Clocking Schedule and Writing VHDL Programs for Synthesis

Chia-Jeng Tseng

Departmentf of Electrical Engineering
Bucknell University
Lewisburg, Pennsylvania 17837

Abstract

Most of the commercial tools for digital synthesis are designed for Register-Transfer-Level (RTL) and logic synthesis. Numerous wonderful books and papers are available for understanding the syntax and semantics of the VHDL language. There are also many books and papers written for VHDL synthesis; these articles generally focus on the syntax for describing logic blocks such as combinational circuits, flip-flops, and simple finite state machines as well as how a synthesizer may infer logic from a VHDL description. Most students, even after they have learned the language features, still encounter tremendous difficulty when they begin to use the VHDL to describe a digital system for synthesis. In this paper, we describe the essence of modeling digital functions and present a powerful concept, called clocking schedule, for writing a VHDL program for RTL and logic synthesis. This technique facilitates seamless integration of all the modules in a digital design. A motion-guide project is used to demonstrate the applications and effectiveness of the technique to RTL and logic synthesis using the VHDL. Based on our experience from an "Advanced Digital Design" course, the methodology is very instructive. The students appreciated the power of digital synthesis with the VHDL in a very short period of time.

1. Introduction

Several Electronic Design Automation (EDA) companies [12, 15, 22] offer synthesis tools supporting the VHDL and Verilog languages [2, 3, 5, 7]. Most of these programs are designed for RTL and logic synthesis; as a matter of fact, behavioral or high-level synthesis is still in an experimental phase for the design communities. The main difference between high-level synthesis and RTL/logic synthesis is on the conceptual modeling of digital systems.

High-level synthesis assumes a micro-architectural view for digital design. Temporal properties are specified in a description. Clocking signals are, however, not explicitly contained in a behavioral description for high-level synthesis. A description for RTL/logic synthesis, on the other hand, does not contain temporal sequence. Clocking is explicitly specified in a VHDL

description; multiple clocks may be used in a digital design. A VHDL program describes all possible events within each clock cycle.

In this paper, we will discuss how to write a VHDL program for RTL and logic synthesis. In particular, we focus on using clocking schedule to guide the writing of a VHDL program. Section 2 gives a brief overview for the VHDL. Section 3 describes the modeling essence of RTL and logic synthesis. Based on the modeling concept for traditional logic design, useful guidelines for describing digital designs using the VHDL for RTL/logic synthesis are presented. These guidelines facilitate seamless integration of all the modules contained in a digital circuit. Section 4 presents a motion-guide design project to illustrate the concepts explored in this paper. An alarm clock and a discrete cosine transform project are also briefly described. Finally, Section 5 draws a conclusion for this paper.

2. VHDL Overview

VHDL stands for VHSIC (Very High Speed Integrated Circuits) Hardware Description Language. Initiated by the Department of Defense, the development of the VHDL began in 1983. Over these years, the language had gone through numerous revisions. A VHDL program may describe a complex digital circuit; it may also specify a simple logic component. A basic VHDL description consists of an entity declaration and an architecture specification of a digital circuit. The entity declaration identifies all the input and output ports of the digital circuit while the architecture specification defines the function or structure of the circuit. The specification of a circuit can be expressed in a structural style, a dataflow style, a behavioral style, or a combination of all three styles.

A VHDL description may be used for formal design documentation, for logic simulation, and for synthesis. The commonly used VHDL constructs include assignment statements, conditional control constructs such as if-then-else and case-when, arithmetic and logic expressions as well as hardware specific control constructs such as expressions for specifying the rising-edge triggered flip-flops. Many other VHDL constructs are available for one to describe a digital circuit. Most of them are very handy for specifying digital components. The reader may consult with a VHDL book or paper for their applications [2, 3, 6, 7]. The language also allows a user to define sub-circuits in terms of processes, components, blocks, functions, and procedures.

For the purpose of illustration, a VHDL description of an odd parity generator for seven-bit data with the output bit being saved into a rising edge-triggered D flip-flop in each clock cycle is shown in Figure 1.

