Finite State Machine Design and VHDL Coding Techniques

Iuliana CHIUCHISAN, Alin Dan POTORAC, Adrian GRAUR
"Stefan cel Mare" University of Suceava
str.Universitatii nr.13, RO-720229 Suceava
iulia@eed.usv.ro , alinp@eed.usv.ro, Adrian.Graur@usv.ro

Abstract — The first part of paper discusses a variety of issues regarding finite state machine design using the hardware description language. VHDL coding styles and different methodologies are presented. Our study of FSM focuses on the modeling issues such as VHDL coding style, state encoding schemes and Mealy or Moore machines. Our discussion is limited to the synchronous FSM, in which the transition is controlled by a clock signal and can occur only at the triggering edge of the clock.

The second part contains a worked example of a model that detects a unique pattern from a serial input data stream and generates a '1' value to output whenever the sequence '10' occurs. The string detector is modeled at the RTL level in VHDL and Verilog, for comparison purposes.

The last part of this paper presents a view on VHDL and Verilog languages by comparing their similarities and contrasting their difference.

Index Terms — VHDL code, Verilog code, finite state machine, Mealy machine, Moore machine, modeling issues, state encoding.

I. INTRODUCTION

The automata theory is the basis behind the traditional model of computation and is used for many purposes other than controller circuit design, including computer program compiler construction, proofs of algorithm complexity, and the specification and classification of computer programming languages [1].

Because automata are mathematical models that produce values dependent upon internal state and possibly some dependent input values, they are referred to as *state machines* [2]. A state machine may allow for a finite or an infinite set of possible states and further more, they may have nondeterministic or deterministic behavior. A deterministic state machine is one whose outputs are the same for a given internal state and input values. A *finite state machine* (FSM) is one where all possible state values made a finite set. The synchronous sequential circuits that are the focus of this paper are modeled as deterministic finite state machines and they are modeled as either Mealy or Moore machines.

II. OVERVIEW OF FINITE STATE MACHINES

Finite state machines (FSM) constitute a special modeling technique for sequential logic circuits. Such a model can be very helpful in the design of certain types of systems, particularly those whose tasks form a well-defined sequence [3].

The main application of an FSM is to realize operations that are performed in a sequence of steps [4]. A large digital system usually involves complex algorithms or tasks, which can be expressed as a sequence of actions based on system status and external commands. An FSM can function as the control circuit that coordinates and governs the operations of other units of the system [4].

Figure 1 shows the general structure for a finite state machine. The current state of the machine is stored in the *state memory register*, a set of *k* flip-flops clocked by a single clock signal. The *current state* is the value currently stored by the state memory register. The *next state logic circuit* of the machine is a function of the state vector and the inputs. *Mealy outputs* are a function of the state vector and the inputs, while *Moore outputs* are a function of the state vector only [5].

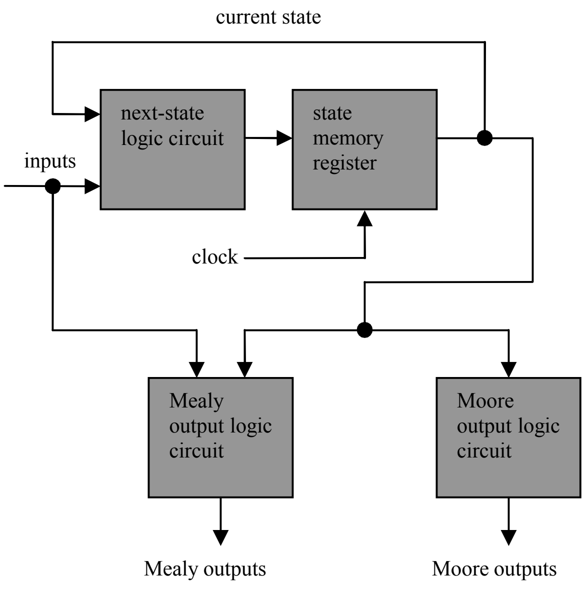


Figure 1 State Machine Structure

A finite state machine is specified by five entities: symbolic states, input signals, output signals, next-state function and output function [4]. A state specifies a unique internal condition of a system and as time progresses, the FSM transits from one state to another. The new state is determined by the next-state function, which is a function of the current state and input signals.

