

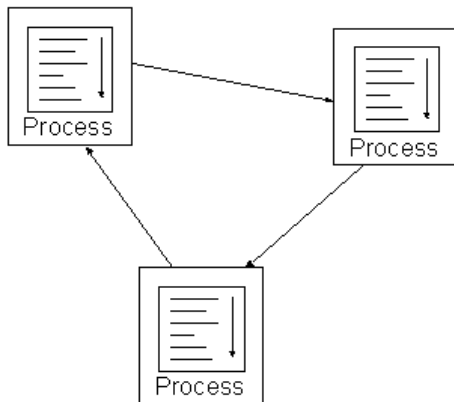
Figure 1. Sequential execution flow in a processor

```

mux: process (a, b, sel)
begin
    if sel = '1' then
        z <= a;
    else
        z <= b;
    end if;
end process mux;

```

Process is invoked whenever there is an event on any signal in the sensitivity list.



The processes in the same architecture are working concurrently. One process can produce an event in the sensitivity list of the other processor to awake it.

Figure 2. Multiple processor interaction

```

architecture a of e is
begin
    -- concurrent statements
    p1 : process
    begin
        -- sequential statements
    end process p1;
    -- concurrent statements
    p2 : process
    begin
        --sequential statements
    end process p2;
    -- concurrent statements
end a;

```

Pay attention – if the process has a label it should be repeated at the end of this processor.

Figure 3. Concurrent statements within the architecture

### Two kinds of processes

```
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>  
    . . .  
end process waitst;
```

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

### Decoder 2x4 with sensitivity list

```
library IEEE;  
use IEEE.std_logic_1164.all;  
  
entity decod2x4 is  
    port (a, b, en : in std_logic;  
          y : out std_logic_vector(0 to 3));  
end decod2x4;  
  
architecture behav_if of decod2x4 is  
begin  
    process (a, b, en)  
        variable na, nb : std_logic;  
    begin  
        na := not a;  
        nb := not b;  
        if en = '1' then  
            y(0) <= na and nb;  
            y(1) <= na and b;  
            y(2) <= a and nb;  
            y(3) <= a and b;  
        else  
            y <= "0000";  
        end if;  
    end process;  
end behav_if;  
  
configuration cfg_behav_if of decod2x4 is  
    for behav_if  
    end for;  
end cfg_behav_if;
```

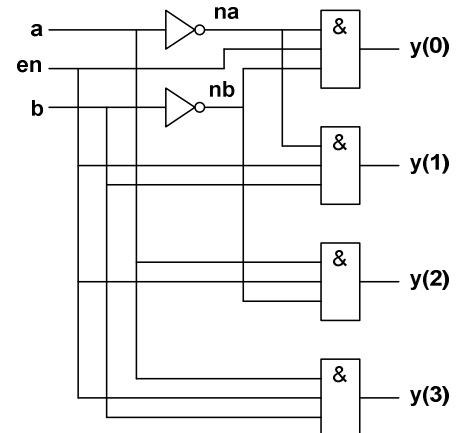


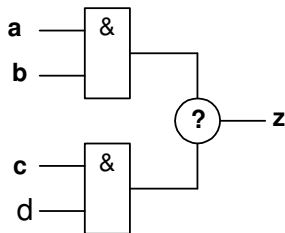
Figure 4. Decoder 2x4 - logic circuit

### Decoder 2x4 with wait statement

```
architecture behav_wait of decod2x4 is
begin
    process
        variable na, nb : std_logic;
    begin
        na := not a;
        nb := not b;
        if en = '1' then
            y(0) <= na and nb;
            y(1) <= na and b;
            y(2) <= a and nb;
            y(3) <= a and b;
        else
            y <= "0000";
        end if;
        wait on a, b, en;
    end process;
end behav_wait;

configuration cfg_behav_wait of decod2x4 is
    for behav_wait
    end for;
end cfg_behav_wait;
```

```
architecture concurr of twodrives is
    signal z, a, b, c, d : std_logic;
begin
    z <= a and b;
    z <= c and d;
end concurr;
```

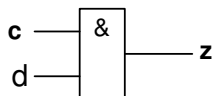


Such assignments need two drivers.  
Signal z needs resolution function to  
determine the final value.

***Don't use such assignments!***

Figure 5. Two drivers in the architecture

```
architecture sequential of multiple is
    signal z, a, b, c, d : std_logic;
begin
    process (a, b, c, d)
    begin
        z <= a and b;
        z <= c and d;
    end process;
end sequential;
```

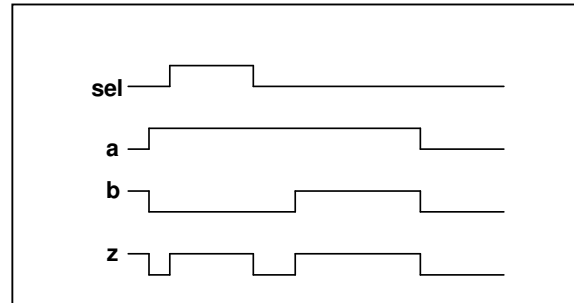


Signals assigned within the process  
are updated when the process  
suspends. So, z will be equal **c and d**  
and is never updated with **a and b**.

