Filename="ch9,doc"

9.0 Data Objects

In VHDL, a data object holds a value of some specified type and can be classified into one of the following three classes: constants, variables, or signals. The declaration syntax is :

OBJECT_CLASS identifier [,identifier ...] :TYPE [:=value];

9.1.1 Constant Class

An object of class constant holds a single value of of a given type. It must be assigned a value upon declaration, and the value can not be changed subsequently. The declaration syntax is:

CONSTANT identifier [,identifier ...]:TYPE:=value;

Example:

```
CONSTANT a1:REAL :=1.2;
CONSTANT word_size:INTEGER:= 16;
```

9.1.2 Variable Class

An object of class variable holds a single value of a given type. It can be assigned new value any number of times during program executions. It needs not be initialized upon declaration. The declaration syntax is:

VARIABLE identifier [,identifier ...]:TYPE [:=value];

Example:

```
VARIABLE  counter: BIT_VECTOR(3 DOWNTO 0) := "0000";
VARIABLE  sum: REAL;
```

Variables are changed by executing an assignment operator. For example,

```
counter := "0001";
sum := 0.0;
```

The variable assignments have no time dimension associated with them. That is, the assignments take their effect immediately. Thus variable has no direct analogue in hardware. Variable can only be used within sequential areas, within a PROCESS, in subprograms (functions and procedures) and not within ARCHITECTURE BODY.

```
PROCESS(a,b)
        VARIABLE val1:STD_LOGIC:='0';
BEGIN
        val1 := a;              --variable val1 is assigned the value of signal a.
        b <= val1;              --signal b is assigned the value of variable val1.
END;
```

9.1.3 Signal Class

An object of class signal can hold or pass logic values, while variables cannot. A signal is a pathway along which information passes from one component in a VHDL description to another. It is analogous to a wire in hardware. It needs not be initalized upon declaration.

SIGNAL identifier [,identifier ...] :TYPE [:=value];

Example:

SIGNAL sigA, sigB: BIT;

Signals are objects whose values may be changed and have a time dimension. Signal values are changed by signal assignment operator (<=).

Example:

```
a <= b AFTER 10 ns;
c<= d OR e;
```

The AFTER 10 ns clause in the first example means that signal a will be assigned signal b 10 ns later. In the second example there is no AFTER clause; this is equivalent to AFTER delta. That is, signal assignments are used to represent real circuit phenomena, so if there is no AFTER clause, it is assumed that the signal takes on its new value delta time later. Delta is an arbitrary small time greater than zero.

9.1.4 Signal Attributes supported by Autologic VHDL

Signal_name'EVENT – returns true if an event (i.e. its value change) has occurred on the signal.

Signal_name'LAST_VALUE – returns the previous value of the signal immediately before its last change.

Signal_name'STABLE – returns true when the signal has had no events (i.e. its value has not changed).

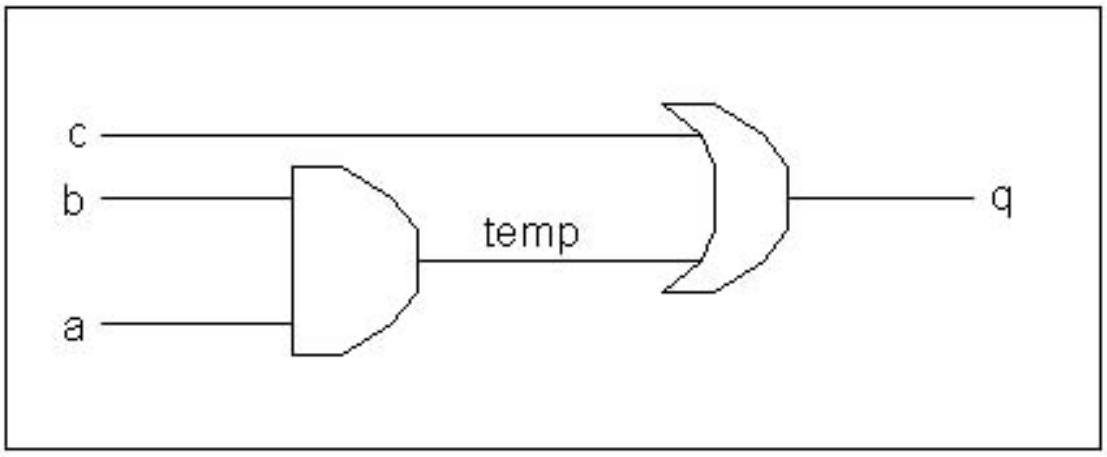
9.2 Signals and Variables

Whenever the code assigns a value to a variable, the simulator simply updates the current value of that variable, as you would expect in any programming language.

