

Methodology of System Design using VHDL

Sea H. Choi and Moon J. Chung
Department of Computer Science
Michigan State University
East Lansing, MI 48824-1027

Abstract

This paper describes research work in system redesign using VHDL. This research is centered on the redesign of the *Remote I/O Module* in the *SINCGARS Radio* [1]. Using VHDL, the behavior of the controller, which currently uses micro-processor, is modeled as a Finite State Machine, FSM. By implementing the FSM as an ASIC chip, the microprocessor could be eliminated. A modeling technique of FSM, in which each state is represented by a single **process** is presented. The scheme contrasts with other approaches in which whole FSM is represented by a single **process** and each state is represented by either a **case** statement or an **if** statement. Our approach has some advantages such as the modularity of design, state decomposition and availability to easily describe a complicated state. It also has the benefits of separate compilation and individual simulation. When a circuit is implemented by a structural description, it needs to be checked a timing correctness such as hazard and race. Hazard and race condition detection in VHDL, which is one of the critical parts of timing verification, is also presented. Using ternary logic in VHDL, the hazard and critical race condition can be detected. In real hardware gates, gate delays can be varied depending on the environment, such as temperature and fan-out. If a precise gate delay is assigned, a critical race condition can be detected with a high degree of confidence; a false detection may in reality be due to delay varying.

1 Introduction

Many engineering communities are gaining experience with VHDL not only as a design documentation medium but also as the simulation tool [2, 3]. As the design complexity increases dramatically, top-down approach is necessary. VHDL supports a top-down approach by allowing high level design abstractions as well as a bottom-up approach.

This paper describes the role of VHDL in the system redesign. The system redesign process can be divided into the following steps:

- Behavioral description of the system is developed,
- Structural description is obtained, and
- Timing correctness is checked.

In this paper, we present the system redesign of the *Remote I/O Module* in the *SINCGARS Radio* [1]. Using VHDL, the current system which is using micro-processor is retargeted to the behavioral description of a FSM. Figure 1 shows the block diagram of the *Remote I/O Module* in the *SINCGARS Radio*. FSM modeling with VHDL will be discussed as well as other issues such as interrupt handling and stack operation. These modeling techniques were successfully used to implement the SINCGARS circuit [2, 4]. The structural description can be obtained from the behavioral description by using a High-Level Synthesis system. At this point, the equivalence between the behavioral and the structural description must be checked [8] and the timing correctness of the structural description must be checked as well. One of timing problem is timing hazard conditions. In this paper, behavioral

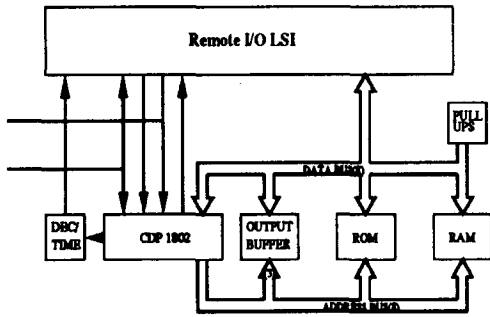


Figure 1. Block Diagram

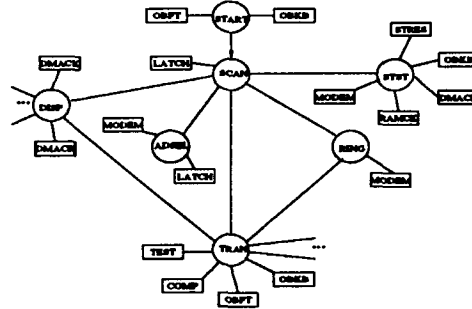


Figure 2. Remote I/O State Flow Diagram

description of a system using VHDL is presented and hazard detection methods using VHDL in combinational circuits and race detection methods in sequential circuits are also discussed.

