Transformation and VHDL Code Generation from Coarse-grained Dataflow Graph

Moonwook Oh
Department of Computer Engineering, Seoul National University
Kwanak-ku Shinlim-dong San 56-1, Seoul, Korea
mwoh@iris.snu.ac.kr

January 21, 1999

Contents

1 Introduction **1**

2 Correctness of Operation **3**

3 Performance Improvement via Transformations **5**

3.1 Transformations for Feed-forward DFGs 5

3.2 Transformation for Recursive DFGs 6

3.2.1 Unfolding Transformation 6

3.2.2 Buffer Registering 8

4 VHDL Code Generation **11**

5 Examples **15**

5.1 Implementation of a Feed-forward DFG 15

5.2 Buffer Registering and Reduction of Area Cost 16

5.3 A DSP Application - The QAM Module 18

6 Conclusions **20**

List of Figures

1.1 Two scenarioes of circuit design automation 2

2.1 Implementation of DFG semantics 3

3.1 Unfolding of a recusrisve DFG 7
3.2 Buffer registering 9

4.1 A sample DFG with pipelining registers 11
4.2 The VHDL code for an enable generator 13
4.3 The VHDL code for a reset generator 14

5.1 Diverse VHDL generation schemes for an original DFG. 16
5.2 Area saving via buffer registering 17
5.3 The synthesis result of a QAM module 19

List of Tables

5.1 Characteristics of the synthesized hardware 18

Abstract

This paper discusses how we generate VHDL codes for DSP applications described in dataflow graphs. Because the generated VHDL code implements the details of the control structure we can easily transform it into a running circuit without any modification, using logic synthesis tools. To improve the quality of the synthesized circuit we apply some graph transformation techniques to an original dataflow graph. We mainly consider coarse-grained dataflow graphs in which each node corresponds to an IP component of considerable size. The proposed facility is very useful for dataflow graph based high level design tools including our codesign framework PeaCE (Ptolemy extension as Codesign Environment).

Keywords : VHDL Code Generation, Dataflow Graph Transformation, EDA

Chapter 1

Introduction

Recently HDLs have gained considerable popularity because they provide hardware designers with software-like development process and help them complete complex chip design projects successfully. While HDL plays a very important role in system design, there is also increasing need for easier and more intuitive method for system description. Moreover, emerging opportunities for hardware-software codesign requires a specification method of higher level abstraction than HDL which are biased to hardware model.

Among many candidates of higher level specification methods, dataflow graph(DFG) has some attractive merits such as intuitiveness and readability. Many high level system design tools adopt DFG as system specification method instead of HDL [1] [2] [3]. Our codesign environment PeaCE (Ptolemy extension as Codesign Environment) which is being developed in Seoul National University, also uses a DFG along with VHDL [3] .

As the number of the DFG based EDA tools increases, HDL generation from a DFG becomes more important. Currently we have many novel tools which can synthesize a digital circuit from a HDL description. The design path of these tools is illustrated in Fig. 1.1(a). Therefore, if we can generate a synthesizable HDL code from a DFG description, we can build a seamless design path beginning with a DFG description and ending with a running circuit. This process is shown in Fig. 1.1(b). In this process it is very important to generate a HDL code of good quallity because the quallity of the final circuit is highly dependant on the input HDL code.

There exist some tools which support HDL generation for synthesis from

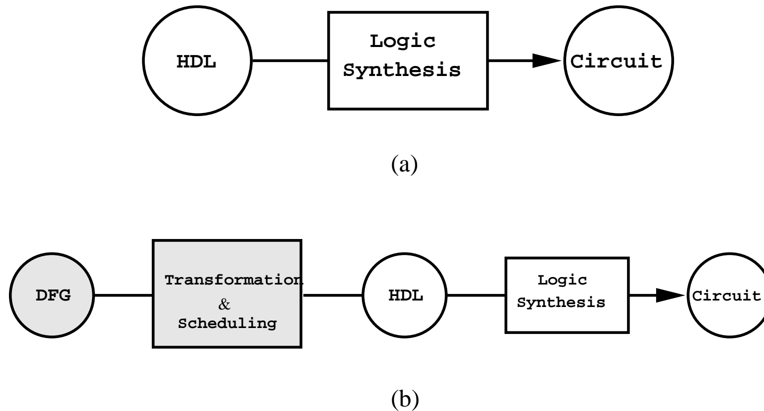


Figure 1.1: (a) The design path of current EDA tools. (b) The design path of DFG based future EDA tools.

DFG [1] [4] [5]. But most of them are not concerned with DFG transformation(or algorithm transformation) issue which can contribute to the performance of hardware significantly [1] [5] , and in some cases they even fail to produce a correct behaviour of synthesized hardware [4] . While some systems are concerned with fine-grained DFGs [6], we are mainly interested in synthesizable VHDL code generation from coarse-grained DFGs. In a coarse-grained DFG, each node is defined with pseudo-VHDL code describing its functionality. To guarantee the correct behavior of the synthesized hardware, we automatically build control structures in the generated VHDL code. In addition, we provide parallelizing facility and pipelining scheduler to improve the system throughput. The generated VHDL code is synthesized into a real hardware module by logic synthesis tools without any modification.