Figure 6. Two drivers in the process

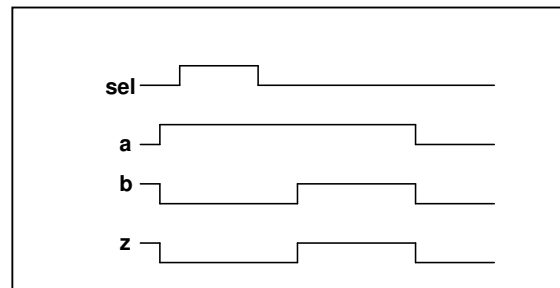
### Sensitivity list

```
mux: process (a, b, sel)
begin
    if sel = '1' then
        z <= a;
    else
        z <= b;
    end if;
end process mux;
```



The same process without *sel* in sensitivity list:

```
mux: process (a, b)
begin
    if sel = '1' then
        z <= a;
    else
        z <= b;
    end if;
end process mux;
```



**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.***

### If statement

```
if condition then
    -- sequential statements
end if;
```

```
if condition then
    -- sequential statements
else
    -- sequential statements
end if;
```

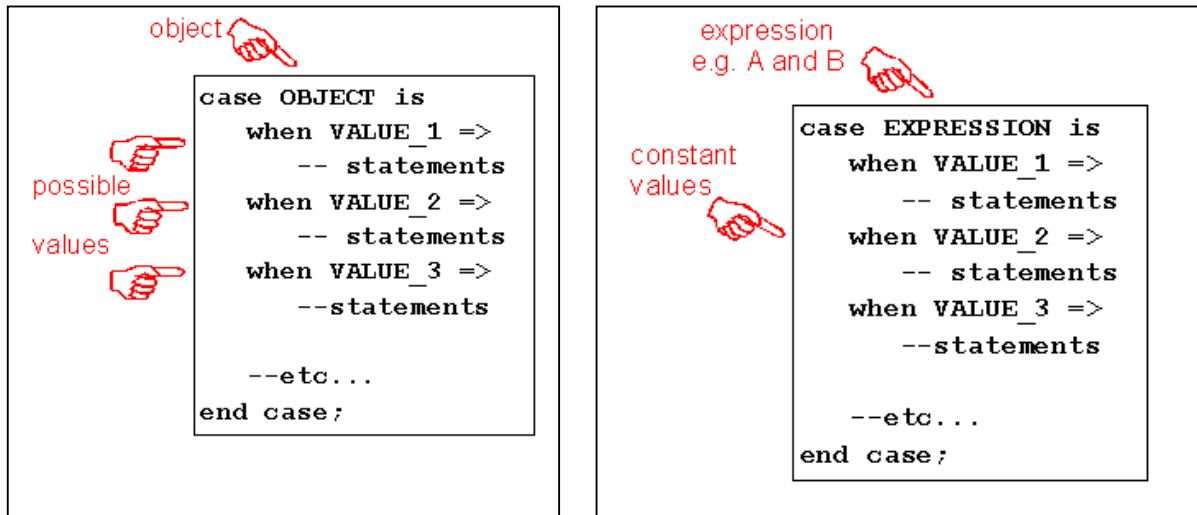
```
if condition then
    -- sequential statements
elsif condition then
    -- sequential statements
elsif condition then
    -- sequential statements
else
    -- sequential statements
end if;
```

***Always only one branch executed.***

```
process (a, b, c, x)
begin
    if (x = "0000") then
        z <= a;
    elsif (x <= "0101") then
        z <= b;
    else
        z <= c;
    end if;
end process;
```

Here the conditions are overlapped: if  $x = "0000"$  then both conditions are true. But because the condition  $x = "0000"$  is written first, then the assignment  $z <= a$  will be implemented. Operator if has a built in priority.

