egy órajel alatt elvégezhető (általában létezik egy előre elkészített szorzó áramkör)

A lényeges különbség egy algoritmusban használt változó és egy regiszterművelet között, hogy az RT művelet egy órajelre hajtódik végre.

Az 5.3. ábra az adatutat is tartalmazó komplex állapotgép tömbvázlatát szemlélteti, amelyen a KÁL a következő állapotdekodoló logika, az ÁR pedig állapotregiszter.



5.3. ábra. Adatutas véges állapotautomata tömbvázlata

Az elvégzendő műveletsorozatot a véges állapotú állapotgép vezérli. A következő állapot függ az aktuális állapottól, a pillanatnyi bemenettől és az adatregiszterekben eltárolt eredménytől (részeredménytől).

A reset jelre az állapotgép visszaáll a kezdeti állapotra. Az aktuális bemenetek és változók értékei alapján a következő állapotdekodoló meghatározza a következő állapotot.

Az adatút részen a kapcsoló hálózatok (bal oldali) meghatározzák, milyen adatok kerüljenek a funkcionális egységre. A funkcionális egység több műveletet tartalmazhat. A jobb oldali kapcsolóhálózat meghatározza, hogy melyik művelet eredményét kapcsolja az adatregiszter bemenetére.

A következő felfutó órajelre, amely szinkronban vezérli az állapotregisztert és az adatregisztert, az állapotregiszterbe beíródik a következő állapotsínről a következő állapot (ezáltal az állapotgép átvált az új állapotba), az adatregiszterbe pedig beíródik az új eredmény.

A funkcionális egységek a pillanatnyi bemeneti adatok és az adatregiszterek értékeinek megfelelően elvégzik a műveleteket. A kapcsolóhálózat az automata pillanatnyi állapotának függvényében kiválasztja az adatregiszterben tárolandó eredményt [2].

5.2.1. A komplex állapotgép tervezésének lépései

A komplex állapotgép kiemelendő tervezési lépései a következőek:
- azonosítjuk az elvégzendő RT műveleteket,
- azonosítjuk az adatfüggőségeket,
- azonosítjuk a célregisztereket (eredmény vagy részeredmény tárolására),
- az RT műveletek csoportosítása a célregiszter függvényében, az adatfüggőségeket figyelembe véve, minden fázisban meghatározzuk az elvégzendő műveletet (aritmetikai, logikai, forgatás, léptetés stb.).

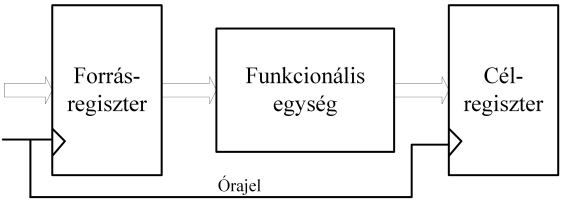
Minden csoportra:
- meghatározzuk a célregiszter méretét,
- minden típusú RT műveletre meg kell tervezni a kombinációs áramkört,
- ha a célregiszterhez több RT művelet tartozik, abban az esetben multiplexereket kell az adatregiszter elé iktatni a megfelelő adatút elbírálására,
- a megfelelő kapcsoló áramkörök (kapcsolóhálózat) megtervezése a statusjelek generálására.

A kapcsolóhálózat és a funkcionális egységek megvalósíthatóak konkurens VHDL utasításokkal. Az állapotregiszter és adatregiszterek folyamatként implementálhatók.

5.2.2. Adatút megvalósításának szemléltetése

A következőkben példákon keresztül szemléltetjük a regiszterműveleteket. Az 5.4 ábrán egy egyszerű regiszterművelet van szemléltetve, amely a következő alegységeket tartalmazza:

- a forrásregiszter tartalmazza a bemenő adatot, amelyen egy műveletet (funkcionális egység) végzünk el,
- az első órajelre az új adat beíródik a forrásregiszterbe,
- a kombinációs áramkör kiszámolja az f logikai függvényt,
- a következő új órajelre az eredmény a célregiszterbe, egy új adat pedig a forrásregiszterbe íródik.



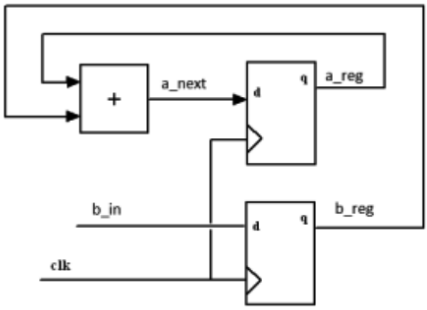
5.4. ábra. Regiszterművelet szemléltetése

Az 5.5. ábrán egy egyszerű adatutas példát szemléltetünk két adatúttal, mindkét adatút egy adott típusú műveletet valósít meg. Az 5.6. ábra a példa idődiagramját mutatja be.

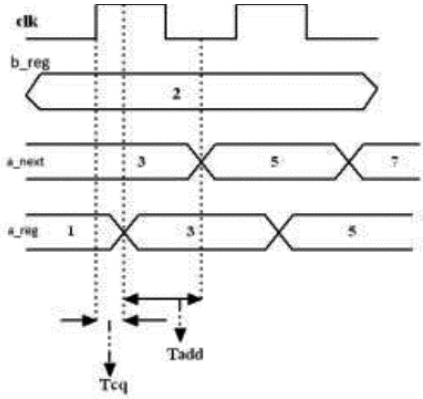
A *clk* órajel felfutó élére mindkét regiszterbe beíródik a d bemenetről az adat és a Tcq késéssel megjelenik a regiszterek kimenetén. A $Tadd$ idő elteltével az összegzőbe bemenő két adat összege megjelenik az összeadó áramkör kimenetén (a_next). A következő felfutó órajelre az új eredmény beíródik az a_reg regiszterbe, ab_reg regiszterbe pedig bekerül a b_in bemenetről az adat.

Az 5.7. ábrán az $r1\_reg$ köré épített adatút a következő műveleteket hajtja végre:

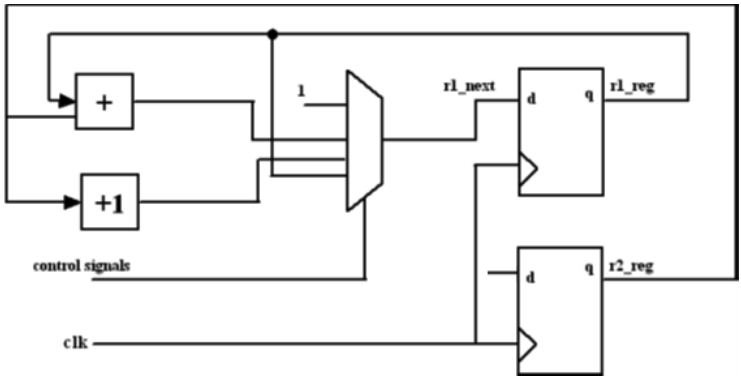
- $r1\_reg <= 1$,
- $r1\_reg <= r1\_reg + r2\_reg$
- $r1\_reg <= r2\_reg + 1$
- $r1\_reg <= r1\_reg$



5.5. ábra. Adatút, két adatregiszterrel [2]



5.6. ábra. Idődiagram [2]



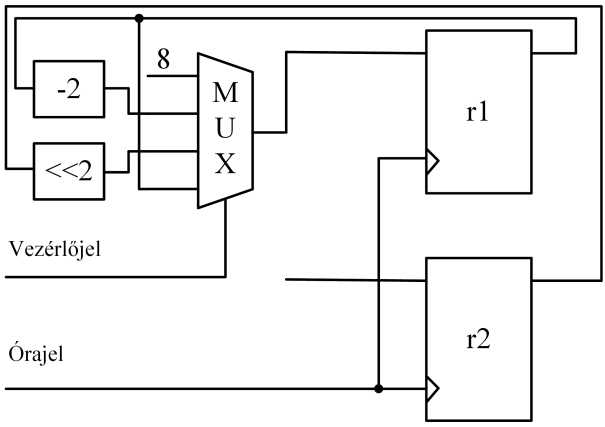
5.7. ábra. Adatút, példa

Azt, hogy éppen milyen műveletet hajtunk végre, a multiplexer áramkörnek a szelekciós bemenete határozza meg. A multiplexer szelekciós bemenetét egy véges állapotú állapotgéppel vezérelve, meghatározhatjuk a műveletek sorrendjét.

Ha az elvégezendő műveletek párhuzamosíthatóak, akkor mindenik változónak megfelelően egy-egy regisztert alkalmazunk, és a regiszterek köré kiépítjük a megfelelő adatutat. Az adatút kapcsolgatásával cseréljük az éppen elvégzett műveletet.

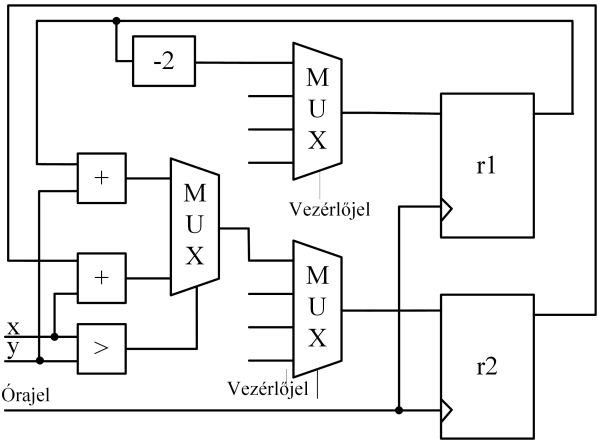
Az 5.8. ábrán az $r1$ regiszter négy típusú művelet eredményét tárolja. A négy állapotnak megfelelő műveletek a következőek:

- $r1 <= 8$
- $r1 <= r1 - 2$
- $r1 <= r2 << 2$
- $r1 <= r1$



5.8. ábra. Adatút, példa

Az 5.9. ábrán annak függvényében, hogy az x vagy y bemenet nagyobb, más műveletet végez az $r2$ regiszter köré tervezett adatút.



5.9. ábra. Elágazás

5.3. Adatfeldolgozást végző komplex állapotgép tervezése

Tervezzünk VHDL hardverleíró nyelven egy adatutas automatát, amely kiszámolja a két vektor skalárszorzatát. A modul bemenetei:

- N – vektorok elemeinek a száma,
- *start* – bemeneti jel (logikai '1-re állítva indítjuk a számításokat),
- *reset* – kezdeti állapotba kapcsoljuk az automatát,
- w_i,x_i – bemeneti adatok, az i-dik lépésben kerülnek a rendszer bemenetére; az x_i és w_i tárolhatóak például egy BRAM memóriában.

Az elvégzendő műveletet a következő képlet szemlélteti:

$$y = w_1x_1 + w_2x_2 + \cdots + w_Nx_N \tag{5.1}$$

$$y = \sum_{i=0}^{N-1} w_ix_i \tag{5.2}$$

Felírjuk rekurzívan a vektorszorzat egyenletét:

$$y = y + w_ix_i, i = 0 \cdots N-1 \tag{5.3}$$

Elvileg mindegy, hogy a ciklusváltozót inkrementáljuk vagy dekrementáljuk. Gyakorlatilag viszont egyszerűbb és kevesebb erőforrást igényel, ha csökkentjük a ciklusváltozó értékét, mivel csökkentés során mindig nullát kell ellenőrizni.

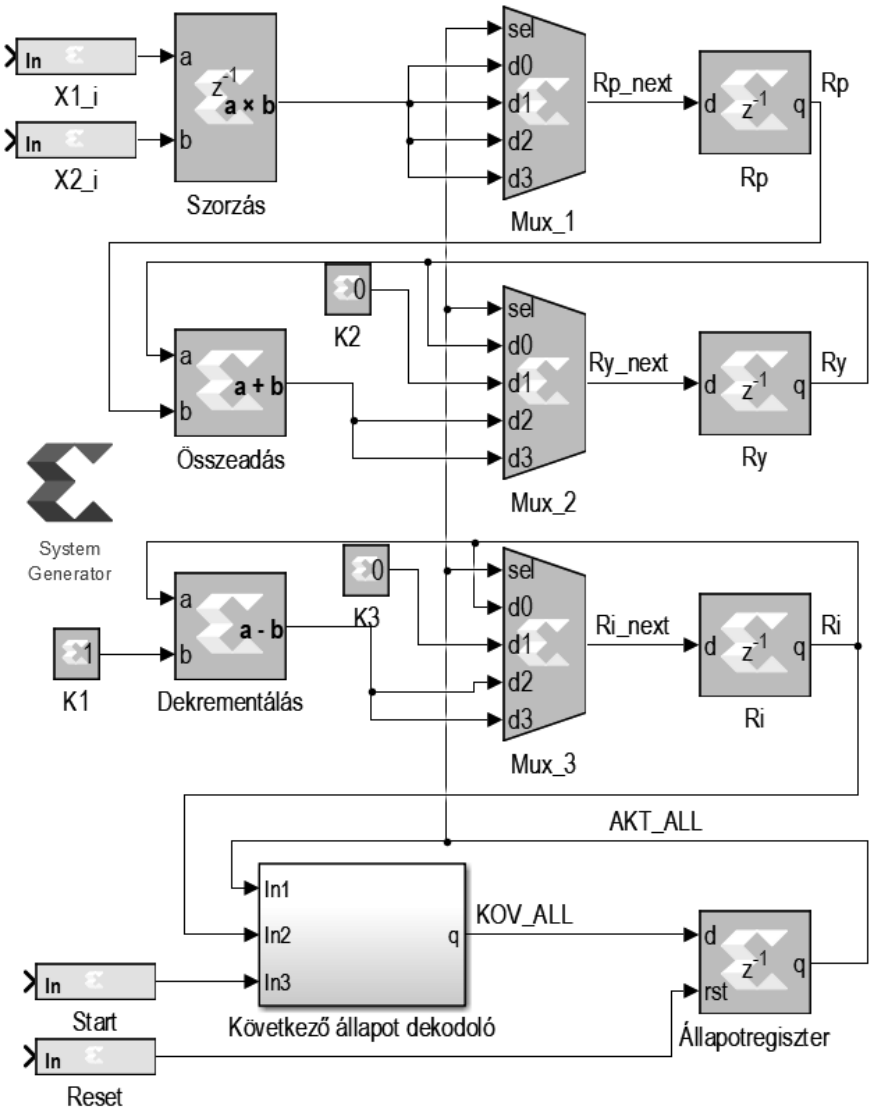
Azonosítva az elvégzendő RT műveleteket, a következő műveletek implementálására van szükség:

- $y = 0$,$i = 0$ kezdőértékek inicializálása nullával
- $p_i = w_ix_i$ w_i és x_i értékek szorzása
- $y = y + w_ix_i$ összeadás, rekurzívan a szorzás eredményének hozzáadása a végeredményhez
- $i = i - 1$ ciklusváltozó dekrementálása

Az eredmény, részeredmények tárolására a következő regisztereket alkalmazzuk:

- *Ry* – tároljuk a végeredményt
- *Rp* – tároljuk a szorzás eredményét
- *Ri* – ciklusváltozó tárolására szolgáló regiszter

A skaláris vektorszorzat számítására szolgáló adatutas automata tömbvázlata az 5.10. ábra alapján tanulmányozható.



5.10. ábra. Vektorszorzat komplex állapotgépként való megvalósításának részletes terve

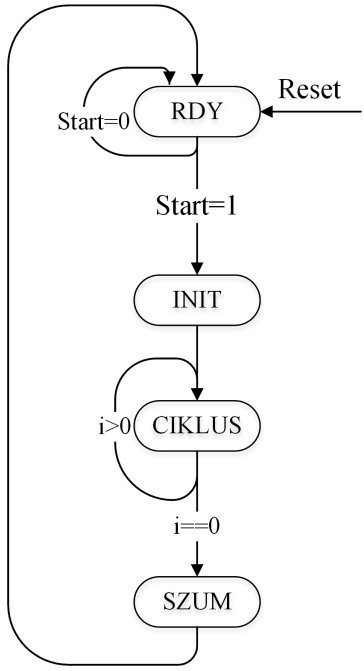
Figyelembe véve az egyes adatok közötti függőségeket, a különböző fázisokban elvégzendő műveletek az 5.1 táblázat szerint rendszerezhetőek.

5.1. táblázat. Különböző fázisokban elvégzendő műveletek

Állapot	Ri regiszter	Ry regiszter	Rp regiszter
RDY	$Ri \leftarrow Ri$	$Ry \leftarrow Ry$	x
$INIT$	$Ri \leftarrow N$	$Ry \leftarrow 0$	$Rp \leftarrow w_1 x_1$
$CIKLUS$	$Ri \leftarrow Ri + 1$	$Ry \leftarrow Ry + Rp$	$Rp \leftarrow w_i x_i$
$SZUM$	$Ri \leftarrow Ri$	$Ry \leftarrow Ry + Rp$	x

A RDY készenléti állapotban megtartjuk az előbbi számítás eredményeit, és az automata készen áll egy új számítási ciklus elvégzésére. Az $INIT$ állapotban a megfelelő kezdeti értékekkel inicializáljuk a regisztereket. Az összeadás egy ciklussal el van maradva a szorzáshoz képest, tehát az INIT fázisban elvégezhetjük a $w_1 x_1$ szorzást. A $CIKLUS$ állapotban inkrementáljuk az R_i értékét, az Ry regiszterhez hozzáadjuk az Rp-ben előbbi lépésben tárolt szorzatot, és elvégezzük a $w_i x_i$ szorzást.

Az automata állapotdiagramja az 5.11. ábrán van szemléltetve.



5.11. ábra. A vektorszorzatot megvalósító áramkör állapotdiagramja

A $Start = 1$-re a RDY állapotból átvált az $INIT$-be, amelyben megtörténik a regiszterek kezdeti értékekkel való inicializálása Ry, illetve Ri. Az $INIT$ állapotból a következő órajelre átlép a $CIKLUS$ állapotba. Ebben az állapotban a ciklusváltozó értéke minden órajelre csökken. Addig marad ebben az állapotban, amíg $Ri < N$, majd átlép a $SZUM$ állapotba és elvégzi az utolsó összeadást. A $SZUM$ állapotból visszalép a kezdő RDY állapotba, az eredményregiszterben tárolva van az eredmény mindaddig, amíg egy új számítási ciklust el nem kezdünk.

