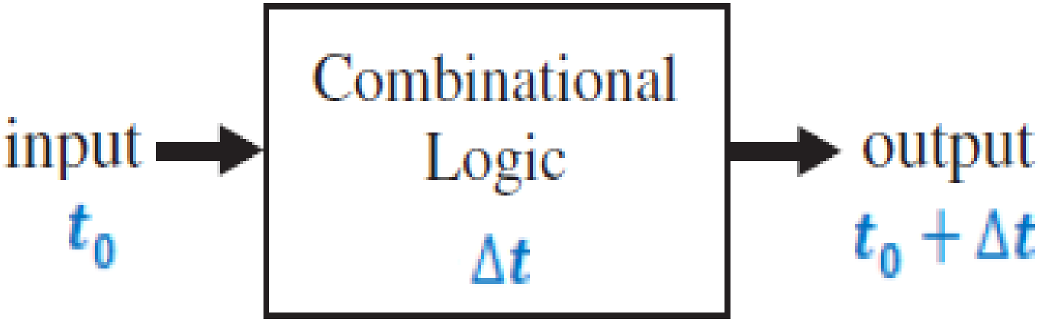
VHDL - Concurrent Code Dataflow modeling

©Hanan Ribo

Introduction

- VHDL code can be concurrent (combinational logic) or sequential (sequential logic).
- **Combinational logic - definition**:

  The output is a pure function of the present input only (implemented by Boolean circuits, using conventional logic gates only – no memory, no feedback).
- Intuitively, the circuit information flows in parallel.

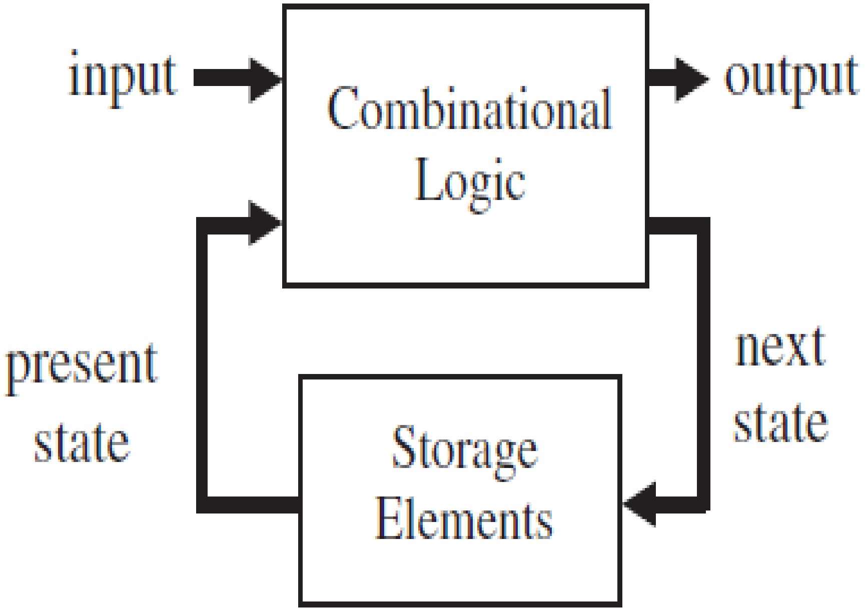


©Hanan Ribo

Introduction

- **Sequential logic - definition**:

  The output does depend on present inputs and previous inputs (implemented using storage, flip-flops elements, which are connected to the combinational logic block through a feedback loop).
- Intuitively, the circuit information flows in serial triggered by clk signal.

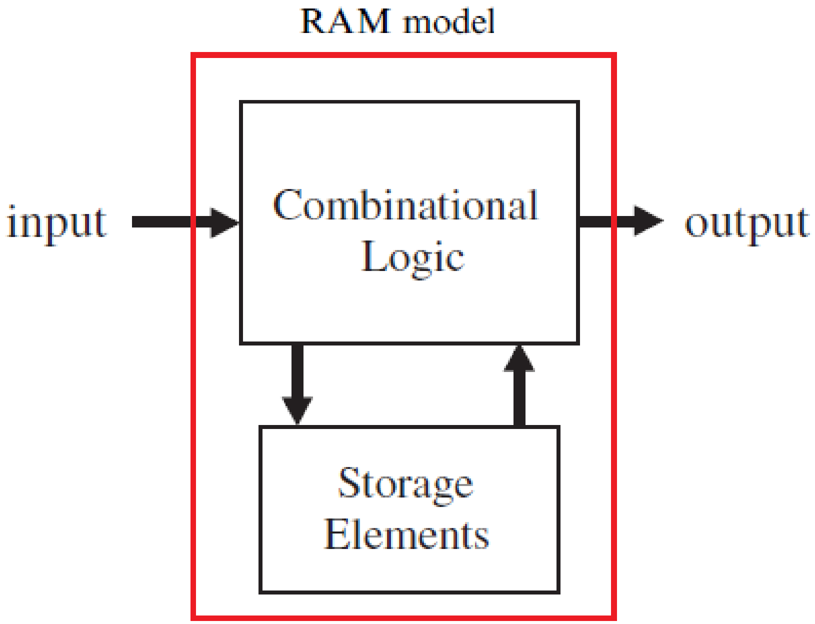


©Hanan Ribo

Introduction

- **Note**: not any circuit that possesses storage elements is sequential.
- Example: RAM memory.

  The storage elements appear in a forward path rather than in a feedback loop. The memory-read operation depends only on the present address vector input (with nothing to do with previous memory accesses).



©Hanan Ribo

Introduction

- There are four types of **ARCHITECTURE** Modeling styles:

✓ Dataflow modeling = Concurrent Code

✓ Structural modeling = Concurrent Code

✓ Behavioral modeling = Sequential Code

✓ Mixed modeling = Concurrent Code

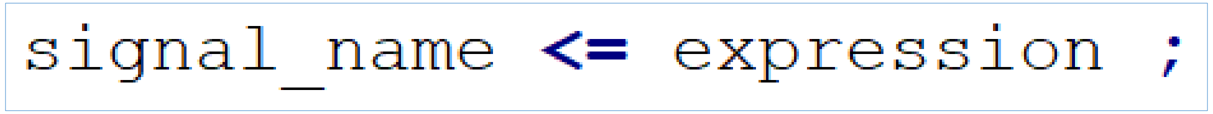
Concurrent code - Dataflow

- Concurrent code dataflow approach is based on four kind of statements that can only be used outside **PROCESSES**, **FUNCTIONS**, or **PROCEDURES** (Three sequential code mechanism)**.**
  - ✓ Signal assignment using **Operators** (with no feedback)
  - ✓ The **WHEN** statement (two kinds - **WHEN/ELSE** or **WITH/SELECT/WHEN**)
  - ✓ The **GENERATE** statement
  - ✓ The **BLOCK** statement
- **The order of concurrent statements doesn't matter, due to concurrent execution.**
- **Notepad++ Download (highly recommended)**

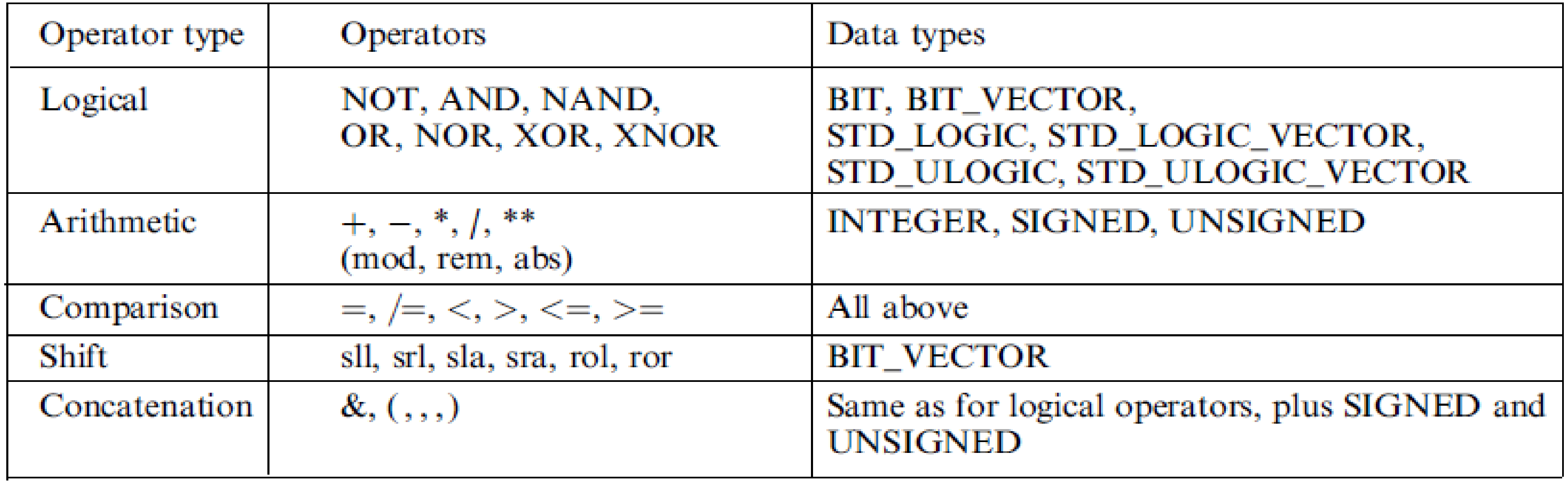
©Hanan Ribo

Signal assignment-using Operators

- Using Operators is the most basic way of creating concurrent code. Operators can be used to implement any combinational circuit.
- **Syntax**:



```
signal_name <= expression ;
```



