Ninevah University

Collage of Electronics Engineering

System and Control Department

Lecture No.: 1

Lecture title: Programmable Logic Devices

Submitted by: Dr. Hussein Aideen

Programmable Logic Devices

Digital Electronic systems consist of:

Memory,
Microprocessor,
Logic Devices:

* Fixed Function Logic Devices.
* **Programmable Logic Devices.**

A programmable logic device (PLD) is an electronic component used to build reconfigurable digital circuits.

Unlike a logic gate, a PLD has an undefined function at the time of manufacture.

There are two types of PLDs based on design architecture:

1- Simple PLDs.
2- Complex PLDs.

Software Programing:

Run on: Microprocessor
Architecture: Serial
Depend on: Instructions
Languages: C, C++, JAVA, Visual BASIC, Python, etc…

Hardware Programing:

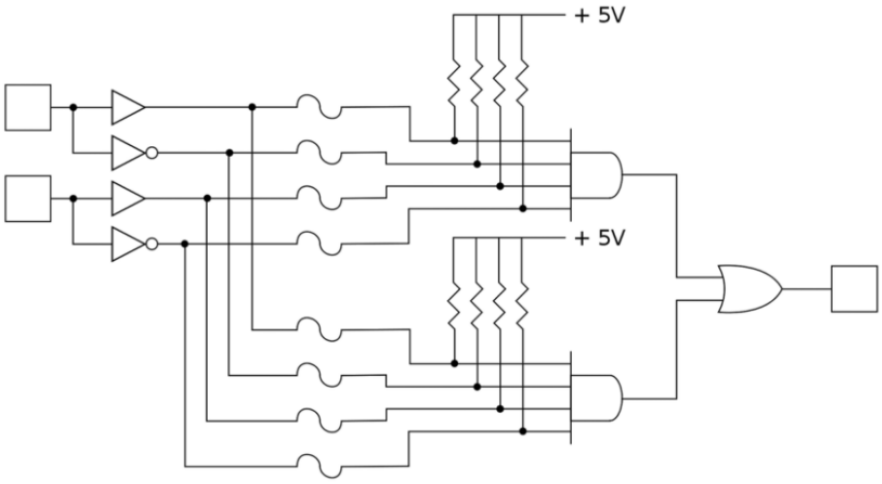
Run on: PLDs
Architecture: Parallel
Depend on: Hardware components
Languages: VHDL, Verilog, SystemC.

How PLDs be programed?

In this simple PLD we can implement the following logic functions: $F = AE + \bar{A}E$

$F = AE + \bar{A}\bar{E}$

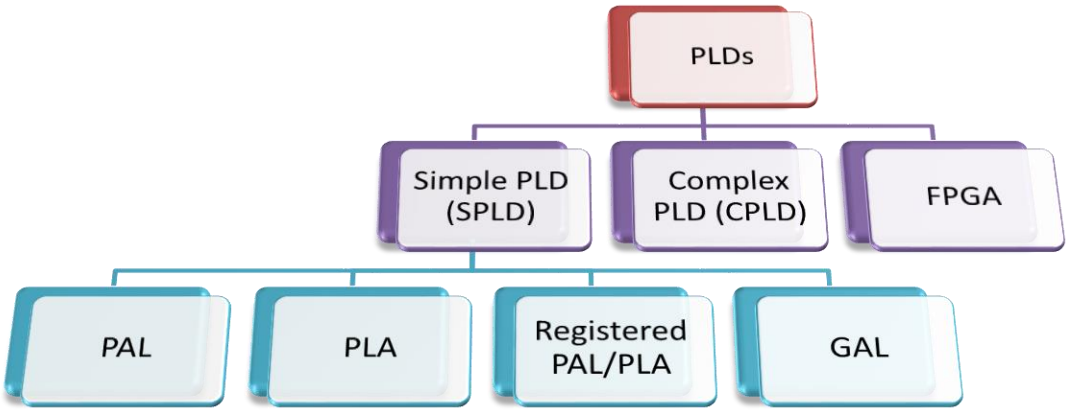
$F = A\bar{E} + \bar{A}E$ ……..



Advantages Programmable Logic Devices:

* Less board space.
* Easy to change with rewiring.
* High speed.
* Less cost.

PLDs types:



1- Programmable Array Logic (PAL):

- Consists of a <u>programmable</u> array of **AND gates**, followed by a <u>fixed</u> array of **OR gates**.
- The main limitation: allowed only the implementation of combinational functions and **Look-Up Table (LUT).**

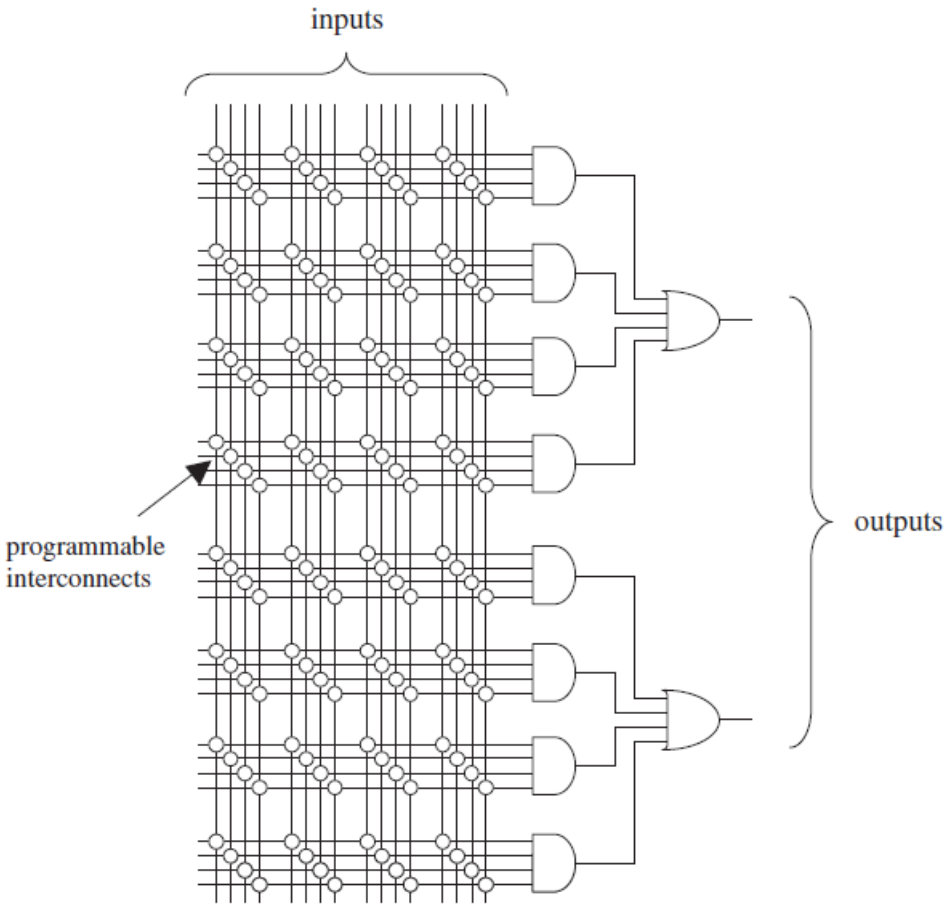


Figure A1
Illustration of PAL architecture.

2- Programmable Logic Array (PLA):

* Programmable array of **AND gates**, followed by a programmable array of **OR gates**.
* Advantage: has greater flexibility than PAL.
* Disadvantage: Higher time constants at the internal nodes lowered the circuit speed.

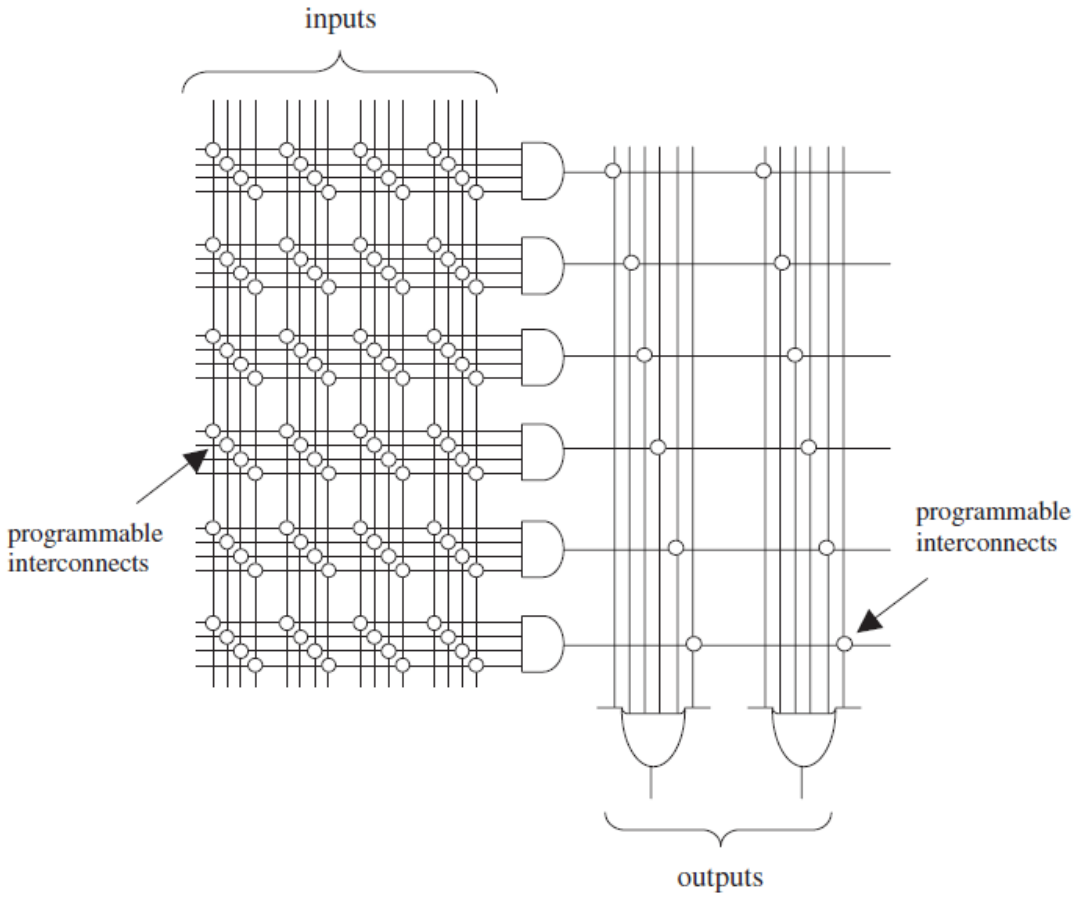
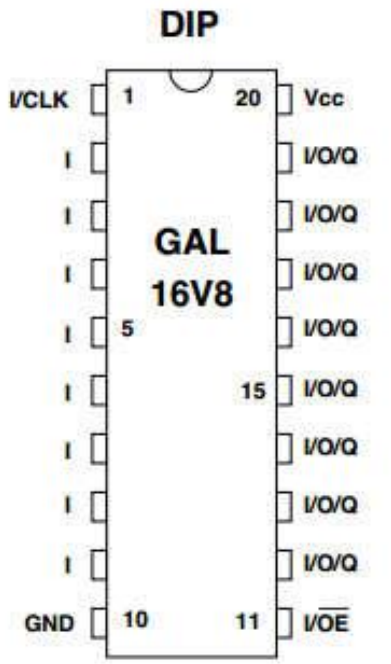


Figure A2
Illustration of PLA architecture.

3- **Generic PAL (GAL):**

- Introduced by Lattice in 1980s.
- A more sophisticated output cell (**Macrocell**):
  - Included besides the flip-flop, several gates and multiplexers.
  - Macrocell itself was programmable.
  - A ‘return’ signal from the output of the Macrocell to the programmable array.
- **EEPROM** was employed instead of PROM or EPROM.



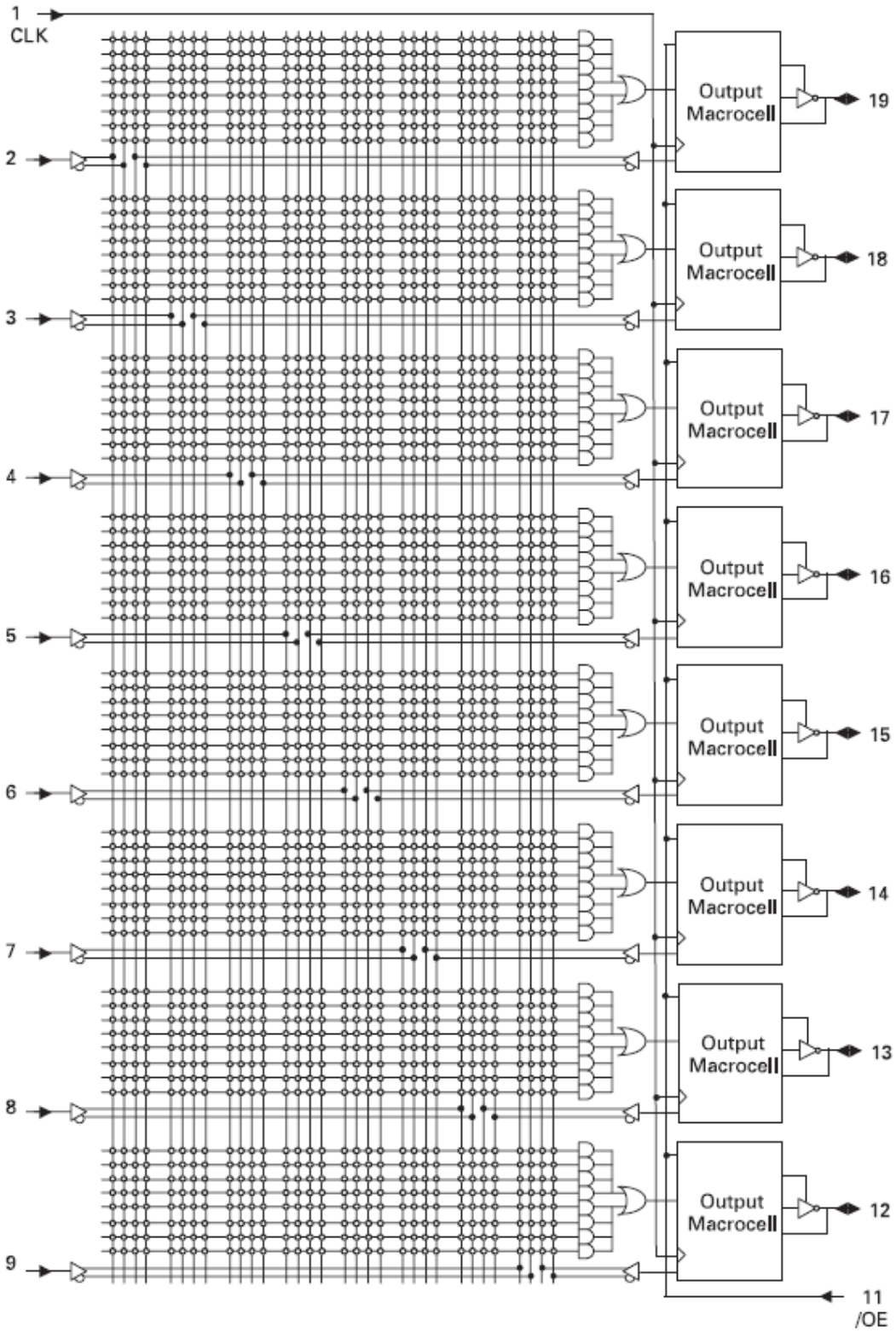


Figure A3
GAL 16V8 chip.

4- Complex PLD (**CPLD**):

- Several PLDs (in general of **GAL** type) fabricated on a single chip. With programmable switch matrix.
- Altera, Xilinx, Lattice, Atmel, Cypress, etc.

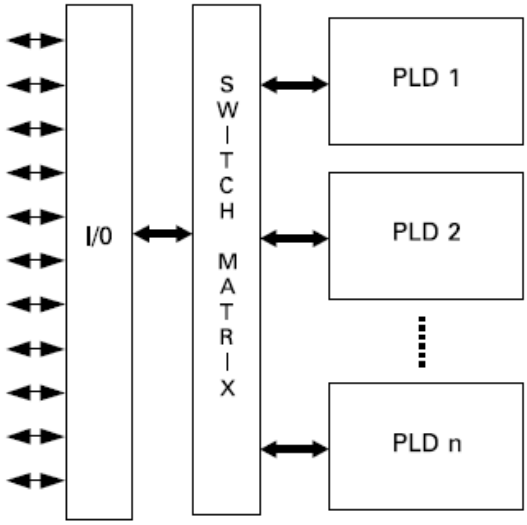


Figure A4
CPLD architecture.

* Applications:
  * Decoders
  * Encoders
  * Multiplexers
  * De-Multiplexers

Table A1
Altera CPLDs.

Family	*Max7000 (B, AE, S)*	*MAX3000 (A)*	*MAX II (G)*
Macrocells/ LUTs	32–512 macrocells	32–512 macrocells	240–2,210 LUTs (192–1,700 equiv. macrocells)
System gates	600–10,000	600–10,000	
I/O pins	32–512	34–208	80–272
Max. internal clock freq.	303 MHz	227 MHz	304 MHz (I/O limited)
Supply voltage	2.5 V (B), 3.3 V (AE), 5 V (S)	3.3 V	1.8 V (G), 2.5 V, 3.3 V
Interconnects	EEPROM	EEPROM	Flash + SRAM
Static current	9 mA–450 mA	9 mA–150 mA	2 mA–50 mA
Technology	0.22 u CMOS EEPROM 4-layer metal (7000 B)	0.3 u, 4-layer metal	0.18 u, 6-layer metal

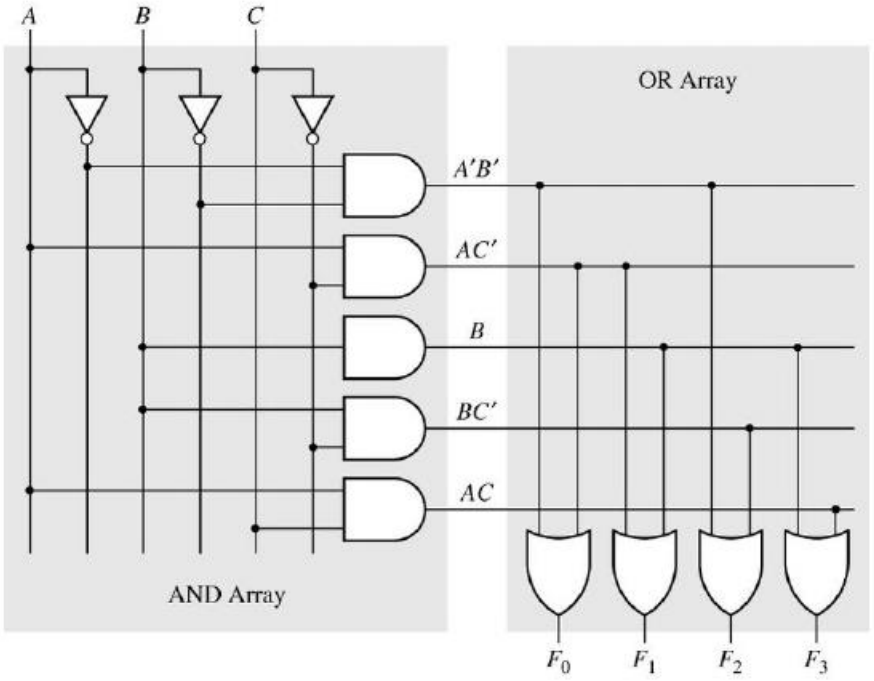
Table A2
Xilinx CPLDs.

Family	*XC9500 (XV, XL, –)*	*CoolRunner XPLA3*	*CoolRunner II*
Macrocells	36–288	32–512	32–512
System gates	800–6,400	750–12,000	750–12,000
I/O pins	34–192	36–260	33–270
Max. internal clock frequency	222 MHz	213 MHz	385 MHz
Building block	GAL 54V18 (XV, XL) GAL 36V18 (–)	PLA block	PLA block
Supply voltage	2.5 V (XV), 3.3 V (XL), 5 V	3.3 V	1.8 V
Interconnects	Flash	EEPROM	
Technology	0.35 u CMOS	0.35 u CMOS	0.18 u CMOS
Static current	11–500 mA	<0.1 mA	22 uA–1 mA

Tutorial sheet:

1- For the following Figure:

- Specify the device type: _____ (PAL, PLA, GAL)
- Find Boolean expression for each of the functions F0 – F3.

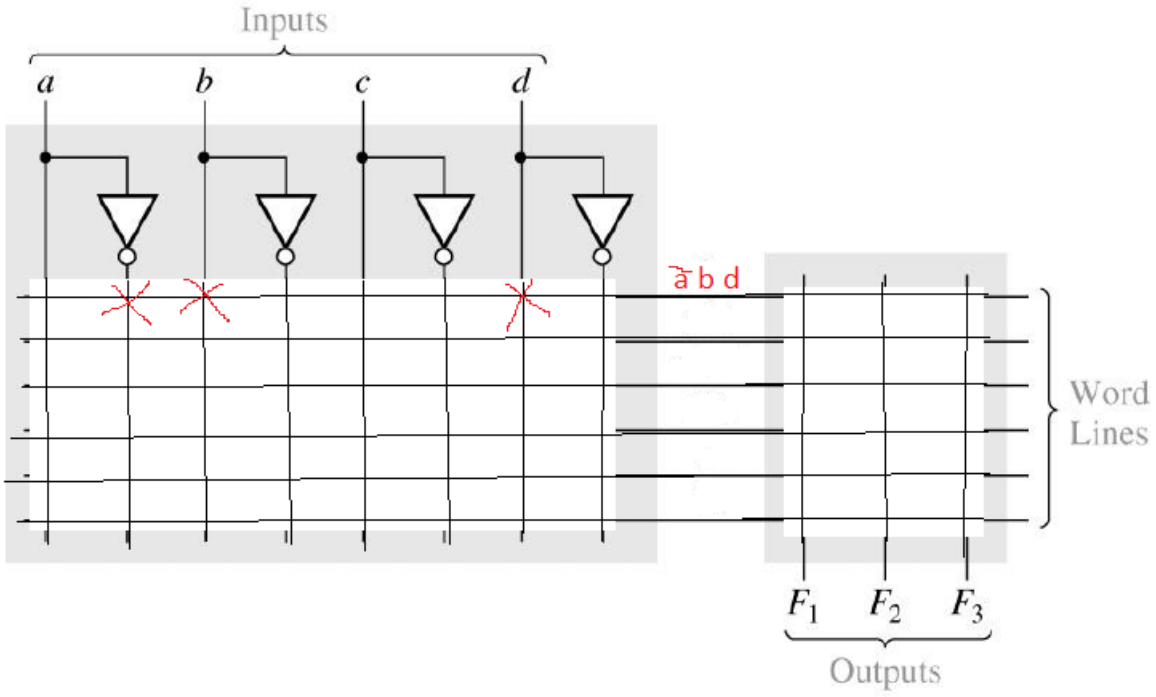


Answer:

2- For the following truth table, find a PLA realization.

a	b	c	d	f_1	f_2	f_3
0	1	-	1	1	1	0
1	1	-	1	1	0	1
1	0	0	-	1	0	1
-	0	1	-	1	0	0
-	-	1	-	0	1	0
-	1	1	-	0	0	1

Answer:



5- Field Programmable Gate Array (FPGA):

Introduced by Xilinx in the mid-1980s. Its differs from CPLDs in architecture, storage technology, number of built-in features, and cost. Aimed at the implementation of high performance, large-size circuits.

* **Applications**:
    * Aerospace and Defense.
    * Medical Electronics.
    * Wired Communications.
    * Wireless Communications.
    * High performance computing.

It consists of a **programmable** matrix of:

- Configurable Logic Blocks **(CLBs**),
- interconnected by an array of **switch matrices**.
- Array **Input/output Blocks**.
- **Clock Circuitry**.

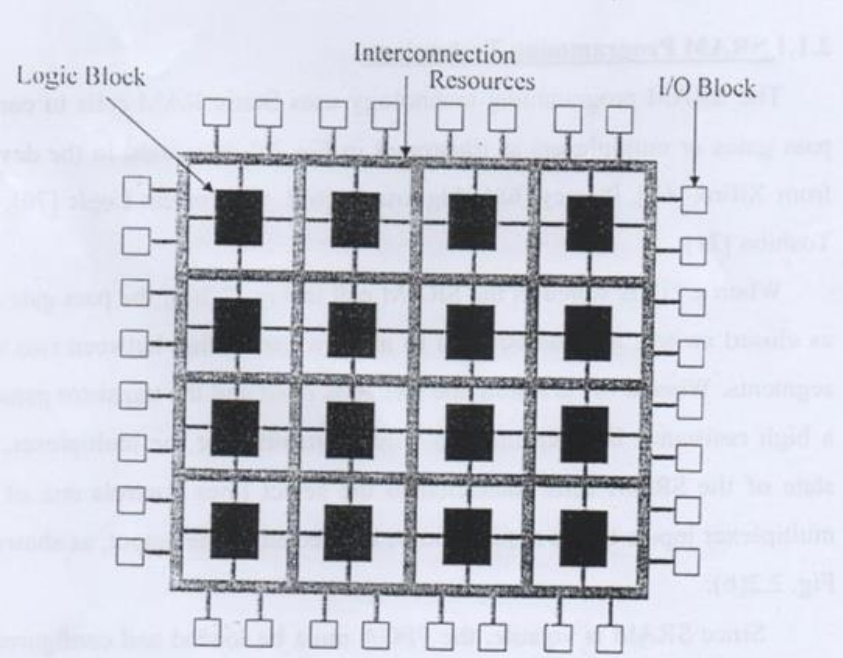
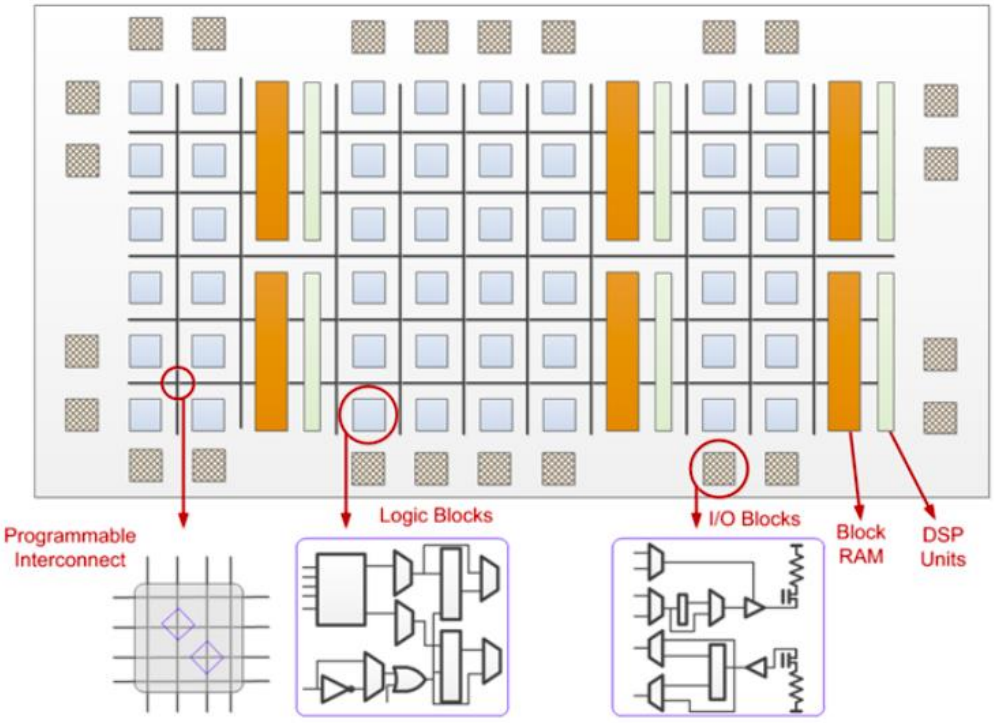


Fig. 2.1 FPGA architecture.