2 Circuit Functionality

The current implementation of the *Remote I/O CLSI* consists of the *CDP 1802 microprocessor*, *ROM*, *RAM*, *OUTPUT BUFFER*, *DEC/TIM*, and a *REMOTE INTERFACE* shown in the Figure 1. The microprocessor is used in the *SINGARS Receiver-Transmitter(RT)* to control the status of the radio. It controls the inputting, decoding, outputting and encoding of remote control words sent from the other modules in the radio. Thus the radio can be controlled at the distance via a physical wire.

The *ROM* contains the stored program that controls the functioning of the *CDP 1802*. The *RAM* is used for temporary storage during the operation of the *CDP 1802* system. The *OUTPUT BUFFER* can be loaded with parallel data which can be shifted out in a serial data stream later. The output clocks and gates are also provided. The *DECODER/TIMER* is designed to decode the *CDP 1802 N.Lines* into the *OUTs* and *INs* for use with the microprocessor and to provide timing for the system. During a microprocessor *SCAN* state shown in Figure 2, the latched signals are read by the microprocessor via the data bus. The signals are then analyzed and appropriate action is taken. It provides direct memory access to the *RAM*.

3 VHDL Modeling of Finite State Machine

The micro-code in the *CDP 1802 processor* is modeled using VHDL. First of all, the overall behavioral description of the processor is described with VHDL behavioral description. This whole description is further divided into a set of states in the FSM. In Figure 2, the circles represent the main states and the rectangles represent the subroutines. These states and subroutines are further divided into states to model the generation of output signals, the access of *RAM*, and the wait for the clock such as *EF4* which is 640Hz clock input. The *subroutines* in the micro-program are also implemented as states.

The states in the finite state machine are implemented using the **process** statement of VHDL. The FSM is synchronized to the rising edge of the *640KHz* clock. The interrupt routine is executed when the interrupt line goes high and the interrupt enable is high.

The major disadvantage of using the **process** statement is the extensive use of the **null** assignment. If the same signal is driven in more than one **process**, it must have a bus resolution function associated with this driver. In our design a signal variable, *transition*, which selects the proper next

state value must be driven from only one process, i.e. rest of the drivers should be disconnected at any given simulation time.

3.1 Basic Structure of State Modeling using VHDL

<pre> state_assign:process begin -- see Figure:State Assign end process; fsm:process(present_state,x) begin case present_state is when 0 => output <= '0'; if(x='0') then transition <= 0; else transition <= 1; end if; when 1 => ... end case; end process; </pre> <p>(A) Case Statement</p>	<pre> process -- state 0 begin wait on clk; if(present_state=0 and clk='0')then output <= '0'; if(x='0') then transition <= 0; else transition <= 1; end if; else output <= null; transition <= null; end if; end process; process -- state 1 ... </pre> <p>(B) Process Statement</p>
---	--

Figure 3. FSM Implementation

There are several ways of implementing a FSM in VHDL. Using the **case** statement is the most frequently used. An example of using the **case** statement is shown in Figure 3 (A). The other method, which we are suggesting in this paper, uses the **process** statement. This is shown in Figure 3 (B). These examples are for synchronous sequential machines.

Each state has a fixed structure and a template is shown in Figure 5. One **process** represents one state. Each **process** waits for the changing value of *clk* (**wait on** statement). The next **if** statement controls the execution of the statements in that particular state. If the conditions are satisfied, the sequential statements inside of **if** statement are executed. There can be any number of sequential statements in any **process**. Otherwise, the **null** assignment statements are executed. At any given point of the simulation, only one **process** among all the processes can actively update the new signal values. The rest of **processes** are simply disconnecting all the signal drivers because of the **if** statement. A **process** may or may not have local variables.