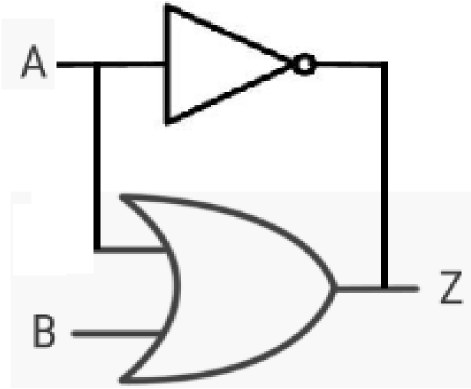
Operator type	Operators	Data types
Logical	NOT, AND, NAND, OR, NOR, XOR, XNOR	BIT, BIT_VECTOR, STD_LOGIC, STD_LOGIC_VECTOR, STD_ULOGIC, STD_ULOGIC_VECTOR
Arithmetic	+, −, *, /, ** (mod, rem, abs)	INTEGER, SIGNED, UNSIGNED
Comparison	=, /=, <, >, <=, >=	All above
Shift	sll, srl, sla, sra, rol, ror	BIT_VECTOR
Concatenation	&, (, , ,)	Same as for logical operators, plus SIGNED and UNSIGNED

- **Note 1**: Signal assignment with feedback causes buffer inferred, in order to save the last signal value.

Signal assignment-using Operators

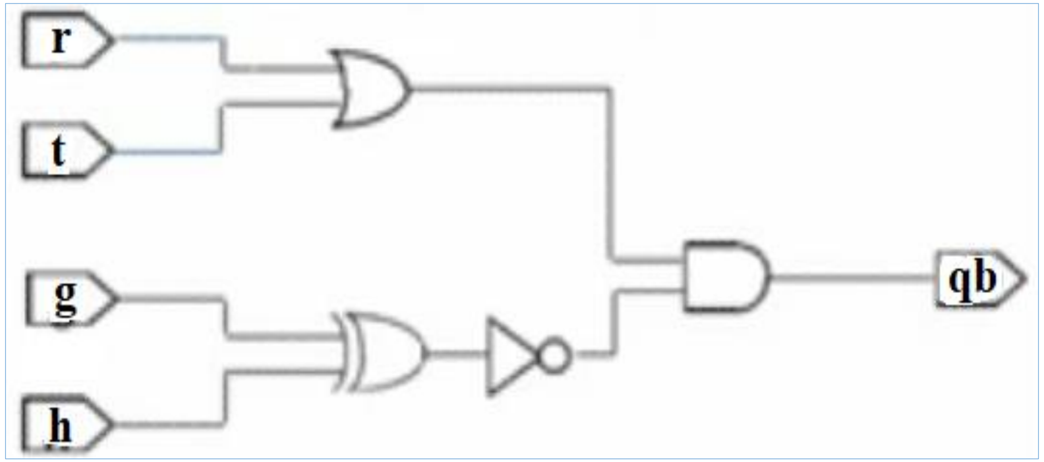
- **Note 2**: concurrent code must not use multiple driven assignments. Its implementation shortcuts Vcc to gnd.

```
Z <= not A;
Z <= A or B;
```



- **Example 1:**

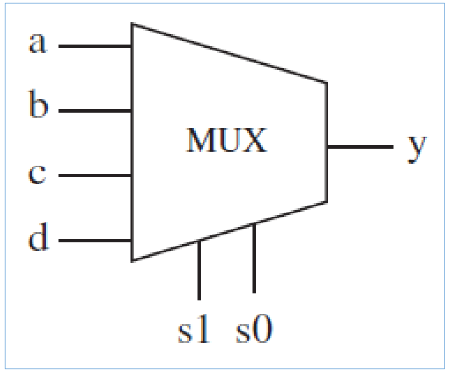
```
qa <= r or t;
qb <= (qa and not(g xor h));
```

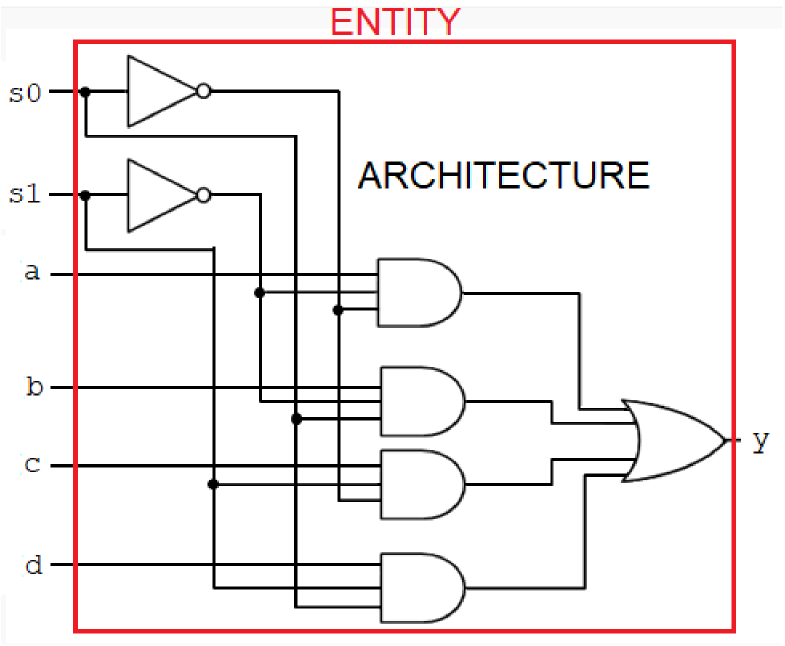


Signal assignment-using Operators (Mux 4-1)

Example 2:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------
ENTITY mux IS
  PORT ( a, b, c, d, s0, s1: IN STD_LOGIC;
                y: OUT STD_LOGIC);
END mux;
---------------------------------------
ARCHITECTURE pureLogic OF mux IS
BEGIN
  y <= (a AND NOT s1 AND NOT s0) OR
       (b AND NOT s1 AND s0) OR
       (c AND s1 AND NOT s0) OR
       (d AND s1 AND s0);
END pureLogic;
```

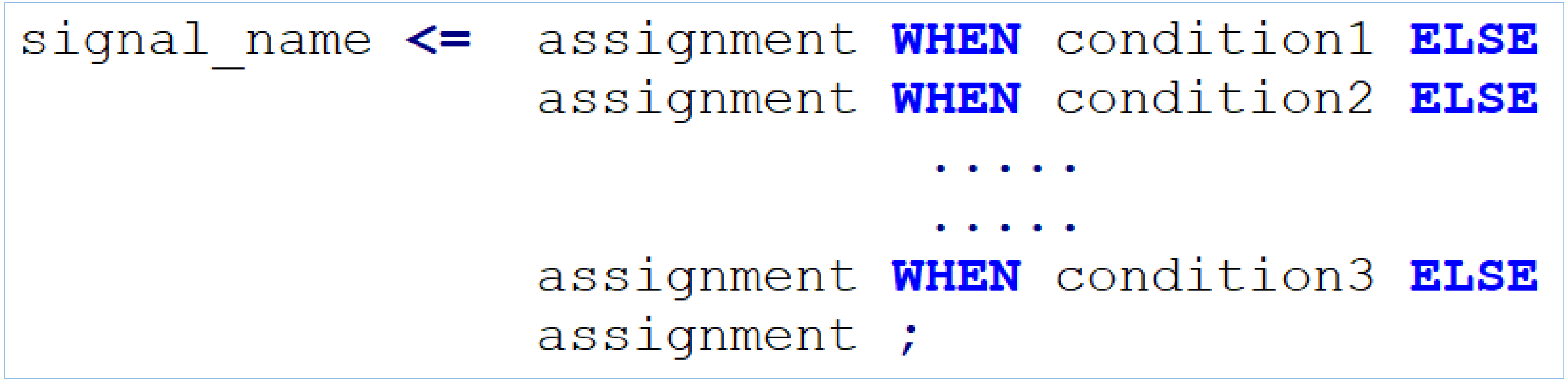




Conditional signal assignment

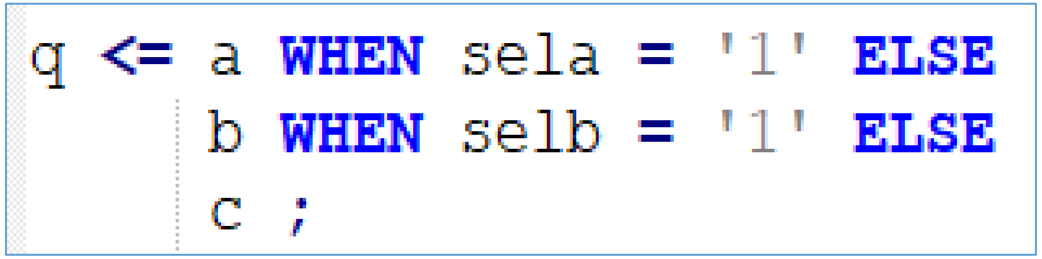
- **Syntax:**

Priority Mux based on 2-1 MUX cascade

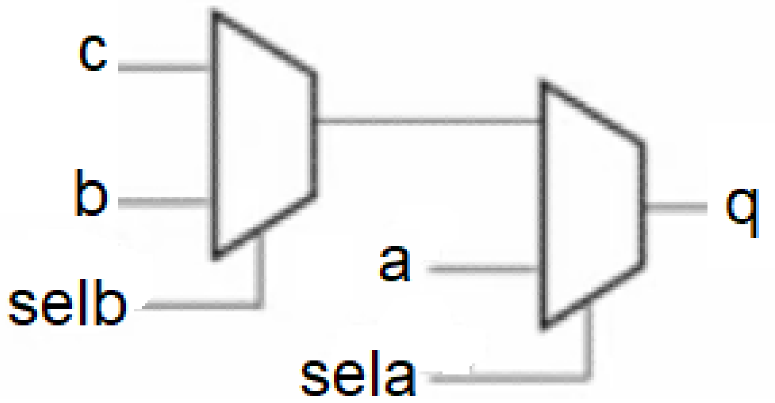


