VHDL 3 – Sequential Logic Circuits

Reference: Roth/John Text: Chapter 2
VHDL “Process” Construct

[labell:]  process (sensitivity list)
   declarations
   begin
   sequential statements
   end process;

- Process statements are executed in sequence
- Process statements are executed once at start of simulation
- Process halts at “end” until an event occurs on a signal in the “sensitivity list”
- Allows conventional programming language methods to describe circuit behavior
-- Edge-triggered flip flop/register

entity DFF is
  port (D,CLK: in bit;
       Q: out bit);
end DFF;
architecture behave of DFF is
begin
  process(clk) -- “process sensitivity list”
  begin
    if (clk’event and clk='1') then
      Q <= D after 1 ns;
    end if;
  end process;
end;

- Process statements executed sequentially (sequential statements)
- clk’event is an attribute of signal clk which is TRUE if an event has occurred on clk at the current simulation time
Edge-triggered flip-flop

Alternative methods for specifying clock transition

process (clk)
begin
  if rising_edge(clk) then  -- std_logic_1164 function
    Q <= D ;
  end if;
end process;

Synthesis tool may also recognize: not clk’stable
as equivalent to: clk’event
Alternative to sensitivity list

process  -- no “sensitivity list”
begin
    wait on clk; -- suspend process until event on clk
    if (clk='1') then
        Q <= D after 1 ns;
    end if;
end process;

- Other “wait” formats:  wait until (clk’event and clk='1')
                        wait for 20 ns;
- This format does not allow for asynchronous controls
- Process executes endlessly if no sensitivity list or wait statement!
Level-Sensitive D latch vs. D flip-flop

entity Dlatch is
  port (D, CLK: in bit;
       Q: out bit);
end Dlatch;

architecture behave of Dlatch is begin
  process (D, clk)
  begin
    if (clk='1') then
      Q <= D after 1 ns;
    end if;
  end process;
end;

Latch, Q changes whenever the latch is enabled by CLK='1' (rather than edge-triggered)
entity Reg8 is
  port (D: in bit_vector(0 to 7);
       Q: out bit_vector(0 to 7);
       LD: in bit);
end Reg8;
architecture behave of Reg8 is begin
  process(LD)
  begin
    if (LD’event and LD=‘1’) then
      Q <= D after 1 ns;
    end if;
  end process;
end;

D and Q can be any abstract data type
Basic format for synchronous and asynchronous inputs

process (clock, asynchronous_signals )
begin
    if (boolean_expression) then
        asynchronous signal_assignments
    elsif (boolean_expression) then
        asynchronous signal assignments
    elsif (clock'event and clock = constant) then
        synchronous signal_assignments
    end if;
end process;
Synchronous vs. Asynchronous Flip-Flop Inputs

entity DFF is
  port (D,CLK: in bit;
       PRE,CLR: in bit;
       Q: out bit);
end DFF;
architecture behave of DFF is
begin
  process(clk,PRE,CLR)
  begin
    if (CLR='0') then -- async CLR has precedence
      Q <= '0' after 1 ns;
    elsif (PRE='0') then  -- then async PRE has precedence
      Q <= '1' after 1 ns;
    elsif (clk'event and clk='1') then
      Q <= D after 1 ns; -- sync operation only if CLR=PRE='1'
    end if;
  end process;
end;
Synchronous Sequential Circuit Model

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

Next State $Y = f(x,y)$
Synchronous Sequential Circuit (FSM) Example – Mealy model

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

Present state

Input x

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

Next state/output

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

entity seqckt is
    port ( x: in bit; -- FSM input
                  z: out bit; -- FSM output
                  clk: in bit ); -- clock
end seqckt;
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)
-- 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 on 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;
FSM – omitting use of \textit{next\_state}

process (clk) – trigger state change on rising edge of clock
begin
  if (clk’event and clk = ‘1’) then  --detect rising clock edge
    case pres\_state is  -- change \textit{pres\_state} according to \textit{x}
      when A => if (x = ‘0’) then
        pres\_state <= A;
      else  -- if (x = ‘1’)
        pres\_state <= B;
      end if;
      when B => if (x = ‘0’) then
        pres\_state <= A;
      else
        pres\_state <= C;
      end if;
      Continue for other states
    end case;
  end if;
end process;
Alternative form for output and next state functions