5.1 Configurable Logic Blocks (CLBs),

A CLB is the fundamental component of an FPGA, allowing the user to implement virtually any logical functionality within the chip. This is achieved by the usage of two sets of similar components within a block, known as slices.

There are two different types of slices:

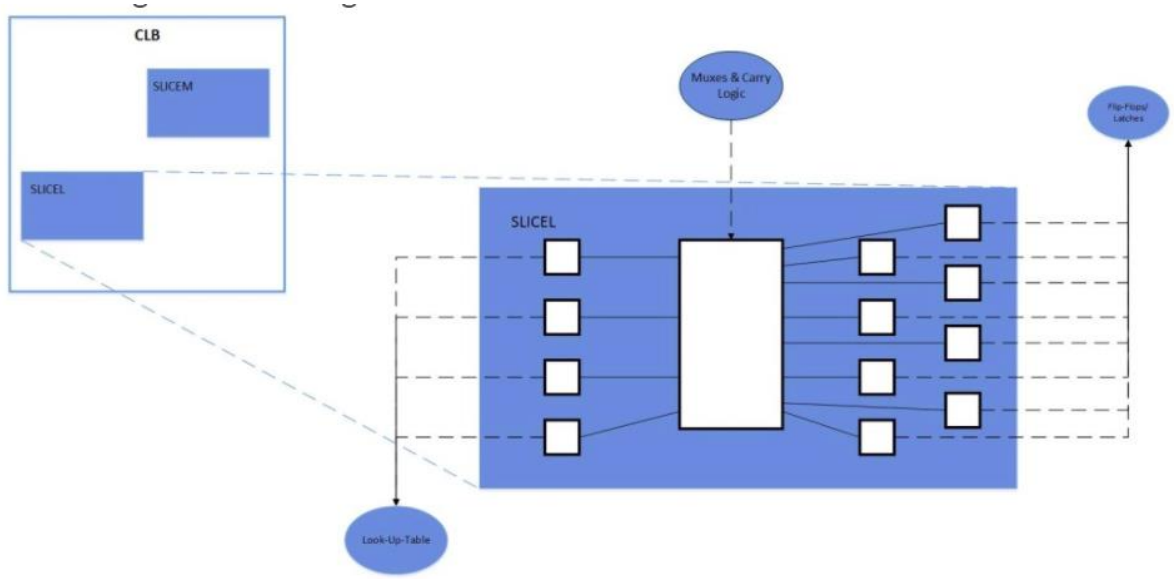
- **SLICEM:** ability to act as distributed memory in addition to its normal logic functionality.
- **SLICEL:** in addition to its normal logic functionality only.

These slices contain four look-up-tables (LUTs), eight flip-flops (FF), a network of carry logic, and three types of multiplexers.

CLB = 2 slices,

one slice = 4 LUTs + 8 FF.

Therefore, one CLB = 8 LUTs + 16FF.



CLB

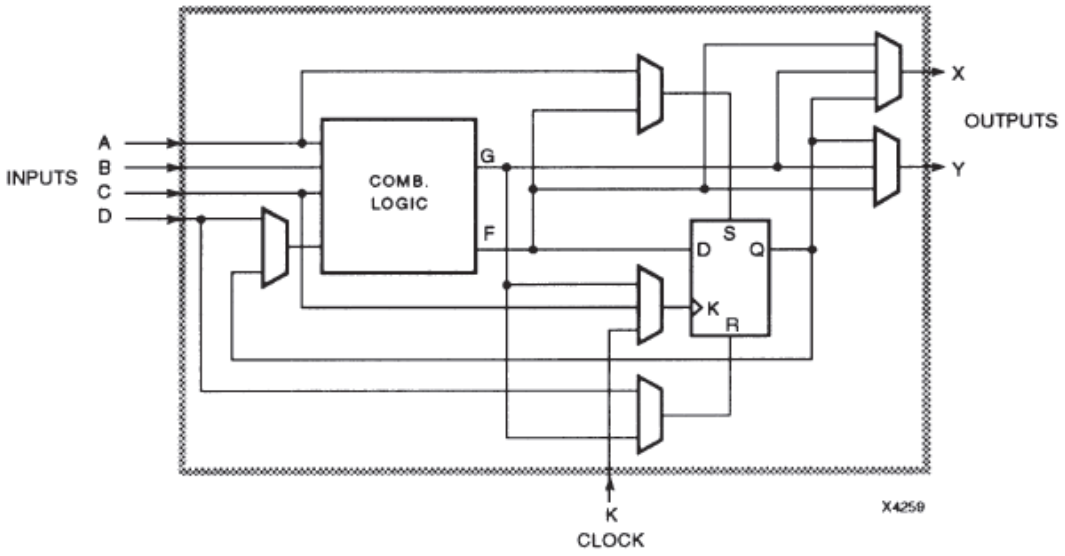


Figure 2.3.8. The XC2000 CLB.

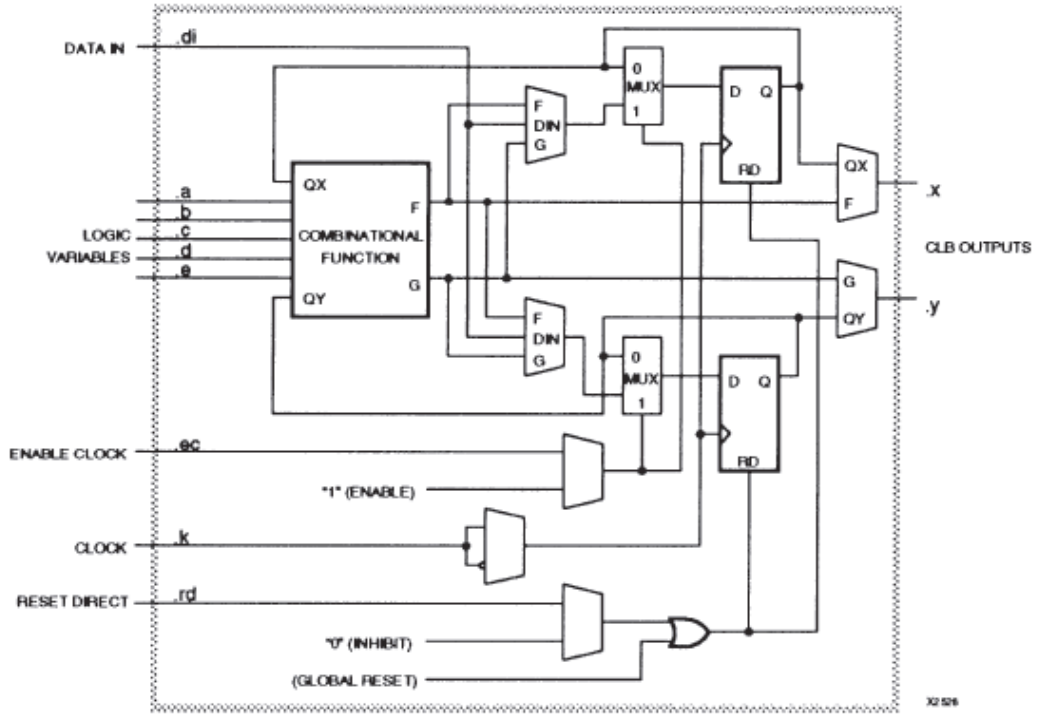


Figure 2.3.15. The XC3000 CLB.

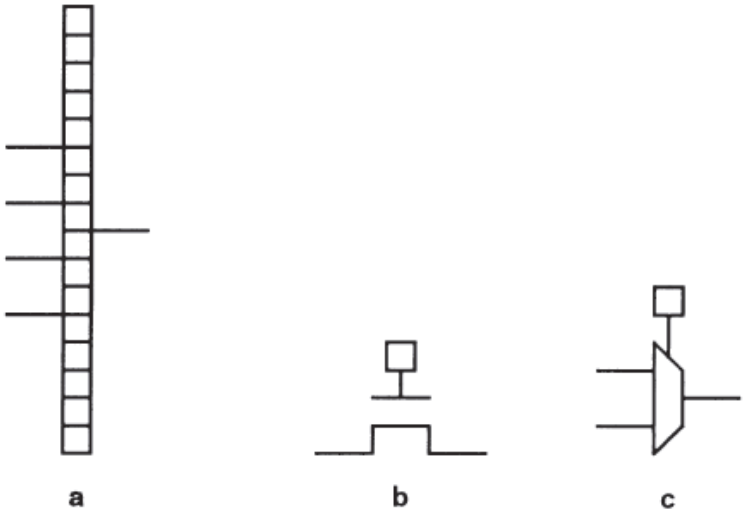


Figure 2.3.2. Three Important Pieces. a) Lookup Table. b) Pip. c) Multiplexer Controlled by a Configuration Memory Cell.

Lookup Table. Figure 2.3.2a shows a four-input *lookup table* (abbreviated *LUn* or *function generator*, a basic unit of configurable logic. A lookup table implements combinational logic as a 2^n *x* 1 memory composed of configuration memory cells. The memory is used as a lookup table, addressed by the *n* inputs. A lookup table can implement any of the 2^{2^n} functions of its inputs.

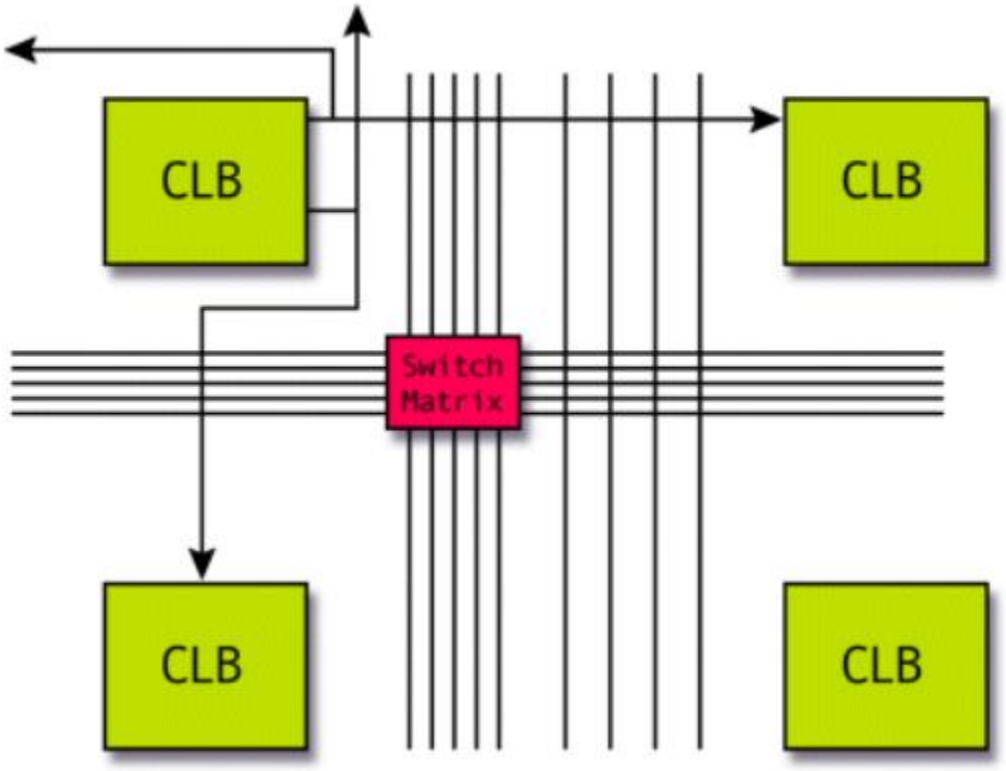
Programmable Interconnect Point. The second building block is called a programmable interconnect point, (Pip). Some authors use the term "configurable interconnect point" (cip). Pips control the connection of wiring segments in the programmable interconnect.

Multiplexer. The third building block is a multiplexer controlled by a configuration memory cell (figure 2.3.2c). The multiplexer is a special-case, one-directional routing structure. It may be of any width, with more configuration bits for wider multiplexers. Switches built with multiplexers reduce the number of memory cells required for controlling the switch, giving an area savings for large switches.

5.2 interconnection matrix (**switch matrices):**
There are **long lines** that can be used to connect critical CLBs that are physically far from each other on the chip without inducing much delay. Theses long lines can also be used as buses within the chip.

There are also **short lines** that are used to connect individual CLBs that are located physically close to each other.

Transistors are used to turn on or off connections between different lines.



Switch matrix

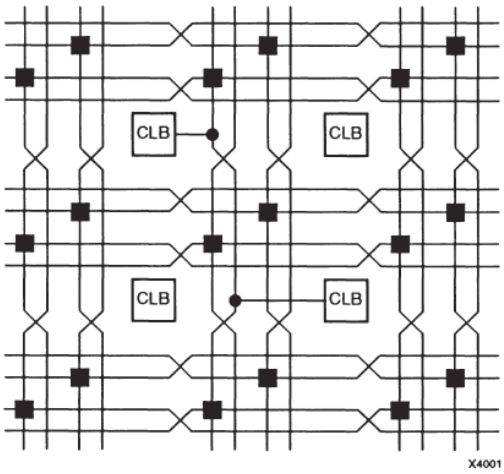
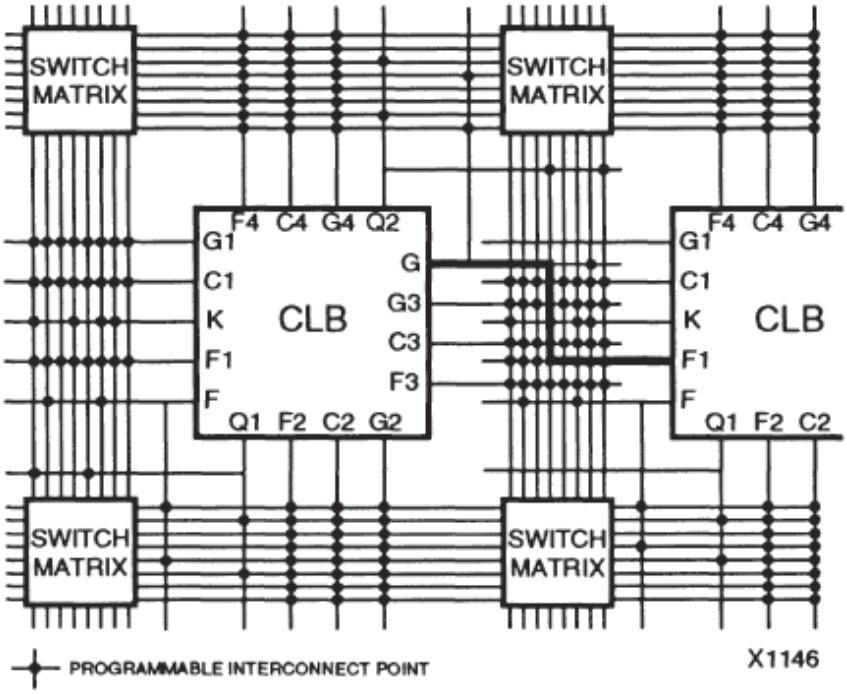


Figure 2.3.25. XC4000 Interconnect.



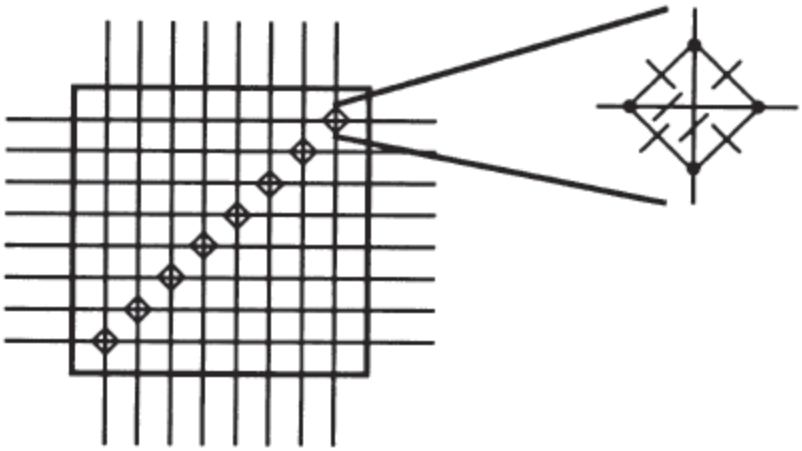


Figure 2.3.26. The XC4000 Switchbox Connections.

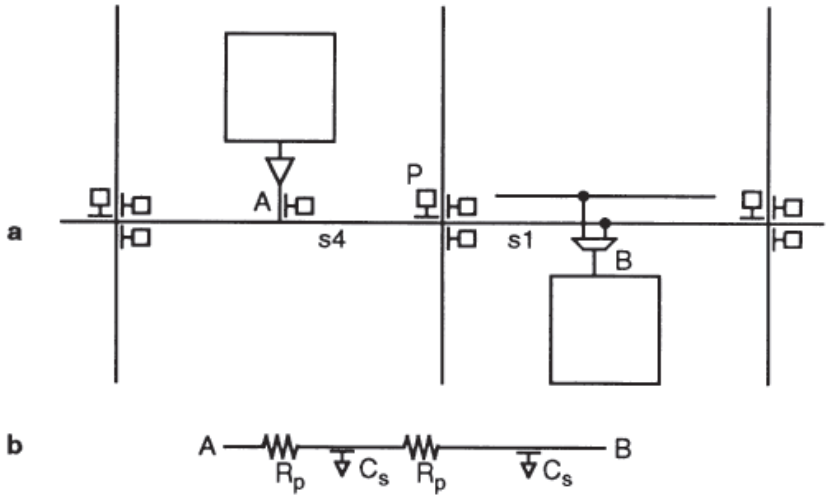


Figure 2.3.6. Interconnect Segment Detail. a) Architecture-Level. b) Electrical Equivalent.

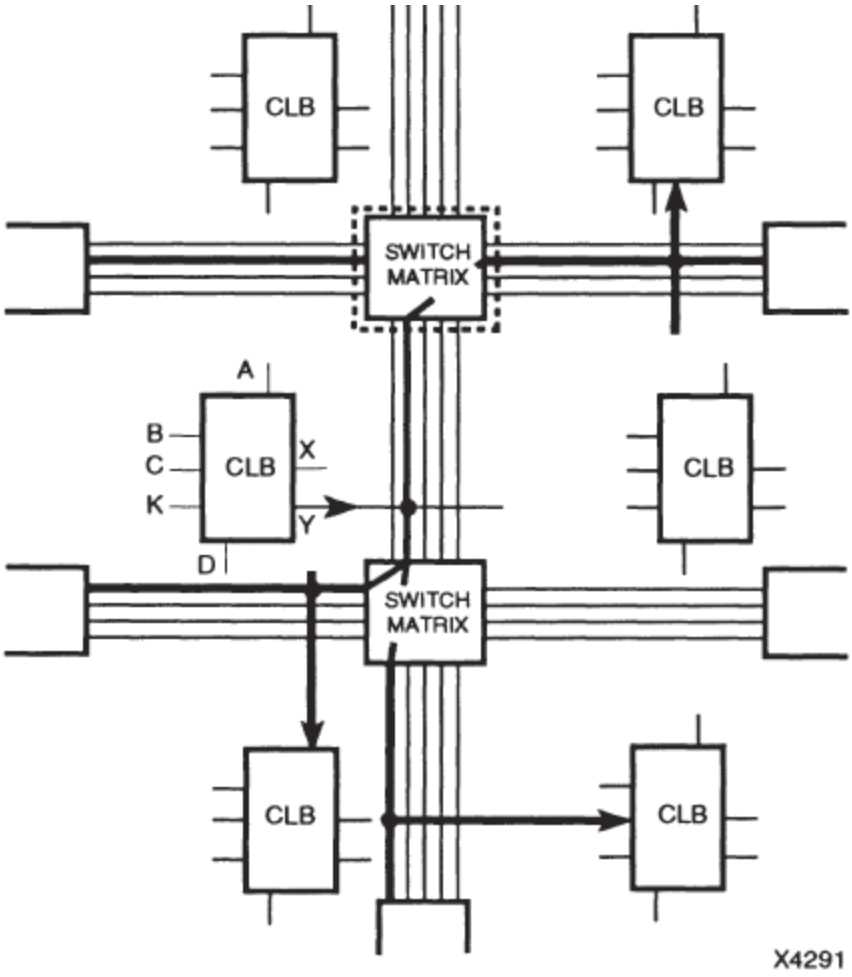
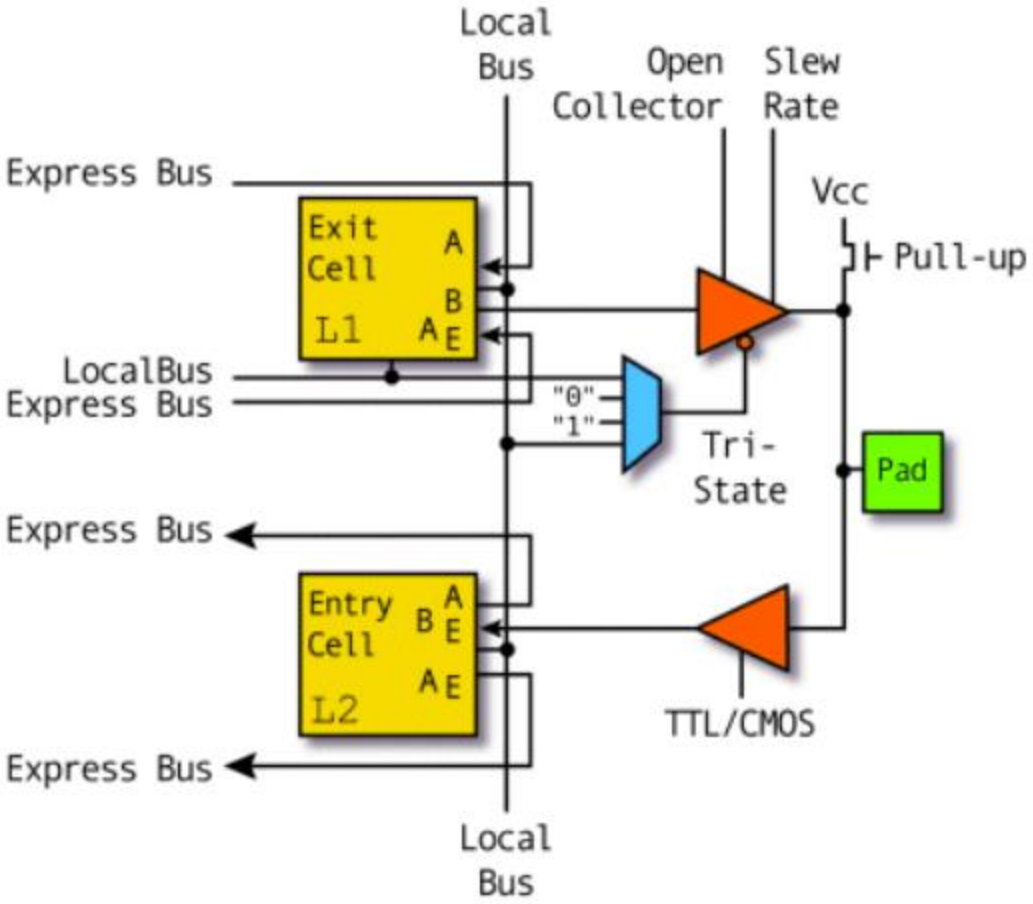


Figure 2.3.10. XC2000 Interconnect Structure.

5.3 Configurable I/O Blocks:

A Configurable input/output (I/O) Block is used to bring signals onto the chip and send them back off again. It consists of an input buffer and an output buffer with three-state and open collector output controls.


Local
Bus
Open
Collector
Slew
Rate
Express Bus
Vcc
Pull-up
Exit
Cell
L1
A
B
A
E
LocalBus
Express Bus
"0"
"1"
Tri-
State
Pad
Express Bus
Entry
Cell
L2
B
A
E
A
E
TTL/CMOS
Express Bus
Local
Bus

I/O Block

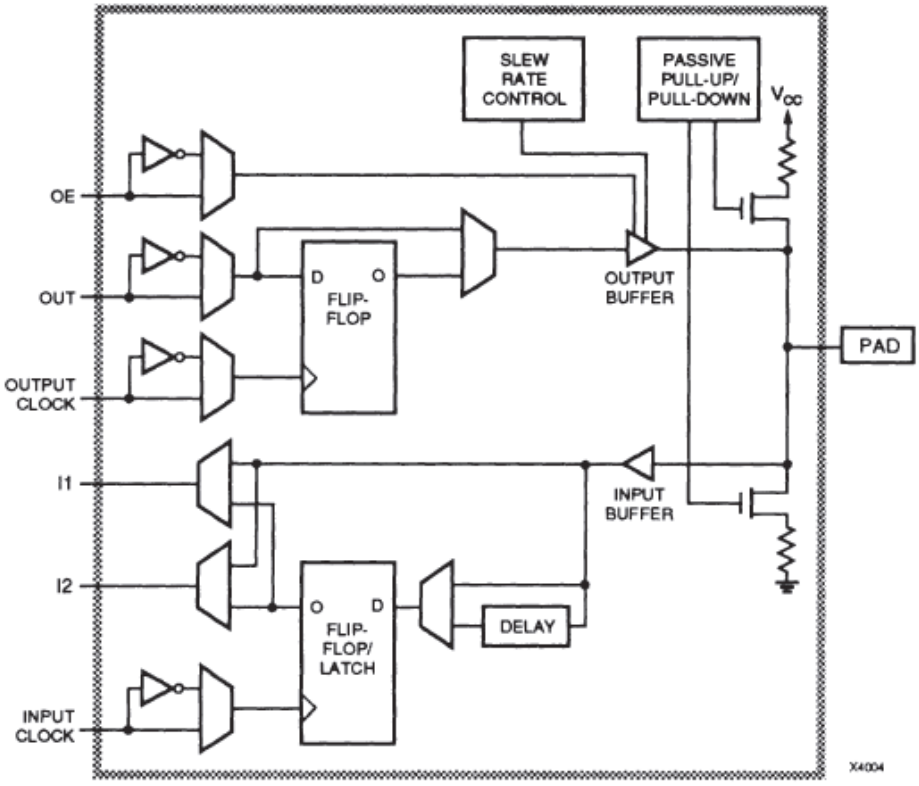


Figure 2.3.24. The XC4000 IOB.

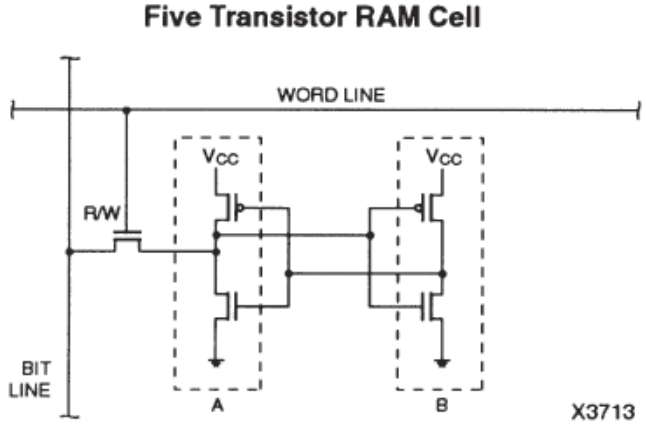
5.4 **Clock Circuitry:**

Special I/O blocks with special high drive clock buffers, known as clock drivers , are distributed around the chip. These buffers connect to clock input pads and drive the clock signals onto the global clock lines described above. These clock lines are designed for low skew times and fast propagation times. Note that synchronous design is a must with FPGAs, since absolute skew and delay cannot be guaranteed anywhere but on the global clock lines.