The output function specifies the value of the output signals. If it is a function of the state only, the output is known as a *Moore output* and if it is a function of the state and input signals, the output is known as a *Mealy output*. An FSM is called a *Moore machine* or *Mealy machine* if it contains only Moore outputs or Mealy outputs, but a complex FSM has both types of outputs.

FSMs are commonly modeled in a variety of ways, including state diagrams, state equations, state tables, and algorithmic state machine (ASM) charts.

In synthesis of FSM, we start with a functional description of the circuit. From this description, we need precise operation of the circuit using a state diagram. The state diagram allows us to complete the next-state and output tables and then the circuit can be derived from these tables.

During the synthesis process, there are many possible circuit optimizations in terms of the circuit size, speed, and power consumption that can be performed [6].

III. ENCODING STYLE

The most important decision to make when describing a finite state machine is what state encoding to use. To encode the states of a state machine, we can select from several styles, the default encoding style being binary. The advantage in using the binary code to encode state assignment is that requires the least number of flip-flops (with *n* flip-flops can be encoded up to *2n* states). The disadvantage is that it requires more logic and is slower than the others.

A highly encoded state assignment will use fewer flip-flops for the state vector; however, additional logic will be required simply to encode and decode the state [5].

A style that uses one flip-flop per state is one-hot encoded style, because only one bit of the state vector is asserted for any given state and all other state bits are zero. In this case, with n flip-flops can be encoded only *n* states.

There are more advantages to using the one-hot style to design a state machine:

- One-hot state machines are faster. Speed depends on the number of transitions into a particular state.
- It is equally "optimal" for all machines.
- One-hot state machines are easy to design and HDL code can be written directly from the state diagram without coding a state table.
- Adding and deleting states, or changing excitation equations, can be implemented easily without affecting the rest of the state machine.
- Easily synthesized from HDL languages, VHDL or Verilog.
- It is easy to debug.

An style that is in between the two styles above is the two-hot encoding style, which presents two bits active per state and therefore, with n flip-flops can be encoded up to *n(n-1)/2* states.

The encoding styles and the number of flip-flops required for a finite state machine with eight states is shown below:

TABLE 1. STATE ENCODING OF A 8-STATE FSM

STATE	ENCODING		
	BINARY STYLE	ONE-HOT STYLE	TWO-HOT STYLE
STATE1	000	00000001	00011
STATE2	001	00000010	00101
STATE3	010	00000100	01001
STATE4	011	00001000	10001
STATE5	100	00010000	00110
STATE6	101	00100000	01010
STATE7	110	01000000	10010
STATE8	111	10000000	01100
FLIP-FLOPS NUMBER	THREE FLIP-FLOPS	EIGHT FLIP-FLOPS	FIVE FLIP-FLOPS

The one-hot style is recommended in applications where flip-flops are abundant, like in FPGA circuits. CPLD circuits have fewer flip-flops available to the designer.

While one-hot encoding is sometimes preferred because it is easy, a large state machine will require a large number of flip-flops. Therefore, when implementing finite state machines on CPLD circuits, in order to conserve available resources, it is recommended that binary or gray encoding be used [7]. That enables the largest number of states to be represented by as few flip-flops as possible.

IV. HDL LANGUAGES

Most hardware designers use hardware description languages (HDLs) to describe designs at various levels of abstraction. A hardware description language is a high level programming language, with programming constructs such as assignments, conditions, iterations and extensions for timing specification, concurrency and data structure proper for modeling different aspects of hardware. The most popular hardware description languages are VHDL [8] and Verilog [9].

