if-then else : 2-1 mux

mux: process (A, B, Select)
begin
  if (select='1') then
    Z <= A;
  else
    Z <= B;
  end if;
end process;
Conditional signal assignment
2-1 Mux

\[ Z \leq A \text{ when } (\text{Select} = '1') \]
\[ \text{else B; } \]

Concurrent statement - I.e. outside process.
Case statement: 2-1 Mux

mux: process
begin
  case Select is
    when '0' => Z <= B;
    when '1' => Z <= A;
    when others => Z <= '-';  -- cover all cases
  end case;
end process;
Priority Encoder using if-then-elsif

\[ Z \leftarrow E; \quad \text{-- default} \]

\[
\text{if (SELECT}_D = \text{‘1’}) \text{ then } Z \leftarrow D; \\
\text{elsif (SELECT}_C = \text{‘1’}) \text{ then } Z \leftarrow C; \\
\text{elseif (SELECT}_B = \text{‘1’}) \text{ then } Z \leftarrow B; \\
\text{elseif (SELECT}_A = \text{‘1’}) \text{ then } Z \leftarrow A; \\
\text{end if;}
\]
Decoder using case statement

case_value := A & B & C; -- concatenate bits

CASE case_value IS
  WHEN "000" => Z <= "00000001";
  WHEN "001" => Z <= "00000010";
  ...  
  WHEN others => Z <= "00000000";

END CASE;
Case statements

- Truth table can be directly translated into CASE statement. (I.e. hex to 7-segment)
- State transition table can also be implemented using CASE statement.
- Synthesized as combinational logic (unless incompletely specified.)
Case vs. If-then-elsif

• Case statement generates hardware with more parallelism.
• If-then-elseif has built-in priority; Can require lots of logic for low-priority conditions.
• If conditions are mutually exclusive, use case statement to simplify logic.
Inferred Latch (undesirable!)

process (A, Select)
begin
    if (Select='1') then
        Z <= A;
    end if;
end process;
Preventing Latch Inference

• If-statements and case statements must be completely specified or VHDL compiler infers latches.

• A default assignment must be made so that an assignment occurs for all conditions.

• #1 synthesis problem for Xilinx - although simulation will work, the final hardware most likely will NOT work!
D-FF Simulation Model

process (CLK) begin
  if CLK='1' then
    Q <= D;
  end if;
end process;

• Incomplete sensitivity list
• What will be synthesized??
D flip-flop

DFF : process
begin
    wait until clk’event and clk=‘1’;
    Q <= D;
end process;
D flip-flop with Asynch Reset

process (clk, reset, D)
begin
  if (reset='1') then
    Q <= '0';
  elsif (clk'event and clk='1') then
    Q <= D;
  end if;
end process;
D-FF with async reset and enable

process (clk, reset, D, enable) begin
  if (reset='1') then
    Q <= '0';
  elsif (clk'event and clk='1') then
    if (enable='1') then
      Q <= D;
    end if;
  end if;
end if;
end process;
D-FF with Synch Reset

process begin
  wait until clk’event and clk=‘1’;
  if (reset=‘1’) then
    Q <= ‘0’;
  else
    Q <= D;
  end if;
end process;
Synthesis Guidelines

• Avoid inferring latches.
• Don’t specify time delays.
• Avoid incomplete sensitivity lists.
• Don’t specify initial values in entity, signal or variable declarations.
• Use basic constructs and behavioral code.
• Use std_logic and std_logic_vector types.
Synthesis Guidelines (cont.)

• A process must have a sensitivity list or one or more wait statements.

• All paths through process code must be balanced. I.e. if one path has a wait statement, all paths must have a wait statement.
IEEE-1164 Standard Logic

- accurately models hardware.
- std_logic replaces type bit
- std_logic_vector replace bit_vector
- Recommended for simulation and synthesis
- Special operator overloading functions and resolution functions.
Using IEEE-1164

Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

- Library source code available on our system.
- Can add std_logic vectors, model tri-state buffers, etc.
How will this code be synthesized?

process
begin
    wait until clk’event and clk=‘1’;
    synch_in <= asynch_in;
    stage_out <= synch_in;
end process;
Variables

- Local to a process.
- Instantaneously updated - no delta delay.
- Useful for algorithmic description which needs to generate/use intermediate values. (I.e. inside for loop)
- Contrast: signals are global to process only
- Signals are *scheduled*, not updated instantly.
Parity generator using variable

process (WORD) -- bit_vector(0 to 4)
variable ODD : bit;
begin
  ODD := '0';
  for I in 0 to 4 loop
    ODD := ODD xor WORD(I);
  end loop;
PARITY <= ODD;
end process;
Summary

• Know how to use *basic constructs* to produce predictable, reliable synthesis results.
• Know how to avoid inferring latches!
• Signals are *scheduled*; Variables update instantly.
• Model state machines using *separate processes* for state registers and for next-state combinational logic.