

Design with Hardware Description Languages (HDL)

(DHDL-94952)

Amin Mehranzadeh, Ph.D.

CE Department of
Islamic Azad University of Dezful

mehranzadeh@iaud.ac.ir
Mehran.students@gmail.com

Concurrent Code

Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

Concurrent Code:

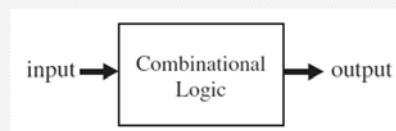
- VHDL code can be concurrent (parallel) or sequential. The concurrent statements in VHDL are WHEN and GENERATE. Besides them, assignments using only operators (AND, NOT, +, *, sll, etc.) can also be used to construct concurrent code. Finally, a special kind of assignment, called BLOCK, can also be employed in this kind of code.

Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

Combinational versus Sequential Logic:

- Combinational logic is that in which the output of the circuit depends solely on the current inputs and the system requires no memory and can be implemented using conventional logic gates.

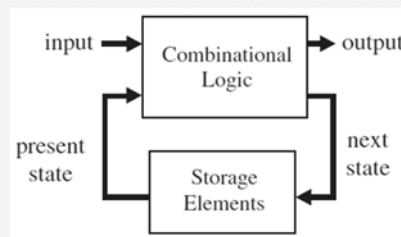


Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

Combinational versus Sequential Logic:

- In Sequential logic, the output is depended on previous inputs. Therefore, storage elements are required, which are connected to the combinational logic block through a feedback loop.

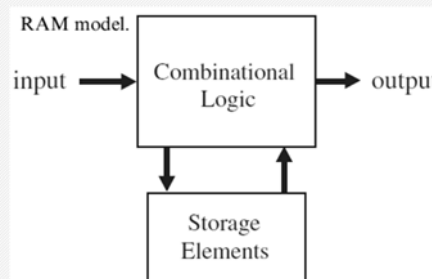


Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

Combinational versus Sequential Logic:

- A common mistake is to think that any circuit that possesses storage elements (flip-flops) is sequential. A RAM (Random Access Memory) is an example (the storage elements appear in a forward path rather than in a feedback loop).



Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

Concurrent versus Sequential Code:

- VHDL code is inherently concurrent (parallel). Only statements placed inside a PROCESS, FUNCTION, or PROCEDURE are sequential. Concurrent code is also called dataflow code.

Example: let us consider a code with three concurrent statements (stat1, stat2, stat3).

stat1		stat3		stat1	
stat2	≡	stat2	≡	stat3	≡ etc.
stat3		stat1		stat2	

the order does not matter.

Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

Concurrent versus Sequential Code:

- Concurrent code can not be used to implement synchronous circuits (the only exception is when a GUARDED BLOCK is used). In other words, in general we can only build combinational logic circuits with concurrent code.
- To obtain sequential logic circuits, sequential code must be employed.

Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

Concurrent versus Sequential Code:

- in concurrent code the following can be used:
- Operators;
- The WHEN statement:
(WHEN/ELSE or WITH/SELECT/WHEN);
- The GENERATE statement;
- The BLOCK statement.

Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

Operators:

- This is the most basic way of creating concurrent code. Operators (AND, OR, +, -, *, sll, sra, etc.):

Operators.

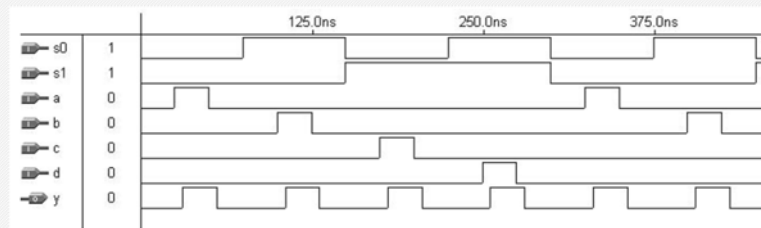
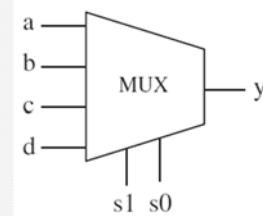
Operator type	Operators	Data types
Logical	NOT, AND, NAND, OR, NOR, XOR, XNOR	BIT, BIT_VECTOR, STD_LOGIC, STD_LOGIC_VECTOR, STD_ULOGIC, STD_ULOGIC_VECTOR
Arithmetic	+, -, *, /, ** (mod, rem, abs)	INTEGER, SIGNED, UNSIGNED
Comparison	=, /=, <, >, <=, >=	All above
Shift	sll, srl, sla, sra, rol, ror	BIT_VECTOR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

Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

Example1: Multiplexer:

- 4-input, one bit per input multiplexer:
The output must be equal to the input selected by the selection bits, s1-s0.



Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezfoul

Example1: Multiplexer:

Its implementation, using only logical operators, can be done as follows:

```

1  -----
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -----
5  ENTITY mux IS
6      PORT ( a, b, c, d, s0, s1: IN STD_LOGIC;
7              y: OUT STD_LOGIC);
8  END mux;
9  -----
10 ARCHITECTURE pure_logic OF mux IS
11 BEGIN
12     y <= (a AND NOT s1 AND NOT s0) OR
13          (b AND NOT s1 AND s0) OR
14          (c AND s1 AND NOT s0) OR
15          (d AND s1 AND s0);
16 END pure_logic;
17 -----

```

Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezfoul

WHEN (Simple and Selected):

- WHEN is one of the fundamental concurrent statements (along with operators and GENERATE). It appears in two forms: WHEN / ELSE (simple WHEN) and WITH / SELECT / WHEN (selected WHEN). Its syntax is shown below.

```
assignment WHEN condition ELSE
assignment WHEN condition ELSE
...;
```

WHEN / ELSE:

```
WITH identifier SELECT
assignment WHEN value,
assignment WHEN value,
...;
```

WITH / SELECT / WHEN

Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

WHEN (Simple and Selected):

- Whenever WITH / SELECT / WHEN is used, all permutations must be tested, so the keyword OTHERS is often useful. Another important keyword is UNAFFECTED, which should be used when no action is to take place.

Example:

```
----- With WHEN/ELSE -----
outp <= "000" WHEN (inp='0' OR reset='1') ELSE
      "001" WHEN ctl='1' ELSE
      "010";
```

```
---- With WITH/SELECT/WHEN -----
WITH control SELECT
  output <= "000" WHEN reset,
            "111" WHEN set,
            UNAFFECTED WHEN OTHERS;
```

Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

WHEN (Simple and Selected):

- Another important aspect related to the WHEN statement is that the “WHEN value” shown in the syntax above can indeed take up three forms:

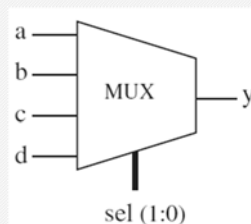
```
WHEN value           -- single value
WHEN value1 to value2 -- range, for enumerated data types
                    -- only
WHEN value1 | value2 | ... -- value1 or value2 or ...
```

Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

Example 2: Multiplexer #2

- This example shows the implementation of the same multiplexer of example 1. However, in it WHEN was employed instead of logical operators. Two solutions are presented:
 - using WHEN/ELSE (simple WHEN)
 - using WITH/SELECT/WHEN (selected WHEN).



Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

Example 2: Multiplexer #2

```

1  ----- Solution 1: with WHEN/ELSE -----
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -----
5  ENTITY mux IS
6      PORT ( a, b, c, d: IN STD_LOGIC;
7              sel: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
8              y: OUT STD_LOGIC);
9  END mux;
10 -----
11 ARCHITECTURE mux1 OF mux IS
12 BEGIN
13     y <=  a WHEN sel="00" ELSE
14           b WHEN sel="01" ELSE
15           c WHEN sel="10" ELSE
16           d;
17 END mux1;
18 -----