VHDL (VHSIC (*Very High Speed Integrated Circuits*) *Hardware Description Language*) [8] is an IEEE Standard since 1987 while Verilog was standardized in 1995.

Both languages are programming language that has been designed and optimized for describing the behavior of digital systems. This HDL languages support the development, verification, synthesis, and testing of hardware designs.

In this paper we chose the VHDL language. One important aspect related to the FSM approach in VHDL code is that, though any sequential circuit can in principle be modeled as a state machine, this is not always advantageous [3]. The reason is that the code might become longer, more complex, and more error prone than in a conventional approach [3].

The FSM approach is adequate in systems whose tasks constitute a well-structured list so all states can be easily enumerated. That is, in a typical state machine implementation, we will encounter, at the beginning of the *ARCHITECTURE*, a user-defined enumerated data type, containing a list of all possible system states [3].

V. FSM VHDL DESIGN AND MODELING ISSUES

A Finite State Machines are an important aspect of hardware design. A well written model will function correctly and meet requirements in an optimal manner.

Finite state machine VHDL design issues to consider are:

- VHDL coding style.
- How many processes we use?
- State encoding.
- Mealy or Moore type outputs.

A. VHDL coding style

There are many ways of modeling the same state machine. Our example of FSM focuses on simple tasks, such as detecting a unique pattern from a serial input data stream and generating a '1' value to output whenever the sequence '10' occurs.

The state diagram of our string detector circuit is shown in *figure 2*. There are three states, which we called s0, s1, and s2.

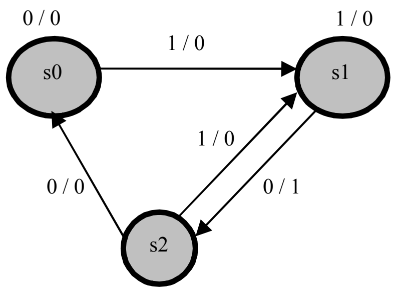


Figure 2. FSM State diagram

TABLE 2. TABLE WITH CURRENT STATE, NEXT STATE AND MEALY/MOORE OUTPUT FOR STRING DETECTOR CIRCUIT

CURRENT STATE	NEXT STATE		MEALY OUTPUT		MOORE OUTPUT
	A=0	A=1	A=0	A=1	
s0	s0	s1	0	0	0
s1	s2	s1	1	0	0
s2	s0	s1	0	0	1

Simulation results are shown in *figure 3*. As can be seen, the data sequence A="010110110" was applied to the circuit, resulting the response F= "001001010" at the output F.

HDL code may be divided into three different parts to represent current state logic, next state logic and Mealy or Moore output logic. It may also be structured so that the three different parts are combined in the model. In VHDL, it is impossible to synthesize a combined current state, next state, and output logic in a single always statement.

A FSM with *n* state flip-flops may have *2n* binary numbers that can encode states and often, all states are not needed. Therefore, next-state logic is best modeled using the case statement even though this means the state machine cannot be modeled in one process. The default clause used in a case statement avoids having to define the unused states.

B. How many processes?

Generally every finite state machine can be described either by one process or by two separated processes.

In the following table, are presented the corresponding parts of the VHDL source code for one process and two processes design for the string detector circuit.