But when the code assigns a value to a signal, that assignment is treated differently. The signal has a current value, which is used whenever the signal appears in an expression, and a list of "next values," each of which consists of a pair of data items: a value for the signal, and the number of simulation cycles after which the signal is actually given that value. So, a signal assignment does not affect the current value of the signal but only the value for a future simulation cycle. At the end of each simulation cycle, the simulator scans through all the signals, and updates each one from its "next values" list for the next cycle that is to occur.

```
1   LIBRARY IEEE;
2   USE IEEE.STD_LOGIC_1164.ALL;
3
4   ENTITY  and_or IS
5       PORT(a,b,c : IN BIT;
6                q: OUT BIT);
7   END and_or;
8    ARCHITECTURE archand_or OF and_or IS
```

```
 9          SIGNAL temp : BIT;
10      BEGIN
11          temp <= a AND b;
12          q <= temp OR c;
13      END archand_or
```



TIME	cba	temp	q
0	001	0	0
t	011	0	0
t + Δ	011	1	0
t + 2Δ	011	1	1

The current value of temp is calculated based on the previous values of a and b. The current value of q is calculated based on the previous values of c and temp.

9.2.1 When to use Variables

Signals assigned to in a process are updated at the end of the process. The signal driver is filled with the new value when the assignment statement is executed. Then, as the last step before the process is suspended, that driver is passed to the signal.

Variables, on the other hand, are updated as soon as the variable assignment is executed.

The following process based code implementation of the and_or circuit **can't be implemented**.

```
1       LIBRARY IEEE;
2       USE IEEE.STD_LOGIC_1164.ALL;
3
4       ENTITY  and_or IS
5           PORT(a,b,c : IN BIT;
6                    q: OUT BIT);
7       END and_or;
```

```
 8      ARCHITECTURE archand_or OF and_or IS
 9          SIGNAL temp : BIT;
10      BEGIN
11          PROCESS(a,b,c)
12          BEGIN
13              temp <= a AND b;
14              q <= temp OR c;
15          END PROCESS;
16      END archand_or
```

TIME	cba	temp	q
0	001	0	0
t	011	0	0
t + Δ	011	1	0

Since temp is not in the sensitivity, the event in temp does not cause the activation of q in line 14. In addition, because temp is not in the sensitivity list, this implies that its value must be stored, hence the flip-flop. The problem then is to determine the clock for the flip-flop. Because the process is triggered by any event on a, b, or c, the flip-flop must be active to both a rising and a falling edge for any or all three of the signals. Because very few libraries have flip-flops that are active to edges of a clock, a new function must be built. The edge detector would have to create an active edge for the flip-flop whenever an input signal changes, but this cannot be synthesized. Modify the process sensitivity list to include temp, as follows:

```
PROCESS(a,b,c,temp);
```

That is, for a process based description to be synthesized as a combinatorial network, any signal that appears on both the right-hand-side (RHS) and the left-hand-side (LHS) of assignment statements must be included in the process sensitivity definition.

To more efficiently describe intermediate results of operations that are not needed outside of a process you can use variables.

```
 1      LIBRARY IEEE;
 2      USE IEEE.STD_LOGIC_1164.ALL;
 3
 4      ENTITY  and_or IS
 5          PORT(a,b,c : IN BIT;
 6                   q: OUT BIT);
 7      END and_or;
 8       ARCHITECTURE archand_or OF and_or IS
 9       BEGIN
11          PROCESS(a,b,c)
12              VARIABLE temp : BIT;
12          BEGIN
13              temp := a AND b;
14              q <= temp OR c;
15          END PROCESS;
16      END archand_or
```

TIME	cba	temp	q
0	001	0	0
t	011	1	0
t + Δ	011	1	1

9.3 Autologic VHDL STATEMENTS AND THEIR CORRESPONDING HARDWARE SYNTHESIS

Not all VHDL statements have corresponding hardware synthesis. In particular, ASSERT statement which generates messages to the screen do not have a hardware synthesis.

