

Processes (cont)

```
senslist: process (a, b);  
begin  
    <sequential  
statement>;  
    <sequential  
statement>; . . .  
end process senslist;
```

First kind of process – with a sensitivity list. It executes from the first line and suspends at the last line.

```
waitst: process;  
begin  
    <sequential  
statement>;  
    <wait statement>  
    <sequential  
statement>;  
    <wait statement>
```

Second kind of process – without a sensitivity list. It suspends and re-executes from the *wait* statement.

The first rule for synthesizing the combinational circuits from the process:
The sensitivity list should be complete. It means that it must contain each signal which value is read within the process.

Latches and Registers

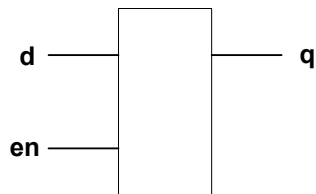
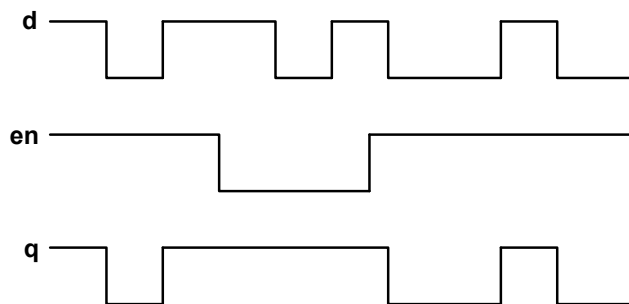


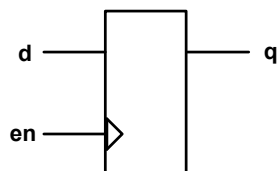
Figure 1. Latch

Latch is a memory element which output is followed its input as long as it is enabled. When the enable signal is invalid the output retains it's last value. We can call a latch as a level sensitive register. As a rule, latches are accidental in the design and we should know how avoid them.

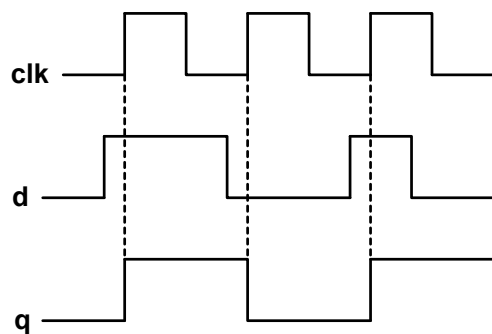


| en | q |
|----|---|
| 1 | d |
| 0 | q |

Figure 2. Latch behavior



| clk | q |
|--------|---|
| 0 or 1 | d |
| | q |



The output of a flip-flop takes the input value only on the edge of an enable signal (on the edge of the clock).

Figure 3. Flip-flop

Inferred Latches

Inferred latches are registers described within VHDL code. Usually, a designer may not want latches and they are a common problem in digital design using HDLs.

Incomplete Assignment

```
library ieee;
use ieee.std_logic_1164.all;

entity latch is
port (en, d : in std_logic;
      q : out std_logic);
end latch;

architecture beh_ latch of latch is
begin
  process (en, d)
  begin
    if (en = '1') then
      q <= d;
    end if;
  end process;
end beh_ latch;
```

| en | q |
|----|---|
| 1 | d |
| 0 | q |

The process shown in this example synthesizes a transparent latch. When en = '1', the value of d is assigned to q, but when en = '0', q retains it's last value.

Special care is necessary in order to avoid the generation of latches.

```
architecture bad_ mux of
latch is
begin
  process (a, b, sel)
  begin
    if (sel = '1') then
      y <= a;
    end if;
  end process;
end bad_ mux;
```

```
architecture good1_ mux of
latch is
begin
  process (a, b, sel)
  begin
    y <= b;
    if (sel = '1') then
      y <= a;
    end if;
  end process;
end good1_ mux;
```

```

architecture good2_mux of
  latch is
begin
  process (a, b, sel)
  begin
    if (sel = '1') then
      y <= a;
    else
      y <= b;
    end if;
  end process;
end good2_mux;

```

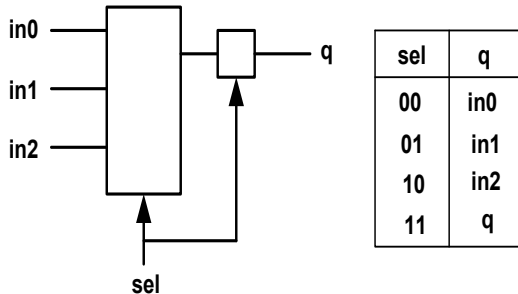
The two last files – *good1_mux* and *good2_mux* are identical and synthesis tool implements them as multiplexer. In *good1_mux* if *sel* = '1' the first assignment *y* <= *b* will be overridden by *y* <= *a* because signals are updated only at the end of process execution.