```
signal_name <=  assignment WHEN condition1 ELSE
                assignment WHEN condition2 ELSE
                           .....
                           .....
                assignment WHEN condition3 ELSE
                assignment ;
```

- **Note**: If **ELSE** is not required, this method should not be used.
- **Example 1:**

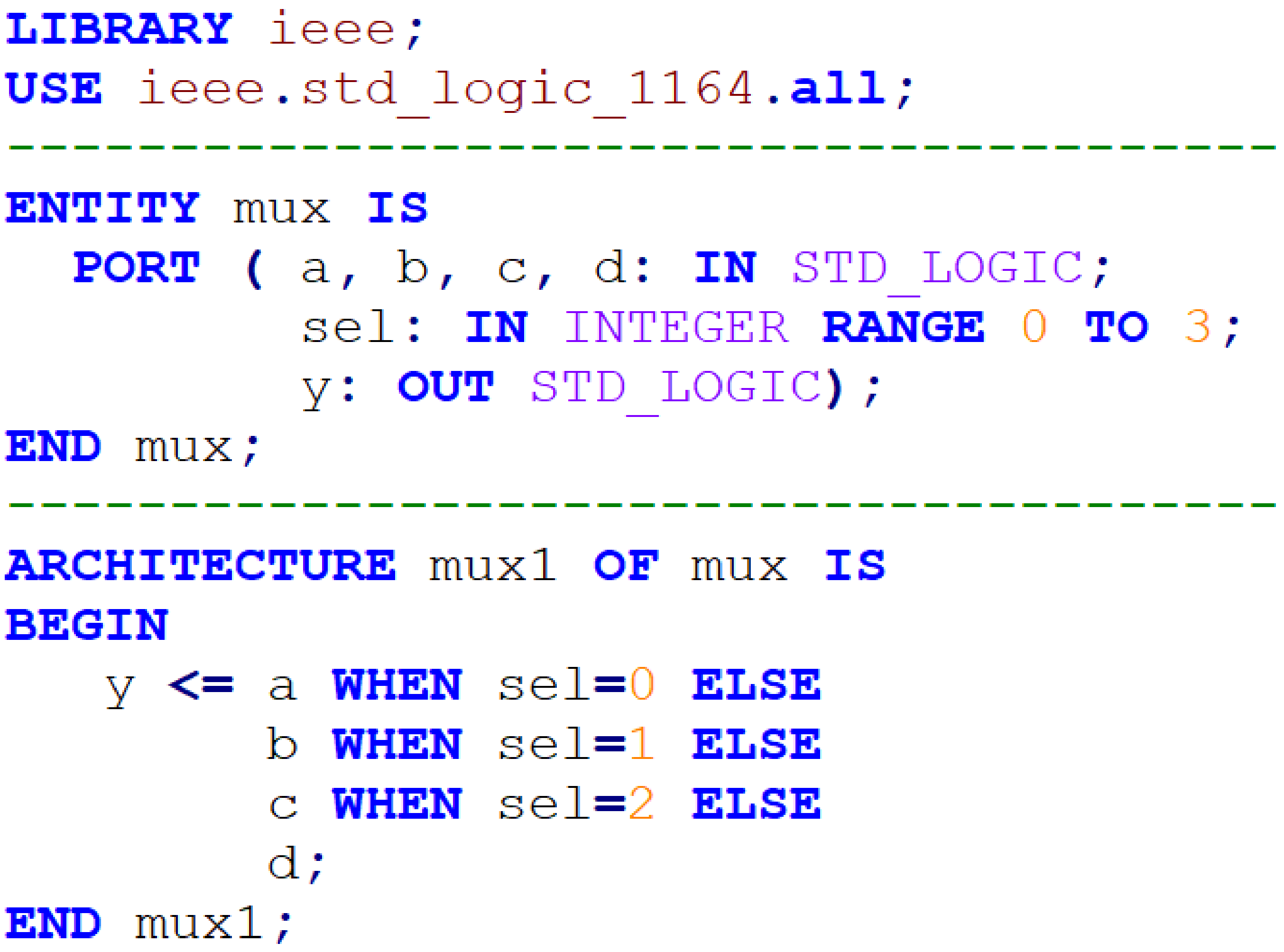


```
q <= a WHEN sela = '1' ELSE
     b WHEN selb = '1' ELSE
     c ;
```

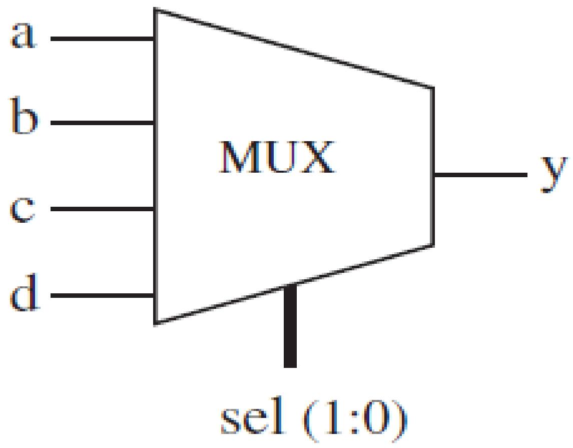


Conditional signal assignment (Mux 4-1)

- **Example 2:**

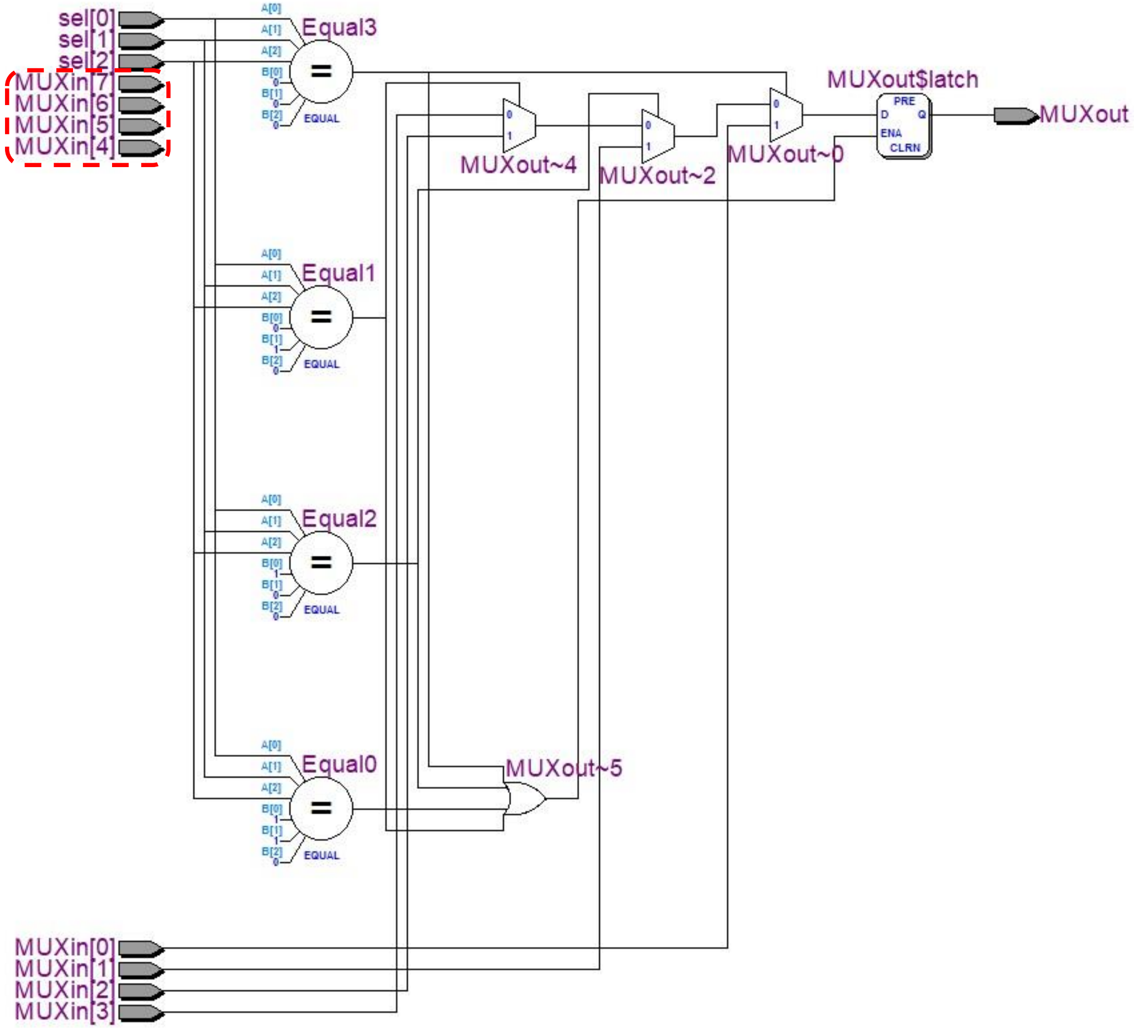


