Concurrent Code
WHEN, GENERATE, BLOCK

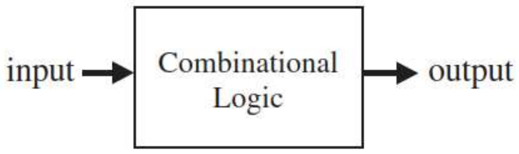
We study the staments required to create a concurrent code.

VHDL

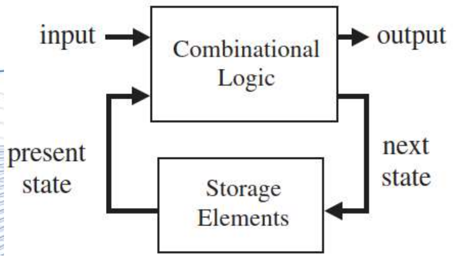
VHSIC HARDWARE DESCRIPTION LANGUAGE

Concurrent Code
Combinational vs Sequential Logic

- We have covered the basics of VHDL. Now we start describing how to create a code properly.
- A VHDL code can be concurrent or sequential
- For concurrent CODE we use the following statements in VHDL: **WHEN** and **GENERATE**
- Also, operators can be used to create concurrent code.
- Also, a special kind of assignment, namely BLOCK can be used.
- In a **Combinational logic** circuit, the output depends only on current inputs. (System does not require stored data)



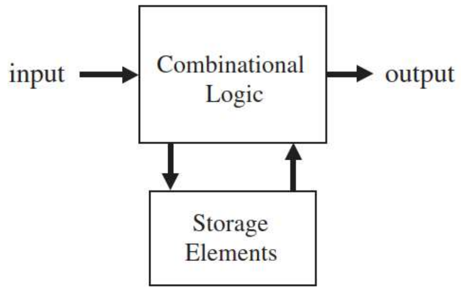
- In a **Sequential logic**, the output depends on previous and/or current inputs. The system require stored data which is feeded-back to the system.



Concurrent Code

Combinational vs Sequential Logic

- Not any circuit that contain storage elements (flip-flops) is a sequential logic!. DO not get confused! See this example:

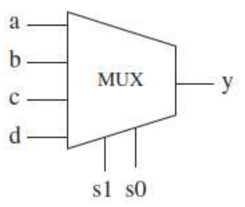


In a RAM, like this figure, the storge elements are in forward path instead of feedback. The memory-read operation depends only on the current address vector applied to the RAM input. No access to previous data is needed!

Concurrent Code

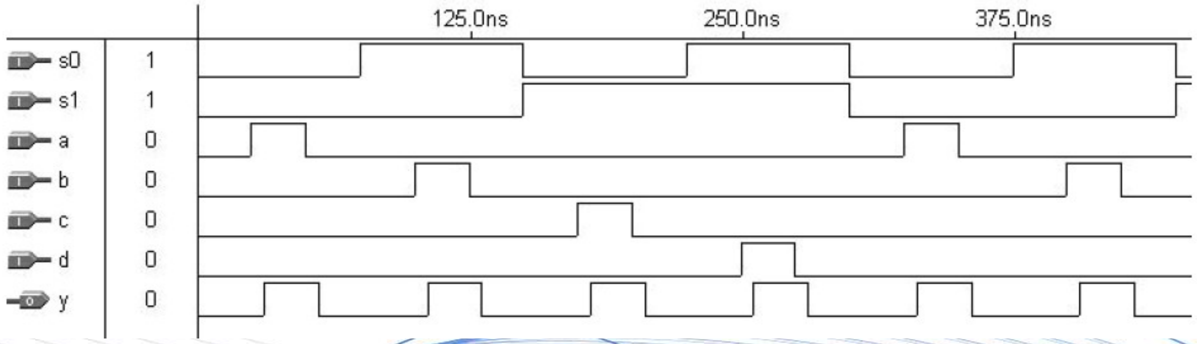
Concurrent vs Sequential Code

- Only statements placed inside PROCESS, FUNCTION or PROCEDURE are sequential. (we see this in next section).
- Concurrent code is called dataflow.
- To build combinational logic circuits, we need concurrent code:
  - Operators;
  - The WHEN statement (WHEN/ELSE or WITH/SELECT/WHEN);
  - The GENERATE statement;
  - The BLOCK statement



4 inputs
1 bit per input
Output is selected by s1, s0

- We have reviewed concurrent code using operators, for instance:

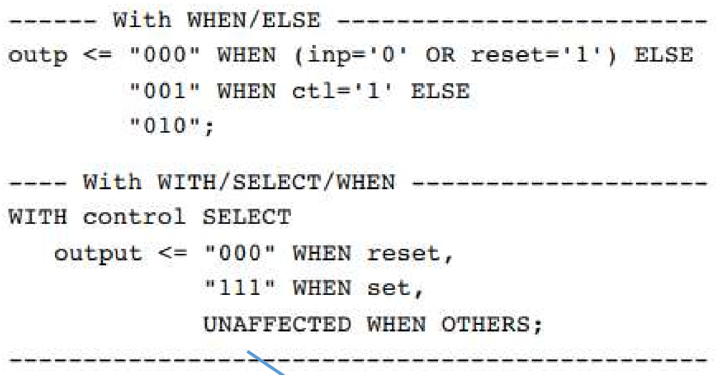


```
---------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------
ENTITY mux IS
   PORT ( a, b, c, d, s0, s1: IN STD_LOGIC;
          y: OUT STD_LOGIC);
END mux;
---------------------------------------
ARCHITECTURE pure_logic OF mux IS
BEGIN
   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 pure_logic;
---------------------------------------
```

Concurrent Code
WHEN (WHEN/ELSE and WITH/SELECT/WHEN)

- **WHEN/ELSE** syntax:
  - assigment WHEN condition ELSE
  - assigment WHEN condition ELSE
  - …;

- **WITH/SELECT/WHEN**
  - WITH identifier SELECT
    assigment WHEN value,
    assigment WHEN value,
    …;



