FSMs - Design Considerations and VHDL Modeling for use with RTL level Synthesis

Douglas J. Smith
Intergraph Corporation
One Madison Industrial Estate, Huntsville, AL 35894-0001, USA
e-mail: djsmith@ingr.com

Abstract

The different issues to consider when designing and modeling Finite State Machines (FSMs) in VHDL for use with RTL level synthesis are shown. The issues covered are: coding style, resets and fail safe behavior, state encoding, Mealy or Moore type outputs, additional sequential next state or output logic, and interactive FSMs. VHDL models are included.

1 Introduction

Designers of digital circuits are invariably faced with needing to design circuits that perform specific sequences of operations, e.g. controllers used to control the operation of other circuits. Finite State Machines (FSMs) have proven to be a very efficient means of modeling sequencer circuits. By modeling FSMs in VHDL for use with synthesis tools, designers can concentrate on modeling the desired sequences of operations without being overly concerned with circuit implementation; this is left to the synthesis tool. FSMs are an important part of hardware design and hence VHDL hardware modeling.

A designer should consider the different aspects of an FSM before attempting to write a model. A well written model is essential for a functionally correct circuit that meets your requirements in the most optimal manner. A badly written model may not meet either criteria. For this reason, it is important to fully understand FSMs and to be familiar with the different VHDL modeling issues.

2 Definitions

2.1 The FSM

A FSM is any circuit specifically designed to sequence through specific patterns of states in a sequential manner, and which conforms to the structure shown in Figure 1. A state is represented by the binary value held on the current state register. The FSM structure consists of three parts and may, or may not, be reflected in the structure of the VHDL code that is used to model it.

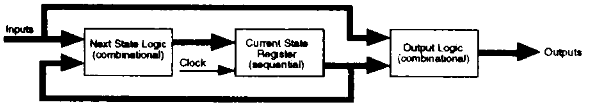
Inputs
Next State Logic (combinational)
Clock
Current State Register (sequential)
Output Logic (combinational)
Outputs

Figure 1. Basic structure of a Finite State Machine

1. Current State Register: Register of n-bit flip-flops used to hold the current state of the FSM. Its value represents the current stage in the particular sequence of operations being performed. When operating, it is always clocked from a free running clock source.

2. Next State Logic: Combinational logic used to generate the next stage (state) in the sequence. The next state output is a function of the FSM's inputs and it's current state.

3. Output Logic: Combinational logic used to generate required output signals. Outputs are a function of the state register output and possibly FSM inputs.

2.2 State Diagrams and State Tables

A state diagram is a graphical representation of an FSM's sequential operation. State diagrams are often supported as a direct input to Electronic Design Automation (EDA) tools from which synthesized circuits and VHDL simulation models are generated. Figure 2 shows two state diagram representations of the same five state, state machine; the equivalent state table is shown in Table 1.

Inputs		Current state	Next state	Outputs	
A	Hold			Y_Me	Y_Mo
0	X	000 (ST0)	000 (ST0)	1	0
1	X	000 (ST0)	001 (ST1)	0	0
0	X	001 (ST1)	000 (ST0)	0	1
1	X	001 (ST1)	010 (ST2)	1	1
X	X	010 (ST2)	011 (ST3)	0	0
X	1	011 (ST3)	011 (ST3)	1	1
0	0	011 (ST3)	000 (ST0)	1	1
1	0	011 (ST3)	100 (ST4)	0	1
X	X	100 (ST4)	000 (ST0)	0	1