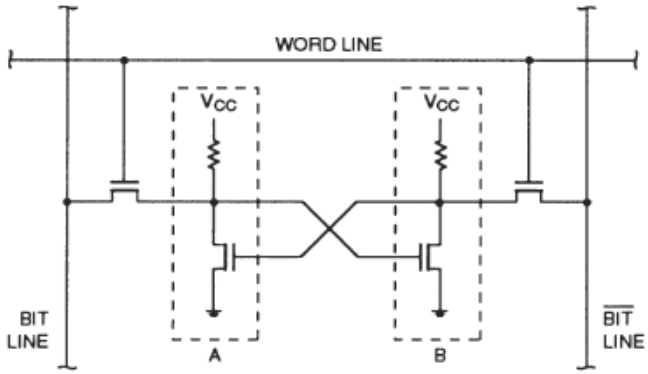
5.5 Technologies for programming FPGAs:

- SRAM programming: involves a small static RAM bit for each programming element.
- flash EPROM bits for each programming element.

a



b



c

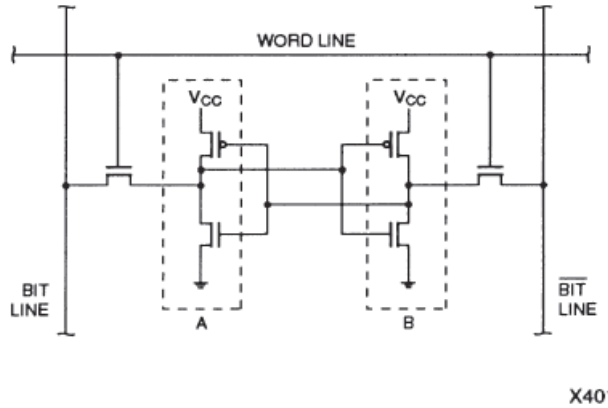


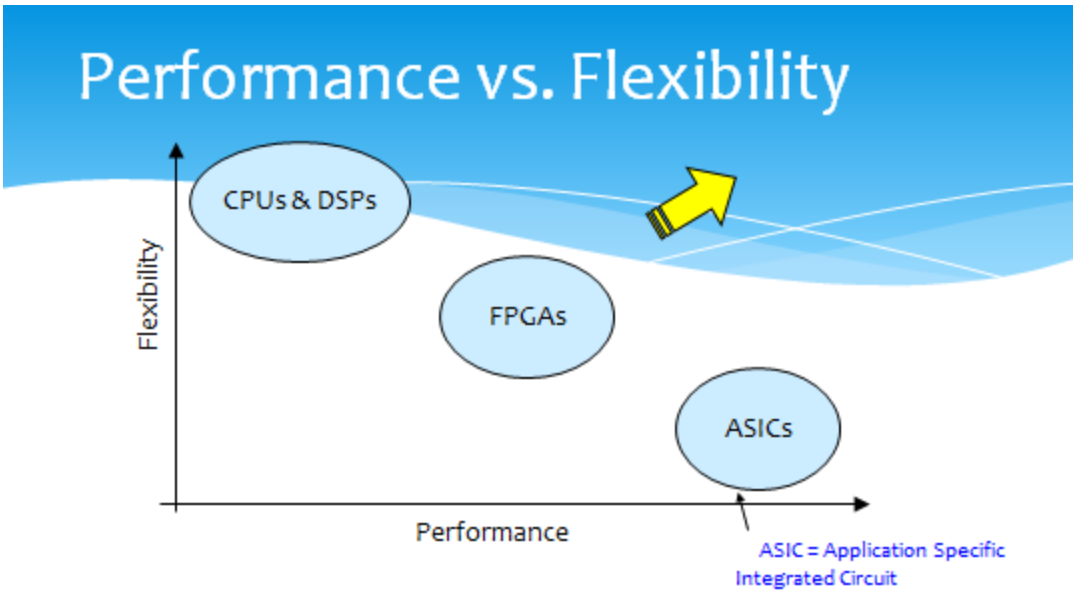
Figure 2.2.1. a) Xilinx Five-Transistor Configuration Memory Cell. b) Four-Transistor Memory Cell. c) Six-Transistor Memory Cell.

Example FPGA families

- Altera Stratix II and Cyclone II families
- Atmel AT6000 and AT40K families
- Lattice LatticeEC and LatticeECP families
- Xilinx Spartan-3 and Virtex-4 families.

Table A3
Xilinx FPGAs.

Family	*Virtex II Pro (X)*	*Virtex II*	*Virtex E*	*Virtex*	*Spartan 3*	*Spartan IIE*	*Spartan II*
Logic blocks (CLBs)	352–11,024	64–11,648	384–16,224	384–6,144	192–8,320	384–3,456	96–1,176
Logic cells	3,168–125,136	576–104,882	1,728–73,008	1,728–27,648	1,728–74,880	1,728–15,552	432–5,292
System gates		40 k–8 M	72 k–4 M	58 k–1.1 M	50 k–5 M	23 k–600 k	15 k–200 k
I/O pins	204–1,200	88–1108	176–804	180–512	124–784	182–514	86–284
Flip-flops	2,816–88,192	512–93,184	1,392–64,896	1,392–24,576	1,536–66,560	1,536–13,824	384–4,704
Max. internal frequency	547 MHz	420 MHz	240 MHz	200 MHz	326 MHz	200 MHz	200 MHz
Supply voltage	1.5 V	1.5 V	1.8 V	2.5 V	1.2 V	1.8 V	2.5 V
Interconnects	SRAM	SRAM	SRAM	SRAM	SRAM	SRAM	SRAM
Technology	0.13 u 9-layer copper CMOS	0.15 u 8-layer metal CMOS	0.18 u 6-layer metal CMOS	0.22 u 5-layer metal CMOS	0.09 u 8-layer metal CMOS		
SRAM bits (Block RAM)	216 k–8 M	72 k–3 M	64 k–832 k	32 k–128 k	72 k–1.8 M	32 k–288 k	16 k–56 k



Goal: the performance of ASIC's with the flexibility of programmable processors.

جامعة نينوى
كلية هندسة الإلكترونيات

HDL Programming
-VHDL-

Textbook: Volnei A. Pedroni, “Circuit Design with VHDL”, MIT Press London, England, 2004.

Submitted By: Hussein M. H. Aideen

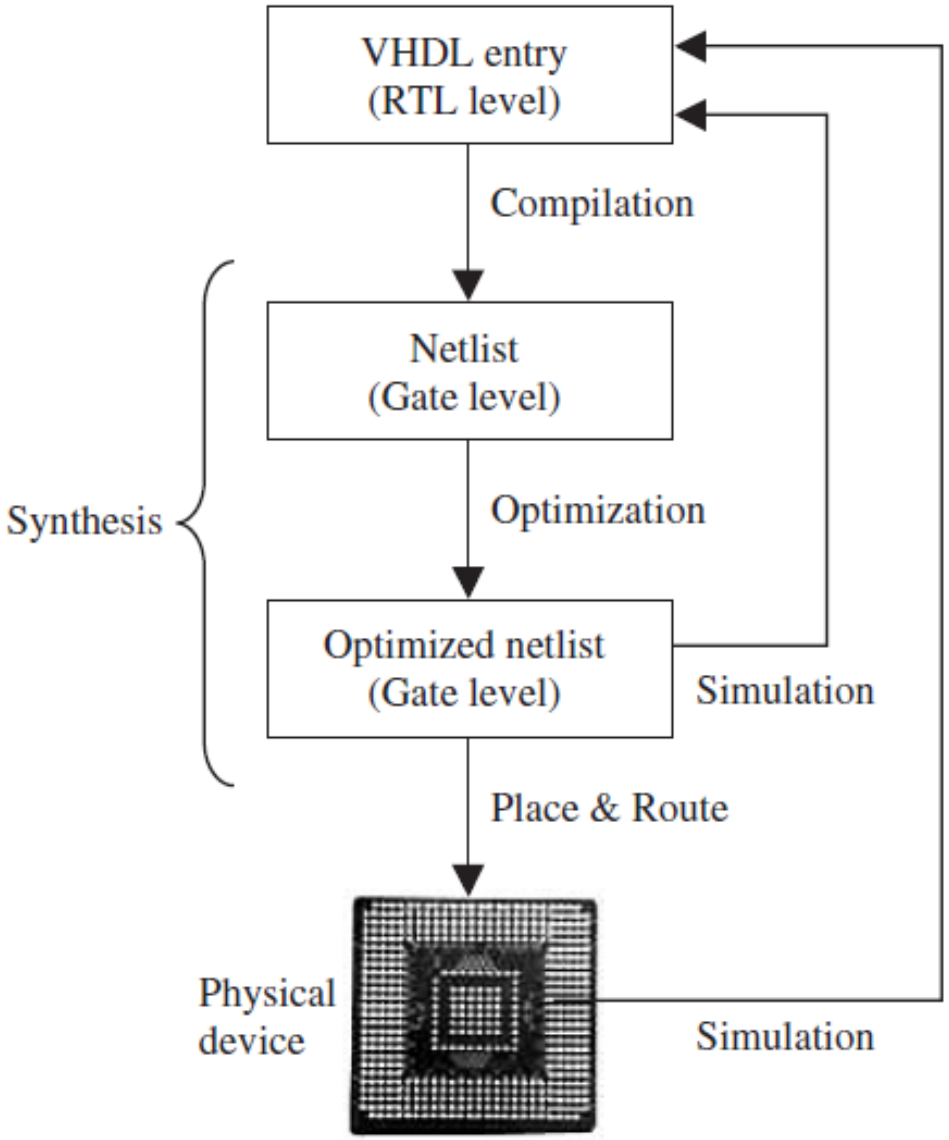
VHDL> Introduction

- **VHDL** stands for **VHSIC** Hardware Description Language.
- **VHSIC** is itself an abbreviation for Very High Speed Integrated Circuits.
- Describes the behavior of an electronic circuit or system, from which the physical circuit or system can then be implemented.
- first HDL standardized, IEEE 1076 standard.

VHDL> Introduction

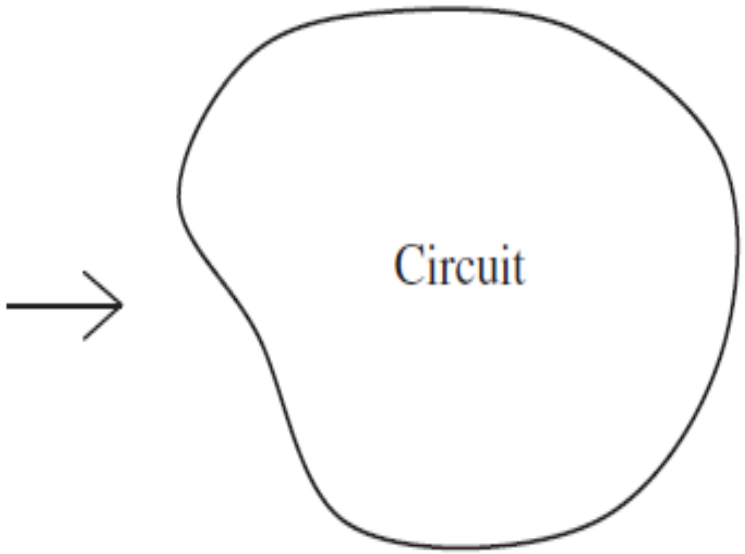
- Once the **VHDL** code has been written:
  - used either to implement the circuit in a programmable device (from Altera, Xilinx, Atmel, etc.)
  - or can be submitted to a foundry for fabrication of an **ASIC** chip.
- Currently, many complex commercial chips (microcontrollers, for example) are designed using such an approach.
- its statements are concurrent (parallel).

VHDL> Design Flow



VHDL> Design Flow

```
ENTITY full_adder IS
PORT (a, b, cin: IN BIT;
      s, cout: OUT BIT);
END full_adder;
---------------------------------------
ARCHITECTURE dataflow OF full_adder IS
BEGIN
   s <= a XOR b XOR cin;
   cout <= (a AND b) OR (a AND cin) OR
           (b AND cin);
END dataflow;
```



VHDL> Design Flow

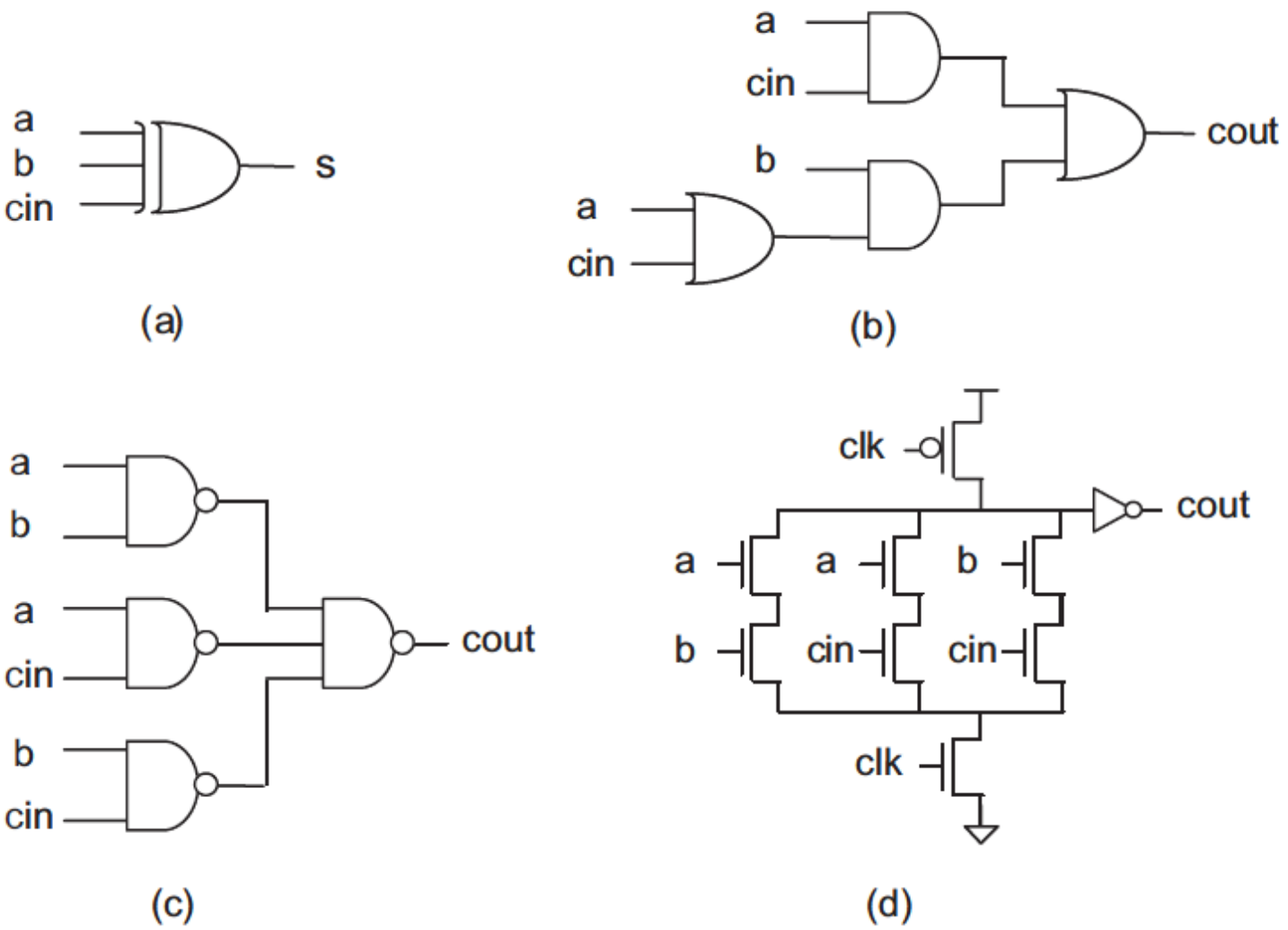


Figure 1.4
Examples of possible circuits obtained from the full-adder VHDL code of figure 1.3.

VHDL> Code Structure

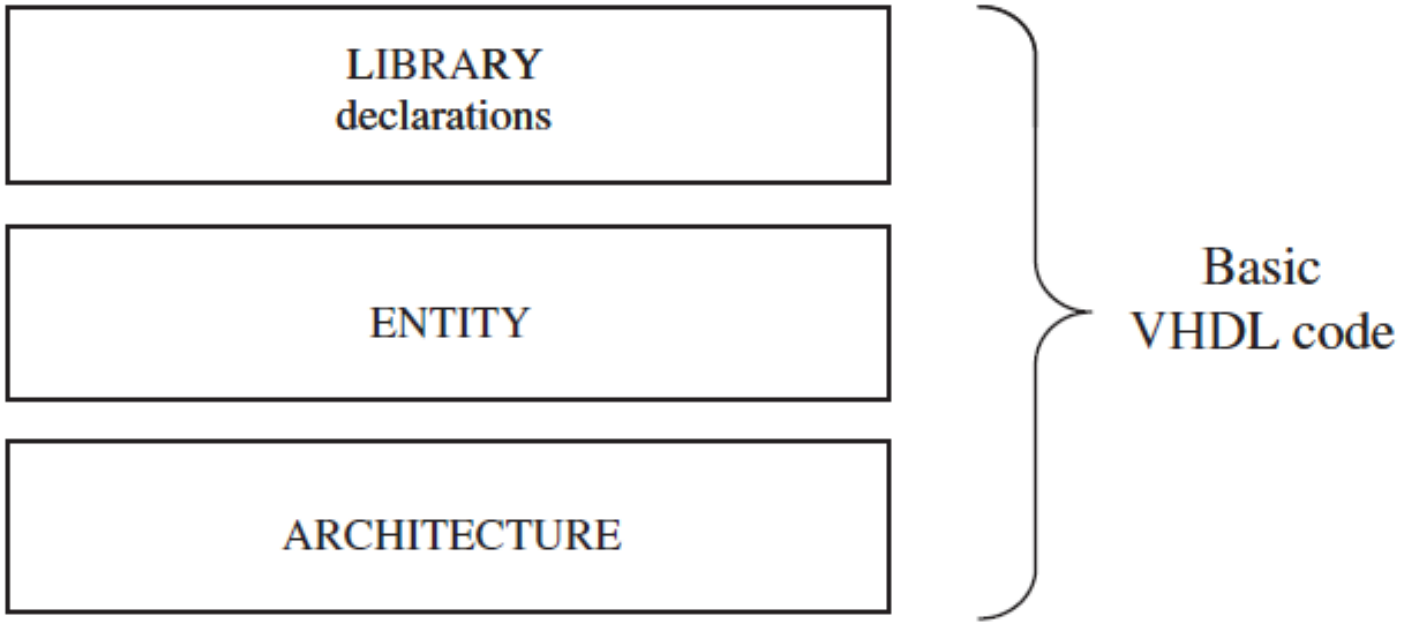


Figure 2.1
Fundamental sections of a basic VHDL code.

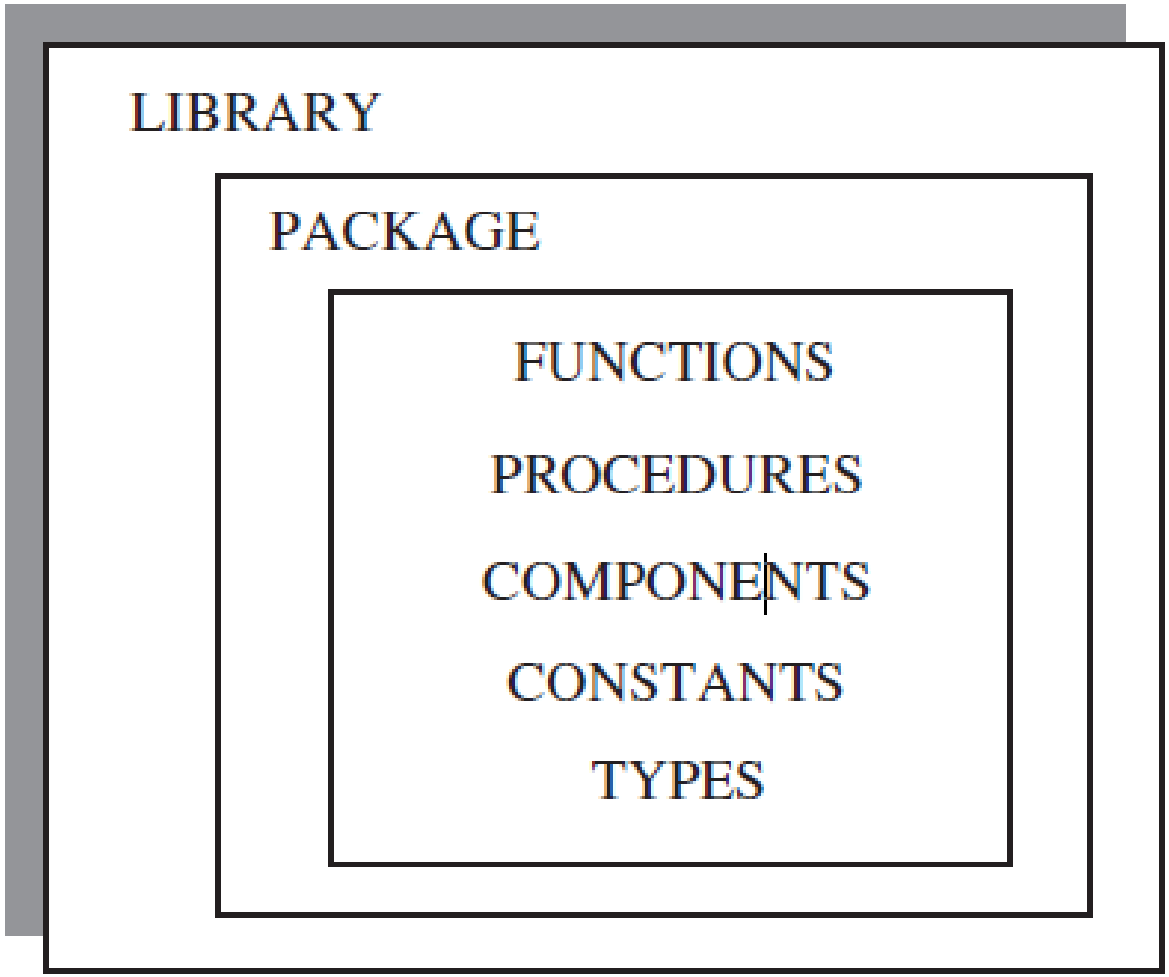
VHDL> Code Structure

LIBRARY declarations:
A LIBRARY is a collection of commonly used pieces of code. Placing such pieces inside a library allows them to be reused or shared by other designs.

```
LIBRARY library_name;
USE library_name.package_name.package_parts;
```

```
LIBRARY ieee;                       -- A semi-colon (;) indicates
USE ieee.std_logic_1164.all; -- the end of a statement or

LIBRARY std;                        -- declaration, while a double
USE std.standard.all;               -- dash (--) indicates a comment.

LIBRARY work;
USE work.all;

library UNISIM;
use UNISIM.VComponents.all;
```

VHDL> Code Structure



VHDL> Code Structure

An ENTITY is a list with specifications of all input and output pins (PORTS) of the circuit. Its syntax is shown below.

```
ENTITY entity_name IS
   PORT (
      port_name : signal_mode signal_type;
      port_name : signal_mode signal_type;
      ...);
END entity_name;
```

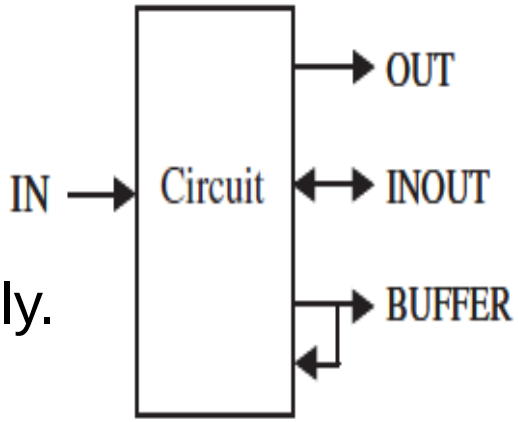
```
ENTITY nand_gate IS
PORT (a, b : IN BIT;
x : OUT BIT);
END nand_gate;
```

```
entity fulladder1 is
   Port ( a : in  STD_LOGIC;
          b : in  STD_LOGIC;
          cin : in  STD_LOGIC;
          s : out  STD_LOGIC;
          cout : out  STD_LOGIC);
end fulladder1;
```

VHDL> Code Structure

```
ENTITY entity_name IS
   PORT (
      port_name : signal_mode signal_type;
      port_name : signal_mode signal_type;
      ...);
END entity_name;
```

- The mode of the signal can be:
  - IN, OUT, INOUT, or BUFFER.
  - **IN** and **OUT** are truly unidirectional pins,
  - **INOUT** is bidirectional.
  - **BUFFER**, output signal must read internally.



- The type of the signal can be BIT, STD_LOGIC, INTEGER, etc. (discussed later).
- Finally, the name any name, except VHDL reserved words

VHDL> Code Structure

- The ARCHITECTURE is a description of how the circuit should behave (function). Its syntax is the following:

```
ARCHITECTURE architecture_name OF entity_name IS
    [declarations]
BEGIN
    (code)
END architecture_name;
```

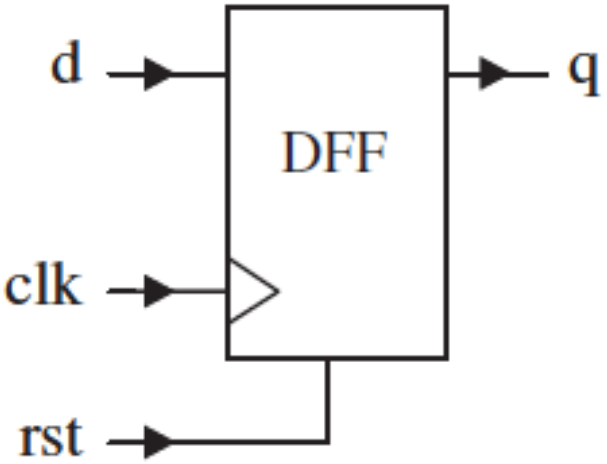
- declarative part (optional), where signals and constants are declared.

- the **code** part (from BEGIN down)

VHDL> Introductory Examples

- **DFF with Asynchronous Reset**

- Exam: Q1) Write a VHDL code to synthesis the following circuit (DFF with Asynchronous Reset) shown in figure below:



VHDL> Introductory Examples

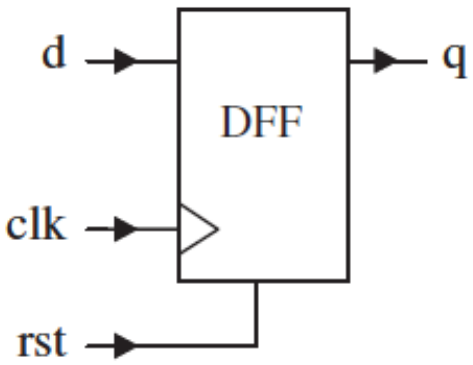
- **DFF with Asynchronous Reset**

- VHDL is inherently concurrent (contrary to regular computer programs, which are sequential),
- so to implement any clocked circuit (flip-flops, for example) we have to “force” VHDL to be sequential.

```
PROCESS (                    )
BEGIN
      (sequential code)
END PROCESS;
```

VHDL> Introductory Examples

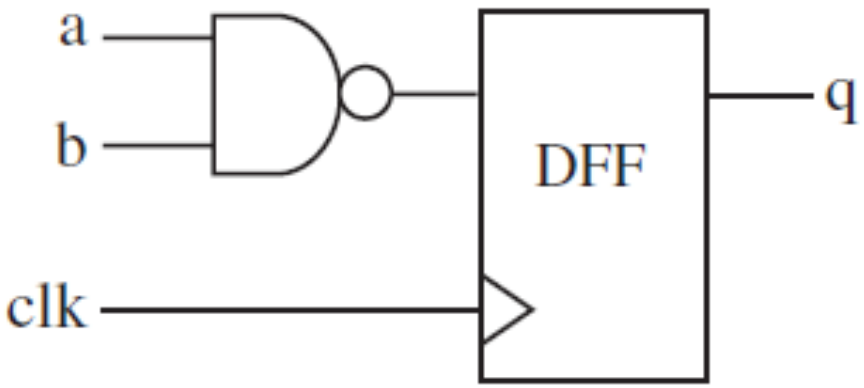
```
1  ----------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------
5  ENTITY dff IS
6  PORT ( d, clk, rst: IN STD_LOGIC;
7  q: OUT STD_LOGIC);
8  END dff;
9  ----------------------------------------
10 ARCHITECTURE behavior OF dff IS
11 BEGIN
12 PROCESS (rst, clk)
13 BEGIN
14 IF (rst='1') THEN
15 q <= '0';
16 ELSIF (clk'EVENT AND clk='1') THEN
17 q <= d;
18 END IF;
19 END PROCESS;
20 END behavior;
21 ----------------------------------------
```



VHDL> Introductory Examples

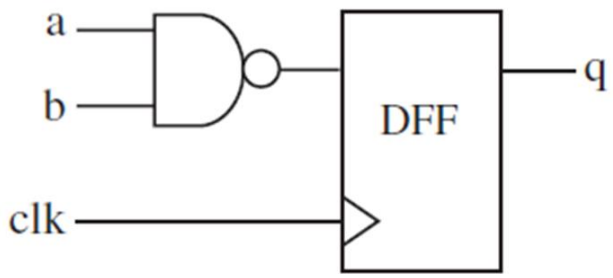
- **DFF plus NAND Gate**

- Exam: Q2) Write a VHDL code to synthesis the following circuit (DFF plus NAND Gate) shown in figure below:



VHDL> Introductory Examples

```
1  ----------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------
5  ENTITY DFFwNANDgate IS
6  PORT ( a, b, clk: IN BIT;
7         q: OUT BIT);
8  END DFFwNANDgate;
9  ----------------------------------------
10 ARCHITECTURE behavior OF DFFwNANDgate IS
11   SIGNAL temp : BIT;
12 BEGIN
13   temp <= a NAND b;
14   PROCESS (clk)
15   BEGIN
16      IF (clk'EVENT AND clk='1') THEN q<=temp;
17      END IF;
18   END PROCESS;
19 END behavior;
20 ----------------------------------------
```



VHDL> Introductory Examples

- **Home work 1**

- Q3) Write a VHDL code for the circuit of figure P2.2. Notice that it is purely combinational, so a PROCESS is not necessary. Write an expression for d using only logical operators (AND, OR, NAND, NOT, etc.).