5.4. Két vektor skalárszorzatát megvalósító adatutas automata VHDL specifikálása

A következő programrész tartalmazza a vektorszorzat adatutas automataként való VHDL alapú megvalósítását.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
USE ieee.std_logic_arith.all;
-- entitás rész: bemeneti port jelek deklarálása és
-- típusának meghatározása
entity FSMD is
Port (
   src_clk : in STD_LOGIC;
   reset : in STD_LOGIC;
   start : in STD_LOGIC;
   --N vektor elemeinek a száma
   N : in STD_LOGIC_VECTOR (7 downto 0);
   x1 : in STD_LOGIC_VECTOR (15 downto 0);  -- bemeneti adat
   x2 : in STD_LOGIC_VECTOR (15 downto 0); --bemeneti adat
   y : out STD_LOGIC_VECTOR (31 downto 0)); -- kimeneti jel
end FSMD;

architecture Behavioral of FSMD is
-- az automata aktuális állapotának és következő állapotának
-- tárolására szolgáló adattípus és signalok deklarálása
-- az automata állapotainak tárolására létrehozott felsorolás
-- típus
type allapot_tipus is (RDY, INIT, CIKLUS, SZUM);
-- a létrehozott típus alapján az automata aktuális és
-- következő állapotának tárolására létrehozott jel
signal akt_all, kov_all : allapot_tipus;
```

```
--az adatútban alkalmazott jelek deklarálása
-- RI: index tárolására szolgáló jel
-- RY: részösszeg tárolására szolgáló jel
-- RP: szorzat tárolására szolgáló jel
signal RI, RI_next : std_logic_vector(7 downto 0);
signal RY, RY_next : std_logic_vector(31 downto 0);
signal RP, RP_next : std_logic_vector(31 downto 0);

begin
--állapotregiszter megvalósítása
AR : process(src_clk,reset) --élesítő jelek
begin
   if reset='1' then  --reset feltétel ellenőrzése
      akt_all<=RDY;  --kezdeti állapot beállítása
   -- felfutó óraél detektálása
   elsif src_clk'event and src_clk='1' then
      akt_all<=kov_all; --felfutó óraélre, az új állapot
      --beírása az állapotregiszterbe
   end if;
end process AR;
--következő állapotlogika megvalósítása
KAL : process(akt_all, start, RI)
begin
-- a pillanatnyi (akt_all) alapján a megfelelő leágazás
-- kiválasztása, majd a következő állapot meghatározása
-- az aktuális állapot és a modul bemeneti jelei alapján
case akt_all is
   when RDY =>
      if start='1' then
         --ha start, átlépés az INIT állapotba
         kov_all<=INIT;
      else
         --egyébként vissza a kezdeti állapotba
         kov_all<=RDY;
      end if;
   when INIT =>
      --feltétel nélküli átlépés a CIKLUS állapotba
      kov_all<=CIKLUS;
   when CIKLUS =>
      if RI>1 then
         -- ha nem történt meg a szorzás az összes
         -- elemre maradunk a CIKLUS állapotban
         kov_all<=CIKLUS;
      else
         -- egyébként átlépés a SZUM állapotba
         -- az utolsó összegzés elvégzése céljából
```

```
         kov_all<=SZUM;
      end if;
   WHEN SZUM=>
      kov_all<=RDY;  --feltétel nélkül visszaugrás
      --a kezdeti állapotba
end case;
end process KAL;
-- a kapcsoló hálózatok megvalósítása with select
-- when utasítással és az elvégzendő műveletek
    implementálása.
with akt_all select
-- index regiszter körüli kapcsolóhálózat kialakítása
   RI_next<=(others=>'0') when RDY,
         N-1 when INIT,
         RI-1 when CIKLUS,
         RI when SZUM;

--RP regiszter körüli kapcsolóhálózat kialakítása
with akt_all select
   RP_next<=(others=>'0') when RDY,
      x1*x2 when INIT,
      x1*x2 when CIKLUS,
      (others=>'0') when SZUM;

--RY regiszter körüli kapcsolóhálózat kialakítása
with akt_all select
   RY_next<=RY when RDY,
      (others=>'0') when INIT,
      RY+RP when CIKLUS,
      RY+RP when SZUM;

--RI adatregiszterek megvalósítása
ADAT_RI : process(src_clk,reset)
begin
   if src_clk'event and src_clk='1' then
      RI<=RI_next;
   end if;
end process ADAT_RI;

--RP adatregiszterek megvalósítása
ADAT_RP : process(src_clk,reset)
begin
   if src_clk'event and src_clk='1' then
      RP<=RP_next;
   end if;
end process ADAT_RP;
```

```
--RY adatregiszterek megvalósítása
ADAT_RY : process(src_clk,reset)
begin
   if src_clk'event and src_clk='1' then
      RY<=RY_next;
   end if;
end process ADAT_RY;

y<=RY; --RY regiszter tartalmának a kivezetése
-- az y kimeneti jelre
end Behavioral;
```

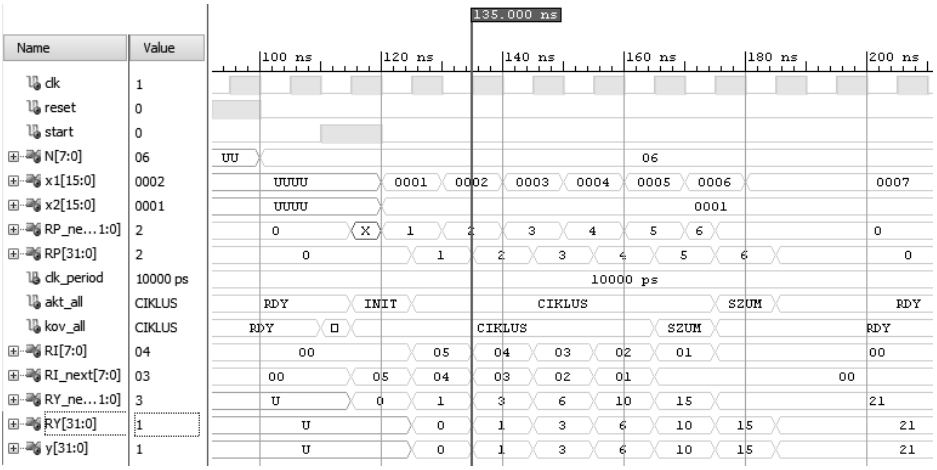
A fennebb bemutatott modul szimulációjához szükséges bemeneti jelek létrehozására szolgáló VHDL programrész:

```
-- könyvtár deklarálás
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.all;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
-- USE ieee.numeric_std.ALL;
-- a szimulációs modul nem tartalmaz bemeneteket, a modul
-- feladata, hogy a szimulálandó alegység számára létrehozza
-- a megfelelő jelszekvenciát.
ENTITY FSMD_tb IS
END FSMD_tb;

ARCHITECTURE behavior OF FSMD_tb IS
-- Tesztelendő modul komponensként való deklarálása
COMPONENT FSMD
Port (
   src_clk : in STD_LOGIC;
   reset : in STD_LOGIC;
   start : in STD_LOGIC;
   N : in STD_LOGIC_VECTOR (7 downto 0);
   x1 : in STD_LOGIC_VECTOR (15 downto 0);
   x2 : in STD_LOGIC_VECTOR (15 downto 0);
   y : out STD_LOGIC_VECTOR (31 downto 0));
END COMPONENT;
-- signalok deklarálása: a szimulálandó modul mindenik
    jelének
-- deklarálunk egy signalt. A jelek segítségével a megfelelő
-- jelsorozatot rákapcsoljuk a tanulmányozandó modul jeleire.
-- bemeneti jelek
```

```
signal clk : STD_LOGIC;
signal reset :  STD_LOGIC;
signal start :  STD_LOGIC;
signal N :  STD_LOGIC_VECTOR (7 downto 0);
signal x1 : STD_LOGIC_VECTOR (15 downto 0);
signal x2 : STD_LOGIC_VECTOR (15 downto 0);

--kimeneti jelek
signal y : std_logic_vector(31 downto 0);
-- Órajel periódusának  meghatározása [ns]
constant clk_period : time := 10 ns;

BEGIN
-- szimulálandó modul példányosítása, és a modul port
-- jeleire a megfelelő signal jelek kapcsolása.
-- baloldalon: a modul portjelei
-- jobboldalon: az Architecture részben meghatározott jelek
-- tesztelt áramkör példányosítása (UUT)
   uut: FSMD PORT MAP (
   src_clk => clk,
   reset => reset,
   start => start,
   N => N,
   x1 => x1,
   x2 => x2,
   y => y);

-- órajel generálása, ha a szimulálandó rendszerben több
-- órajelet kell alkalmazni, minden órajelet az alábbi
-- minta alapján lehet létrehozni.
clk_process :process
begin
   clk <= '0';  -- órajel logikai nullára való kapcsolása
wait for clk_period/2;  -- félperiódusnyi várakozás
   clk <= '1'; -- órajel logikai egyesre való kapcsolása
wait for clk_period/2; -- félperiódusnyi várakozás
end process;

-- a modul szimulációjához a megfelelő jelek létrehozása
-- tesztjelek generálását megvalósító folyamat
stim_proc: process
begin
   -- reset jel logikai egyesre való kapcsolása.
   reset<='1';
   start<='0';
   wait for 100 ns; -- várakozás 100 ns-ig
```

```
   reset<='0';     -- reset felszabadítása
   -- tömbök elemeinek meghatározása
   N<=conv_std_logic_vector(6,8);
   wait for clk_period; -- várakozás egy óraciklusra
   start<='1';
   wait for clk_period;   -- várakozás egy óraciklusra
   -- első bemeneti adatpárok bevitele
   start<='0'; -- egy számolási ciklust követően ne induljon
       el
   -- a következő ciklus
   x1<= conv_std_logic_vector(1,16);
   x2<= conv_std_logic_vector(1,16);
   wait for clk_period;

   -- adatpárok bevitele
   x1<= conv_std_logic_vector(2,16);
   x2<= conv_std_logic_vector(1,16);
   wait for clk_period;

   -- adatpárok bevitele
   x1<= conv_std_logic_vector(3,16);
   x2<= conv_std_logic_vector(1,16);
   wait for clk_period;

   -- adatpárok bevitele
   x1<= conv_std_logic_vector(4,16);
   x2<= conv_std_logic_vector(1,16);
   wait for clk_period;

   -- adatpárok bevitele
   x1<= conv_std_logic_vector(5,16);
   x2<= conv_std_logic_vector(1,16);
   wait for clk_period;

   -- adatpárok bevitele
   x1<= conv_std_logic_vector(6,16);
   x2<= conv_std_logic_vector(1,16);
   wait for clk_period;

   x1<= conv_std_logic_vector(7,16);
   x2<= conv_std_logic_vector(1,16);
   -- további jelek létrehozása
   wait;
end process;
END;
```

Az 5.12. ábra a vektorszorzat megvalósító komplex állapotgép szimulációs eredményét szemlélteti. A bemeneti tesztjeleknek megfelelően, 100 ns-ig a *reset* jel logikai egyesre kapcsolásával kezdeti állapotba (RDY) kapcsoljuk a rendszert. A következő óraciklusra a reset jel logikai nullásra van visszaállítva, a *start* jel egyelőre marad logikai nullás szinten.



5.12. ábra. Vektorszorzatot megvalósító komplex állapotgép szimulációja

A következő órajelre a start jel átvált logikai egyesre, az állapotgép átlép az $INIT$ állapotba, és megtörténik a bemeneti vektor első elemének a beolvasása és az első elempár szorzási eredményének a kiszámolása. Az órajel felfutó élére a szorzás eredménye el van mentve az Rp regiszterbe, és az állapotgép átlép a $CIKLUS$ állapotba. A $CIKLUS$ állapotban hozzáadja az Ry regiszterhez az előbbi lépésben kapott szorzás eredményét és kiszámolja $N-1$ elempár szorzatát. Az utolsó elempár szorzási eredményét a $SZUM$ állapotba hozzáadja az Ry regiszterhez.

A $SZUM$ állapotot követően az állapotgép visszalép a RDY kezdeti állapotba, és várja egy következő ciklus elkezdését. A szimuláció szerint az RI_next, RY_next és RP_next jeleken azonnal megjelenik az eredmény, majd a következő órajel felfutó élére az említett jelekről elmentődik az eredmény az Ri, Ry és Rp regiszterekbe. A szimuláció alapján egy 7-ik elempárra és beviszünk egy értéket ($x1 = 7$), viszont, mivel az állapotgép visszatért a RDY állapotba, az RP_next jel Rp regiszter nullával van inicializálva a következő állapotdekodoló modul működésének megfelelően.

6. fejezet

PWM modul adatutas megvalósítása és szimulációja

A gyakorlat célja a Vivado tervezőeszköz alkalmazásának elsajátítása. Egy előre elkészített VHDL modult kell egy tervbe integrálni, majd a modul szimulációjához szükséges VHDL modult (jelgenerátort) elkészíteni.

6.1. Bevezető

A feladat egy impulzusszélesség modulációs (PWM) egység VHDL hardverleíró nyelven való megvalósítása. A PWM modul adatutas automatával van megvalósítva. A PWM modul előtt egy előosztó található, amely a rendszer fő órajelét leosztva hozza létre az impulzus szélesség moduláció egységnek szükséges alapórajelet.

A modul fontosabb portjelei a következők:

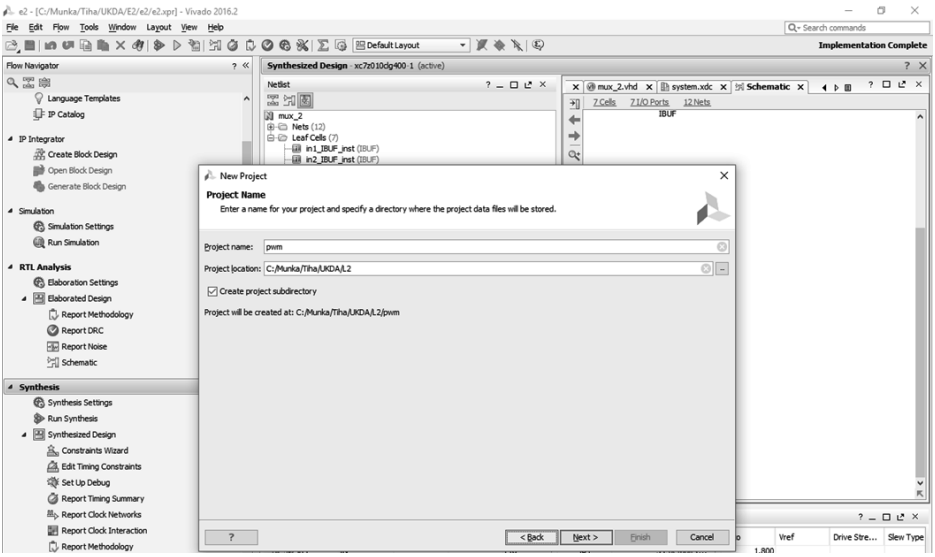
- *src_clk*: a rendszer órajele. A system Generator környezetben követelmény, hogy a clk helyett *src_clk* elnevezést alkalmazzunk.
- *src_ce*: órajel-engedélyező, szintén a modulnak System Generator típusú tervbe való integrálása szempontjából fontos. Közösen kell alkalmazni a *src_clk* órajellel
- *start*: a start jel logikai egyesre való kapcsolása engedélyezi a PWM modul működését.
- *reset*: kezdőállapotba kapcsolja az állapotautomatát (kezdőállapotnak megfelelő állapot RDY)

- *min_val*: bemeneten meghatározzuk minden periódusban, minimum hány órajelnyit lesz a generált PWM kimeneti jel logikai egyesben.
- *max_val*: meghatározza a PWM jel periódusát. A PWM modul előtt egy előosztó található. A PWM modul órajelét a rendszer fő órajele alapján származtatjuk, *div_val* értékkel leosztva a fő órajel frekvenciáját.
- *div_val*: előosztó, amelynek alkalmazásával leoszthtató a fejlesztőrendszer főórajelének frekvenciája, lehetővé téve a PWM modulnak, hogy az alapórajel frekvenciájánál kisebb frekvenciájú órajelen működjön. Az előosztó alkalmazásával szélesebb tartományban skálázható a jelgenerátor periódusa.

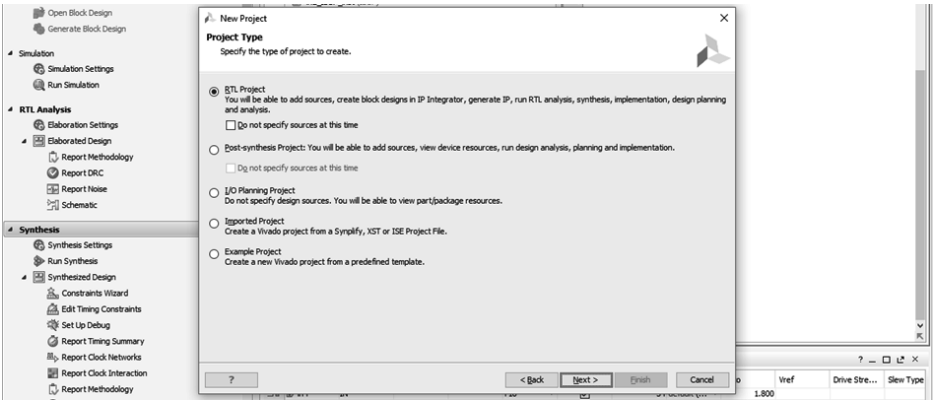
6.2. A PWM modul tervezőeszközbe való integrálásának fontosabb lépései

Első lépésben létre kell hozni egy új tervet a File-> New Project utasítással (6.1):

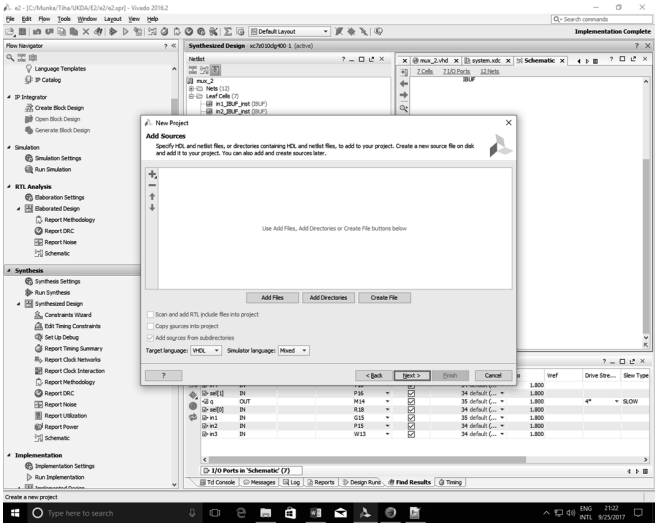
- Meg kell határozni a terv nevét.
- Meg kell adni a terv elérési útvonalát.
- RTL típusú terv kiválasztása (6.2. ábra).
- Forrásállományoknak a tervhez való csatolása (6.3). Jelen esetben szükséges egy VHDL modul létrehozása, amely tartalmazza a PWM egységet, valamint egy megkötésállomány (*constraint file*) létrehozása. A megkötésállományt egyelőre csak hozzáadjuk a tervhez, de nem fogjuk alkalmazni, szerkeszteni. Az új VHDL állományt a Create File paranccsal lehet létrehozni.
  Amint a 6.4 ábrán szemléltetve van, a tervhez csatolt állomány neve PWM, az állomány típusa pedig VHDL.
  Ha szükséges, újabb állományok csatolhatók a tervhez.
- *Add Existing IP* (optional), átugorjuk ezt a lépést. Abban az esetben alkalmazható, ha egy létező/előre elkészített IP magot szeretnénk a tervhez csatolni. (IP = Intellectual Property).
- *Add Constraints*, *Create Files*-ra kattintva létrehozunk egy új megkötésállományt. A file neve *system.xdc*, a kiterjesztést nem kell megadni.



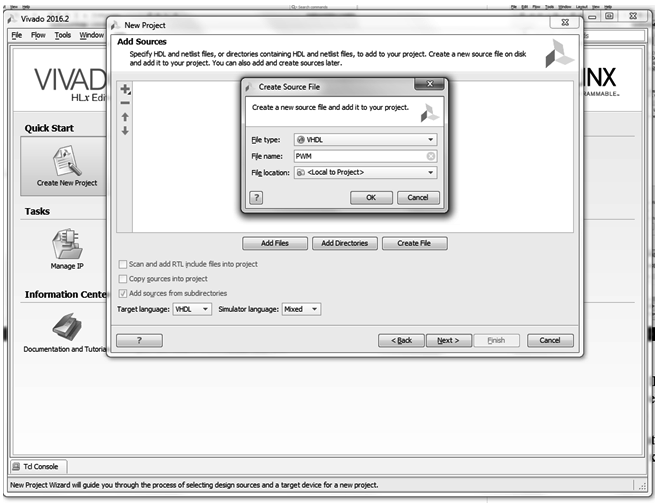
6.1. ábra. Új terv létrehozása



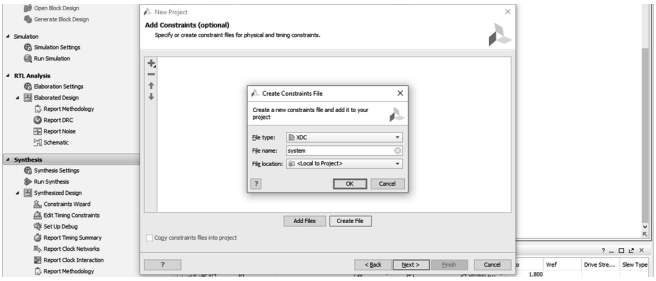
6.2. ábra. Terv típusának kiválasztása



6.3. ábra. Forrásállományok tervhez való csatolása

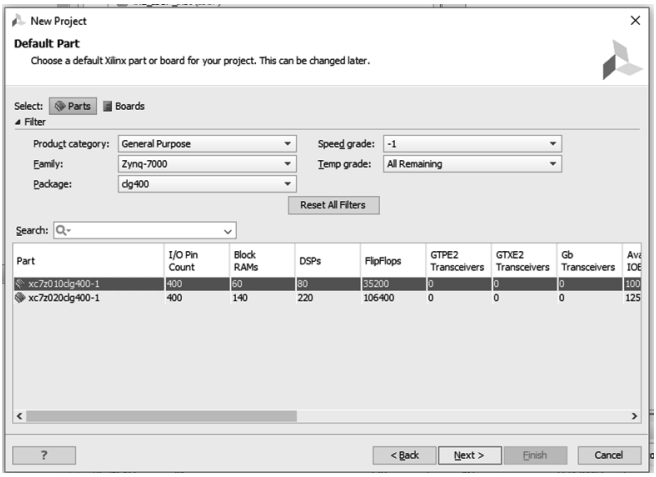


6.4. ábra. Állomány nevének meghatározása



6.5. ábra. Megkötésállomány létrehozása

– Az alkalmazandó FPGA áramkör típusának és paramétereinek kiválasztása.



6.6. ábra. Az alkalmazandó FPGA áramkör paraméterezése

A következő lépésben meghatározzuk a tervben a PWM modul portjeleit: *src_clk*, *src_ce*, *reset*, *STD_LOGIC*, *start*, *h*, *min_val*, *max_val*, *div_val*, *pwm_out*.