X = don't care condition

Table 1. State table for state diagrams of Figure 1

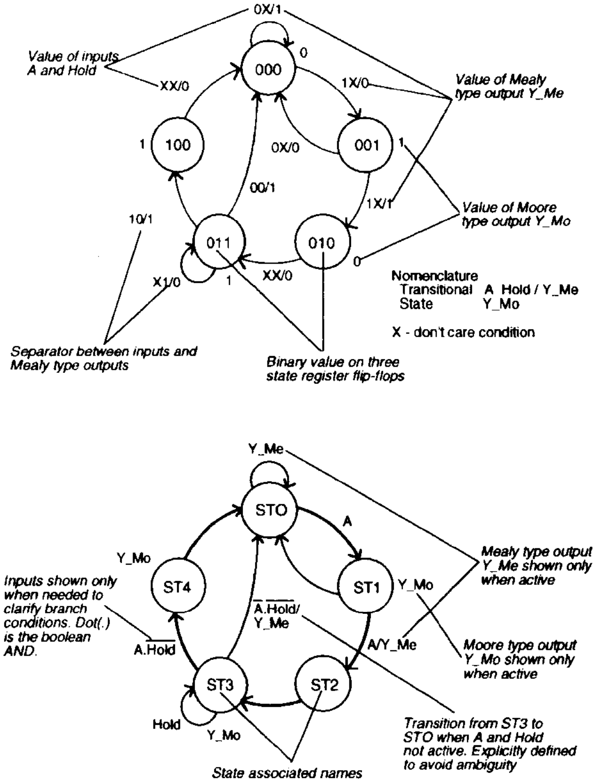
0X/1
0
Value of inputs A and Hold
XX/0
000
1X/0
Value of Mealy type output Y_Me
1
100
0X/0
001
1
00/1
1X/1
Value of Moore type output Y_Mo
10/1
011
010
0
X1/0
1
XX/0
Nomenclature
Transitional A Hold / Y_Me
State Y_Mo
X - don't care condition
Separator between inputs and Mealy type outputs
Binary value on three state register flip-flops
Y_Me
STO
A
Y_Mo
Mealy type output Y_Me shown only when active
Inputs shown only when needed to clarify branch conditions. Dot(.) is the boolean AND.
ST4
ST1
Y_Mo
$\overline{A}.\overline{Hold}$/ Y_Me
A.$\overline{Hold}$
A/Y_Me
Moore type output Y_Mo shown only when active
ST3
ST2
Hold
Y_Mo
Transition from ST3 to STO when A and Hold not active. Explicitly defined to avoid ambiguity
State associated names

Figure 2. Two equivalent state diagrams

The description of the two state diagrams of Figure 2 now follows.

Circles represent states and arrowed lines represent transitions between states which occur after every clock cycle. The clock signal is implied, but not shown on a state diagram, nor in a state table.

The binary number representing the value on the state register flip-flops (first state diagram), or it's associated state name (second state diagram) is contained inside the circle. The input signal conditions that dictate state transitions, are indicated next to the appropriate line and before any slash (/). A slash is used to separate input and output signals. The two inputs, A and Hold, are shown before the slash. Values shown after the slash, if any, indicate output signal values that are a function of both the inputs and current state register; these are Mealy type outputs described later. The value of output signals that are a function of the current state register only, are shown next to the circle representing the appropriate state. These are Moore type outputs, also described later. The major difference of the second state diagram, is that input and output signals are shown only when they are active, otherwise they are left off to aid functional comprehension, and avoid cluttering the diagram.

3 Design and Modeling Issues

FSM design and modeling issues to consider are:

1. VHDL coding style,
2. resets and fail safe behavior,
3. state encoding,
4. Mealy or Moore type outputs,
5. additional sequential next state & output logic,
6. Interactive FSMs.

The structure of a state machine can take one of three forms, depending upon the type of output see Figure 3. The current state is stored using flip-flops; latches would cause state oscillations when transparent. The next state and output logic blocks may contain additional sequential logic, inferred from within the body of the VHDL model, but is not considered part of the state machine. A state machine can only be in one state at any given time, and each active transition of the clock causes it to change from its current state to the next as defined by the next state logic.

A state machine with "n" state flip-flops has 2^n possible binary numbers that can be used to represent states. Often, not all 2^n numbers are needed, so the unused ones should be designed not to occur during normal operation. A state machine with five states, for example, requires a minimum of 3 flip-flops in which case there are 2^3-5 = 3 unused binary numbers.

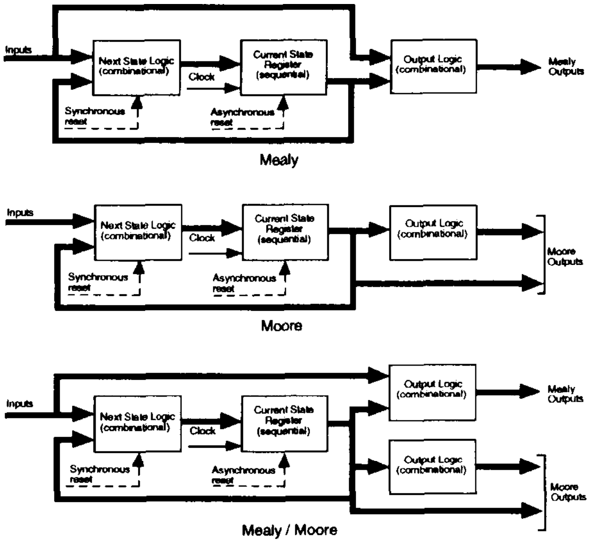
Inputs
Next State Logic (combinational)
Clock
Current State Register (sequential)
Output Logic (combinational)
Mealy Outputs
Synchronous reset
Asynchronous reset
Mealy
Inputs
Next State Logic (combinational)
Clock
Current State Register (sequential)
Output Logic (combinational)
Moore Outputs
Synchronous reset
Asynchronous reset
Moore
Inputs
Output Logic (combinational)
Mealy Outputs
Next State Logic (combinational)
Clock
Current State Register (sequential)
Output Logic (combinational)
Synchronous reset
Asynchronous reset
Moore Outputs
Mealy / Moore

Figure 3. FSM Structures

3.1 VHDL coding style

There are different ways of modeling the same state machine, on the other hand, a small code change can cause a model to behave differently than expected. Designers should be aware of the different modeling styles supported by the synthesis tool being used, and should consider modeling FSMs to be tool independent; this applies to the model of any circuit. VHDL code may be structured into three separate parts representing the three parts of a state machine, as shown in Figure 1. Alternatively, different combinations of the three parts can be combined within the code. Either way the chosen coding style is independent of the state machine being modeled.