```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------
ENTITY mux IS
  PORT ( a, b, c, d: IN STD_LOGIC;
         sel: IN INTEGER RANGE 0 TO 3;
         y: OUT STD_LOGIC);
END mux;
---------------------------------------
ARCHITECTURE mux1 OF mux IS
BEGIN
   y <= a WHEN sel=0 ELSE
        b WHEN sel=1 ELSE
        c WHEN sel=2 ELSE
        d;
END mux1;
```



unaffected *value using in WHEN statement*

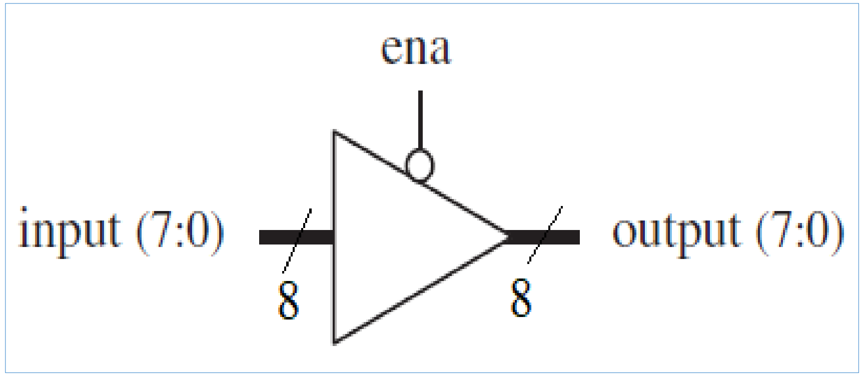
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------
ENTITY mux3_8 IS
    GENERIC (n : INTEGER := 8;
             k : integer := 3  -- k=log2(n)
             );

    PORT (MUXin:  IN STD_LOGIC_VECTOR(n-1 downto 0);
          sel:    IN STD_LOGIC_VECTOR(k-1 downto 0);
          MUXout: OUT STD_LOGIC);
END mux3_8;
-------------------------------
ARCHITECTURE rtl OF mux3_8 IS
BEGIN

        MUXout <= MUXin(0)  when  sel = "000" else
                  MUXin(1)  when  sel = "001" else
                  MUXin(2)  when  sel = "010" else
                  MUXin(3)  when  sel = "011" else
                  unaffected ; -- when other options

END rtl;
```



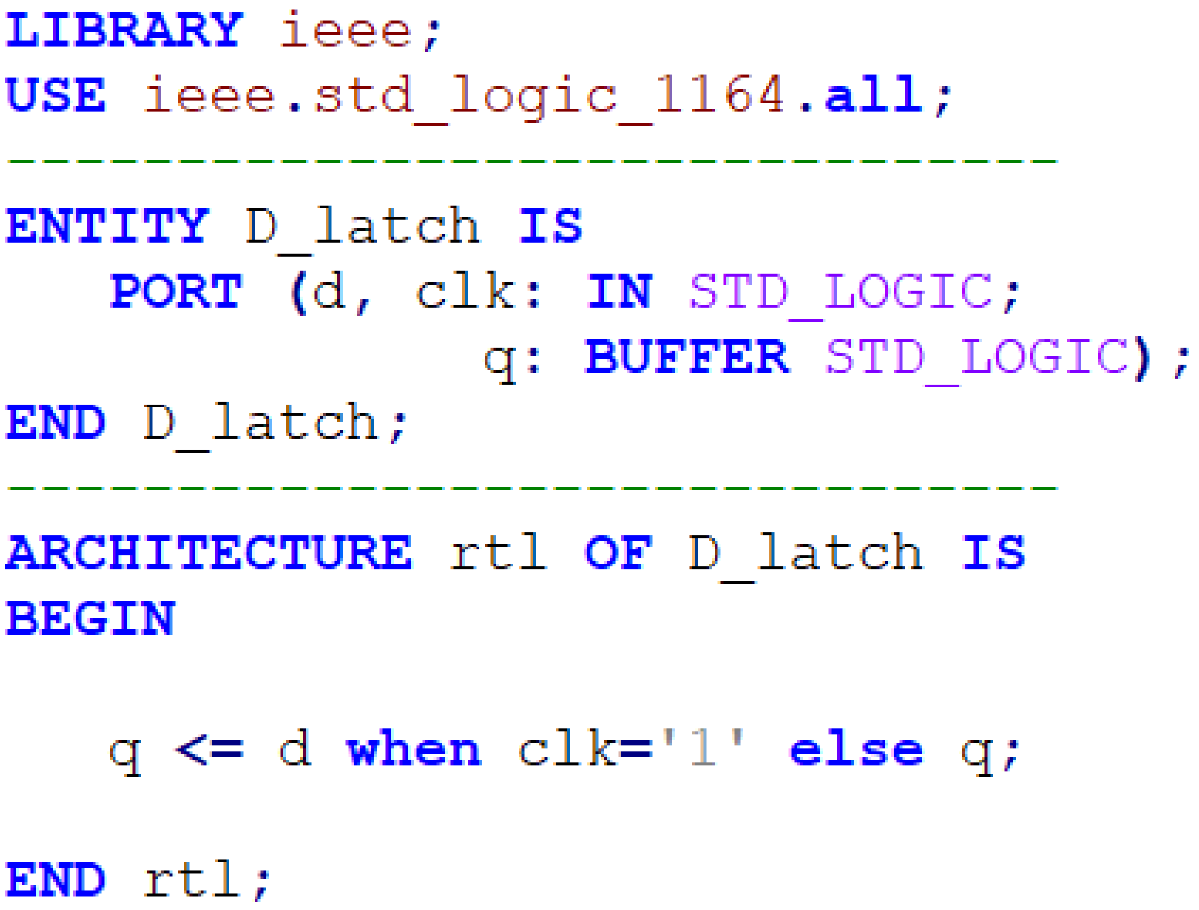
Conditional signal assignment (tri-state)