```
entity pwm is
Port (
   src_clk : in STD_LOGIC; -- bemeneti órajel
   src_ce : in STD_LOGIC; --órajel engedélyező]
   reset : in STD_LOGIC;  --reset jel
   start : in STD_LOGIC;  --modul indítására szolgáló jel
   -- h: kitöltési tényező, src_clk impulzusokban számolva
```

```
   h : in STD_LOGIC_VECTOR (31 downto 0);
   -- min_val: minimális kitöltési tényező, src_clk
   -- impulzusokban számolva
   min_val : in STD_LOGIC_VECTOR (31 downto 0);
   --max_val: a generálandó jel periódusa src_clk
       impulzusokban számolva
   max_val : in STD_LOGIC_VECTOR (31 downto 0);
   --div_val: előosztó
   div_val : in STD_LOGIC_VECTOR (9 downto 0);
   --pwm_out: impulzusszélességben modulált kimeneti jel
   pwm_out : out STD_LOGIC);
end pwm;
```

Deklaráljuk az alkalmazandó típusokat és jeleket (*signal*) az *Architecture* rész deklarációs részében, vagyis az *Architecture* és a *begin* között:

```
architecture Behavioral of pwm is
-- az állapotautomata állapotainak kódolására szolgáló típus
-- RDY: kezdeti állapot
-- INIT: inicializálási állapot
-- HIGH: pwm jel logikai egyesbe
-- LOW: pwm jel logikai nullába
type casee is(RDY,INIT,HIGH,LOW);
-- aktuális és következő állapotok tárolására szolgáló jelek
signal actual_case : casee;
signal next_case : casee;
signal pwm_sig, pwm_next_sig : STD_LOGIC;
signal counter, counter_next : STD_LOGIC_VECTOR(31 downto 0);
signal q_clk: STD_LOGIC;
```

Állapotregiszter implementációja:

```
State_R:process(q_clk,reset)
begin
   if reset = '1' then
      actual_case<= RDY;
   elsif (q_clk'event and q_clk='1') then
      actual_case<=next_case;
   end if;
end process State_R

--számláló regiszter megvalósítása
counter_R:process(q_clk,reset)
begin
   if reset = '1' then
      counter<=(others => '0');
   elsif (q_clk'event and q_clk='1') then
```

```
      counter<=counter_next;
   end if;
end process counter_R;

--pwm regiszter megvalósítása
pwm_R:process(q_clk,reset)
begin
   if reset = '1' then
      pwm_sig<='0';
   elsif (q_clk'event and q_clk='1') then
      pwm_sig<=pwm_next_sig;
   end if;
end process pwm_R;
```

Következő állapotlogika megvalósítása *CASE WHEN* utasítással:

```
next_case_log:process(actual_case, start,counter)
begin
case(actual_case) is
when RDY =>
   if start='1'  then
      next_case<=INIT; -- ha start='1' a következő állapot
          INIT
   else
      next_case<=RDY; -- egyébként marad a RDY kezdeti á
          llapotban
   end if;

when INIT =>
   if counter <min_val then
      next_case<=INIT;  -- ha a számláló nem érte el a
      -- min_val értéket, marad az INIT állapotba
   else
      next_case<=HIGH; --egyébként átlépés a HIGH állapotba
   end if;

when HIGH =>
   if counter<(h+min_val)  then
      -- ha a számláló nem érte el a min_val+h értéket
      -- marad a HIGH állapotba
      next_case<=HIGH;
   else
      next_case<=LOW; --egyébként átlépés a LOW állapotba
   end if;

when LOW =>
   if counter<max_val
```

```
   then
      next_case<=LOW; -- ha még van a periódusból marad a
          LOW állapotba
   else
      next_case<=RDY; --egyébként vissza a kezdeti állapotba
   end if;
end case;
end process next_case_log;
```

Adatutakat kiválasztó logika megvalósítása *WITH SELECT WHEN* utasítással:

```
-- számláló kiválasztó logikájának megvalósítása
WITH actual_case SELECT
   counter_next <= (others => '0')WHEN RDY,
               counter+1     WHEN others;

-- PWM regiszter köröli kiválasztólogika megvalósítása
WITH actual_case SELECT
   pwm_next_sig <= '0' WHEN RDY,
               '1' WHEN INIT,
               '1' WHEN HIGH,
               '0' WHEN LOW;
```

Előosztó modult megvalósító folyamat:

```
modulo:process(src_clk,reset)

variable x: integer range 1023 downto 0 := 0;
variable q: STD_LOGIC:= '0';

begin
   if src_clk'event and src_clk='1' then
      if x<div_value then
         x:=x+1;
         q:=q;
      else
         x:=0;
         q:=not(q);
      end if;
      q_clk<=q;
   end if;
end process modulo;

pwm_out<=pwm_next_sig;
```

6.3. Tesztelőállomány létrehozása

A library részben deklaráljuk a szükséges könyvtárakat/csomagokat. Jelen esetben a következő csomagra lesz szükség:

```
use IEEE.STD_LOGIC_SIGNED.ALL;
```

A Szimuláció során a következő fontosabb lépéseket kell végrehajtani:

1. A Szimulációs környezet beállítása. A Vivado 2016.2 nem generálja automatikusan a VHDL (vagy Verilog model) alapján a szimulációhoz szükséges sablont. Egy kiegészítő plugin programnak a telepítésére van szükség.
   - *Tools -> Xilinx Tcl Store*, majd telepíteni kell a *Design Utilities* plugint.
   - A *TCL Console* parancsablakban le kell futtani a következő parancsokat:

     ```
     xilinx::designutils::write_template -vhdl
         -return_string -stub
     ```

     valamint

     ```
     xilinx::designutils::write_template -vhdl
         -return_string -template
     ```

     A parancsok eredménye a modul példányosításához szükséges programrészek létrehozásához:

     ```
     entity pwm is
     port (
        -- Input Ports - Single Bit
        reset   :  in  std_logic;
        src_ce  :  in  std_logic;
        src_clk :  in  std_logic;
        start   :  in  std_logic;
        -- Input Ports - Busses
        div_val :  in  std_logic_vector(9 downto 0);
        h       :  in  std_logic_vector(31 downto 0);
        max_val :  in  std_logic_vector(31 downto 0);
        min_val :  in  std_logic_vector(31 downto 0);
        -- Output Ports - Single Bit
        pwm_out :  out  std_logic
        -- Output Ports - Busses
        -- InOut Ports - Single Bit
        -- InOut Ports - Busses
     );
     end entity pwm;
     ```

Az *entity* részből származtatjuk az alkatrészt, úgy, hogy az entity kifejezést kicseréljük *component*-re, és a bezáró sorban csak *end component* kifejezés marad. A példányosításhoz szükséges generált programrész módosítás nélkül alkalmazható.

```
pwm_inst: pwm
port map (
   -- Input Ports - Single Bit
   reset                => reset,
   src_ce               => src_ce,
   src_clk              => src_clk,
   start                => start,
   -- Input Ports - Busses
   div_val(9 downto 0)  => div_val(9 downto 0),
   h(31 downto 0)       => h(31 downto 0),
   max_val(31 downto 0) => max_val(31 downto 0),
   min_val(31 downto 0) => min_val(31 downto 0),
   -- Output Ports - Single Bit
   pwm_out              => pwm_out
   -- Output Ports - Busses
   -- InOut Ports - Single Bit
   -- InOut Ports - Busses
);
```

2. Szimulációs állomány, és azon belül a szimulációhoz szükséges alegységek létrehozása. A szimulációs állomány a következő fontosabb részeket tartalmazza:

   a) szimulálandó modult
      i. *component*
      ii. a példányosításhoz szükséges sablont
   b) az órajel generálása egy folyamatban van megvalósítva
   c) a stimulus jelek, amelyeket a megfelelő időpillanatban kell kapcsolni a modul bemenetére. A stimulus jelek generálására szekvenciális utasításokat lehet alkalmazni
   d) a stimulus jelek és a szimulálandó modul jeleinek összekapcsolásához szükséges jelek deklarálása. Az összekapcsoláshoz szükséges jelek egyszerűen létrehozhatók a szimulálandó modul portjeleit átszerkesztve, úgy, hogy megfeleljen a jeldeklarációnak
   e) szükséges könyvtárak deklarálása

A modulban szereplő jelek kiemelése:

```
src_clk : in STD_LOGIC;
src_ce : in STD_LOGIC;
reset : in STD_LOGIC;
start : in STD_LOGIC;
h : in STD_LOGIC_VECTOR (31 downto 0);
min_val : in STD_LOGIC_VECTOR (31 downto 0);
max_val : in STD_LOGIC_VECTOR (31 downto 0);
div_val : in STD_LOGIC_VECTOR (9 downto 0);
```

A *src_clk* jel deklarálására a portjelből származtatva a következőképpen valósítható meg:

- a port jel elé beírjuk a *signal* kulcsszót,
- töröljük jel irányát meghatározó kifejezést.

```
signal src_clk : std_logic;
```

Könyvtárdeklaráció

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if
    instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;
```

A szimulációs modul *entity* részének deklarálása.

Ebben az esetben nem kell megadni portjeleket. Azokat a jeleket, amelyeket a szimulálandó modulra kapcsolunk, a stimulus rész generálja.

```
entity sim_1 is
--  Port ( );
end sim_1;
```

Az *architecture* deklarációs részében a szimulációs állományban alkalmazott alegységek összekapcsolásához szükséges jelek, típusok deklarálása.

```
architecture Behavioral of sim_1 is
--az alegységek összekapcsolásához szükséges jelek
    (vezetékek)
signal src_clk : STD_LOGIC;
signal src_ce : STD_LOGIC;
signal reset : STD_LOGIC;
signal start : STD_LOGIC;
signal h : STD_LOGIC_VECTOR (31 downto 0);
signal min_val : STD_LOGIC_VECTOR (31 downto 0);
signal max_val : STD_LOGIC_VECTOR (31 downto 0);
signal div_val : STD_LOGIC_VECTOR (9 downto 0);
signal pwm_out : STD_LOGIC;

-- Clock period definitions / Órajel periódusának
    meghatározása
constant clk_period : time := 10 ns;
```

Szimulálandó modul alkatrészként való deklarálása

```
component pwm is
port (
   -- Input Ports - Single Bit
   reset   :  in  std_logic;
   src_ce  :  in  std_logic;
   src_clk :  in  std_logic;
   start   :  in  std_logic;
   -- Input Ports - Busses
   div_val :  in  std_logic_vector(9 downto 0);
   h       :  in  std_logic_vector(31 downto 0);
   max_val :  in  std_logic_vector(31 downto 0);
   min_val :  in  std_logic_vector(31 downto 0);
   -- Output Ports - Single Bit
   pwm_out :  out  std_logic
   -- Output Ports - Busses
   -- InOut Ports - Single Bit
   -- InOut Ports - Busses
);
end component;

begin
```

A következő részben az egyes alegységek specifikálása és az alegységek összekötése van megvalósítva. Az órajel létrehozása egy külön folyamatrészben valósul meg a *wait for* utasítást alkalmazva. A *wait for* utasítás a szimuláció során csak szintézisre alkalmazható.

```
clk_process :process
begin
```

```
   src_clk <= '0';
wait for clk_period/2;
   src_clk <= '1';
wait for clk_period/2;
end process;
```

A következő programrészben a szimulálandó modul van példányosítva, és a modul portjelei a megfelelő jelekre kapcsolva.

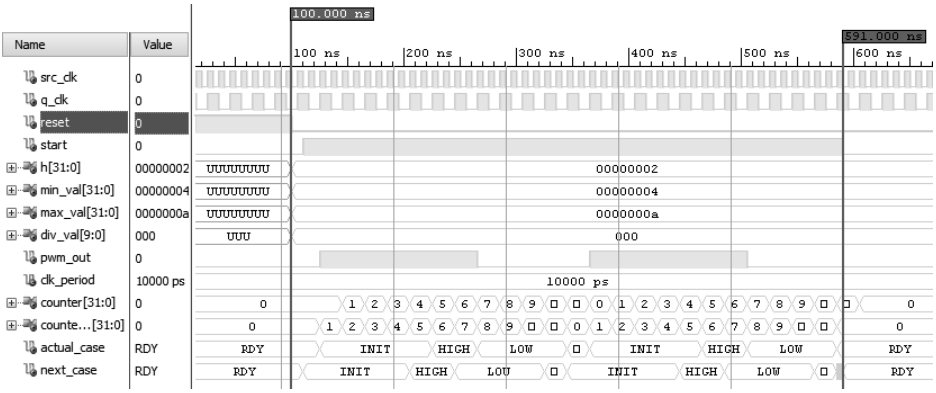
```
pwm_inst: pwm
port map (
   -- Input Ports - Single Bit
   reset                => reset,
   src_ce               => src_ce,
   src_clk              => src_clk,
   start                => start,
   -- Input Ports - Busses
   div_val(9 downto 0)  => div_val(9 downto 0),
   h(31 downto 0)       => h(31 downto 0),
   max_val(31 downto 0) => max_val(31 downto 0),
   min_val(31 downto 0) => min_val(31 downto 0),
   -- Output Ports - Single Bit
   pwm_out              => pwm_out
   -- Output Ports - Busses
   -- InOut Ports - Single Bit
   -- InOut Ports - Busses
);
```

Stimulusjelek generálása. Az idődiagramnak megfelelően az időzítés a wait for utasítással van megvalósítva. A modul első részében 100 ns-ig aktív a reset jel, majd a következőkben a start jel logikai '1' re kapcsolásával indítjuk a pwm generáló modult. A *Start* előtt meghatározzuk a modul bemenetén az inicializáláshoz szükséges jelek értékeit.

```
stim_proc: process    begin

   reset<='1';       -- reset bekapcsolása
   start<='0';
   wait for 100 ns;    --várakozás 10 óraciklust

   reset<='0';  --reszet feloldása
   start<='0';
   -- jelek értékeinek a meghatározása
   -- conv_std_logic utasítással egész std_logic_vector
   -- típusra való átalakítása

   h<=conv_std_logic_vector(2,32);
```

```
    min_val<=conv_std_logic_vector(4,32);
    max_val<=conv_std_logic_vector(10,32);
    div_val<=conv_std_logic_vector(0,10);

    wait for clk_period;   --várakozás egy óraciklust

    start<='1';  --modul indítása
    wait for 48*clk_period;  -- várakozás egy adott ó
        raciklusig
    start<='0';   -- start jel nullásra kapcsolása

    wait;
end process;
end Behavioral;
```

A komplex állapotgépként megvalósított PWM jelgenerátor szimulációs eredményét a 6.7. ábra szemlélteti.



6.7. ábra. PWM generátor komplex állapotgépként való megvalósításának szimulációja

A tesztelő állományban az előosztó nullára, a minimális kitöltési tényező (*min_val*) kettővel, a tényleges kitöltési tényező (*h*) néggyel, a jel periódusát meghatározó *max_val* tízzel van inicializálva. A *reset* jel logikai '1'-re való kapcsolásával a rendszer beáll a *RDY* kezdeti állapotba. A *start* jel engedélyezésével működésbe lép a jelgenerátor. A szimuláció szemlélteti az állapotgép egyik állapotból a másikba való átlépését (*RDY*, *INIT*, *LOW*, *HIGH*), a *counter* számláló értékének növekedését. Az állapotváltás a számláló értéke alapján történik a következő állapotdekodoló modul működésének megfelelően. Az impulzusszélességben modulált jel

$min\_val+h-1$ q_clk órajelnyit van logikai egyesben, valamint max_val+2 q_clk impulzusnyi a jel periódusa. Ha az előosztó értéke nulla, a q_cl k órajel periódusa duplája a főórajel (src_clk) periódusának. Az előosztó egyes, kettes, hármas értékre való állításával a belső q_clk órajel periódusa háromszorosa, négyszerese, ötszöröse lesz az eredeti src_cl k órajel periódusának.

Javasolt feladat a megadott sablon szerint a PWM egység tesztelése szimuláció során.

A szimuláció közben javasolt változtatni az egyes bemenetek értékét (h, min_val, max_val, div_val) és tanulmányozni a hatását a szimuláció eredményére.

A szimulációs grafikonon az alapjeleket (a PWM modul ki- és bemeneti jelei) ajánlott kibővíteni a PWM egység belső jeleivel, és figyelni a jelek (signalok) értékének alakulását a szimuláció során.

7. fejezet

System Generator alapú rendszertervezés és -tesztelés

A fejezet célja System Generator alapú rendszer tervezése és hardver-szoftver ko-szimuláció bemutatása. A System Generator alapú környezet alkalmazása egyszerűsíti a hardvertervezést és -tesztelést. Míg a VHDL alapú tervezés alapos hardverismereteket igényel, a HLS technikák és egyben a System Generator alapú környezet alkalmazásával minimális hardverismeretek mellett is könnyedén megvalósítható egy alkalmazás.

7.1. Bevezetés

A magas szintű szintézis automatizált tervezési folyamat, amely értelmezi egy adott viselkedés algoritmus szinten való leírását (ANSI C, C++, System C, Matlab) és létrehoz egy digitális hardvert FPGA vagy SoC eszközre.

A System generator for DSP^{TM} magas szintű szintetizáló eszköz alkalmazható XILINX programozható áramkörökben nagy teljesítményű DSP rendszerek tervezésére. Lehetővé teszi rendszerek tervezését és szimulációját MATLAB, Simulink és XILINX könyvtárak alkalmazásával. Szintetizálható HDL kódot generál [15]. A létrehozott HDL terv szintetizálható FPGA és SoC hardver implementálására.

A System Generator környezetben speciális Simulink tömbök alkalmazásával tervezhető meg egy áramkör. Az eszköz alkalmazása könnyíti

a fixpontos és lebegőpontos aritmetika alkalmazását, nagymértékben egyszerűsítve algoritmusok megvalósítását. Az eszköz lehetővé teszi Matlab programkód, C, C++ vagy akár HDL nyelven (VHDL, Verilog) megvalósított moduloknak a tervbe való integrálását. A megvalósított terv első fázisban szimuláció során a Simulink környezetben, majd hardver ko-szimulációs üzemmódban tesztelhető.

A terv elkészítése előtt néhány fontosabb beállítás és megjegyzés:

A létrehozott és elmentett Simulink modellbe be kell illeszteni az áramkört megvalósító alegységeket, valamint a konfigurációs eszközöket. Minden egyes alegységnek (alrendszernek) tartalmaznia kell egy *System Generator* konfigurációs eszközt. A *System Generator* eszköz segítségével lehet meghatározni az alkalmazott FPGA vagy SoC áramkör típusát, működési frekvenciáját, azt, hogy az alrendszernek melyik jelen szolgáltatjuk majd az órajelet.

A hardver koszimuláció (vegyes szimuláció) lényege, hogy egy algoritmust több célhardveren futtatunk. Az algoritmus egyik része a gazda számítógépen a *Simulink* környezetben fog végrehajtódni, míg egy másik része az FPGA áramkörben, pontosabban a *Gateway In* és *Gateway Out* közötti részek az FPGA áramkörben, míg a *Gateway In* előtti részek és a *Gateway Out* utáni részek a *Simulink* környezetben. Az alább szemléltetett példában a VHDL hardverleíró nyelven tervezett PWM generátort teszteljük hardver koszimulációs üzemmódban. Jelen feladat esetében a Simulink környezetet a bemeneti adatok meghatározására, valamint az eredmény megjelenítésére alkalmazzuk.

7.2. System Generator adattípusok

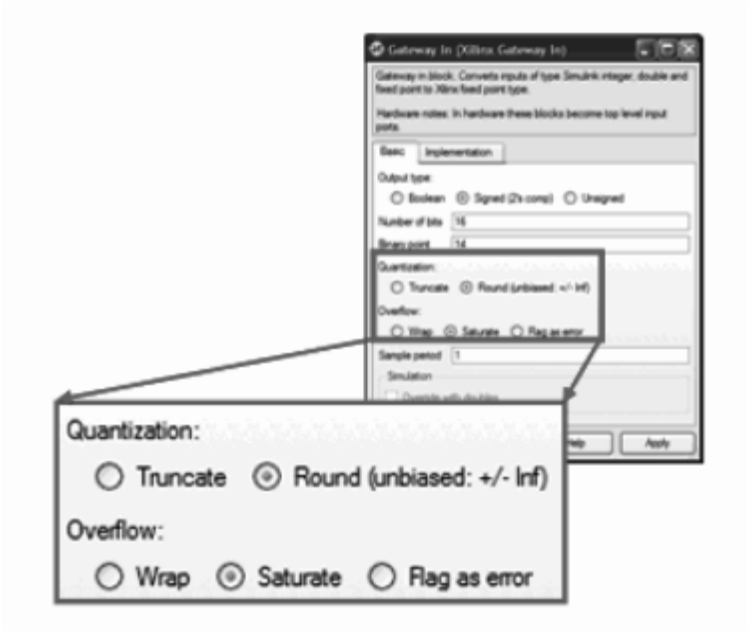
A Matlab és System Generator környezetben a következő típusokat alkalmazhatjuk a különböző jelek, adatok ábrázolására:

- Boole típusú változó (Boolean)
- Fixpontos
  - Előjel nélküli fixpontos aritmetika *UFix_5_3*, 5 biten ábrázolt előjel nélküli szám, 2 bit az egész részre és 3 bit a törtrészre
  - Előjeles fixpontos aritmetika *Fix_16_8*, kettes komplemens, 16 biten van ábrázolva a szám, 8 bit a törtrész
- Lebegőpontos

A fixpontos aritmetika testreszabása során a következő paraméterek állíthatók be:

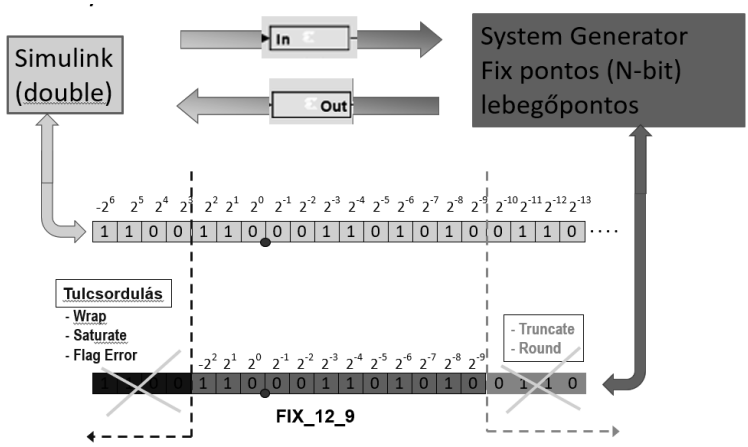
– *full precision*: a modul bemenetének megfelelő pontosságot alkalmazzuk
– *user-specified precision*: a felhasználó által meghatározott pontosságot alkalmazzuk. A felhasználó által megválasztott pontosság általában kisebb, mint például a Simulink környezetből átvett változó pontossága, emiatt az alacsony helyértékű és/vagy a magas helyértékű bitekről is le kell mondani. Az alacsony helyértékű bitek esetében kvantálásról, míg a magas helyértékű bitek esetében túlcsordulásról beszélünk.