3.2 Resets and fail safe behavior

Depending upon the application; a reset signal may not be available, there may only be a synchronous or asynchronous reset, or there may be both. To guarantee fail safe behavior, one of two things must be done, depending on the type of reset:

1. Use an asynchronous reset: This ensures the state machine is always initialized to a known valid state, before the first active clock transition and normal operation commences. This has the advantage of not needing to decode any unused current state register values, and so minimizes the next state logic.

2. With no reset or a synchronous reset: In the absence of an asynchronous reset there is no way of predicting the initial value of the state register flip-flops when implemented in an IC and "powered up". It could power up and become permanently stuck in an uncoded state. All 2^n binary values must therefore be decoded in the next state logic, whether they form part of the state machine or not. There is generally only a small area overhead in the next state logic in doing this, and is partially offset by using smaller flip-flops that do not have an asynchronous reset input.

3.3 State Encoding

The way in which binary numbers are assigned to states, is called the state encoding. The different state encoding formats are:

sequential,
gray,
Johnson,
one-hot,
define your own,
defined by synthesis.

These are described below and the four standard state encoding formats are shown in Table 2 for 16 states.

Sequential: Each state is simply assigned increasing binary numbers.

Gray and Johnson: Each state in both Gray and Johnson state encoding, is assigned

No.	Sequential	Gray	Johnson	One-Hot
0	0000	0000	00000000	0000000000000001
1	0001	0001	00000001	0000000000000010
2	0010	0011	00000011	0000000000000100
3	0011	0010	00000111	0000000000001000
4	0100	0110	00001111	0000000000010000
5	0101	0111	00011111	0000000000100000
6	0110	0101	00111111	0000000001000000
7	0111	0100	01111111	0000000010000000
8	1000	1100	11111111	0000000100000000
9	1001	1101	11111110	0000001000000000
10	1010	1111	11111100	0000010000000000
11	1011	1110	11111000	0000100000000000
12	1100	1010	11110000	0001000000000000
13	1101	1011	11100000	0010000000000000
14	1110	1001	11000000	0100000000000000
15	1111	1000	10000000	1000000000000000

Table 2. Standard State Machine Encoding Formats

successive binary numbers where only one bit changes from one number to the next. A primary motive for using such coding, is to reduce the possibility of state transition errors caused by asynchronous inputs changing during flip-flop setup times. This causes metastable states, that may last as long as the duty cycle of the clock. This occurs anywhere asynchronous signals are synchronized.

All 2^n binary numbers can be used when Gray coded state encoding is used. However, because of the pattern of 1's and 0's in Johnson state encoding, more flip-flops are required, and there are always unused binary numbers. This means that an asynchronous reset is preferred, though not essential for reasons given in 3.2.

One-hot: In "one hot" state encoding, each state is assigned its own flip-flop, so 'n' states requires 'n' flip-flops and only one flip-flop is in it's true state at any one time. The increased number of flip-flops and often increased next state logic usually results in a larger overall circuit.

Define your own: Each state is assigned a binary number according to a particular design requirement.

Defined by Synthesis: These formats are chosen by the synthesis tool to minimize next state logic and means the actual assignments are design dependent

.

3.4 Mealy or Moore type Outputs

The structure of; a Mealy, a Moore, and a combined Mealy/Moore state machine was shown in Figure 3. A Mealy state machine has outputs that are a function of the current state, and primary inputs. A Moore state machine has outputs that are a function of the current state only, and so includes outputs direct from the state register. If all outputs come direct from the state register, there is no output logic. A combined Mealy/Moore state machine has both types of output. The choice between using Mealy or Moore type outputs are clearly application dependent.

3.5 Sequential Next State or Output Logic

Both the next state and output logic in a state machine, consists of combinational logic only. However, depending upon the application, you may want to model additional sequential logic in either of these blocks, and which may be imbedded within the code of the state machine model.

Sequential Next State Logic: Extra imbedded sequential next state logic is used to control state branching from previously set signals. The output from this sequential logic can be set when; the state machine was in another state, it passed through a particular sequence of states, or because of some accumulated value resulting from looping around successive sequences of states. These next state control signals could also provide overall model outputs.