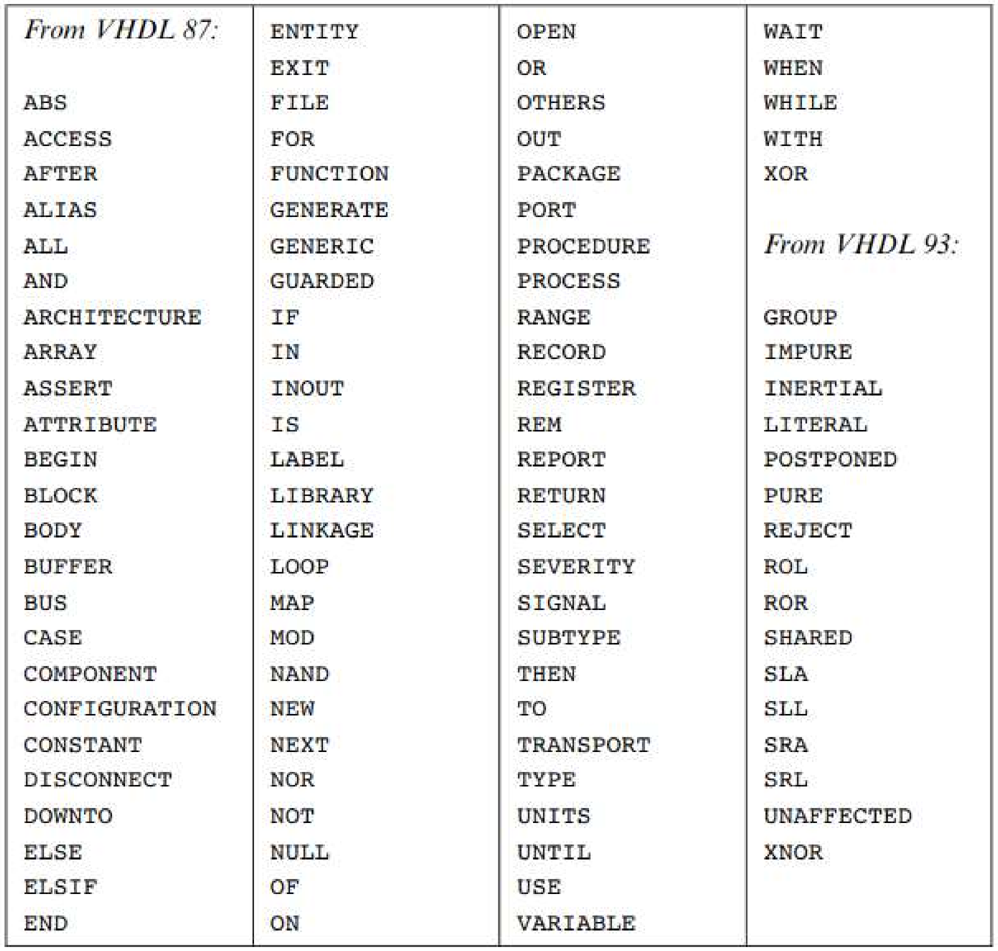
```
------ With WHEN/ELSE -------------------------
outp <= "000" WHEN (inp='0' OR reset='1') ELSE
        "001" WHEN ctl='1' ELSE
        "010";

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

Note that using WITH/SELECT/WHEN all combinations must be listed/tested. A nice way to avoid a long list is using keywords like UNAFFECTED and OTHERS. (full list of keywords is in next slide)

<u>Note that WHEN can also assign:</u>

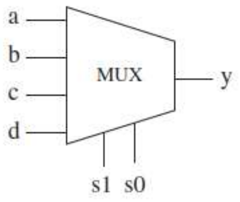
WHEN value	-- single value
WHEN value1 to value2	-- range, for enumerated data types only
WHEN value1 | value2 |...	-- value1 or value2 or ...



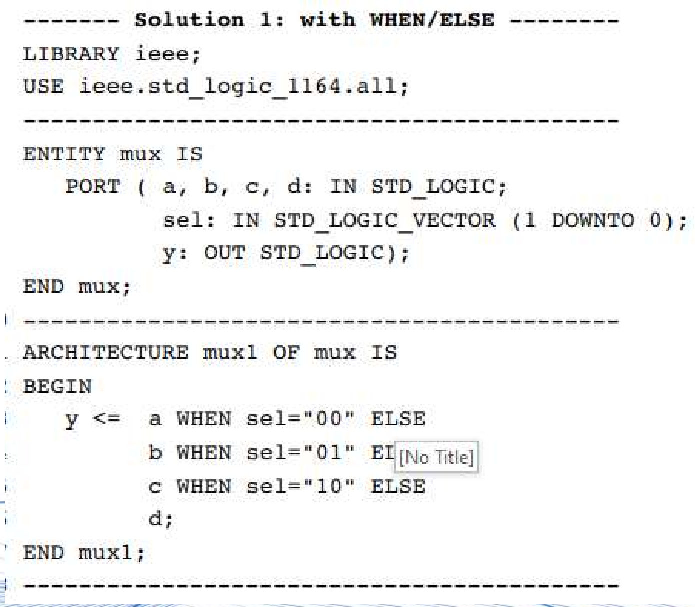
From VHDL 87:	ENTITY	OPEN	WAIT
	EXIT	OR	WHEN
ABS	FILE	OTHERS	WHILE
ACCESS	FOR	OUT	WITH
AFTER	FUNCTION	PACKAGE	XOR
ALIAS	GENERATE	PORT	
ALL	GENERIC	PROCEDURE	*From VHDL 93:*
AND	GUARDED	PROCESS	
ARCHITECTURE	IF	RANGE	GROUP
ARRAY	IN	RECORD	IMPURE
ASSERT	INOUT	REGISTER	INERTIAL
ATTRIBUTE	IS	REM	LITERAL
BEGIN	LABEL	REPORT	POSTPONED
BLOCK	LIBRARY	RETURN	PURE
BODY	LINKAGE	SELECT	REJECT
BUFFER	LOOP	SEVERITY	ROL
BUS	MAP	SIGNAL	ROR
CASE	MOD	SUBTYPE	SHARED
COMPONENT	NAND	THEN	SLA
CONFIGURATION	NEW	TO	SLL
CONSTANT	NEXT	TRANSPORT	SRA
DISCONNECT	NOR	TYPE	SRL
DOWNTO	NOT	UNITS	UNAFFECTED
ELSE	NULL	UNTIL	XNOR
ELSIF	OF	USE	
END	ON	VARIABLE	

Concurrent Code
WHEN (WHEN/ELSE and WITH/SELECT/WHEN)

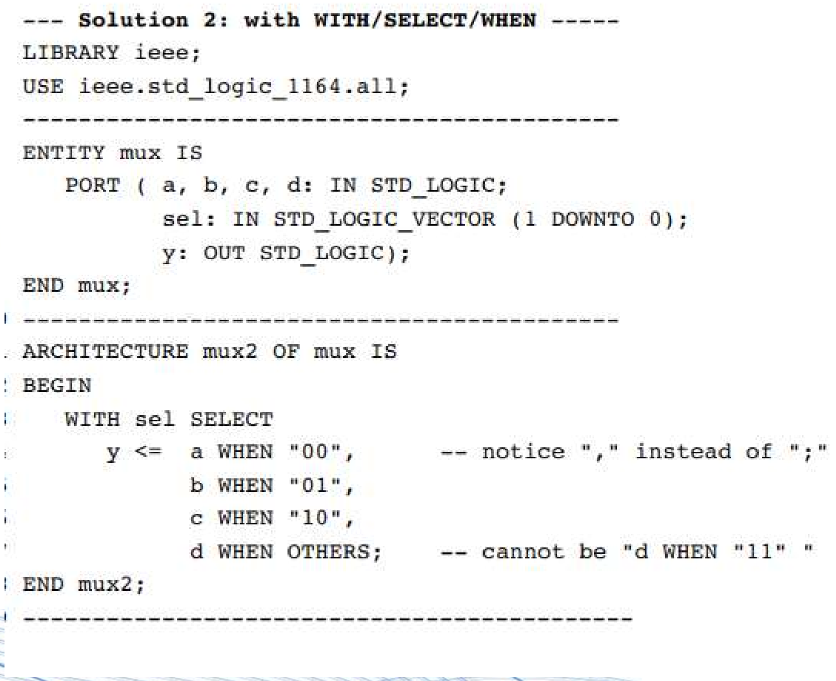
Implement a multiplexer



4 inputs
1 bit per input
Output is selected by s1, s0