A kvantálás megoldható az alacsony helyértékű bitek levágásával (*Truncate*) vagy kerekítésével (*Round*). Túlcsordulás esetében a (*Wrap*) opció alkalmazásával egyszerűen elhagyjuk a magasabb helyértékű biteket, míg a *Saturate* opcióval szaturáljuk a formátumnak megfelelő legnagyobb vagy legkisebb értékre a jelet.



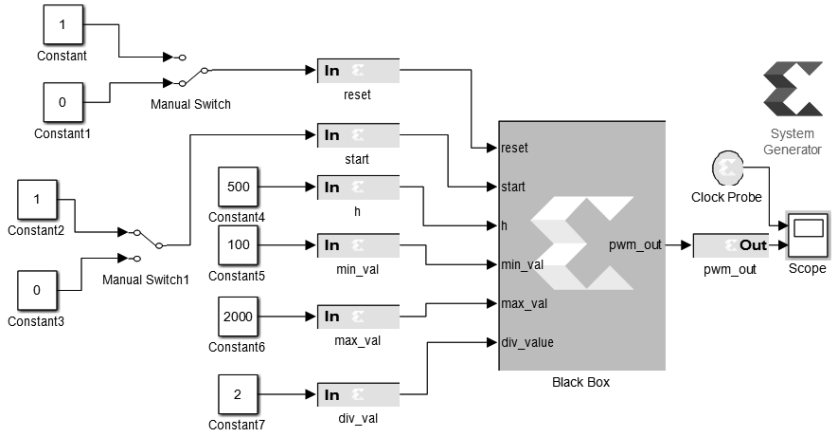
7.1. ábra. Számformátum testreszabása

A kvantálás és túlcsordulás a 7.2. ábrán van összefoglalva.



7.2. ábra. Túlcsordulás, kvantálás szemléltetése

A Matlab környezetben az adatok double (dupla pontosságú lebegőpontos) típusként vannak kódolva, míg az FPGA áramkörben általában egész számként, fixpontos vagy lebegőpontos formátumban. Általában a Matlabból (*Simulink*-ből) az adatoknak az FPGA áramkörre való áramlása során veszítünk a pontosságból. Az átalakítást, illetve a visszaalakítást a *Gateway In*, valamint a *Gateway Out* modulok végzik el.



7.3. ábra. PWM modul system generator alapú tömbvázlata

7.3. System Generator eszköztár

Fontosabb Xilinx alapú Simulink tömbök [15]:

- *System Generator*: az alkalmazott FPGA áramkör paraméterezése, működési frekvencia meghatározása.

- *Gateway In*: bemeneti jelek egy egységben. A *Gateway In* modulok a bemenetek mintavételezését végzik, és a bemeneti adatokat átalakítják a modulban konfigurált típusnak megfelelően, valamint a következő mintavételig megtartják az utolsó mintavételi értéket.

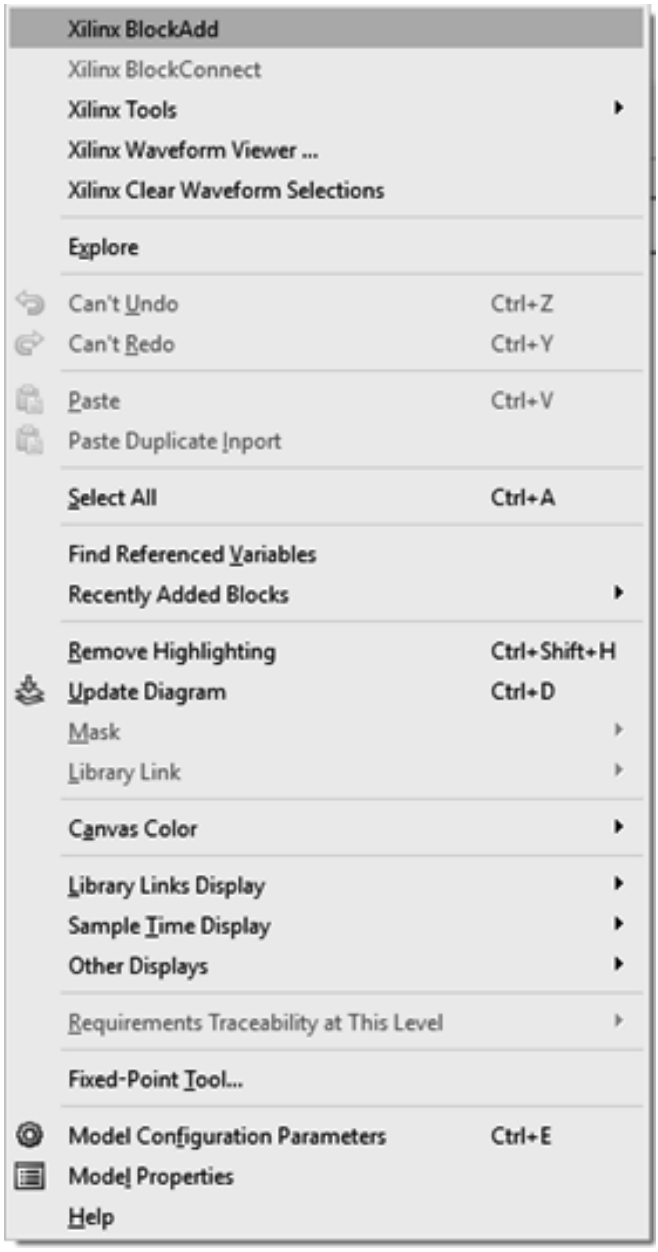
- *Gateway Out*: tömbök az áramkör kimeneteit jelentik, hardver ko-szimuláció során visszaalakítják a Xilinx típusú adatokat a MATLAB-ban alkalmazott double típussá.

- *BlackBox*: VHDL, illetve Verilog alapú moduloknak a tervhez való csatolása. A *Black Box* alkalmazása során létrejön egy konfigurációs .m állomány, amelyben a modul működésével kapcsolatosan több paraméter is beállítható. Többek között meghatározható (lecserélhető) a kimeneti adatok típusa.

- *Register*: a tervbe regisztereknek való integrálását teszi lehetővé. A regiszter esetében nem kell meghatározni a bitek számát, ugyanis ezt automatikusan örökli a bemeneti jel sínszélessége függvényében. A Xilinx tömbök nagy részének esetében alkalmazhatók (validálható) a *reset* (*rst-Provide Synchronous reset port*), illetve az órajelet engedélyező (*EN-Provide Enable port*) bemeneti jelek.

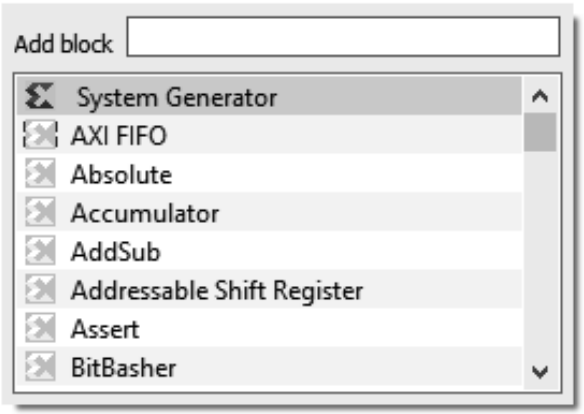
- *Bitbasher*: sínek kezelése, jelek összevonása, illetve jelek sínről való leválasztása.

Az alapkönyvtárban megtalálható Xilinx tömbök a *Xilinx BlockAdd* gyorsparanccsal: jobb kattintás a modellre, majd kattintás a *Xilinx Block Add* almenüpontra (7.4. ábra) vagy *Simulink* könyvtár keresőben (*Simulink Library Search*) megtalálható *Xilinx Blockset*, illetve *Xilinx Reference Blockset* eszköztárakból érhetők el.



7.4. ábra. System Generator alapú modul tervbe való integrálása

A listából ki kell választani a modult, amelyet a tervbe szeretnénk illeszteni (7.5. ábra).



7.5. ábra. Block add lista

A System Generator eszköztárában megtalálhatók:
- egyszerű áramköri elemek: regiszterek, összeadók, számlálók,
- digitális jelprocesszálásra szolgáló modulok: *DSP* modul, *FIR* szűrő, Komplex szorzómodul, *FFT*, *Inverz FFT*,
- memóriamodulok: *FIFO*, *BRAM*, *Dual port BRAM*,
- kommunikációs modulok: *Reed-Solomon Encoder*, *Reed-Solomon Decoder*,
- típuskonverzió.

A *System Generator* modul keretében meghatározzuk az alkalmazott FPGA áramkör típusát és fontosabb paramétereit. A *System Generator* tömb keretében a következő három konfigurációs ablak érhető el (7.6., 7.7. ábrák):
- *Compilation* az alkalmazott fejlesztő lap (vagy FPGA) típusának kiválasztása, az alkalmazás típusa, az alkalmazott hardverleíró nyelv (VHDL) vagy Verilog, alkalmazott VHDL könyvtár, szintézisre kiválasztott stratégia (*Vivado Synthesis Deafaults*), implementációra kiválasztott stratégia (Vivado Implementation Defaults), automatikus dokumentáció generálás (*Create interface document*), automatikus tesztelő áramkör (*Create testbench*);
- *Clocking Enable multiple clock* több órajeles rendszer esetében, az FPGA áramkör működési frekvenciájának megfelelő órajel periódusa (*FPGA clock period*), a modul órajelének meghatározása (*Clock pin location*), Simulinkben az FPGA órajelnek megfelelő periódus;

– *General* modul verziószámának és egyéb paramétereknek a beállítása.

7.6. ábra. System Generator elem fő konfigurációs része

7.7. ábra. System Generator, órajel paraméterezése

Gateway In modulok paraméterezése:

A *Reset* jel *Bool* típusú, míg a *min_val UFix_32_0*, vagyis előjelnélküli fixpontos szám. A kvantálásra *Round*, míg a túlcsordulásra szaturáció (*Saturate*) van beállítva.

Fontosabb beállítások:

– *Reset*: *Boolean* típusú bemenet

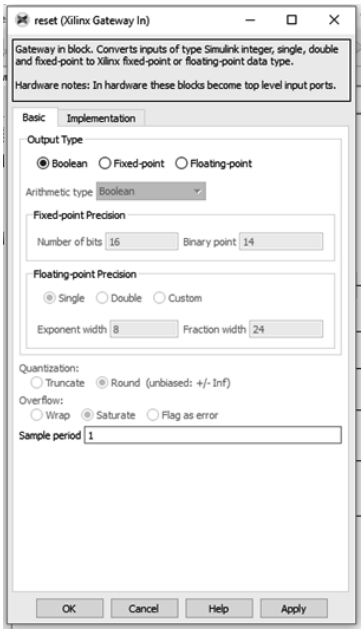
– *Start*: *Boolean* típusú bemenet

– *h*: vezérlőjel, jelen esetben 32 bites előjel nélküli szám (*UFix_32_0*)

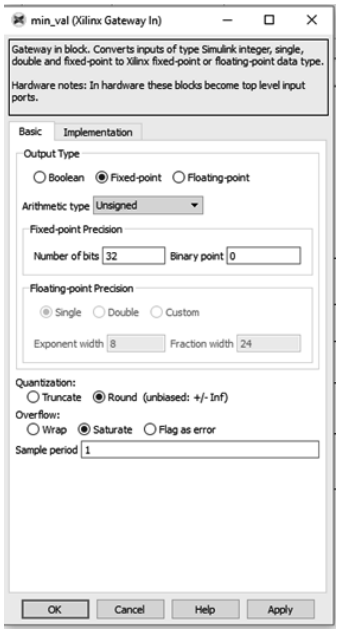
– *Min_val*: meghatározza a minimális kitöltést, jelen esetben 32 bites előjel nélküli szám (*UFix_32_0*)

– *Max_val*: meghatározza a jel periódusát, jelen esetben 32 bites előjel nélküli szám (*UFix_32_0*)

– *Div_val*: előosztó, skálázni lehet a PWM jel periódusát, tíz bites előjel nélküli szám (*UFix_10_0*)



7.8. ábra. PWM modul Reset bemeneti portjelének konfigurálása



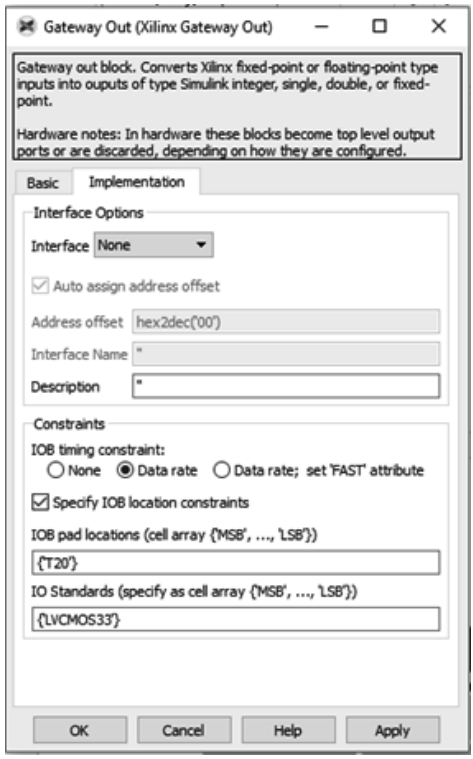
7.9. ábra. PWM modul min_val bemeneti portjelének konfigurálása

Az FPGA lábak megnevezését ki kell keresni az FPGA fejlesztőlap adatlapjából, vagy fejlesztőlaphoz a gyártó által szolgáltatott xdc vagy ucf állományból. A *Zybo* fejlesztőlapra a PMod csatlakozókra érvényes jelek elnevezése a 7.10. ábrán van szemléltetve.

Pmod JA (XADC)	Pmod JB (Hi-Speed)	Pmod JC (Hi-Speed)	Pmod JD (Hi-Speed)	Pmod JE (Std.)	Pmod JF (MIO)
JA1: N15	JB1: T20	JC1: V15	JD1: T14	JE1: V12	JF1: MIO-13
JA2: L14	JB2: U20	JC2: W15	JD2: T15	JE2: W16	JF2: MIO-10
JA3: K16	JB3: V20	JC3: T11	JD3: P14	JE3: J15	JF3: MIO-11
JA4: K14	JB4: W20	JC4: T10	JD4: R14	JE4: H15	JF4: MIO-12
JA7: N16	JB7: Y18	JC7: W14	JD7: U14	JE7: V13	JF7: MIO-0
JA8: L15	JB8: Y19	JC8: Y14	JD8: U15	JE8: U17	JF8: MIO-9
JA9: J16	JB9: W18	JC9: T12	JD9: V17	JE9: T17	JF9: MIO-14
JA10: J14	JB10: W19	JC10: U12	JD10: V18	JE10: Y17	JF10: MIO-15

7.10. ábra. Zybo fejlesztőlap PMod kivezetései [16]

A jeleket, amelyeket az FPGA kivezetéseire szeretnénk kapcsolni, a *Gateway In* és *Gateway Out* modulok *implementation* alrészében kell meghatározni az FPGA kivezetését, illetve a jel típusát (*LVCMOSS33*, *LVCMOSS25*, *LVTTLxx* stb). A PWM kimeneti jelre az FPGA kivezetésének hozzárendelése a 7.11 ábrán van szemléltetve.



7.11. ábra. FPGA kivezetések illesztése

A Simulinkben a beállított bemenetek függvényében lefuttatva a szimulációt, ki lehet értékelni, hogy a jelforma megfelel-e a valóságnak.

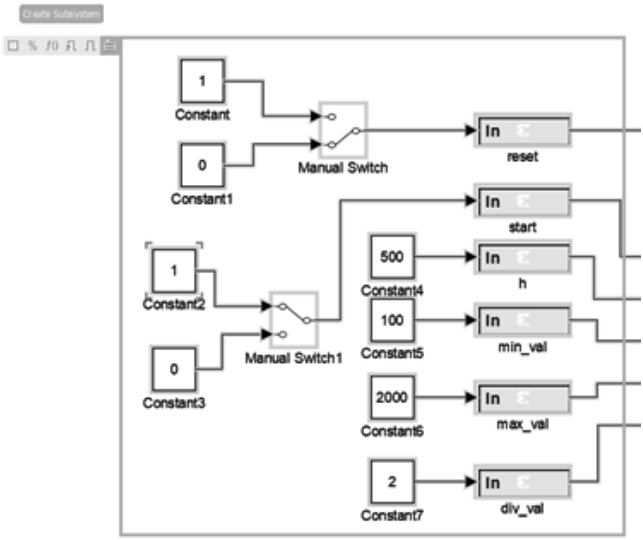
A hardver koszimulációs üzemmódban, ahhoz, hogy az FPGA részben a PWM modul, az FPGA áramkör órajelén működjön, a tervet át kell alakítani több órajel alapú tervvé, mivel a *Gateway In* és *Gateway Out* modulok a JTAG interfész órajelén működnek, a PWM modult pedig az FPGA áramkör órajelén szeretnénk működtetni. Korábbi ISE alapú *System Generator* környezetekben hardver koszimuláció során lehetőség van több üzemmód

közül választani. *Free running* üzemmódban a rendszer órajelét az FPGA fő órajele szolgáltatja, Single clock üzemmódban pedig a JTAG interfész.

Jelen esetben a *Zybo* fejlesztőlapra, ha nem alakítunk ki egy több órajel által vezérelt rendszert, vagyis egy *Multiple Clock* típusú rendszert, csak a JTAG egység órajelén van lehetőség futtatni az FPGA-ban integrált egységeket (legalábbis az egyetemen csak ebben a változatban sikerült a *Zybo* lapon létrehozni a valós idejű hardver koszimulációt).

7.4. Multiple Clock típusú rendszer kialakítása

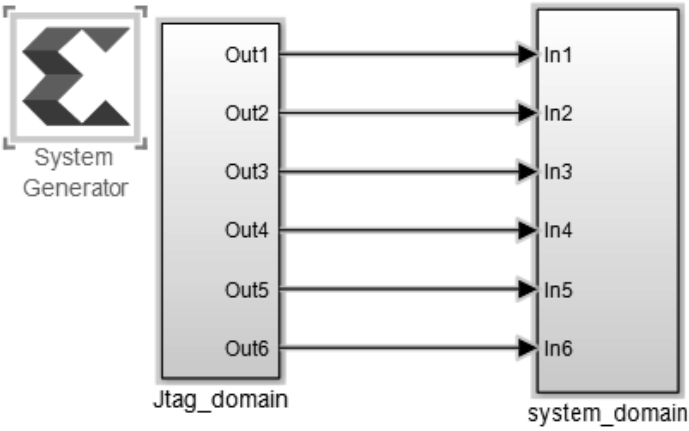
A *Multiple Clock* típusú rendszer kialakítása egyszerűen megvalósítható: egy részét a tervnek be kell vinni egy alrendszerbe (Simulinkhez kapcsolódó egységek, *Gateway In*, modulok) (7.12. ábra) és a Xilinx tömböket (vagyis a terv azon részét, amely az FPGA áramkörön kell lefusson) egy másik alegységbe (subsystem típusú Simulink elem).