TABLE 3.VHDL DESIGN FOR STRING DETECTOR CIRCUIT

ONE VHDL PROCESS	TWO VHDL PROCESS
FSM_ONE: PROCESS (CLK, RST)	--*PROCESS TO HOLD SYNCHRONOUS ELEMENTS*
BEGIN	**FSM_SYNCH**: PROCESS (CLK, RST)
IF (RST='1') THEN	BEGIN
CURRENT_STATE <= S0;	IF (RST='1') THEN
ELSIF	CURRENT_STATE <= S0;
(CLK'EVENT AND CLK='1')	ELSIF (CLK'EVENT AND CLK='1')
THEN	THEN
CASE CURRENT_STATE IS	CURRENT_STATE <= NEXT_STATE;
WHEN S0 =>	END IF;
IF (A='0') THEN	END PROCESS FSM_SYNCH;
F<='0';	--*PROCESS TO HOLD COMBINATIONAL LOGIC*
NEXT_STATE <= S0;	**FSM_COMB**: PROCESS (A, CURRENT_
ELSE	STATE) BEGIN
F<='0';	CASE CURRENT_STATE IS
NEXT_STATE <= S1;	WHEN S0=>
END IF;	IF (A='0') THEN
WHEN S1 =>	F<='0';
IF (A='0') THEN	NEXT_STATE <=S0;
F<='01';	ELSE
NEXT_STATE <= S2;	F<='0';
ELSE	NEXT_STATE <= S1;
F<='0';	END IF;
NEXT_STATE <= S1;	WHEN S1=>
END IF;	IF (A='0') THEN
WHEN S2 =>	F<='1';
IF (A='0') THEN	NEXT_STATE <= S2;
F<='1';	ELSE
NEXT_STATE <= S0;	F<='0';
ELSE	NEXT_STATE <= S1;
F<='0';	END IF;
NEXT_STATE <= S1;	WHEN S2=>
END IF;	IF (A='0') THEN
WHEN OTHERS =>	F<='0';
CURRENT_STATE <= S0;	NEXT_STATE <= S0;
END CASE;	ELSE
END IF;	F<='0';
END PROCESS FSM_ONE;	NEXT_STATE <= S1;
	END IF;
	WHEN OTHERS =>
	CURRENT_STATE <= S0;
	END CASE;
	END PROCESS FSM_COMB;

In one process version, in the *CASE* statement which models the state transitions, the current state of the string detector FSM is detected and it is examined whether input values are present that lead to a change of the state.

The same FSM is used to show an implementation based on two VHDL processes. The VHDL source code contains two processes, one process defines synchronous elements of the design (state registers) and the other process defines the combinational part of the design (case statement). The result is a clocked process describing the storing elements and another combinational process describing the logic. In the *CASE* statement, the current state is checked and the input values are examined. If the state has to change, then *NEXT_STATE* and *CURRENT_STATE* will differ and with the next appearance of the active clock edge, this new state will be taken over as the current state.

There are different advantages and disadvantages of using one process or two processes and these differences are presented below.

1) Structure and legibility

The one process description is more adequate for observing changes of the states of the string detector FSM from the outside of the VHDL module. The graphical description resembles more a one process than a two process description.

The VHDL source code should be split into two exactly the time and location where the error occurs for the first time and therefore the source of the error.

3) Synthesis

Several synthesis tools tend to produce better results when two processes are used to describe a finite state machine.

C. State encoding in VHDL

A finite state machine is an abstract description of digital structure and therefore the synthesis tools requires states of the automaton to be encoded as binary values or the synthesis tool itself will transform the state into a binary description. The way in which states are assigned binary values is called state encoding.

Some different state encoding schemes frequently used are presented in *table 1*.

Most synthesis tools selects a binary code by default, except the designer specifies another code explicitly. From all possibilities of state encoding is used frequently one-hot code, which is needed for speed optimized circuits.

TABLE 4. STATE ENCODING

STATE ENCODING		
TYPE STATE_TYPE IS (S0, S1, S2) ; SIGNAL STATE : STATE_TYPE ;		
BINARY CODE	**ONE-HOT CODE**	**GRAY CODE**
s0 → "00" s1→ "01" s2→ "10"	s0 → "001" s1→ "010" s2→ "100"	s0 → "00" s1→ "01" s2→ "11"

The circuit requires two flip-flops, which encode the three states of the string detector state machine.

As the automat consists only of three states and two flip-flops can represent up to four states, there is one invalid state which leads to an unsafe state machine where the behavior of the design is not determined.

The best method of state encoding is hand coding in which case the designer decides by himself what code will be used.