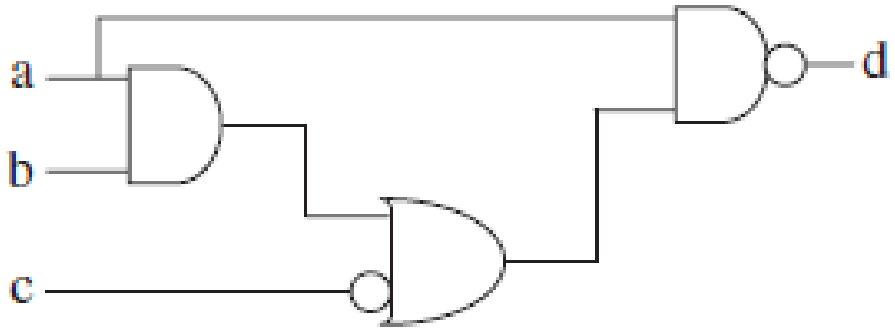
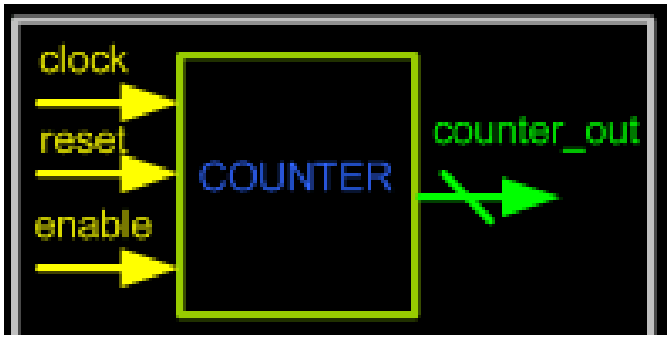


Figure P2.2

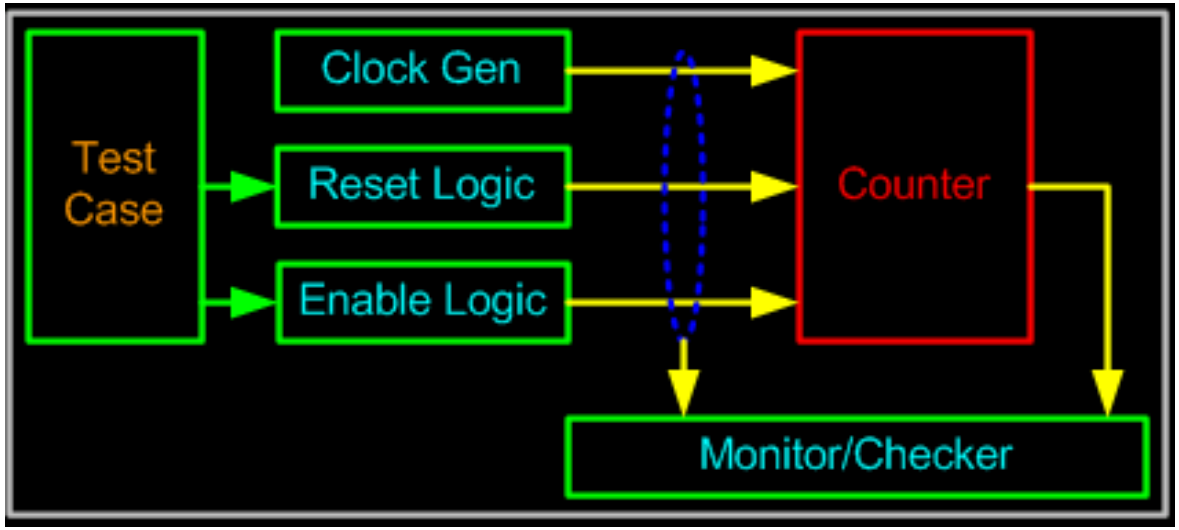


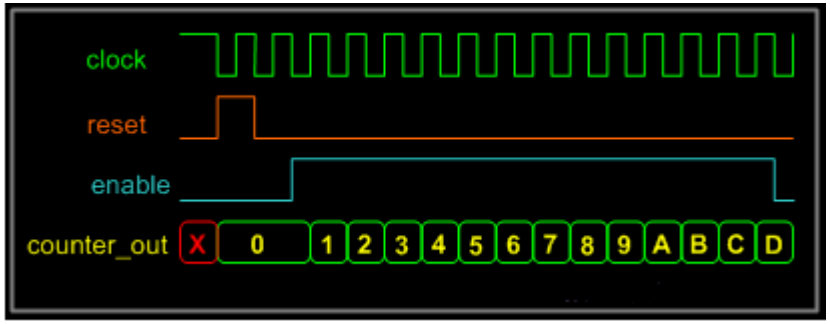

```
 1  library ieee ;
 2  use ieee.std_logic_1164.all;
 3  use ieee.std_logic_unsigned.all;
 4
 5  entity counter is
 6  port(  clk:  in std_logic;
 7    reset:  in std_logic;
 8    enable:  in std_logic;
 9    count:  out std_logic_vector(3 downto 0)
10  );
11  end counter;
12
13  architecture behav of counter is
14    signal pre_count: std_logic_vector(3 downto 0);
15    begin
16      process(clk, enable, reset)
17      begin
18        if reset = '1' then
19          pre_count <= "0000";
20        elsif (clk='1' and clk'event) then
21          if enable = '1' then
22            pre_count <= pre_count + "1";
23          end if;
24        end if;
25      end process;
26      count <= pre_count;
27  end behav;
```

Counter Test Bench

- Any digital circuit, no matter how complex, needs to be tested. For the counter logic, we need to provide a clock and reset logic. Once the counter is out of reset, we toggle the enable input to the counter, and check the waveform to see if the counter is counting correctly. The same is done in VHDL

Counter Test Bench





Testbench file

```
 1  library ieee ;
 2  use ieee.std_logic_1164.all;
 3  use ieee.std_logic_unsigned.all;
 4  use ieee.std_logic_textio.all;
 5  use std.textio.all;
 6
 7  entity counter_tb is
 8  end;
 9
10  architecture counter_tb of counter_tb is
11
12  COMPONENT counter
13    PORT ( count : OUT std_logic_vector(3 downto 0);
14           clk   : IN std_logic;
15           enable: IN std_logic;
16           reset : IN std_logic);
17  END COMPONENT ;
18
19  SIGNAL   clk    : std_logic := '0';
20  SIGNAL   reset  : std_logic := '0';
21  SIGNAL   enable : std_logic := '0';
22  SIGNAL   count  : std_logic_vector(3 downto 0);
23
24  begin
```

```
25
26  dut : counter
27  PORT MAP (
28     count => count,
29     clk   => clk,
30     enable=> enable,
31     reset => reset );
32
33    clock : PROCESS
34    begin
35       wait for 1 ns; clk  <= not clk;
36    end PROCESS clock;
37
38    stimulus : PROCESS
39    begin
40      wait for 5 ns; reset  <= '1';
41      wait for 4 ns; reset  <= '0';
42      wait for 4 ns; enable <= '1';
43      wait;
44    end PROCESS stimulus;
45
```

VHDL> Data Types

- Pre-Defined Data Types:
  - **BIT** (and **BIT_VECTOR**): 2-level logic (‘0’, ‘1’).

```
SIGNAL x: BIT;

-- x is declared as a one-digit signal of type BIT.

SIGNAL y: BIT_VECTOR (3 DOWNTO 0);

-- y is a 4-bit vector, with the leftmost bit being the MSB.

SIGNAL w: BIT_VECTOR (0 TO 7);

-- w is an 8-bit vector, with the rightmost bit being the MSB.
```

VHDL> Data Types

```
x <= '1';

-- x is a single-bit signal (as specified above), whose value is
-- '1'. Notice that single quotes (' ') are used for a single bit.

y <= "0111";

-- y is a 4-bit signal (as specified above), whose value is "0111"
-- (MSB='0'). Notice that double quotes (" ") are used for
-- vectors.

w <= "01110001";

-- w is an 8-bit signal, whose value is "01110001" (MSB='1').
```

VHDL> Data Types

- **STD_LOGIC** (and **STD_LOGIC_VECTOR**):

8-valued logic system introduced in the IEEE 1164 standard.

  - ‘X’ Forcing Unknown (synthesizable unknown)
  - ‘0’ Forcing Low (synthesizable logic ‘1’)
  - ‘1’ Forcing High (synthesizable logic ‘0’)
  - ‘Z’ High impedance (synthesizable tri-state buffer)
  - ‘W’ Weak unknown
  - ‘L’ Weak low
  - ‘H’ Weak high
  - ‘–’ Don’t care

VHDL> Data Types

```
SIGNAL x: STD_LOGIC;

-- x is declared as a one-digit (scalar) signal of type STD_LOGIC.

SIGNAL y: STD_LOGIC_VECTOR (3 DOWNTO 0) := "0001";

-- y is declared as a 4-bit vector, with the leftmost bit being
-- the MSB. The initial value (optional) of y is "0001". Notice
-- that the ":=" operator is used to establish the initial value.
```

VHDL> Data Types

- **BOOLEAN**: True, False.
- **INTEGER**: 32-bit integers (from -2,147,483,648 to +2,147,483,647).
- **NATURAL**: Non-negative integers (from 0 to +2,147,483,647).
- **SIGNED and UNSIGNED**: data types defined in the std_logic_arith package of the ieee library. They have the appearance of STD_LOGIC_VECTOR, but accept arithmetic operations, which are typical of INTEGER data types.
- **REAL**: Real numbers ranging from -1.0E38 to +1.0E38. Not synthesizable.
- **Physical literals**: Used to inform physical quantities, like time, voltage, etc. Useful in simulations. Not synthesizable.

VHDL> Data Types

- Examples:

```
x0 <= '0'; -- bit, std_logic, or std_ulogic value '0'

x1 <= "00011111"; -- bit_vector, std_logic_vector,
                  -- std_ulogic_vector, signed, or unsigned

x2 <= "0001_1111"; -- underscore allowed to ease visualization

x3 <= "101111" -- binary representation of decimal 47
```

VHDL> Data Types

- Examples:

```
x4 <= B"101111" -- binary representation of decimal 47
x5 <= O"57" -- octal representation of decimal 47
x6 <= X"2F" -- hexadecimal representation of decimal 47

n <= 1200; -- integer

m <= 1_200; -- integer, underscore allowed

IF ready THEN... -- Boolean, executed if ready=TRUE

y <= 1.2E-5; -- real, not synthesizable

q <= d after 10 ns; -- physical, not synthesizable
```

VHDL> Data Types

```
SIGNAL a: BIT;
SIGNAL b: BIT_VECTOR(7 DOWNTO 0);
SIGNAL c: STD_LOGIC;
SIGNAL d: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL e: INTEGER RANGE 0 TO 255;
...
a <= b(5); -- legal (same scalar type: BIT)
b(0) <= a; -- legal (same scalar type: BIT)
c <= d(5); -- legal (same scalar type: STD_LOGIC)
d(0) <= c; -- legal (same scalar type: STD_LOGIC)

a <= c; -- illegal (type mismatch: BIT x STD_LOGIC)
b <= d; -- illegal (type mismatch: BIT_VECTOR x
        -- STD_LOGIC_VECTOR)
e <= b; -- illegal (type mismatch: INTEGER x BIT_VECTOR)
e <= d; -- illegal (type mismatch: INTEGER x
        -- STD_LOGIC_VECTOR)
```

Ex: Write VHDL code to design 8-3 encoder (use if statement)

```vhdl
library ieee;
    use ieee.std_logic_1164.all;

entity encoder_using_if is
    port (
        enable     :in  std_logic;                          --  Enable for the encoder
        encoder_in :in  std_logic_vector (7 downto 0); --  16-bit Input
        binary_out :out std_logic_vector (2 downto 0)    --  4 bit binary Output

    );
end entity;

architecture behavior of encoder_using_if is

begin
    process (enable, encoder_in) begin
        binary_out <= "00";
        if (enable = '1') then
            if (encoder_in = X"02") then binary_out <= "001"; end if;
            if (encoder_in = X"04") then binary_out <= "010"; end if;
            if (encoder_in = X"08") then binary_out <= "011"; end if;
            if (encoder_in = X"10") then binary_out <= "100"; end if;
            if (encoder_in = X"20") then binary_out <= "101"; end if;
            if (encoder_in = X"40") then binary_out <= "110"; end if;
            if (encoder_in = X"80") then binary_out <= "111"; end if;

        end if;
    end process;
end architecture;
```

VHDL> Data Types

- User-Defined Data Types:
  - VHDL also allows the user to define his/her own data types.

```
TYPE integer IS RANGE -2147483647 TO +2147483647;
-- This is indeed the pre-defined type INTEGER.

TYPE natural IS RANGE 0 TO +2147483647;
-- This is indeed the pre-defined type NATURAL.

TYPE my_integer IS RANGE -32 TO 32;
-- A user-defined subset of integers.

TYPE student_grade IS RANGE 0 TO 100;
-- A user-defined subset of integers or naturals.
```

VHDL> Data Types

```
TYPE bit IS ('0', '1');
-- This is indeed the pre-defined type BIT

TYPE my_logic IS ('0', '1', 'Z');
-- A user-defined subset of std_logic.

TYPE state IS (idle, forward,
               backward, stop);
-- An enumerated data type, typical of
--    finite state machines.

TYPE color IS (red, green, blue, white);
-- Another enumerated data type.
```

VHDL> Data Types

- Sub-Types:
  - The main reason for using a subtype rather than specifying a new type is that, though operations between data of different types are not allowed, they are allowed between a subtype and its corresponding base type.

```
SUBTYPE natural IS INTEGER RANGE 0 TO INTEGER'HIGH;
-- As expected, NATURAL is a subtype (subset) of INTEGER.

SUBTYPE my_logic IS STD_LOGIC RANGE '0' TO 'Z';
-- Recall that STD_LOGIC=('X','0','1','Z','W','L','H','-').
 -- Therefore, my_logic=('0','1','Z').

SUBTYPE my_color IS color RANGE red TO blue;
-- Since color=(red, green, blue, white), then
-- my_color=(red, green, blue).

SUBTYPE small_integer IS INTEGER RANGE -32 TO 32;
-- A subtype of INTEGER.
```

VHDL> Data Types

- Signed and Unsigned Data Types:
  - defined in the *std_logic_arith* package of the *ieee* library.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;    -- extra package necessary
...
SIGNAL a: IN SIGNED (7 DOWNTO 0);
SIGNAL b: IN SIGNED (7 DOWNTO 0);
SIGNAL x: OUT SIGNED (7 DOWNTO 0);
...
v <= a + b;       -- legal (arithmetic operation OK)
w <= a AND b;     -- illegal (logical operation not OK)
```

VHDL> Data Types

Example: Legal and illegal operations with std_logic_vector.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;    -- no extra package required
...
SIGNAL a: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL x: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
...
v <= a + b;       -- illegal (arithmetic operation not OK)
w <= a AND b;     -- legal (logical operation OK)
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;    -- extra package included

SIGNAL a: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL x: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
...
v <= a + b;       -- legal (arithmetic operation OK), unsigned
w <= a AND b;     -- legal (logical operation OK)
```

VHDL> Data Conversion

- VHDL does not allow direct operations between data of different types.
- it is necessary to convert data from one type to another.

- If the data are closely related: std_logic_1164 of the ieee library provides straightforward conversion functions.

```
TYPE long IS INTEGER RANGE -100 TO 100;
TYPE short IS INTEGER RANGE -10 TO 10;
SIGNAL x : short;
SIGNAL y : long;
...
y <= 2*x + 5;          -- error, type mismatch
y <= long(2*x + 5);    -- OK, result converted into type long
```

VHDL> Data Conversion

- Data conversion functions: *std_logic_arith* package of the *ieee* library.

keyword	**Input data type**	**Output data type**
conv_integer(p)	INTEGER, UNSIGNED, SIGNED, or STD_ULOGIC	INTEGER
conv_unsigned(p, b)	INTEGER, UNSIGNED, SIGNED, or STD_ULOGIC	UNSIGNED * Where b is number of bits.
conv_signed(p, b):	INTEGER, UNSIGNED, SIGNED, or STD_ULOGIC	SIGNED
conv_std_logic_vector(p, b)	INTEGER, UNSIGNED, SIGNED, or STD_ULOGIC	STD_LOGIC_VECTOR

VHDL> Data Conversion

- Data conversion functions: std_logic_signed or std_logic_unsigned package of the *ieee* library.

keyword	**Input data type**	**Output data type**
unsigned(p)	STD_LOGIC_VECTOR	UNSIGNED
signed(p):	STD_LOGIC_VECTOR	SIGNED

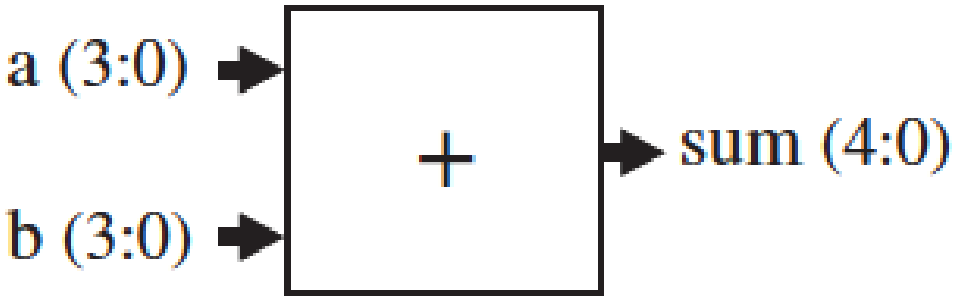
VHDL> Data Conversion

- Example:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
...
SIGNAL a: IN UNSIGNED (7 DOWNTO 0);
SIGNAL b: IN UNSIGNED (7 DOWNTO 0);
SIGNAL y: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
...
y <= CONV_STD_LOGIC_VECTOR ((a+b), 8);
-- Legal operation: a+b is converted from UNSIGNED to an
-- 8-bit STD_LOGIC_VECTOR value, then assigned to y.
```

VHDL> Examples:

- A 4-bit adder:



VHDL> Examples:

- A 4-bit adder:

```
----- Solution 1: in/out=SIGNED -----
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
-------------------------------------
ENTITY adder1 IS
PORT ( a, b : IN SIGNED (3 DOWNTO 0);
sum : OUT SIGNED (4 DOWNTO 0));
END adder1;
-------------------------------------
ARCHITECTURE adder1 OF adder1 IS
BEGIN
sum <= a + b;
END adder1;
-------------------------------------
```

```
----- Solution 2: out=INTEGER -----
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
-------------------------------------
ENTITY adder2 IS
PORT ( a, b : IN SIGNED (3 DOWNTO 0
sum : OUT INTEGER RANGE -16 TO 15);
END adder2;
-------------------------------------
ARCHITECTURE adder2 OF adder2 IS
BEGIN
sum <= CONV_INTEGER(a + b);
END adder2;
-------------------------------------
```

VHDL>Static and non-static data

- CONSTANT:
  - establish default values
  - can be declared in a PACKAGE, ENTITY, or ARCHITECTURE.

```
CONSTANT name : type := value;
```

```
CONSTANT set_bit : BIT := '1';
CONSTANT datamemory : memory := (('0','0','0','0'),
                                 ('0','0','0','1'),
                                 ('0','0','1','1'));
```

VHDL>Static and non-static data

- GENERIC:
  - specifying a generic parameter (that is, a static parameter).
  - code more flexibility and reusability.
  - must be declared in the ENTITY.

```
GENERIC (parameter_name : parameter_type := parameter_value);
```

```
ENTITY adder2 IS
GENERIC (n : INTEGER := 8);
PORT ( a, b : IN SIGNED (3 D
sum : OUT INTEGER RANGE -16
```

VHDL>Static and non-static data

- SIGNAL:
  - pass values in and out the circuit, as well as between its internal units.
  - circuit interconnects (wires).

```
SIGNAL name : type [range] [:= initial_value];
```

```
SIGNAL control: BIT := '0';
SIGNAL count: INTEGER RANGE 0 TO 100;
SIGNAL y: STD_LOGIC_VECTOR (7 DOWNTO 0);
```

VHDL>Static and non-static data

- VARIABLE:
  - represents only local information.
  - It can only be used inside a sequential code (PROCESS for example).

```
VARIABLE name : type [range] [:= init_value];
```

```
variable control: BIT := '0';
variable count: INTEGER RANGE 0 TO 100;
variable y: STD_LOGIC_VECTOR (7 DOWNTO 0);
```

VHDL>Static and non-static data

Table 7.1
Comparison between SIGNAL and VARIABLE.

	SIGNAL	VARIABLE
Assignment	<=	:=
Utility	Represents circuit interconnects (wires)	Represents local information
Scope	Can be global (seen by entire code)	Local (visible only inside the corresponding PROCESS, FUNCTION, or PROCEDURE)
Behavior	Update is not immediate in sequential code (new value generally only available at the conclusion of the PROCESS, FUNCTION, or PROCEDURE)	Updated immediately (new value can be used in the next line of code)
Usage	In a PACKAGE, ENTITY, or ARCHITECTURE. In an ENTITY, all PORTS are SIGNALS by default	Only in sequential code, that is, in a PROCESS, FUNCTION, or PROCEDURE

VHDL> Operators

- VHDL provides several kinds of pre-defined operators:
  - Assignment operators
  - Logical operators
  - Arithmetic operators
  - Relational operators
  - Shift operators
  - Concatenation operators

VHDL> Operators

- Assignment operators

Operator	using
<=	SIGNAL.
:=	VARIABLE, CONSTANT, GENERIC, initial values.
=>	vector elements or with OTHERS.

VHDL> Operators

- Assignment operators

```
SIGNAL x : STD_LOGIC;
VARIABLE y : STD_LOGIC_VECTOR(3 DOWNTO 0); -- Leftmost bit is MSB
SIGNAL w: STD_LOGIC_VECTOR(0 TO 7); -- Rightmost bit is -- MSB
-------------------------------------------------------------
x <= '1'; -- '1' is assigned to SIGNAL x using "<="
y := "0000"; -- "0000" is assigned to VARIABLE y using ":="
w <= "10000000"; -- LSB is '1', the others are '0'
w <= (0 =>'1', OTHERS =>'0'); -- LSB is '1', the others are '0'
```

VHDL> Operators

- **Logical operators:**
- perform logical operations.
- Data types: BIT, STD_LOGIC, STD_ULOGIC, BIT_VECTOR, STD_LOGIC_VECTOR, or STD_ULOGIC_VECTOR

- NOT
  - AND
  - OR
  - NAND
  - NOR
  - XOR
  - XNOR

Examples:

```
y <= NOT a AND b;        -- (a'.b)
y <= NOT (a AND b);      -- (a.b)'
y <= a NAND b;           -- (a.b)'
```

VHDL> Operators

- **Arithmetic Operators:**
- perform arithmetic operations
- data types: INTEGER, SIGNED, UNSIGNED, or REAL
- With *std_logic_signed* or *std_logic_unsigned* package: STD_LOGIC_VECTOR.

+	Addition	
–	Subtraction	
*	Multiplication	
/	Division	only power of two dividers
**	Exponentiation	only static values of base and exponent are accepted

VHDL> Operators

- N bit adder circuit:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

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

entity ADDER is

generic(n: natural :=2);
port( A: in std_logic_vector(n-1 downto 0);
   B: in std_logic_vector(n-1 downto 0);
   carry:   out std_logic;
   sum:  out std_logic_vector(n-1 downto 0)
);

end ADDER;
```

VHDL> Operators

- N bit adder circuit:

```
architecture behv of ADDER is

-- define a temparary signal to store the result

signal result: std_logic_vector(n downto 0);

begin

    -- the 3rd bit should be carry

    result <= ('0' & A)+('0' & B);
    sum <= result(n-1 downto 0);
    carry <= result(n);

end behv;
```

VHDL> Operators

- **Comparison Operators:**
- Used for making comparisons.
- Data types: any.

=	Equal to
/=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

VHDL> Operators

N bit comparator:

```
library ieee;
use ieee.std_logic_1164.all;

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

entity Comparator is

generic(n: natural :=2);
port( A: in std_logic_vector(n-1 downto 0);
   B: in std_logic_vector(n-1 downto 0);
   less:    out std_logic;
   equal:      out std_logic;
   greater: out std_logic
);
end Comparator;

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

VHDL> Operators

```
architecture behv of Comparator is

begin

    process(A,B)
    begin
        if (A<B) then
       less <= '1';
       equal <= '0';
       greater <= '0';
   elsif (A=B) then
       less <= '0';
       equal <= '1';
       greater <= '0';
   else
       less <= '0';
       equal <= '0';
       greater <= '1';
   end if;
    end process;

end behv;
```

VHDL> Operators

- **Shift Operators:**
  - sll Shift left logic – positions on the right are filled with '0's
  - srl Shift right logic – positions on the left are filled with '0's
- Syntax:

⟨left operand⟩ ⟨shift operation⟩ ⟨right operand⟩

BIT_VECTOR

INTEGER

sll, srl, sla, sra, rol, ror

VHDL> Data Attributes

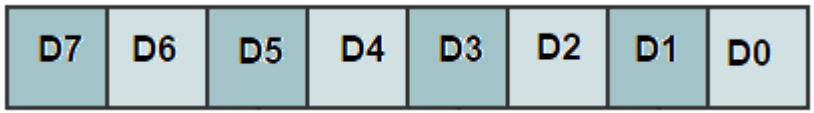
- The pre-defined, synthesizable data attributes are the following:
- d’LOW: Returns lower array index
- d’HIGH: Returns upper array index
- d’LEFT: Returns leftmost array index
- d’RIGHT: Returns rightmost array index
- d’LENGTH: Returns vector size
- d’RANGE: Returns vector range
- d’REVERSE_RANGE: Returns vector range in reverse order

VHDL> Data Attributes

Example: Consider the following signal:

```
SIGNAL d : STD_LOGIC_VECTOR (7 DOWNTO 0);
```

Then:

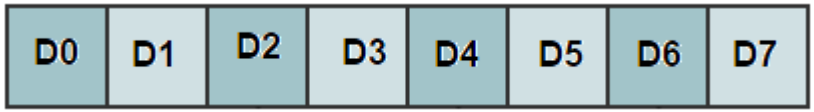



```
d'LOW=0, d'HIGH=7, d'LEFT=7, d'RIGHT=0, d'LENGTH=8,
d'RANGE=(7 downto 0), d'REVERSE_RANGE=(0 to 7).
```

Example: Consider the following signal:

```
SIGNAL x: STD_LOGIC_VECTOR (0 TO 7);
```

Then all four LOOP statements below are synthesizable and equivalent.