7.12. ábra. Modulok alrendszerbe rendezése

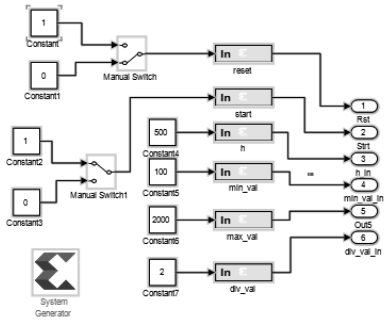
Kijelölve az alrendszerekhez tartozó elemeket, majd a *Create Subsystem* utasításra kattintva az elemek automatikusan bekerülnek egy alrendszerbe.

A kialakított doméniumok a 7.13. ábrán vannak szemléltetve.

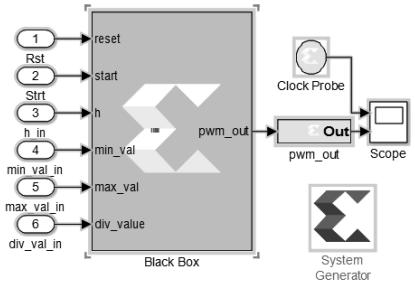


7.13. ábra. Kialakított doméniumok

A *Jtag_domain*-nek megfelelő alegység az órajelet a *Jtag_clk* jel, míg a *system_domain* alegységnek az órajelet a *sys_clock* jel fogja szolgáltatni. A felső szintű *System Generator* konfigurációs eszközben engedélyezni kell a *Multiple Clock* típusú rendszert. A *System Generator* modult be kell integrálni minden egyes alrendszerbe (jelen esetben *Jtag_domain* (7.14. ábra), illetve *system_domain* (7.15. ábra)) és az alrendszerekben *Single Clock* típusú egységként működnek.



7.14. ábra. JTAG alrendszer elemei



7.15. ábra. Rendszer doménium elemei

A két modul különböző frekvencián működik, ebben az esetben a két modul közé regiszterek közbeiktatásával oldjuk meg a két modul illesztését.

csak a *Jtag_domain*-ből kapcsolódnak a jelek a *System_domain* irányába, tehát a tervben csak a bemenettől a kimenet irányába vannak regiszterek illesztve. A két alrendszer a 7.16, 7.17. ábrákon bemutatott elemeket tartalmazza.

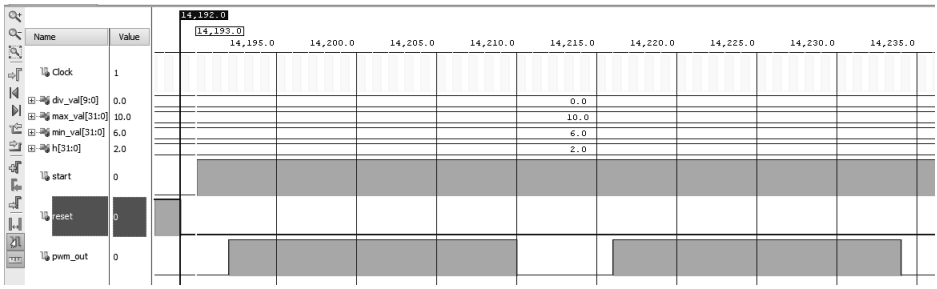
7.16. ábra. Órajel konfigurálása JTAG doméniumra

7.17. ábra. Órajel konfigurálása rendszer doméniumra

A Simulink *Scope*-okon kívül lehetőség van a jelformák megjelenítésére a *Xilinx Waveform Viewer* eszköznek az alkalmazásával is. Egyszerűen ki kell jelölni a jeleket, amelyeket meg szeretnénk jeleníteni, majd ki kell választani a gyorsmenüből a *Xilinx Waveform Viewer* parancsot.

Xilinx Clear Waveform Selection paranccsal pedig visszavonhatjuk a megjelenítést. A jelformáknak ebben a változatban való megjelenítése több lehetőséget nyújt a rendszer tesztelése szempontjából, ugyanis többek között lehetővé teszi a lefutó/felfutó élek keresését.

A *System Generator* környezetbe integrált PWM jelgenerátor szimulációs eredményét a 7.18. ábra szemlélteti. A szimuláció eredménye a *Xilinx Waveform Viewer* eszközzel van megjelenítve. A modul bemeneteire a jeleket Simulinkből határozzuk meg. A *reset* jel elengedésével és a *start* logikai '1'-re való kapcsolásával működésbe lép a PWM jelgenerátor. A szimuláció eredménye is megegyezik az előbbi fejezetben részletezett PWM modul szimulációs eredményével, mivel a két modul megegyezik. Ebben a változatban a VHDL-ben specifikált PWM modul a System Generator környezetbe van integrálva.



7.18. ábra. Szimuláció eredménye

7.5. Hardvermodell létrehozása

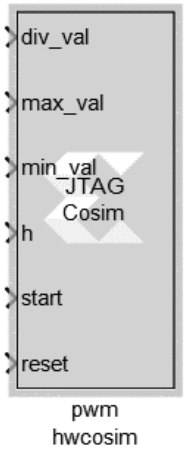
A *Multiple Clock* típusú rendszer kialakítását követően létre kell hozni a hardver koszimuláció elvégzéséhez szükséges Simulink tömböt, a *System Generator* modulból a Generate gombra kattintva. Minden egyes fordítást követően a *System_domain* egységnek meg kell határozni, hogy honnan származtatja a *Sys_clock* órajelet. Ehhez a módosításhoz a fordítás kezdetét követően meg kell keresni a *netlist/hwcosim/pwm_a.srcs/constrs_1/imports/sysgen* könyvtárban található megkötés állományt, valamint a tartalmát az alábbi minta szerint átszerkeszteni.

```
set_property -dict {PACKAGE_PIN L16    IOSTANDARD LVCMOS33}
    [get_ports {sys_clock}];
create_clock -add -name sys_clk_pin -period 10.00
  -waveform {0 5} [get_ports {sys_clock}];
```

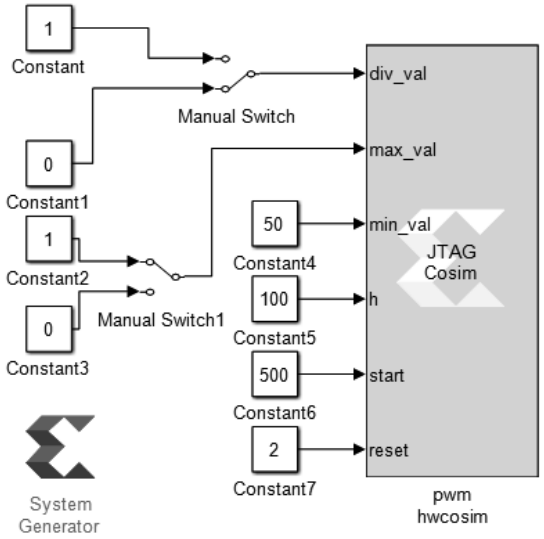
Ha a terv sikeresen lefordul, létrejön a ko-szimulációhoz szükséges könyvtárelem (7.19. ábra).

Kimásolva a könyvtárelemet, egy újabb Simulink modellben ki kell alakítani a ko-szimulációs modellt. Ez egyszerűen úgy is megoldható, hogy az eredeti modellből kivágjuk a Xilinx elemeket, és helyettük behelyezzük a létrehozott könyvtárelemet. Ebben az esetben a System Generator modulban a tervet egy órajeles rendszerré kell állítani. A létrehozott könyvtárelem hátterében megtalálható az FPGA áramkör konfigurálásához szükséges bitfolyam, valamint a System Generator beállítása alapján azonosítható az alkalmazandó fejlesztő rendszer.

Összekapcsolva a számítógéppel és lefuttatva a szimulációt, első lépésben felprogramozódik az FPGA áramkör, majd azt követően elindul a szimuláció. Ahhoz, hogy a megvalósított automata működésbe lépjen, a



7.19. ábra. Koszimulációs modul



7.20. ábra. Koszimulációs model tömbvázlata

reset jelet logikai nullára, illetve a start jelet logikai egyesre kell állítani. A szimuláció futási ideje alapértelmezetten 10 s, ami az FPGA áramkörben 10 órajelnek felel meg. A szimuláció időtartamát javasolt átállítani inf értékre.

Összefoglalva a gyakorlati feladat elvégzésének fontosabb lépéseit:

- A leírások alapján a terv megvalósítása, felhasználva a VHDL-ben létrehozott PWM modult.
- Szimulációval ellenőrizzük, hogy a rendszer működése megfelel-e az elvárásoknak
- A *Multiple Clock* típusú rendszer kialakítása.
- A koszimulációhoz szükséges könyvtári elem létrehozása, figyelve, hogy minden újrafordítás során helyesen állítottuk-e be a *sys_clock* órajel származtatását.
- Oszcilloszkópot alkalmazva, különböző bemeneti paraméterekre tanulmányozzuk a generált jelformát.
- Hasonlítsuk össze a valós mérések alapján, valamint a szimuláció során elért eredményeket.

8. fejezet

HLS alapú hardvertervezés

A téma keretében a Vivado HLS (High Level Synthesis) magas szintű szintetizáló eszközt alkalmazva valósítunk meg néhány egyszerű áramkört: összeadó vagy szorzó áramkör, tömb elemei összegének a kiszámítása. Bevezetünk a HLS módszerhez kapcsolódó alapvető fogalmakat és kényszerlehetőségeket, amelyek alkalmazhatóak az interfészek, portok, memóriák, ciklusok specifikálására.

8.1. Bevezető

A magas szintű szintetizáló eszközök alkalmazása során meg kell határozni, hogy például a C, C++ programkód különböző részei milyen módon valósíthatóak meg hardverben. Egy-egy függvénynek hardverre fordítására, tehát egy függvény alapján egy áramköri elem létrehozására nagyon sok lehetőség közül kell választani. A tervező a szintetizáló programnak recepteket kell megadjon, amely alapján elkészül a hardver. Annak függvényében, hogy egy adott forráskódra milyen kényszerfeltételeket alkalmazunk, ugyanabból a programkódból különböző felépítésű hardverelem hozható létre.

A HLS eszköz alkalmazása során azt tapasztaltam, hogy a tervezés folyamán a legfontosabb az interfészek és bemeneti portok megértése, hiszen ezek ismeretében átláthatóvá és érthetővé válik az egyes alegységek összekapcsolása, a modulokba az adatok bevitele, valamint az eredmény kinyerése. Annak függvényében, hogy az interfészekre és portjelekre milyen protokollt alkalmazunk, különböző típusú bemeneti és kimeneti vezérlőjelek vannak alkalmazva.

A fejezetben egyszerű példák szemléltetésével elsajátítható a különböző kényszerfeltételek alkalmazása, mint például az interfész és port típusú kényszerfeltételek és a kapcsolódó vezérlőjelek.

8.2. Interfészek és protokollok meghatározása

A magas szintű szintézis során az egyik legfontosabb résznek a C, C++ forráskódból az RTL fordítás során az interfész kialakítását és megértését tartom. Pontosabban, hogy megértsük, hogy a függvényből, a függvény argumentumaiból és a visszatérített értékből hogyan lesznek hardver szintű portjelek, valamint különböző direktívák alkalmazása mellett milyen vezérlőjelek jönnek létre, amelyek segítik a tervezett modulok szinkronizálását.

Egy RTL implementáció portjeleit a következő elemekből lehet származtatni:

- bármely megadott függvény szintű protokollból,
- függvények argumentumából,
- globális változók, amelyeket a legfelső szintű (top level) függvény elér.

Az RTL szintézis során azt, hogy hogyan jönnek létre a modul portjelei, az INTERFACE kulcsszóval lehet meghatározni. A kényszerfeltételeket meg lehet adni a forráskódban a *#pragma* kulcsszót követően, vagy egy külön .tcl szkript állományban [17].

Függvény szintű protokoll vagy blokk szintű I/O protokoll vezérlőjeleket szolgáltat/illeszt a modul vezérléséhez. Ezen vezérlőjelekkel jelzünk a függvényből származtatott modulnak, amikor elkezdi a műveletek végrehajtását (*start*) és jelzi, amikor az elkezdett műveletsorozat befejeződik (*done*), a modul készenléti állapotban van (*idle*) és készen áll egy új bemenet fogadására (*ready*).

A blokk szintű protokoll lényegében meghatározza, hogy hogyan kezeljük a tervezett alegységet: mikor kezdhet el működni, mikor fejezte be az adott számítási ciklust, valamint mikor áll készen új adatok fogadására. Az illesztett vezérlőjelek segítenek az egyes modulok összekapcsolásában, egy rendszerbe integrált alegységek szinkronizálásában.

Függvényszintű protokollok típusát a mode kulcsszóval határozzuk meg. A Vivado HLS a következő négy függvényszintű protokollt alkalmazza [18]:

- *Ap_ctrl_none*: e protokoll kiválasztása során nem generálódnak függvény szintű vezérlő jelek. Abban az esetben javasolt ennek a protokolltípusnak az alkalmazása, ha az adatfolyam folyamatos és

nem kell megszakítani, leállítani a számolást, tehát nem szükséges egyéb vezérlőjelek csatolása.
- *Ap_ctrl_hs*: különböző vezérlő portjeleket alkalmaz a műveletvégzés indítására (*start*), valamint jelzi, hogy a modul készenléti állapotban van (*idle*), befejezte a műveletvégzést (*done*) és készen áll új adat fogadására. Az *ap_ctrl_hs* az alapértelmezett blokk szintű I/O protokoll.
- *Ap_ctrl_chain*: blokk szintű vezérlőjeleket implementál a modul működésének elkezdésére, a műveletek folytatására, valamint jelzi, amikor a modul készenléti állapotban van, befejezte a műveletvégzést és készen áll egy következő adat fogadására.
- *s_axilite* AXI sínrendszerre illeszthető interfész jön létre.

Az *ap_ctrl_hs* protokoll jelei [18] [17]:
- *ap_start*: ez a jel vezérli a modul működését. A jelet '1' logikai szintre kell kapcsolni ahhoz, hogy a modul elkezdjen működni, a modul bemenetén található adatokat elkezdje feldolgozni. Ha az *ap_start* jelet hamarabb visszakapcsoljuk logikai '0' szintre, mint hogy az ap_ready logikai '1' szintre váltana, valószínű, hogy a modulnak nem sikerül az összes bemenetet beolvasni. A műveletvégzés emiatt leállhat a következő bemenet beolvasására várva.
- *ap_ready*: a kimenet jelzi, hogy a modul készen áll vagy sem egy új bemeneti adat fogadására. Az *ap_readylogikai* '1' szintre való váltásakor a modulnak sikerült ebben a műveletvégzési ciklusban processzálni az összes beolvasott adatot és készen áll az új adatok fogadására.
- *ap_done*: kimeneti vezérlőjel, jelzi, hogy a modul befejezte az aktuális tranzakció összes műveletét. Mivel ez a tranzakció végét is jelenti, szintén jelzi, hogy az *ap_return* kimeneti porton érvényes az adat.
- *ap_idle*: kimeneti vezérlőjel jelzi azt, hogy az alegység működik vagy nem végez műveletet (*idle*). Várakozási állapotban (*idle*) az *ap_idle* kimenet magas logikai szinten van. Amikor a modul elkezdi a műveletvégzést, a kimeneti jel logikai '0' szintre vált. Logikai '1' szintre vált, amikor a modul befejezte a műveletvégzést.

Minden egyes függvény argumentumhoz, vagy abban az esetben, ha a függvény visszatérít értéket, a visszatérítési értékhez port szintű I/O interfész protokollt lehet meghatározni, mint például érvényes kézfogás (*ap_vld*) vagy kézfogás nyugtázása (*ap_ack*). Abban az esetben, ha az INTERFACE pragma *mode* paraméterét csak felső szintű függvényekre lehet alkalmazni, az alfüggvényekre az INTERFACE pragmának csak a register paramétere

alkalmazható. A *mode* kulcsszóval meghatározható az interfész protokoll típusa a függvények argumentumaira, függvények által alkalmazott globális változókra, vagy a blokk szintű vezérlő protokollra. Az alábbi listában felsorolunk a *mode* kulcsszóra néhány lehetséges paramétert:

- *ap_none*: nem alkalmaz protokollt, az interfész egy egyszerű adatport.
- *ap_stable*: nincs alkalmazva protokoll, a Vivado HLS feltételezi, hogy a reset jelet követően az adat a porton mindig stabil, ami lehetővé teszi a terv optimalizálása során a számítási láncban a felesleges regiszterek törlését.
- *ap_vld*: az adatporthoz egy jelző jelet rendel (*valid*), amely jelzi, hogy az adat készen áll az írásra vagy olvasásra.
- *ap_ack*: az adatporttal együtt implementál egy nyugtázó jelet (*ack*), amelyen nyugtázza, hogy az adaton megtörtént az írás vagy olvasás.
- *ap_hs*: az adatport mellett alkalmazza az érvényes adat (*valid*), valamint a nyugtázó (*acknowledge*) jelet, egy kétirányú kézfogásos protokollt biztosítva, jelezve, amikor az adat érvényes írásra vagy olvasásra (*vld*), valamint nyugtázza, hogy az írás vagy olvasás megtörtént (*ack*).
- *ap_ovld*: a kimeneti adatporthoz egy érvényes jelző bitet csatol (vld), amelyen jelzi, hogy az adat készen áll az olvasásra.
- *ap_fifo*: a portot egy standard FIFO interfészként valósítja meg, alkalmazza a ki- és bemeneti adatportokat, kiegészítve az üres (empty) és tele (full) portjelekkel. Az *ap_fifo* nem biztosítja a kétirányú írást, vagy olvasást, vagy csak az argumentumok írását vagy az olvasását teszi lehetővé.
- *ap_bus*: egy mutatót sín interfészként valósít meg.
- *ap_memory*: tömb típusú argumentumot standard RAM interfészként valósít meg.
- *bram*: a tömb típusú argumentumokat standard RAM interfészként valósítja meg. RTL alapú tervezés esetében a memória interfész egy portos memóriaként jelenik meg.
- *axis*: az összes portot AXI4-Stream interfészként valósítja. meg. Az AXI (Advanced eXtensible Interface) az ARM szabványosított sínrendszere és az Advanced Microcontroller Bus Architecture (AMBA) specifikáció része.
- *s_axilite*: az összes portot AXI4-Lite interfészként valósítja meg.

– *m_axi*: az összes portjelet AXI4 interfészként értelmezi. A *config_interface* paraméterrel egyéb AXI4 típusú paraméterek konfigurálhatóak, mint például 32 vagy 64 bites címzés kiválasztása.

Register: fakultatívan alkalmazható kulcsszó, amely egy kimeneti/bemeneti jelhez vagy vezérlőjelhez regisztert rendel. Az eredmény, hogy a regiszterrel ellátott jelek mindaddig fennmaradnak, amíg legalább a függvény végrehajtásának utolsó ciklusa meg nem történik. A regiszter opció a következő interfészmódok esetében alkalmazható:

– *ap_none*
– *ap_ack*
– *ap_vld*
– *ap_ovld*
– *ap_hs*
– *ap_stable*
– *axis*
– *s_axilite*

A függvény-argumentumokra és globális változókra alapértelmezetten a következő protokoll van alkalmazva:

– Csak olvasható (bemenet): *ap_none*.
– Csak írható (kimenet): *ap_ovld*.
– Írható /olvasható (ki-bemenet): *ap_vld*.
– Tömbök: a*p_memory*.

8.3. Kényszerparancsok, direktívák

Kényszerparancsok, direktívák értelmezése [18]:

– ALLOCATION: korlátozza az alkalmazható műveletek, műveletvégző egységek számát, ezáltal megosztott hardvererőforrás van alkalmazva.
– ARRAY_MAP: több kisebb tömböt egy nagy tömbbe egyesít, ezáltal csökkenti a szükséges blokk RAM erőforrások számát.
– ARRAY_PARTITION: nagyobb méretű tömböket feloszt kisebb méretű tömbökre vagy regiszterekre, javítva az adatokhoz való hozzáférést és csökkentve a BRAM memóriák alkalmazásából származó szűk keresztmetszetet.
– ARRAY_RESHAPE: átalakítja a memóriatömböket úgy, hogy egy többelemű tömböt átalakít kisebb elemű, viszont nagyobb szószélességet tároló tömbre (több adatbitet). Javítja az adatokhoz való

hozzáférést anélkül, hogy megnövelnénk a használt BRAM memóriák számát.
- DATA_PACK: egy struktúra adatmezőit egyetlen szélesebb szószélességgel rendelkező skalárba csomagolja.
- DATAFLOW: feladatszintű csővezeték (pipeline) kialakítását engedélyezi, biztosítva függvények és ciklusok párhuzamos működését, csökkentve a műveletvégzéshez szükséges ciklusok számát.
- DEPENDENCE utasítás: egyéb kiegészítő információkat szolgáltat a szintézishez a ciklusokból származó függőségek feloldására, lehetővé téve a ciklusokból csővezeték kialakítását.
- EXPRESSION_BALANCE: engedélyezi az automata kifejezés kiegyensúlyozás kikapcsolást. C-ben egy feladat egy szekvenciális műveletsorozattal van specifikálva, amely az RTL szintézisben egy hosszú műveletláncot eredményez, növelve a műveletvégzéshez szükséges órajelek számát. A HLS szintetizáló eszköz, kihasználva a műveletek asszociatív és kommutatív tulajdonságát, újrarendezi a műveletek végrehajtási sorrendjét. Az újrarendezés egy kiegyensúlyozott számítási fát eredményez, csökkentve a számítási lánchoz szükséges órajelek számát.
- FUNCTION_INSTANTIATE: függvények többszöri példányosítása során lehetővé teszi a lokális optimalizálást.
- INLINE: ezzel a kényszerfeltétellel eltávolítja a függvényhierarchiát. Függvények határain átívelő logikai optimalizálást engedélyez. Csökkenti az óraciklusok számát, kiiktatva a függvényhívások költségeit.
- INTERFACE: meghatározza a függvényből és argumentumaiból az RTL modul portjainak származtatását.
- LATENCY: ezzel a megkötéssel meghatározható a minimum és maximum óraciklus.
- LOOP_FLATTEN: lehetővé teszi egymásba ágyazott ciklusok egyetlen ciklusba való összevonását, csökkentve a művelethez szükséges óraciklusokat.
- LOOP_MERGE: egyesíti az egymást követő ciklusokat, csökkentve az általános késleltetést, növelve a megosztott erőforrás-használatot és javítja a logikai optimalizálást (egyszerre az összevont ciklusokra).
- LOOP_TRIPCOUNT: változó ciklusiteráció esetében megad egy becsült iterációszámot. Nincs hatása a szintézisre, csak a szintézis eredményét kiértékelő jelentésben.