Sequential Output Logic: Extra sequential output logic is used to indicate the fact that a certain state, or sequence of states has occurred. Like the sequential next state signals, outputs signals can be set when; the state machine was in another state, it passed through a particular sequence of states, or because of some accumulated value resulting from looping around successive sequences of states.

3.6 Interactive FSMs

If a state machine's current state or output signals are used to influence the operation of other state machines, they are known to be interactive. Interaction between state machines may be unidirectional or bidirectional.

State machines may be hierarchically structured, in which case, they are useful in breaking down large complicated control path structures into more manageable pieces. Figure 4 shows the structure of two state machines where FSM1 has unidirectional control over FSM2. The next state of FSM2 is dependent upon it's own inputs and current state, plus the current state of FSM1.

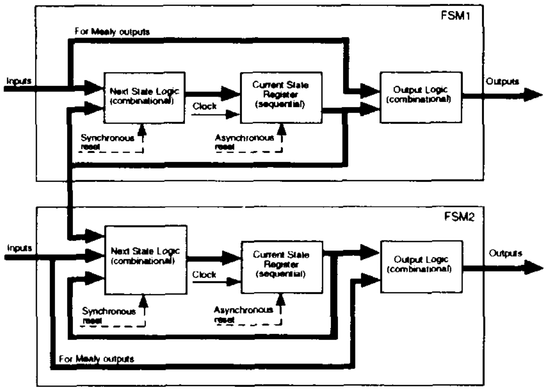
FSM1
For Mealy outputs
Inputs
Next State Logic (combinational)
Clock
Current State Register (sequential)
Output Logic (combinational)
Outputs
Synchronous reset
Asynchronous reset
FSM2
Inputs
Next State Logic (combinational)
Clock
Current State Register (sequential)
Output Logic (combinational)
Outputs
Synchronous reset
Asynchronous reset
For Mealy outputs

Figure 4. Structure of two FSMs with unidirectional interaction

State machines having bidirectional control over each other are useful for modeling circuits requiring handshaking mechanisms. Figure 5 shows the structure of three interactive state machines where each state machine has bidirectional control over the other two.

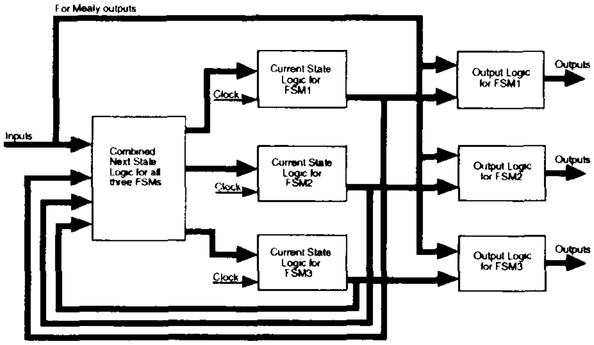
For Mealy outputs
Inputs
Combined Next State Logic for all three FSMs
Current State Logic for FSM1
Clock
Output Logic for FSM1
Outputs
Current State Logic for FSM2
Clock
Output Logic for FSM2
Outputs
Current State Logic for FSM3
Clock
Output Logic for FSM3
Outputs

Figure 5. Structure of three bidirectional interactive FSMs

4 Coding FSMs in VHDL

4.1 VHDL coding styles

Next state logic is best modeled using the **case** statement, though a "selected signal assignment" can also be used, but means the FSM cannot be modeled in the same process. The **others** clause must always be used in a **case** statement for Language Reference Manual (LRM) compliance, even though all branch conditions may have already been explicitly defined. This also avoids the need to explicitly define all 2^n register values that are not used in the state machine.

Example 1. Four VHDL Models of the same FSM

A FSM whose state diagram is shown in Figure 6, is shown modeled in the following four ways:

FSM1A Separate CS, NS and OL
FSM1B Combined CS and NS. Separate OL
FSM1C Combined NS and OL. Separate CS
FSM1D Combined CS, NS and OL
where CS - Current State
NS - Next State
OL - Output Logic