Incomplete Case Statements

```

process (in0, in1, in2, ctr)
begin
  case ctr is
    when "00" => q <= in0;
    when "01" => q <= in1;
    when "10" => q <= in2;
  end case;
end process;

```



This piece of code is syntactically correct, but no case branch is included for *sel* = 11 so it is incomplete case statement.

If *sel* goes to 11, *q* will retain its former value determined by the previous state of *sel* and such a behavior corresponds to our earlier definition of a latch. So the synthesis tool will infer that a latch is required.

Figure 4. Code for bad multiplexer - latch inferred

The second rule for synthesizing the combinational circuits from the process: Always include a default statement to make sure that incomplete assignment does not take place.

```

process (x1, x2, ctr)
begin
    case ctr is
        when '0' =>
            y1 <= x1;
            y2 <= x2;
        when '1' =>
            y1 <= x2;
    end case;
end process;

```

| ctr | y1 | y2 |
|-----|----|----|
| 0 | x1 | x2 |
| 1 | x2 | y2 |

When ctr is 1, the output y2 is not assigned and this is an incomplete assignment as well. The general description of an incomplete assignment is where signal is assigned to in some branches of the case statement but not within the others.

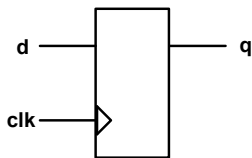
```

process (x1, x2, ctr)
begin
    y2 <= x1;
    case ctr is
        when '0' =>
            y1 <= x1;
            y2 <= x2;
        when '1' =>
            y1 <= x2;
    end case;
end process;

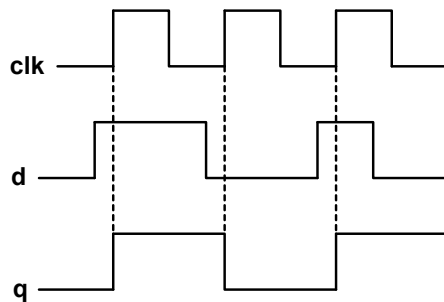
```

A reliable method is to make default assignments at the top of the procedure, so a new value is assigned to all of the output variables each time the case statement is invoke.

D flip-flops in VHDL



| clk | q |
|-------------|--------|
| ↑ 0 or 1 | d q |



The output of a flip-flop takes the input value only on the edge of an enable signal (on the edge of the clock).

Clock and Clocking

Clock is a signal that:

- Synchronizes complete design
- “Tells” individual blocks when to exchange data
- Clock is defined by its active edge, frequency, duty cycle, rise/fall time, etc
- On this course we are concerned with the clock active edge – when flops latch data

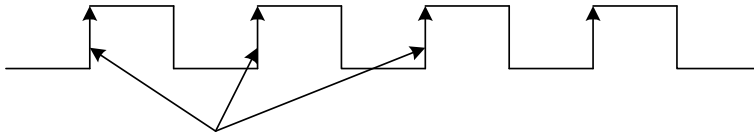


Figure 5. Clock Active Edge (in this case the rising edge)

Two templates to present D flip flops in VHDL

```
process (clock)
begin
    if clock'event and clock = '1' then
        q <= d;
    end if;
end process;
```

The signal attribute 'event' is presented the change of a signal. *clock'event and clock = '1'* means that the clock was changed and it's value became '1'.

```
process
begin
    wait until clock'event and clock = '1' then
        q <= d;
    end if;
end process;
```

```
process (clock)
begin
    if clock'event and clock = '1' then
        q0 <= d0;
        q1 <= d1;
    end if;
end process;
```

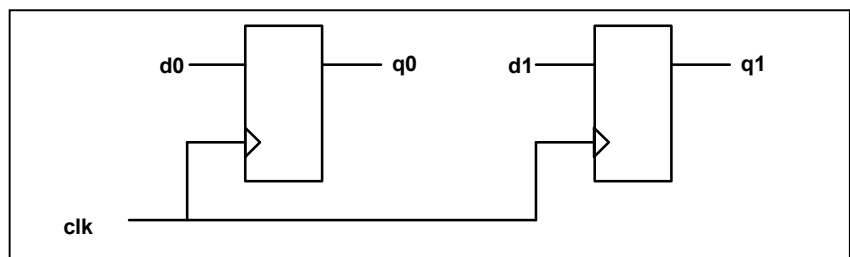


Figure 6. Circuit with two flip-flops

```

library ieee;
use ieee.std_logic_1164.all;

entity dff_logic is
    port (clk, in1, in2: in std_logic;
          q : out std_logic);
end dff_logic;

architecture arc_dff_logic of dff_logic is
begin
    process (clk)
    begin
        if (clk'event and clk='1') then
            q <= (not in1) and in2;
        end if;
    end process;
end arc_dff_logic;

configuration cnf_dff_logic of dff_logic is
    for arc_dff_logic end for;
end cnf_dff_logic;