- PIPELINE: ciklusokon és függvényeken csővezetéket hoz létre, csökkenti az inicializálási intervallumot, engedélyezve a párhuzamos végrehajtást.
- PROTOCOL: a programkód egy részének megengedi, hogy protokoll részként kezeljük. A protokoll programrészből interfész protokoll határozható meg (alakítható ki).
- RESET: a reset jel globális vagy lokális állapotváltozókhoz való hozzáadását vagy eltávolítását teszi lehetővé.
- RESOURCE: egy változóra, tömbre, aritmetikai műveletre vagy függvényargumentumra meghatározza, hogy milyen könyvtári erőforrással legyen megvalósítva.
- STREAM: meghatározza, hogy agy adott tömb FIFO vagy memóriacsatornával legyen megvalósítva az adatfolyam optimalizálása során.
- UNROLL: kifejti, kitekeri a ciklusokat, több független műveletet végző modult alkalmaz. Teljes cikluskifejtés esetében annyi műveletvégző modult alkalmaz, ahány ciklusiteráció van.

8.4. Kényszerfeltételek szemléltetése példákkal

A könyv terjedelme nem teszi lehetővé az interfész vagy port típusú kényszerfeltételek teljes körű bemutatását, de néhány példán keresztül megpróbálja az érdeklődőt rávezetni a fontosabb tervezési ötletek megértésére és alkalmazására. Első lépésben egy egyszerű szorzó áramkört mutatunk be, és különböző kényszerfeltételeket alkalmazva tárgyaljuk a létrejött interfész vezérlőjeleit (adat, illetve vezérlőjelek).

Bemutatjuk a különbséget a létrejött portjelek szempontjából, ha:

- az eredményt a függvény teríti vissza,
- az eredményt egy argumentum szolgáltatja,
- a kimenetekre regisztereket illesztünk.

Szorzómodul, az eredmény a függvény visszatérítési értéke

Első esetben a művelet eredményét függvény téríti vissza, a C program a következő részben van bemutatva. A header állomány tartalma:

```
ifndef _FUNC_SIZED_H_
#define _FUNC_SIZED_H_
```

```
#include <stdio.h>
#include "ap_cint.h"

typedef int12 din_t;  // típus deklarálás 12 bites előjeles
    egész szám
typedef int24 dout_t; // típus deklarálás 24 bites előjeles
    egész szám

dout_t szorzas_a(din_t a,din_t b);

#endif
```

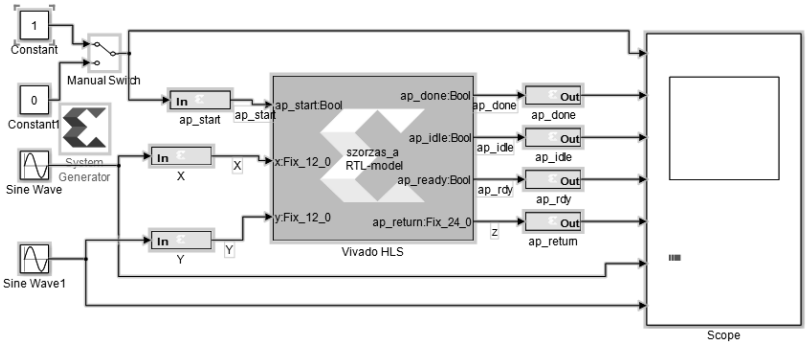
A szorzást megvalósító felső szintű (top level) függvény:

```
#include "szorzas_a.h"

dout_t szorzas_a(din_t x, din_t y) {
// x,y: bemeneti adat
  int tmp; // segédváltozó
  tmp = (x * y); // szorzás eredményének tárolása a
    segédváltozóban
  return tmp;
}
```

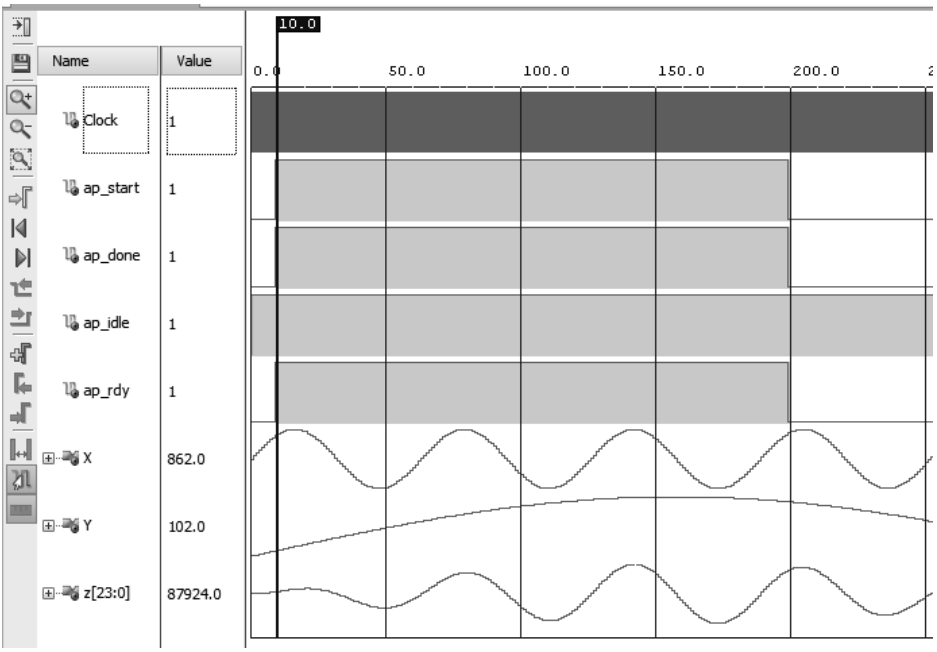
Ha nem alkalmazunk egyetlen megkötést (direktívát) sem, a ki- és bemeneti jelek mellett a következő függvény szintű vezérlőjelek generálódnak: *ap_start*, *ap_done*, *ap_ready*, *ap_idle*. Tehát az alapértelmezetten alkalmazott függvény szintű protokoll: *ap_ctrl_hs*.

A példa egyszerűbb értelmezése és megértése végett a szintetizált RTL modellt generáltuk és exportáltuk *System Generator*ba (8.1. ábra). Hasznosnak találom grafikusan megjeleníteni a hardver modult, ezáltal könnyen integrálható egy célfeladatban (akár a modul tesztelése céljából), és vizuálisan is értelmezhetőek a különböző kényszerfeltételek mellett létrejött vezérlőjelek.

HLS-ben generált szorzó modul *System Generator* alapú tömb vázlata a 8.1. ábrán van szemléltetve. Alapértelmezett függvényszintű protokoll: *ap_ctrl_hs*. A két összeadandó értéket az x és y bemeneteken kapcsoljuk a szorzó modulra. A két különböző periódusú szinuszjelet két szinuszgenerátor szolgáltatja.



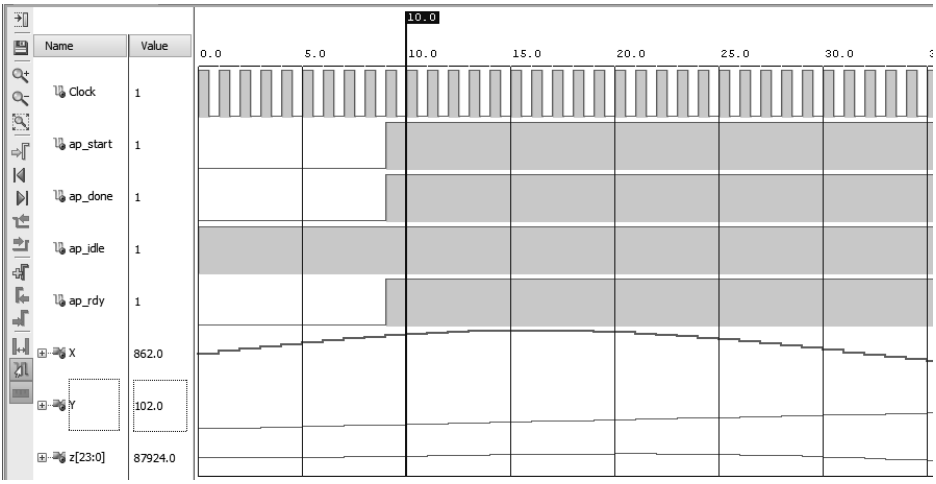
8.1. ábra. HLS-ben tervezett szorzó áramkör System Generator tömbvázlata



8.2. ábra. Szimulációs eredmények, szorzómodul

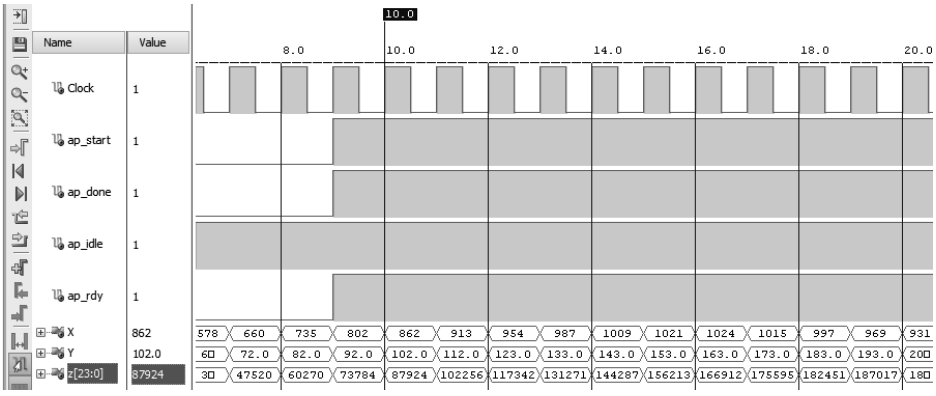
A 8.2. ábrán láthatóak a vezérlőjelek, a két bemeneti szinuszjel és a kimenetként a két szinuszjel szorzata.

A 8.3 ábrán órajel szinten értelmezhető a műveletvégzés. Az *ap_start* logikai '1'-esre való kapcsolását követően az *ap_done* és *ap_ready* is logikai magas szintre vált.



8.3. ábra. Részlet a szorzómodul szimulációs eredményéből

Sem a bemenethez, sem a kimenethez nem volt regiszter illesztve. A szimulációs ábrán megfigyelhető, hogy amelyik pillanatban a bemenet start vezérlőjelet logikai egyesre kapcsoljuk (modulműködés engedélyezve), a kimeneten a vezérlőjelek is azonnal váltanak, és a szorzás eredménye is azonnal megjelenik a kimeneten.



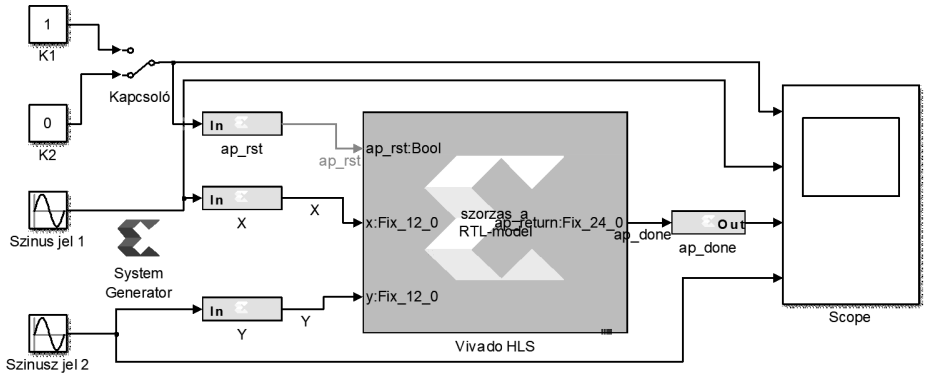
8.4. ábra. Szimulációeredmény ellenőrzése

A 8.4. ábrán értékszinten is ellenőrizhető a szorzás eredménye. A z kimenet esetében az alapértelmezett *ap_ctrl_hs* protokoll, míg a $z1$ kimenet esetében az *ap_ctrl_none* protokoll van alkalmazva. A szimulációs diagramon csak az első modul vezérlőjelei vannak szemléltetve. A két modul párhuzamosan futott ugyanabban a szimulációban: mindkét modulra ugyanazok a bemeneti adatok vannak rávezetve. A $z1$ kimenethez regiszter van hozzárendelve, amint látható a szimulációs kimeneten is.

A következő változatban a függvényszintű protokollként *ap_ctrl_none* mód van beállítva és a kimenethez regiszter van illesztve.

```
#include "szorzas_a.h"
dout_t szorzas_a(din_t x, din_t y)
{ // x,y: bemeneti adat
  //kényszerfeltétel ap_ctrl_none interfész alkalmazására
    valamint regiszter alkalmazása a kimeneten
  #pragma HLS INTERFACE ap_ctrl_none register port=return
  dout_t tmp=0; //átmeneti változó inicializálása
  tmp = (x * y);  //szorzás eredményének a tárolása a tmp á
    tmeneti változóban
  return tmp;
}
```

Abban az esetben, ha a függvényszintű protokollként az *ap_ctrl_none* módot választjuk és a kimenethez regiszter van illesztve, a vezérlőjelek között nem szerepel a bemeneten a start és a kimeneten az *ap_done*, *ap_ready* és *ap_idle* jelzőbitek. A szimulációs eredményeket a 8.5. ábra szemlélteti.

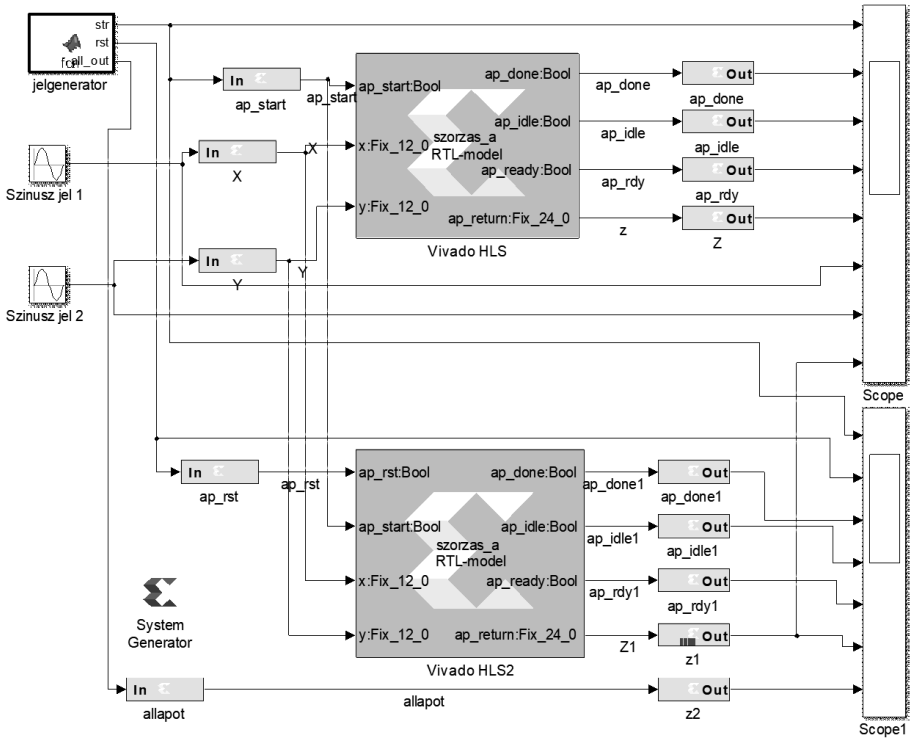


8.5. ábra. HLS-ben generált szorzómodul, regiszterelt kimenettel

Az *ap_start*, *ap_done*, *ap_ready*, *ap_idle* nem jelennek meg, mivel a függvény szintű *ap_ctrl_none* interfész protokoll nem alkalmaz vezérlőjeleket. Az *ap_rst* bemeneti vezérlőjel a kimenethez rendelt regiszter miatt jelenik meg. A z kimeneten a bemeneti adatokkal megegyező óraciklusban elérhető az eredmény. A $z1$ kimeneten a regiszter illesztése miatt egy óraciklussal később érhető el az eredmény (8.5. ábra).

A következő példában megpróbáljuk összehasonlítani, mi történik, ha az *ap_ctrl_hs* függvény szintű protokollra alkalmazunk, illetve nem regisztereket. Az alábbi Simulink modellben a felső modul esetében nincsenek, míg az alsó modul esetében vannak regiszterek alkalmazva.

A 8.6. ábrán mindkét modulra az alapértelmezett *ap_ctrl_hs* protokoll van alkalmazva. Az alsó modulban a kimenetekhez regiszterek vannak illesztve.



8.6. ábra. A HLS-ben generált két szorzómodul összehasonlítása

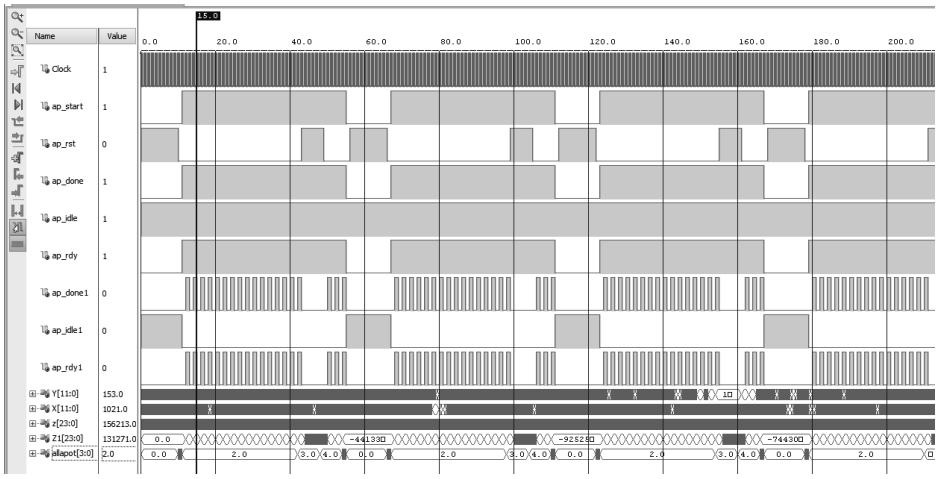
```
dout_t szorzas_a(din_t x, din_t y)
{// x,y: bemeneti adat
//kényszerfeltétel ap_ctrl_hs interfész alkalmazására valamint
    regiszter alkalmazása a kimeneten
#pragma HLS INTERFACE ap_ctrl_hs register port=return
dout_t tmp=0;
tmp = (x * y);
return tmp;
}
```

A 8.1. táblázat tartalmazza a jelek elnevezéseit.

8.1. táblázat. Jelek elnevezése

Jel megnevezése	Melyik modulhoz kapcsolódik a jel	Jel típusa
clock	közös	órajel (bemenet)
ap_start	közös start	bemenet
ap_rst	2	reset jel (bemenet)
ap_done	1	a műveletvégzés befejeződött, kimeneti függvényszintű vezérlőjel
ap_idle	1	a modul készenléti állapotban van, kimeneti függvényszintű vezérlőjel
ap_rdy	1	az eredmény elérhető, kimeneti függvényszintű vezérlőjel
*ap_done*1	2	a műveletvégzés befejeződött, kimeneti függvényszintű vezérlőjel
*ap_idle*1	2	a modul készenléti állapotban van, kimeneti függvényszintű vezérlőjel
*ap_rdy*1	2	az eredmény elérhető, kimeneti függvényszintű vezérlőjel
Y	közös k	bemeneti adat
X	közös	bemeneti adat
z	1	első modullal számolt eredmény
*Z*1	2	második modullal számolt eredmény
allapot		a szimuláció során a szimuláció állapotát jelzi, egyszerűsítette a különböző tesztfázisok azonosítását

A 8.7. ábrán értelmezhető a bemutatott áramköri modul portjelein az értékek alakulása.



