Lecture 4: Modeling in VHDL (Continued...)

```vhdl
process (Clk, Start)
begin
  if Start = '0' then
    Q <= "0000" & Taeller;
  elsif rising_edge (Clk) then
    if Count(2) = '0' then
      -- IF not done THEN
      if Diff(4) = '1' then
        -- IF Diff < 0 THEN
        Q <= Q(6 downto 0) & '0';
      else
        Q <= Diff(3 downto 0) & Q(2 downto 0) & '1';
      end if;
    end if;
  end if;
end if;
end process;

process (Clk, Start)
begin
  if Start = '0' then
    Count <= "000";
  elsif rising_edge (Clk) then
    if Count(2) = '0' then
      Count <= Count+1;
    end if;
  end if;
end process;

result output

Heltal <= Q(3 downto 0);
Rest  <= Q(7 downto 4);
Faerdig <= Count(2);

end Behavioral;
```
Sequential Statements

Use **Process**

```process (sensitivity list)
variable/constant declarations
begin
    Sequential Statements
end process;
```
Sequential Statements

- **Process** waits for 1 or more of the signals in the sensitivity list to change
- It executes the statements in order then waits again
- **Sequential Assignment**

  `<signal> <= <expression> ;`

  - evaluates expression and schedules assignment to the signal at $t + \Delta$
Sequential Statements

- What does this process do?

  ```
  process (A, B)
  begin
    C <= A and B;
    Not_C <= not C;
  end process
  ```

- C never gets updated.

- Add C in the sensitivity list

- If you use “variables”, then assignment is instantaneous (e.g. `variable C : std_logic := '1'` comes before begin and variables are local to the process)
Sequential Statements

- Sequential if-then-else

  ```
  if <condition> then
    statements
  elsif <condition> then
    statements
  .
  .
  .
  else
    statements
  end if
  ```

else if needs

endif

elsif doesn’t...
Q: What does this model?

process(D, G)
begin
    if (G='1') then
        Q <= D;
    end if;
end process;
A: Transparent Latch (D Latch)

process(G, D)
begin
  if G = '1' then Q <= D; end if;
end process;
What is wrong with this model?

```vhdl
process (clk)
begin
    Q <= D;
end process;
```
process (clk, rst)
    begin
        if rst='1' then
            Q <= 0;
        else if clk='1' then
            Q <= D;
        end if
    end process;
Flip Flop with Asynchronous Reset

process (clk, rst)
    begin
        if rst='1' then
            Q <= 0;
        else if clk='1' then
            Q <= D;
        end if
    end process;

- This creates phantom clk when reset goes to ‘0’
- It won’t synthesize
Flip Flop with Asynchronous Reset

We only want to execute the clocked code if a change in clock (edge) triggered this execution:

```vhdl
process (clk, rst)
begin
  if rst='1' then
    Q <= 0;
  else if clk'event and clk='1' then
    Q <= D;
  end if
end process;
```

- Can also use
- `rising_edge(clk)`
Clocked vs. non-clock processes

Non-clocked process
(clock is NOT in the sensitivity list)

process (sel, a, my_data)
begin
  -- default all driven signals
  a_out    <= x"00";
data_out <= x"00";
if (sel = '1') then
  a_out      <= a;
data_out <= my_data;
end if;
end process;

Clocked process
(clock is ONLY in the sensitivity list)

process (clk)
begin
  -- check for rising edge of the clk
  if(clk'event and clk = '1')
  then
    -- initialize all driven signals during reset
    if(reset = '1') then
      a_out      <= x"00";
data_out   <= x"00";
    else
      if (sel = '1') then
        a_out <= a;
data_out <= my_data;
      end if;
    end if;
  end if;
end process;
Wait Statements

- Wait Statements replace sensitivity list
  - Makes process un-synthesizable

```
Wait on <sensitivity list>
Wait on A,B,C;

Wait for <time expression> --Use in test bench only
Wait for 5 ns

Wait until <Boolean expression>
Wait until A=B
```
Example

2-to-1 Multiplexer

*when statement*

```
-- conditional signal assignment statement
F <= I0 when A = '0' else I1;
```
Example

Cascaded 2-to-1 Multiplexer

*when statement*

```
F <= A when E = '1'
else B when D = '1'
else C;
```
Example

4-to-1 Multiplexer

*when statement*

If \(A = '0' \) and \(B = '0'\) then
\[ F \leq I_0; \]
elsif \(A = '0' \) and \(B = '1'\) then
\[ F \leq I_1; \]
elsif \(A = '1' \) and \(B = '0'\) then
\[ F \leq I_2; \]
elsif \(A = '1' \) and \(B = '1'\) then
\[ F \leq I_3; \]
else -- (\(A \) or \(B \) are not 0 or 1)
\[ F \leq 'X'; \]
end if;
Example

4-to-1 Multiplexer

*when statement*

\[
F \leq I_0 \text{ when } (A\&B) = "00" \text{ else } \\
I_1 \text{ when } (A\&B) = "01" \text{ else } \\
I_2 \text{ when } (A\&B) = "10" \text{ else } \\
I_3 \text{ when } (A\&B) = "11" \text{ else } 'X';
\]
Example

4-to-1 Multiplexer

With and when statement

```
 sel <= A&B;  --selected signal assignment statement
 with sel select
  F <= I0 when "00"
    I1 when "01"
    I2 when "10"
    I3 when "11"
```
Example

4-to-1 Multiplexer

Case statement (inside process)

```process (A,B,I0,I1,I2,I3)
variable sel:std_logic_vector(1 downto 0);
begin
  sel:= A & B;
  --concatenate A and B
  case sel is
    when  "00"  =>  F <= I0;
    when  "01"  =>  F <= I1;
    when  "10"  =>  F <= I2;
    when  "11"  =>  F <= I3;
    when  others=>  F <= 'X';
  end case;
end process
```
case <expression> is

  when <case1> => <sequential statements>

  when <case2> => <sequential statements>

  when others => <sequential statements>

end case
entity DFF is
  port (D, CLK, CLR: in bit;
    Q: out bit; QN: out bit:='1');
---initialize Q' to '1' since bit signals are initialized to '0' by default
end DFF
architecture so of DFF is
begin
  process (CLK,CLR)
  begin
    if CLR='0' then
      Q <= '0'; QN<= '1';
    else if CLK'event and CLK='1' then
      Q <= D after 10 ns;
      QN <= not D after 10 ns;
    end if;
  end if;
end process;

Rising Edge of the Clock
Can also use if rising_edge(clk)) then