-- Output function (combinational logic)
\[
z \leq '1' \text{ when } ((\text{curr\_state} = B) \text{ and } (x = '1')) \quad \text{--all conditions}
\]
\[
\quad \text{or } ((\text{curr\_state} = C) \text{ and } (x = '1')) \quad \text{--for which } z = 1.
\]
\[
\quad \text{else '0';} \quad \text{--otherwise } z = 0
\]

-- Next state function (combinational logic)
\[
\text{next\_state } \leq A \text{ when } ((\text{curr\_state} = A) \text{ and } (x = '0'))
\]
\[
\quad \text{or } ((\text{curr\_state} = B) \text{ and } (x = '0'))
\]
\[
\quad \text{or } ((\text{curr\_state} = C) \text{ and } (x = '1')) \text{ else}
\]
\[
\quad \text{B when } ((\text{curr\_state} = 1) \text{ and } (x = '1')) \text{ else}
\]
\[
\quad C;
\]
entity FSM is
   port (CLK, EN, TDI: in bit;
       RST, SHIFT: out bit);
end entity FSM;
architecture RTL of FSM is
    type STATES is (Reset, BIST, Result, NOP);  -- abstract state names
    signal CS: STATES; -- current state
begin
SYNC: process (CLK) begin -- change states on falling edge of CLK
    if (CLK’event and CLK=’0’) then
        if (EN = ‘1’) then -- change only if EN = 1
            if (CS = Reset) then
                if (TDI=’0’) then CS <= BIST; end if; --EN,TDI = 10
            elsif (CS = BIST) then
                if (TDI=’1’) then CS <= Result; end if; --EN,TDI = 11
            elsif (CS = Result) then
                if (TDI=’1’) then CS <= NOP; end if; --EN,TDI = 11
            elsif (CS = NOP) then
                if (TDI=’0’) then CS <= BIST; else CS <= Reset; end if; --EN,TDI = 11
            end if;
        end if;
    end if;
end if; end if; end process SYNC;

(Outputs on next slide)
Moore model outputs

-- Outputs = functions of the state
COMMAND: process (CS) begin
    if (CS = Reset) then
        RST <= '1';
        SHIFT <= '0';
    elsif (CS = Result) then
        RST <= '0';
        SHIFT <= '1';
    else
        RST <= '0';
        SHIFT <= '0';
    end if;
end process COMMAND;
end architecture RTL;

-- more compact form
RST <= '1' when CS = Reset else '0';
SHIFT <= '1' when CS = Result else '0';
end architecture RTL;

Note that Moore model outputs are independent of current inputs.
Sequential Constructs: *if-then-else*

**General format:**

if (*condition*) then  
    *do stuff*  
elsif (*condition*) then  
    *do more stuff*  
else  
    *do other stuff*  
end if;

**Example:**

if (*S* = “00”) then  
    *Z* <= *A*;
elsif (*S* = “11”) then  
    *Z* <= *B*;
else  
    *Z* <= *C*;
end if;

*elsif* and *else* clauses are optional, BUT incompletely specified if-then-else (no else) implies memory element
Sequential Constructs: *case-when*

**General format:**

case `expression` is
  when `value` =>
    `do stuff`
  when `value` =>
    `do more stuff`
  when others =>
    `do other stuff`
end case;

**Example:**

case S is
  when “00” =>
    `Z <= A;`
  when “11” =>
    `Z <= B;`
  when others =>
    `Z <= C;`
end case;
Sequential Constructs: *for loop*

General format:

```
[label:] for identifier in range loop
    do a bunch of junk
end loop [label];
```

Example:

```
init: for k in N-1 downto 0 loop
    Q(k) <= '0';
end loop init;
```

Note: variable k is “implied” in the for-loop and does not need to be declared
Sequential Constructs: *while loop*

**General format:**

```
[label:] while condition loop
   do some stuff
end loop [label];
```

**Example:**

```
init: while (k > 0) loop
  Q(k) <= '0'
  k := k - 1;
end loop init;
```

**Note:** Variable k must be declared as variable in process, between *sensitivity list* and *begin*, with format:

```
variable k: integer := N-1;
```