VHDL 3 – Sequential Logic Circuits

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

- Allows conventional programming language structures to describe circuit behavior – especially sequential behavior
  - Process statements are executed in sequence
  - Process statements are executed once at start of simulation
  - Process is suspended at “end process” until an event occurs on a signal in the “sensitivity list”

```vhdl
label: process (sensitivity list)  
  declarations
  begin
    sequential statements
  end process;
```
Modeling combinational logic as a process

-- All signals referenced in process must be in the sensitivity list.

entity And_Good is
  port (a, b: in std_logic; c: out std_logic);
end And_Good;

architecture Synthesis_Good of And_Good is
  begin
    process (a,b)  -- gate sensitive to events on signals a and/or b
    begin
      c <= a and b; -- c updated (after delay on a or b “events”
    end process;
  end;

  -- Above process is equivalent to simple signal assignment statement:
  -- c <= a and b;
Bad example of combinational logic

-- This example produces unexpected results.
entity And_Bad is
    port (a, b: in std_logic; c: out std_logic);
end And_Bad;
architecture Synthesis_Bad of And_Bad is
begin
    process (a) -- sensitivity list should be (a, b)
    begin
        c <= a and b; -- will not react to changes in b
    end process;
end Synthesis_Bad;
-- synthesis may generate a flip flop, triggered by signal a
-- 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    -- rising edge of clk
                Q <= D; -- optional "after x" for delay
                QB <= not D;
            end if;
        end process;
end;

- *clk'event* is an “attribute” of signal *clk*  (*signals have several attributes*)
- *clk'event* = TRUE if an event has occurred on *clk* at the current simulation time
  and FALSE if no event on *clk* at the current simulation time
- *clk'stable* is a complementary attribute (TRUE of no event at this time)
Edge-triggered flip-flop

- Special functions in package `std_logic_1164` for `std_logic` types
  - `rising_edge(clk) = TRUE` for 0->1, L->H and several other “rising-edge” conditions
  - `falling_edge(clk) = TRUE` for 1->0, H->L and several other “falling-edge” conditions

**Example:**

```vhdl
signal clk: std_logic;
begin
  process (clk)                       -- trigger process on clk event
  begin
    if rising_edge(clk) then    -- detect rising edge of clk
      Q   <= D ;                -- Q and QB change on rising edge
      QB <= not D;
    end if;
  end process;
```
Common error in processes

- Process statements are evaluated only at time instant $T$, at which an event occurs on a signal in the sensitivity list
  - Statements in the process use signal values that exist at time $T$.
  - Signal assignment statements “schedule” future events.

**Example:**

```vhdl
process (clk)                     -- trigger process on clk event
begin
  if rising_edge(clk) then      -- detect rising edge of clk
    Q   <= D ;                 -- Q and QB change $\delta$ time after rising edge
    QB <= not Q;               -- Timing error here!!
  end if;
end process;                    -- Desired QB appears one clock period late!
```

As written above, if clk edge occurs at time $T$:
- $Q$ will change at time $T+\delta$, to $D(T)$
- $QB$ will change at time $T+\delta$, to “not $Q(T)$” – using $Q(T)$ rather than new $Q(T+\delta)$
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;

- BUT - sensitivity list is preferred for sequential circuits!

- Other “wait” formats:  wait until (clk’event and clk='1')
  wait for 20 ns;

- This format does not allow for asynchronous controls
- Cannot have both sensitivity list and wait statement
- Process executes endlessly if neither sensitivity list nor wait statement provided!
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;

Qlatch can change when CLK becomes ‘1’ and/or when D changes while CLK=‘1’ (rather than changing only at a clock edge)
RTL “register” model (not gate-level)

entity Reg8 is
  port (D: in std_logic_vector(0 to 7);
       Q: out std_logic_vector(0 to 7);
       LD: in std_logic);
end Reg8;
architecture behave of Reg8 is
begin
  process (LD)
  begin
    if rising_edge(LD) then
      Q <= D;
    end if;
  end process;
end;