```
------- Solution 1: with WHEN/ELSE --------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------
ENTITY mux IS
   PORT ( a, b, c, d: IN STD_LOGIC;
          sel: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
          y: OUT STD_LOGIC);
END mux;
-------------------------------------------
ARCHITECTURE mux1 OF mux IS
BEGIN
   y <=  a WHEN sel="00" ELSE
         b WHEN sel="01" EL[No Title]
         c WHEN sel="10" ELSE
         d;
END mux1;
-------------------------------------------
```



```
--- Solution 2: with WITH/SELECT/WHEN -----
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------
ENTITY mux IS
   PORT ( a, b, c, d: IN STD_LOGIC;
          sel: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
          y: OUT STD_LOGIC);
END mux;
-------------------------------------------
ARCHITECTURE mux2 OF mux IS
BEGIN
   WITH sel SELECT
      y <=  a WHEN "00",      -- notice "," instead of ";"
            b WHEN "01",
            c WHEN "10",
            d WHEN OTHERS;    -- cannot be "d WHEN "11" "
END mux2;
--------------------------------------------
```

Concurrent Code
GENERATE

- It is another concurrent statement. It allow a section of a code to be repeated a number of times. It create several instances (examples or cases of something) of the same assignment.
- **GENERATE** is used together with **FOR** (regular form) or **IF** (irregular form).
- Syntax **FOR/GENERATE**:

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

- Syntax **IF**/**GENERATE** (ELSE is not allowed)**:**

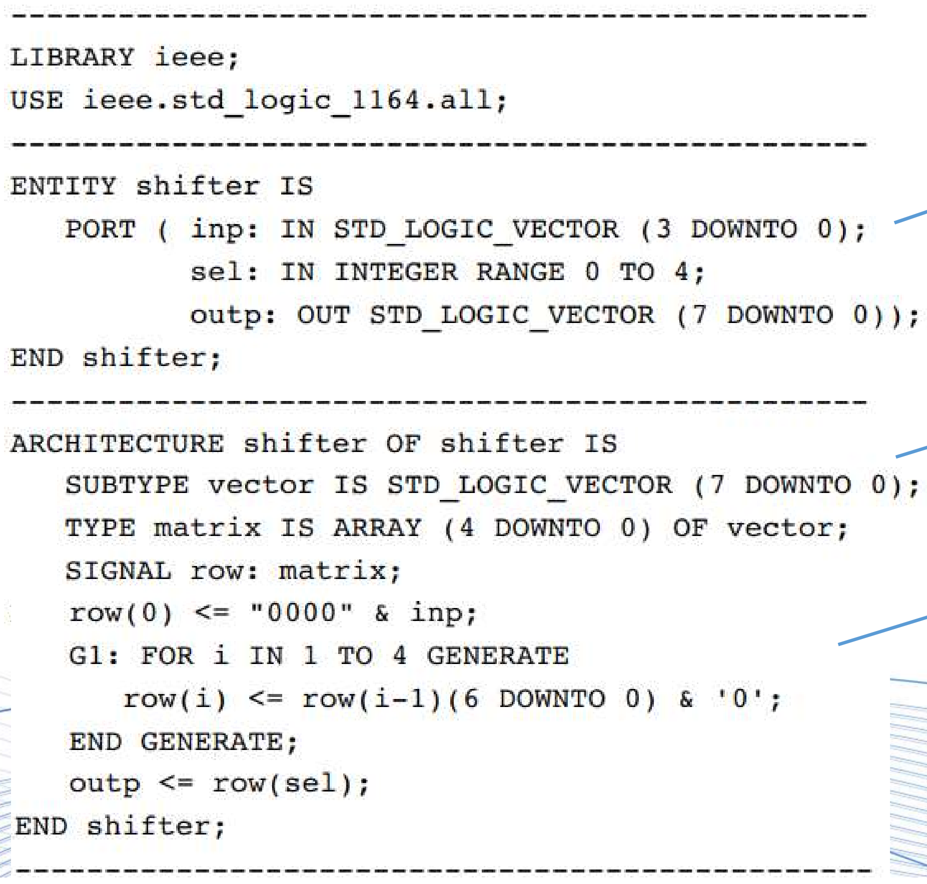
```
label1: IF condition GENERATE
        (concurrent assignments)
END GENERATE;
```

Nested structures are allowed (i.e., a FOR/GENERATE within IF/GENERATE or other way around).

Concurrent Code

GENERATE-Example

Generate a circuit which receives an input vector of 4 bits and generate an output vector of 8bits. The 4-bits input vector is contained in the last 4-bits of the output vector (other bits are 0). An additional input can shift the position of the vector from 0 to 4 positions to the left. For instance, if input is **1010** and shift is 2, output is 00**1010**00