TABLE 5. HAND CODING

```
SUBTYPE STATE_TYPE IS STD_LOGIC_VECTOR (1 DOWNTO 0) ;
SIGNAL STATE : STATE_TYPE ;
CONSTANT S0: STATE_TYPE := "00";
CONSTANT S1: STATE_TYPE := "10";
CONSTANT S2: STATE_TYPE := "11";
```

The constants are defined to represent the corresponding states of the state machine and the code can be fixed by the designer. The behavior in case of errors can be verified in a simulation and therefore the hand coding alternative is the best method to design a safe finite state machine and is furthermore portable among different synthesis tools.

This type of state encoding has the advantage of using a vector type and the only disadvantage is a little more effort in writing the VHDL code, when the code is changed.

processes because the combinational elements and synchronous elements (state registers) are two different structural elements.

2) Simulation

For two process version, can be determined

D. MOORE MACHINE VERSUS MEALY MACHINE

The two most popular state machines are referred to as Moore or Mealy machines, named after researchers who published early papers on their structure [10, 11]. There are different reasons for a designer to use one or other version of the two different type of finite state machine. The primary difference between these two state machines is that the output of a Moore machine depends only upon the state of the circuit whereas the output of a Mealy machine depends upon both the state and the inputs of the circuit. This has a practical effect in that the output signals of a Moore machine only change after output logic delays following a clock signal edge whereas the output signals of a Mealy machine may change at any time shortly after an input signal changes value [1].

In theoretical computer science, a Moore machine and a Mealy machine are considered to have similar efficiency because both can recognize “regular expressions” [4]. When the FSM is used as a control circuit, the control signals generated by a Moore machine and a Mealy machine have different timing characteristics and for the efficiency of a control circuit the timing difference is critical. We used a simple edge detection circuit to observe the difference between these two state machines.

There are three major differences between the Moore machine and Mealy machine. First, a Mealy machine requires fewer states to perform the same task because its output is a function of states and external inputs, and several possible output values can be specified in one state.

Second, a Mealy machine can generate a faster response. Since a Mealy output is a function of input, it changes whenever the input meets the designated condition and a Moore machine reacts indirectly to input changes.

The third difference involves the control of the width and timing of the output signal. In a Mealy machine, the width of an output signal varies with input and can be very narrow. A Mealy machine is susceptible to disturbances in the input signal and passes to the output. The output of a Moore machine is synchronized with the clock edge and its width is about the same as a clock period.

From this perspective, selection between a Mealy machine and a Moore machine depends on the need of control signals. If we divide control signals into two categories: edge-sensitive and level-sensitive, then for the first case, both the Mealy and the Moore machines can generate output signals that meet this requirement. However, a Mealy machine is preferred since it uses fewer states and responds one clock faster than does a Moore machine. A level-sensitive control signal means that a signal has to be asserted for a certain amount of time. When asserted, it has to be stable and free of spikes.

A Moore machine is preferred since it can accurately control the activation time of its output, and can shield the control signal from input glitches.