```
FOR i IN RANGE (0 TO 7) LOOP ...
FOR i IN x'RANGE LOOP ...
FOR i IN RANGE (x'LOW TO x'HIGH) LOOP ...
FOR i IN RANGE (0 TO x'LENGTH-1) LOOP ...
```

VHDL> Signal Attributes

- Let us consider a signal s. Then:
  - s'EVENT: Returns true when an event occurs on s.
  - s'STABLE: Returns true if no event has occurred on s.

```
IF (clk'EVENT AND clk='1')...          -- EVENT attribute used
                                       -- with IF
IF (NOT clk'STABLE AND clk='1')...     -- STABLE attribute used
                                       -- with IF
WAIT UNTIL (clk'EVENT AND clk='1');    -- EVENT attribute used
                                       -- with WAIT
IF RISING_EDGE(clk)...                 -- call to a function
```

VHDL> User-Defined Attributes

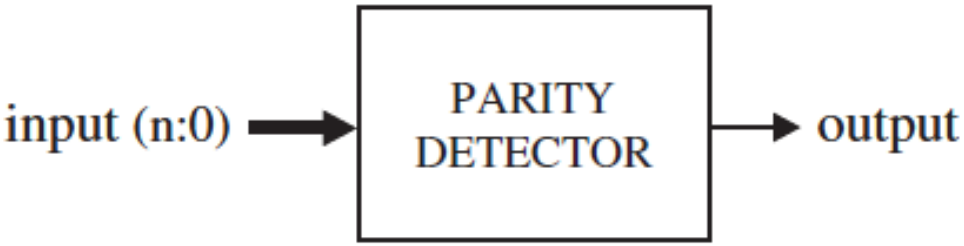
- VHDL also allows the construction of user defined attributes.

```
ATTRIBUTE attribute_name: attribute_type;

ATTRIBUTE attribute_name OF target_name: class IS value;
```

```
 -- declaration
ATTRIBUTE number_of_inputs: INTEGER;
 -- specification
ATTRIBUTE number_of_inputs OF nand3: SIGNAL IS 3;
...
 -- attribute call, returns 3
inputs <= nand3'number_of_pins;
```

VHDL> Examples

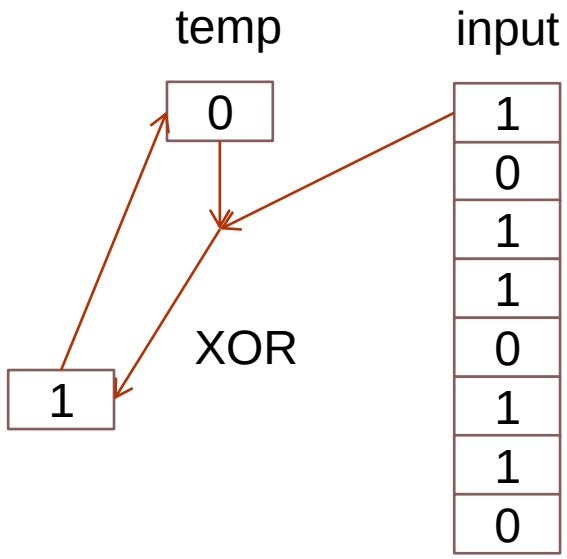
- Generic Parity Detector:
- The circuit must provide output = ‘0’ when the number of ‘1’s in the input vector is odd, or output = ‘1’ otherwise.



VHDL> Examples

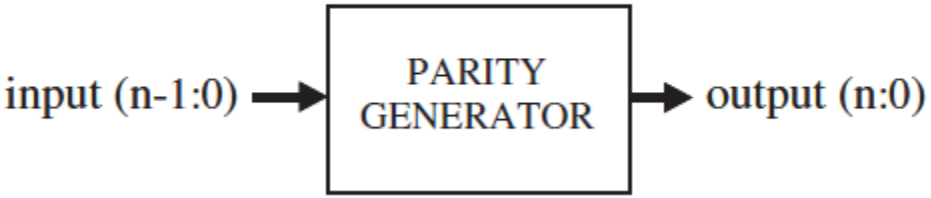
- Generic Parity Detector:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------
ENTITY parity_det IS
GENERIC (n : INTEGER := 7);
PORT ( input: IN Std_logic_VECTOR (n DOWNTO 0);
output: out Std_logic);
END parity_det;
-------------------------------------------
ARCHITECTURE parity OF parity_det IS
BEGIN
PROCESS (input)
VARIABLE temp: Std_logic;
BEGIN
temp := '0';
FOR i IN input'RANGE LOOP
temp := temp XOR input(i);
END LOOP;
output <= not temp;
END PROCESS;
END parity;
-------------------------
```



VHDL> Examples

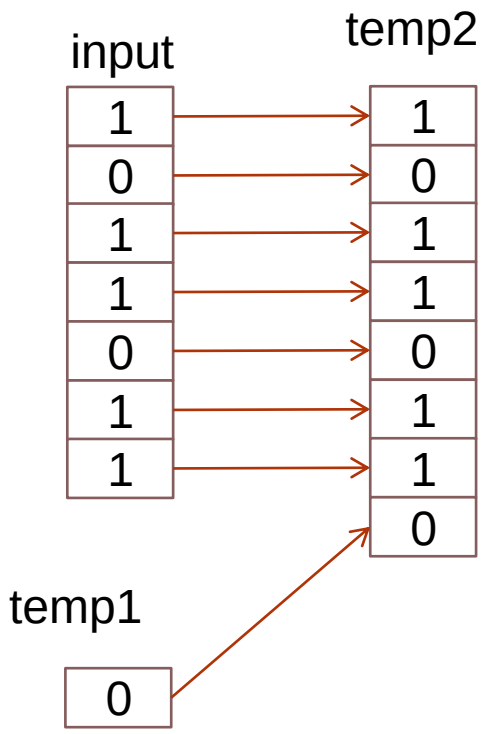
- Generic Parity **Generator**:
  - The circuit must add one bit to the input vector (on its left).
  - Such bit must be a ‘0’ if the number of ‘1’s in the input vector is even, or a ‘1’ if it is odd, such that the resulting vector will always contain an even number of ‘1’s (even parity).



VHDL> Examples

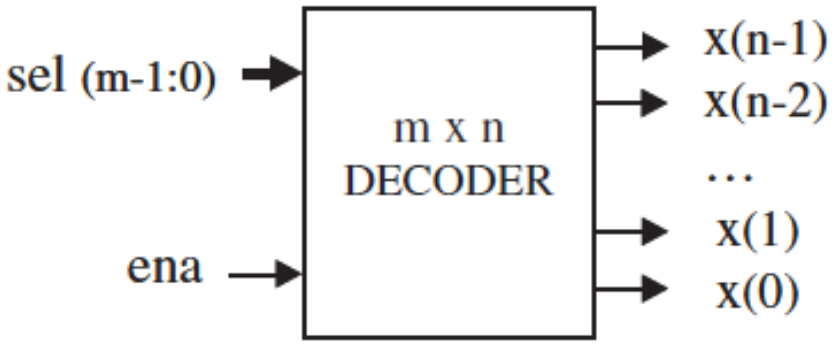
- Generic Parity **Generator**:

```
----------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
----------------------------------------
ENTITY parity_gen IS
GENERIC (n : INTEGER := 7);
PORT ( input: IN STD_LOGIC_VECTOR (n-1 DOWNTO 0);
output: OUT STD_LOGIC_VECTOR (n DOWNTO 0));
END parity_gen;
----------------------------------------------
ARCHITECTURE parity OF parity_gen IS
BEGIN
PROCESS (input)
VARIABLE temp1: STD_LOGIC;
VARIABLE temp2: STD_LOGIC_VECTOR (output'RANGE);
BEGIN
temp1 := '0';
FOR i IN input'_RANGE LOOP
temp1 := temp1 XOR input(i);
temp2(i) := input(i);
END LOOP;
temp2(output'HIGH) := temp1;
output <= temp2;
END PROCESS;
END parity;
```



VHDL> Examples

- Generic **Decoder:**
  - If ena = ‘0’, then all bits of x should be high; otherwise, the output bit selected by sel should be low.



$m = \log_2 n$

ena	sel	x
0	00	1111
1	00	1110
	01	1101
	10	1011
	11	0111

VHDL> Examples

- Generic **Decoder:**

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
--------------------------------------------
ENTITY decoder IS
PORT ( ena : IN STD_LOGIC;
sel : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
x : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END decoder;
ARCHITECTURE generic_decoder OF decoder IS
BEGIN
PROCESS (ena, sel)
VARIABLE temp1 : STD_LOGIC_VECTOR (x'HIGH DOWNTO 0);
VARIABLE temp2 : INTEGER RANGE 0 TO x'HIGH;
BEGIN
temp1 := (OTHERS => '1');
IF (ena='1') THEN
temp2:=conv_integer(signed(sel));
temp1(temp2):='0';
END IF;
x <= temp1;
END PROCESS;
END generic_decoder;
```

جامعة نينوى
كلية هندسة الإلكترونيات

HDL Programming -VHDL- 2

Textbook: Volnei A. Pedroni, “Circuit Design with VHDL”, MIT Press London, England, 2004.

Submitted By: Hussein M. H. Aideen

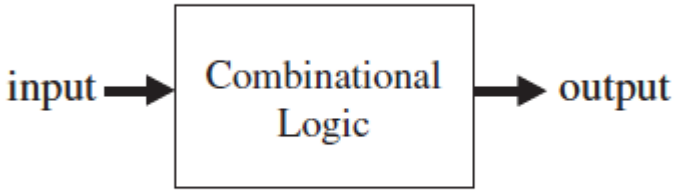
VHDL> Concurrent & sequential Code

- Concurrent Code:
  - WHEN,
  - GENERATE,
  - Assignments using only operators (AND, NOT, +, *, sll, etc.),
  - A special kind of assignment, called BLOCK.

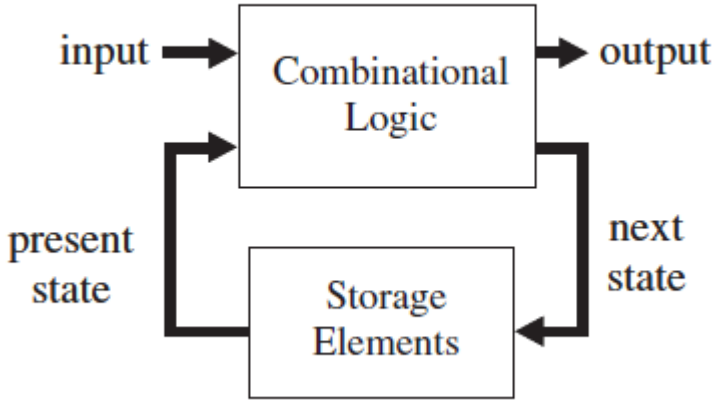
- Sequential Code:
  - PROCESSES, FUNCTIONS, PROCEDURES.
  - IF, WAIT, CASE, and LOOP.
  - VARIABLES.

VHDL>Combinational vs Sequential Logic

- **Combinational Logic:** output depends solely on the current inputs.



- **sequential logic:** output depend on previous inputs.



VHDL> Concurrent versus Sequential

- **VHDL** code is inherently concurrent (parallel).
- Only statements placed inside a PROCESS, FUNCTION, or PROCEDURE are sequential.
  - the block, as a whole, is concurrent with any other (external) statements.
- Concurrent code is also called **dataflow** code.
- Concurrent: The order does not matter.

VHDL> Concurrent Code

- 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.

VHDL> Concurrent Code

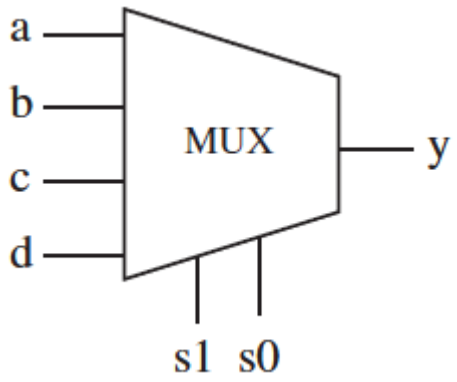
- Operators: Operators are type of Concurrent code.

Table 5.1
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

VHDL> Concurrent Code Examples

- Multiplexer #1




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

VHDL> Concurrent Code

- WHEN (Simple and Selected)

WHEN / ELSE:

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

simple

WITH / SELECT / WHEN:

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

selected

VHDL> Concurrent Code

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

- "WHEN value" 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 ...
```

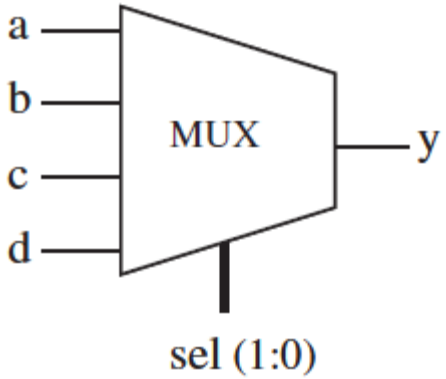
VHDL> Concurrent Code

- Examples:

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

VHDL> Concurrent Code

- Multiplexer #2: when/else

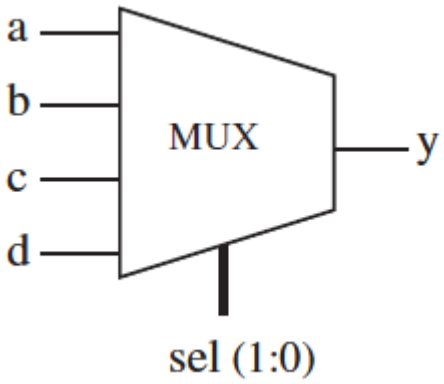



```
1  ------- Solution 1: with WHEN/ELSE --------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------
5  ENTITY mux IS
6     PORT ( a, b, c, d: IN STD_LOGIC;
7            sel: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
8            y: OUT STD_LOGIC);
9  END mux;
10 ----------------------------------------
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;
```

Simple when

VHDL> Concurrent Code

- Multiplexer #2: with/select/when

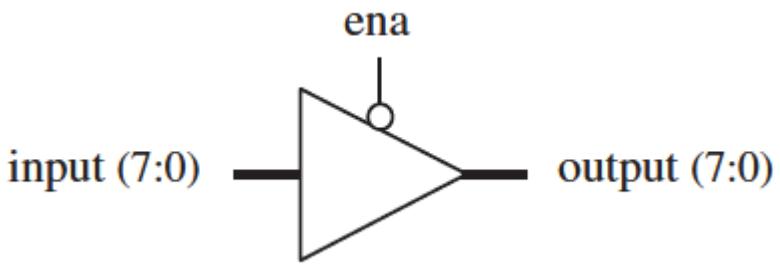



```
1  --- Solution 2: with WITH/SELECT/WHEN -----
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------
5  ENTITY mux IS
6     PORT ( a, b, c, d: IN STD_LOGIC;
7            sel: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
8            y: OUT STD_LOGIC);
9  END mux;
10 ----------------------------------------
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;
```

selected when

VHDL> Concurrent Code

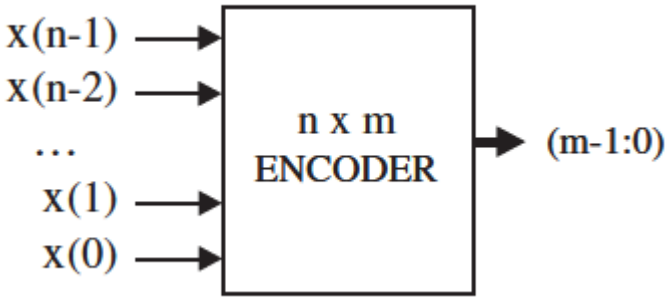
- Tri-state Buffer:




```
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 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;
```

VHDL> Concurrent Code

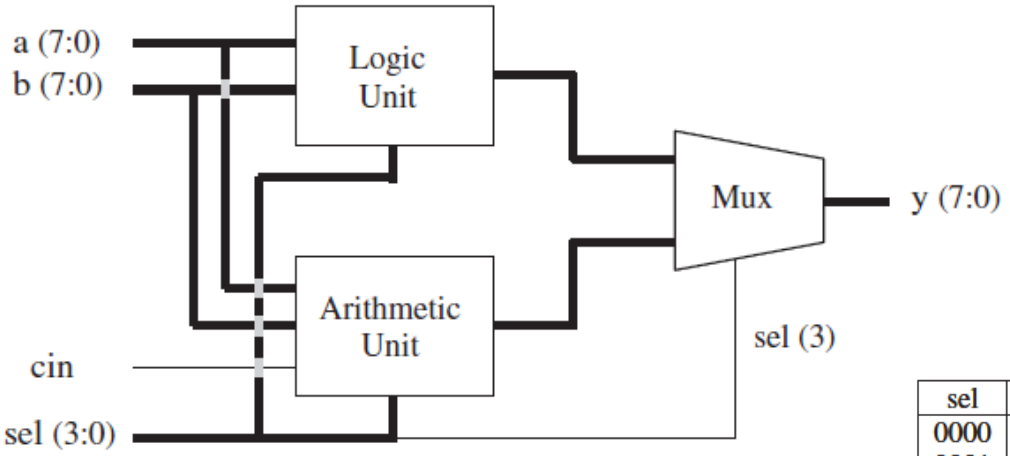
- Home Works: Encoder: page 73:




```
ARCHITECTURE encoder1 OF encoder IS
BEGIN
   y <=    "000" WHEN x="00000001" ELSE
           "001" WHEN x="00000010" ELSE
           "010" WHEN x="00000100" ELSE
           "011" WHEN x="00001000" ELSE
           "100" WHEN x="00010000" ELSE
           "101" WHEN x="00100000" ELSE
           "110" WHEN x="01000000" ELSE
           "111" WHEN x="10000000" ELSE
           "ZZZ";
```

VHDL> Concurrent Code

- Home Works: ALU: page 75



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	

VHDL> Concurrent Code

- The **GENERATE** statement:
  - allows a section of code to be repeated a number of times (loop).
  - GENERATE must be labeled.
  - limits of the range must be **static**.

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

VHDL> Concurrent Code

- IF/GENERATE: (ELSE is not allowed).
  - IF/GENERATE can be nested inside FOR/GENERATE, the opposite can also be done.

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

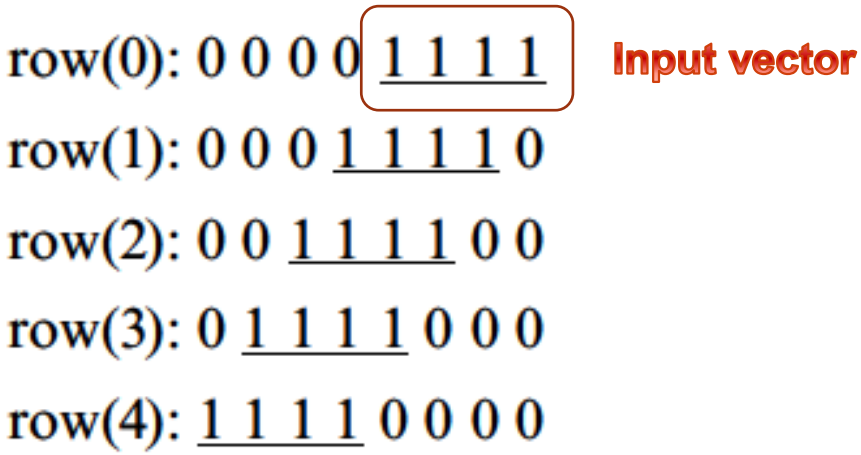
VHDL> Concurrent Code

- 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;
```

VHDL> Concurrent Code

- Vector Shifter:
  - 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.



VHDL> Concurrent Code

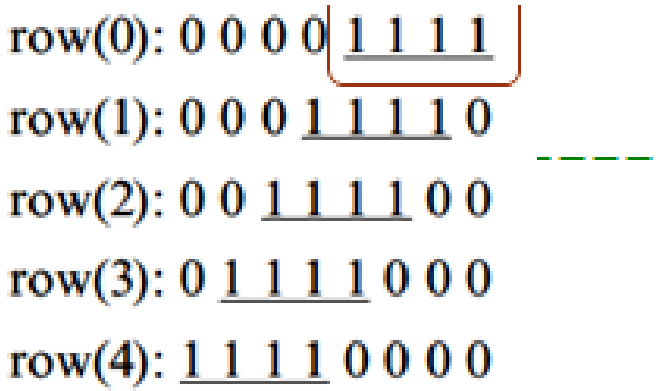
- Vector Shifter:

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

VHDL> Concurrent Code

- Vector Shifter:

```
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;
22  ------------------------------
```



VHDL> Concurrent Code

- BLOCK:
- Simple BLOCK
  - locally partitioning the code.
  - turning the overall code more readable (long codes).
- can be nested inside another BLOCK.

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

VHDL> Concurrent Code

- Simple BLOCK:

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

VHDL> Concurrent Code

- Nested BLOCK:

nested

```
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;
```

VHDL> Concurrent Code

- Guarded BLOCK:
  - includes an additional expression, called guard expression.
- A guarded statement executed only when the guard expression is TRUE.
- sequential circuits can be constructed.

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

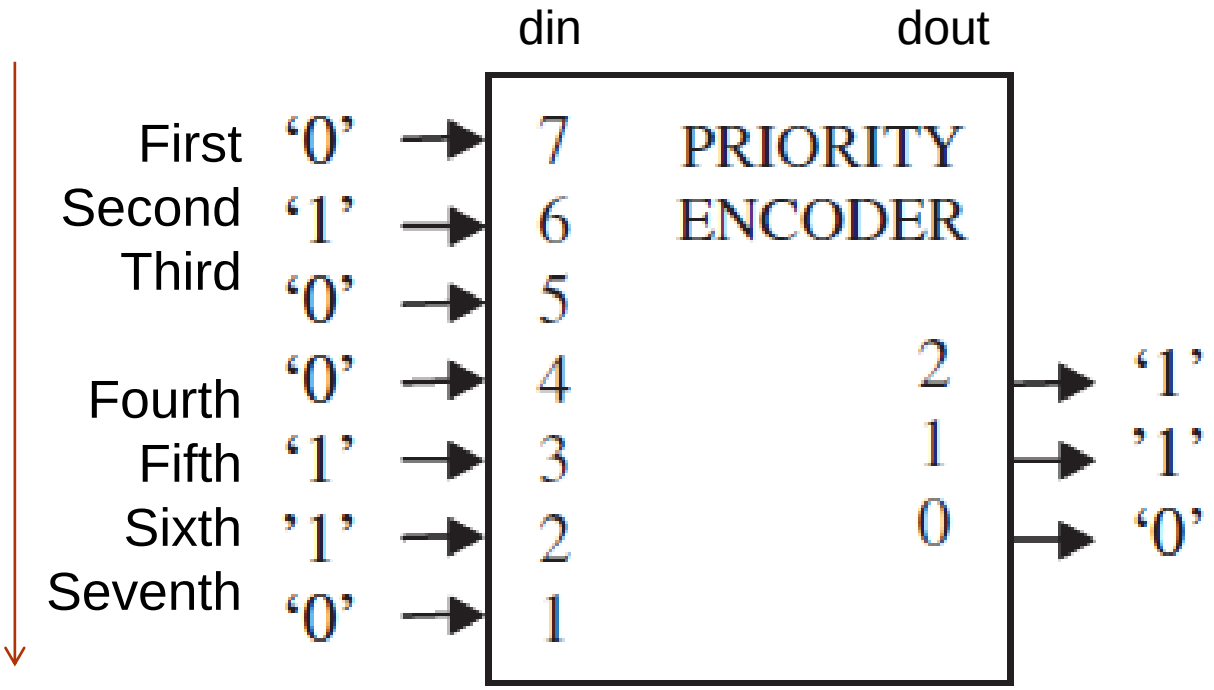
VHDL> Concurrent Code Example

- DFF with Guarded BLOCK:

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

VHDL> Concurrent Code Example

- Problem 5.2: Priority Encoder



Binary
110

```
dout <= "000" when din(7)='1' else
        "001" when din(6)='1' else
        "010" when din(5)='1' else
```

VHDL> Concurrent Code

- Problem 5.2: Priority Encoder

```
 1  library IEEE;
 2  use IEEE.STD_LOGIC_1164.all;
 3  use ieee.numeric_std.all;
 4  ----------------------------------------------------
 5  entity pri_encoder_8_3 is
 6       port(
 7            din : in STD_LOGIC_VECTOR(7 downto 0);
 8            dout : out STD_LOGIC_VECTOR(2 downto 0)
 9            );
10  end pri_encoder_8_3;
11  ----------------------------------------------------
```

VHDL> Concurrent Code

- Problem 5.2: Priority Encoder

```
12  architecture pri_enc_arc of pri_encoder_8_3 is
13  begin
14    dout <= "000" when din(7)='1' else
15            "001" when din(6)='1' else
16            "010" when din(5)='1' else
17            "011" when din(4)='1' else
18            "100" when din(3)='1' else
19            "101" when din(2)='1' else
20            "110" when din(1)='1' else
21            "111" when din(0)='1';
22  end pri_enc_arc;
```

VHDL> Sequential Code

VHDL> Sequential Code

- PROCESSES, FUNCTIONS, and PROCEDURES are executed sequentially.
- any of these blocks is still concurrent with any other statements placed outside it.
- with it we can build sequential circuits as well as combinational circuits.

VHDL> Sequential Code

- IF, WAIT, CASE, and LOOP.
- VARIABLES restricted to be used in sequential code.

VHDL> PROCESS

- A PROCESS must be installed in the main code.
- executed every time a signal in the sensitivity list changes.

```
[label:] PROCESS (sensitivity list)
  [VARIABLE name type [range] [:= initial_value;]]
BEGIN
  (sequential code)
END PROCESS [label];
```

VHDL> PROCESS

- initial value is not synthesizable.

- monitoring a signal (clock, for example) is necessary. A common way of detecting a signal change is by means of the EVENT attribute.
- For instance, if clk is a signal to be monitored, then clk'EVENT returns TRUE when a change on clk occurs (rising or falling edge).

VHDL> PROCESS

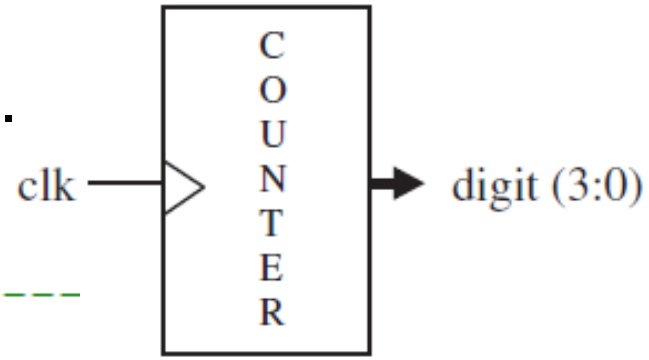
- IF statement:

```
IF conditions THEN assignments;
ELSIF conditions THEN assignments;
...
ELSE assignments;
END IF;
```

```
IF (x<y) THEN temp:="11111111";
ELSIF (x=y AND w='0') THEN temp:="11110000";
ELSE temp:=(OTHERS =>'0');
end if
```

VHDL> IF statement

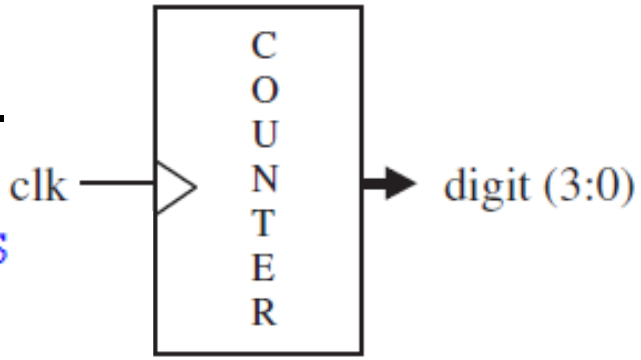
- One-digit Counter #1
  - 1-digit decimal counter (0 → 9 → 0).