9.3.1 Signal Assignments

There are three types of signal assignment: simple signal assignment, conditional signal assignment, and selected signal assignment.

9.3.1.1 Simple Signal Assignment

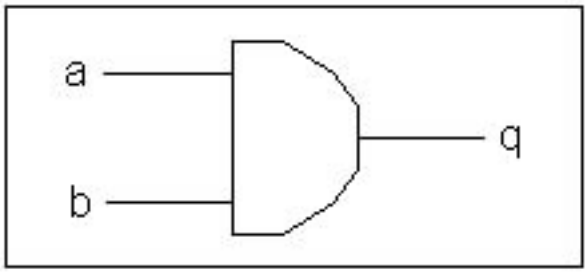
The simple signal assignment statement specifies that a target signal is to receive some waveform. Its syntax is:

```
target_signal <= waveform;
```

Example:

```
q <= a AND b
```

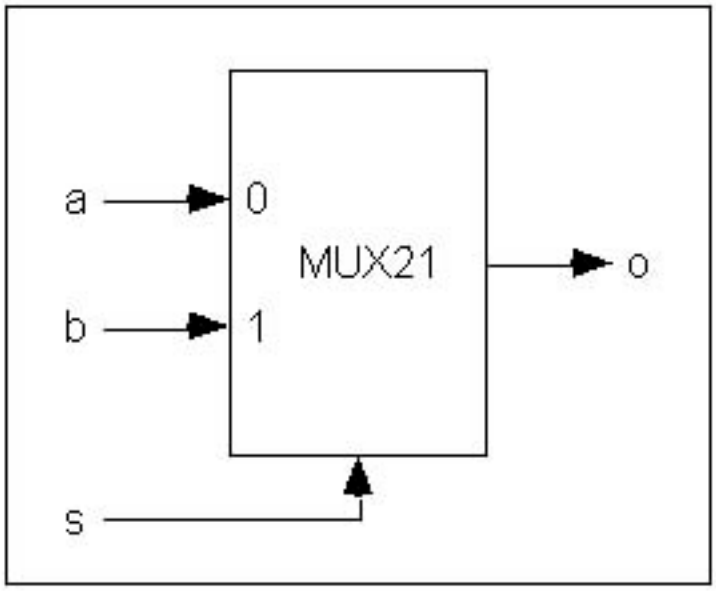
Assumming a, b, and q are signals of type BIT, it is synthesize as an AND gate shown below:



7.3.1.2 Conditional Signal Assignment

The conditional signal assignment assigns waveforms to a target signal based on the validity of a condition. Its syntax is:

```
target_signal <= waveform WHEN condition ELSE
                 waveform;
```



Example: The following code implements a two-to-one multiplexer (MUX21)

```
1       LIBRARY IEEE;
2       USE IEEE.STD_LOGIC_1164.ALL;
3       ENTITY mux21 IS
4               PORT(a,b,s :IN BIT; o :OUT BIT);
5       END mux21;
6       ARCHITECTURE  archmux21 OF mux21 IS
7       BEGIN
8               o <= a WHEN (s='0') ELSE
9                       b;
10      END archmux21
```

9.3.1.3 Selected Signal Assignment

The selected signal assignment statement assigns waveforms to a target signal based on the value of the included expression. Its syntax is:

```
WITH expression SELECT
        target_signal <= waveform_1 WHEN expression_value_1,
                         waveform_2 WHEN expression_value_2,
                         waveform_n WHEN expression_value_n;
```

You can use any number of WHEN clauses, but no two WHEN clauses can have the same expression value. You must account for all possible values of “expression”. You can use a WHEN OTHERS clause for all unspecified elements and “don’t care” situations.