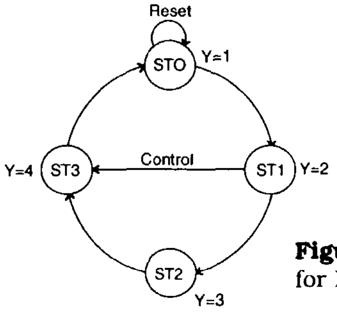

Figure 6. State diagram for FSM1A/B/C/D

FSM1A Seperate CS, NS and OL

```
entity FSM1A is
  port ( Clock,Reset:   in bit;
         Control:       in bit;
         Y:             out integer range 0 to 4);
end FSM1A;

architecture RTL of FSM1A is
  type StateType is (ST0,ST1,ST2,ST3);
  signal CurrentState, NextState: StateType;
begin
  COMB: process (Control, CurrentState)
  begin
     case CurrentState is
        when ST0 =>
           NextState <= ST1;
        when ST1 =>
           if (CONTROL = '1') then
              NextState <= ST2;
           else
              NextState <= ST3;
           end if;
        when ST2 =>
           NextState <= ST3;
        when ST3 =>
           NextState <= ST0;
        when others =>
           NextState <= ST0;
     end case;
  end process;
```

next state logic

```
  SEQ: process (Clock,Reset,NextState)
  begin
     if (Reset = '1') then
        CurrentState <= ST0;
     elsif (Clock'event and Clock='1') then
        CurrentState <= NextState;
     end if;
  end process;
```

current state logic

```
  -- Moore output logic
  -- (Concurrent select signal assignment)
  with CurrentState select
     Y <=   1 when ST0,
            2 when ST1,
            3 when ST2,
            4 when ST3,
            1 when others;
end RTL;
```

output logic

FSM1B - Combined CS and NS. Separate OL

```
entity FSM1B is
  port ( Clock, Reset: in  bit;
     Control:       in  bit;
     Y:             out integer range 0 to 4);
end FSM1B;

architecture RTL of FSM1B is
  type StateType is (ST0, ST1, ST2, ST3);
  signal STATE: StateType;
begin
   NEXT_CURR: process
   begin
     if (Reset = '1') then
        STATE <= ST0;
     elsif (Clock'event and Clock='1') then
        case (STATE) is
           when ST0 =>
              STATE <= ST1;
           when ST1 =>
              if (Control = '1') then
                 STATE <= ST2;
              else
                 STATE <= ST3;
              end if;
           when ST2 =>
              STATE <= ST3;
           when ST3 =>
              STATE <= ST0;
           when others =>
              null;
        end case;
     end if;
   end process;
```

current state and next state logic

```
   with STATE select
        Y <=    1 when ST0,
                2 when ST1,
                3 when ST2,
                4 when ST3,
                1 when others;
end RTL;
```

output logic

FSM1C - Combined NS and OL. Separate CS

```
entity FSM1C is
  port ( Clock,Reset:    in  bit;
         Control:        in  bit;
         Y:              out integer range 0 to 4);
end FSM1C;

architecture RTL of FSM1C is
  type StateType is (ST0,ST1,ST2,ST3);
  signal CurrentState, NextState: StateType;
begin

  COMB: process (Control, CurrentState)
  begin
     case CurrentState is
        when ST0 =>
           Y <= 1;
           NextState <= ST1;
        when ST1 =>
           Y <= 2;
           if (CONTROL = '1') then
              NextState <= ST2;
           else
              NextState <= ST3;
           end if;
        when ST2 =>
           Y <= 3;
           NextState <= ST3;
        when ST3 =>
           Y <= 4;
           NextState <= ST0;
        when others =>
           Y <= 4;
           NextState <= ST0;
     end case;
  end process;
```

next state and output logic

```
  SEQ: process (Clock,Reset)
  begin
     if (Reset = '1') then
        CurrentState <= ST0;
     elsif (Clock'event and Clock='1') then
        CurrentState <= NextState;
     end if;
  end process;
end RTL;
```

current state logic

FSM1D - Combined CS, NS and OL

```
entity FSM1D is
  port ( Clock,Reset:    in bit;
         Control:in  bit;
         Y:              out integer range 0 to 4);
end FSM1D;

architecture RTL of FSM1D is
  type StateType is (ST0,ST1,ST2,ST3);
  signal State: StateType;
begin
  ALL_IN_1: process (Clock,Reset)
  begin
     if (Reset = '1') then
        State <= ST0;
     elsif (Clock'event and Clock='1') then
        case State is
            when ST0 =>
               State <= ST1;
            when ST1 =>
               if (CONTROL = '1') then
                  State <= ST2;
               else
                  State <= ST3;
               end if;
            when ST2 =>
               State <= ST3;
            when ST3 =>
               State <= ST0;
            when others =>
               State <= ST0;
        end case;
     end if;
     case State is
        when ST0 =>    Y <= 1;
        when ST1 =>    Y <= 2;
        when ST2 =>    Y <= 3;
        when ST3 =>    Y <= 4;
        when others =>Y <= 1;
     end case;
  end process;
end RTL;
```

current state, next State and output logic

4.2 Modeling resets

The modeling styles for resets is fairly common for all commercial synthesis tools. An asynchronous reset can only be modeled using the **if** statement, while a synchronous reset can be modeled using either a **wait** or **if** statement. If the current and next state logic are modeled separately, an asynchronous reset must be included in the sequential current state logic while a synchronous reset may be included with either the current or next state logic. Clearly by always including a reset in the current state logic it is easy to change it from an asynchronous to synchronous reset or vice versa if needed. Example code used for resets in FSM models follow.