- **Example 3:**

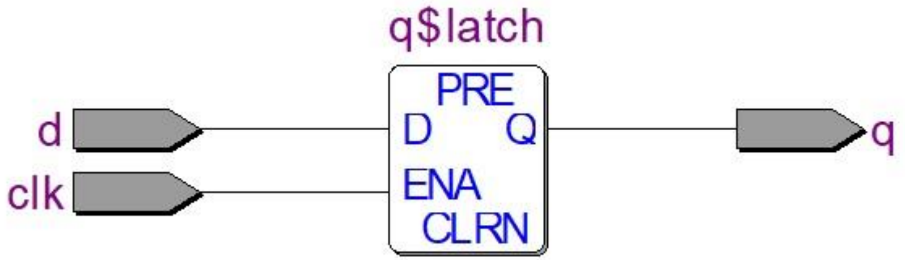
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
----------------------------------------------
ENTITY tri_state IS
  PORT ( ena: IN STD_LOGIC;
        input: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        output: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END tri_state;
----------------------------------------------
ARCHITECTURE tri_state OF tri_state IS
BEGIN
   output <= input WHEN (ena='0') ELSE
             (OTHERS => 'Z');
END tri_state;
```



***D latch** using WHEN statement*

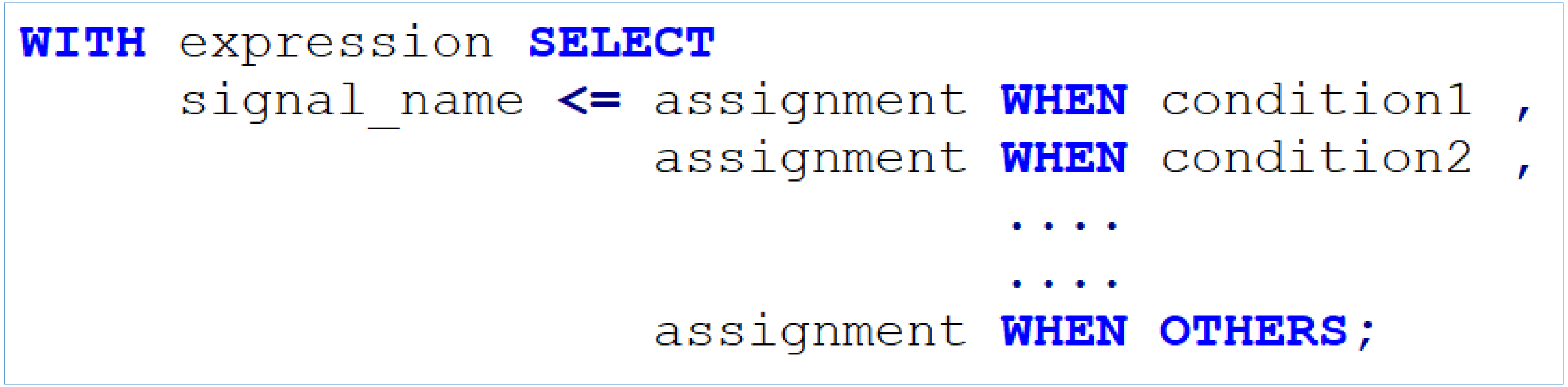


```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
----------------------------------------
ENTITY D_latch IS
   PORT (d, clk: IN STD_LOGIC;
              q: BUFFER STD_LOGIC);
END D_latch;
----------------------------------------
ARCHITECTURE rtl OF D_latch IS
BEGIN

   q <= d when clk='1' else q;

END rtl;
```



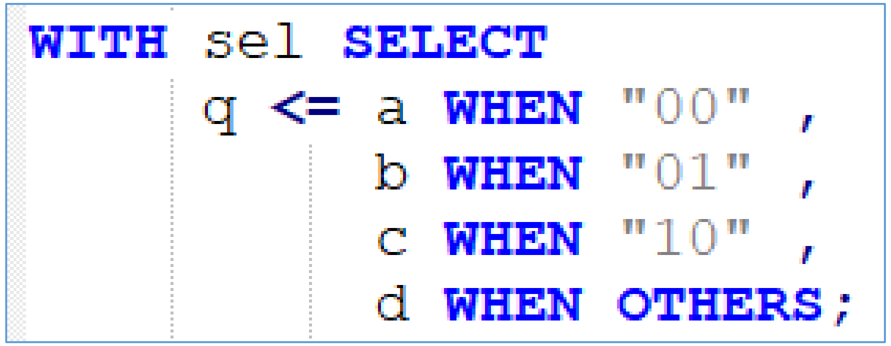
Selected signal assignment (mux as a True Table)

- **Syntax:**

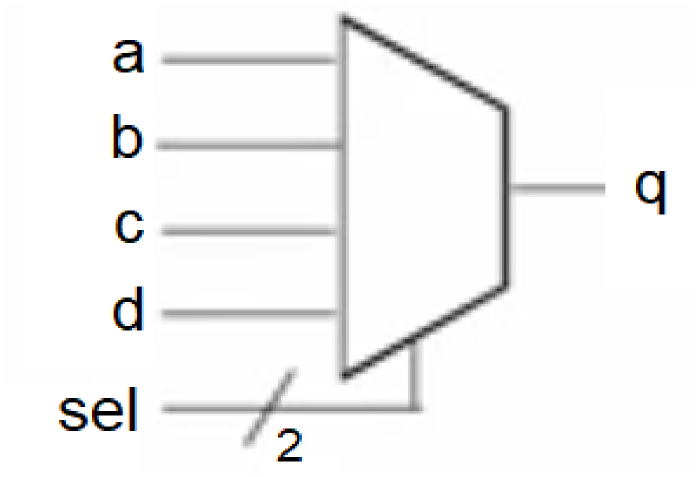


```
WITH expression SELECT
     signal_name <= assignment WHEN condition1 ,
                    assignment WHEN condition2 ,
                               ....
                               ....
                    assignment WHEN OTHERS;
```

- **Example 1:**



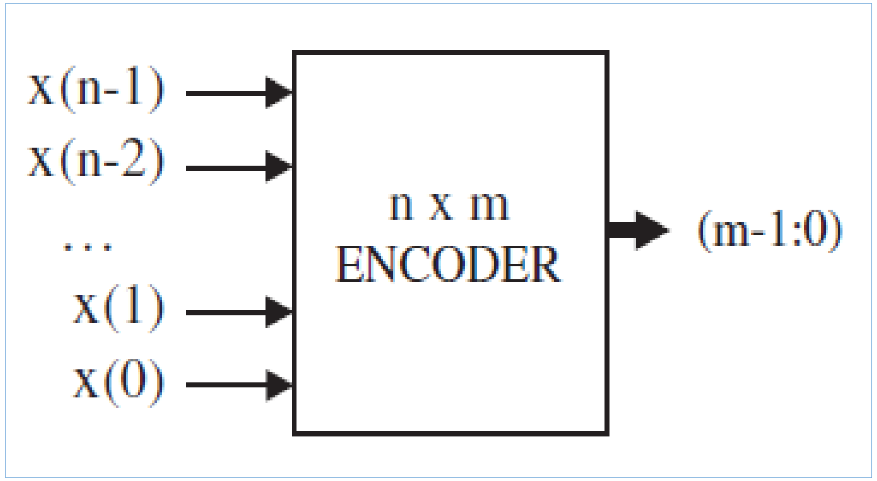
```
WITH sel SELECT
     q <= a WHEN "00" ,
          b WHEN "01" ,
          c WHEN "10" ,
          d WHEN OTHERS;
```



Selected signal assignment (mux as a True Table)

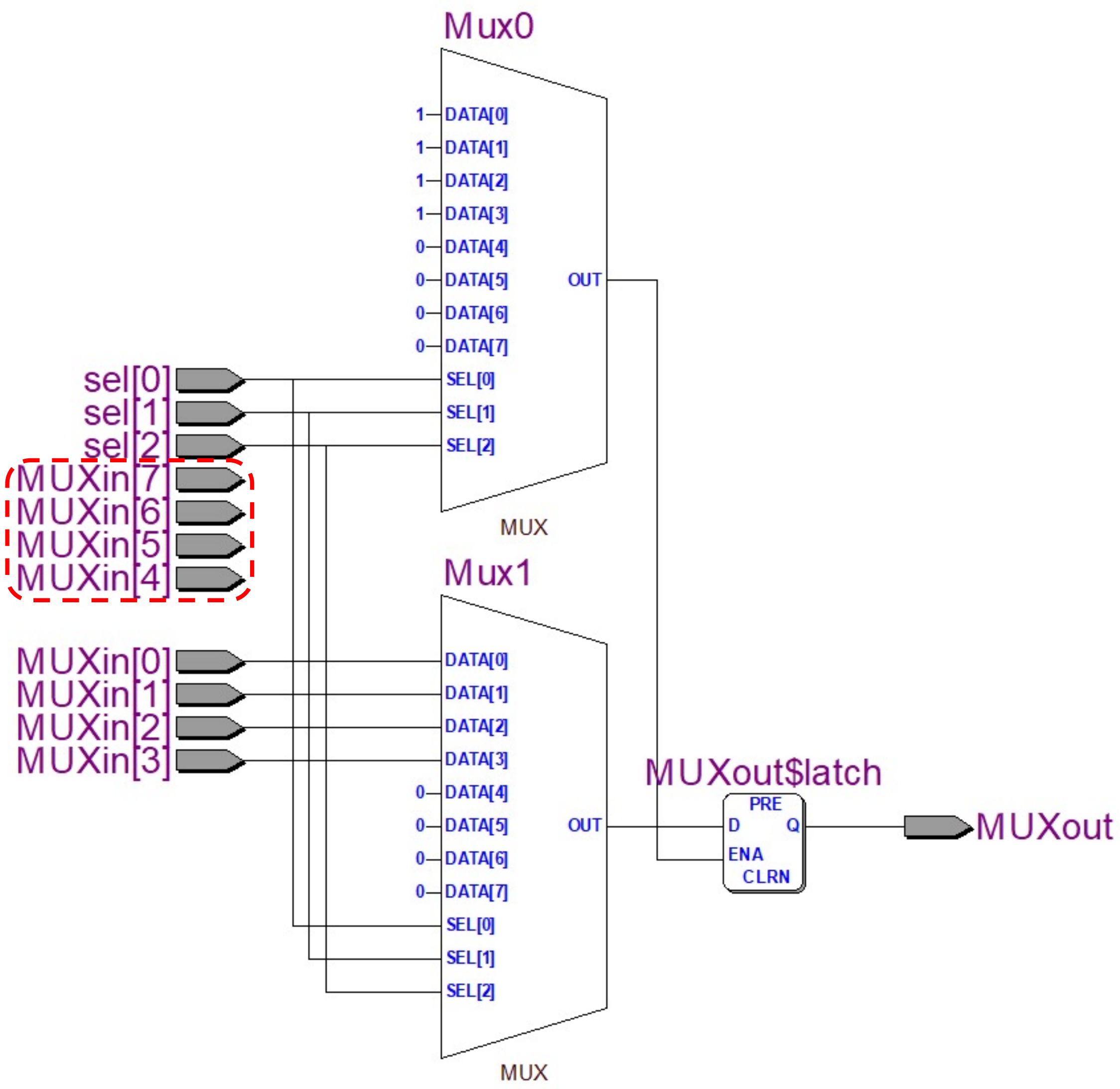
Example 2:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------------
ENTITY encoder IS
  PORT ( x: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
         y: OUT STD_LOGIC_VECTOR (2 DOWNTO 0));
END encoder;
---------------------------------------------
ARCHITECTURE encoder2 OF encoder IS
BEGIN
  WITH x SELECT
    y <= "000" WHEN "00000001",
         "001" WHEN "00000010",
         "010" WHEN "00000100",
         "011" WHEN "00001000",
         "100" WHEN "00010000",
         "101" WHEN "00100000",
         "110" WHEN "01000000",
         "111" WHEN "10000000",
         "ZZZ" WHEN OTHERS;
END encoder2;
```