The following chapters contain discussions about correctness and performance of synthesized circuit, which are two important issues in VHDL generation for synthesis. Afterwards, we explain some details of VHDL coding style with automatically generated VHDL codes. Next, we provide design examples and the synthesis results obtained with PeaCE. Finally we make conclusion by clarifying what is done and what is not yet.

Chapter 2

Correctness of Operation

The DFG model in PeaCE is based on the synchronous data flow(SDF) graph [7] which has been widely used among DSP designers. In an SDF graph, a node can be fired only after all of its input samples are prepared, and every state register has an initial token. While we generate a VHDL code we should implement these semantics carefully in the code so that the resulting hardware has the same behaviour intended by the original DFG. This is the concept of correctness. By correctness we mean that a VHDL description and the synthesized hardware should have the same input/output characteristics as that of the original DFG.

In PeaCE we implement semantics of DFG with automatically generated control logics such as multiplexor, register and reset, enable generators.

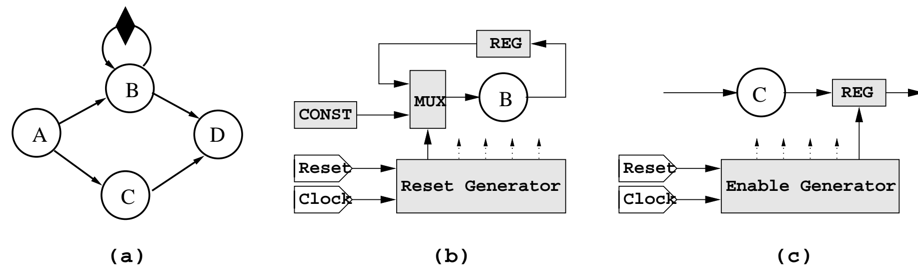


Figure 2.1: (a) A sample DFG. (b) A multiplexer for a feed-back loop. (c) An output register for firing control.

Any feedback in DFG needs a state register (or a delay register) and during the first iteration it should produce designated initial token. In Fig. 2.1(a) node B shows this case. As illustrated in Fig. reffigII(b), this requirement can be

realized by a register and a multiplexor, one of whose inputs is a constant source and controlled by a reset signal [4] . All the reset signals feeding multiplexors are controlled by a central reset generator, which is triggered by the system-wide reset signal.

To control the firing timing of each node, we add registers to proper output ports [4] and synthesize an enable generator, which asserts an enable signal to an output register when the output sample is prepared and makes the register hold the value until the successor completes its execution. By doing so, every node can start its new execution at proper time with correct inputs. Fig. reffigII(c) depicts the structure of the firing control logics. As well as output registers, all state registers are also controlled by the central enable generator.

A reset generator and an enable generator are centralized rather than distributed in each node. Any signal controller synthesized by PeaCE has a counter-based architecture and the operation of a node is related to the iteration period of a system; therefore if we allocate a signal controller for each node, we may have many duplicate counter structures. In this situation, by centralizing the controllers we can save the silicon area. [5]

Chapter 3

Performance Improvement via Transformations

Performance is another important issue of VHDL generation. Performance of a circuit can be characterized by latency, iteration period, power consumption, and so on. Among many existing approaches to improve the performance of a circuit, PeaCE mainly relies on DFG transformation and scheduling techniques, which are complementary to other EDA tools. DFG transformation contains a large scope of techniques such as pipelining, parallelizing [4] , unfolding [8] [9] , retiming [10] , and look-ahead transformation [11] .

3.1 Transformations for Feed-forward DFGs

PeaCE first transforms a given DFG into a homogeneous graph, where each port of a node consumes or produces only one sample per iteration [2] [4] . By this transformation, we can parallelize a multi-rate system into a single-rate system. Compared to such tools that do not support this facility [1] [5] we can provide designers with shorter iteration period of the synthesized hardware by parallelizing multi-rate nodes.

As well as parallelizing, currently PeaCE supports automatical pipelining to implement an optimal iteration period. We implement functional pipelining with output registers of each node and behavioral description of an enable controller. When PeaCE performs pipelining, it consults the required iteration period of the system and execution time of each node, given in the unit of one

system clock tick (each node is assumed as combinational logic). No additional registers are needed because existing output registers [4] can function as pipeline registers. The central enable generator feeds enable signals so that hardware may operate at optimum iteration rate.

3.2 Transformation for Recursive DFGs

If a DFG has any feed-back loop with delay(s), we call it a recursive DFG. Although paralleizing and pipelining are very useful for feed-forward DFGs but they are not applicable to recursive DFGs [10]. So we developed an unfolding transformation, which maximizes the throughput of recursive algorithms [12].