In the VHDL description, the string of "--" represents a comment indicator. All the letters following a "--" indicator are for comments; in Figure 1, we place an integer after each comment operator as a line identifier. The library declaration in line 1 makes the `ieee` pre-defined library available to the synthesizer. The `use` clause in line 2 shows that the types and objects defined in the std_logic_1164 package can be accessed. In the entity declaration between lines 3 and 7, the following identifiers define the input and output ports of the circuit.

- `clk` for clock source
- `indata` for seven-bit data input

- `parity` as an output port

A VHDL process named `odd_parity` between lines 10 and 17 is defined to compute the parity bit as an exclusive-or function of the seven-bit input data. The `if-then` statement between lines 18 and 20 defines that an event occurs at the rising edge of the clock signal. The clock control and the signal assignment operator "<=" would be used by a VHDL synthesizer to infer a rising-edge-triggered D flip-flop for the signal of "`parity`." The `odd_parity` process and the signal assignment statement for parity define concurrent events.

In the VHDL description shown in Figure 1, `cmbparity` is defined by a combinational exclusive-or function of `indata` (line 14 to 16). Each time `indata` changes its value, the value of `cmbparity` will change after a delay defined by the exclusive-or function. The value of parity, defined by `cmbparity`, is updated at the rising-edge of the `clk` signal.

```
library ieee;                                                    --  1
use std_logic_1164.all;                                          --  2

entity parity_generator of                                       --  3
port(indata: in std_logic_vector(6 downto 0),                    --  4
     clk: in std_logic;                                          --  5
     parity: out std_logic);                                     --  6
end parity_generator;                                            --  7

architecture parity_function of parity_generator is              --  8
begin                                                            --  9
  odd_parity:                                                    -- 10
  process(indata)                                                -- 11
    variable cmbparity: std_logic;                               -- 12
  begin                                                          -- 13
    cmbparity := indata(6) xor indata(5) xor indata(4)           -- 14
                           xor indata(3) xor indata(2)           -- 15
                           xor indata(1) xor indata(0);          -- 16
  end process;                                                   -- 17

  if (clk'event and clk='1') then                                -- 18
    parity <= cmbparity;                                         -- 19
  end if;                                                        -- 20
end parity_function;                                             -- 21
```

Figure 1: A VHDL description of a parity generator for seven-bit data

3. Clocking Schedule and VHDL Synthesis

A digital design may contain a number of random logic combinational circuits, structured data-path components such as adders and multipliers, flip-flops, counters and finite state machines. The tasks of traditional digital design involve the integration of these modules using schematic capture, net-list descriptions, or other design entries. Using a VHDL or Verilog description as an input to a synthesis tool for digital design is a state-of-the-art design entry. The major issues

include the specification and integration of logic modules using the VHDL or Verilog language. The methodology described in this paper is applicable to both languages; for convenience, the VHDL is used to illustrate the concepts.

Writing a VHDL program is very different from writing a conventional software program such as a C program [8]. A conventional program is procedural; the function of a task is described in a step-by-step manner; the statements are formed as an ordered-list. In general, the resultant binary codes are executed in a micro-architectural computer. On the other hand, in a VHDL program for RTL/logic synthesis, the statements for specifying different hardware components may be randomly placed.

Among the logic components defined in a VHDL model for RTL/logic synthesis, an event would occur whenever an input signal to a combinational component changes its value. For all other types of logic components an event takes effect according to the clocking schedule. The coordination among circuit components are determined by the schedule of reading events from their outputs and writing events to their inputs. In other words, clocking and event schedule is a useful reference for defining system interfaces, coordination protocols, and partitioning boundaries. As a result, the specification of clocking schedule is one of the most important tasks for the design of a digital circuit. Once the clocking schedule is determined, the coding of a VHDL program for the circuit becomes a mechanical process.

In this section, we discuss the main tasks of the methodology. We will also offer useful guidelines to ensure seamless integration of logic modules using VHDL descriptions.

