Datapath Synthesis Using Adiabatic Logic

LÁSZLÓ VARGA, GÁBOR HOSSZÚ, FERENC KOVÁCS
Department of Electron Devices
Budapest University of Technology and Economics
Goldmann Gy. 3, H-1111 Budapest
HUNGARY
vargal@nimrud.eet.bme.hu http://nimrud.eet.bme.hu/vargal

Abstract: - We present an integer linear programming formulation and a heuristic scheduling approach for high-level synthesis to synthesize pipeline datapaths using adiabatic logic. Adiabatic logic families that rely on charge recovery is attractive to achieve low energy dissipation, and these circuits are most suitable for DSP applications. However, existing scheduling techniques are incapable to deal with scheduling of adiabatic circuits, since they do not take multiplexer delay into account. We also present a description technique to perform functional simulation of the synthesized adiabatic datapath together with the other part of a digital system.

Key-Words: - Low power logic, adiabatic computing, high-level synthesis, scheduling

1 Introduction

Power dissipation becomes a major concern in VLSI design as the feature size decreases and the corresponding chip density increases. The trend is driven primarily by the expensive packaging requirements and by the demand for portable devices, where the battery life is of primary concern.

The adiabatic charge recovery logic is a promising approach to design VLSI circuits with extremely low energy dissipation. Such logic circuits achieve low energy consumption by restricting the currents to flow across devices, and by recycling the energy stored in their capacitors. This requires an ac power supply rather than dc. The adiabatic charge recovery logic have substantial advantages in energy consumption over static CMOS especially at low operating frequencies. Although this gain diminishes at higher frequencies, the adiabatic circuits well above 200 MHz are still about 2 times more energy-efficient than static CMOS [1]. In addition, their energy-efficiency for a given frequency can be improved by transistor size optimization [2].

In recent years several adiabatic techniques have been proposed [3][4][5]. These adiabatic logic families are functionally complete, and they are most suitable for arithmetic functions. A low-power adiabatic microprocessor [6] and an adiabatic FIR filter [7] show their potential for real applications. Although previous work implemented various logic functions with adiabatic circuits, no work was done to support automatic synthesis of complete adiabatic system. While the adiabatic logic is not suitable for memory intensive applications, due to its inherent micropipeline structure it is especially attractive for embedded DSP functions, where a sequence of operations are performed on consecutively initiated data. Time often plays an important role in these real-time embedded systems. Obviously, increasing the clock frequency is one way to improve the throughput, but adiabatic circuits cannot be clocked at very high frequencies. However, architectural optimizations, such as parallelism exploitation and pipelining are much more effective in increasing throughput than bare clock speedup. Therefore, it is important to be able to provide a synthesis system producing high-quality application-specific datapaths. Pipeline scheduling techniques can be found in the literature [8][9]. However, these techniques are incapable to deal with scheduling of adiabatic circuits, since they do not take multiplexer delay into account.

In this paper we describe an integer linear programming (ILP) formulation and a heuristic scheduling technique to synthesize pipeline datapath using adiabatic circuits. The time constraints are given in data initiation interval and the maximal allowed schedule length. Both approaches produce a pipeline schedule using a minimal number of functional units, but the heuristic approach may fail to schedule all operation within the allowed schedule length in some cases. We also present a description technique to perform functional simulation of the synthesized adiabatic datapath together with the other part of a digital system.

2 Target architecture

In adiabatic logic, the flow of data through cascaded gates is controlled by multi-phase clock. The adiabatic logic computes only one logic level per phase, therefore we need multiple phases to implement a multilevel logic function. We use four-phase clock with 90° phase lag, where each clock phase repeats the charge, hold, discharge and the wait periods [10]. The inputs of a logic gate must be stable during the charge period, and the logic gate maintains a valid output during the hold period. The discharge period is used to recover the energy stored in the output capacitor, and during the wait period new inputs are being prepared by the previous gate, which is in the charge period. In this way, the logic gates are pipelined without any pipeline registers.