The following code implements a two-to-one multiplexer (MUX21). Since s is a STD_LOGIC type of signal it has four possible states. WHEN OTHERS is used to accouunt for other values of s.

Example: Multiplexer two-to-one using selected signal assignment.

```
1       LIBRARY IEEE;
```

```
2       USE IEEE.STD_LOGIC_1164.ALL;
3       ENTITY mux21 IS
4               PORT(a, b, s :IN STD_LOGIC; o :OUT STD_LOGIC);
5       END mux21;
6       ARCHITECTURE archmux21 OF mux21 IS
7       BEGIN
8               WITH  s  SELECT
9                     o <= a WHEN ‘0’,
10                         b WHEN ‘1’,
11                        ‘X’ WHEN OTHERS
12      END archmux21;
```

Example: three to eight decoder

```
1.  LIBRARY IEEE;
2.  USE IEEE.STD_LOGIC_1164.ALL;
3.
4.  ENTITY Decoder IS
5.      GENERIC(Delay:TIME:=5 ns);
6.      PORT(Sel:IN BIT_VECTOR(2 DOWNTO 0);
7.      DOut:OUT BIT_VECTOR(7 DOWNTO 0);
8.  END Decoder;
9.
10. ARCHITECTURE  behav OF Decoder IS
11. BEGIN
12. WITH Sel SELECT
13. DOut <=
14.     "00000001" AFTER Delay WHEN "000",
15.     "00000010" AFTER Delay WHEN "001",
16.     "00000100" AFTER Delay WHEN "010",
17.     "00001000" AFTER Delay WHEN "011",
18.     "00010000" AFTER Delay WHEN "100",
19.     "00100000" AFTER Delay WHEN "101",
20.     "01000000" AFTER Delay WHEN "110",
21.     "10000000" AFTER Delay WHEN "111",
22. END behav;
```

9.3.2 IF Statements

- Provides conditional control of sequential statements.
- Condition in statement must evaluate to a Boolean value.
- Execution of statements with IF occurs when condition is TRUE.
- IF statements can only be used in sequential areas of the model.
- Format:

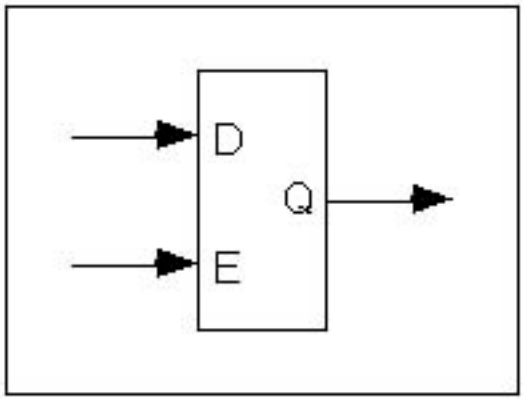
```
IF condition THEN
  -- sequential statements
END IF;

IF condition  THEN
  -- sequential statements
ELSE
  -- sequential statements
END IF;
```

```
IF condition THEN
  -- sequential statements
ELSIF condition THEN
  --sequential statements
--------
ELSE
END IF;
```

Example 1: Transparent Latch Level Sensitive Latch

Is a latch which is sensitive signal level and not to clock edge.

INPUTS		OUTPUT
D	E	Q(t+1)
X	L	Q(t)
L	H	L
H	H	H




```
1       LIBRARY IEEE;
2       USE IEEE.STD_LOGIC_1164.ALL;
3
4       ENTITY dltch IS
5               PORT(d, e : IN BIT; q : OUT BIT);
6       END dltch;
7
8       ARCHITECTURE arch1_dltch OF dltch IS
9       BEGIN
10          PROCESS(d, e)
11              IF (e = '1') THEN
12                      q <= d;
13              END IF;
14          END PROCESS;
15      END arch2_dltch;
```

In this example, the process latch is sensitive to two signals d and e. The target signal q is only assigned to when e = '1'. If e changes to a '0', q must retain its state even if d changes. That is, the simple IF statement does not contain enough information to synthesize a combinational network. What happen to the target

signal when the condition is false is not specified. Autologic VHDL assumes that the target signal will retain its old value. The result is the simple IF statement is synthesize as transparent latch.