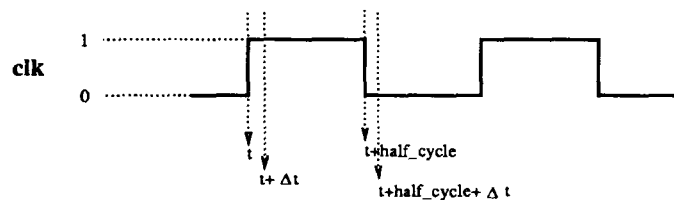


Figure 4. Timing Diagram

3.2 State Transition and Output Generation

New state transition occurs at the rising edge of the clock, say at time *t*, in Figure 4. This condition is checked at statement 1 of Figure 6. This value is available after a Δt time delay. Thus when the clock is '0', this new value is already available to all the processes in which the conditions can be checked. This is the case of the normal state assignment(statement 5) in Figure 6. However if the

interrupt request is raised (conditions in statement 2), the next state value is saved(statement 4), and the flow of control goes to the state "900" which is the beginning state of the interrupt handling routine to serve the interrupt routine(statement 3).

Any output can be generated at the rising edge of the clock at the same time when the state transition occurs. All the output signals can be assigned at the region of statement 6.

```

PROCESS
  -- local variables declarations
BEGIN
  WAIT ON clk;
  IF(present_state = ? AND clk = '0') THEN
    -- ? = the current state number
    -- Sequential statements
    output <= new_output; --output assign
    transition <= next_state; --state trans
    -- Sequential statements
  ELSE
    output <= null;
    transition <= null;
  END IF;
END PROCESS;

state_assign : PROCESS
BEGIN
  WAIT ON clk;
  IF(clk = '1' AND NOT clk'stable) THEN
    IF(interrupt_enable = '1' AND interrupt = '1') THEN -- 1
      present_state <= 900; -- interrupt routine -- 2
      state_save <= next_state; -- 3
    ELSE
      present_state <= next_state; -- 4
    END IF;
    -- Output assignments -- 5
  END IF;
END PROCESS;

```

Figure 5. Basic State Implementation

Figure 6. State Assign And Output Assign

3.3 RAM Operation

RAM is implemented as the one dimensional array of integers. The address range of RAM is from 0 to 127, which is X"00" to X"7F" respectively. Since the actual address of RAM is from X"0800" to X"087F" and in this VHDL model we do not use the space between X"0000" to X"07FF", we just eliminate the high address and use only the low portion of address space as the RAM space. At the rising edge of the clock, either data can be stored into the RAM or data can be read from the RAM depending on the value of the signal *memory_operation*.

If the signal *memory_operation* has the value *load*, the data is read from the RAM and saved to the signal *ramdata* and the signal *ramdata* has a new value after $t+\Delta t$. This is because the execution of any state starts when the clock is '0' at which the time has already passed $t+\Delta t$ and *ramdata* can be used as the D [5] in the state. Thus we can implement the RAM operation safely.

3.4 Subroutine Handling

Subroutines are implemented just like any state in the FSM. The execution of the subroutine is done by assigning the beginning state of the subroutine to *transition* and saving the return state onto the stack. There is an independent **process** which handles the stack operation shown in program 7. And **RETURN** can be implemented by assigning state "1000" to the variable *transition*. At the state "1000", it simply *pop* the return address and makes the next transition available. This **process** is shown in Program 7.

Since the **procedure** in VHDL can not directly generate an output signal, it is necessary not to use the **procedure** if there is any output from subroutines. Each subroutine is consisted of several states and the last state of each subroutine is simply to restore the return state into the *transition*. Using same technique as the main FSM, subroutine calling can be easily implemented. The nested subroutine calls are allowed by using the *stack*.

```

stack_operation : PROCESS
VARIABLE pointer : INTEGER := 0;
VARIABLE stack : stack_ty := (others => 0);
BEGIN
WAIT ON clk;
IF( clk = '0') THEN
  IF(stack_operation = push) THEN
    stack(pointer) := return_state;
    pointer := pointer + 1;
    next_state <= transition;
  ELSIF(stack_operation = pop) THEN
    pointer := pointer - 1;
    next_state <= stack(pointer);
  ELSE
    next_state <= transition;
  END IF;
ELSE
  next_state <= null;
END IF;
END PROCESS;

PROCESS -- RETURN
BEGIN
WAIT ON clk;
IF(present_state = 1000 AND clk = '0') THEN
  stack_operation <= pop;
  memory_operation <= no;
ELSE
  stack_operation <= null;
  memory_operation <= null;
END IF;
END PROCESS;