3.1 Understanding Design Requirements

The most important step of designing a digital system is to correctly understand and interpret the design specification. In order to produce a design meeting the specification, the requirements must be unambiguously interpreted. The VHDL language allows a designer to formally define a design. Simulation tools can be used to verify the correctness of a VHDL specification. Through the application of a set of test vectors to a completed design, test tools will be able to enhance the confidence level of the final implementation. Based on the assumption of correct-by-construction offered by a synthesis tool, VHDL synthesis significantly simplifies the complexity of digital design.

3.2 System Decomposition

Divide-and-conquer is a powerful technique for decomposing a digital system into a number of sub-circuits. Many criteria may be used to guide system partitioning, including functionality and connectivity. Clocking and event schedule may also be used as an efficient reference for system partitioning. Further details of these decomposition criteria are given below.

- Functionality
  In a digital system, tightly coupled events are often defined as a process. Several processes, which are generally loosely coupled, are then integrated into a system. For example, in processor design, a digital system is often decomposed into a data part and a

control part. At the system level, modern computer systems may contain special-purpose processing units such as input and output processor, memory management processor, and floating-point processor. From design point of view, a digital system can often be functionally decomposed into several modules.

- Connectivity
  Interface signals constitute a useful reference for system partitioning; for example, the min-cut algorithm for circuit partitioning was based on connectivity structure [9].
- Clocking schedule
  Synchronization signals associated with the same clock can be used to identify closely related circuit components. If multiple clocks are used, each clock signal is generally associated with those components which are closely related. As a result, the clocking signals serves as a great reference for selecting related circuit components.

The criteria of functionality, connectivity, and clocking schedule often offer complementary references for circuit partitioning.

3.3 User Interface

Most digital systems interact with external world through input and output interface. The design of interface logic is therefore a crucial part of digital design. A digital circuit generated by a contemporary synthesis tool is generally a synchronous system. An input device may introduce asynchronous behavior to the system, which may result in unpredictable impacts on the circuit functionality. A pushbutton for generating a pulse input is an example of this type of devices. A finite state machine driven by a clock signal may eliminate the bouncing concerns.

Decimal number system is most accessible to most people. Computers, on the other hand, use a binary number system to perform computation internally. Binary-to-decimal conversion is often required for output display.

3.4 Subsystem Coordination

The techniques frequently used for interfacing subsystems include direct coupling, handshaking, interrupt, and polling. In a single-clock system, an event may not occur in every cycle. In a multiple-clock system, handshaking signals may be needed to synchronize an event. Clocking schedule, along with necessary triggering and acknowledgement event schedule, is an efficient reference for defining detailed coordination for these interface schemes. For example, a sequential multiplier may be initiated by a strobe signal and return a done signal after a multiplication is completed. If it is needed, a third acknowledgement signal can be introduced by the calling module to close a handshaking loop. The circuit which invokes the multiplier may be driven by a clock signal with a higher, the same, or a lower clock speed. The clocking schedule can be used to define the needed protocols for seamless coordination.

3.5 Clock Signals Generation

Clocks are important signals in a digital design. Given a clock source, counters may be used to generate clock signals of various frequencies. The duty cycle of a clock signal is also an

important parameter for generating a reliable and efficient digital circuit. If the events in a digital circuit are scheduled for both the high and low duration, each phase should be long enough to accommodate the critical combinational delay. Once a clock signal is defined, all the events associated with it can be described in terms of the clocking schedule.

3.6 Bundling Register Definition Statements

Each register should be driven by a unique clock signal. It is a good practice to gather the statements of defining a register under different conditions in a centralized context enclosed by the clocking statement. This practice may eliminate a lot of coding and design errors.

3.7 Concise Finite-State-Machine Description

There are numerous ways of implementing a finite state machine using VHDL [6]. Two examples are given below.

- Explicitly describe the function of a finite state machine as a state transition automaton.
- Use a case-statement to define the state transition and output functions.

State reduction has been a topic for extensive research. To implement a finite state machine with the minimum number of states, however, may not be the wisest choice. The use of a few additional states may sometimes significantly simplify the VHDL specification and combinational logic.