Asynchronous reset

```
CURR_NEXT_LOGIC: process (Clock, AsyncReset)
begin
   if (AsyncReset = '1') then
      Y <= ST1;
   elsif (Clock'event and Clock = '1') then
      case (Y) is
         when ST1 => Y <= ST2;
         when ...
```

Synchronous reset

```
CURR_NEXT_LOGIC: process (Clock, SyncReset)
begin
   if (Clock'event and Clock = '1') then
      if (SyncReset = '1') then
         Y <= ST1;
         else
            case (Y) is
               when ST1 => Y <= ST2;
               when ...

CURR_NEXT_LOGIC: process (Clock, SyncReset)
begin
   wait until (Clock'event and Clock = '1');
      if (SyncReset = '1') then
         Y <= ST1;
         else
            case (Y) is
               when ST1 => Y <= ST2;
               when ...
```

4.3 Defining the state encoding

The assignment of binary numbers to the state, called the state encoding, can be achieved in four ways.

1. Using an enumerated data type for the current and next state signals, the synthesis tool will automatically assign binary numbers to the states; normally in increasing order. e.g.

```
type StateType is (ST0, ST1, ST2, ST3, ST4);
signal CurrentState, NextState: StateType;
```

The state assignments defined by the synthesis tool will therefore be:

```
ST0 = 000
ST1 = 001
ST2 = 010
ST3 = 011
ST4 = 100
```

2. An enumerated data type may be used in the same way as in 1 above, but this time a synthesis specific attribute is used to explicitly define the desired state encoding e.g.

```
attribute ENUM_TYPE_ENCODING: string; -- (1)
type StateType is (ST0, ST1, ST2, ST3, ST4); -- (2)
attribute ENUM_TYPE_ENCODING of StateType : type is -- (3)
   "011 010 000 100 110";
signal CurrentState, NextState: StateType; -- (4)
```

(1). ENUM_TYPE_ENCODING is the name of the attribute known by the synthesis tool and which is used specifically for this purpose. The name may be different for different synthesis tools and must be defined to be of type string.

(2). The enumerated data type, StateType, is defined to have five values. (ST0-ST4)

(3). The attribute attributes the five values to the five states e.g. ST0=011, ST1=010 etc.

(4). The CurrentState and NextState data types are defined to be of type StateType.

3. Constants can be defined and used to represent state values. Unlike using the synthesis specific attribute, models can be synthesized using different synthesis tools, e.g.

```
constant ST0: bit_vector(2 downto 0) := "011";
constant ST1: bit_vector(2 downto 0) := "010";
constant ST2: bit_vector(2 downto 0) := "000";
constant ST3: bit_vector(2 downto 0) := "100";
constant ST4: bit_vector(2 downto 0) := "110";
```

4. It may be desirable to allow the synthesis tool to choose an optimal state encoding format that minimizes next state logic. If so, consult the appropriate synthesis manual on how it is achieved. The synthesis tools provided by Intergraph allow a panel entry of FSM parameters from which it defines the state encoding for an optimal netlist and also provides a VHDL model for simulation purposes.

4.4 Modeling Mealy or Moore type outputs

The VHDL model of the example state diagram of Figure 2 is shown. There is one Mealy type output, Y_ME, and one Moore type output, Y_Mo. The Moore type output is seen to be dependent upon the state value only, while the Mealy type output is dependent upon the state value and the two inputs A and Hold. Because the Mealy output is dependent upon inputs, it must be modeled in a section of code that infers only combinational logic.

```
entity FSM_MEALY_MOORE is
  port (Clock, Reset:  in bit;
        A, Hold:       in bit;
        Y_Me, Y_Mo:    out bit);
end FSM_MEALY_MOORE;

architecture RTL of FSM_MEALY_MOORE is
  type StateType is (ST0, ST1, ST2, ST3, ST4, ST5);
  signal CurrentState, NextState: StateType;
begin

  FSM: process (Clock, A, Hold)
  begin
     case CurrentState is
        when ST0 =>                               Moore
           Y_Mo <= '0';                           type
           if (A = '1') then                      output
              Y_Me <= '1';                        Mealy type
              NextState <= ST0;                   output
           else                                   dependent
              Y_Me <= '0';                        upon A
              NextState <= ST1;
           end if;
        when ST1 =>
           Y_Mo <= '1';
           if (A = '1') then
              Y_Me <= '1';
              NextState <= ST2;
           else
              Y_Me <= '0';
              NextState <= ST0;
           end if;
        when ST2 =>
           Y_Mo <= '0';
           Y_Me <= '0';
           NextState <= ST3;
        when ST3 =>
           Y_Mo <= '1';
           if (Hold = '1') then
              Y_Me <= '0';
              NextState <= ST3;
           else
              Y_Me <= '1';
              if (A = '1') then
                 NextState <= ST4;
              else
                 NextState <= ST0;
              end if;
           end if;
        when ST4 =>
           Y_Mo <= '1';
           Y_Me <= '0';
           NextState <= ST3;
        when others =>
           Y_Mo <= '0';
           Y_Me <= '1';
                 NextState <= ST0;
     end case;

     if (Clock'event and Clock='1') then
        if (Reset = '1') then
           CurrentState <= STO;
        else
           CurrentState <= NextState;
        end if;
     end if;
  end process;
end;
```