3.2.1 Unfolding Transformation

If a DFG has any feed-back loop with delay(s), we call it a recursive DFG. In a recursive DFG, the current output depends on the result of previous iterations. By this fundamental property, any recursive DFG has an inherent lower bound on the achievable iteration period referred to as the *iteration bound* [13]. This iteration bound T_∞ is given by

$$T_\infty = max[\frac{T_l}{D_l}]$$

where the maximum is taken over all loops in the DFG, and T_l is the sum of the computation times of all nodes in loop l, and D_l is the number of delay elements in loop l [8]. We call a DFG schedule *rate optimal* if the iteration period equals the iteration bound. The DFG in Fig. 3.1(a) has an iteration period of 60 time units and it is much larger than the iteration bound of 35 time units. A retimed DFG is depicted in Fig. 3.1(b) and it still does not have a rate optimal schedule.

Unfolding is suggested to produce a rate optimal scheduling of a recursive DFG [8] . If we unfold any DFG by *optimum unfolding factor*, $\alpha_{prg} = lcm[D_l]$[1] which is the least common multiple of the number of loop delays in the original DFG, we can always obtain a rate optimal schedule. After unfolding was introduced, much research was studied to find algorithms that produce unfolding

[1] *prg* means *perfect rate graph* which is introduced in [8]. Perfect rate graph has only 1 delay element for each loop.

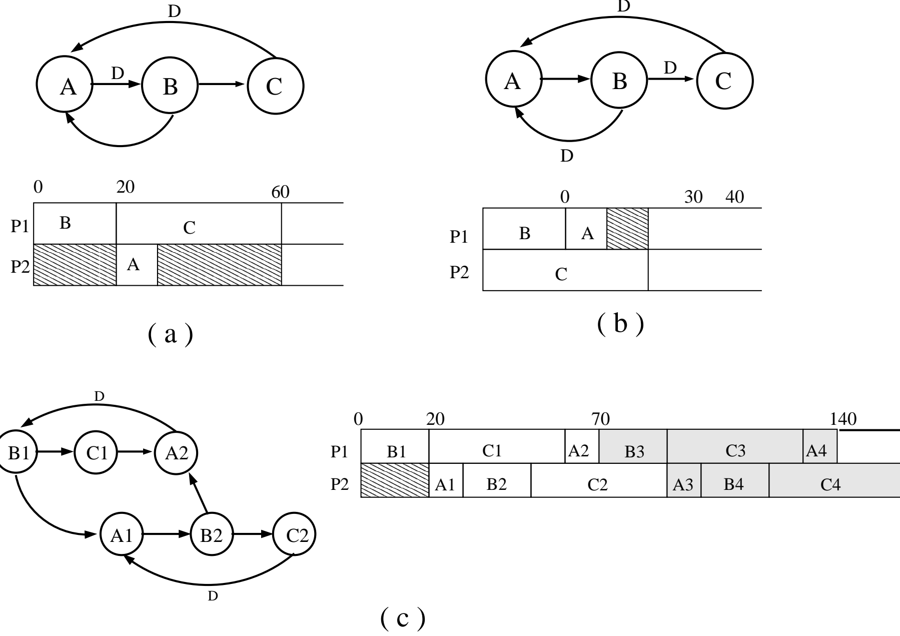


Figure 3.1: **(a)** Node execution time of A, B, and C are 10, 20, and 40 respectively. The Iteration bound is 35 time unit and the critical path time (or the iteration period) is 60 time unit. **(b)** A retimed DFG of (a). The critical path time is reduced to 40 time unit. **(c)** 2-unfolded DFG. The iteration period is 70 time unit which is 35×2. It has an overlapped rate optimal schedule.

factors smaller than α_{prg} [14][9][15][16]. It is important to have a smaller unfolding factor because a large unfolding factor gives a large number of nodes in the unfolded DFG and as a consequence it increases the complexity of scheduling polynomially. One of the smallest unfolding factors was proposed by [9] as α_{gprg}[2] and it is given by

$$\alpha_{gprg} = \lceil \frac{max[n_i]}{T_{ip}} \rceil$$

where n_i is the execution time of node i and T_{ip} is the required iteration period. Fig. 3.1(c) shows $\alpha_{prg} = \alpha_{gprg} = 2$ unfolded DFG and its schedule. Now the iteration period is 35 and satisfies the rate optimal schedule.

[2] *gprg* means *generalized perfect rate graph*. A DFG is gprg iff its iteration period is greater than or equal to maximum node computation time.

3.2.2 Buffer Registering

Unfolding has been studied mainly by multiprocessor groups. And most unfolding algorithms are likely to give overlapped schedules like Fig. 3.1(c). While an overlapped schedule can be easily implemented in multiprocessor systems , this is not the case in VLSI design.

Let us assume that we want to design a VLSI which executes the schedule in Fig. 3.1(c). We assume that all nodes are combinational logic and that only delay elements correspond to registers. Then we can easily notice that B3 can not start its execution at time unit 70 because the value of B1 should be maintained until the execution of C2 completes (at time unit 90). This constraint is added from the characteristics of VLSI but not from the semantics of DFG, which allows B3 to start at time unit 70 after the executions of B1 and A2 are completed.