4. Illustrative Design Projects

In the fall semester of 2003, an "Advanced Digital Design" course was offered to the senior and graduate classes at Bucknell University. This course consisted of two components, including lectures and laboratories. The lectures were comprised of three modules. First, logic synthesis was taught. The issues covered included Quine-McClusky method for two-level logic minimization [11, 13, 14], multiple-level logic optimization [4], technology mapping for FPGA and standard-cell implementation [10] as well as finite-state-machine synthesis. The second module covered the VHDL language [2, 3, 7], including the syntax and semantics of the language as well as the modeling techniques described in this paper. The third module addressed the issues of high-level synthesis, which included clique-partitioning for data-path synthesis [16], global slicing technique for control synthesis [17], scheduling methods, and behavioral transformations for design space exploration [18].

The laboratory components consisted of the study of Field-Programmable Gate Arrays architectures [21], Xess FPGA boards [19], and Xilinx FPGA design flow [20]. Three FPGA commercial synthesis tools were made available for the students to use [12, 15, 22]. Comparative studies were performed on these tools.

Three projects, which included a motion guide, an alarm clock, and a discrete cosine transform [1], were assigned to the students to practice digital design methodologies. The students were requested to write VHDL descriptions for each of these three projects, to generate an FPGA implementation for each project, and to demonstrate the features of each design. The motion

guide was a simple state machine while the discrete cosine transform represented a special-purpose processor design. The complexity of these designs grew one by one. In the following subsections, we will describe the design issues of these three projects. The motion-guide project will be detailed to illustrate the methodologies described in this paper.

4.1 A Motion Guide

As depicted in Figure 2, a motion guide consists of three arrows filled with Light-Emitting Diodes (LEDs). These three arrows are used for motion indicators to direct its user the direction of body movement and to regulate the duration of each move. The device can assist its user to do physical exercise. The assistance frees the user from memorizing the direction and duration of body movement so that he/she can concentrate on the exercise. Further details of the project are given below:

1. When the circuit is activated it should only light the "Up" arrow. This state is considered as the starting state.
2. Only one of the three arrows is lit at any time.
3. The LEDs in arrows should light up in the following sequence Up, Right, Up, Left, Up, Right, Up, Left, and so on for a user specified number of cycles. A cycle is Up, Right, Up and Left. The allowable number of cycles is between 1 and 15.
4. The on time for the LEDs in each arrow is user specified. It can be set to an integer value between 10 and 41 seconds. The on time of the LEDs is the same for all three arrows.
5. Two 7-segment LED digits are available for displaying a user-specified data, including the number of cycles or the number of seconds for illuminating an LED. Your design should take care of any error conditions. Appropriate decisions should be made for any missing information as well.

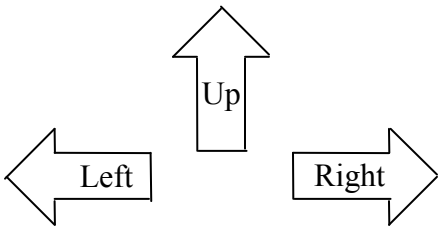


Figure 2: A Motion Guide

This circuit is essentially a finite state machine with some additional logic for controlling the cycle duration and the number of cycles. Functionally, the system was partitioned into three modules, including a cycle-duration control, a cycle-number control, and a motion-guide module. The following subsections describe the design considerations for each of these circuit blocks.

4.1.1 System Configuration

As depicted in Figure 3, from implementation point of view, the system was partitioned into the following blocks: a clocks generator, a cycle-duration capture circuit, a number-of-cycles capture circuit, a motion-guide finite state machine as well as input and output interface circuits. Each of these modules performs a specific function and can be described as VHDL processes. These processes interact with one another through appropriate coupling mechanisms.