4.5 Sequential next state or output logic

Two examples are shown, one with extra synchronous next state logic and one with extra synchronous output logic. (By not defining next state or output signals in all branches of a state machine's **case** statement, it is easy to inadvertently model extra unwanted sequential logic in the form of latches.)

Example 2. FSM with sequential next state logic

A state machine with an extra flip-flop in the next state logic is shown. The state diagram is shown in Figure 7 and the implied architecture from the model is shown in Figure 8.

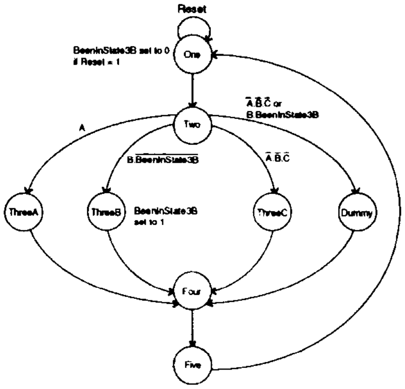

Figure 7. State diagram implying sequential next state logic

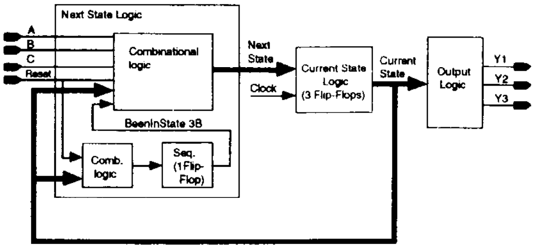

Figure 8. Architecture of FSM with extra embedded sequential next state logic

The three inputs A, B and C each cause the state machine to branch to states ThreeA, ThreeB and ThreeC respectively on a priority encoded basis as it passes around the loop of five states. The synchronous reset is guaranteed to be high for at least five clock cycles thus ensuring the state machine starts in state One. After a reset, the output Y1 is high for one clock cycle every five clock cycles while A remains high, likewise for input C and corresponding output Y3. However, when B goes high, its corresponding output Y2 goes high only once. The reason for this is that when the state machine is in state ThreeB the signal BeenInState3B is set high from an additional flip-flop in the next state logic and which is used to inhibit the state machine from entering state ThreeB again until after a reset occurs on Reset.

```
entity FSM_SEQ_NEXT is
  port ( Clock, Reset:    in  bit;
         A, B, C:         in  bit;
         Y1, Y2, Y3:      outbit);
end FSM_SEQ_NEXT;

architecture RTL of FSM_SEQ_NEXT is
  type   StateType is (One, Two, ThreeA, ThreeB, ThreeC,
                       Dummy, Four, Five);
  signal CurrentState: StateType;
  signal BeenInState3B: bit;
begin
  FSM_1: process (Clock)
  begin
    if (Clock'event and Clock='1') then
      case (CurrentState) is
        when One =>                              BeenInState3B
          if (Reset = '1') then                  set to 0
            BeenInState3B <= '0';
            CurrentState <= One;
          else
            CurrentState <= Two;
          end if;
        when Two =>
          if (A = '1') then
            CurrentState <= ThreeA;
          elsif (B = '1') then
            if (BeenInState3B = '1') then
              CurrentState <= Dummy;
            else
              CurrentState <= ThreeB;
            end if;
          elsif (C = '1') then
            CurrentState <= ThreeC;
          else
            CurrentState <= Dummy;
          end if;
        when ThreeA =>
          CurrentState <= Four;
        when ThreeB =>
          BeenInState3B <= '1';
          CurrentState <= Four;                  BeenInState3B
        when ThreeC =>                           set to 1
          CurrentState <= Four;
        when Dummy =>
          CurrentState <= Four;
        when Four =>
          CurrentState <= Five;
        when Five =>
          CurrentState <= One;
        when others =>
          CurrentState <= One;
      end case;
    end if;
    Y1 <= '0';
    Y2 <= '0';
    Y3 <= '0';
    case (CurrentState) is
      when ThreeA =>
        Y1 <= '1';
      when ThreeB =>
        Y2 <= '1';
      when ThreeC =>
        Y3 <= '1';
      when others =>
        Y1 <= '0';
        Y2 <= '0';
        Y3 <= '0';
    end case;
  end process FSM_1;
end RTL;
```

Example 3. FSM with sequential output logic

The model of a FSM within an embedded counter is shown. The counter forms part of the FSM's output logic as shown in the implied architecture of Figure 9. The state diagram of the FSM is shown in Figure 10 .