## Case statement



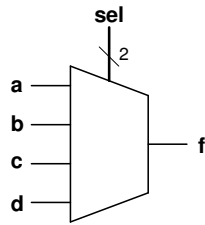
```
case sel is
  when "00" =>
    f <= a;
  when "01" =>
    f <= b;
  when "10" =>
    f <= c;
  when "11" =>
    f <= d;
  when others =>
    f <= 'x';
end case;
```

All possible values of the object must be covered by the *when* branches. The *others* clause covers all other possible values of *sel* that have not been specified, for example, covers the cases including 'X', 'U', 'Z', etc of std\_logic.

```
case sel is
  when "00" =>
    f <= a;
  when "01" =>
    f <= b;
  when "01" =>
    f <= c;
  when "10" =>
    f <= d;
  when "11" =>
    f <= e;
end case;
```

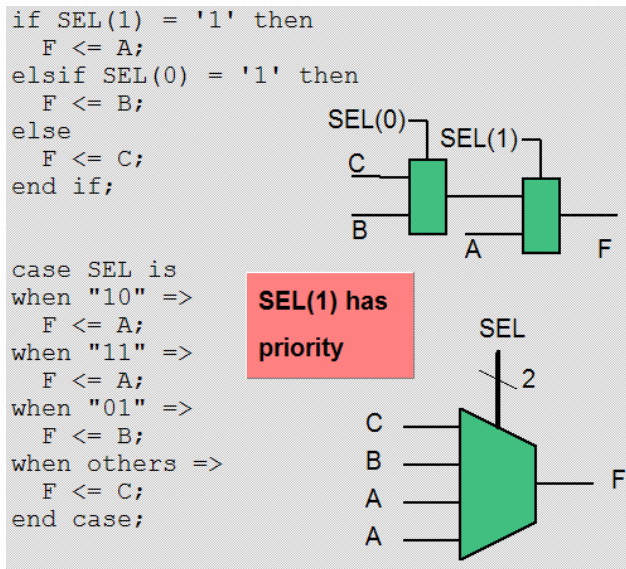
Error, each expression can be covered only once.

```
case address is
  when 0 to 7 => -- values presented as a range
    a <= '1';
  when 8 to 15 => -- values presented as a range
    b <= '1';
  when 16 | 20 | 24 | 28 => -- the pipe '|' symbol
    -- is treated here as OR
    a <= '1'; -- any number of sequential statements
    b <= '1'; -- can be here
  when others => null; -- empty operator
end case;
```



Synthesis tools implement case statement as a multiplexer

Figure 7. MUX 4x1



The conditions in an *if statement* are tested in sequence according priority inputs. The choices in *case statement* are tested in parallel, which make simulation faster.

Good ASIC synthesis tools can optimize both implementations very well. For FPGAs *elsif* and nested if statements can result in more logic levels and a non-optimal implementation.

Figure 8. If versus case

#### 4(8bit) – to – 1(8bit) line multiplexer in VHDL presentation using “IF” form

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

entity MUX4x8 is
    port (in1,in2,in3,in4 : in std_logic_vector (7 downto 0);
          cntrl : in std_logic_vector (1 downto 0);
          q: out std_logic_vector (7 downto 0));
end MUX4x8 ;

architecture arc_MUX4x8 of MUX4x8 is
begin
    process (in1, in2, in3, in4, cntrl)
    begin
        if cntrl = "00" then q <= in1;
        elsif cntrl = "01" then q <= in2;
        elsif cntrl = "10" then q <= in3;
        elsif cntrl = "11" then q <= in4;
        else q <= "00000000"; -- q <= (others => '0');
        end if;
    end process ;
end arc_MUX4x8;

configuration cfg_MUX4x8 of MUX4x8 is
    for arc_MUX4x8
    end for;
end cfg_MUX4x8;
```

#### 4(8bit) – to – 1(8bit) line multiplexer in VHDL presentation using “CASE” form

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

entity MUX4x8 is
    port(in1, in2, in3, in4 : in std_logic_vector (7 downto 0);
          cntrl : in std_logic_vector (1 downto 0);
          q: out std_logic_vector (7 downto 0));
end MUX4x8 ;

architecture arc_MUX4x8 of MUX4x8 is
begin
    process (in1, in2, in3, in4, cntrl)
    begin
        case cntrl is
            when "00" => q <= in1;--mux output := in1
            when "01" => q <= in2;--mux output := in2
            when "10" => q <= in3;--mux output := in3
            when "11" => q <= in4;--mux output := in4
            when others => q <= "00000000";
            --when others => q<=(others => '0');
        end case;
    end process ;
end arc_MUX4x8;

configuration cfg_MUX4x8 of MUX4x8 is
    for arc_MUX4x8
    end for;
end cfg_MUX4x8;
```

### Loop statement

```
for i in 0 to 3 loop
  f(i) <= a(i) and b(3-i);
  v := v xor a(i);
end loop;
```

*i* is a loop parameter. Two ranges are available – ascending and descending, but the loop parameter can decrement or increment only by 1.