2.1 Functional units

The functional units (FUs) are designed in a pipeline structure by using buffers for maintaining the pipeline. In this way, an adiabatic FU is itself a pipeline, it can execute a new operation for every clock cycle (cycle of four periods), but its latency is usually larger than a clock cycle. For example, the carry-lookahead adder (CLA), which is the best suitable adder for adiabatic implementation, requires O(log N) stages, where N is the bit width of the adder. A 16-bit CLA requires six stages, so its latency is 1.5 clock cycle, but it accepts new inputs for every clock cycle.

2.2 Multiplexers

Up to 4 to 1 multiplexer can be efficiently implemented in one complex gate, which requires one stage. If additional multiplexer input is needed, we use cascaded multiplexers.

2.3 Registers

Registers are built using flip-flops. An adiabatic flip-flop can be implemented by a ring of four logic gates. One gate contains logic to write in a new value, while the remainder gates are buffers to propagate the correct logic value. Instead of a ring of gates, a chain of buffers can also be used to temporarily store and propagate a logic value. There is no need control signal in this case, but it requires as many buffers as the delay of the chain.

2.4 Control Unit

The control unit is also implemented with adiabatic logic gates to control the adiabatic multiplexers. It consists of one or more rings of gates, where the number of logic gates in a ring is equal to the data initiation interval (DII). There are DII/4 complex gates in a ring, which are used to reset to the appropriate logic value, while the other gates are only buffers.

3 The ILP formulation

First, we give a list of notations which will be used throughout this paper:

o_i	---	the *i.* operation in the input description
s_i	---	earliest (as soon as possible) execution time of the operation o_i
l_i	---	latest (as late as possible) execution time of the operation o_i
$x_{i,t}$	---	is one if o_i starts in the clock phase t; otherwise it is zero.
t_i	---	the number of clock phases required by o_i to complete its task
d_i	---	the number of clock phases required by the multiplexer before o_i
L	---	the maximal allowed schedule length in clock phase
DII	---	data initiation interval in clock phase
M_k	---	the number of functional units of type k

3.1 Scheduling constraints

All operation has to be scheduled between their earliest and latest execution times. Since we don't know the multiplexer delays a priori, we can determine the time frames of the operations without considering the multiplexers only, using the as soon as possible (ASAP) and the as late as possible (ALAP) schedules. Incorporating the multiplexers into scheduling the actual time frame will be tighter. However, we use the ASAP and ALAP values to remove a number of trivial zero variables, thus reducing the complexity. So:

$$\sum_{t=s_i}^{l_i} x_{i,t} = 1, \quad \text{for all } i.$$

3.2 Dependency constraints

An FU receives its operands through multiplexer to perform the operation assigned to the unit. As described earlier, a multiplexing takes at least one

clock phase depending on the size of the multiplexer, therefore we must take its delay into account in scheduling. To keep the dependencies between operations as in the input description, an operation can start only if its predecessors have finished their operation. The corresponding inequalities are as follows:

$$\sum_{t=s_i}^{l_i} t * x_{i,t} + t_i + d_i - \sum_{t=s_j}^{l_j} t * x_{j,t} \le 0, \text{ for all } i, j \text{ such}$$

$$\text{that } o_i \text{ is a direct predecessor of } o_j$$

$$\sum_{t=s_i}^{l_i} t * x_{i,t} + t_i + d_i - L \le 1, \text{ for all } i \text{ such}$$

$$\text{that } o_i \text{ does not have a sucessor.}$$

3.3 Pipelining overlap constraints

An FU, which starts a calculation at time t, can accept the next data at time *t+4* due to its pipeline structure. An operation started at time *T* is considered to be occupying one FU for $T \le t < T+4$. This can be described using the following function for all *i*:

$$x'_{i,t} = 1 \text{ if } T + d_i \le t < T + d_i + 4$$
$$= 0 \text{ otherwise.}$$