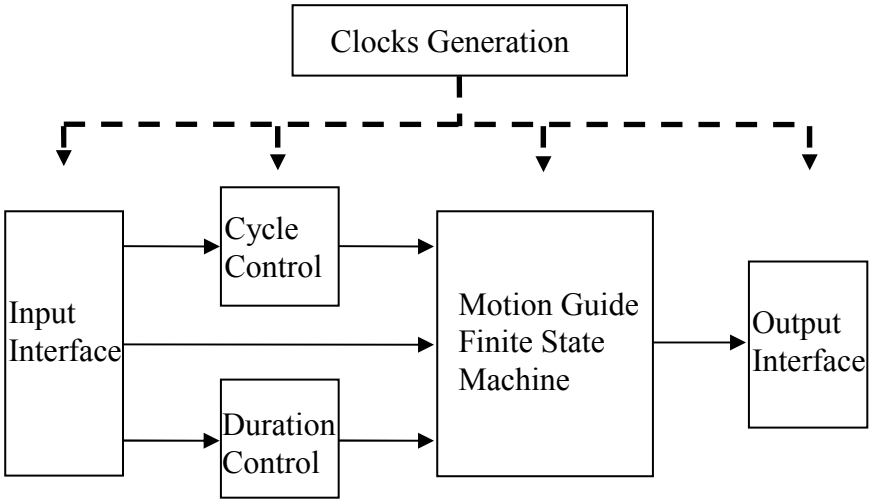


Figure 3: Motion Guide System Configuration

4.1.2 User Interface

Two types of switches were available for user's input interface. One type is called dipswitches, which are toggle switches. The other type belongs to pushbutton switches; they are for generating pulses.

We used dipswitches to define system states. As depicted in Figure 4, the following four system states were defined.

- Idle state:
  This is the default state. When the system is in this state, the upward arrow is lit.
- Motion guide operating state:
  When the system is in this state, motion guiding is activated by a pushbutton pulse. While the system is being operated in a motion guidance mode, pushbutton events are ignored. The system can be asynchronously reset by switching to the idle state.
- Cycle duration specification state:
  This system state allows a user to set the duration of a motion guiding state.
- Number of cycles specification state:

This system state allows a user to specify the number of cycles for the motion-guide to operate.

A three-state finite-state-machine depicted in Figure 4 was used to capture an event generated by a pushbutton switch. VHDL descriptions for the state machines depicted in Figure 4 are shown in Figure 5. In Figure 4, the three pushbutton states are embedded in each system state. For example, let the system be in the state of DS3. When a pulse is generated, the pushbutton state machine moves from PS0, through PS1, to PS2. The state machine does not return to PS0 till the guiding events are completed. Except for the request of a system reset, the setting of any other system states would not be acknowledged till the processing of a pushbutton command is completed. The close coupling of the system states and the pushbutton state machine simplifies input command handling. No matter an input pulse is short or long, this scheme is able to gracefully handle the input request. The side effects due to an asynchronous behavior of input interface are effectively eliminated.

In addition to the two state machines depicted in Figure 4, a binary-to-decimal conversion circuit was also developed to support a friendly user interface.