Remark 1 *In VLSI design, all nodes in a delay-free path are combinational logic and they must not change their output values until the last node finishes its operat ion.*

Now we know that unfolding transformation and the following overlapped schedules are not directly applicable in VLSI design. To solve this problem we suggest a technique which is called 'buffer registering'.

If we insert buffer registers to the delay-free arcs they preserve the value of the source node and set all the ancestor nodes free from the constraint in Remarks 1. In general, insertion of registers to a recursive DFG might corrupt the original functionality. This is the reason why we can not pipeline a recursive DFG. But we can insert buffer registers to DFG without corrupting the original semantics by carefully scheduling their activations.

Remark 2 *If a DFG has an iteration period of T_{ip} and node A produces its valid output at time offset of k ($k \geq 0$) each iteration interval, a buffering register in $A \rightarrow B$ may have a periodical activation schedule, $T_{ip} \cdot i + k (i = 0, 1, 2, \dots)$ without corrupting the original functionality.*

Fig. 3.2 depicts the modified DFG of Fig. 3.1(c) after inserting a buffer register and the activation schedule of the buffer register. We can verify that an overlapped rate optimal schedule is feasible now in VLSI implementations.

Note also that delay registers on arc (A2,B1) and on arc (C2,A1) are clocked the first time at 70 time units and 90 time units respectively, although clocking periods (70 time units) are same.

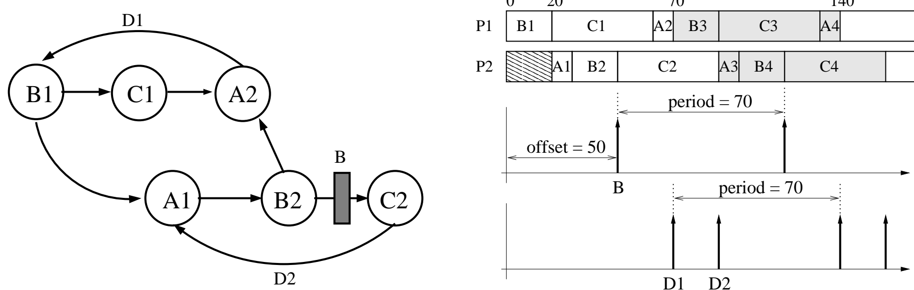


Figure 3.2: The DFG has a buffer register on arc (B2,C2). Because the buffer register preserves the computation result of ancestor nodes, node B1 can start the next iteration before node C2 finishes. The control signal diagram below the schedule shows activation timing of buffer register **B**.

Below is suggested algorithm which decides where to locate buffering registers and their activation schedules. The input is a DFG which has an overlapped rate optimal schedule and the outputs are a new DFG with buffer registers and the schedules of them.

Algorithm 1 BufferRegistering
(DFG_{org} : input, DFG_{new}, schedule : output)

Step 1: *Let schedule be an empty set.*
Let T_∞ = iteration bound of DFG_{org}.
Step 2: *If there is any path of execution time $> T_\infty$*
then let the path $P_{critical}$.
else return.
end if
Step 3: *Let $T_{elapsed} = t = 0$.*
Step 4: *While traversing $P_{critical}$ from start to end,*
$T_{elapsed} = T_{elapsed} + N_{curr}$
(N_{curr} is the execution time of the current node)
$t = t + N_{curr}$
If $t > T_\infty$ then
place a buffer register before the current node.
put $T_\infty \cdot i + (T_{elapsed} - N_{curr})$ to schedule.
$t = 0$.
end if
Step 5: *goto Step 2.*

Thanks to the Algorithm 1, we can apply unfolding to an arbitrary recursive

DFGs and implement them in VLSI of maximum throughput. The usefullness of this transformation will be clarified by an example in chapter 5.

No one transformation (and scheduling) technique alone can satisfy diverse requirements of digital system designs [11]. So it is desirable that a high level design tool like PeaCE provides designers with multiple choices and let them choose and apply adequate transformation techniques according to their own tastes. In PeaCE, we separate the modules for graph transformation and scheduling from the other parts of the kernel so that any newly developed technique can be easily integrated to the existing framework. Thanks to this approach, most of scheduling techniques can be implemented by simply substituting the VHDL processes of reset and enable generators with new codes.

Chapter 4

VHDL Code Generation

In this chapter we show how to generate a VHDL code from a DFG. A sample DFG is illustrated in Fig. 4.1. The number within a parenthesis is the execution time of the node; Node A and C take 2 clock cycles for execution, and Node B takes 3 clock cycles. And two feedback delay elements (D1, D2) are given by user, while pipelining registers (P1,P2, and P3) are inserted by PeaCE automatically.

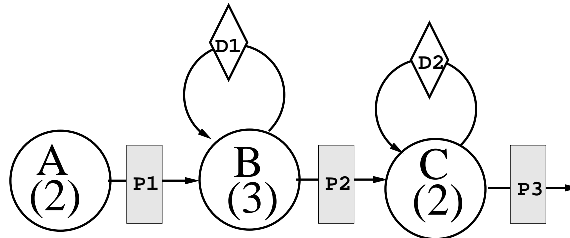


Figure 4.1: A sample DFG with pipelining registers

The maximum throughput of this pipelined DFG can be implemented as the iteration period of 3 clock cycles. The VHDL code in Fig. 4.2 is for an enable generator which feeds all registers in circuit, so that they have activation period of 3 clock cycles. The main body of the VHDL code consists of `Reset_loop` and `Main_loop`. `Reset_loop` synchronizes the operation of an enable generator to the system wide reset and `Main_loop` implements periodic activation of registers.