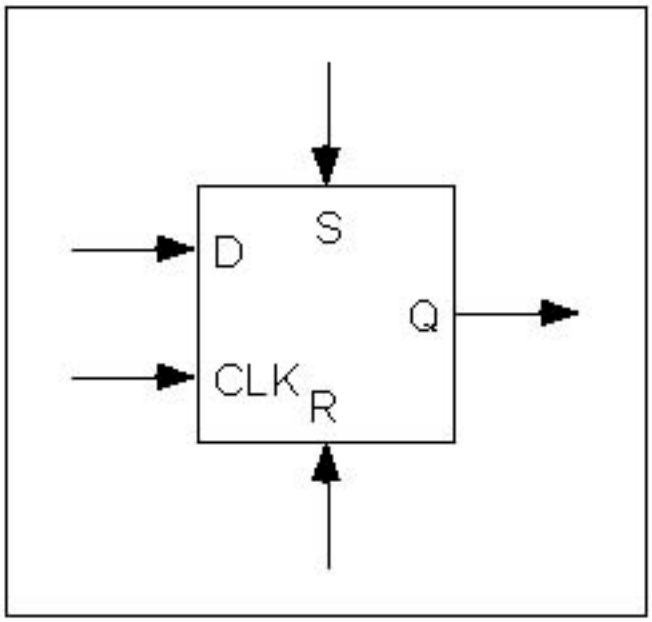
In order for AutoLogic VHDL to synthesize a combinational network using an if statement, the if statement must explicitly define the behavior of all possible evaluations of the condition. The addition of the ELSE clause completes the definition of the behavior of an if statement. In this example, the synthesized result is a combinatorial 2-to-1 multiplexer.

Example 2: Multiplexer implemented as combinational circuit, using IF-THEN-ELSE

```
1       LIBRARY IEEE;
2       USE IEEE.STD_LOGIC_1164.ALL;
3
4       ENTITY mux21 IS
5               PORT(a, b, s: IN BIT; o: OUT BIT);
6       END mux21;
7
8       ARCHITECTURE  archmux21 OF mux21 IS
9       BEGIN
10              PROCESS(a,b,s)
11              BEGIN
12                      IF(s='1') THEN
13                              o<=a;
14                      ELSE
15                              o<=b;
16                      END IF;
17              END PROCESS;
18      END archmux21;
```

Example 3: D-Type Edge Triggered Flip-Flop with Set and Reset

INPUTS				OUTPUT
D	S	R	CLK	Q
X	X	H	X	L
X	H	L	X	H
L	L	L	^	L
H	L	L	^	H




```
1       LIBRARY IEEE;
2       USE IEEE.STD_LOGIC_1164.ALL;
3
4       ENTITY dsrff  IS
5             PORT(d, s, r, clk : IN BIT; q : OUT BIT);
6       END dsrff;
 7
8       ARCHITECTURE arch1_dsrff OF dsrff IS
9       BEGIN
10      latch:     PROCESS(d, s, r, clk)
11                 BEGIN
12                       IF (r = '1') THEN
13                             q <= '0';
 14                      ELSIF (s = '1') THEN
 15                            q <= '1';
 16                      ELSIF (clk'EVENT AND clk = '1') THEN
 17                            q <= d;
 18                      END IF;
 19                END PROCESS;
 20       END arch1_dsrff;
```

Example 4: Algorithmic and_gate

```
1       LIBRARY IEEE, ARITHMETIC;
2       USE IEEE.STD_LOGIC_1164.ALL;
3       USE ARITHMETIC.STD_LOGIC_ARITH.ALL;
4
5       ENTITY and_gate IS
6            PORT(a,b:IN STD_LOGIC;c:OUT STD_LOGIC);
7       END and_gate;
8
9       ARCHITECTURE algorithm OF and_gate IS
10      BEGIN
```

```
11      and_process:
12      PROCESS(a,b)
13      BEGIN
14              IF a='1' AND b='1' THEN
15                      c <= `1' AFTER 10 ns;
16                  ELSIF (a='0') OR (b='0') THEN
17                      c <= `0' AFTER 10 ns;
18                  ELSE
19                      c <= `X' AFTER 5 ns;
20              END IF;
21      END PROCESS and_process;
22      END algorithm;
```