```

Figure 7. Stack Operation and State 1000 RETURN

3.5 Interrupt Handling

If the interrupt request occurs in any state, which is shown in Figure 6 statement number 2, the next state is saved and the transition is changed to state "900", which is the start of the interrupt handling routine. At the end of the service the saved state is restored to the signal *transition* in order to resume the execution. In the microprocessor, execution is divided into three phases, fetching, decoding, and executing of an instruction. The interrupt can occur in any of these three phases. If the interrupt occurs, the current statuses are saved, the interrupt handling routine is invoked, and upon the completion of the interrupt handling routine, the saved statuses are restored and control is returned.

In a VHDL simulation, we can not find any suitable way of implementing this scenario. We can only allow an interrupt to occur when an instruction is finished executing. Although we allow an interrupt to occur after finishing execution of an instruction, it is not very helpful that this can happen in a FSM. This is because if we want to model the micro-code directly into the VHDL FSM, we have to have one state per micro-instruction. Thus we grouped several micro-instructions into one state and an interrupt can be allowed after finishing all instructions in a state.

The interrupt handler is implemented the same as a FSM and it can be viewed as a subroutine. When the interrupt request is given and the *interrupt_enable = '1'*, the next state transition is saved into the signal variable called *state_save*. Then the new present state becomes "900" which is the beginning of the interrupt handling routine. Upon completion of the interrupt handling routine, the next state is 1000 which pops the saved state and restores the right transition information into the signal variable *present_state*. The stack is popped and the return address is restored.

4 VHDL Modeling of Hazard Detection

Once the structural design of FSM is obtained from a result of High-Level Synthesis, timing correctness must be checked. One way of checking the correct timing is to check if this circuit contains any timing *hazard*. When input signals of a circuit change their values, the output values can be predicted by examining its flow table or truth table. If the output signals behave in a different manner, the circuit is said to have a timing fault, such as *hazard* or *race* for the input transition [6, 7]. For the given circuit, we need to identify whether it contains a hazard or not. Using the ternary system

with VHDL, such hazards can be detected in a combinational circuit.

In a combinational circuit, spurious output pulses may not be harmful, depending on how the output is used. However, a hazard becomes very complicated in a sequential circuit. In an asynchronous sequential circuit, a *race* can exist when two or more next state signals change together. If all of these change at the exact same time, the circuit does not contain any hazard. However, this is very unlikely in a real circuit. There may exist a little time difference between the signals changing. This affects the next state value. If the next stable state is the one that the designer predicted, the circuit is correct although it contains a race condition. If the next stable state, however, is not the one which the designer intended, which is incorrect, and it is said to contain a *critical race*. The critical race must be eliminated. We suggested VHDL modeling techniques for detecting race in general and critical race for a given asynchronous sequential circuit with a given delay.

In order to get the transient analysis for a digital circuit, a hazard condition should be checked for on both of the combinational circuits or asynchronous sequential circuits by VHDL. The two-valued logic of VHDL, i.e. '0' and '1', is extended into multiple-valued logic, i.e. '0', 'U' for unknown or middle value, and '1'. This ternary-valued logic is used for the hazard detection procedures [7, 8].

In our procedures, all of the components in a circuit are assumed to have a unit delay, or a Δ delay. This assumption is necessary because circuit behaviors differ depending on their delay model. In this situation, it is difficult to find whether there exist hazards or not. The goal is to find out the circuit's transient behavior, whether or not it possibly creates a hazard for a particular transition from the design based on the Δ delay model. It is also assumed that for any circuit, all the effects of one input change reach all the outputs before any of the inputs change again. In other words, input is allowed to change only after all the internal values are stabilized including outputs.

4.1 Hazard Detection in the Combinational Circuit

The problem of using ternary functions to detect both function hazards and logic hazards in combinational switching circuits has been proposed in the paper [7].