```
---------------------------------------------------
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;
   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);
END shifter;
---------------------------------------------------
```

Inputs and outpus are defined. Shifting is selected as integer, input is 4-bit vector ad output is 8-bit vector-

Within our architecture we define a signal row, which is of the matrix. Defined as an array of 5 rows of 8-bit vectors.
Each vector is defined as a subtype std_logic_vector.
This allow further assignments.

We **GENERATE** the concurrent code
(filling the matrix, each row shift the vector one to the left):

Ejemplo: input 1010.
row(0) <= “0000” & inp; (00001010)
row(1) <= row(0)(6 down to 0) & ‘0’; (00010100)
row(2) <= row(1)(6 down to 0) & ‘0’; (00101000)
row(3) <= row(2)(6 down to 0) & ‘0’; (01010000)
row(4) <= row(3)(6 down to 0) & ‘0’; (10100000)

Concurrent Code
Simple BLOCKS

- In its **simple form a BLOCK** statement allows to ***cluster*** a ***set of concurrent statements***. This makes the overall code more readable and manageable. Its syntax is:

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

- Also, a BLOCK can be nested into another BLOCK:

```
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;
```

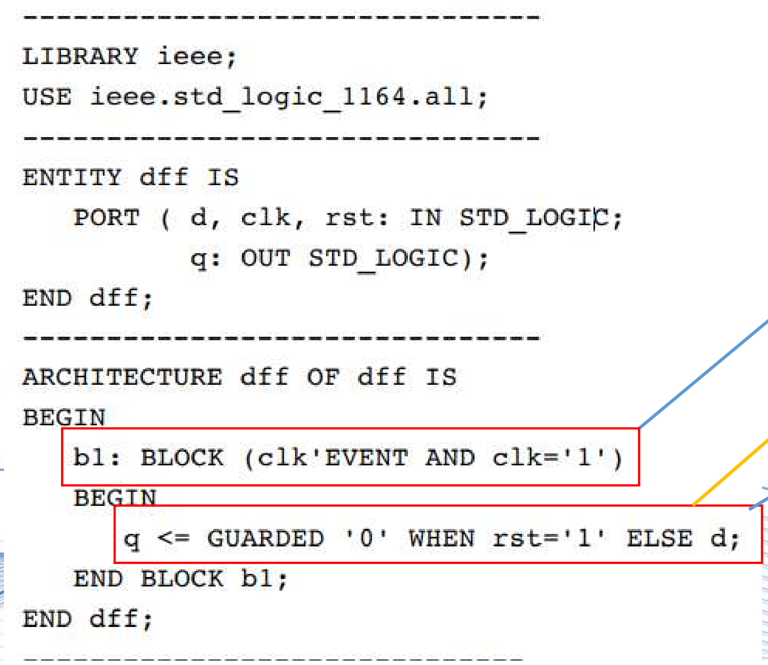
```
b1: BLOCK
   SIGNAL a: STD_LOGIC;
BEGIN
   a <= input_sig   WHEN ena='1' ELSE 'Z';
END BLOCK b1;
```

Concurrent Code

Guarded BLOCKS

- A **guarded BLOCK** is a special form of BLOCK which includes a *guard* expression. The statements within the guarded BLOCK are only executed when the *guard* expression is true.

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