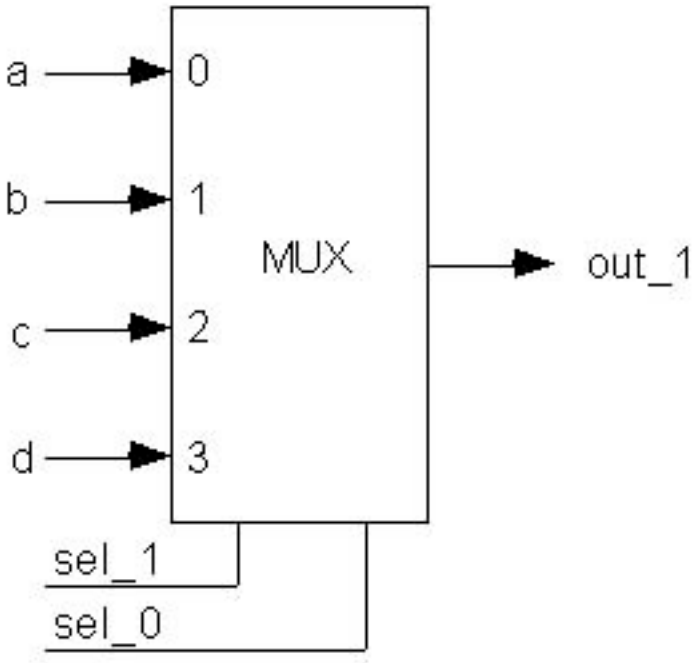
Example 5: Oscillator: switch back and forth between two states

```
1       LIBRARY IEEE;
2       USE IEEE.STD_LOGIC_1164.ALL;
3
4       ENTITY clock IS
5          PORT(clk,clk_bar:OUT STD_LOGIC);
6       END clock;
7
8       ARCHITECTURE algorithm OF clock IS
9       BEGIN
10         PROCESS
11              VARIABLE switch:BIT:='0';
12            BEGIN
13              IF switch='0' THEN
14                      clk <= `0';
15                      clk_bar <= `1';
16                  ELSE
17                      clk <= `1';
18                      clk_bar <= `0';
19              END IF;
20              switch:=NOT switch;
21              WAIT FOR 10 ns;
22      END PROCESS;
23      END algorithm;
```

Example6: one of four multiplexer

```
1       LIBRARY IEEE;
2       USE IEEE.STD_LOGIC_1164.ALL;
3
4       ENTITY mux41 IS
5          PORT(sel_0, sel_1, a, b, c, d: IN STD_LOGIC; out_1:OUT STD_LOGIC);
6       END mux41
7
8       ARCHITECTURE behav OF multiplexer IS
9       BEGIN
10      one_of_four:
11      PROCESS(sel_0,sel_1,a,b,c,d)
12      BEGIN
13              IF sel_1='0' THEN
14                  IF sel_0 = `0' THEN
15                      out_1 <= a;
```

```
16                      ELSE
17                              out_1 <= b;
18                      END IF;
19              ELSIF sel_1 = `1' THEN
20                      IF sel_0 = `0' THEN
21                              out_1 <= c;
22                      ELSE
23                              out_1 <= d;
24                      END IF;
25              END IF;
26      END PROCESS one_of_four;
27      END behav;
```



9.3.3 GENERATE Statements

Generate statements provide the ability to describe regular and/or slightly irregular structures by automatically generating component instantiations instead of manually writing each instantiation. There are two kinds of generate statements: iteration and conditional. An iterate generate statement is also called a **for generate** statement; a conditional generate statement is also called an **if generate** statement. The **for generate** statement is similar to a **for loop** statement. There are two syntax **for generate** statement:

Syntax 1

```
GenLabel:       --Iterate generate label
FOR  i  IN 7 DOWNTO 0 GENERATE

        --sequential statements

END GENERATE GenLabel;
```