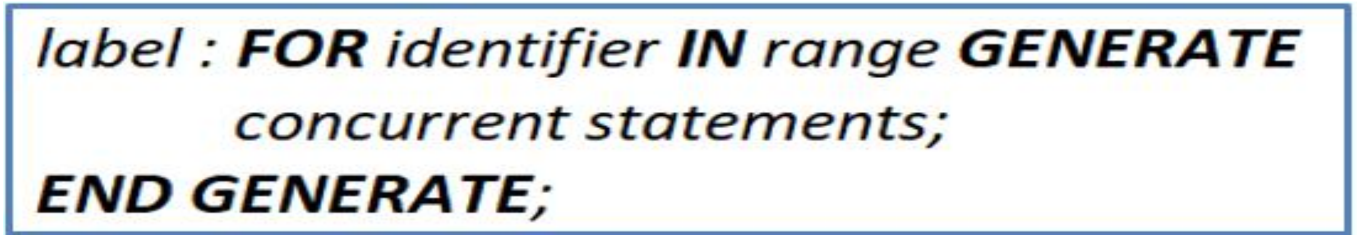
unaffected value using in *With-Select* statement

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------
ENTITY mux3_8 IS
    GENERIC (n : INTEGER := 8;
             k : integer := 3   -- k=log2(n)
             );

    PORT (MUXin:   IN STD_LOGIC_VECTOR(n-1 downto 0);
          sel:     IN STD_LOGIC_VECTOR(k-1 downto 0);
          MUXout:  OUT STD_LOGIC);
END mux3_8;
-------------------------------
ARCHITECTURE rtl OF mux3_8 IS
BEGIN

    with sel(2 downto 0) select -- sel must be static
        MUXout <= MUXin(0)   when  "000",
                  MUXin(1)   when  "001",
                  MUXin(2)   when  "010",
                  MUXin(3)   when  "011",
                  unaffected when others; -- when oth

END rtl;
```



Generate

- **GENERATE** is another concurrent statement. *It allows a section of concurrent code to be repeated a number of times,* thus creating several instances of the same assignments.
- **GENERATE** statement must be labeled.
- **Formula 1 - FOR / GENERATE:**



```
label : FOR identifier IN range GENERATE
            concurrent statements;
END GENERATE;
```

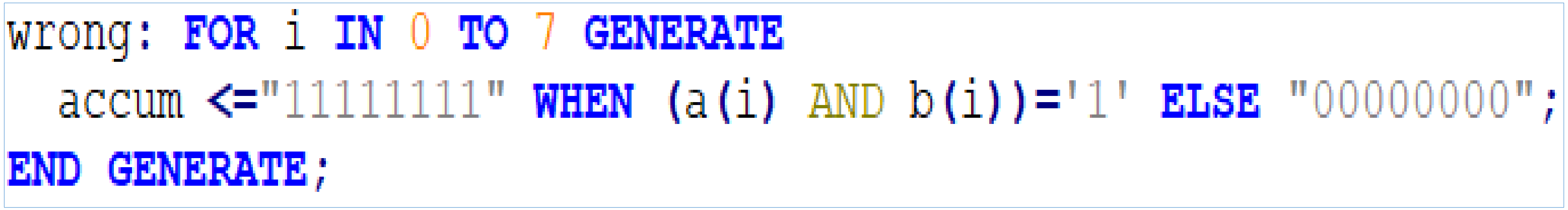
Notes:

- ➢ the identifier range must be static.
- ➢ Be aware of avoiding from overlap assignments (multiple driven), causes compilation error.

©Hanan Ribo

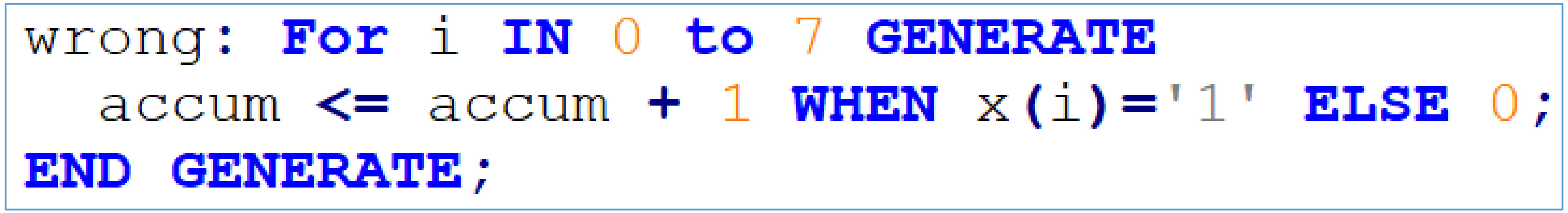
FOR / GENERATE – multiple driven

- **Example 1:**



```
wrong: FOR i IN 0 TO 7 GENERATE
  accum <="11111111" WHEN (a(i) AND b(i))='1' ELSE "00000000";
END GENERATE;
```

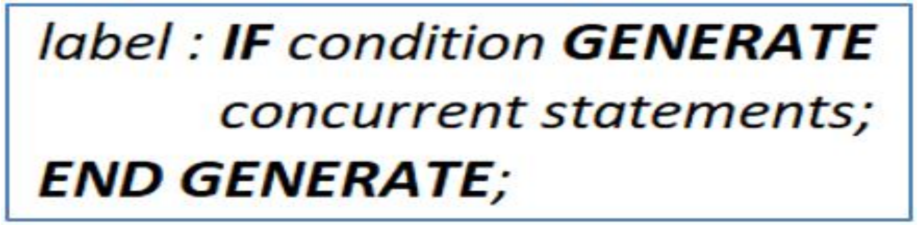
- **Example 2:**



```
wrong: For i IN 0 to 7 GENERATE
   accum <= accum + 1 WHEN x(i)='1' ELSE 0;
END GENERATE;
```

Generate

- **Formula 2 - IF / GENERATE :**



```
label : IF condition GENERATE
        concurrent statements;
END GENERATE;
```

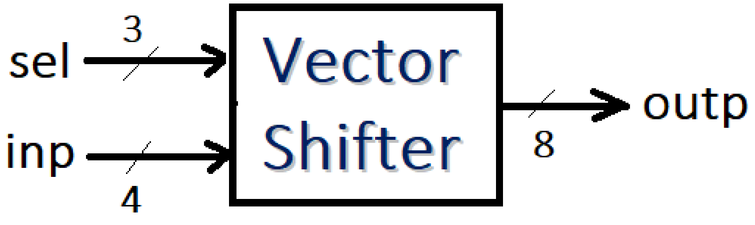
- **Formula 3 - IF / GENERATE** and **FOR / GENERATE** can be combined**:**

```
label1: IF condition GENERATE
                . . . . .
     label2: FOR identifier IN range GENERATE
       concurrent assignments;
     END GENERATE;
                . . . . .
END GENERATE;
```

```
label1: FOR identifier IN range GENERATE
                . . . . .
     label2: IF condition GENERATE
       concurrent assignments;
     END GENERATE;
                . . . . .
END GENERATE;
```

GENERATE - Vector Shifter Example

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
----------------------------------------------------
ENTITY shifter IS
   PORT ( inp: IN STD_LOGIC_VECTOR (3 DOWNTO 0);
          sel: IN INTEGER RANGE 0 TO 4;
         outp: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END shifter;
----------------------------------------------------
ARCHITECTURE shifter OF shifter IS
   SUBTYPE vector IS STD_LOGIC_VECTOR (7 DOWNTO 0);
   TYPE matrix IS ARRAY (4 DOWNTO 0) OF vector;
   SIGNAL row: matrix;
BEGIN
   row(0) <= "0000" & inp;
   G1: FOR i IN 1 TO 4 GENERATE
       row(i) <= row(i-1)(6 DOWNTO 0) & '0';
   END GENERATE;
   outp <= row(sel); -- Mux inferred (pure logic of Memory structure)
END shifter;
```