```
1  ----------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------
5  ENTITY counter IS
6  PORT (clk : IN STD_LOGIC;
7  digit : OUT INTEGER RANGE 0 TO 9);
8  END counter;
9  ----------------------------------------
```

VHDL> IF statement

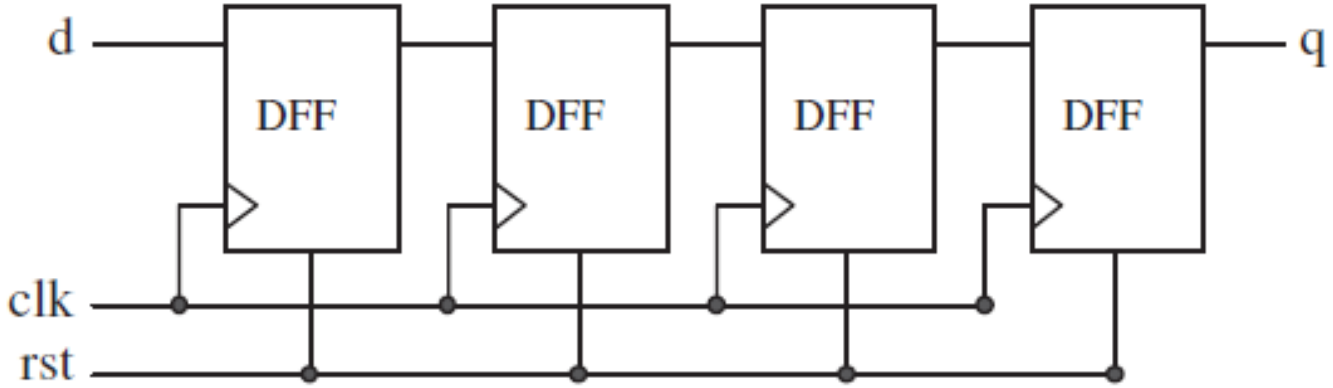
- One-digit Counter #1
  - 1-digit decimal counter (0 → 9 → 0).




```
10  ARCHITECTURE counter OF counter IS
11  BEGIN
12  count: PROCESS(clk)
13  VARIABLE temp : INTEGER RANGE 0 TO 10;
14  BEGIN
15  IF (clk'EVENT AND clk='1') THEN
16  temp := temp + 1;
17  IF (temp=10) THEN temp := 0;
18  END IF;
19  END IF;
20  digit <= temp;
21  END PROCESS count;
22  END counter;
23  ------------------
```

VHDL> IF statement

- 4 bit Shift Register:




```
1  ----------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------
5  ENTITY shiftreg IS
6  GENERIC (n: INTEGER := 4); -- # of stages
7  PORT (d, clk, rst: IN STD_LOGIC;
8  q: OUT STD_LOGIC);
9  END shiftreg;
```

VHDL> IF statement

- 4 bit Shift Register:

```
11  ARCHITECTURE behavior OF shiftreg IS
12  SIGNAL internal: STD_LOGIC_VECTOR (n-1 DOWNTO 0);
13  BEGIN
14  PROCESS (clk, rst)
15  BEGIN
16  IF (rst='1') THEN
17  internal <= (OTHERS => '0');
18  ELSIF (clk'EVENT AND clk='1') THEN
19  internal <= d & internal(internal'LEFT DOWNTO 1);
20  END IF;
21  END PROCESS;
22  q <= internal(0);
23  END behavior;
24  ------------------------------------------------
```

VHDL> WAIT statement

- WAIT statement:
- the PROCESS cannot have a sensitivity list when WAIT is employed.

```
WAIT UNTIL signal_condition;
```

```
WAIT ON signal1 [, signal2, ... ];
```

```
WAIT FOR time;
```

Simulation only

VHDL> WAIT statement

- WAIT statement:
- the PROCESS cannot have a sensitivity list when WAIT is employed.

```
Example: 8-bit_register _with synchronous reset.
PROCESS -- no sensitivity list
BEGIN
 WAIT UNTIL (clk'EVENT AND clk='1');
 IF (rst='1') THEN
   output <= "00000000";
 ELSIF (clk'EVENT AND clk='1') THEN
   output <= input;
 END IF;
END PROCESS;
```

VHDL> WAIT statement

- WAIT ON:

```
Example: 8-bit_register _with asynchronous reset.
PROCESS
BEGIN
 WAIT ON clk, rst;
 IF (rst='1') THEN
   output <= "00000000";
 ELSIF (clk'EVENT AND clk='1') THEN
   output <= input;
 END IF;
END PROCESS;
```

- WAIT FOR is intended for simulation only (waveform generation for testbenches). Example: WAIT FOR 5ns;

VHDL> WAIT statement

- Home Works:
- DFF with Asynchronous Reset #2, P99.
- One-digit Counter #2, P99-100.

VHDL> CASE statement

- CASE statement:

```
CASE identifier IS
    WHEN value => assignments;
    WHEN value => assignments;
    ...
END CASE;
```

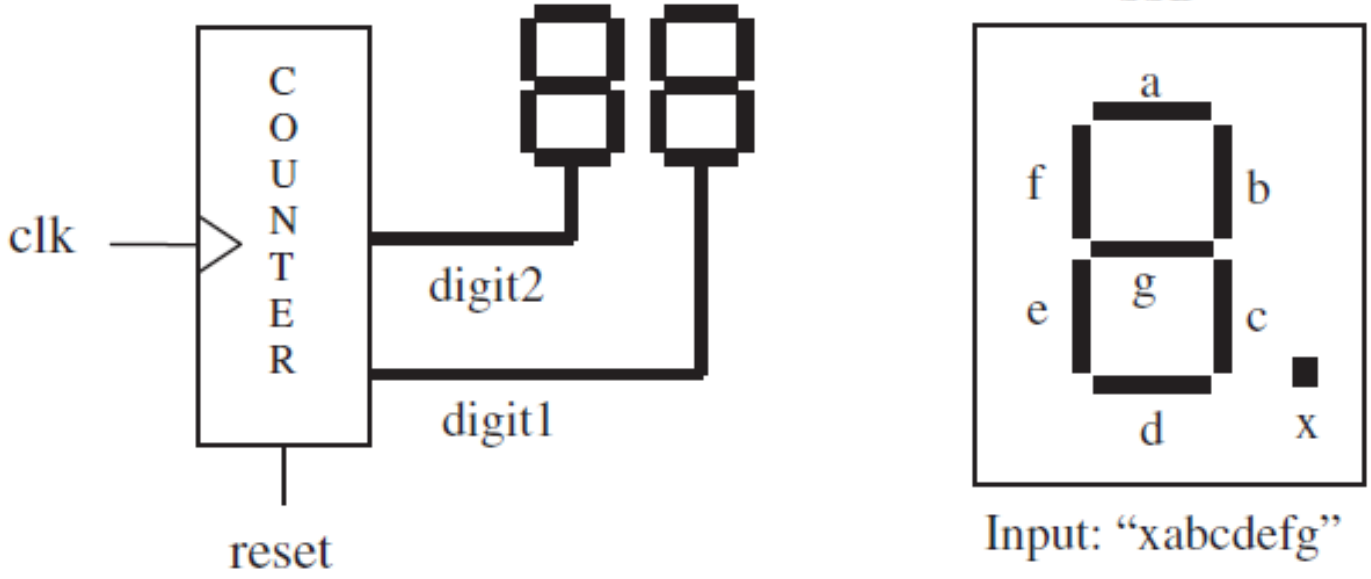
```
CASE control IS
   WHEN "00" => x<=a; y<=b;
   WHEN "01" => x<=b; y<=c;
   WHEN OTHERS => x<="0000"; y<="ZZZZ";
END CASE;
```

VHDL> CASE statement

- CASE statement:
- CASE statement (sequential) is very similar to WHEN (combinational)
- keyword **OTHERS** is often helpful.
- Another important keyword is **NULL** (the counterpart of UNAFFECTED), which should be used when no action is to take place.
- CASE allows multiple assignments for each test condition.

VHDL> CASE statement

- Two-digit Counter with SSD Output:

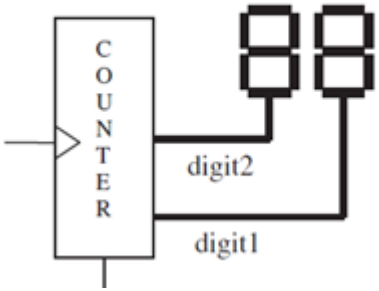


VHDL> Two-digit Counter with SSD

```
1  ----------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------------------
5  ENTITY counter2digit IS
6  PORT (clk, reset : IN STD_LOGIC;
7  digit1, digit2 : OUT STD_LOGIC_VECTOR (6 DOWNTO 0));
8  END counter2digit;
9  ----------------------------------------------------
10 ARCHITECTURE counter OF counter2digit IS
11 BEGIN
12 PROCESS(clk, reset)
13 VARIABLE temp1: INTEGER RANGE 0 TO 10;
14 VARIABLE temp2: INTEGER RANGE 0 TO 10;
15 BEGIN
16 ---- counter: ----------------------
17 IF (reset='1') THEN
18 temp1 := 0;
19 temp2 := 0;
20 ELSIF (clk'EVENT AND clk='1') THEN
```

VHDL> Two-digit Counter with SSD

```
20   ELSIF (clk'EVENT AND clk='1') THEN
21   temp1 := temp1 + 1;
22   IF (temp1=10) THEN
23   temp1 := 0;
24   temp2 := temp2 + 1;
25   IF (temp2=10) THEN
26   temp2 := 0;
27   END IF;
28   END IF;
29   END IF;
30   ---- BCD to SSD conversion: --------
```



VHDL> Two-digit Counter with SSD

```
30   ---- BCD to SSD conversion: --------
31   CASE temp1 IS
32   WHEN 0 => digit1 <= "1111110"; --7E
33   WHEN 1 => digit1 <= "0110000"; --30
34   WHEN 2 => digit1 <= "1101101"; --6D
35   WHEN 3 => digit1 <= "1111001"; --79
36   WHEN 4 => digit1 <= "0110011"; --33
37   WHEN 5 => digit1 <= "1011011"; --5B
38   WHEN 6 => digit1 <= "1011111"; --5F
39   WHEN 7 => digit1 <= "1110000"; --70
40   WHEN 8 => digit1 <= "1111111"; --7F
41   WHEN 9 => digit1 <= "1111011"; --7B
42   WHEN OTHERS => NULL;
43   END CASE;
```

VHDL> Two-digit Counter with SSD

```
44   CASE temp2 IS
45   WHEN 0 => digit2 <= "1111110"; --7E
46   WHEN 1 => digit2 <= "0110000"; --30
47   WHEN 2 => digit2 <= "1101101"; --6D
48   WHEN 3 => digit2 <= "1111001"; --79
49   WHEN 4 => digit2 <= "0110011"; --33
50   WHEN 5 => digit2 <= "1011011"; --5B
51   WHEN 6 => digit2 <= "1011111"; --5F
52   WHEN 7 => digit2 <= "1110000"; --70
53   WHEN 8 => digit2 <= "1111111"; --7F
54   WHEN 9 => digit2 <= "1111011"; --7B
55   WHEN OTHERS => NULL;
56   END CASE;
57   END PROCESS;
58   END counter;
59   ----------------------------------
```

جامعة نينوى
كلية هندسة الإلكترونيات

HDL Programming -VHDL- 3

Textbook: Volnei A. Pedroni, “Circuit Design with VHDL”, MIT Press London, England, 2004.

Submitted By: Hussein M. H. Aideen

VHDL> Loop statement

- LOOP is useful when a piece of code must be instantiated several times.
- inside a PROCESS, FUNCTION, or PROCEDURE.

- FOR / LOOP: repeated a fixed number of times.

```
[label:] FOR identifier IN range LOOP
  (sequential statements)
END LOOP [label];
```

VHDL> Loop statement

- WHILE / LOOP: repeated until a condition no longer holds.

```
[label:] WHILE condition LOOP
  (sequential statements)
END LOOP [label];
```

VHDL> Loop statement

- EXIT: Used for ending the loop.

```
[label:] EXIT [label] [WHEN condition];
```

- NEXT: Used for skipping loop steps.

```
[label:] NEXT [loop_label] [WHEN condition];
```

VHDL> Loop statement

- Example of **FOR** / LOOP:

```
FOR i IN 0 TO 5 LOOP
   x(i) <= enable AND w(i+2);
   y(0, i) <= w(i);
END LOOP;
```

- Example of **WHILE** / LOOP:

```
WHILE (i < 10) LOOP
  WAIT UNTIL clk'EVENT AND clk='1';
  (other statements)
END LOOP;
```

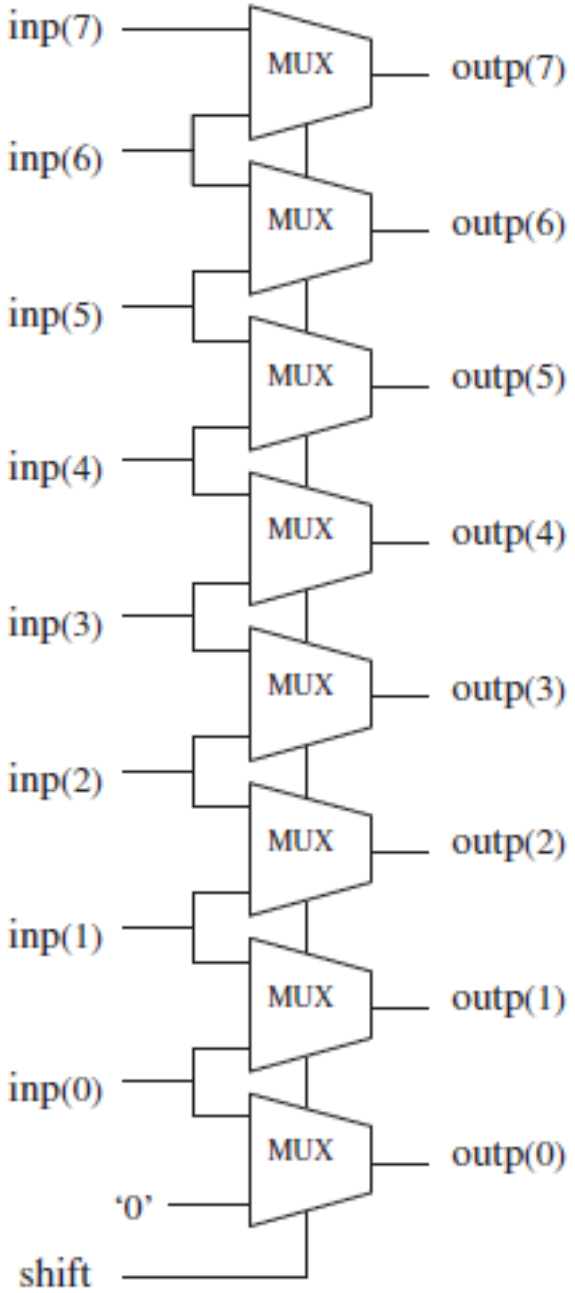
VHDL> Loop statement

- Simple Barrel Shifter:
- The circuit must shift the input vector (of size 8) either 0 or 1 position to the left. When actually shifted (shift = 1), the LSB bit must be filled with ‘0’ (shown in the botton left corner of the diagram).

- If shift = 0, then outp = inp;
- if shift = 1, then outp(0) = ‘0’ and outp(i) = inp(i - 1), for $1 \leq i \leq 7$.

VHDL> Loop statement

- If shift = 0, then outp = inp;
- if shift = 1, then outp(0) = ‘0’ and outp(i) = inp(i - 1), for 1≤ i ≤7.

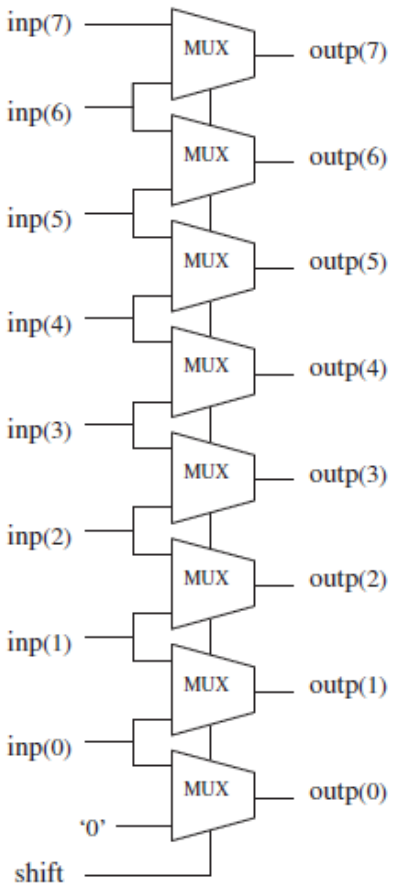


VHDL> Simple Barrel Shifter

```
 1  ----------------------------------------------
 2  LIBRARY ieee;
 3  USE ieee.std_logic_1164.all;
 4  ----------------------------------------------
 5  ENTITY barrel IS
 6  GENERIC (n: INTEGER := 8);
 7  PORT ( inp: IN STD_LOGIC_VECTOR (n-1 DOWNTO 0);
 8   shift: IN INTEGER RANGE 0 TO 1;
 9   outp: OUT STD_LOGIC_VECTOR (n-1 DOWNTO 0));
10  END barrel;
11  ----------------------------------------------
```

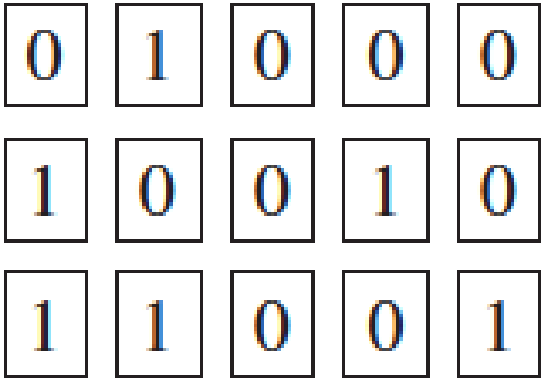
VHDL> Simple Barrel Shifter

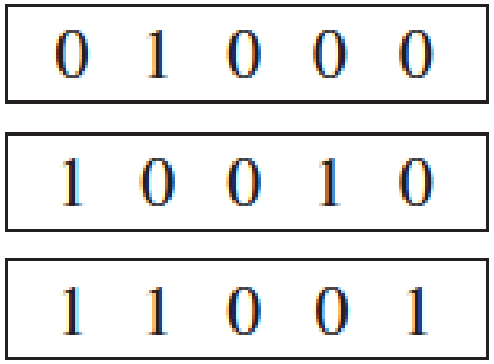
```
11  ----------------------------------------
12  ARCHITECTURE RTL OF barrel IS
13  BEGIN
14     PROCESS (inp, shift)
15       BEGIN
16       IF (shift=0) THEN
17         outp <= inp;
18       ELSE
19         outp(0) <= '0';
20         FOR i IN 1 TO inp'HIGH LOOP
21             outp(i) <= inp(i-1);
22         END LOOP;
23       END IF;
24     END PROCESS;
25  END RTL;
26   ----------------------------------------
```



VHDL> Arrays in VHDL

- Arrays are collections of objects of the same type.
- one-dimensional (1D),
- two-dimensional (2D),
- one-dimensional-by-one-dimensional (1Dx1D).





VHDL> Arrays in VHDL

- First the new TYPE must be defined,
- Then the new SIGNAL, VARIABLE, or CONSTANT can be declared using that data type.

```
TYPE type_name IS ARRAY (specification) OF data_type;

SIGNAL signal_name: type_name [:= initial_value];
```

VHDL> Arrays in VHDL

- 1D-1D Array

```
TYPE row IS ARRAY (7 DOWNTO 0) OF STD_LOGIC;     -- 1D array
TYPE matrix IS ARRAY (0 TO 3) OF row;            -- 1Dx1D array
SIGNAL x: matrix;                                -- 1Dx1D signal

TYPE matrix IS ARRAY (0 TO 3) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
```

VHDL> Arrays in VHDL

- 2D Array

```
TYPE matrix2D IS ARRAY (0 TO 3, 7 DOWNTO 0) OF STD_LOGIC;
-- 2D array
```

Initial value

```
... :="0001";                                          -- for 1D array
... :=('0','0','0','1')                                -- for 1D array
... :=(('0','1','1','1'), ('1','1','1','0'));         -- for 1Dx1D or
                                                       -- 2D array
```

VHDL> Arrays in VHDL

- Arrays assignments

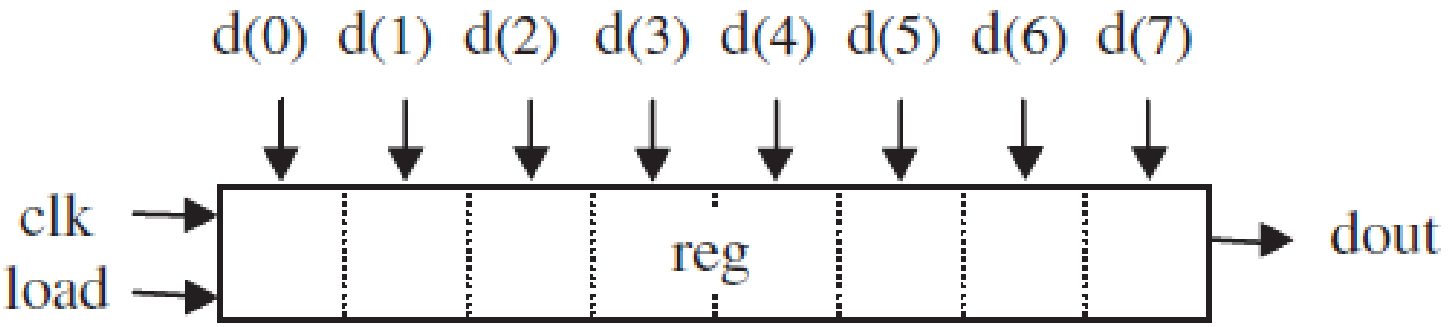
```
x(0) <= y(1)(2);        -- notice two pairs of parenthesis
                        -- (y is 1Dx1D)
x(1) <= v(2)(3);        -- two pairs of parenthesis (v is 1Dx1D)
x(2) <= w(2,1);         -- a single pair of parenthesis (w is 2D)
```

VHDL> Examples

- Signed and Unsigned Comparators

```
1  ---- Signed Comparator: ---------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE ieee.std_logic_arith.all;  -- necessary!
5  ----------------------------------------
6  ENTITY comparator IS
7     GENERIC (n: INTEGER := 7);
8     PORT (a, b: IN SIGNED (n DOWNTO 0);
9           x1, x2, x3: OUT STD_LOGIC);
10 END comparator;
11 ----------------------------------------
12 ARCHITECTURE signed OF comparator IS
13 BEGIN
14    x1 <= '1' WHEN a > b ELSE '0';
15    x2 <= '1' WHEN a = b ELSE '0';
16    x3 <= '1' WHEN a < b ELSE '0';
17 END signed;
18 ----------------------------------------
```

- Parallel-to-Serial Converter



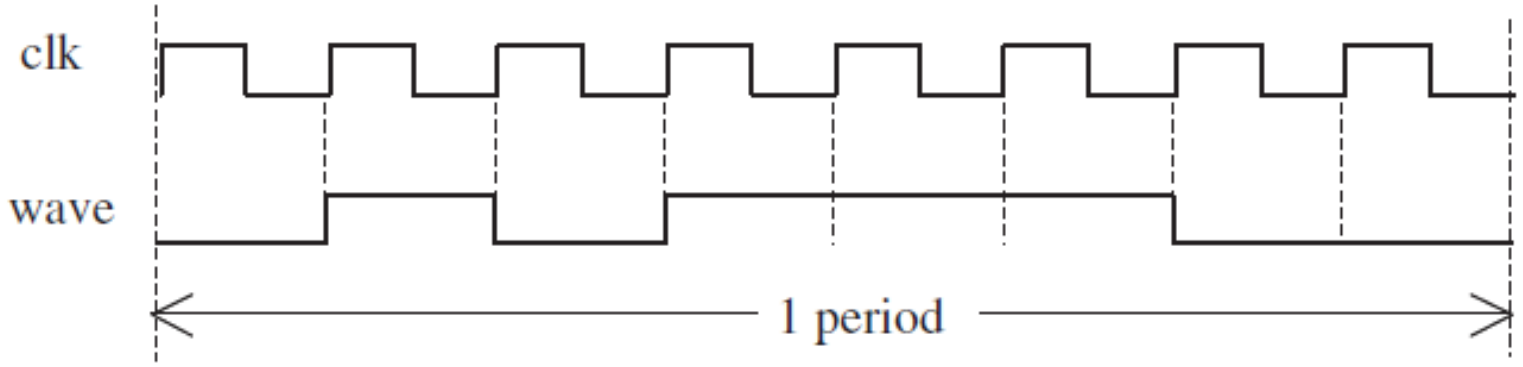
- Parallel-to-Serial Converter

```
1     ----------------------------------------
2     LIBRARY ieee;
3     USE ieee.std_logic_1164.all;
4     ----------------------------------------
5     ENTITY serial_converter IS
6        PORT ( d: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
7               clk, load: IN STD_LOGIC;
8               dout: OUT STD_LOGIC);
9     END serial_converter;
10 ----------------------------------------
```

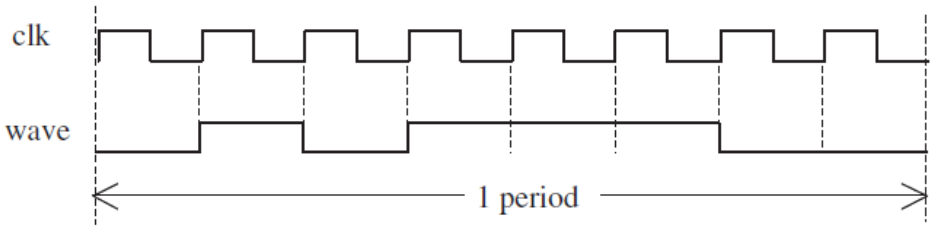
- Parallel-to-Serial Converter

```
11 ARCHITECTURE serial_converter OF serial_converter IS
12     SIGNAL reg: STD_LOGIC_VECTOR (7 DOWNTO 0);
13 BEGIN
14     PROCESS (clk)
15     BEGIN
16         IF (clk'EVENT AND clk='1') THEN
17             IF (load='1') THEN  reg <= d;
18             ELSE reg <= reg(6 DOWNTO 0) & '0';
19             END IF;
20         END IF;
21     END PROCESS;
22     dout <= reg(7);
23 END serial_converter;
24 ---------------------------------------------
```

- Signal Generators



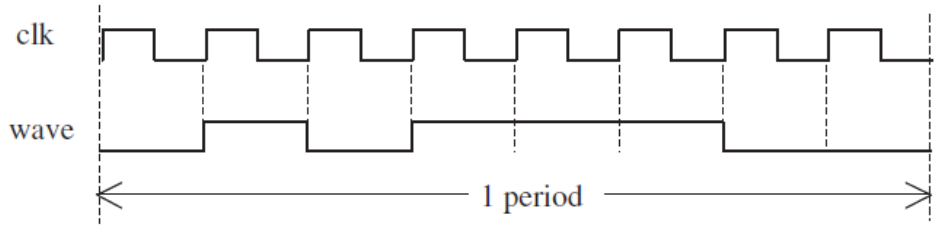
- Signal Generators




```
1  ----------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------
5  ENTITY signal_gen1 IS
6     PORT (clk: IN BIT;
7           wave: OUT BIT);
8  END signal_gen1;
9  ----------------------------------------
```

- Signal Generators