The simultaneously used FUs of type *k* at time *t* is equal to the number of operations of type *k* which occupy an FU at time *t*. The required number FUs of type *k* is the maximal number of simultaneously used FUs of type *k* at any time. This is described by the following set of functions:

$$M_k = \max \sum_{t=1}^{L/DII} x'_{i,j+t*DII} \text{ for all } i, j, k \text{ where}$$
$$0 \le j < DII, \text{ and } o_i \text{ is of type } k.$$

3.4 Multiplexer constraints

We need an expression to calculate the number of multiplexer inputs for the FUs of type *k* as a function of the number of FUs of that type. Let this value be denoted by S_k. At the beginning $S_k=0$. From the input description we create a list, in which we collect all the different sources, form where the operations of type *k* receive operands, and count the number of occurrences (denoted by E_a) for each different source. For each list element we do the following: if the operand source is an input port, then $S_k=S_k+E_a$. If the operand source is an operation of type *i*, then $S_k=S_k+\min(E_a,M_i)$. If the operand source is a constant value, then $S_k=S_k+1$. This means, that a constant value can be selected among other constants in time, because it does not depend on other operations. An example of this calculations using the 16-point FIR filter is the following: the addition operations receive operands from 16 input ports, from 8 multiplication and from 6 addition operation. The multiplication operations receive operands from 8 addition operation and they receives 8 constant value. So $S_+=16+\min(8,M_*)+\min(6,M_+)$ and $S_*=\min(8, M_+)+1$. An FU of type *k* has two inputs, and we have M_k FUs of type *k*.

$$\frac{S_k}{2} \le \sum_{o_i} d_i \le \sum_{k=1}^{M_k} 4^{d_k} \text{ , for all } k$$
$$\text{where } o_i \text{ is of type } k$$

4 The heuristic algorithm

The algorithm is based on the uniformly distribution of the multiplexer inputs among the same type of FUs. To determine the size of the multiplexers before each FU, we need to know the number of FUs of each type, and the number of sources from where a certain FU receives operands. We first calculate the minimal number of FUs of each type without the actual scheduling, while satisfying the DII constraint. Then, based on the number of resources, the sizes and the delays of the multiplexers can be determined by analyzing the dependencies in the input description.

The minimal number of FUs of type *k* can be calculated by $M_k=\lceil N_k/\text{DII} \rceil$, where N_k is the number of operation of type *k* in the input description. Now, we can determine the size of the multiplexers by calculating the S_k values for each type of FUs as described in the previous section. Since an FU has two inputs, and we have M_k FUs of type k, the final number for S_k will be: $S_k=\lceil S_k/(2*M_k) \rceil$. This assumes a balanced distribution of the different operand sources among the same type of FUs, which will the task of the module allocation. We calculate the delay of the multiplexer before FUs of type *k* as the following: $d_k=\lceil (\log(S_k)/\log(2))/2 \rceil$.

For scheduling purpose, a multiplexer before an FU can be seen as the execution time of the operation is increased by the delay of the multiplexer: $t'_i = t_i + di$. We perform a modified list scheduling to schedule each operation. We calculate

the ASAP and ALAP values for each operation to determine its mobility using the t'_i values for the execution times. Our modified list scheduler maintains a reservation table for each type of FUs, which has M_k rows and as many columns as the maximal allowed schedule length. If an operation of type k is scheduled to begin its operation in the clock phase t, then the scheduler places a mark in every column of the table, which satisfies $C_k=t+4*n*DII$, where C_k is the number of the column of the reservation table for the FUs of type k, and n is integer number and $0<n<L/DII$. If this operation is the first occurrence of the type k, then the scheduler fully fills every other column with mark. The former takes care for pipeline operation, since the original list scheduling is incapable of deal with pipelined execution, and the later ensures, that all FUs of same type will begin their operation in the $n*L$ shift of the same phase of the clock, which is necessary for sharing FUs among operations. An operation of type k can be scheduled into the clock phase t only, if the C_t has empty place.