Along with pipeling registers, this DFG should also have two multiplexors

to implement initial values of D1 and D2, we should provide a reset generator to control that multiplexors. During the first iteration, Multiplexors should select the initial values instead of the latched values (refer to Fig. 2.1(b)). And because the iteration period is 3 clock cycles, Multiplexors for D1 and D2 must provide the initial values until 6th and 9th cycle respectively.

The VHDL code for a reset generator which controls the multiplexors is provided in Fig. 4.3. The process body consists of 3 loops – `Reset_loop`, `Main_loop`, and `Hold_loop`. Compared to the code of an enable generator in Fig. 4.2, `Hold_loop` is newly inserted. This is because the operation of a reset generator is not periodic, while that of an enable generator is periodic. A multiplexor for a delay element should pass the initial value during the first iteration, but for the following iterations, it should pass the signal of a delay register. `Hold_loop` is an infinite loop which implements this requirment.

```
Entity EnableGenerator is
  port( CLK : IN boolean, -- system clock
        RST : IN boolean, -- system reset
        P1,P2,P3 : OUT boolean,
        D1,D2 : OUT boolean );
end EnableGenerator;

Architecture Bev of EnableGenerator is
begin
Process begin
-- For synchronization with system reset
Reset_loop : loop
  P1 <= FALSE; P2 <= FALSE; P3 <= FALSE;
  D1 <= FALSE; D2 <= FALSE;

-- Main loop of the iteration period 3
Main_loop : loop
--State 1
  wait until CLK'event and CLK=TRUE;
  if RST=TRUE then exit Reset_loop; end if;
     P1 <= FALSE; P2 <= FALSE; P3 <= FALSE;
     D1 <= FALSE; D2 <= FALSE;
--State 2
  wait until CLK'event and CLK=TRUE;
  if RST=TRUE then exit Reset_loop; end if;
--State 3
  wait until CLK'event and CLK=TRUE;
  if RST=TRUE then exit Reset_loop; end if;
     P1 <= TRUE; P2 <= TRUE; P3 <= TRUE;
     D1 <= TRUE; D2 <= TRUE;
end loop; -- Main_loop
end loop; -- Reset_loop

end Process;
end Bev; -- Architecture
```

Figure 4.2: The VHDL code for an enable generator

```
Entity ResetGenerator is
  port( CLK : IN boolean, -- system clock
        RST : IN boolean, -- system reset
        D1_InitVal, D2_InitVal : OUT boolean );
end ResetGenerator;

Architecture Bev of ResetGenerator is
begin
Process begin
Reset_loop : loop
  D1_InitVal <= TRUE; D2_InitVal <= TRUE;

Main_loop : loop
--State 1
  wait until CLK'event and CLK=TRUE;
  if RST=TRUE then exit Reset_loop; end if;
  ...
--State 6
  wait until CLK'event and CLK=TRUE;
  if RST=TRUE then exit Reset_loop; end if;
  D1_InitVal <= FALSE;
  ...
--State 9
  wait until CLK'event and CLK=TRUE;
  if RST=TRUE then exit Reset_loop; end if;
  D2_InitVal <= FALSE;

-- To hold D1, D2 in FALSE state.
Hold_loop : loop
  wait until CLK'event and CLK=TRUE;
  if RST=TRUE then exit Reset_loop; end if;
end loop; -- Hold_loop
end loop; -- Main_loop
end loop; -- Reset_loop

end Process;
end Bev; -- Architecture
```

Figure 4.3: The VHDL code for a reset generator

Chapter 5

Examples

In this chapter we explain our system with three examples. The transformation scheme and the synthesis results are presented in the following two examples respectively.

5.1 Implementation of a Feed-forward DFG

As we have mentioned already, DFG transformation can improve the overall performance of the synthesized circuit significantly. This is true not only for fine-grained DFGs but also for coarse-grained DFGs. The Fig. 5.1(a) shows a DFG for a DSP application containing multi-rate relationship between node B and C.

The simplest way of synthesis is to generate a HDL netlist (Fig. 5.1(b)) which contains the behavioral description of the original DFG and to run some behavioral synthesis tools using the generated HDL as an input. In this approach, no serious consideration is given to the original DFG and synthesis effort – DFG transformation, scheduling, binding, and so on – is concentrated on the fine-grained DFG generated from the HDL.