8.7. ábra. Szimuláció során elért eredmények

Követve a szimulációs eredményeket (8.7. ábra), a következő következtetéseket lehet levonni:

1. Az *ap_start* jelnek az első modulra nézve nincsen különösebb szerepe. Logikai '1'-re való kapcsolásával azonnal, ugyanabban az óraciklusban az *ap_done* és ap_ready kimenetek is egyesre váltanak, és folyamatosan a logikai '1' szinten maradnak. Alacsony logikai szintre való kapcsolást követően az *ap_done* és *ap_ready* kimenetek is alacsony logikai szintre kapcsolnak. Viszont ha megfigyeljük az *ap_idle* kimeneti vezérlő bit logikai szintjét, levonható a következtetés, hogy az *ap_start* bemeneti vezérlőjeltől függetlenül folyamatosan logikai '1' állapotban van. A logikai '1' állapot azt jelenti, hogy a modul működik, logikai '0' pedig azt, hogy készenléti állapotban van.

2. Ha megfigyeljük a második modul vezérlőjeleit, az *ap_done*1 és *ap_ready* jelek a műveletvégzés engedélyezését követően (*ap_start*=1) nincsenek folyamatosan logikai '1' állapotban. A szimuláció szerint csak minden második órajelre történik új adatbeolvasás.

3. Második modul esetében

   a) Ha az *ap_start*='0', *ap_rst*='1', vagyis a modulra kiadtunk egy reset jelet és a műveletvégzést nem indítottuk:

i. a szimuláció kezdetén a $z1$ kimenet határozatlan,

ii. az ap_idle1 jel logikai egyesben van, a modul várakozik a műveletvégrehajtás elkezdésére.

b) Ha az ap_start=0, ap_rst=0, mivel a műveletvégzés még nem kezdődött el, a modul várakozási állapotban van (ap_idle='1').

c) Ha az ap_start='1', ap_rst='0', az ap_start='1' jelezzük, hogy elkezdődhet a műveletvégzés (készen állnak az adatok a bemeneten), az ap_done1 illetve ap_ready1 vezérlő bitek jelzik, hogy a modul befejezett egy műveletvégzést (ap_done1) és készen áll a következő adat fogadására.

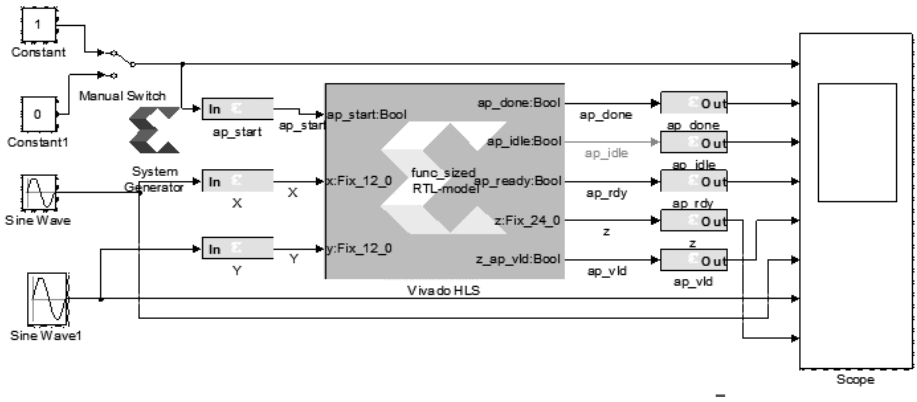
d) Ha az ap_rst1 a modul nem végez műveletet, az állapot bitek ap_done1 és ap_read1 logikai '0' szinten maradnak.

Szorzó modul, a szorzás eredménye argumentumként van visszatérítve

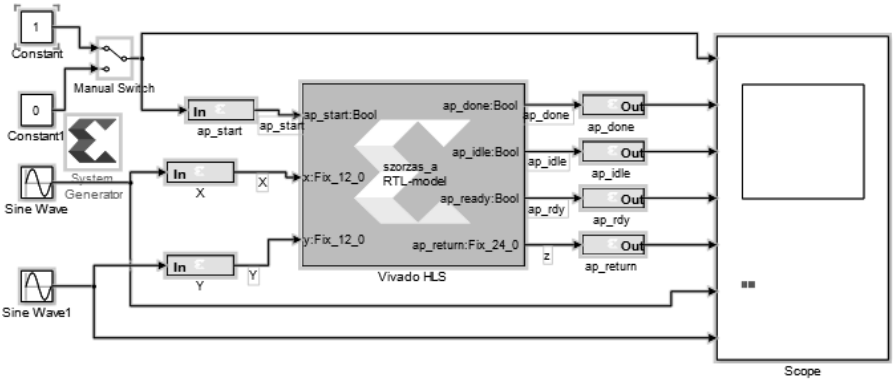
```
#include "szorzas_b.h"

void szorzas_b(din_t x, din_t y, dout_t *z) {
\\x,y: bemeneti adatok
\\z: eredmény visszatérítése (modulból kimeneti jelt
    eredményez)
\\tmp: átmeneti segédváltozó
  int tmp;
  tmp = (x * y);
  *z=tmp;
  return ;
}
```

Összehasonlítva a két változatban visszatérített értéket, ugyanazt a függvény szintű protokollt alkalmazva mindkét változatra, megfigyelhető, hogy az argumentumként visszatérített érték esetében (8.8. ábra) a z kimenet mellett megjelenik z_ap_vld állapotjel, amelynek logikai '1' állapota jelzi, hogy érvényes az adat. A függvény argumentumából származtatott kimeneti porthoz alapértelmezetten ap_vld vezérlőjel van csatolva.

```
dout_t szorzas_a(din_t x, din_t y)


void szorzas_b(din_t x, din_t y, dout_t *z)
```



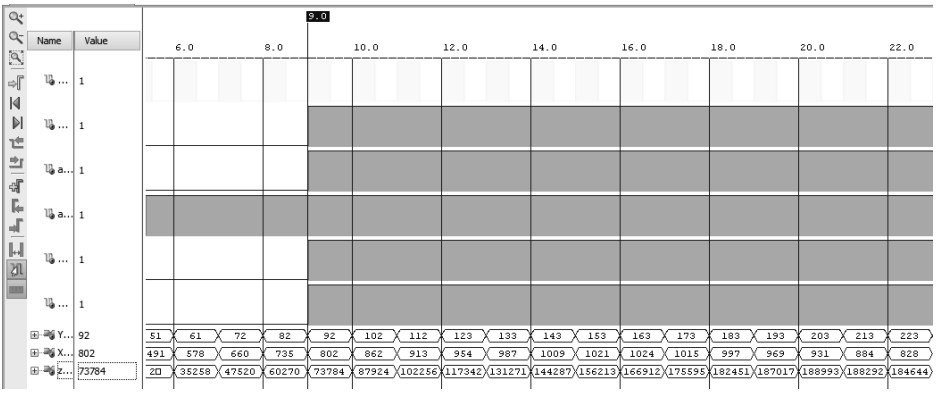
8.8. ábra. Argumentumként visszatérített eredmény esetében az ap_ctrl_hs protokollra a vezérlőjelek



8.9. ábra. Függvényértékként visszatérített eredmény esetében ap_ctrl_hs protokollra a vezérlőjelek

Amint megfigyelhető a 8.8. ábrán, a függvény argumentumként visszatérített eredmény, az *ap_ctrl_hs* hez rendelt vezérlőjeleken kívül megjelenik az *ap_valid* port szintű vezérlőjel.

Mivel nem regiszterelt az *ap_ctrl_hs*, a kimenet azonnal érvényes. A Simulink szimuláció szerint, amelyik pillanatban jelezzük, hogy van érvényes bemeneti adat, az *ap_vld* vezérlő jel is logikai '1'-re vált, amint a 8.10. ábrán megfigyelhető.



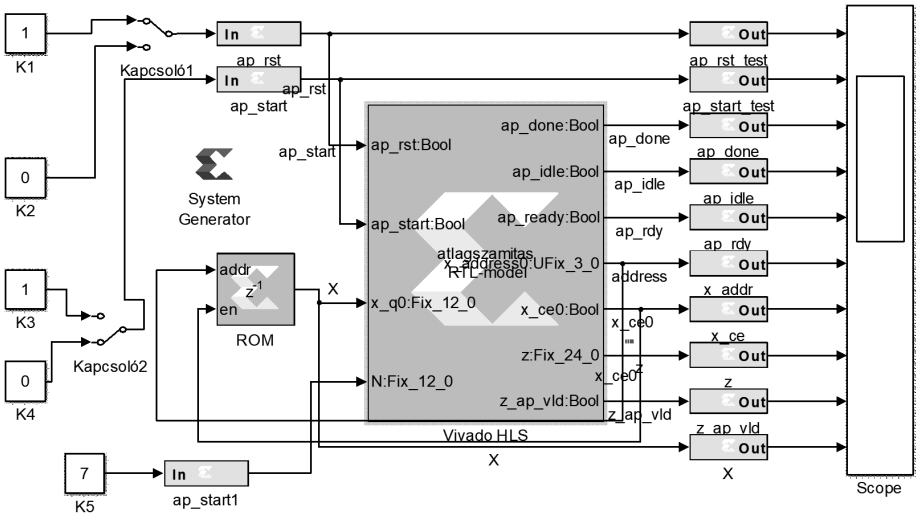
8.10. ábra. Szimulációs eredmény nem regiszterelt kimenetre

Tömb elemeinek összege

A következő példában HLS-ben tervezett áramkör kiszámolja N szám összegét. A példákban a tömb szintű bemenetekre alkalmazható interfész megoldásokat mutatunk be. Az alábbi változat szerint a z kimenethez nincsen regiszter hozzárendelve, és az alapértelmezett függvény és port szintű protokoll van alkalmazva.

```
#include "osszegszamitas.h"

void osszegszamitas(din_t x[M], din_t N, dout_t *z)
{
  int tmp;
  int i;
  tmp=0;
  for (i=0;i<N;i++)
  tmp =tmp+x[i];
  *z=tmp;
  return ;
}
```

A következő megoldásban szintén az alapértelmezett protokoll van alkalmazva, viszont a z kimenethez regiszter van illesztve. Mindkét esetben az eredményt függvény argumentumként térítjük vissza. Szimuláció során összehasonlítjuk a szimuláció eredményét, ha a z kimenethez regisztert csatolunk. A direktívát a programkódba illesztettük.



8.11. ábra. Vektor elemeinek számítása tömbvázlat

```
#include "osszegszamitas.h"

void osszegszamitas(din_t x[M], din_t N, dout_t *z)
{
  #pragma HLS INTERFACE register port=z
  int tmp;
  int i;
  tmp=0;
  for (i=0;i<N;i++)
  tmp =tmp+x[i];
  *z=tmp;
  return ;
}
```

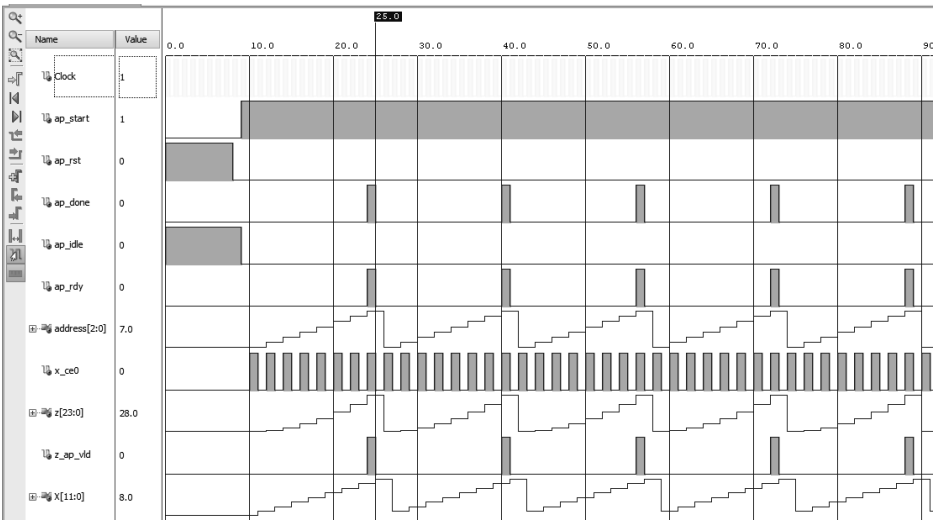
A fenti megoldás szerint alapértelmezett függvény és port szintű vezérlőjelek vannak alkalmazva. Ha a bemenet tömb, a bemeneti tömbhöz alapértelmezetten bemeneti interfészként egyportos RAM interfész van alkalmazva.

Értelmezzük a tervezést követően, milyen vezérlőjelekkel találkozunk a modulon:

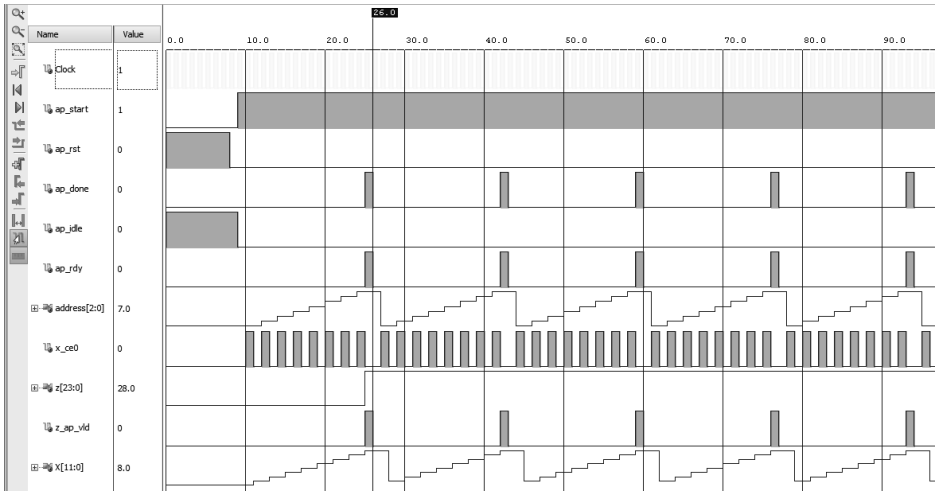
- függvény szintű (vagy blokk szintű) protokolljelek
  - *ap_start*

  - *ap_done*
  - *ap_idle*
  - *ap_rdy*
  - *ap_rst*
- port szintű protokoll (argumentumhoz rendelt) jelek
  - *ap_vld* érvényes a *z* kimenet
  - *x_address*0 az *x* tömbhöz rendelt memória címzésére szolgál
  - *x_ce* az x tömbhöz rendelt órajel engedélyező vezérlőjel, ütemezi a memóriából való olvasást

Mivel az *x* bemenet a függvény argumentumában deklarált tömb, alapértelmezetten BRAM típusú interfészként van kezelve. Ennek alapján a memória vezérlésére létrejöttek az *x_address*, *x_ce* vezérlőjelek. A szemléltetett példában az adatokat egy ROM memória szolgáltatja. A szimuláció során elért eredmények a következő ábrákon vannak szemléltetve: 8.12., illetve 8.13. ábra.



8.12. ábra. Szimuláció tömb elemeinek összege



8.13. ábra. Szimuláció tömb elemeinek összege

A 8.12. ábra szerinti szimuláció esetében a kimeneti adat porthoz (z) nincsen regiszter illesztve. A szimuláció során a tömb elemeinek a száma 7. A szimuláció alapján is értelmezhető, hogy hét lépésben történik (x_ce0 jel) az adatok beolvasása és a nyolcadik lépésben érvényes az eredmény a kimeneten. Mivel a kimenethez nincsen regiszter csatolva, a részleges összeg folyamatosan megjelenik a kimeneten.

A kimeneti adat porthoz (z) regiszter van illesztve. A szimuláció során a tömb elemeinek a száma 7. A tömb értékei a ROM memóriában vannak eltárolva. Mivel a kimenethez regiszter van csatolva, a kimeneten nem a részösszeget, hanem egy lépéssel korábbi eredményt tárol a regiszter. Az *ap_ready* és *ap_done* vezérlőjelek egy óraciklust késnek a 8.13. ábrán bemutatott eredményekhez képest.

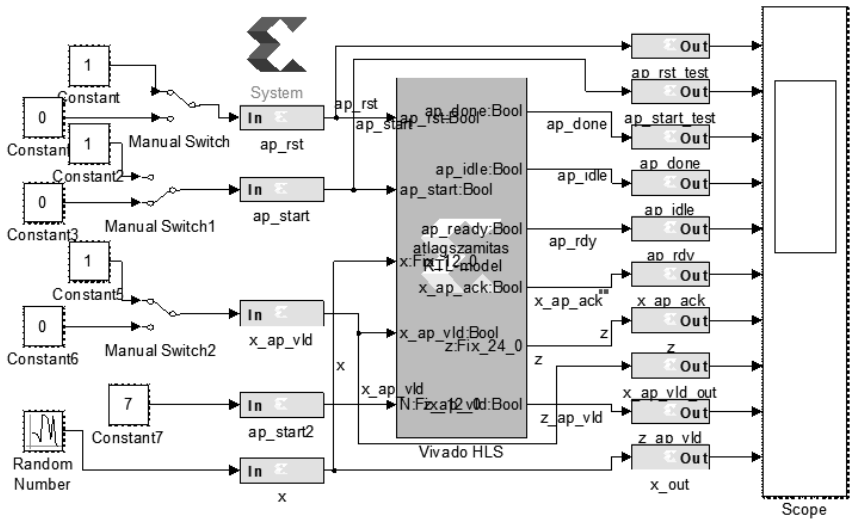
Összehasonlítva a két szimuláció eredményét, a következő következtetéseket tudjuk levonni:

- Az első változat szerint, amikor nincsen a kimenethez regiszter csatolva, a z kimeneten működés közben megjelenik a részleges eredmény is. A második változatban csak a végleges eredmény jelenik meg. A következő művelet végéig megmarad az azelőtti műveletvégzési ciklusból a művelet eredménye.
- A második változatban, az utolsó memória olvasáshoz viszonyítva, egy órajellel később aktívak az *ap_done*, *ap_ready*, *ap_vld* jelek. Ebben az esetben a műveletvégzés egy óraciklussal többet tart.

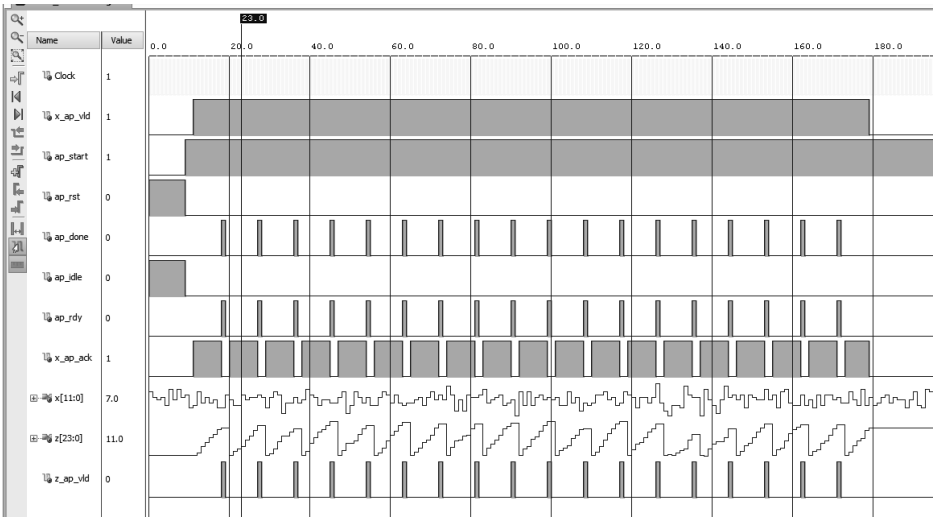
Az alábbi példában az x bemeneti porthoz kézfogásos protokollt (*ap_hs*) rendeltünk. Ez alapján a tömb elemeit egyenként töltjük be a modulba. A protokoll alapján az x port jelhez az *x_ap_vld* és az *x_ap_ack* vezérlőjelek vannak csatolva. Az alábbi programrészben értelmezhető a programsor, amelyben az x bemenethez rendeltük az ap_hs protokollt.

```
void osszegszamitas(din_t x[M], din_t N, dout_t *z) {
  #pragma HLS INTERFACE ap_hs port=x
   int tmp;
   int i;
   tmp=0;
   for (i=0;i<N;i++)
   tmp =tmp+x[i];
   *z=tmp;
   return ;
}
```

A 8.14. ábrán a tömb elemeinek számítására létrehozott *System Generator* tömbvázlat van bemutatva. Az x bemenethez ap_hs protokoll van rendelve. Vezérlőjelek *x_ap_vld* bemenet és *x_ap_ack* kimenet.