TABLE 6. MEALY AND MOORE VHDL DESIGN FOR A STRING DETECTOR

MEALY VHDL DESIGN	**MOORE VHDL DESIGN**
ENTITY **MEALY** IS	ENTITY **MOORE** IS
PORT(A,CLK,RST: IN BIT;	PORT(A,CLK,RST: IN BIT;
F: OUT BIT); END MEALY;	F: OUT BIT); END MOORE;
ARCHITECTURE **FSM** OF MEALY IS	ARCHITECTURE **FSM** OF MOORE IS
SUBTYPE STATE_TYPE IS STD_	SUBTYPE STATE_TYPE IS STD_
LOGIC_VECTOR (2 DOWNTO 0);	LOGIC_VECTOR (2 DOWNTO 0);
SIGNAL STATE : STATE_TYPE;	SIGNAL STATE : STATE_TYPE;
CONSTANT S0:	CONSTANT S0:
STATE_TYPE:="001";	STATE_TYPE:="001";
CONSTANT S1:	CONSTANT S1:
STATE_TYPE:="010";	STATE_TYPE:="010";
CONSTANT S2:	CONSTANT S2:
STATE_TYPE:="100";	STATE_TYPE:="100";
SIGNAL CURRENT_STATE,	SIGNAL CURRENT_STATE,
NEXT_STATE : STATE_TYPE;	NEXT_STATE : STATE_TYPE;
BEGIN	BEGIN
FF: PROCESS (CLK, RST) BEGIN	**FF**: PROCESS (CLK, RST) BEGIN
IF (RST='1') THEN	IF (RST='1') THEN
CURRENT_STATE <= S0 ;	CURRENT_STATE <= S0 ;
ELSIF (CLK'EVENT AND CLK='1')	ELSIF (CLK'EVENT AND CLK='1')
THEN	THEN
CURRENT_STATE<= NEXT_STATE;	CURRENT_STATE<= NEXT_STATE;
END IF ;	END IF ;
END PROCESS;	END PROCESS FF ;
LOGIC: PROCESS (A,	**LOGIC**: PROCESS (A,
CURRENT_STATE) BEGIN	CURRENT_STATE) BEGIN
CASE CURRENT_STATE IS	CASE CURRENT_STATE IS
WHEN S0 => IF (A='0') THEN	WHEN S0=> F <= '0';
F <= '0';	IF (A='0') THEN
NEXT_STATE <= S0;	NEXT_STATE <=S 0;
ELSE F <= '0';	ELSE
NEXT_STATE <= S1;	NEXT_STATE <= S1;
END IF;	END IF;
WHEN S1 => IF (A='0') THEN	WHEN S1=> F <= '0';
F <= '0';	IF (A='0') THEN
NEXT_STATE <= S2;	NEXT_STATE <= S2;
ELSE F<='0';	ELSE
NEXT_STATE<=S1;	NEXT_STATE <= S1;
END IF;	END IF;
WHEN S2 => IF (A='0') THEN	WHEN S2=> F <= '1';
F<='1';	IF (A='0') THEN
NEXT_STATE<=S0;	NEXT_STATE <= S0;
ELSE F<='0';	ELSE
NEXT_STATE<=S1;	NEXT_STATE <= S1;
END IF;	END IF;
WHEN OTHERS =>	WHEN OTHERS =>
CURRENT_STATE <= S0;	CURRENT_STATE <= S0;
END CASE;	END CASE;
END PROCESS;	END PROCESS;
END FSM;	END FSM ;

VI. VHDL/VERILOG COMPARED & CONTRASTED

Hardware structures can be modeled equally effectively in both languages, VHDL and Verilog. Choosing which of these hardware description languages can use them only depends on personal preferences, EDA tool availability or commercial and marketing issues.

Verilog HDL allows a designer to describe designs at a high level of abstraction such as at the structural or behavioral level as well as the lower implementation levels leading to Very Large Scale Integration Integrated Circuits layouts and chip fabrication. A basic use of HDL languages is the simulation of designs before the designer must commit to fabrication.

VHDL is also a hardware description language that describes the behavior of an electronic circuit or system, from which the physical circuit can then be implemented [3]. This language is designed to fill a number of needs in the design process:

- It allows description of the structure of a design that is how it is decomposed into sub-designs, and how those sub-designs are interconnected;
- It allows the specification of the function of designs using familiar programming language forms;
- It allows a design to be simulated before being manufactured, so that designers can test for correctness and compare alternatives.

Regarding *design reusability* in Verilog the functions and procedures must be placed in a separate file and included using include compiler directive to make them accessible from different module statements [12]. In the VHDL case, the procedures and functions may be placed in a *package* so that they are available to any *design-unit* that wishes to use them.