```

Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

Example 2: Multiplexer #2

```

1  --- Solution 2: with WITH/SELECT/WHEN ---
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -----
5  ENTITY mux IS
6      PORT ( a, b, c, d: IN STD_LOGIC;
7              sel: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
8              y: OUT STD_LOGIC);
9  END mux;
10 -----
11 ARCHITECTURE mux2 OF mux IS
12 BEGIN
13     WITH sel SELECT
14         y <=  a WHEN "00",      -- notice "," instead of ";"
15               b WHEN "01",
16               c WHEN "10",
17               d WHEN OTHERS;    -- cannot be "d WHEN "11" "
18 END mux2;
19 -----

```

Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

Example 2: Multiplexer #2

In the solutions above, sel could have been declared as an INTEGER, in which case the code would be the following:

```

1 -----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -----
5 ENTITY mux IS
6     PORT ( a, b, c, d: IN STD_LOGIC;
7           sel: IN INTEGER RANGE 0 TO 3;
8           y: OUT STD_LOGIC);
9 END mux;
10 ---- Solution 1: with WHEN/ELSE -----
11 ARCHITECTURE mux1 OF mux IS
12 BEGIN
13     y <= a WHEN sel=0 ELSE
14         b WHEN sel=1 ELSE
15         c WHEN sel=2 ELSE
16         d;
17 END mux1;
18 -- Solution 2: with WITH/SELECT/WHEN -----
19 ARCHITECTURE mux2 OF mux IS
20 BEGIN
21     WITH sel SELECT
22         y <= a WHEN 0,
23             b WHEN 1,
24             c WHEN 2,
25             d WHEN 3;    -- here, 3 or OTHERS are equivalent,
26 END mux2;              -- for all options are tested anyway
27 -----

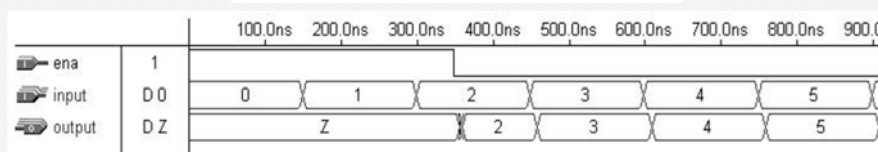
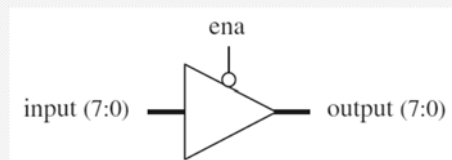
```

Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

Example 3: Tri-state Buffer

- This is another example that illustrates the use of WHEN. The 3-state buffer must provide output = input when ena (enable) is low, or output = "ZZZZZZZZ" (high impedance) otherwise.



Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

Example 3: Tri-state Buffer

```

1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  -----
4  ENTITY tri_state IS
5      PORT ( ena: IN STD_LOGIC;
6             input: IN STD_LOGIC_VECTOR (7 DOWNT0 0);
7             output: OUT STD_LOGIC_VECTOR (7 DOWNT0 0));
8  END tri_state;
9  -----
10 ARCHITECTURE tri_state OF tri_state IS
11 BEGIN
12     output <= input WHEN (ena='0') ELSE
13         (OTHERS => 'Z');
14 END tri_state;
15 -----

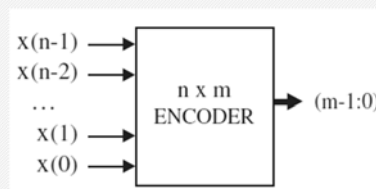
```

Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

Example 4: Encoder (n-by-m encoder)

- We assume that n is a power of two, so $m = \log_2 n$. One and only one input bit is expected to be high at a time, whose address must be encoded at the output. Two solutions are presented, one using WHEN / ELSE, and the other with WITH / SELECT / WHEN.



Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

Example 4: Encoder
(n-by-m encoder)

```

1  ---- Solution 1: with WHEN/ELSE ----
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -----
5  ENTITY encoder IS
6      PORT ( x: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
7            y: OUT STD_LOGIC_VECTOR (2 DOWNTO 0));
8  END encoder;
9  -----
10 ARCHITECTURE encoder1 OF encoder IS
11 BEGIN
12     y <=  "000" WHEN x="00000001" ELSE
13           "001" WHEN x="00000010" ELSE
14           "010" WHEN x="00000100" ELSE
15           "011" WHEN x="00001000" ELSE
16           "100" WHEN x="00010000" ELSE
17           "101" WHEN x="00100000" ELSE
18           "110" WHEN x="01000000" ELSE
19           "111" WHEN x="10000000" ELSE
20           "ZZZ";
21 END encoder1;
22 -----

```

Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

Example 4: Encoder
(n-by-m encoder)

```

1  ---- Solution 2: with WITH/SELECT/WHEN ----
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -----
5  ENTITY encoder IS
6      PORT ( x: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
7            y: OUT STD_LOGIC_VECTOR (2 DOWNTO 0));
8  END encoder;
9  -----
10 ARCHITECTURE encoder2 OF encoder IS
11 BEGIN
12     WITH x SELECT
13         y <=  "000" WHEN "00000001",
14               "001" WHEN "00000010",
15               "010" WHEN "00000100",
16               "011" WHEN "00001000",
17               "100" WHEN "00010000",
18               "101" WHEN "00100000",
19               "110" WHEN "01000000",
20               "111" WHEN "10000000",
21               "ZZZ" WHEN OTHERS;
22 END encoder2;
23 -----

```

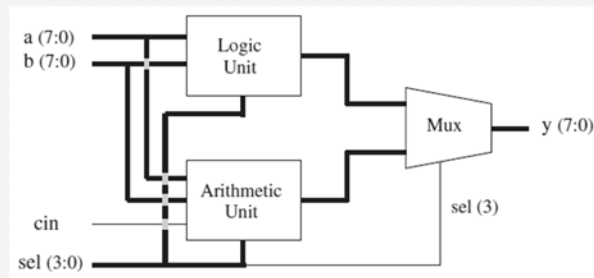
Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

Example 5: ALU

(Arithmetic Logic Unit)

- ALU is a circuit capable of executing both kinds of operations, arithmetic as well as logical.



Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

Example 5: ALU

(Arithmetic Logic Unit)

- The output (arithmetic or logical) is selected by the MSB of sel, while the specific operation is selected by sel's other three bits.

sel	Operation	Function	Unit
0000	$y \leftarrow a$	Transfer a	Arithmetic
0001	$y \leftarrow a+1$	Increment a	
0010	$y \leftarrow a-1$	Decrement a	
0011	$y \leftarrow b$	Transfer b	
0100	$y \leftarrow b+1$	Increment b	
0101	$y \leftarrow b-1$	Decrement b	
0110	$y \leftarrow a+b$	Add a and b	
0111	$y \leftarrow a+b+cin$	Add a and b with carry	
1000	$y \leftarrow \text{NOT } a$	Complement a	Logic
1001	$y \leftarrow \text{NOT } b$	Complement b	
1010	$y \leftarrow a \text{ AND } b$	AND	
1011	$y \leftarrow a \text{ OR } b$	OR	
1100	$y \leftarrow a \text{ NAND } b$	NAND	
1101	$y \leftarrow a \text{ NOR } b$	NOR	
1110	$y \leftarrow a \text{ XOR } b$	XOR	
1111	$y \leftarrow a \text{ XNOR } b$	XNOR	

Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

Example 5: ALU
(Arithmetic Logic Unit)

```

1  -----
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE ieee.std_logic_unsigned.all;
5  -----
6  ENTITY ALU IS
7      PORT (a, b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
8            sel: IN STD_LOGIC_VECTOR (3 DOWNTO 0);
9            cin: IN STD_LOGIC;
10           y: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
11 END ALU;
12 -----
13 ARCHITECTURE dataflow OF ALU IS
14     SIGNAL arith, logic: STD_LOGIC_VECTOR (7 DOWNTO 0);
15 BEGIN
16     ----- Arithmetic unit: -----
17     WITH sel(2 DOWNTO 0) SELECT
18         arith <=  a WHEN "000",
19                   a+1 WHEN "001",
20                   a-1 WHEN "010",
21                   b WHEN "011",
22                   b+1 WHEN "100",