8.14. ábra. Tömb elemei összegének a számítása, létrehozott System Generator modul



8.15. ábra. Tömb elemeinek összege szimuláció

A 8.15. ábrán szemléltetett tömb elemei összegének a számítására alkalmazott szimulációban a bemeneti adatok szekvenciálisan vannak betolva a modulba. Az *ap_rst* logikai '0'-ra való kapcsolását követően az *ap_start* vezérlő jelet logikai '1'-re kapcsoljuk, majd az *x_ap_vld* jellel jelezzük, hogy az adatok készen állnak a beolvasásra Az *x_ap_ack* kimeneti jel logikai '0'-ra való változása nyugtázza, hogy az adatok beolvasása megtörtént. A beolvasást követő óraciklusban az *ap_done*, *ap_ready*, *ap_vld* vezérlőjelek is egy óraciklus időre átváltanak logikai '1' szintre.

A következő példákban azt szemléltetem, hogy ciklusra és tömbre kényszerfeltételeket alkalmazva hogyan lehet optimalizálni, csökkenteni a számításhoz szükséges óraciklusok számát. Ha a bemeneti x tömböt két tömbre tördeljük, fele idő alatt kiszámolható a tömb elemeinek összege. A megvalósításhoz a bemeneti tömböt kell feldarabolni két tömbre, és a ciklust ki kell fejteni (*unrolling*) úgy, hogy egy ciklus helyett párhuzamosan két ciklus legyen. Az egyik ciklus végzi az összegzést az egyik tömb elemeivel, a másik ciklusban pedig a másik tömb elemeire történik az összegzés.

```
void osszegszamitas(din_t x[M], din_t N, dout_t *z) {
  #pragma HLS ARRAY_PARTITION variable=x block factor=2 dim=1
  int tmp;
  int i;
  tmp=0;
```

```
  ciklus:for (i=0;i<N;i++)

  #pragma HLS UNROLL factor=2
  tmp =tmp+x[i];
  *z=tmp;
  return ;
}
```

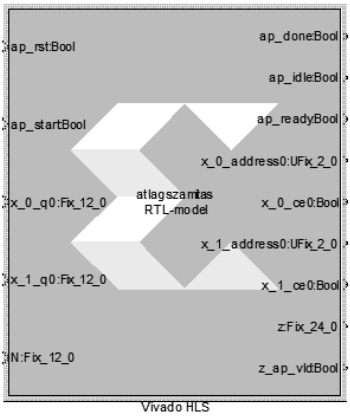
A fenti program alapján két kényszerfeltétel van alkalmazva. Egy a bemeneti x tömb particionálására:

```
#pragma HLS ARRAY_PARTITION variable=x block factor=2 dim=1
```

valamint egy a ciklus kifejtésére:

```
#pragma HLS UNROLL factor=2
```

Eredményül a 8.16. ábrán látható áramköri elemet kapjuk: egy-egy memóriát kell illeszteni az x_0 és x_1 bemenetekre. A memóriák vezérlésére az $x\_0\_address$, x_0_ce, $x\_1\_address$ és x_1_ce vezérlőjelek szolgálnak. A memóriamodulok illesztését a 8.11. ábra szerint kell megvalósítani.

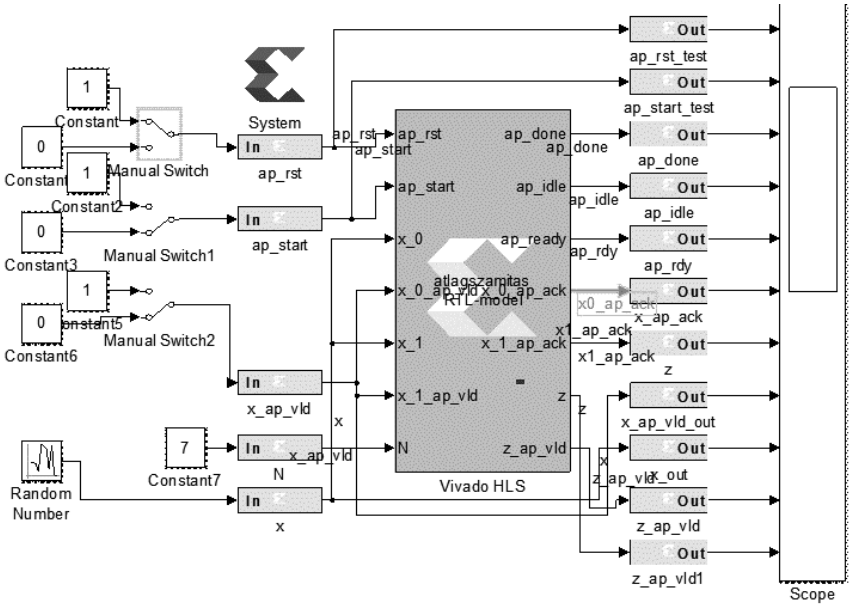


8.16. ábra. Tömb elemei összegének számítása, HLS-ből létrehozott System Generator modul

A 8.16. ábrán alapértelmezett függvény szintű protokoll, alapértelmezett port szintű interfészek, vagyis memóriainterfész van alkalmazva az x bemenetre.

Az alábbi programrész alapján az x tömb elemeit sorosan töltjük fel *ap_hs* protokollt alkalmazva. Az x tömböt felosztva két tömbre és ciklusra, összhangban a memória felosztásával, a megfelelő kényszerfeltételt szabjuk, felére csökkenthető az összegszámításhoz szükséges óraciklusok száma.

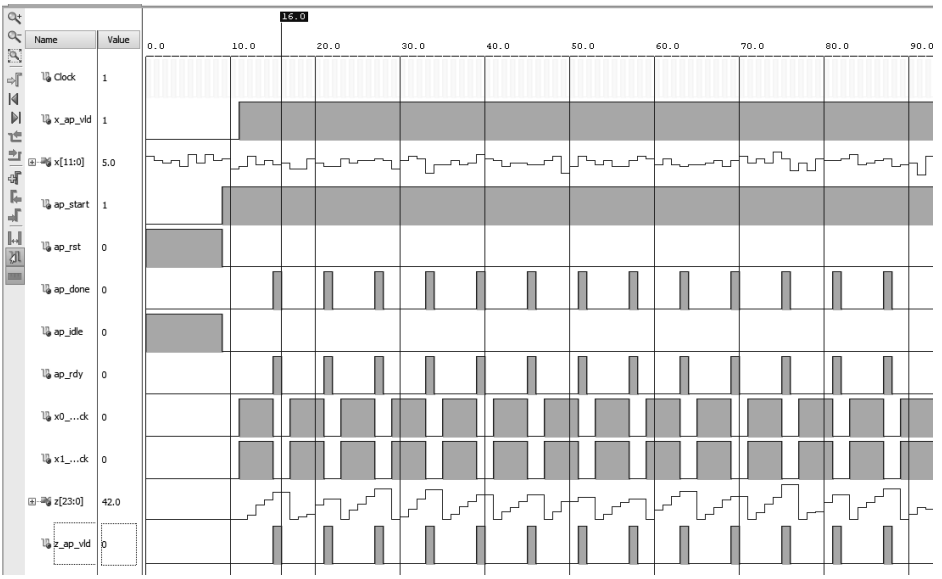
```
void osszegszamitas(din_t x[M], din_t N, dout_t *z) {
  #pragma HLS ARRAY_PARTITION variable=x cyclic factor=2 dim=1
  #pragma HLS INTERFACE ap_hs port=x
   int tmp;
   int i;
   tmp=0;
   ciklus:for (i=0;i<N;i++)

  #pragma HLS UNROLL skip_exit_check factor=2
   tmp =tmp+x[i];
   *z=tmp;
   return ;
}
```



8.17. ábra. Particionált bemeneti tömb

A 8.17 ábrának megfelelően az x bementi tömb két kisebb tömbre van particionálva, az (*ap_hs*) protokoll , két sínen szekvenciálisan biztosítja

az adatbetöltést, közös *ap_vld* vezérlőjelt alkalmazva mindkét bemenetre (x_0, x_1).



8.18. ábra. Szimuláció eredménye particionált bemeneti tömbökre

Az egyik ciklus végzi az összegzést az egyik tömb elemeivel, a másik ciklus pedig a másik tömb elemeivel. A tömb particionálása többféleképpen valósítható meg. Az egyik esetben, ha a block típusú felosztást alkalmazzuk, a tömb elemeinek az első része az első tömbbe kerül, míg a második fele a második tömbbe. Jelen esetben a tömböt két kisebb tömbre particionáltuk. Block paraméter alkalmazása a tömb feldarabolására:

```
#pragma HLS ARRAY_PARTITION variable=x block factor=2 dim=1
```

8.19. ábra. Particionálás, első változat

Ha a ciklikus (cyclic) típusú felosztást választjuk, jelen esetben a következő a felosztás:

```
pragma HLS ARRAY_PARTITION variable=x cyclic factor=2 dim=1
```

8.20. ábra. Particionálás, második változat

A különbség a *cyclic* és *block* paraméterek esetében a következő:

- Blokk esetében az egymást követő elemek egy tömbbe kerülnek, míg a ciklikus felosztás során az első elem az első tömbbe, a második a második tömbbe, a harmadik első tömbbe, a negyedik második tömbbe stb.
- A ciklikus szétdarabolás lehetővé teszi, hogy a két kiterjesztett ciklus párhuzamosan működjön, ezáltal fele idő alatt elvégezve a műveleteket. A block típusú felosztás során a két ciklus egymás után végzi a számításokat.

IRODALOMJEGYZÉK

[1] D. D. Gajski, N. D. Dutt, A. C.-H. Wu, and S. Y.-L. Lin, *High-level Synthesis: Introduction to Chip and System Design*. Norwell, MA, USA: Kluwer Academic Publishers, 1992.

[2] P. P. Chu, *FPGA prototyping by VHDL examples: Xilinx Spartan-3 version*. John Wiley & Sons, 2011.

[3] P. P. Chu, *RTL hardware design using VHDL: coding for efficiency, portability, and scalability*. John Wiley & Sons, 2006.

[4] IEEE, IEEE standard VHDL language reference manual, *ANSI/IEEE Std 1076-1993*, i–246, 1994.

[5] IEEE, IEEE standard VHDL language reference manual, *IEEE Std 1076-2000*, i–290, 2000.

[6] IEEE, IEEE standard for VHDL register transfer level (RTL) synthesis, *IEEE Std 1076.6-2004 (Revision of IEEE Std 1076.6-1999)*, 1–118, Oct 2004.

[7] IEEE, IEEE standard vhdl language reference manual, *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, c1–626, Jan 2009.

[8] J. Vásárhelyi, *Hardver leíró nyelvek* – VHDL. Accessed: 2018-10-28.

[9] G. Hosszú, P. Keresztes, *VHDL-alapú rendszertervezés*. Szak Kiadó Kft., 2012.

[10] A. Fodor, Zs. Vörösházi, *Beágyazott rendszerek és programozható logikai eszközök*: egyetemi tananyag, 2011. Bibliogr.: 250–251.

[11] V. A. Pedroni, *Circuit design with VHDL*. MIT press, 2004.

[12] D. L. Perry, *VHDL: Programming by Example*. New York, NY, USA: McGraw-Hill, Inc., 4 ed., 2002.

[13] D. Ltd., *VHDL Golden Reference Guide: A Concise Guide to IEEE Std 1076-2002*. Ringwood, UK Doulos Ltd., 2003.

[14] Xilinx, *Vivado Design Suite Tutorial Programming and Debugging*. Digilent.

[15] Xilinx, *Vivado Design Suite User Guide: Model-Based DSP Design Using System Generator*. Xilinx.

[16] D. Inc., *ZYBO FPGA Board Reference Manual*. Digilent.

[17] Xilinx, *Vivado HLS optimization methodology guide*, 2018.

[18] V.-H. Xilinx, *Vivado design suite user guide-high-level synthesis*, 2016.

ABSTRACT

Reconfigurable Digital Circuits Design and Test Methods

Nowadays, it is very important to implant an idea into practice as soon as possible. In digital circuit design, the FPGA circuits and the use of the suitable design methods play a particularly important role.

The hardware design languages (VHDL, Verilog) have an important role in circuit design. Current design tools, besides HDL-based design, have now been able to specify the functionality of a circuit in different high-level programming languages (C, C ++, System C, Matlab).

The book summarizes the design and testing methods used in FPGA-based circuit design. The first chapter provides insights into design methods, detailing the basic steps of digital circuit design and testing.

The first chapter presents the design methods of digital systems, the description of the digital circuits in different domains such as behavioural, structural or physical. It describes the steps to be followed to give a physically functional digital circuit, starting from formulating the requirements. It also gives an introduction to VHDL description-based digital hardware design. It highlights the basic standards associated with VHDL. It presents the grammatical elements, the specifics of VHDL, the structure of the VHDL language, the use of libraries, and the connection between the planned hardware and the VHDL.

Chapter Two details the sequential VHDL instructions. In the next chapter, concurrent VHDL instructions are introduced. Multiple circuits' functionality can be described also by sequential and concurrent VHDL expressions. Sequential- and concurrent-expressions-based implementation for basic digital circuits are discussed by comparative examples.

The following section presents various possible testing solutions of the designed digital circuit. In the first part of the chapter, a simulation-based testing of the circuit is presented with an example of how to create the testbench. In the second half of the section, the-run time testing of an FPGA-implemented circuit is presented with the integration of ILA logic analyzer into FPGA circuit.

Chapter five details the design steps for a finite-state machine with datapath (FSMD) and its VHDL-based implementation. In chapter six, an

FSMD-based implementation in VHDL in Vivado environment of a specific task is presented.

The last two chapters discuss the high level synthesis (HLS) based design solution of the circuit. In Chapter seven, the use of System Generator tool for digital circuit design and testing is detailed. From the point of view of rapid prototyping of an application the hardware co-simulation is a useful design and test solution. The chapter presents the hardware co-simulation based implementation and test for pulse width modulation signal generator.

In the last chapter, the basics of circuit design based on High Level Synthesis are detailed: the description of the constraint in design, the description of the operation of the circuit, the interfaces and the port signals in C high level programming language. A series of simple examples introduces the reader to HLS based circuit design details.

Based on the knowledge presented in the book the reader will have an insight into the choice of appropriate design tools and design methods as well as the efficient design testing solutions.

REZUMAT

Circuite digitale reconfigurabile. Metode de proiectare și testare

În zilele noastre este foarte importantă concretizarea unei idei în practică în cel mai scurt timp posibil. În proiectarea circuitelor digitale, aplicarea circuitelor FPGA și utilizarea unor metode de proiectare adecvate joacă un rol deosebit de important.

Utilizarea limbajelor de descriere hardware (VHDL, Verilog) are un rol deosebit de important în proiectarea și simularea circuitelor digitale electronice.

Instrumentele de proiectare actuale, în afară de proiectarea bazată pe HDL, permit specificarea funcționalității circuitelor digitale prin utilizarea diferitelor limbaje de programare de nivel înalt (C, C ++, System C, Matlab).

Cartea rezumă metodele de proiectare și testare utilizate în proiectarea circuitelor bazate pe FPGA. Primul capitol prezintă metodele de proiectare a sistemelor digitale, descrierea circuitelor digitale în diferite domenii cum ar fi funcțional, structural sau cel fizic.

Sunt descrise etapele care trebuie urmate pentru a realiza un circuit digital funcțional fizic, pornind de la formularea cerințelor. Oferă, de asemenea, o introducere privind proiectarea de hardware digital pe baza descrierii în VHDL. Sunt evidențiate standardele de bază asociate limbajului VHDL. În cadrul capitolului sunt prezentate elementele gramaticale, specificul VHDL, structura limbajului VHDL, utilizarea bibliotecilor, legătura dintre hardware-ul planificat și VHDL.

Capitolul al doilea detaliază instrucțiunile secvențiale VHDL. În capitolul următor sunt introduse instrucțiuni concurente VHDL. Funcționalitatea mai multor circuite poate fi descrisă și prin expresii VHDL secvențiale și concurente. Implementarea bazată pe expresii secvențiale și concurente pentru circuitele digitale de bază este discutată prin exemple comparative.

Următoarea secțiune prezintă diverse soluții posibile de testare ale circuitelor digitale. În prima parte a capitolului este prezentată soluția de testarea a circuitului digital prin simulare detaliat printr-un exemplu. În a doua jumătate a secțiunii se prezintă soluția de testarea în timpul funcționării unui circuit, implementat pe FPGA, prin integrarea analizorului logic ILA în circuitul FPGA.

Capitolul cinci detaliază etapele de proiectare ale unui automat cu stări finite cu cale de date (FSMD) și implementarea în VHDL.

În capitolul șase este prezentată o implementare a unui FSMD în VHDL în mediul Vivado pentru o aplicație specifică.

Ultimele două capitole tratează soluția de proiectare a circuitelor digitale bazată pe sinteza de nivel înalt (HLS). În capitolul şapte este detaliat instrumentul System Generator utilizat pentru proiectarea și testarea circuitelor digitale. Co-simularea hardware este un instrument util din punctul de vedere al realizării şi testării rapide a unui aplicații prototip. În cadrul capitolului este prezentată soluția propusă pentru un generator de semnale modulate în lățime.

În ultimul capitol sunt prezentate noțiunile de bază ale proiectării circuitelor bazate pe sinteze de nivel înalt. Sunt tratate soluții de sinteză de nivel înalt, pentru impunerea unor constrângeri de proiectare, descrierea funcționării circuitului, soluții pentru implementarea interfețelor protocoalelor de comunicare între module, în limbajul de programare C. Printr-o serie de exemple sunt prezentate detalii de proiectare a circuitelor bazate pe sinteză de nivel înalt.

Bazându-se pe cunoștințele prezentate în carte, cititorul va avea o perspectivă asupra alegerii instrumentelor și a metodelor de proiectare adecvate, precum și a soluțiilor eficiente de testare a circuitelor digitale electronice.

A SZERZŐRŐL

A szerző a marosvásárhelyi Petru Maior Egyetem Villamosmérnöki Karán végzett és a Brassói Transilvania Egyetem Villamosmérnöki és Számítástechnika Tudományok Karán doktorált.

1997-től hat évet rendszergazdaként tevékenykedett a Dimitrie Cantemir magánegyetem keretében. 1997-től társult oktatóként dolgozott egyetemi gyakornokként a Petru Maior Egyetemen és 1998-tól mint oktató a Dimitrie Cantemir Egyetemen. 2001-től társult oktatóként kezdett el dolgozni a Sapientia Erdélyi Magyar Tudományegyetemen, majd 2003 őszétől kezdődően főállásban egyetemi tanársegédként folytatta munkásságát a Sapientia EMTE Villamosmérnöki Karán.

2006-tól a Sapientia Erdélyi Magyar Tudományegyetem főállású adjunktusa. A fontosabb oktatott tantárgyak laborgyakorlatok szintjén: számítógép architektúrák, operációs rendszerek, mesterséges intelligencia, speciális fejezetek mesterséges intelligenciából, újrakonfigurálható digitális áramkörök, beágyazott elektronikai rendszerek, softcomputing módszerek.

Alapképzésen operációs rendszerek, újrakonfigurálható digitális áramkörök, valamint mesterképzésen beágyazott elektronikai rendszerek, fejlett irányítási rendszerek a mechatronikában tantárgyakat oktatja.

Doktori munkája során neuronhálók FPGA áramkörben való megvalósítását tanulmányozta. A doktori munkája, majd azt követően tudományos munkája során RBF, CMAC, KOHONEN neuronhálókra, valamint Mamdani és Takagi Sugeno, ANFIS (Adaptív Neuro Fuzzy Következtető Rendszer) modellekre, folytonos wavelet transzformáció, valamint egyéb algoritmusokra dolgozott ki párhuzamosított, FPGA áramkörben implementálható modelleket.

Scientia Kiadó
400112 Kolozsvár (Cluj-Napoca)
Mátyás király (Matei Corvin) u. 4. sz.
Tel./fax: +40-364-401454
E-mail: scientia@kpi.sapientia.ro
www.scientiakiado.ro

Korrektúra:
Szenkovics Enikő

Műszaki szerkesztés:
Brassai Sándor Tihamér

Tipográfia:
Könczey Elemér

Készült a kolozsvári Idea nyomdában
70 példányban

A jegyzet bemutatja a digitális áramkörök alapvető tervezési és tesztelési módszereit és a témához kapcsolódó alapvető fogalmakat. A jelenlegi tervezőeszközök, a hardverleíró nyelv alapú tervezés mellett, mára már lehetővé teszik az áramkör működésének a specifikálását különböző magas szintű programozási nyelveken is (C, C++, System C, Matlab).

A könyv összefoglalja az alapvető áramkörök szekvenciális és konkurens utasításokkal való megvalósítását, illetve véges állapotú automaták és komplex állapotgépek VHDL alapú specifikálását. Szemlélteti a tervezett digitális áramkör tesztelését, szimulációval történő ellenőrzését, valamint FPGA-ba integrált logikai analizátor alkalmazásával működés közbeni tesztelését.

ISBN 978-606-975-020-9