Concurrent Code

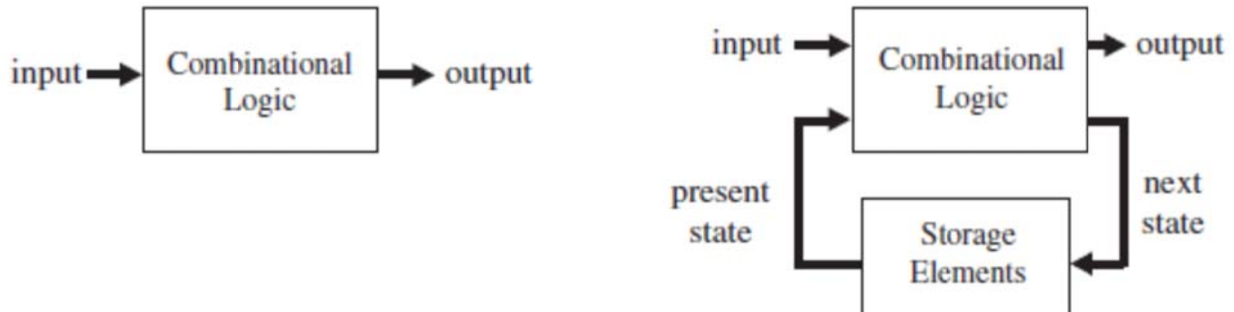
Combinational versus Sequential Logic:

By definition, combinational logic is that in which the output of the circuit depends solely on the current inputs. It is then clear that, in principle, the system requires no memory and can be implemented using conventional logic gates.

In contrast, sequential logic is defined as that in which the output does depend on previous inputs. Therefore, storage elements are required, which are connected to the combinational logic block through a feedback loop, such that now the stored states (created by previous inputs) will also affect the output of the circuit.

Concurrent versus Sequential Code:

VHDL code is inherently concurrent (parallel). Only statements placed inside a PROCESS, FUNCTION, or PROCEDURE are sequential. Still, though within these blocks the execution is sequential, the block, as a whole, is concurrent with any other (external) statements. Concurrent code is also called dataflow code.



In concurrent code the statements outside PROCESSES, FUNCTIONS, or PROCEDURES can be used.

In summary, in concurrent code the following can be used:

Operators;
The WHEN statement (WHEN/ELSE or WITH/SELECT/WHEN);
The GENERATE statement;
The BLOCK statement.

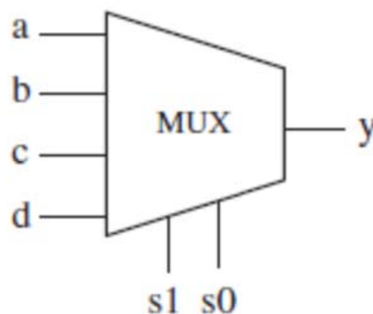
Using Operators

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

Example: Multiplexer #1

Figure below shows a 4-input, one bit per input multiplexer. The output must be equal to the input selected by the selection bits, s1-s0. 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 ---------------------------------------
```

WHEN (Simple and Selected)

As mentioned above, 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.

WHEN / ELSE:

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

WITH / SELECT / WHEN:

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

Example:

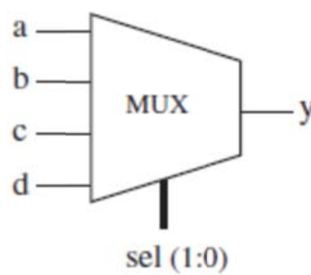
```
------ With WHEN/ELSE -------------------------
        outp <= "000" WHEN (inp='0' OR reset='1') ELSE
                    "001" WHEN ctl='1' ELSE
                    "010";
```

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

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 ...
```

Example: Multiplexer #2



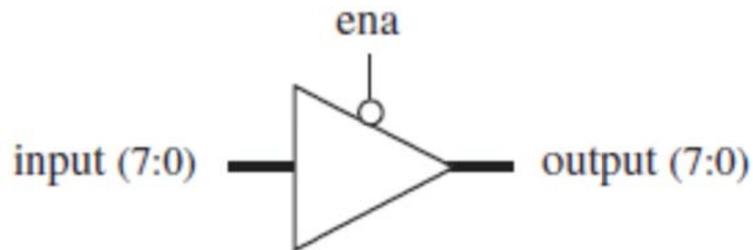
```
1 ------- Solution 1: with WHEN/ELSE --------

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 -------------------------------------------

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 -------------------------------------------
```