```
10 ARCHITECTURE arch1 OF signal_gen1 IS
11 BEGIN
12    PROCESS
13       VARIABLE count: INTEGER RANGE 0 TO 7;
14    BEGIN
15       WAIT UNTIL (clk'EVENT AND clk='1');
16       CASE count IS
17          WHEN 0 => wave <= '0';
18          WHEN 1 => wave <= '1';
19          WHEN 2 => wave <= '0';
20          WHEN 3 => wave <= '1';
21          WHEN 4 => wave <= '1';
22          WHEN 5 => wave <= '1';
23          WHEN 6 => wave <= '0':
24           WHEN 7 => wave <= '0';
25       END CASE;
26       count := count + 1;
27    END PROCESS;
28 END arch1;
29 ---------------------------
```



- ROM (Read Only Memory)

```
1  ------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------------------
5  ENTITY rom IS
6     GENERIC ( bits: INTEGER := 8;     -- # of bits per word
7               words: INTEGER := 8);  -- # of words in the memory
8     PORT ( addr: IN INTEGER RANGE 0 TO words-1;
9            data: OUT STD_LOGIC_VECTOR (bits-1 DOWNTO 0));
10 END rom;
```

- ROM (Read Only Memory)

```
11 ------------------------------------------------
12 ARCHITECTURE rom OF rom IS
13    TYPE vector_array IS ARRAY (0 TO words-1) OF
14       STD_LOGIC_VECTOR (bits-1 DOWNTO 0);
15    CONSTANT memory: vector_array := (  "00000000",
16                                        "00000010",
17                                        "00000100",
18                                        "00001000",
19                                        "00010000",
20                                        "00100000",
21                                        "01000000",
22                                        "10000000");
23 BEGIN
24    data <= memory(addr);
25 END rom;
26 ------------------------------------------------
```

The End. Thanks for listen

جامعة نينوى
كلية هندسة الإلكترونيات

HDL Programming -VHDL- 3

Textbook: Volnei A. Pedroni, “Circuit Design with VHDL”, MIT Press London, England, 2004.

Submitted By: Hussein M. H. Aideen

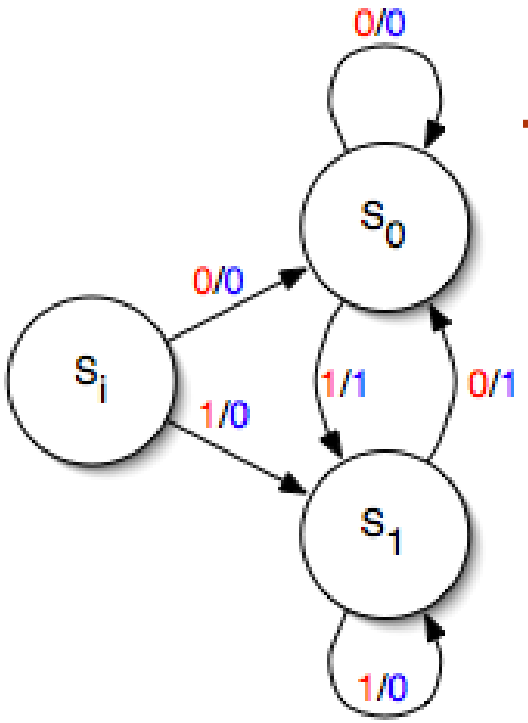
VHDL> State Machines

- Finite state machines (FSM) constitute a special modeling technique for sequential logic circuits.

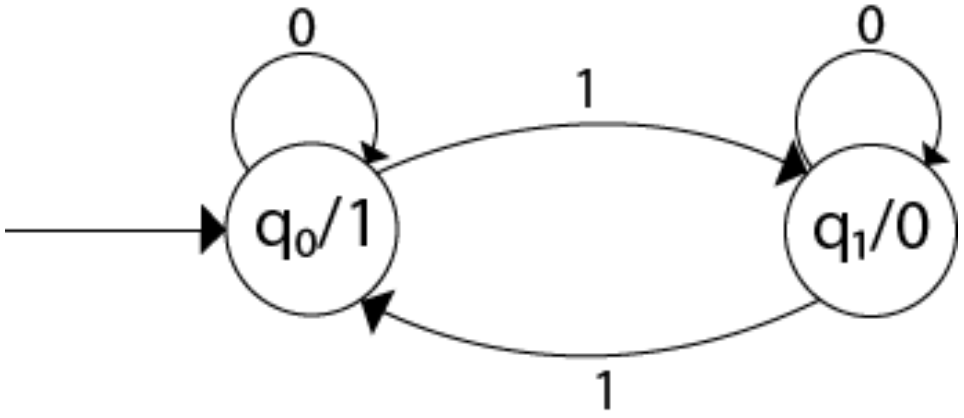
- helpful in the design of certain types of systems, (digital controllers, for example).

VHDL> State Machines

- Types of Machine modeling:
- Mealy machine: the output of the machine depends not only on the present state but also on the current input.



- Moore machine: the output depends only on the current state.



VHDL> State Machines

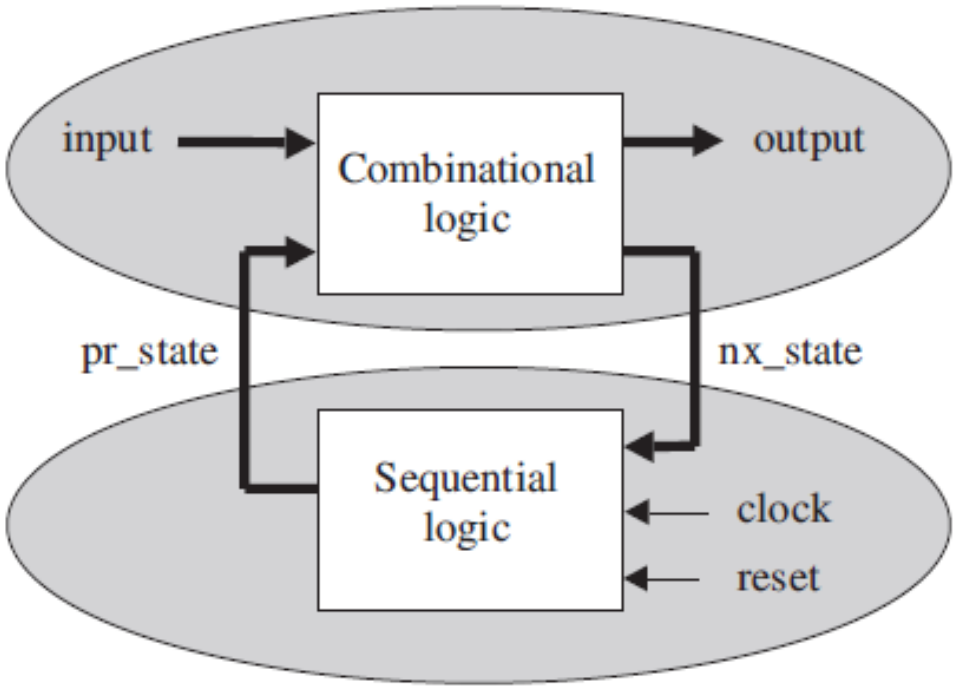


Figure 8.1
Mealy (Moore) state machine diagram.

VHDL> State Machines

FSM type definition:

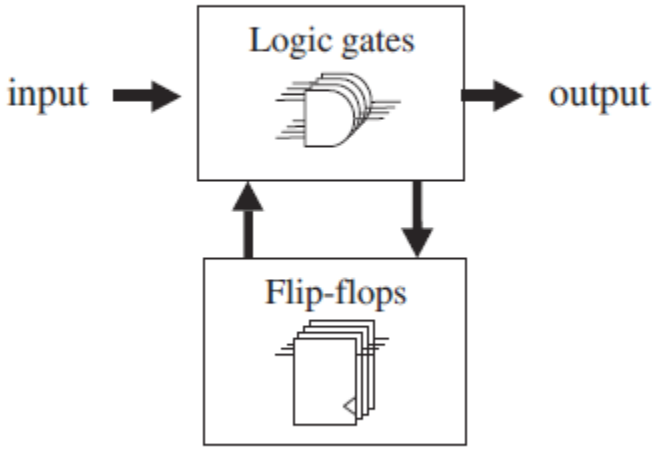
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------------
ENTITY <entity_name> IS
   PORT ( input: IN <data_type>;
          reset, clock: IN STD_LOGIC;
          output: OUT <data_type>);
END <entity_name>;
-----------------------------------------------
ARCHITECTURE <arch_name> OF <entity_name> IS
   TYPE state IS (state0, state1, state2, state3, ...);
   SIGNAL pr_state, nx_state: state;
BEGIN
```

VHDL> State Machines

FSM Design Styles:

Design Style #1:

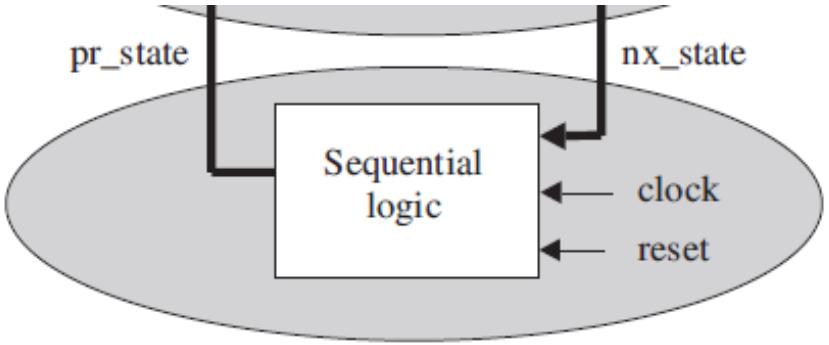
- The design of the lower section is completely separated from that of the upper section.



VHDL> State Machines

Design of the Lower (Sequential) Section:

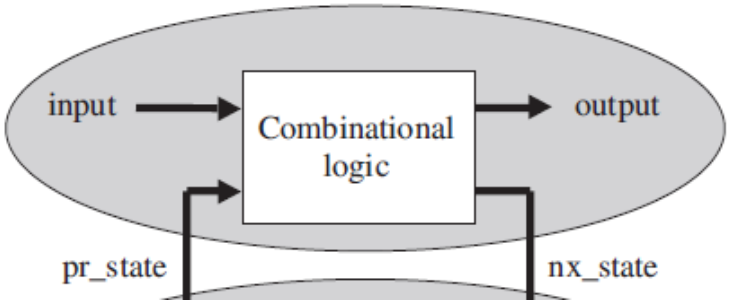
```
PROCESS (reset, clock)
BEGIN
    IF (reset='1') THEN
       pr_state <= state0;
    ELSIF (clock'EVENT AND clock='1') THEN
       pr_state <= nx_state;
    END IF;
END PROCESS;
```



VHDL> State Machines

Design of the Upper (Combinational) Section:

```
PROCESS (input, pr_state)
BEGIN
   CASE pr_state IS
      WHEN state0 =>
         IF (input = ...) THEN
            output <= <value>;
            nx_state <= state1;
         ELSE ...
         END IF;
      WHEN state1 =>
         IF (input = ...) THEN
            output <= <value>;
            nx_state <= state2;
```

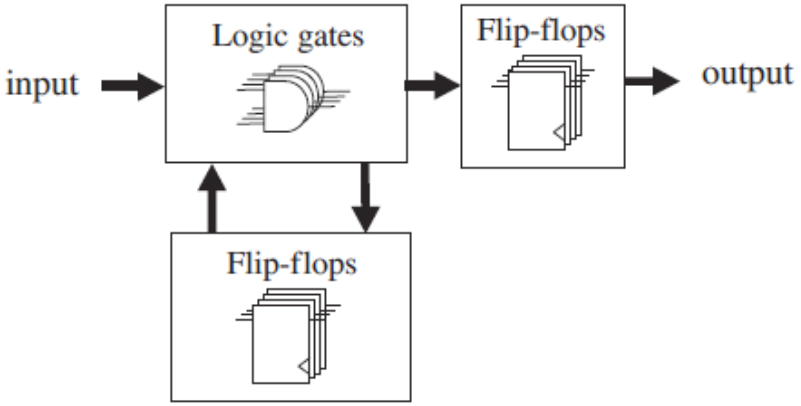


VHDL> State Machines

Design Style #2 (Stored Output):

- In Design style #1: Notice that in this case, if it is a Mealy machine (one whose output is dependent on the current input), the output might change when the input changes (asynchronous output).

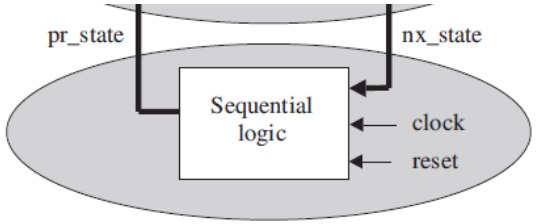
- To make Mealy machines synchronous.



VHDL> State Machines

State Machine Template for Design Style #2

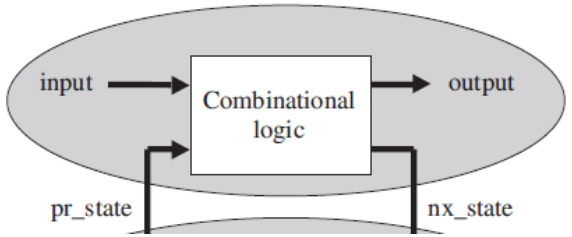
```
-------------------------------------------------------
ENTITY <ent_name> IS
   PORT (input: IN <data_type>;
         reset, clock: IN STD_LOGIC;
         output: OUT <data_type>);
END <ent_name>;
-------------------------------------------------------
ARCHITECTURE <arch_name> OF <ent_name> IS
   TYPE states IS (state0, state1, state2, state3, ...);
   SIGNAL pr_state, nx_state: states;
   SIGNAL temp: <data_type>;
BEGIN
   ---------- Lower section: -------------------------
   PROCESS (reset, clock)
   BEGIN
      IF (reset='1') THEN
         pr_state <= state0;
      ELSIF (clock'EVENT AND clock='1') THEN
         output <= temp;
         pr_state <= nx_state;
      END IF;
   END PROCESS;
```



VHDL> State Machines

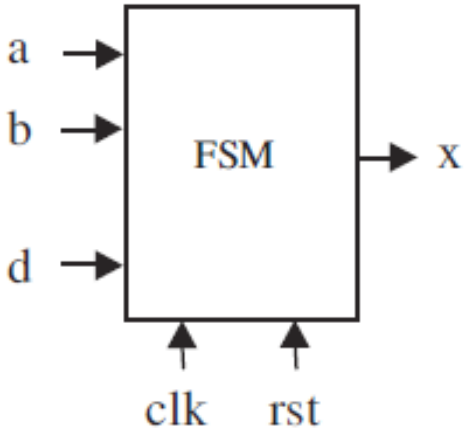
State Machine Template for Design Style #2 – cont.

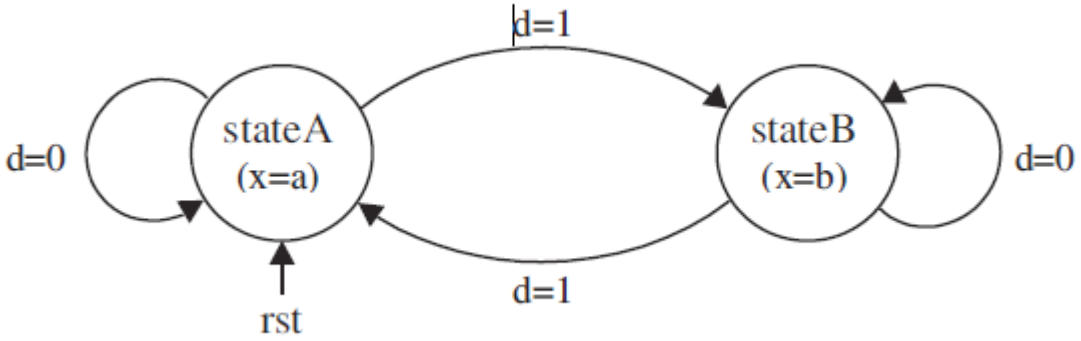
```
---------- Upper section: --------------------------
PROCESS (pr_state)
BEGIN
   CASE pr_state IS
      WHEN state0 =>
         temp <= <value>;
         IF (condition) THEN nx_state <= state1;
         ...
         END IF;
      WHEN state1 =>
         temp <= <value>;
         IF (condition) THEN nx_state <= state2;
         ...
         END IF;
      WHEN state2 =>
         temp <= <value>;
         IF (condition) THEN nx_state <= state3;
         ...
         END IF;
      ...
   END CASE;
END PROCESS;
END <arch_name>;
```



VHDL> State Machines

Example: Simple FSM #1

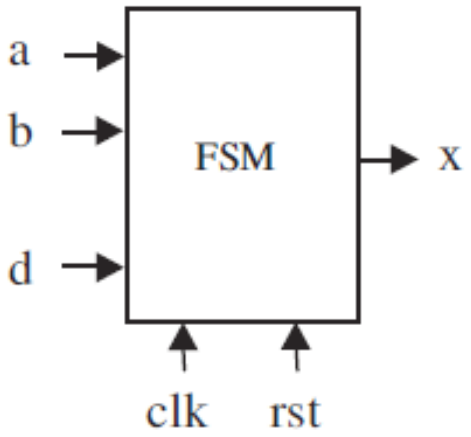




VHDL> State Machines

Example: Simple FSM #1

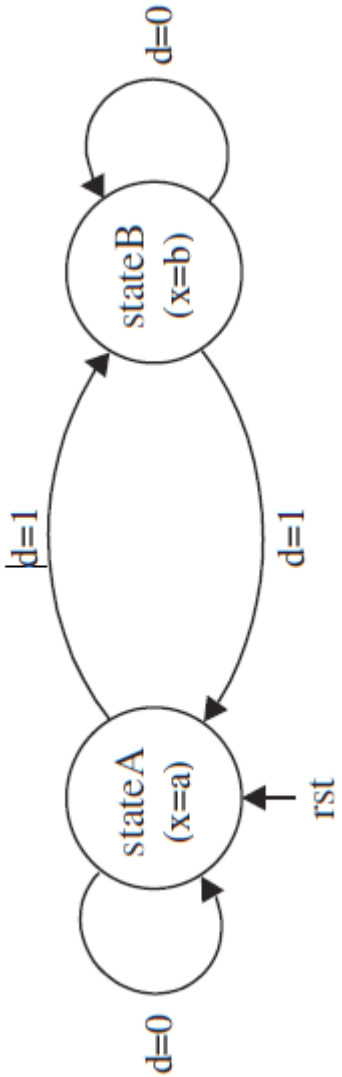
```
1  ----------------------------
2  ENTITY simple_fsm IS

3     PORT ( a, b, d, clk, rst: IN BIT;
4            x: OUT BIT);
5  END simple_fsm;
6  ----------------------------------------------
```



VHDL> State Machines

Example: Simple FSM #1

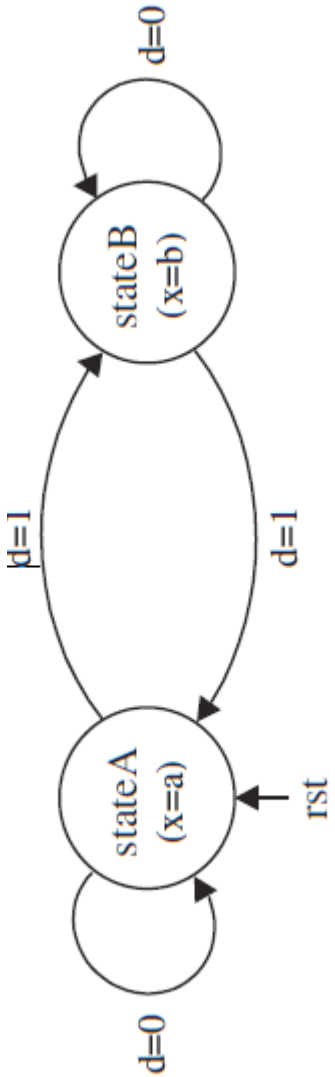
```
7  ARCHITECTURE simple_fsm OF simple_fsm IS
8     TYPE state IS (stateA, stateB);
9     SIGNAL pr_state, nx_state: state;
10 BEGIN
11    ----- Lower section: ----------------------
12    PROCESS (rst, clk)
13    BEGIN
14       IF (rst='1') THEN
15          pr_state <= stateA;
16       ELSIF (clk'EVENT AND clk='1') THEN
17          pr_state <= nx_state;
18       END IF;
19    END PROCESS;
```



VHDL> State Machines

Example: Simple FSM #1

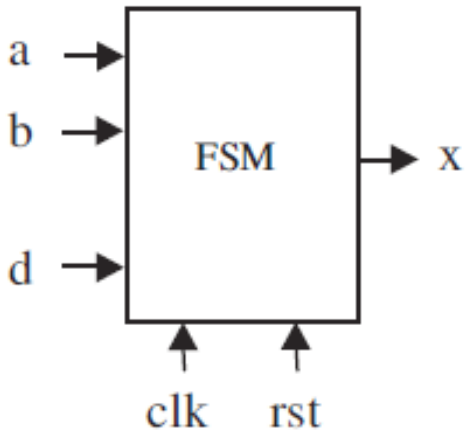
```
20    ---------- Upper section: ----------------
21    PROCESS (a, b, d, pr_state)
22    BEGIN
23       CASE pr_state IS
24          WHEN stateA =>
25             x <= a;
26             IF (d='1') THEN nx_state <= stateB;
27             ELSE nx_state <= stateA;
28             END IF;
29          WHEN stateB =>
30             x <= b;
31             IF (d='1') THEN nx_state <= stateA;
32             ELSE nx_state <= stateB;
33             END IF;
34       END CASE;
35    END PROCESS;
36 END simple_fsm;
```



VHDL> State Machines

Example: Simple FSM #2

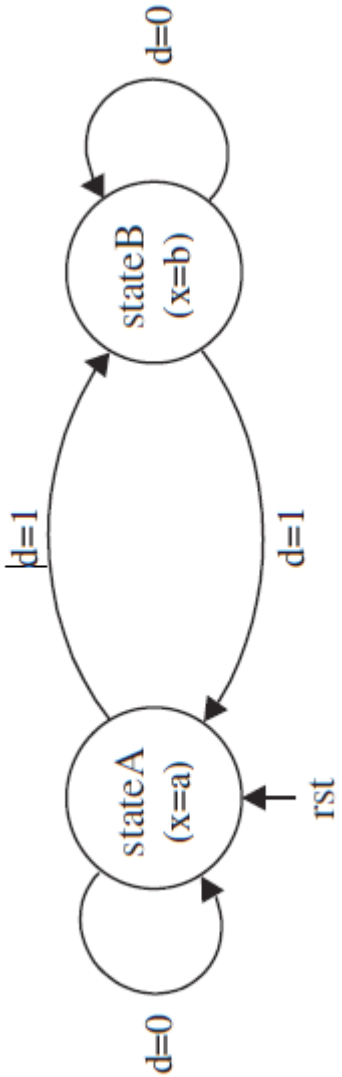
```
1  ----------------------------
2  ENTITY simple_fsm IS

3     PORT ( a, b, d, clk, rst: IN BIT;
4            x: OUT BIT);
5  END simple_fsm;
6  ------------------------------------------
```



VHDL> State Machines

Example: Simple FSM #2

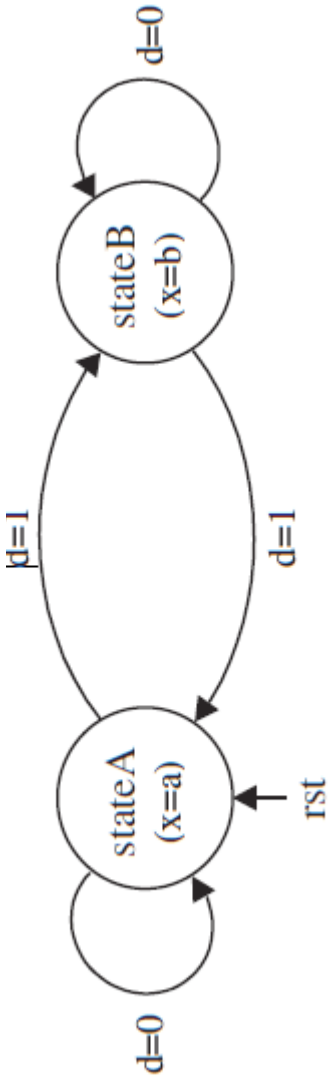
```
 7  ARCHITECTURE simple_fsm OF simple_fsm IS
 8     TYPE state IS (stateA, stateB);
 9     SIGNAL pr_state, nx_state: state;
10     SIGNAL temp: BIT;
11 BEGIN
12     ----- Lower section: ----------------------
13     PROCESS (rst, clk)
14     BEGIN
15        IF (rst='1') THEN
16           pr_state <= stateA;
17        ELSIF (clk'EVENT AND clk='1') THEN
18           x <= temp;
19           pr_state <= nx_state;
20        END IF;
21     END PROCESS;
```



VHDL> State Machines

Example: Simple FSM #2

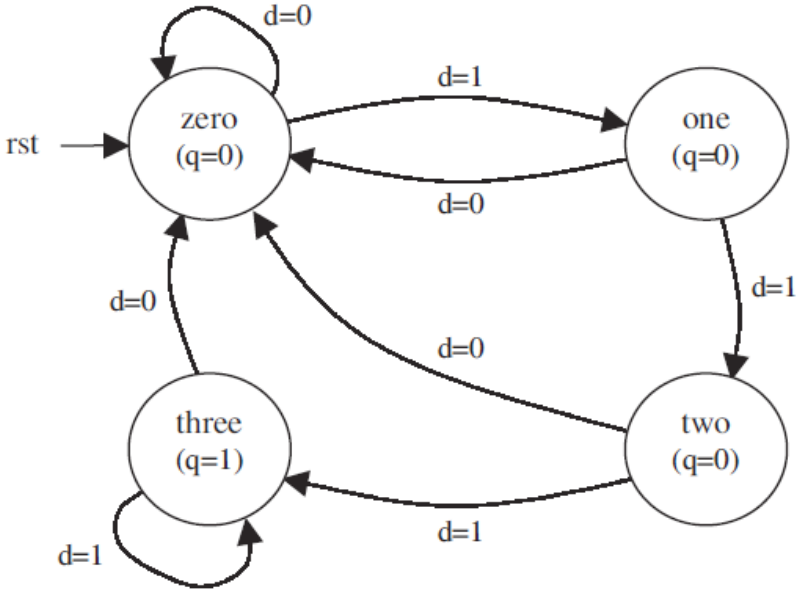
```
22     ---------- Upper section: ----------------
23     PROCESS (a, b, d, pr_state)
24     BEGIN
25        CASE pr_state IS
26           WHEN stateA =>
27              temp <= a;
28              IF (d='1') THEN nx_state <= stateB;
29              ELSE nx_state <= stateA;
30              END IF;
31           WHEN stateB =>
32              temp <= b;
33              IF (d='1') THEN nx_state <= stateA;
34              ELSE nx_state <= stateB;
35              END IF;
36        END CASE;
37     END PROCESS;
   END simple_fsm;
   ----------------------------------------
```



VHDL> State Machines

Example: String Detector

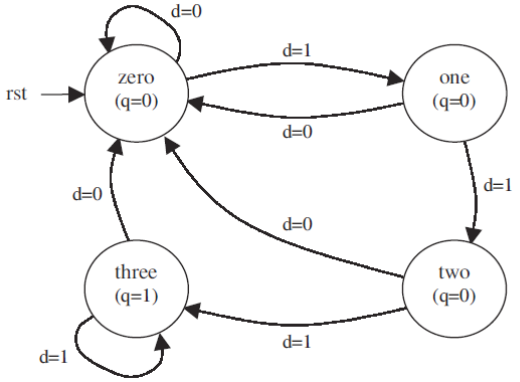
- We want to design a circuit that takes as input a serial bit stream and outputs a ‘1’ whenever the sequence “111” occurs. Overlaps must also be considered, that is, if . . .0111110 . . . occurs, than the output should remain active for three consecutive clock cycles.



