VHDL Modeling for Synthesis

Finite State Machines and Controllers
Modeling Finite State Machines (Synchronous Sequential Circuits)

- FSM design & synthesis process:
  1. Design state diagram (behavior)
  2. Derive state table
  3. Reduce state table
  4. Choose a state assignment
  5. Derive output equations
  6. Derive flip-flop excitation equations

- Synthesis steps 2-6 can be automated, given the state diagram
Synchronous Sequential Circuit Model

Mealy Outputs $z = f(x,y)$,  Moore Outputs $z = f(y)$

Next State $Y = f(x,y)$
Synthesizing Finite state machines

- Model states as enumerated type
- Model state and next_state
- One process updates state with next_state
- One process updates next_state
- Allow synthesis tool to encode states (binary, one-hot, random, gray code, etc.)
- Consider how initial state will be forced
State machine synthesis issues

- Mealy model: outputs = f (inputs, state)
- Moore model: outputs = f (state)
- “case” more efficient than “if-then-elsif…” to test present_state (latter builds a priority encoder)
- “others” in case statement could also generate extra logic (must determine if not one of the other states)
- Assign outputs & next_state for every condition. Any signal not assigned anything should retain its value. If an output or next_state is not assigned something under a certain condition in the case statement, the synthesis tools will have to preserve the value with extra latches/flip-flops.
- Left-most value of enumeration type is default simulation starting value (use reset to initialize real circuit)
Synchronous Sequential Circuit (FSM) Example

<table>
<thead>
<tr>
<th>Present state</th>
<th>Input $x$</th>
<th>Next state/output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>0</td>
<td>A/0</td>
</tr>
<tr>
<td>$B$</td>
<td>0</td>
<td>A/0</td>
</tr>
<tr>
<td>$C$</td>
<td>0</td>
<td>C/0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>B/0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C/1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A/1</td>
</tr>
</tbody>
</table>

## Diagram

- **X/Z:**
  - From the current state (X/Z) to the next state (A), the transition is 0/0.
  - From X/Z to B, the transition is 1/1.
  - From X/Z to C, the transition is 0/0.

- **A:**
  - From A to A, the transition is 0/0.
  - From A to B, the transition is 1/0.
  - From A to C, the transition is 1/1.

- **B:**
  - From B to A, the transition is 1/1.
  - From B to B, the transition is 1/0.
  - From B to C, the transition is 1/1.

- **C:**
  - From C to A, the transition is 0/0.
  - From C to B, the transition is 1/1.
  - From C to C, the transition is 0/0.
FSM Example – entity definition

entity seqckt is

port (  
    x: in bit;  -- FSM input  
    z: out bit; -- FSM output  
    clk: in bit ); -- clock

div end seqckt;
FSM Example - behavioral model

architecture behave of seqckt is
    type states is (A,B,C);  -- symbolic state names (enumerate)
    signal curr_state,next_state: states;
begin
    -- Model the memory elements of the FSM
    process (clk)
    begin
        if (clk’event and clk=‘1’) then
            pres_state <= next_state;
        end if;
    end process;
(continue on next slide)
FSM Example - continued

-- Model next-state and output functions of the FSM
process (x, pres_state) -- function inputs
begin

  case pres_state is -- describe each state
    when A => if (x = '0') then
      z <= '0';
      next_state <= A;
    else -- if (x = '1')
      z <= '0';
      next_state <= B;
    end if;

(continue next slide for pres_state = B and C)
FSM Example (continued)

when B => if (x='0') then
    z <= '0';
    next_state <= A;
else
    z <= '1';
    next_state <= C;
end if;
when C => if (x='0') then
    z <= '0';
    next_state <= C;
else
    z <= '1';
    next_state <= A;
end if;
end case;
end process;
Alternative Format for Output and Next State Functions

-- Output function
\[ z \leq '1' \text{ when } ((\text{curr\_state} = B) \text{ and } (x = '1')) \]
\[ \quad \text{or } ((\text{curr\_state} = C) \text{ and } (x = '1')) \]
\[ \quad \text{else } '0'; \]

-- Next state function
\[ \text{next\_state} \leq A \text{ when } ((\text{curr\_state} = A) \text{ and } (x = '0')) \]
\[ \quad \text{or } ((\text{curr\_state} = B) \text{ and } (x = '0')) \]
\[ \quad \text{or } ((\text{curr\_state} = C) \text{ and } (x = '1')) \text{ else } \]
\[ \quad B \text{ when } ((\text{curr\_state} = 1) \text{ and } (x = '1')) \text{ else } \]
\[ \quad C; \]
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity SM1 is
  port (aIn, clk : in Std_logic; yOut: out Std_logic);
end SM1;

architecture Moore of SM1 is
  type state is (s1, s2, s3, s4);
  signal pS, nS : state;
begin
  process (aIn, pS) begin – next state and output functions
    case pS is
      when s1 => yOut <= '0'; nS <= s4;  --Moore: yOut = f(pS)
      when s2 => yOut <= '1'; nS <= s3;
      when s3 => yOut <= '1'; nS <= s1;
      when s4 => yOut <= '1'; nS <= s2;
    end case;
  end process;

  process begin
    wait until clk = '1';
    pS <= nS;  -- update state variable on next clock
  end process; -- update state variable on next clock

end Moore;
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity SM2 is port (aIn, clk : in Std_logic; yOut: out Std_logic); end SM2;
arithmetic Mealy of SM2 is
  type state is (s1, s2, s3, s4);
signal pS, nS : state;
begin
  process(aIn, pS) begin  -- Mealy: yOut & nS are functions of aIn and pS
    case pS is
      when s1 => if (aIn = '1') then yOut <= '0'; nS <= s4;
        else yOut <= '1'; nS <= s3;  end if;
      when s2 => yOut <= '1'; nS <= s3;
      when s3 => yOut <= '1'; nS <= s1;
      when s4 => if (aIn = '1') then yOut <= '1'; nS <= s2;
        else yOut <= '0'; nS <= s1;  end if;
    end case; end process;
    process begin
      wait until clk = '1' ;
      pS <= nS;
    end process;
  end Mealy;
when s1 =>  -- initiate row access
    ras <= '0'; cas <= '1'; ready <= '0';
    next_state <= s2;
when s2 =>  -- initiate column access
    ras <= '0'; cas <= '0'; ready <= '0';
if (cs = '0') then
    next_state <= s0;  -- end of operation if cs = 0
else
    next_state <= s2;  -- wait in s2 for cs = 0
end if;
when s3 =>  -- start cas-before-ras refresh
    ras <= '1'; cas <= '0'; ready <= '0';
    next_state <= s4;
when s4 =>  -- complete cas-before-ras refresh
    ras <= '0'; cas <= '0'; ready <= '0';
    next_state <= s0;
end case;
end process;
end rtl;