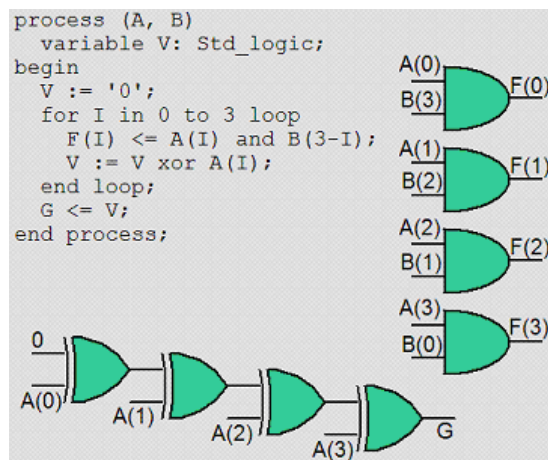
```
for i in 3 downto 0 loop
  f(i) <= a(i) and b(3-i);
  v := v xor a(i);
end loop;
```

```
for i in 0 to 3 loop
  f(i) <= a(i) and b(3-i);
  v := v xor a(i);
  if v = 'x' then
    i := 4; -- It is illegal
  end if;
end loop;
```

A loop parameter cannot be changed by assignment.

```
process (a, b)
  variable i: std_logic;
  ...
begin
  for i in 0 to 3 loop
    f(i) <= a(i) and b(3-i);
    v := v xor a(i);
  end loop;
  i := not i; -- It is legal
end process;
```

The *i* in the last but one row is not a loop parameter. The loop parameter is hidden within the loop and cannot be watched outside the loop.



While synthesized loops make multiple copies of logic inside the loop, one copy for each possible value of the loop parameter. It is the reason why the borders of a range must be constant.

Figure 9. How loops are synthesized



```
For parameter in loop_range loop
```

```
    . . .
end loop;
```

```
while condition loop
```

```
    . . .
end loop;
```

```
loop
```

```
    . . .
end loop;
```

There are three different kinds of loops in VHDL. The while loop is implemented while a condition is true. The unbounded loop statement loops forever. Only the for loop with constant bounds is synthesizable. The other loops are useful in test benches.

```
loop
```

```
    . . .
    exit;
```

```
    . . .
```

```
    exit when condition;
```

```
    . . .
```

```
end loop;
```

The exit statement is a sequential statement that controls the jump to the statement following the loop.

```
l1: for i in 0 to 7 loop
    l2: for j in 0 to 7 loop
        c := c + 1;
        exit l2 when a(j) = b(i);
        exit l1 when b(c) = 'u';
    end loop l2;
end loop l1;
```

Loops can be labeled to indicate which loop to exit

```
main: for i in 0 to 15 loop
    . . .
    next main when reset = '0';
    . . .
end loop main;
```

The next statement causes control to pass back to the top of the loop. The loop parameter takes the next value in its range.

```
ClockGenerator_1: process
begin
    for I in 1 to 1000 loop
        Clock <= '0';
        wait for 5 NS;
        Clock <= '1';
        wait for 10 NS;
    end loop;
    wait;
end process ClockGenerator_1;
```

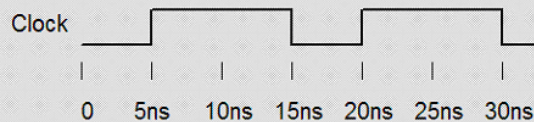
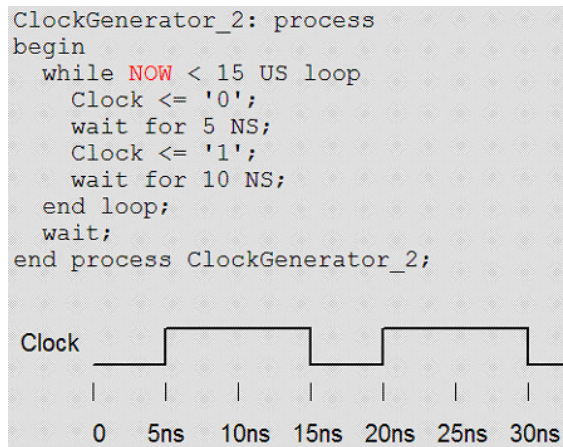
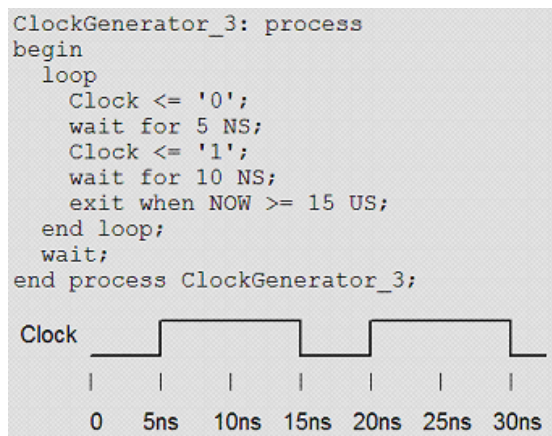


Figure 10. Clock generator (for loop)



**Figure 11. Clock generator (while loop)**



**Figure 12. Clock generator (unbounded loop)**