D and Q can be any abstract data type
RTL “register” with clock enable

--Connect all system registers to a common clock
--Select specific registers to be loaded

entity RegCE is
  port (D: in std_logic_vector(0 to 7);
        Q: out std_logic_vector(0 to 7);
        EN: in std_logic; --clock enable
        CLK: in std_logic);
end RegCE;

architecture behave of RegCE is
begin
  process(CLK)
  begin
    if rising_edge(CLK) then
      if EN = '1' then
        Q <= D; --load only if EN=1 at the clock transition
      end if;
    end if;
  end process;
end;
Synchronous vs asynchronous inputs

- **Synchronous** inputs are synchronized to the clock.
- **Asynchronous** inputs are not, and cause immediate change.
  - Asynchronous inputs normally have precedence over sync. inputs

```vhdl
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 std_logic; --D is a sync input
          PRE,CLR: in std_logic; --PRE/CLR are async inputs
          Q: out std_logic);
end DFF;

architecture behave of DFF is
begin
    process(clk,PRE,CLR)
    begin
        if (CLR='0') then -- async CLR has precedence
            Q <= '0';
        elsif (PRE='0') then  -- then async PRE has precedence
            Q <= '1';
        elsif rising_edge(clk) then   -- sync operation only if CLR=PRE='1'
            Q <= D;
        end if;
    end process;
end;

What happens if CLR = PRE = 0 ??
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:**

\[\text{case expression is}\]

\[\begin{align*}
\text{when value =>} & \quad \text{do stuff} \\
\text{when value =>} & \quad \text{do more stuff} \\
\text{when others =>} & \quad \text{do other stuff}
\end{align*}\]

**Example:**

\[\text{case S is}\]

\[\begin{align*}
\text{when “00” =>} & \quad Z <= A; \\
\text{when “11” =>} & \quad Z <= B; \\
\text{when others =>} & \quad Z <= C;
\end{align*}\]

**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 a process “variable”, between *sensitivity list* and *begin*, with format:

```
variable k: integer := N-1;
```
Modeling Finite State Machines (FSMs)

- “Manual” 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

- Steps 2-6 can be automated, given a state diagram
  - Model states as enumerated type
  - Model output function (Mealy or Moore model)
  - Model state transitions (functions of current state and inputs)
  - Consider how initial state will be forced
FSM structure

Inputs $x$ → Comb. Logic → Outputs $z$

Present State $y$ → FFs → Next State $Y$

Clock

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

Next State $Y = f(x,y)$
entity seqckt is
  port (  x: in   std_logic; -- FSM input
          z: out std_logic; -- FSM output
          clk: in std_logic ); -- clock
end seqckt;
architecture behave of seqckt is
    type states is (A,B,C);  -- symbolic state names (enumerate)
    signal state: states;    -- state variable
begin

    -- Output function (combinational logic)
    z <= '1' when ((state = B) and (x = '1'))     -- all conditions
    or ((state = C) and (x = '1'))    -- for which z=1.
    else '0';                                           -- otherwise z=0

    -- State transitions on next slide
FSM example – state transitions

process (clk) – trigger state change on clock transition
begin
  if rising_edge(clk) then
    case state is
      when A => if (x = '0') then
        state <= A;
      else
        state <= B;
      end if;
      when B => if (x='0') then
        state <= A;
      else
        state <= C;
      end if;
      when C => if (x='0') then
        state <= C;
      else
        state <= A;
      end if;
    end case;
  end if;
end process;
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 (alternate model, continued)

-- Model next-state and output functions of the FSM
-- as combinational logic
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 (alternate model, 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 form for output and next state functions (combinational logic)

-- Next state function (combinational logic)

next_state <= A when ((curr_state = A) and (x = '0'))
    or ((curr_state = B) and (x = '0'))
    or ((curr_state = C) and (x = '1')) else
    B when ((curr_state = 1) and (x = '1')) else
    C;

-- Output function (combinational logic)

z <= '1' when ((curr_state = B) and (x = '1'))     --all conditions
    or ((curr_state = C) and (x = '1'))    --for which z=1.
    else '0';             --otherwise z=0
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 = 10,11
            end if;
        end if;
    end if;
deck process SYNC;
(Outputs on next slide)
Moore model outputs

-- Outputs = functions of the state
COMB: 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 COMB;
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.