Compared to VHDL, Verilog *data types* are very simple, easy to use and unlike VHDL, all data types used in a Verilog model are not defined by the user but by the Verilog language. Objects of type *reg* hold their value over simulation cycles and should not be confused with the modeling of a hardware register. Verilog may be preferred because of simplicity, but VHDL may be recomended because it allows a multitude of user defined data types to be used.

There are more VHDL constructs and features for *high-level* modeling, compared with Verilog, where is no equivalent to the high-level VHDL modeling statements. In VHDL abstract data types can be used along with the following statements: package statements for model reuse, configuration statements for configuring design structure, generate statements for replicating structure and generic statements for generic models [12].

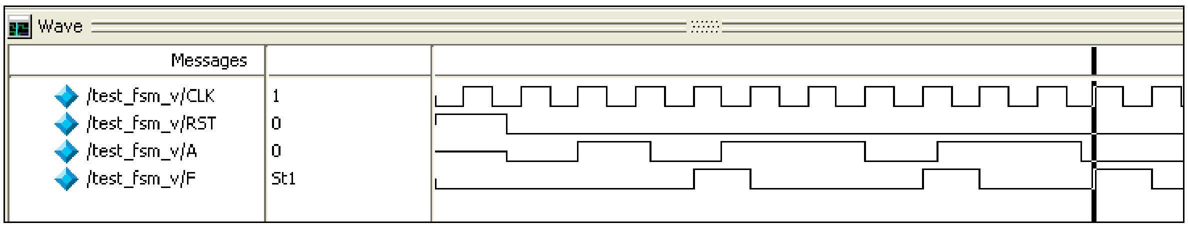
There is no concept of *library* in Verilog and this is due to origins as an interpretive language. In VHDL a library is a store for compiled entities, architectures, packages and configurations and they are useful for managing multiple design projects, compared with Verilog, where are not statements who manage large designs.

A final specification regarding *VHDL* is that, contrary to regular computer programs which are sequential, its statements are concurrent and only statements placed inside a process, function, or procedure are executed sequentially. For that reason, VHDL is referred to as a code rather than a program.

Model of a string detector circuit was proposed as problem for describing a finite state machine using VHDL and in this last part of paper we include a model written in Verilog in addition to VHDL, for comparison purposes.

TABLE 7. MEALY AND MOORE VERILOG DESIGN FOR A STRING DETECTOR CIRCUIT

MEALY VERILOG DESIGN	MOORE VERILOG DESIGN
MODULE	MODULE
MEALY_STRING_DETECTOR (F, A,	**MOORE**_STRING_DETECTOR (F, A,
CLK, RST);	CLK, RST);
OUTPUT F;	OUTPUT F;
INPUT A, CLK, RST;	INPUT A, CLK, RST;
REG F;	REG F;
REG [2:0] CURRENT_STATE,	REG [2:0] CURRENT_STATE,
NEXT_STATE;	NEXT_STATE;
PARAMETER	PARAMETER
S0=3'B001,	S0=3'B001,
S1=3'B010,	S1=3'B010,
S2=3'B100;	S2=3'B100;
ALWAYS	ALWAYS
@ (POSEDGE CLK AND POSEDGE	@ (POSEDGE CLK OR POSEDGE RST)
RST) BEGIN	BEGIN
IF (RST = = 1)	IF (RST = = 1)
CURRENT_STATE <= S0 ;	CURRENT_STATE <= S0 ;
ELSE	ELSE
CURRENT_STATE<= NEXT_STATE;	CURRENT_STATE<= NEXT_STATE;
END	END
ALWAYS	ALWAYS
@(CURRENT_STATE OR A)	@(CURRENT_STATE OR A)
BEGIN	BEGIN
CASE (CURRENT_STATE)	CASE (CURRENT_STATE)
S0 : IF (A = = 0) BEGIN	S0 :
F <= 0;	BEGIN
NEXT_STATE <= S0;	F <= 0;
END	IF (A = = 0)
ELSE BEGIN	NEXT_STATE <= S0;
F <= 0;	ELSE
NEXT_STATE <= S1;	NEXT_STATE <= S1;
END	END
S1 : IF (A = = 0) BEGIN	S1 :
F <= 0;	BEGIN
NEXT_STATE <= S2;	F <= 0;
END	IF (A = = 0)
ELSE BEGIN	NEXT_STATE <= S2;
F<= 0;	ELSE
NEXT_STATE <= S1;	NEXT_STATE <= S1;
END	END
S2 : IF (A = = 0) BEGIN	S2 :
F<=1;	BEGIN
NEXT_STATE <= S0;	F <= 1;
END	IF (A = = 0)
ELSE BEGIN	NEXT_STATE <= S0;
F<=0;	ELSE
NEXT_STATE <= S1;	NEXT_STATE <= S1;
END	END
DEFAULT :	DEFAULT :
CURRENT_STATE <= S0;	CURRENT_STATE <= S0;
ENDCASE	ENDCASE
END	END
ENDMODULE	ENDMODULE



```
/test_fsm_v/CLK   1
/test_fsm_v/RST   0
/test_fsm_v/A     0
/test_fsm_v/F     St1
```