But it is observed that we may have better synthesis results when we try to improve the system in higher abstraction level. Thus it is more preferable to consider DFG transformation and scheduling before HDL generation stage rather than after the stage. Some high level design systems like [1] try to find a novel scheduling of the original DFG and incorporates the result while it generates a HDL netlist. The Fig. 5.1(c) depicts the structure of the generated

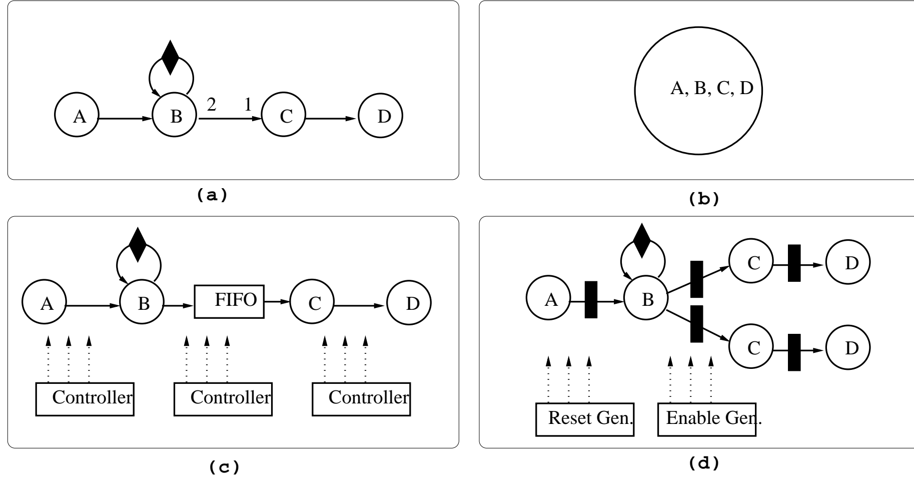


Figure 5.1: Diverse VHDL generation schemes for an original DFG.

HDL which contains some control logics implementing the scheduling result. But the system of [1] does not support DFG transformation facility for the original DFG.

On the other hand, our system transforms the original DFG appropriately and schedules it before it generates a HDL code. So the HDL code contains the transformed structure and control logics for a schedule. DFG transformation in higher abstraction level is very important because it leads to a better HDL code and again the HDL code results in a circuit of better quality. The Fig. 5.1(d) shows the result of parallelization and pipelining followed by the automatic synthesis of control structure.

5.2 Buffer Registering and Reduction of Area Cost

Let us assume we want to implement a time critical algorithm, depicted in Fig. 5.2(a) [8], in a VLSI so that it has the execution speed of 16 time units per iteration. This execution speed can be obtained by no other means except rate optimal implementation. (Please notice that the DFG in Fig. 5.2(a) has the iteration bound of $(N_A + N_D + N_E)/2 = 32/2 = 16$ time units.) Formerly, when we had no support of buffer register insertion, an overlapped multiprocessor schedule was not implementable in VLSI and we had to find a nonoverlapped

rate optimal schedule. For this example, there exists an unfolding factor which gives an nonoverlapped schedule. If we run the algorithm in [17], we can see it is $\alpha_{cpt} = 6$. The 6-unfolded DFG is illustrated in Fig. 5.2(b).

Although we obtained a rate optimal VLSI by 6-unfolding, it requires much area cost. If we insert a buffer register according to Algorithm 1, the existing algorith ms for overlapped multiprocessor schedules can be applied to VLSI design. When we run one of them [9], a rate optimal schedule is obtained by only $\alpha_{gprg} = 2$ unfolding. It means that the area cost is reduced by almost 1/3, compared to nonoverlapped implementation. Fig. 5.2(c) shows the 2-unfolded DFG which is much simpler than that of Fig. 5.2(b).

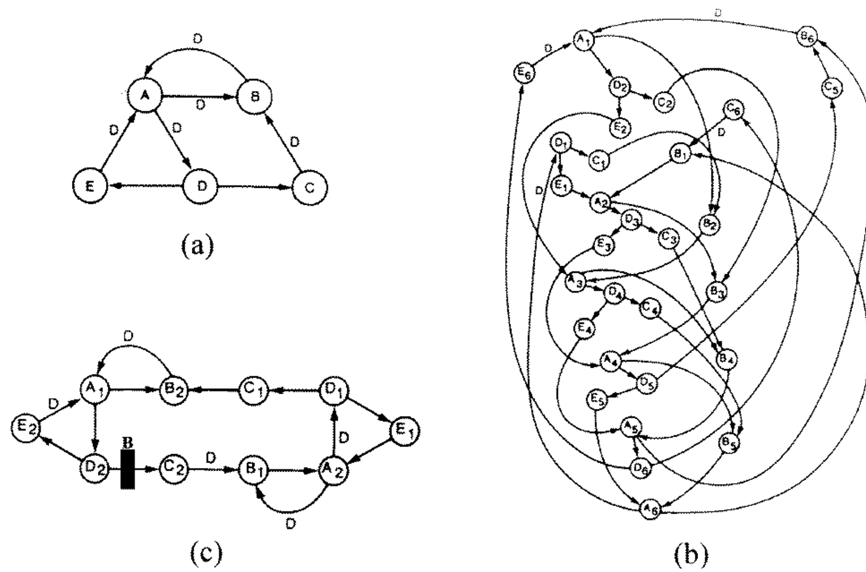


Figure 5.2: **(a)** Original DFG. Node A, B, C, D, E require 20, 5, 10, 10, 2 time unit re spectively. The iteration bound is 16, the maximum node execution time is 20 and the critical path time is 20 time units. **(b)** A 6-unfolded DFG which allows nonoverlapped rate optimal schedule. A c ritical path is (D1,E1,A2,D3,E3,A4,D5,E5,A6) and requires 96 time units, which corresponds to 6×16. **(c)** A 2-unfolded DFG which has an overlapped rate optimal schedule. It is much simpler than the DFG in (b).