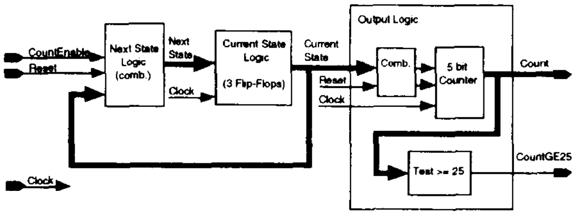

Figure 9. Architecture of FSM with extra embedded sequential output logic

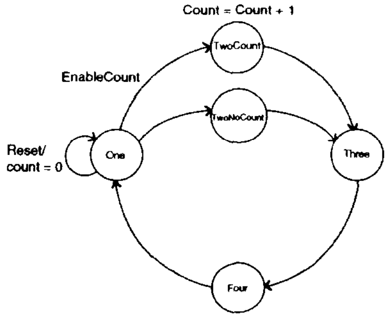

Figure 10. State diagram implying sequential output logic

After a reset the state machine is set to state One and the counter in the output logic is set to zero. After the reset, the state machine cycles around a loop of four states. A two way state branch allows one of two paths for the second stage of the loop and is represented by states TwoCount and TwoNoCount. When input EnableCount is high the state TwoCount is used, otherwise TwoNoCount is used. Therefore while

EnableCount is high the counter is incremented once every four clock cycles. The counters' value plus an indication of whether it is greater than or equal to 25 are output from the model.

```
entity FSM_SEQ_OUT is
  port (  Clock, Reset:     in      bit;
          EnableCount:      in      bit;
          CountGE25:        out     bit;
          Count:            buffer integer range 0 to 31);
end FSM_SEQ_OUT;

architecture RTL of FSM_SEQ_OUT is
  type StateType is (  One, TwoCount, TwoNoCount, Three,
                       Four);
  signal CurrentState: StateType;
begin
  FSM_ACC: process (Clock)
  begin
    if (Clock'event and Clock='1') then
      case CurrentState is
        when One =>
          if (Reset = '1') then
            Count <= 0;
            CurrentState <= One;
          elsif (EnableCount = '1') then
            CurrentState <= TwoCount;
          else
            CurrentState <= TwoNoCount;
          end if;
        when TwoCount =>                    counter
          Count <= Count + 1;  ----------   embedded in
          CurrentState <= Three;            FSM model
        when TwoNoCount =>
          CurrentState <= Three;
        when Three =>
          CurrentState <= Four;
        when Four =>
          CurrentState <= One;
        when others =>
          null;
      end case;
    end if;
    if (Count >= 25) then
      CountGE25 <= '1';
    else
      CountGE25 <= '0';
    end if;
  end process FSM_ACC;
end RTL;
```

4.6 Interactive FSMs

The modeling techniques for interactive FSMs are the same as individual FSMs. For clarity, it is usually better to use more code and keep the code for models separate rather than combining them. It is the designers choice what coding method to use. Either way, the outputs from one FSM are used for the inputs to other FSMs.

5. Conclusions

The different checklist issues to be considered when designing and coding FSM models in VHDL have been discussed i.e. VHDL coding style, resets and fail safe behavior, state encoding, Mealy or Moore type outputs, additional sequential next state or output logic, and interactive FSMs.

The coding style is shown to be independent of the FSM being modeled. If there is no reset, or only a synchronous reset is available all 2^n (n = no. of state register flip-flops) binary values must be decoded within the next state logic whether they form part of the state machine or not. There is generally only a small area overhead in doing this. Seven types of state encoding and four methods of modeling the state encoding have been shown; the choice being design dependent. If area is of critical concern, letting the synthesis tool choose an optimal state encoding is the preferred option. Mealy and Moore type outputs have been discussed indicating that the choice between which to use is also design dependent. Also shown, is additional sequential logic embedded within the code of an FSM. Although this sequential logic does not form part of the standard FSM it does control state transitions based on; a previously transitioned state, a sequence of previously transitioned states, or some accumulated value resulting from looping around successive sequences of states. Interactive FSMs have also been discussed indicating that it is often better to keep the code for different FSMs separate for clarity.