Figure 3. Simulation Results for String Detector circuit

VII. CONCLUSIONS

This paper shows the relationship between finite state machines and VHDL/Verilog code. A fundamental motivation to use VHDL or Verilog is that both are a standard, technology independent language, and are therefore portable and reusable.

There are many ways to describe FSM designs and some design rules in HDL languages are:

- Use parameters to define state encodings. Parameters are constants that are local to a module and after defining the state encodings at the top of the module, never use the state encodings again in the code. This method makes it possible to easily change the state codings in just one place in module.
- Use a two process/always block coding style to describe FSM designs with combinational outputs because this style is efficient and can easily describe Mealy designs.
- Avoid the one process/always block FSM coding style because this code style is more complicate than two process/always block coding style and output assignments are more error prone to coding mistakes.

REFERENCES

[1] Hopcroft, J. E., Ullman, J. D., ”Introduction to Automata Theory, Languages, and Computation”, Addison-Wesley, 1979.

[2] Reese, R. B. , Thornton, M. A., ”Introduction to Logic Synthesis using Verilog HDL”, Morgan & Claypool Publishers' series Synthesis lectures on digital circuits and systems, USA, ISSN: 1930-3166, pp. 46-79, 2006

[3] Volnei, A .Pedroni, ”Circuit Design with VHDL”, MIT Press Cambridge, Massachusetts, London, England, ISBN 0-262-16224-5, pp. 159-186, 2004

[4] Chu, Pong P., ”RTL hardware design using VHDL”, A Wiley-Interscience publication, USA, ISBN: 978-0-471-72092-8, pp. 313-373, 2006

[5] Golson, S. , ”State Machine Design Techniques for Verilog and VHDL”, Synopsys Journal of High-Level Design, pp. 1-2, 1994

[6] Enoch O. Hwang, “Digital Logic and Microprocessor Design With VHDL”, ISBN 0-534-46593-5, pp. 328-377, 2004

[7] XILINX – “A CPLD VHDL Introduction”, 1-800-255-7778, 2001

[8] IEEE Standard 1076-1993, IEEE Standard Description Language Based on the VHDL Hardware Description Language, 1993.

[9] IEEE Standard 1364-2001, IEEE Standard Description Language Based on the Verilog Hardware Description Language, 2001.

[10] Mealy, G. H., “A method for synthesizing sequential circuits,” Bell System Tech. J., Vol. 34, No. 5, pp. 1045–1079, 1955.

[11] Moore, E. F., “Gedanken experiments on sequential machines,” Automata Studies. Princeton, NJ: Princeton University Press, pp. 129–153, 1956.

[12] Smith, D. J., “VHDL & Verilog Compared & Contrasted Plus Modeled Example Written in VHDL, Verilog and C”, Proceedings of the 33rd annual Design Automation Conference, USA 1996