5 Results

We applied the described algorithms to schedule the 16-point FIR filter benchmark. We assumed a datapath width of 16 bit, in which the latency of an adder and a multiplier is 6 and 9 clock phase respectively. We generated different schedules by varying DII from 1 to 3 clock cycle (4 to 12 clock phase). The results of the ILP scheduling are in Table 1. Table 2. shows the results of the heuristic scheduling algorithm. The columns of the table are the data initiation interval, the number of adders and multipliers, the multiplexer inputs, the number of buffers, and the obtained schedule length. Both schedules use a minimal number of FUs, but the heuristic schedule is longer and requires more buffers. This is, because it schedules all the same types of operations to begin at a shift of a same clock phase. The number of buffers decreases significantly if we do not apply a long series of buffers, but a ring of four gates to build registers.

DII	M_+	M_*	Mux in.	Buffers	L
4	15	8	-	126	57
8	8	4	34	156	70
12	5	3	30	180	72

Table 1: ILP scheduling result

DII	M_+	M_*	Mux in.	Buffers	L
4	15	8	-	126	57
8	8	4	34	192	75
12	5	3	30	208	75

Table 2: Heuristic scheduling result

5 Simulation

It is an important task during the design of application specific integrated circuits to checks whether the design fulfills its specification. The specification of the circuit by a hardware description language (HDL), and the simulation of the HDL code is a typical validation technique of current industry practice. The design is defined at the algorithmic level and later refined down to the register transfer level (RT-level). Among the different HDLs for digital circuit design, VHDL is the most widely used and standardized. VHDL can capture the design at several abstraction levels and conveniently represent both the behavioral specification and the RT-level design.

The determination of the cycle-by-cycle behavior of the design and the timing refinement from the causal to clock-related level enable performance simulation. Full system simulations are required to validate the overall system concept. We need description technique to model the clock-phase controlled behaviour of the adiabatic logic and simulate together with the other part of a digital system. A clock related, but still behavioral model is needed to achieve acceptable simulation times.

We describe each different datapath component by a VHDL entity. The entity declaration specifies the name and the input/output port structure of the component. The architecture body is used to specify the functionality and timing of the component. In Listing 1. we show a fragment of the description of a 16 bit adiabatic adder in our library. For brevity we only show the signals for the two less significant bit of the adder.

The *pf1*, *pf2*, *pf3* and the *pf4* signals are the clock phase signals, which controls the flow of data through the cascaded gates. The logic functions of the gates are represented by the VHDL *block*s. The *block* is "*guarded*", wherein "*guarded*" concurrent signal assignments are present. The concurrent signal assignments statements describes the data dependency among the logic levels. The assignments are executed if the "*guard*" expression changes to a true value or if the "*guard*" expression is true and in the same time there is an event on the signal in the right side. This describes the behavior

of the adiabatic logic, where any input change during the active phase signal ruins the calculation. The "*guard*" expressions are controlled by the clock phase signals. The datapath is built up by component instantiation from the library. The clock phase signals are connected to each component in the appropriate order. The I/O ports of the design entities are connected by signals, which can also be used to capture the simulation data of the internal logic.

```
entity adder16 is
port (pa, pb: in std_logic_vector(0 to 15);
    pf1, pf2, pf3, pf4 : in std_logic;
    py: out std_logic_vector(0 to 16)); end
adder16;
architecture DFB of adder16 is
signal P1, G1, P2, G2, P3, G3, P4, G4, P5,
G5, P6, G6 : std_logic_vector(0 to 15);
begin
 dfb1:block (pf1='1') begin
   P1(0)<=guarded pa(0) xor pb(0);
   G1(0)<=guarded pa(0) and pb(0);
   P1(1)<=guarded pa(1) xor pb(1);
   G1(1)<=guarded pa(1) and pb(1);
 end block;
 dfb2:block (pf2='1') begin
   P2(0)<=guarded P1(0);
   G2(0)<=guarded G1(0);
   P2(1)<=guarded P1(1);
   G2(1)<=guarded  (G1(0)  and  P1(1))  xor
G1(1);
 end block;
 dfb3:block (pf3='1') begin
   P3(0)<=guarded P2(0);
   G3(0)<=guarded G2(0);
   P3(1)<=guarded P2(1);
   G3(1)<=guarded G2(1);
 end block;
 dfb4:block (pf4='1') begin
   P4(0)<=guarded P3(0);
   G4(0)<=guarded G3(0);
   P4(1)<=guarded P3(1);
   G4(1)<=guarded G3(1);
 end block;
 dfb5:block (pf1='1') begin
   P5(0)<=guarded P4(0);
   G5(0)<=guarded G4(0);
   P5(1)<=guarded P4(1);
   G5(1)<=guarded G4(1);
 end block;
 dfb6:block (pf2='1') begin
   py(0)<=guarded P5(0);
   py(1)<=guarded G5(0) xor P5(1);
 end block; end DFB;
```