In the combinational circuit, a hazard can be detected by using a simple VHDL process [8]. This process checks the condition of $f(A) = f(B) \neq 1/2$ and $f(1/2) = 1/2$. If these conditions are met, the circuit contains a hazard. Otherwise, it is a hazard-free circuit. In VHDL, '1/2' is represented as 'U'.

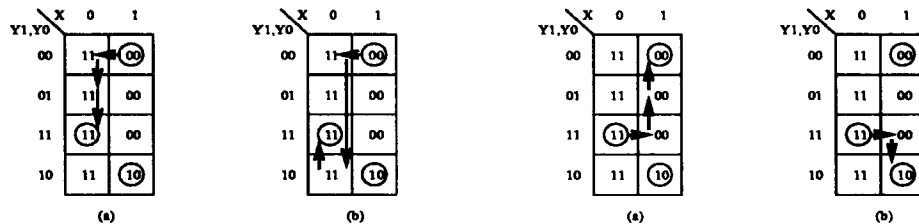


Figure 8. Noncritical Race (a) y0 changes before y1 (b) y1 changes before y0

Figure 9. Critical Race (a) Desired Response (b) Incorrect Response

4.2 Hazard Detection in Sequential Circuit

In the asynchronous sequential circuit, *race* can be a problem. During the time interval that the state is changing from $Y1$ to $Y2$, one or more bits in Y become unstable. This condition is known as a *race*. The excitation tables shown in Figure 8 and Figure 9, illustrate the race condition. Figure 8 illustrates a noncritical race and Figure 9 illustrates a critical race [9].

In the Figure 8, if input X changes from '1' to '0' when the previous state is '00', the required transition is to the new state '11'. This involves a change in the values of two state variable, Y1 and Y0. It is very unlikely in a real circuit that these two actually change simultaneously. Thus, if either Y0 or Y1 changes first, instead of going directly to the stable state '11', the circuit will go to state '01' or '10' and finally to '11'. This condition is called a *race*. In this example, however, no matter how Y's are changing, the circuit ends up with the desired state. This particular race is called a *noncritical race*, and the desired operation is always obtained.

In the Figure 9, if transition takes place when the state is '11' and input changes from '0' to '1', the circuit may or may not operate correctly depending on which of the Y's changes first. If Y1 changes first, the final stable state is '00' and desired response is obtained. This illustrated in the Figure 9 (a). However, if Y0 changes before Y1, the transition is to the state '10', which is not desired. Thus the circuit operation will be incorrect. In this case, the circuit may end up with one of two different stable states, depending on the timing. This race is called a *critical race* and it should be removed.

In VHDL, ternary logic can be used to detect the existence of a race condition in asynchronous circuit. The `process` statement in Figure 10 shows how it can be detected. In this example, however, the code can only detect whether or not the circuit contains a possible critical race condition. For example, in Figure 8, the circuit does not report race condition. In the Figure 9, the circuit does report race condition for the state variable Y1.

<pre> PROCESS BEGIN WAIT FOR TIME'HIGH; ASSERT (NOT (Y0 = 'U')) REPORT "Race Detected in Y0" SEVERITY WARNING; ASSERT (NOT (Y1 = 'U')) REPORT "Race Detected in Y1" SEVERITY WARNING; WAIT; END PROCESS; </pre>	<pre> PROCESS VARIABLE ye1, ye0: BIT ; BEGIN WAIT UNTIL y1'STABLE AND y0'STABLE; WAIT UNTIL NOT b'STABLE; ye1 := not b or (y1 and not y0); ye0 := not b; WAIT UNTIL y1'STABLE AND y0'STABLE; ASSERT (ye1 = y1 AND ye0 = y0) REPORT "Critical Race Occurs"; WAIT; END PROCESS; </pre>
---	--

Figure 10. Race Condition Detection in Sequential Circuit

Figure 11. Critical Race Condition Detection in Sequential Circuit

If we simulate the circuit in the Figure 9 with VHDL, although it may contain the critical race condition, it always ends up with the desired state '00' because we use Δ delay model. The actual circuit, with more precise delay elements associated with each gate including loop delay, must be tested whether it operates correctly or not. If it does not end up with wrong state, i.e. '10' in this example, the assigned delays are valid for this circuit. If not, by reassign delay values, it should be corrected and make sure the Y1 changes before Y0 changes. To detect the actual critical race happened, a different VHDL program is used as shown in the Figure 11.