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 ---------------------------------------------
```

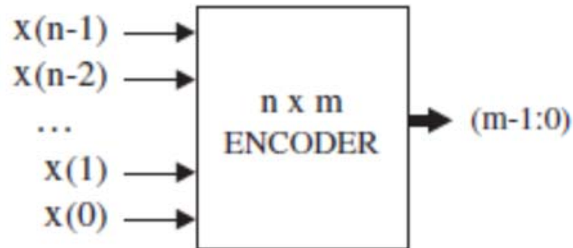
Example: Tri-state Buffer

The 3-state buffer of must provide output =input when ena (enable) is low, or output ="ZZZZZZZZ" (high impedance) otherwise.




```
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 DOWNTO 0);
7                 output: OUT STD_LOGIC_VECTOR (7 DOWNTO 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 ---------------------------------------------
```

Example: 8x3 Encoder

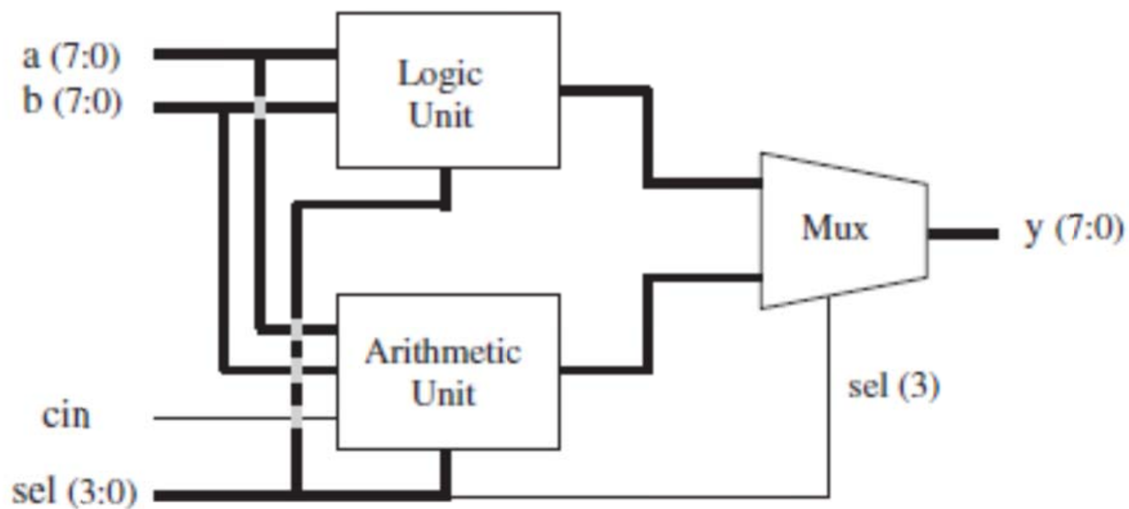



```
1 ---- Solution 1: with WHEN/ELSE -------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;


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 ---------------------------------------------
```

```
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;
```

Example: ALU

As the name says, it is a circuit capable of executing both kinds of operations, arithmetic as well as logical.



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

```
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",
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 ---------------------------------------------
```

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.

FOR / GENERATE:

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

IF / GENERATE nested inside FOR / GENERATE:

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

Example:

```
SIGNAL x: BIT_VECTOR (7 DOWNTO 0);
SIGNAL y: BIT_VECTOR (15 DOWNTO 0);
SIGNAL z: BIT_VECTOR (7 DOWNTO 0);
...
G1: FOR i IN x'RANGE GENERATE

        z(i) <= x(i) AND y(i+8);

END GENERATE;
```

One important remark about GENERATE 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;
```

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;
```

Example: Vector Shifter

This example illustrates the use of GENERATE. In it, the output vector must be a shifted version of the input vector, with twice its width and an amount of shift specified by another input. 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

The first row corresponds to the input itself, with no shift and the most significant bits filled with '0's. Each successive row is equal to the previous row shifted one position to the left. The solution below has input inp, output outp, and shift selection sel. Each row of the array above is defined as subtype vector in VHDL program.

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