For input:
inp="0011"
The result is:
row(0): 0 0 0 0 0 0 1 1 (decimal 3)
row(1): 0 0 0 0 0 1 1 0 (decimal 6)
row(2): 0 0 0 0 1 1 0 0 (decimal 12)
row(3): 0 0 0 1 1 0 0 0 (decimal 24)
row(4): 0 0 1 1 0 0 0 0 (decimal 48)

BLOCK

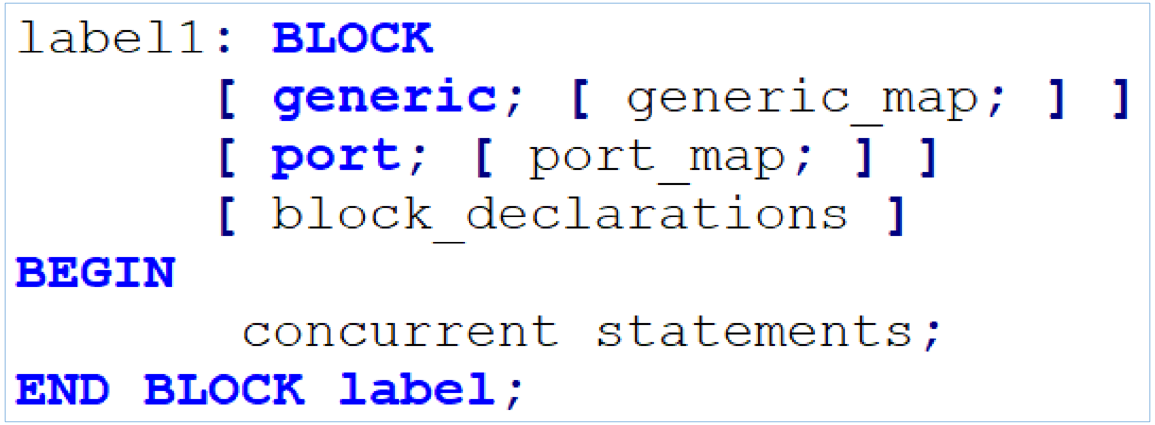
- There are two kinds of BLOCK statements:

  Simple BLOCK and **Guarded BLOCK**.

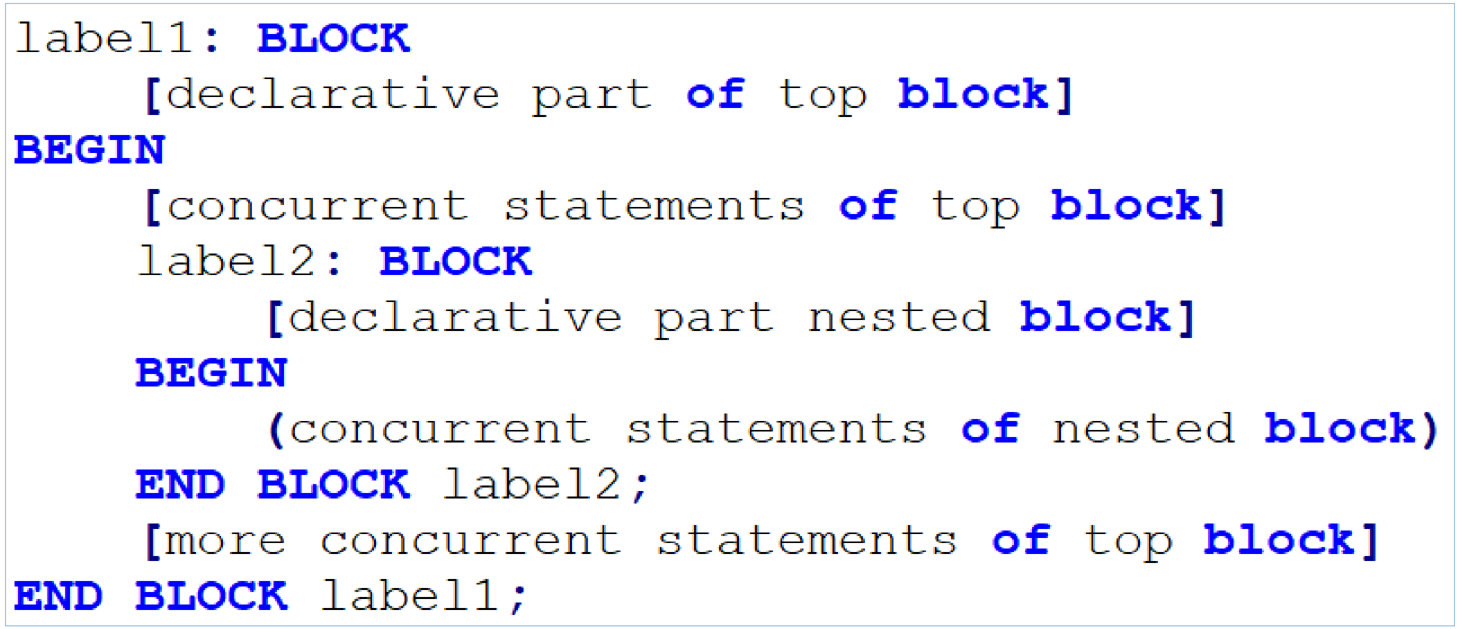
- It allows a set of concurrent statements to be clustered into a **BLOCK**, with *the purpose of turning the overall code more readable and more manageable (helpful when dealing with long codes).*
- Only concurrent statements can be written within a **BLOCK**.
- A **BLOCK** statement must be labeled.
- Declarations inside a **BLOCK** are seen by the **BLOCK** only.
- A **BLOCK** statement is local to the **ARCHITECTURE** where it’s located.
- A **BLOCK** (simple or guarded) can be nested inside another **BLOCK**.

Simple BLOCK

- **Syntax**:



```
label1: BLOCK
        [ generic; [ generic_map; ] ]
        [ port; [ port_map; ] ]
        [ block_declarations ]
BEGIN
         concurrent statements;
END BLOCK label;
```

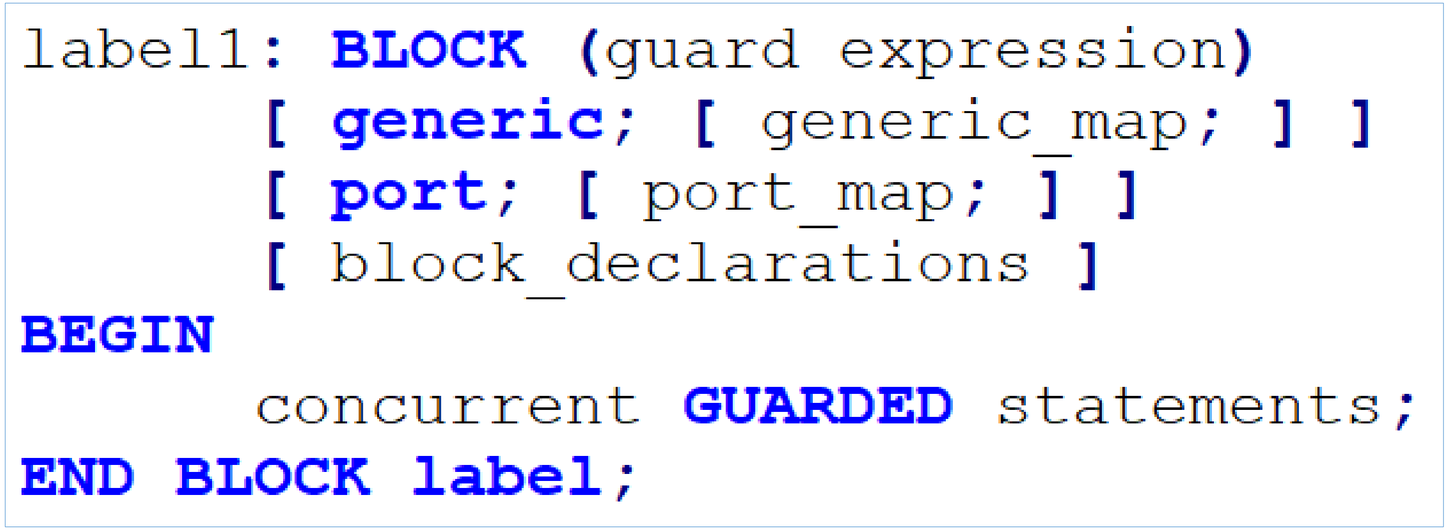


```
label1: BLOCK
    [declarative part of top block]
BEGIN
    [concurrent statements of top block]
    label2: BLOCK
        [declarative part nested block]
    BEGIN
        (concurrent statements of nested block)
    END BLOCK label2;
    [more concurrent statements of top block]
END BLOCK label1;
```