The critical race can be determined by examining the transition table and the simulation output. If the output is the one which was expected, the circuit operates correctly. If it is not the one which was expected, the circuit operates incorrectly. In VHDL, this situation can be tested by calculating the expected outputs from the next state equations and these can then be compared with the simulation outputs. This is shown in the Figure 11.

5 Discussion and Future Research

We have shown that how VHDL can be used to retarget the current design of the *Remote I/O Module*, which uses a micro-processor, to a behavioral description of a FSM. In the redesign, the

behavior of micro-processor controller was modeled as a FSM using VHDL, and removed in the redesign. The behavioral description of the Remote I/O *CLSI* has 13,000 lines of VHDL code. There are 350 states, which are grouped to 7 blocks.

Each state was modeled as one process. This gives more readability. Each state may contain complicate sequential statements. Decomposition can be done easily, and only some of the states can be compiled and simulated independently with the whole system. This style, however, has certain limitations because of excess use of NULL assignment statements which invokes the bus-resolution function [4].

We also discussed the detection method of hazards in a combinational circuit using ternary logic in VHDL. And the same technique can be applied to detect race conditions in asynchronous circuit. The critical race can be detected using VHDL also.

Synthesis is the process which translates the high-level description into the structural description which may be the net-list and/or the structural description of VHDL. Because all of VHDL descriptions can be synthesized, some of non-synthesizable constructs in VHDL should be identified. One example might be the non-hardware descriptions, such as abstract data-types, **assertion** statement and file I/O. Other descriptions that can not be synthesized in a stable manner are the treatment of the unknown values. The **wait** statement is another example of difficulties. The **wait for** statement can not be translated into any hard-ware component; although it may be converted to the counter and latch to hold the state.

If the enumerated types of a state variable is used, it may be synthesized. The RAM is currently implemented with a two dimensional array. Multi-dimensional user defined data types are difficult to be synthesized. If there are some states which are not part of the FSM, it can be recognized by the synthesis tool using a special comment. This special comment can also be used to impose the user's constraints. A high-level synthesis tool is currently being developed which takes VHDL as design description language.

References

- [1] ITT, "Development Specification for Remote I/O Module Firmware, CDRL 2017, Vol 1, Part 2," tech. rep., ITT.
- [2] M. J. Chung, J. H. Lee, C. Y. Lee, and B. E. Reidenbach, "VHDL Modeling Benchmark Test: SINGARS Radio Circuitry," in *Government Microcircuit Applications Conference*, pp. 331-334, 1989.
- [3] J. R. Armstrong, *Chip-Level Modeling with VHDL*. Prentice Hall, 1989.
- [4] S. H. Choi, M. J. Chung, C. Y. Lee, and B. E. Reidenbach, "System Redesign with VHDL," in *Government Microcircuit Applications Conference*, 11 1990.
- [5] RCA, *RCA 1800 User Manual for the CDP 1802 COSMAC Microprocessor*, 1977.
- [6] E. J. McCluskey, *Logic Design Principles with Emphasis on Testable Semicustom Circuit*. Prentice Hall, 1986.
- [7] E. B. Eichelberger, "Hazard Detection in Combinational and Sequential Switching Circuits," *IBM Journal of Research and Development*, vol. 9, pp. 90-99, March 1965.
- [8] M. J. Chung and J. H. Lee, "Design Verification and Timing Analysis with VHDL," tech. rep., Michigan State University, 1989.
- [9] S.-F. Wu and P. D. Fisher, "Automating the Design of Large-Scale Asynchronous Sequential Logic Circuit," tech. rep., Michigan State University, 1991.