4.1.3 Generation of Clock Signals

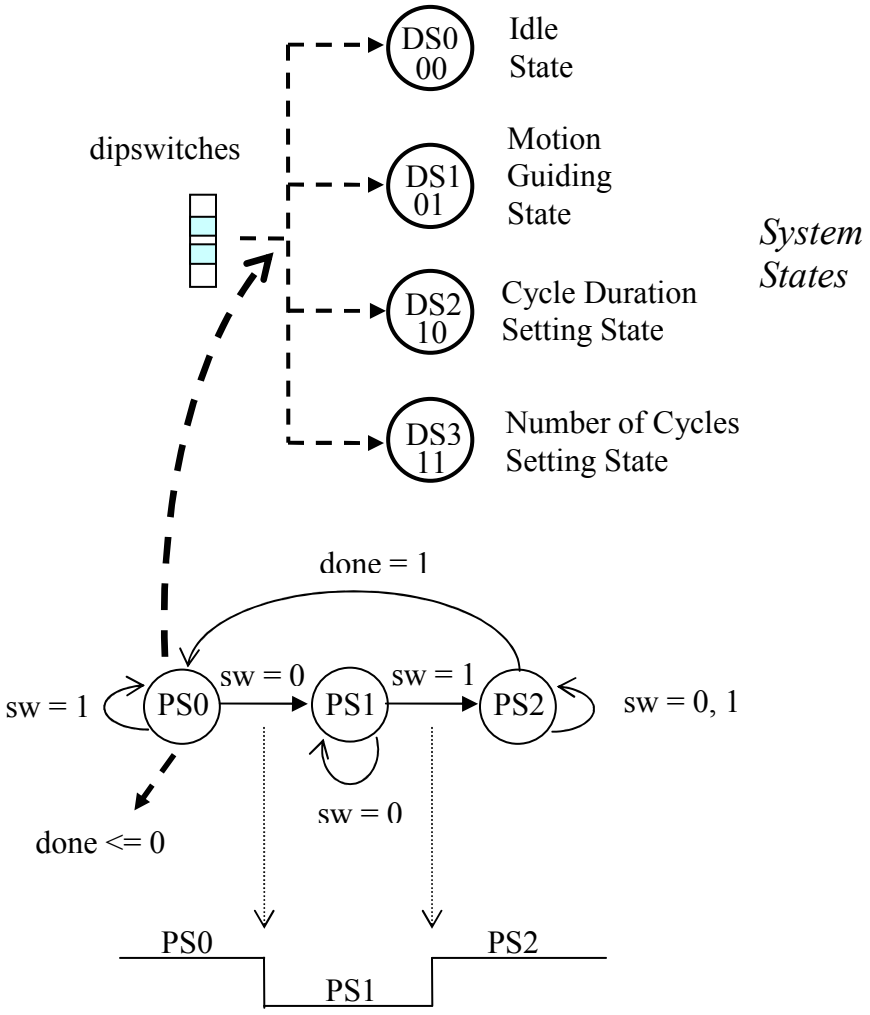
Two clock signals were used in the design; an 8-hertz clock was used for latching a pushbutton signal and a one-hertz clock was used to drive the motion-guide machine. A 100-MHz clock source was available in an XSA-100 board; this clock source was used to derive a 1-MHz clock signal for the FPGA chip. Inside the FPGA, a counter was used to extract an eight-hertz and a one-hertz clocks from the 1-MHz source. The VHDL description for this component consists of a signal for the counter embedded in the context of a rising-edge if-then statement. The counter is reset when it reaches one mega. The eight-hertz and one-hertz clock signals were extracted from two counter outputs. The VHDL description for clocks generation is depicted in Figure 6.

4.1.4 Finite-State-Machine Specification

If a flag is used to identify the direction of state transitions, as shown in Figure 7, three states would be sufficient for the motion-guide finite state machine. To simplify the logic for defining state transition, a four-state machine depicted in Figure 8 was defined. In other words, the upward system state is represented by two different machine states. The finite state machine was turned into a counter with necessary cycle duration and cycle count control logic.

4.1.5 Setting of Cycle Duration and Number of Cycles

Two registers are used to store the cycle duration and the number of cycles, respectively. The entries of input data may be through dipswitches or pushbutton switches. In either case, the data are displayed on LEDs. The first approach uses dipswitches to define input data and a pushbutton pulse can be used to latch the data. The second approach uses a pushbutton pulse to increment the value of a register. A sequence of pulses will define the value to be set.



Pushbutton is normally at 1. A depress of the button corresponds to a zero. "Done" is set by the calling module when the circuit of a system state completes the processing of a pushbutton command; it is reset by the calling module when the pushbutton machine returns to PS0.