```

Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

Example 5: ALU
(Arithmetic Logic Unit)

```

23         b-1 WHEN "101",
24         a+b WHEN "110",
25         a+b+cin WHEN OTHERS;
26     ----- Logic unit: -----
27     WITH sel(2 DOWNTO 0) SELECT
28         logic <=  NOT a WHEN "000",
29                   NOT b WHEN "001",
30                   a AND b WHEN "010",
31                   a OR b WHEN "011",
32                   a NAND b WHEN "100",
33                   a NOR b WHEN "101",
34                   a XOR b WHEN "110",
35                   NOT (a XOR b) WHEN OTHERS;
36     ----- Mux: -----
37     WITH sel(3) SELECT
38         y <=  arith WHEN '0',
39              logic WHEN OTHERS;
40 END dataflow;
41 -----

```

Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

GENERATE:

- GENERATE is another concurrent statement (along with operators and WHEN). It is equivalent to the sequential statement LOOP in the sense that it allows a section of code to be repeated a number of times, thus creating several instances of the same assignments. Its regular form is the FOR / GENERATE construct. Notice that GENERATE must be labeled.

FOR / GENERATE:

```
label: FOR identifier IN range GENERATE
      (concurrent assignments)
END GENERATE;
```

Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

GENERATE:

An irregular form is also available, which uses IF/GENERATE (with an IF equivalent; recall that originally IF is a sequential statement). Here ELSE is not allowed. In the same way that IF/GENERATE can be nested inside FOR/GENERATE (syntax below), the opposite can also be done.

IF / GENERATE nested inside FOR / GENERATE:

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

Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

GENERATE:

Example:

```
SIGNAL x: BIT_VECTOR (7 DOWNT0 0);
SIGNAL y: BIT_VECTOR (15 DOWNT0 0);
SIGNAL z: BIT_VECTOR (7 DOWNT0 0);
...
G1: FOR i IN x'RANGE GENERATE
    z(i) <= x(i) AND y(i+8);
END GENERATE;
```

One important remark about GENERATE (and the same is true for LOOP) is that both limits of the range must be static. As an example, let us consider the code below, where choice is an input (non-static) parameter. This kind of code is generally not synthesizable.

```
NotOK: FOR i IN 0 TO choice GENERATE
    (concurrent statements)
END GENERATE;
```

Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

GENERATE:

We also must to be aware of multiply-driven (unresolved) signals. For example,

```
OK: FOR i IN 0 TO 7 GENERATE
    output(i) <= '1' WHEN (a(i) AND b(i)) = '1' ELSE '0';
END GENERATE;
```

is fine. However, the compiler will complain that “accum” is multiply driven (and stop compilation) in either of the following two cases:

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

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

Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

Example 6: Vector Shifter

- This example illustrates the use of GENERATE. In it, the output vector must be a shifted version of the input vector. For example, if the input bus has width 4, and the present value is "1111", then the output should be one of the lines of the following matrix (the original vector is underscored):

```
row(0): 0 0 0 0 1 1 1 1
row(1): 0 0 0 1 1 1 1 0
row(2): 0 0 1 1 1 1 0 0
row(3): 0 1 1 1 1 0 0 0
row(4): 1 1 1 1 0 0 0 0
```

Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

Example 6: Vector Shifter

```
1  -----
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -----
5  ENTITY shifter IS
6      PORT ( inp: IN STD_LOGIC_VECTOR (3 DOWNTO 0);
7            sel: IN INTEGER RANGE 0 TO 4;
8            outp: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
9  END shifter;
10 -----
11 ARCHITECTURE shifter OF shifter IS
12     SUBTYPE vector IS STD_LOGIC_VECTOR (7 DOWNTO 0);
13     TYPE matrix IS ARRAY (4 DOWNTO 0) OF vector;
14     SIGNAL row: matrix;
15 BEGIN
```

Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

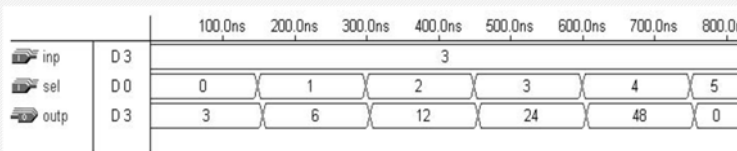
Example 6: Vector Shifter

```

16  row(0) <= "0000" & inp;
17  G1: FOR i IN 1 TO 4 GENERATE
18      row(i) <= row(i-1)(6 DOWNT0 0) & '0';
19  END GENERATE;
20  outp <= row(sel);
21 END shifter;
22 -----

```

As can be seen, inp = "0011" (decimal 3) was applied to the circuit. The result was outp = "00000011" (decimal 3) when sel = 0 (no shift), outp = "00000110" (decimal 6) when sel = 1 (one shift to the left), outp = "00001100" (decimal 12) when sel = 2 (two shifts to the left), and so on.



Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

BLOCK:

(There are two kinds of BLOCK statements: Simple and Guarded)

- Simple BLOCK:

The BLOCK statement, in its simple form, represents only a way of locally partitioning the code. 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 (which might be helpful when dealing with long codes). Its syntax is shown below.

```

label: BLOCK
[declarative part]
BEGIN
(concurrent statements)
END BLOCK label;

```

Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

BLOCK:

(There are two kinds of BLOCK statements: Simple and Guarded)

- Simple BLOCK:

Therefore, the overall aspect of a "blocked" code is the following:

Example:

```
b1: BLOCK
  SIGNAL a: STD_LOGIC;
BEGIN
  a <= input_sig  WHEN ena='1' ELSE 'Z';
END BLOCK b1;
```

```
-----
ARCHITECTURE example ...
BEGIN
  ...
  block1: BLOCK
  BEGIN
    ...
  END BLOCK block1
  ...
  block2: BLOCK
  BEGIN
    ...
  END BLOCK block2;
  ...
END example;
-----
```

Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

BLOCK:

(There are two kinds of BLOCK statements: Simple and Guarded)

- A BLOCK (simple or guarded) can be nested inside another BLOCK. The corresponding syntax is shown below.

```
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;
```

Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

BLOCK:

(There are two kinds of BLOCK statements: Simple and Guarded)

- Guarded BLOCK:

A guarded BLOCK is a special kind of BLOCK, which includes an additional expression, called guard expression. A guarded statement in a guarded BLOCK is executed only when the guard expression is TRUE.

Guarded BLOCK:

```
label: BLOCK (guard expression)
[declarative part]
BEGIN
(concurrent guarded and unguarded statements)
END BLOCK label;
```

Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

Example 7:

Latch Implemented with a Guarded BLOCK

- The example presented below implements a transparent latch. In it, `clk='1'` (line12) is the guard expression, while `q<=GUARDED d` (line 14) is a guarded statement. Therefore, `q<=d` will only occur if `clk='1'`.

```
1  -----
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -----
5  ENTITY latch IS
6      PORT (d, clk: IN STD_LOGIC;
7            q: OUT STD_LOGIC);
8  END latch;
9  -----
10 ARCHITECTURE latch OF latch IS
11 BEGIN
12     b1: BLOCK (clk='1')
13     BEGIN
14         q <= GUARDED d;
15     END BLOCK b1;
16 END latch;
17 -----
```

Amin Mehranzadeh, Ph.D.

CE Department of Islamic Azad University of Dezful

Example 8: DFF Implemented with a Guarded BLOCK

A positive-edge sensitive D-type flip-flop, with synchronous reset. In it, `clk'EVENT AND clk='1'` (line 12) is the guard expression, while `q <= GUARDED '0' WHEN rst='1'` (line 14) is a guarded statement. Therefore, `q<='0'` will occur when the guard expression is true and `rst` is '1'.

```

1  -----
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -----
5  ENTITY dff IS
6      PORT ( d, clk, rst: IN STD_LOGIC;
7            q: OUT STD_LOGIC);
8  END dff;
9  -----
10 ARCHITECTURE dff OF dff IS
11 BEGIN
12     b1: BLOCK (clk'EVENT AND clk='1')
13     BEGIN
14         q <= GUARDED '0' WHEN rst='1' ELSE d;
15     END BLOCK b1;
16 END dff;
17 -----

```