Listing 1: Code fragment of the VHDL model of a 16-bit adiabatic adder

6 Conclusions

This paper presented an integer linear programming formulation and a heuristic scheduling technique for high-level synthesis, which are capable of scheduling operations implemented with adiabatic logic. Both approaches produce a pipeline schedule using a minimal number of resources, but the heuristic approach results in longer schedule, which may not fit in the allowed schedule length in some cases. We have also presented a VHDL description technique to model the clock-phase controlled behaviour of the adiabatic logic and simulate together with the other part of a digital system, which is also described in VHDL.

References:

[1] S. Kim, M. C. Papaefthymion: "True Single-Phase Energy-Recovering Logic for Low-Power, High-Speed VLSI", ISLPED, 1998, pp. 167-172

[2] F. Kovács, L. Varga, G. Hosszú: "Circuit Optimization of Adiabatic Charge-Recovery CMOS PLAs", joint meeting of the 4th World Multiconference on Systemics, Cybernetics and Informatics, SCI2000 and the 6th International Conference on Information Systems Analysis and Synthesis, ISAS2000, Orlando, Florida, USA, July 23-26, 2000, Vol. IX. pp. 153-156.

[3] A . Kramer, J. S. Denker, S. C. Avery, A. G. Dickinson, T. R. Wik: "Adiabatic Computing with the 2N-2N2D logic family", IEEE Symp. On VLSI Circuits, 1994, pp. 25-26.

[4] Y. Moon, D. Jeong: "An Efficient Charge Recovery Logic Circuit", IEEE Journ. Solid-State Circuits, vol-31, No 4, April. 1996, pp. 514-522.

[5] J. Lim, D. G. Kim, S. I. Chae: "A 16-bit Carry-Lookahead Adder Using Reversible Energy Recovery Logic for Ultra-Low-Energy Systems" IEEE Journ. Solid-State Circuits, vol-34, No.6, June 1999, pp. 898-903.

[6] W. C. Athas, N. Tzartzanis, L. J. Svensson, L. Peterson: "A Low-Power Microprocessor Based on Resonant Energy", IEEE Journ. Solid-State Circuits, Nov. 1997, pp. 1693-1701.

[7] W. C. Athas, W-C Liu, L. J. Svensson: "Energy-Recovery CMOS for Highly Pipelined DSP Design"

[8] C. Y. Roger Chen, M. Z. Moricz: "Data Path Scheduling for Two-Level Pipelining", Design Automation Conf., 1991, pp. 603-606.

[9] H. S. Jun, S. Y. Hwang: "Design of a Pipelined DataPath Synthesis System for Digital Signal Processing", IEEE Trans. on VLSI Systems, Sep. 1994, pp. 292-303.

[10] L. Varga, F. Kovács, G. Hosszú: "An Efficient Adiabatic Charge-Recovery Logic" accepted at Southeastcon, 2001