æ

Synthesizable VHDL Code Generation from Data Flow Graph*

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

Abstract

This paper discusses how we generate a VHDL code for a DSP application described in a data flow graph(DFG). Because the generated VHDL code contains only synthesizable constructs and implements details of control logics we can easily transform it into a running hardware module using logic synthesis tools. This facility is very useful for DFG based high level system design tools including our codesign framework PeaCE (Ptolemy extension as Codesign Environment).

1 Introduction

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

Among many candidates of higher level specification methods, data flow graph(DFG) has some attractive merits such as intuitiveness and readability. Many high level system design approaches 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] .

While we design a digital system based on a DFG representation, we need to generate HDL codes for simulation or synthesis. During simulation stage, because designers are concerned with operation in higher abstraction level, HDL codes can be free from many details of a real system. But the situation is quite different in stage of synthesis. Such as register initialization and appropriate clocking should be implemented for correct operation. Thus, we should consider all the details of hardware signals such as reset, enable, and clock.

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

The following sections contain discussions about correctness and performance of the generated hardware, which are two important issues in VHDL generation for synthesis. Next, we provide a design example and the result obtained with PeaCE. Finally we make conclusion by clarifying what is done and what is not yet.

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 a SDF graph, any 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 may have the 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 signals.

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.1(a) node B shows this case. As illustrated in Fig.1(b), this requirement can be realized by a register and a multiplexor, one of whose inputs is a constant

*This research was supported in parts by Ministry of Education through ISRC, under contract number 96-E-2103

To appear in Proceedings of the 5th Asia Pacific Conference on Hardware Description Languages, July 1998, Seoul, Korea.

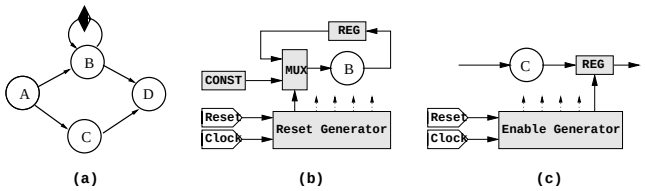


Figure 1: **(a)** A sample DFG. **(b)** Implementation style for an feed-back loop. **(c)** An output register for firing control.

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 externel system wide reset.

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.1(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, most of controllers may have the same counter structure in them. In this situation by centralizing the controllers we can save the silicon area. [5]

3 Performance of The Synthesized Circuit

Performance is another important issue in the 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. DFG transformation contains a large scope of techniques such as pipelining, parallelizing [4] , unfolding [8] [9] , retiming [10] , and look-ahead transformation [11] .

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 implements 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 nodes are assumed as a 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.

Although pipelining is very useful for feed-forward DFGs but it is not applicable to such DFGs that contain feedback loops. [10] So we are now developing unfolding transformation which can enhance the performance of the DFGs with feedback. 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 many 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.

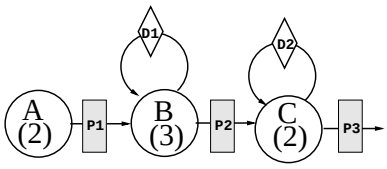


Figure 2: A sample DFG with pipelining registers

4 VHDL Code Generation

In this section we show how to generate a VHDL code from a DFG. A sample DFG is illustrated in Fig.2. The number in a parenthesis means the execution time of each 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.

The maximum throughput of this pipelined DFG can be implemented as the iteration period of 3 clock cycles. The VHDL code in Fig.3 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.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. The process body consists of 3 loops – `Reset_loop`, `Main_loop`, and `Hold_loop`. Compared to the code of an enable generator in Fig.3, `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.

5 An Experiment

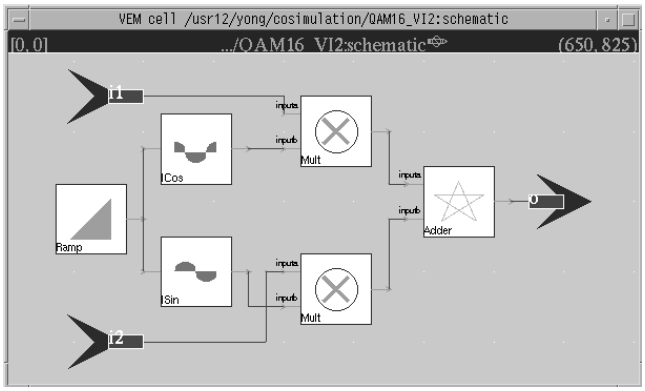
To make the argument on correctness and performance more clear we take a part of QAM system as an example application. [3] The Fig.5(a) shows the DFG of the final stage in

```
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
```

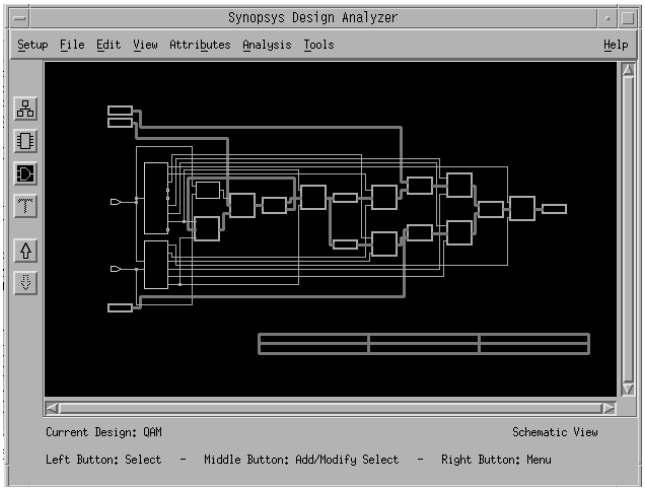
Figure 3: The VHDL code for an enable generator

```
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: The VHDL code for a reset generator



(a)



(b)

Figure 5: **(a)** A DFG representation of a mudule of QAM **(b)** A block diagram of the synthesized hardware

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 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(b) and Table 1.

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 pe-

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 1: Characteristics of the synthesized hardware

riod for a given DFG. As well as pipelining, PeaCE also provides parallelizing facility for multi-rate DFGs.

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 remains for future works.

7 Acknowledgement

Moonwook Oh, an author of this paper, was supported by Graduate Student Scholarship Program of Korea Fund for Advanced Studies from 1997 to 1998.

References

[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*, Kyungju, 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.