©Hanan Ribo

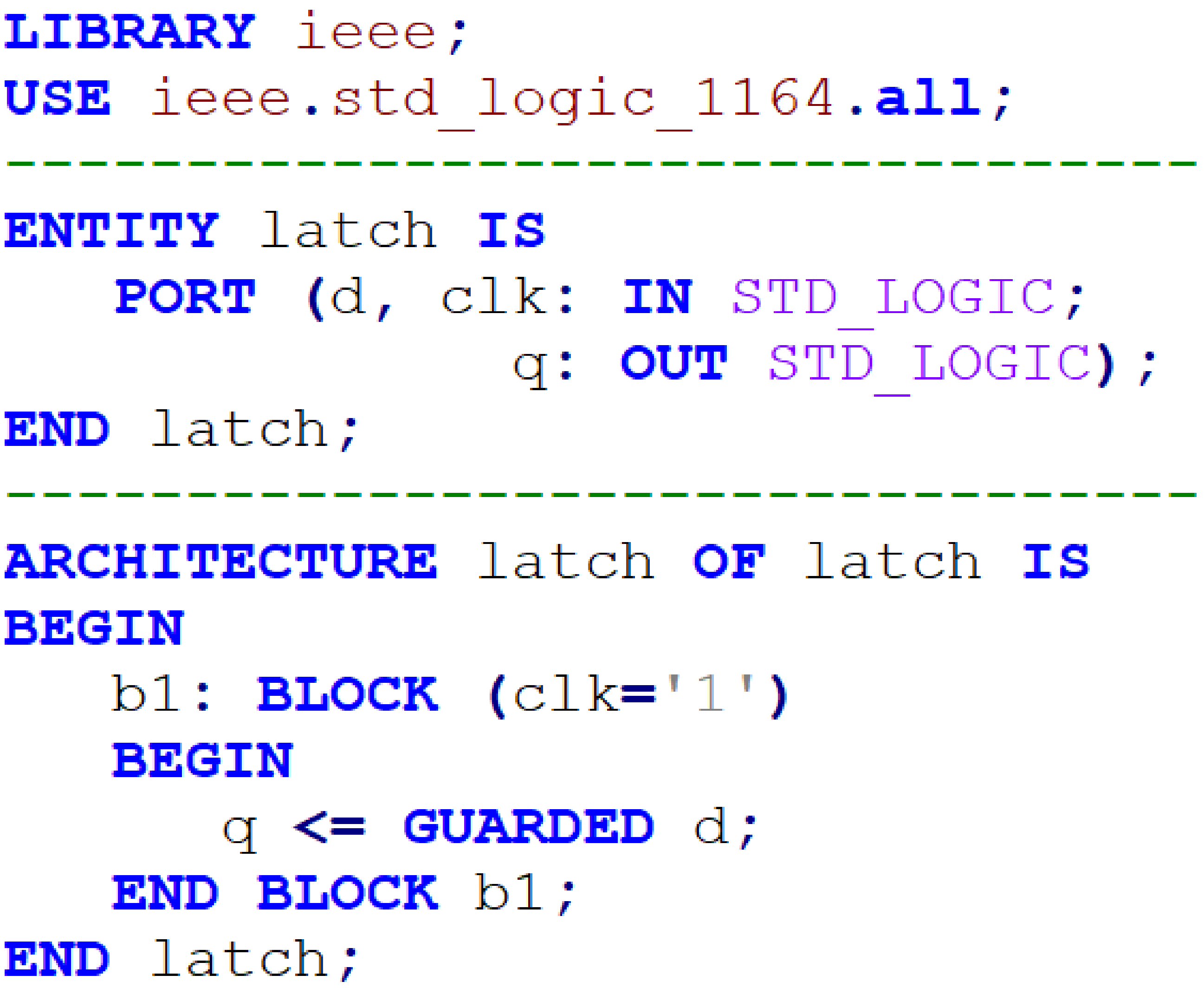
Guarded BLOCK

- A guarded BLOCK is a special kind of BLOCK, which includes an additional expression, called guard expression. A guarded statements in a guarded BLOCK is executed only when the guard expression is TRUE (unguarded statements will be executed anyway).
- **Syntax**:



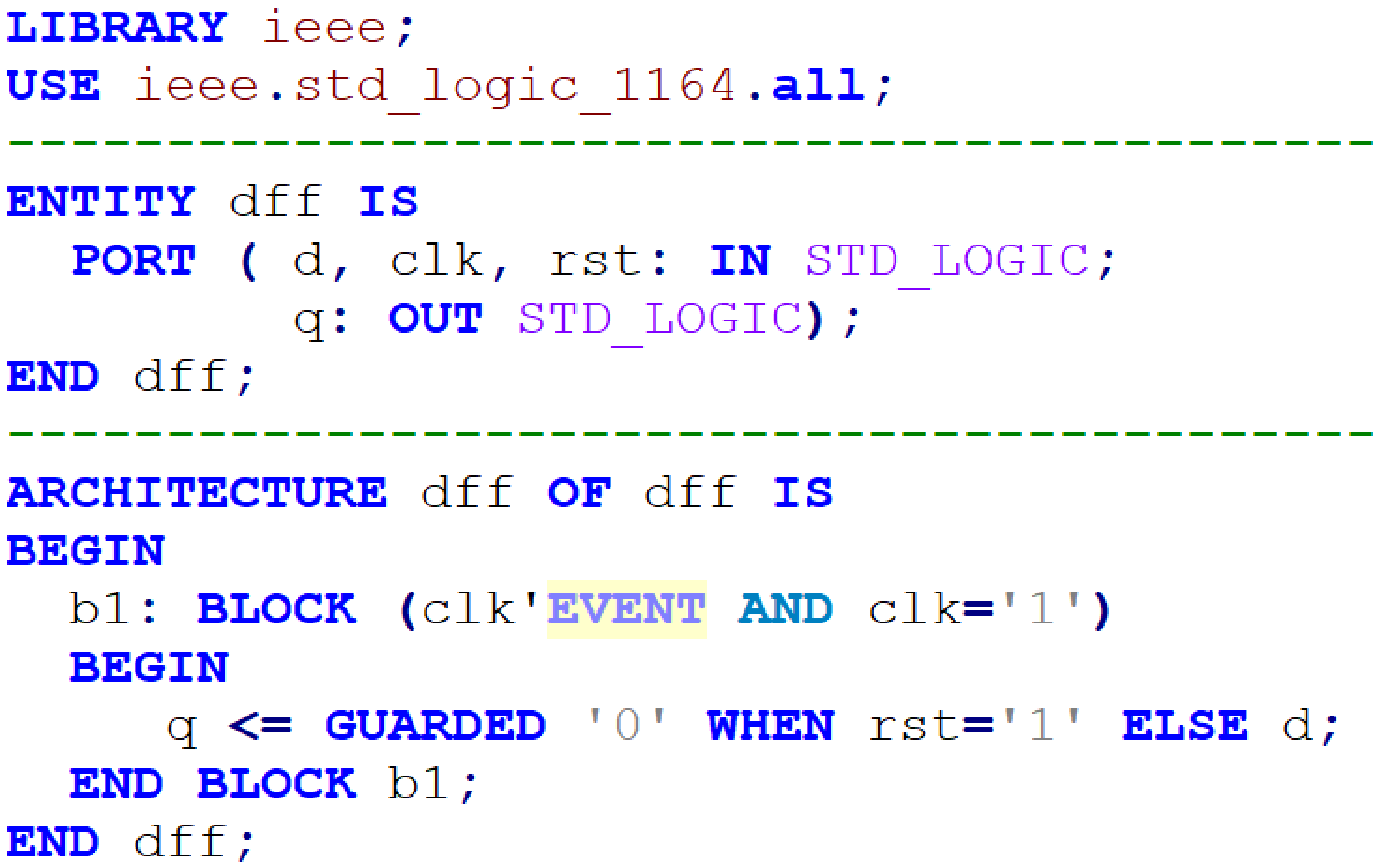
```
label1: BLOCK (guard expression)
        [ generic; [ generic_map; ] ]
        [ port; [ port_map; ] ]
        [ block_declarations ]
BEGIN
        concurrent GUARDED statements;
END BLOCK label;
```

Guarded BLOCK – **D Latch** example



```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------
ENTITY latch IS
   PORT (d, clk: IN STD_LOGIC;
              q: OUT STD_LOGIC);
END latch;
---------------------------------
ARCHITECTURE latch OF latch IS
BEGIN
   b1: BLOCK (clk='1')
   BEGIN
      q <= GUARDED d;
   END BLOCK b1;
END latch;
```

Guarded BLOCK – **DFF** example



```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------
ENTITY dff IS
  PORT ( d, clk, rst: IN STD_LOGIC;
         q: OUT STD_LOGIC);
END dff;
-------------------------------------------
ARCHITECTURE dff OF dff IS
BEGIN
  b1: BLOCK (clk'EVENT AND clk='1')
  BEGIN
     q <= GUARDED '0' WHEN rst='1' ELSE d;
  END BLOCK b1;
END dff;
```