Figure 4: Motion-Guide System States and Pushbutton States

```
pushsw_monitor:
process(clk8hz)
begin
  if (clk8hz'event and clk8hz='1') then
      case push_state is
        when "00" => if (pushsw = '0') then
                       push_state <= "01";
                     else
                       push_state <= "00";
                     end if;
        when "01" => if (pushsw = '1') then
                       push_state <= "10";
                     else
                       push_state <= "01";
                     end if;
        when "10" => if (push_done = '1') then
                       push_state <= "00";
                     else
                       push_state <= "10";
                     end if;
        when others => push_state <= "00";
      end case;
  end if;
end process;

dipsw4_monitor:
process(clk8hz)
begin
  if (clk8hz'event and clk8hz='1') then
    if (push_state = "00") then
      dipsw4_state <= dipsw4;
    end if;
  end if;
end process;
```

Figure 5: VHDL Description for System and Pushbutton State Machines

```
one_hertz_counter:
process(reset, clksrc)
begin
  if (clksrc'event and clksrc='1') then
    if (reset = '1') then
      cntr1hz <= (others => '0');
    elsif (cntr1hz = "11110100001001000000") then
      cntr1hz <= (others => '0');
    else
      cntr1hz <= cntr1hz + 1;
    end if;
  end if;
end process;

clocks_generator:
process(cntr1hz)
begin
  clk1hz <= cntr1hz(19);
  clk8hz <= cntr1hz(16);
end process;
```

Figure 6: Clocks Generator VHDL Description

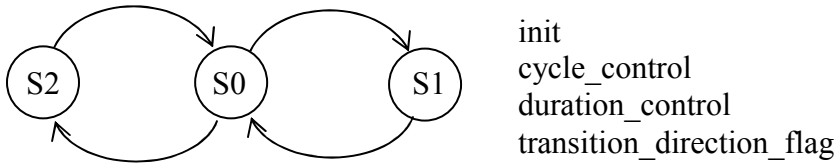


Figure 7: Three-State Motion Guide State Machine

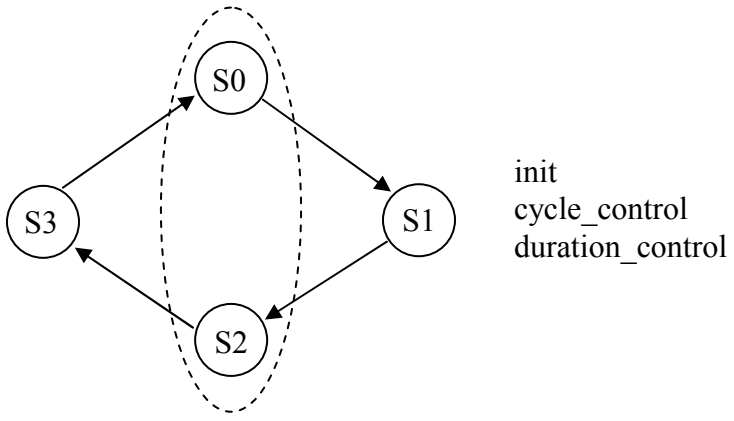


Figure 8: Four-State Motion Guide State Machine

4.2 An Alarm Clock

Though the detailed design of the alarm clock is much more complex than the motion guide, the basic ideas are very similar. However, there is a subtle difference between the motion guide and the alarm clock from design point of view. All of the circuit modules associated with a system state are active only when the motion guide enters the particular system state; clocking signals generator is the only module which is always running. In the alarm clock design, in addition to the clocking signals generator, the wall clock circuit must be running concurrently with most circuit modules defined for the system states. The wall clock halts only if it is in the system state of setting a new wall clock time.

One of the Xess circuit boards contains a Coder/Decoder (CODEC) module [19]. To make the design an open-ended project, the students were encouraged to integrate the CODEC capabilities for the recording and playing of customized wake-up signals.

4.3 A Discrete Cosine Transform Algorithm

In addition to the application of clocking and event schedule for VHDL synthesis, design methodologies based on micro-architectural model were also taught in the course. These methodologies were applied to the design of the discrete cosine transform algorithm. Based on a preliminary analysis, with a schematic capture tool, the design of the discrete cosine transform would probably take a good designer at least a month to complete. Using VHDL synthesis tools, several students were able to complete and demonstrate their designs within two weeks.

5. Conclusion

Traditionally, a digital designer uses a schematic capture tool or a net list specification to create a design. Contemporary designs use a hardware description language to define a design. As the complexity of digital designs continues to grow, it is a horrendous burden for a designer to cope with all the details of a digital design. Software synthesis tools not only improve design productivity but also ensure quality results.