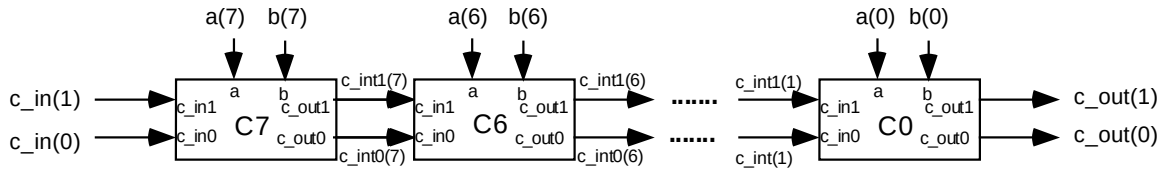
Syntax 2

```
GenLabel:       --Iterate generate label
```

```
FOR I IN 7 DOWNTO 0 GENERATE
        --Declaration statements
BEGIN
        --sequential statements
END GENERATE GenLabel;
```

The **if generate** statement is similar to an if statement. Its syntax is:

```
GenLabel:       --conditional generate label
IF (condition) GENERATE
        --sequential statements
END GENERATE GenLabel;
```




```
LIBRARY work;
USE work.all;
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY compare8 IS
        PORT (a, b : IN BIT_VECTOR(7 DOWNTO 0);
                c_in : IN BIT_VECTOR(1 DOWNRO 0);
                c_out : OUT BIT_VECTOR(1 DOWNTO 0));
END compare8;
```

Manual Structural Modelling

```
ARCHITECTURE man_struct OF compare8 IS
        COMPONENT bit_compare
                PORT(a, b, c_in1, c_in0: IN BIT; c
                        c_out1, c_out0:  OUT BIT));
        END COMPONENT;
        SIGNAL c_int1, c_int0: BIT_VECTOR(7 DOWNTO 1);
        FOR ALL : bit_compare USE ENTITY work.bit_compare(logic);

BEGIN
        C7: bit_compare PORT MAP(a(7), b(7), c_in(1), c_in(0), c_int1(7), c_int0(7));
        C6: bit_compare PORT MAP(a(6), b(6), c_int1(7), c_int0(7), c_int1(6), c_int0(6));
        C5: bit_compare PORT MAP(a(5), b(5), c_int1(6), c_int0(6), c_int1(5), c_int0(5));
        C4: bit_compare PORT MAP(a(4), b(4), c_int1(5), c_int0(5), c_int1(4), c_int0(4));
        C3: bit_compare PORT MAP(a(3), b(3), c_int1(4), c_int0(4), c_int1(3), c_int0(3));
        C2: bit_compare PORT MAP(a(2), b(2), c_int1(3), c_int0(3), c_int1(2), c_int0(2));
        C1: bit_compare PORT MAP(a(1), b(1), c_int1(2), c_int0(2), c_int1(1), c_int0(1));
        C0: bit_compare PORT MAP(a(0), b(0), c_int1(1), c_int0(1), c_out(1), c_out(0));
END man_struct;
```

Generated Structural Modelling

```
ARCHITECTURE gen_struct OF compare8 IS
        COMPONENT bit_compare
```

```
            PORT(a, b, c_in1, c_in0: IN BIT; c
                  c_out1, c_out0:  OUT BIT));
      END COMPONENT;
      SIGNAL c_int1, c_int0: BIT_VECTOR(7 DOWNTO 1);
BEGIN
      CASCADE:      --Iteration generate
      FOR i IN 7 DOWNTO 0 GENERATE
      INPUT_CASE:
      IF (i=7) GENERATE
            C7: bit_compare PORT MAP (a(i), b(i), c_in(1), c_in(0), c_int1(i), c_int0(i));
      END GENERATE INPUT_CASE;

      NORMAL_CASE:
      IF(i<=6 AND i>=1) GENERATE
            CX: bit_compare PORT MAP(a(i), b(i), c_int1(i+1), c_int0(i+1), c_int1(i), c_int0(i));
      END GENERATE NORMAL_CASE;

      OUTPUT_CASE:
      IF(I=0) GENERATE
            C0: bit_compare PORT MAP (a(I), b(I), c_int1(I+1), c_int0(I+1), c_out(1), c_out(0));
      END GENERATE OUTPUT CASE;
      END GENERATE CASCADE;
END gen_struct;
```