```

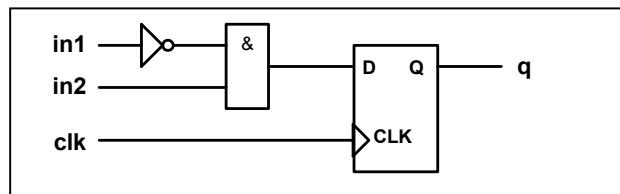


Figure 7. D flip-flop with combinational logic

D flip-flop with asynchronous reset

```

library ieee;
use ieee.std_logic_1164.all;

entity dff_arst is
    port (clk, d, arst: in std_logic;
          q : out std_logic);
end dff_arst;

architecture arc_dff_arst of dff_arst is
begin
    process (clk, arst)
    begin
        if (arst = '1') then
            q <= '0';
        elsif (clk'event and clk='1') then
            q <= d;
        end if;
    end process;
end arc_dff_arst;

configuration cnf_dff_arst of dff_arst is
    for arc_dff_arst end for;
end cnf_dff_arst;

```

D flip-flop with asynchronous preset

```
library ieee;
use ieee.std_logic_1164.all;

entity dff_aprst is
    port (clk, d, aprst: in std_logic;
          q : out std_logic);
end dff_aprst;

architecture arc_dff_aprst of dff_aprst is
begin
    process (clk, aprst)
    begin
        if (aprst = '1') then
            q <= '1';
        elsif (clk'event and clk='1') then
            q <= d;
        end if;
    end process;
end arc_dff_aprst;

configuration cnf_dff_aprst of dff_aprst is
for arc_dff_aprst end for;
end cnf_dff_aprst;
```

D flip-flop with synchronous reset

```
library ieee;
use ieee.std_logic_1164.all;

entity dff_srst is
    port (clk, d, srst: in std_logic;
          q : out std_logic);
end dff_srst;

architecture arc_dff_srst of dff_srst is
begin
    process (clk)
    begin
        if (clk'event and clk='1') then
            if (srst = '1') then
                q <= '0';
            else
                q <= d;
            end if;
        end if;
    end process;
end arc_dff_srst;

configuration cnf_dff_srst of dff_srst is
for arc_dff_srst end for;
end cnf_dff_srst;
```


Rules for a clocked process

```
process
begin
    wait until clk'event and CLK='1';
    if srst = '1' then
        -- synchronous register reset
    else
        -- combinatorics
    end if;
end process;
```

Wait form:

- No sensitivity list;
- Synchronous reset.

```
process (clk, rst)
begin
    if (rst = '1') then
        -- asynchronous register reset
    elsif (clk'event and CLK='1') then
        -- combinatorics
    end if;
end process;
```

If form:

- Only clock and asynchronous signals (reset) is in sensitivity list;
- Synchronous and asynchronous resets

Clocked Process: Examples

--Eight-bit register with asynchronous clear

```
library ieee;
use ieee.std_logic_1164.all;

entity reg8 is
    port (d : in std_logic_vector(7 downto 0);
          reset, clock : in std_logic;
          q : out std_logic_vector(7 downto 0));
end reg8;

architecture behavior of reg8 is
begin
    process (reset, clock)
    begin
        if reset = '1' then
            q <= "00000000";
        elsif clock'event and clock = '1' then
            q <= d;
        end if;
    end process;
end behavior;
```

--Four-bit counter with parallel load, using integer signals

```
library ieee;
use ieee.std_logic_1164.all;

entity upcount is
    port (d : in integer range 0 to 15;
          clock, reset, load : in std_logic;
          q : inout integer range 0 to 15);
end upcount;

architecture behavior of upcount is
begin
    process ( clock, reset )
    begin
        if reset = '1' then
            q <= 0;
        elsif (clock'event and clock = '1') then
            if load = '1' then
                q <= d;
            elsif q = 15 then
                q <= 0;
            else
                q <= q + 1;
            end if;
        end if;
    end process;
end behavior;
```