The methodology described in this paper was taught in the "Advanced Digital Design" classes at Bucknell University. The students applied the techniques to the projects described in Sections 3 and 4. Based on the feedback from the students, the techniques were extremely effective for learning how to write a VHDL program for RTL and logic synthesis and were very instrumental for the completion of the class projects.

Acknowledgements

Professor Maurice F. Aburdene provided us the specifications of the motion guide and the discrete cosine transform algorithm. The constant support and encouragement given by him is greatly appreciated. I am also very grateful for the numerous constructive suggestions given by the students of the ELEC 444/644 class throughout the fall semester of 2003.

References

1. Maurice F. Aburdene, Jianqing Zheng, and Richard J. Kozick, Computation of Discrete Cosine Transform Using Clenshaw's Recurrence Formula, IEEE Signal Processing Letters, Bol. 2, No. 8, August 1995.

2. Peter J. Ashenden, The Designer's Guide to VHDL, 2002, Morgan Kaufmann Publishers, San Francisco, California 94104.

3. J. Bhasker, A VHDL Synthesis Primer, 1996, Star Galaxy Publishing, Allentown, Pennsylvania 18103.

4. R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli and A. R. Wang, MIS: A Multiple-Level Interactive Logic Optimization System, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Pages 1062-1081, CAD-6, 6, November 1987.

5. Michael D. Ciletti, Advanced Digital Design with the Verilog HDL, 2003, Prentice Hall, New Jersey 07458.

6. Clifford E. Cummings, State Machine Coding Styles for Synthesis, Sunburst Design, Inc., SNUG 1998.

7. IEEE Standard VHDL Language Reference Manual (IEEE Std 1076-1987), The Institute of Electrical and Electronics Engineers, Inc., 1988.

8. Brian W. Kerninghan, and Ritchie, Dennis M., The C Programming Language, 1978, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632.

9. Brian W. Kernighan and S. Lin, An Efficient Heuristic Procedure for Partitioning Graphs, The Bell System Technical Journal, 49(1): 291-307, 1970.

10. Kurt Keutzer, DAGON: Technology Binding and Local Optimization by DAG Matching, 24th ACM/IEEE Design Automation Conference, 1987.

11. E. J. McClusky, Introduction to the Theory of Switching Circuits, McGraw-Hill, 1965.

12. Mentor Graphics, Leonardo Spectrum User's Guide, 2002.

13. W. V. Quine, The Problem of Simplifying Truth Functions, The American Mathematical Monthly 59, Pages 521-531, October 1952.

14. W. V. Quine, A Way to Simplify Truth Functions, The American Mathematical Monthly 62, Pages 627-631, November 1955.

15. Synplicity, Synplicity Pro User's Guide, 2002.

16. Chia-Jeng Tseng and Daniel P. Siewiorek, Facet: A Procedure for the Automated Synthesis of Digital Systems, Proceedings of the 20th Design Automation Conference, 1983.

17. Chia-Jeng Tseng, Ruey S. Wei, Steve G. Rothweiler, Michael Tong, Ajoy Bose, Bridge: A Versatile Behavioral Synthesis System, 1988 ACM/IEEE 25th Design Automation Conference, Anaheim, California, June 1988.

18. Chia-Jeng Tseng, Behavioral Transformation for Pipeline Synthesis, Proceedings of the Custom Integrated Circuit Conference, June 1991.

19. Xess Corporation, XSA Board V1.1, V1.2 User Manual, Apex, North Carolina 27502, 2002.

20. Xess Corporation, Introduction to WebPACK 5.1 for FPGAs, Apex, North Carolina 27502, 2002.

21. Xilinx, Xilinx Spartan II FPGA Handbook, California, 2002.

22. Xilinx, Xilinx Synthesis Technology (XST) User Guide, California, 2002.

Biography

CHIA-JENG TSENG is an Assistant Professor in the Department of Electrical Engineering at Bucknell University. His current research interests focus on the study of digital design methodologies as well as innovative computational algorithms and architectures.