9.3.4 Case Statements

The case statement controls the execution of one or more sequential statements on the value of an expression. Its syntax is:

```
CASE selector IS
      WHEN selector_value => sequential statements
END CASE;
```

Example 1: one of two multiplexer

```
1     LIBRARY IEEE;
2     USE IEEE.STD_LOGIC_1164.ALL;
3     ENTITY mux21 IS
4           PORT(a, b, s :IN BIT; o : OUT BIT);
5     END mux21;
6     ARCHITECTURE archmux21 OF mux21 IS
7     BEGIN
8           PROCESS(a, b, s)
9           BEGIN
10                CASE s IS
11                      WHEN '0' => o <= a;
12                      WHEN '1' => o <= b;
13                      WHEN OTHERS => o <= 'X';   --or WHEN OTHERS => NULL;
14                END CASE;
15          END PROCESS;
16    END archmux21;
```

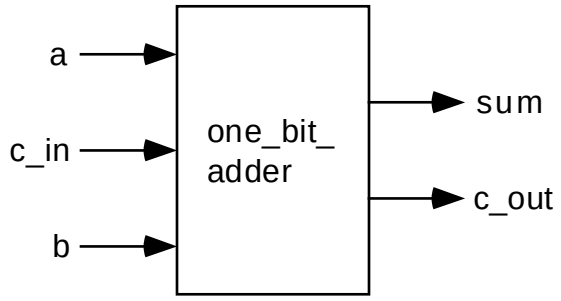
Example 2**:** one of four multiplexer

```
1     LIBRARY IEEE;
```

```
2       USE IEEE.STD_LOGIC_1164.ALL;
3       ENTITY mux41 IS
4            PORT(sel_0, sel_1, a, b, c, d :IN STD_LOGIC; out_1 : OUT STD_LOGIC);
5       END mux41;
6
7       ARCHITECTURE archmux41 OF mux41 IS
8       BEGIN
9       one_of_four:
10      PROCESS(sel_0,sel_1,a,b,c,d)
11      BEGIN
12           CASE sel_1 & sel_0 IS
13                WHEN "00" => out_1 <= a;
14                WHEN "01" => out_1 <= b;
15                WHEN "10" => out_1 <= c;
16                WHEN "11" => out_1 <=d;
17                WHEN OTHERS => NULL;
18           END CASE;
19      END PROCESS one_of_four
20      END example;
```

Example 3**:** one_bit_adder

```
1       LIBRARY IEEE;
2       USE IEEE.STD_LOGIC_1164.ALL;
3
4       ENTITY one_bit_adder IS
5            PORT(a,c_in,b:STD_LOGIC;sum,c_out:OUT STD_LOGIC);
6       END one_bit_adder;
7
8       ARCHITECTURE behav OF one_bit_adder IS
9            SIGNAL total:INTEGER RANGE 0 TO 4;
10      BEGIN
11      majority: WITH a & c_in & b SELECT
12                total <= 0 WHEN "000",
13                          1 WHEN "100"|"010"|"001",
14                          2 WHEN "101"|"011"|"110",
15                          3 WHEN "111",
16                          4 WHEN OTHERS;
17        adder: PROCESS(total)
18               BEGIN
19                    CASE total IS
20                         WHEN 0 => sum <='0'; c_out <='0';
21                         WHEN 1 => sum <='1'; c_out <='0';
22                         WHEN 2 => sum <='0'; c_out <='1';
23                         WHEN 3 => sum <='1'; c_out <='1';
24                         WHEN OTHERS => sum <='X'; c_out <='X';
25                    END CASE;
26               END PROCESS adder;
27      END behav;
```



- `&' Concatenation symbol
- OTHERS indicates all unspecified elements.
- `|' means or
- NULL is a sequential statement indicating no action occurs