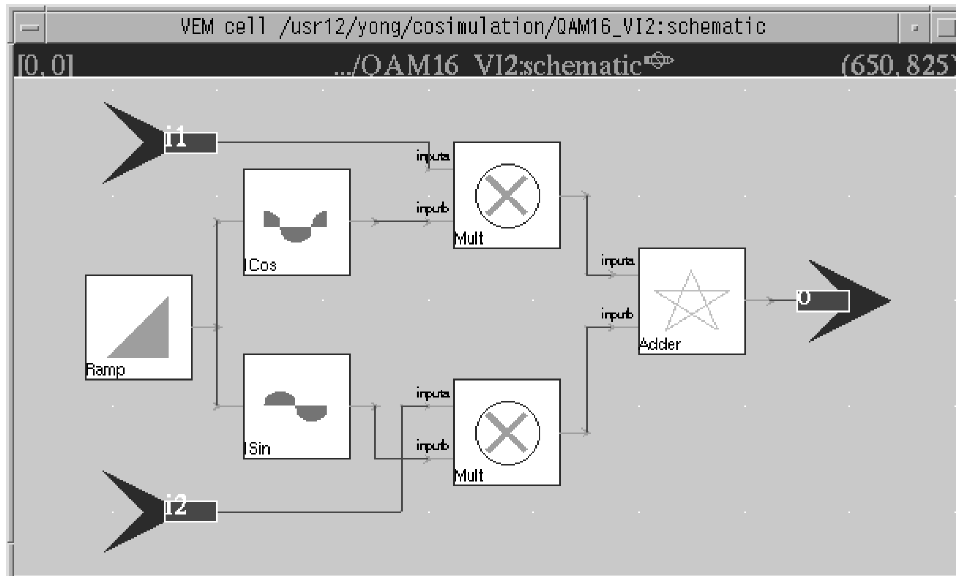
Silicon area (# of Xilinx CLB)	
Total system	131
Control logic	39
Overhead of control logic	29.8%
Iteration period (# of clock counts. freq. = 50MHz)	
Before pipelining	11
After pipelining	5
Speed-up	220%

Table 5.1: Characteristics of the synthesized hardware

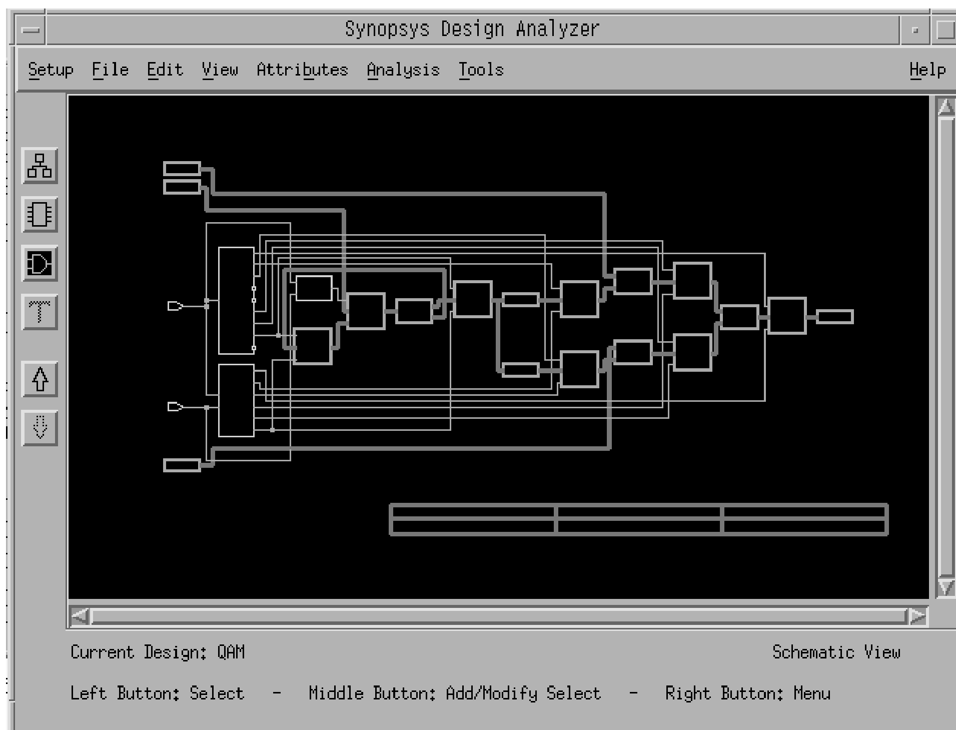
5.3 A DSP Application - The QAM Module

To make the argument more clear we take a part of the QAM system as an example application [3]. The Fig.5.3(a) shows the DFG of the final stage in QAM. This sub-system consists of a ramp counter, cosine and sin generators, multipliers and adders.