```
--------------------------------
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;
--------------------------------
```

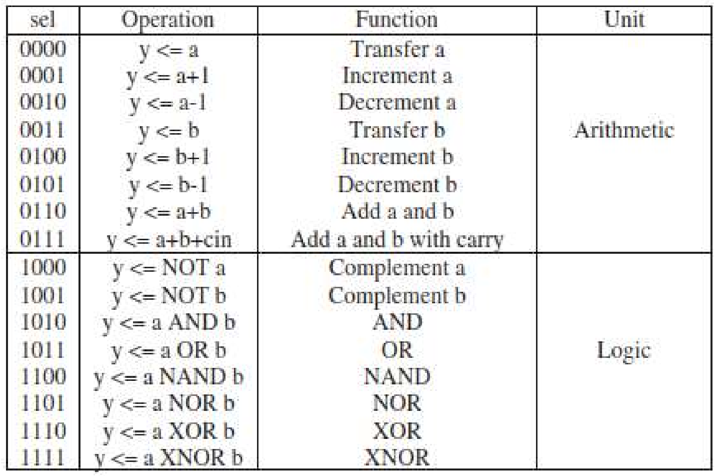
Guard expression.
If this **expression** is true, then the **GUARDED statement** is executed.

Guarded statment assign 0 to q only if reset is 1. Otherwise, assigns the input d.

Concurrent Code

Exercise: WHEN (WHEN/ELSE and WITH/SELECT/WHEN)

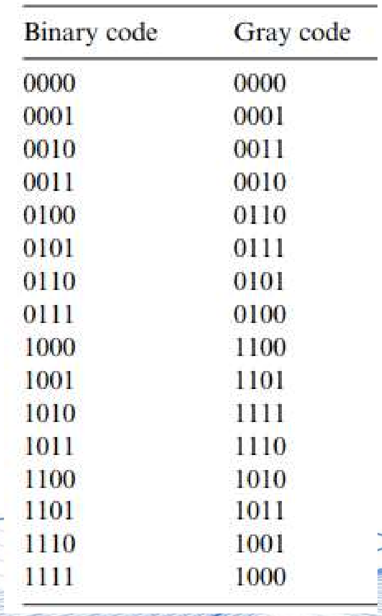
Exercise: Write the code for an Arithmetic Logic Unit (ALU). It must be able to perform the operations described in the table below. Input and output are numbers of 8 bits.



sel	Operation	Function	Unit
0000	y <= a	Transfer a	
0001	y <= a+1	Increment a	
0010	y <= a-1	Decrement a	
0011	y <= b	Transfer b	Arithmetic
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	
1001	y <= NOT b	Complement b	
1010	y <= a AND b	AND	
1011	y <= a OR b	OR	Logic
1100	y <= a NAND b	NAND	
1101	y <= a NOR b	NOR	
1110	y <= a XOR b	XOR	
1111	y <= a XNOR b	XNOR	

Concurrent Code
Exercices

- Using **concurrent** code (operators, WHEN, GENERATE) make the following circuits:
- 1.- **Encoder**: Input of 8-bits and output of 3bits. The three bits should show the total number of high (1) bits at the input. For solving this use : (i) only operators and (ii) WHEN/ELSE
- 2.- Create a circuit that convert binary (standard code where the number is based on 2^n) code to Gray Code (gray code only one-bit changes from one number to the next), as shows in the table:



Binary code	Gray code
0000	0000
0001	0001
0010	0011
0011	0010
0100	0110
0101	0111
0110	0101
0111	0100
1000	1100
1001	1101
1010	1111
1011	1110
1100	1010
1101	1011
1110	1001
1111	1000

POWER & ENERGY CONVERSION LABORATORY

PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE

END-Concurrent Code
WHEN, GENERATE, BLOCK

We study the staments required to create a concurrent code.

VHDL

VHSIC HARDWARE DESCRIPTION LANGUAGE