From this DFG, we generated a VHDL code and synthesized it using Synopsys' Design Compiler. The hardware contains 1 multiplexor for initialization of the ramp counter, because it requires a state register. And 7 output registers are also added in the proper places. The multiplexor and registers are connected to the automatically generated reset and enable generators, which have pipeline schedule codes to control the system. When all data lines are implemented in 8 bit width, the control logic part takes about 30% of total system area. And pipelining transformation improves the throughput of the hardware by 220% when system clock has 50MHz frequency. These results are illustrated in Fig. 5.3(b) and Table 5.1.



(a)



(b)

Figure 5.3: (a) A DFG representation of a mudule of QAM (b) A block diagram of the synthesized hardware

Chapter 6

Conclusions

We implemented the VHDL code generation feature for a data flow graph in codesign framework PeaCE(Ptolemy extension as Codesign Environment). The VHDL code is synthesizable without any modification and contains automatically created control logics so that the correctness of operation is guaranteed. And PeaCE performs pipelining transformation so that it may provides an optimal iteration period for a given DFG. As well as pipelining, PeaCE also provides parallelizing facility for multi-rate DFGs. But unfolding and buffer registering are not supported yet. We believe that this research will finally bridge the gap between the DFG based future design systems and the current EDA tools based on HDL description.

Aside from existing facilities, there are some topics which are to be considered and solved. Currently, PeaCE cannot generate VHDL codes for the DFGs of dynamic behavior which allows asynchronous arrival of input samples, varying sample rate, and change of execution time of a node. And many of graph transformation features are not implemented yet. Because some graph transformations require duplication of the same node, resource sharing among these nodes is another issue to be addressed. These problems are currently under research or remain for future works.

Bibliography

[1] Synopsys Inc., 700 E. Middlefield Rd., Mountain View, CA 94043, USA. *COSSAP User's Manual : VHDL Code Generation*.

[2] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy : A framework for simulating and prototyping heterogeneous systems. *Journal of Computer Simulation, special issue on Simulation Software Development*, 4:155–182, April 1994.

[3] W. Sung, M. Oh, C. Im, and S. Ha. Demonstration of codesign workflow in peace. In *Proc. of International Conference of VLSI Circuit*, Seoul, Korea, 1997.

[4] M. C. Williamson and E. A. Lee. Synthesis of parallel hardware implementations from synchronous dataflow graph specifications. In *30th Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, California, USA, November 1996.

[5] T. Groker P. Zepter and H. Meyr. Digital receiver design using vhdl generation from data flow graphs. In *Proc. of the Design Automation Conference*, 1995.

[6] C. Y. Wang and K. K. Parhi. The mars high-level dsp synthesis system. In *VLSI Design Methodologies for Digital Signal Processing Architectures*. Kluwer Academic Press, 1994.

[7] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. In *Proc. of the IEEE*, September 1987.

[8] K. K. Parhi. Static rate-optimal scheduling of iterative data-flow program via optimum unfoliding. *IEEE Trans. on Computers*, 40(2), February 1991.

[9] D. J. Wang and Y. H. Hu. Fully static multiprocessor array realizability criteria for real-time recurrent dsp applications. *IEEE Trans. on Signal Processing*, 42(5), May 1994.

[10] C. E. Leiserson, F. Rose, and J. Saxe. Optimizing synchronous circuitry by retiming. In *Proc. Third Caltech Conf. VLSI*, pages pp. 87–116, Pasadena, CA., March 1983.

[11] K. K. Parhi. High-level algorithm and architecture transformations for dsp synthesis. *Journal of VLSI Signal Processing*, 9:pp 121–143, 1995.

[12] Moonwook Oh and Soonhoi Ha. Rate optimal vlsi design from dataflow graph. In *Proc. 35th Design Automation Conf.*, San Francisco, CA., June 1998.

[13] R. Reiter. Scheduling parallel computations. *Journal of ACM*, 15(4):pp. 590–599, October 1968.

[14] L. G. Jeng and L. G. Chen. Rate-optimal dsp synthesis by pipeline and minimum unfolding. *IEEE Trans. on VLSI Systems*, 2(1), March 1994.

[15] L. G. Jeng and L. G. Chen. A globally static rate optimal scheduling from recursive dsp algorithms. In *Proc. ICASSP*, pages pp. 1005–1008, May 1991.

[16] L. F. Chao and E. H. M. Sha. Unfolding and retiming data-flow dsp programs for risc multiprocessor scheduling. In *Proc. ICASSP*, volume 5, pages pp. 565–568, 1992.

[17] L. E. Lucke and K. K. Parhi. Data-flow transformations for critical path time reduction in high-level dsp synthesis. *IEEE Trans. on CAD of IC and Systems*, 12(7), July 1993.