VHDL> State Machines

Example: String Detector

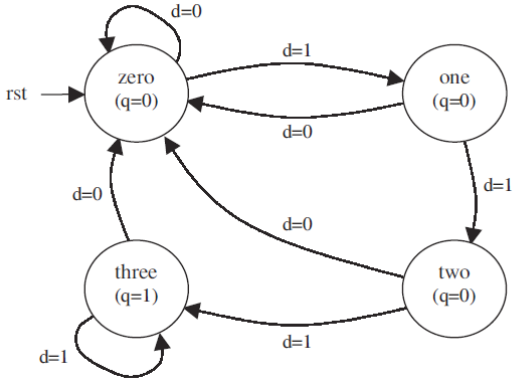
```
1  ----------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------
5  ENTITY string_detector IS
6     PORT ( d, clk, rst: IN BIT;
7            q: OUT BIT);
8  END string_detector;
9  ----------------------------------------
10 ARCHITECTURE my_arch OF string_detector IS
11    TYPE state IS (zero, one, two, three);
12    SIGNAL pr_state, nx_state: state;
```



VHDL> State Machines

Example: String Detector

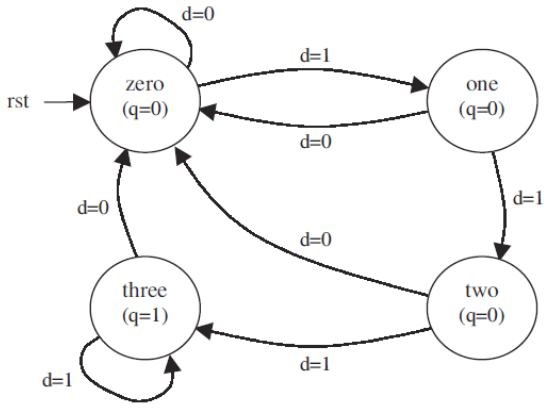
```
13 BEGIN
14     ----- Lower section: --------------------
15     PROCESS (rst, clk)
16     BEGIN
17        IF (rst='1') THEN
18           pr_state <= zero;
19        ELSIF (clk'EVENT AND clk='1') THEN
20           pr_state <= nx_state;
21        END IF;
22     END PROCESS;
```



VHDL> State Machines

Example: String Detector

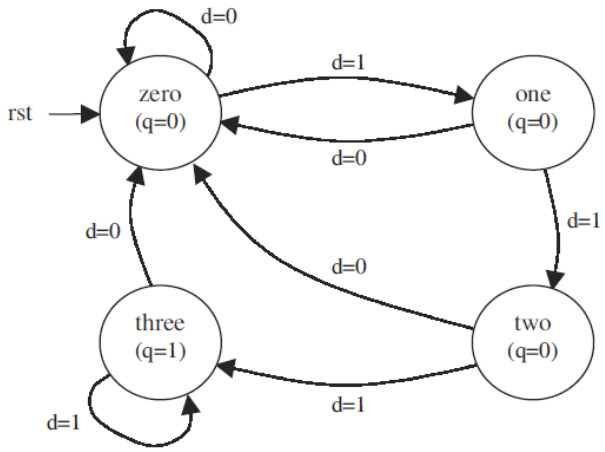
```
23    ---------- Upper section: ---------------
24    PROCESS (d, pr_state)
25    BEGIN
26       CASE pr_state IS
27          WHEN zero =>
28             q <= '0';
29             IF (d='1') THEN nx_state <= one;
30             ELSE nx_state <= zero;
31             END IF;
32          WHEN one =>
33             q <= '0';
34             IF (d='1') THEN nx_state <= two;
35             ELSE nx_state <= zero;
36             END IF;
37          WHEN two =>
38             q <= '0';
39             IF (d='1') THEN nx_state <= three;
```



VHDL> State Machines

Example: String Detector

```
40              ELSE nx_state <= zero;
41              END IF;
42           WHEN three =>
43              q <= '1';
44              IF (d='0') THEN nx_state <= zero;
45              ELSE nx_state <= three;
46              END IF;
47        END CASE;
48     END PROCESS;
49 END my_arch;
50 ----------------------------------------
```



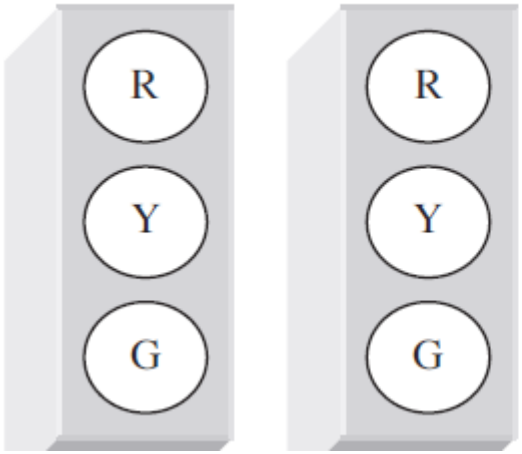
VHDL> State Machines

Example: Traffic Light Controller (TLC)

- Three modes of operation: Regular, Test, and Standby.
- Test mode: allows all pre-programmed times to be overwritten (by a manual switch) with a small value, such that the system can be easily tested during maintenance (1 second per state).
- Standby mode: the system should activate the yellow lights in both directions and remain so while the standby signal is active.
- Assume that a 60 Hz clock (obtained from the power line itself) is available.

VHDL> State Machines

Example: Traffic Light Controller (TLC)



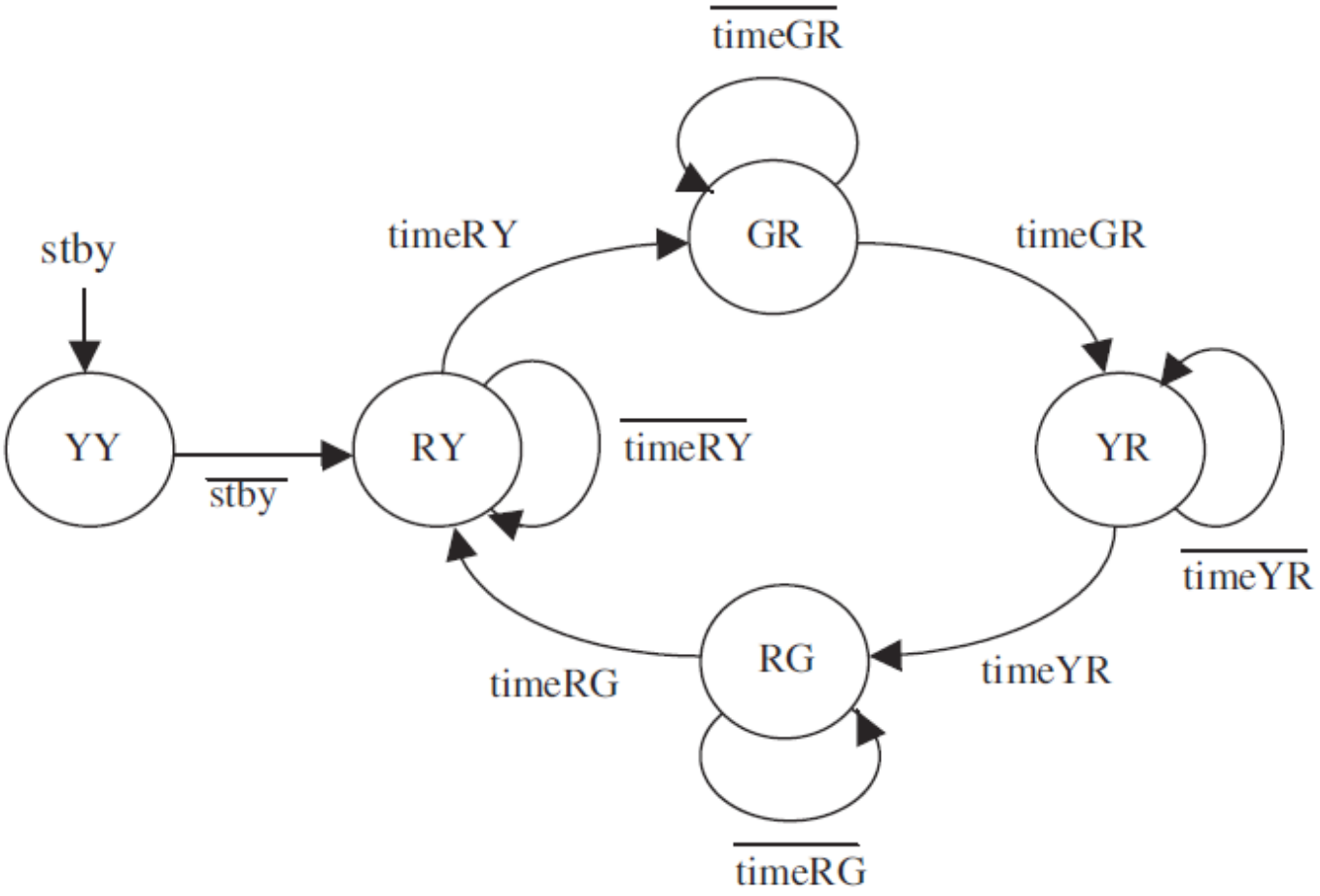
State	Operation Mode		
	REGULAR	TEST	STANDBY
	Time	Time	Time
RG	timeRG (30s)	timeTEST (1s)	---
RY	timeRY (5s)	timeTEST (1s)	---
GR	timeGR (45s)	timeTEST (1s)	---
YR	timeYR (5s)	timeTEST (1s)	---
YY	---	---	Indefinite

```
CONSTANT timeMAX : INTEGER := 2700;
CONSTANT timeRG : INTEGER := 1800;
CONSTANT timeRY : INTEGER := 300;
CONSTANT timeGR : INTEGER := 2700;
CONSTANT timeYR : INTEGER := 300;
CONSTANT timeTEST : INTEGER := 60;
```

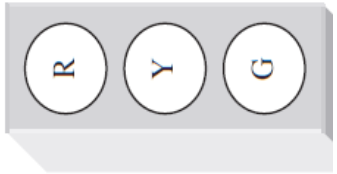
45 * 60
30 * 60
5 * 60
45 * 60
5 * 60
1 * 60

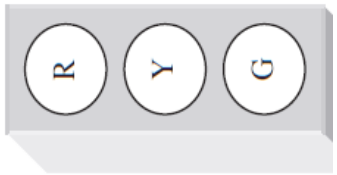
VHDL> State Machines

Example: Traffic Light Controller (TLC)



State	Operation Mode: REGULAR Time	Operation Mode: TEST Time	Operation Mode: STANDBY Time
RG	timeRG (30s)	timeTEST (1s)	---
RY	timeRY (5s)	timeTEST (1s)	---
GR	timeGR (45s)	timeTEST (1s)	---
YR	timeYR (5s)	timeTEST (1s)	---
YY	---	---	Indefinite



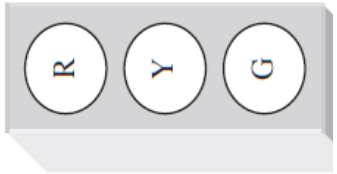


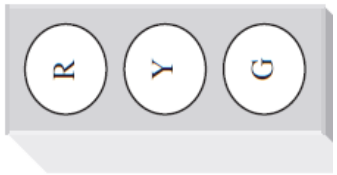
VHDL> State Machines

Example: Traffic Light Controller (TLC)

```
1  ------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------------------
5  ENTITY tlc IS
6     PORT ( clk, stby, test: IN STD_LOGIC;
7            r1, r2, y1, y2, g1, g2: OUT STD_LOGIC);
8  END tlc;
9  ------------------------------------------------
```

State	Operation Mode: REGULAR	Operation Mode: TEST	Operation Mode: STANDBY
	Time	Time	Time
RG	timeRG (30s)	timeTEST (1s)	---
RY	timeRY (5s)	timeTEST (1s)	---
GR	timeGR (45s)	timeTEST (1s)	---
YR	timeYR (5s)	timeTEST (1s)	---
YY	---	---	Indefinite

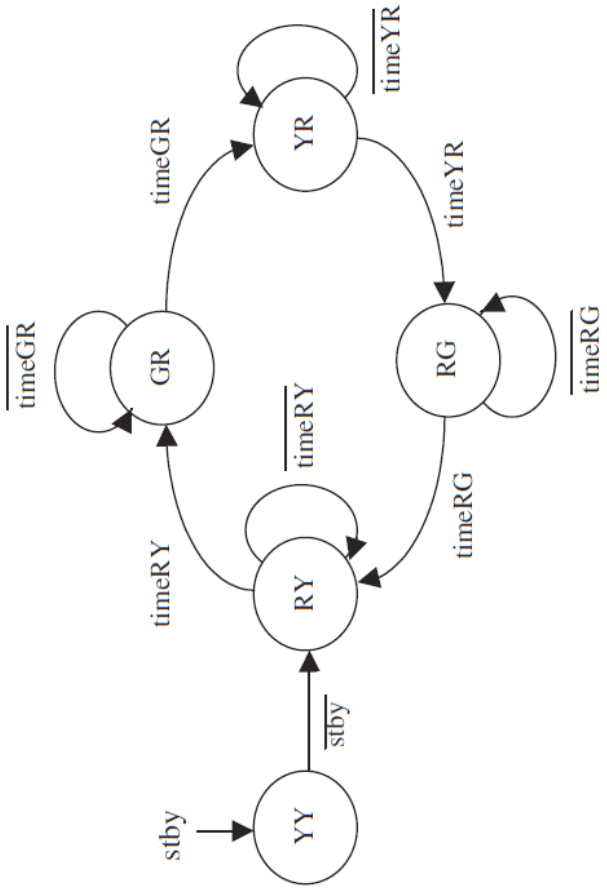




VHDL> State Machines

Example: Traffic Light Controller (TLC)

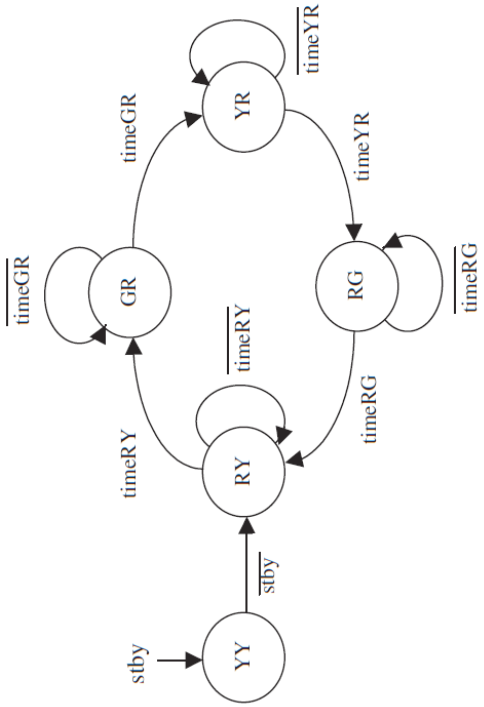
```
10 ARCHITECTURE behavior OF tlc IS
11     CONSTANT timeMAX : INTEGER := 2700;
12     CONSTANT timeRG : INTEGER := 1800;
13     CONSTANT timeRY : INTEGER := 300;
14     CONSTANT timeGR : INTEGER := 2700;
15     CONSTANT timeYR : INTEGER := 300;
16     CONSTANT timeTEST : INTEGER := 60;
17     TYPE state IS (RG, RY, GR, YR, YY);
18     SIGNAL pr_state, nx_state: state;
19     SIGNAL time : INTEGER RANGE 0 TO timeMAX;
20 BEGIN
```



VHDL> State Machines

Example: Traffic Light Controller (TLC)

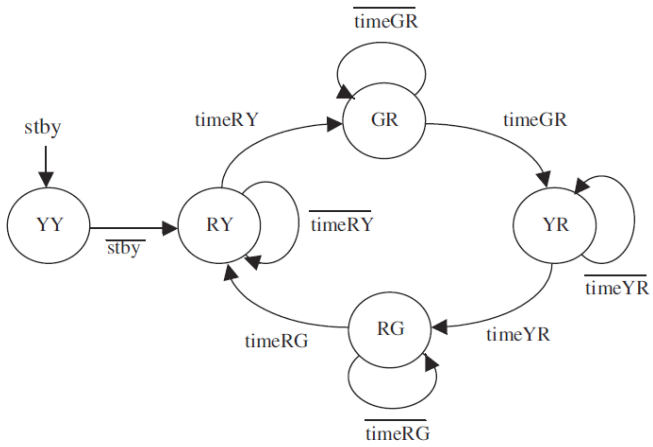
```
21    -------- Lower section of state machine: ----
22    PROCESS (clk, stby)
23       VARIABLE count : INTEGER RANGE 0 TO timeMAX;
24    BEGIN
25       IF (stby='1') THEN
26          pr_state <= YY;
27          count := 0;
28       ELSIF (clk'EVENT AND clk='1') THEN
29          count := count + 1;
30          IF (count = time) THEN
31             pr_state <= nx_state;
32             count := 0;
33          END IF;
34       END IF;
35    END PROCESS;
```



VHDL> State Machines

Example: Traffic Light Controller (TLC)

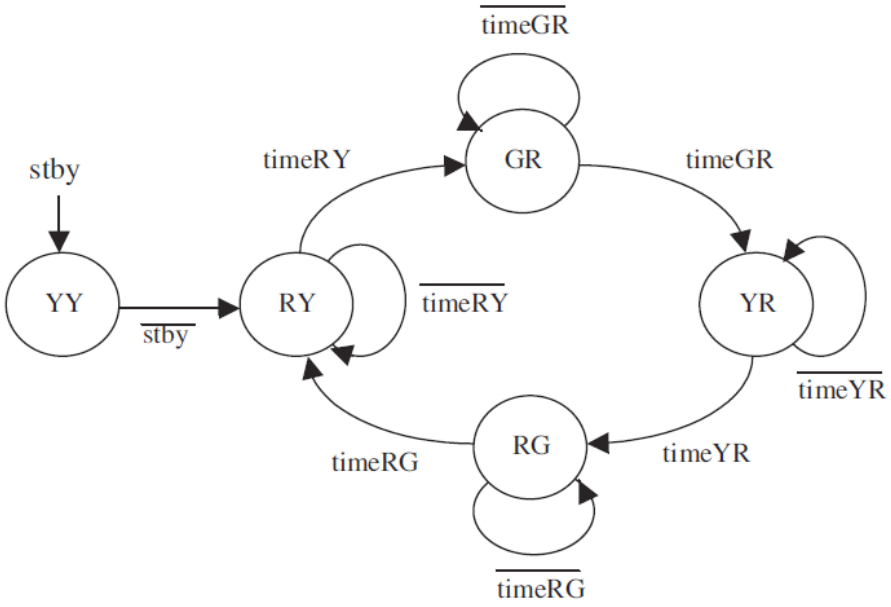
```
36    -------- Upper section of state machine: ----
37    PROCESS (pr_state, test)
38    BEGIN
39       CASE pr_state IS
40          WHEN RG =>
41             r1<='1'; r2<='0'; y1<='0'; y2<='0'; g1<='0'; g2<='1';
42             nx_state <= RY;
43             IF (test='0') THEN time <= timeRG;
44             ELSE time <= timeTEST;
45             END IF;
```



VHDL> State Machines

Example: Traffic Light Controller (TLC)

```
46          WHEN RY =>
47              r1<='1'; r2<='0'; y1<='0'; y2<='1'; g1<='0'; g2<='0';
48              nx_state <= GR;
49              IF (test='0') THEN time <= timeRY;
50              ELSE time <= timeTEST;
51              END IF;
```



VHDL> State Machines

Example: Traffic Light Controller (TLC)

```
52          WHEN GR =>
53             r1<='0'; r2<='1'; y1<='0'; y2<='0'; g1<='1'; g2<='0';
54             nx_state <= YR;
55             IF (test='0') THEN time <= timeGR;
56             ELSE time <= timeTEST;
57             END IF;
58          WHEN YR =>
59             r1<='0'; r2<='1'; y1<='1'; y2<='0'; g1<='0'; g2<='0';
60             nx_state <= RG;
61             IF (test='0') THEN time <= timeYR;
62             ELSE time <= timeTEST;
63             END IF;
64          WHEN YY =>
65             r1<='0'; r2<='0'; y1<='1'; y2<='1'; g1<='0'; g2<='0';
66             nx_state <= RY;
67       END CASE;
68    END PROCESS;
69 END behavior;
```

VHDL> State Machines

FSM Encoding Style:

- To encode the states of a state machine, we can select one among several available styles.
- The default style is binary.
  - requires the least number of flip-flops.
  - with n flip-flops (n bits), up to 2^n states can be encoded
  - requires more logic and is slower than the others.
- One-hot encoding style,
  - which uses one flip-flop per state. Therefore, it demands the largest number of flip-flops.
  - with n flip-flops (n bits), only n states can be encoded.
  - requires the least amount of extra logic and is the fastest.

VHDL> State Machines

FSM Encoding Style:

- Two-hot encoding scheme:
  - An style that is inbetween the two styles above.
  - which presents two bits active per state.
  - Therefore, with n flip-flops (n bits), up to n(n-1)/2 states can be encoded.

	Encoding Style		
STATE	BINARY	TWOHOT	ONEHOT
state0	000	00011	00000001
state1	001	00101	00000010
state2	010	01001	00000100
state3	011	10001	00001000
state4	100	00110	00010000
state5	101	01010	00100000
state6	110	10010	01000000
state7	111	01100	10000000

جامعة نينوى
كلية هندسة الإلكترونيات

HDL Programming -VHDL- 5

Textbook: Volnei A. Pedroni, “Circuit Design with VHDL”, MIT Press London, England, 2004.

Submitted By: Hussein M. H. Aideen

VHDL> Additional System Design

- FUNCTION
  - A FUNCTION is a section of sequential code.
  - Its purpose is to create new functions to deal with commonly encountered problems, like data type conversions, logical operations, arithmetic computations, and new operators and attributes.
  - By writing such code as a FUNCTION, it can be shared and reused, also propitiating the main code to be shorter and easier to understand.

VHDL> Additional System Design

- FUNCTION

```
FUNCTION function_name [<parameter list>] RETURN data_type IS
    [declarations]
BEGIN
    (sequential statements)
END function_name;
```

- Example

```
FUNCTION f1 (a, b: INTEGER; SIGNAL c: STD_LOGIC_VECTOR)
    RETURN BOOLEAN IS
BEGIN
    (sequential statements)
END f1;
```

VHDL> Additional System Design

- Digital Filters
  - Digital signal processing (DSP) finds innumerable applications in the fields of audio, video, and communications, among others. Such applications are generally based on LTI (linear time invariant) systems, which can be implemented with digital circuitry.
  - Any LTI system be represented by the following equation:

$$\sum_{k=0}^{N} a_k y[n-k] = \sum_{k=0}^{M} b_k x[n-k]$$

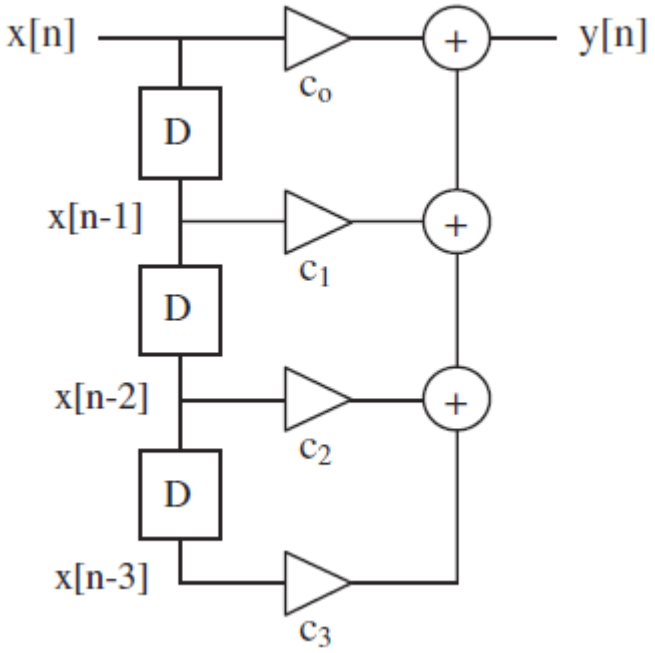
$$y[n] = \frac{1}{a_0}\sum_{k=0}^{M} b_k x[n-k] - \frac{1}{a_0}\sum_{k=1}^{N} a_k y[n-k]$$

VHDL> Additional System Design

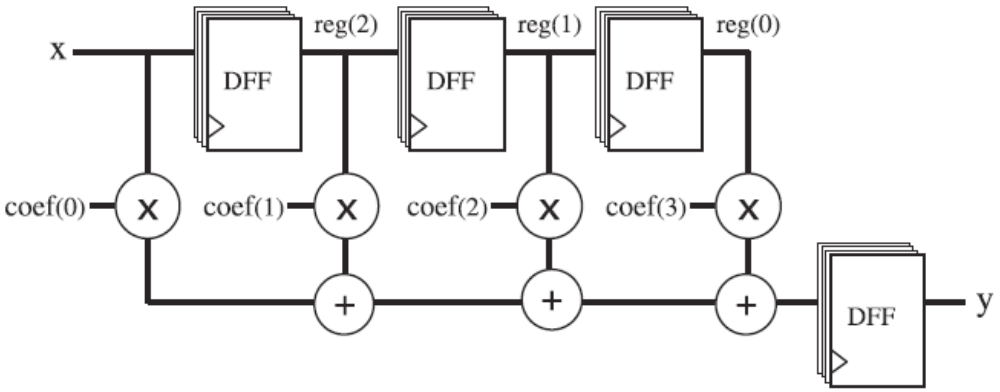
- Digital Filters
  - FIR Filter:

$$Y(n) = \sum_{K=0}^{N} b_K x(n-k)$$

4-tap FIR Filter



RTL Representation



VHDL> Additional System Design

- Digital Filters
  - VHDL code for FIR Filter:

```
1  ----------------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE ieee.std_logic_arith.all;  -- package needed for SIGNED
5  ----------------------------------------------------------
6  ENTITY fir2 IS
7     GENERIC (n: INTEGER := 4; m: INTEGER := 4);
8     -- n = # of coef., m = # of bits of input and coef.
9     -- Besides n and m, CONSTANT (line 19) also need adjust
10    PORT ( x: IN SIGNED(m-1 DOWNTO 0);
11           clk, rst: IN STD_LOGIC;
12           y: OUT SIGNED(2*m-1 DOWNTO 0));
13 END fir2;
14 ----------------------------------------------------------
```

VHDL> Additional System Design

- Digital Filters
  - VHDL code for FIR Filter: (continued)

```
14 ------------------------------------------------------------
15 ARCHITECTURE rtl OF fir2 IS
16    TYPE registers IS ARRAY (n-2 DOWNTO 0) OF
17                             SIGNED(m-1 DOWNTO 0);
18    TYPE coefficients IS ARRAY (n-1 DOWNTO 0) OF
19                                SIGNED(m-1 DOWNTO 0);
20    SIGNAL reg: registers;
21    CONSTANT coef: coefficients := ("0001", "0010", "0011",
22                                     "0100");
```

VHDL> Additional System Design

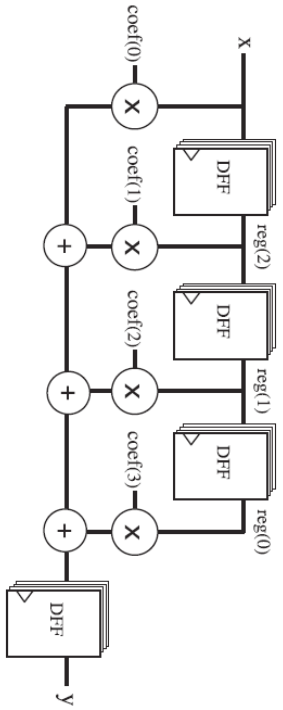
- Digital Filters
  - VHDL code for FIR Filter: (continued)

```
23 BEGIN
24    PROCESS (clk, rst)
25       VARIABLE acc, prod:
26          SIGNED(2*m-1 DOWNTO 0) := (OTHERS=>'0');
27       VARIABLE sign: STD_LOGIC;
28    BEGIN
29       ----- reset: -------------------------

30       IF (rst='1') THEN
31          FOR i IN n-2 DOWNTO 0 LOOP
32             FOR j IN m-1 DOWNTO 0 LOOP
33                reg(i)(j) <= '0';
34             END LOOP;
35          END LOOP;
```

VHDL> Additional System Design

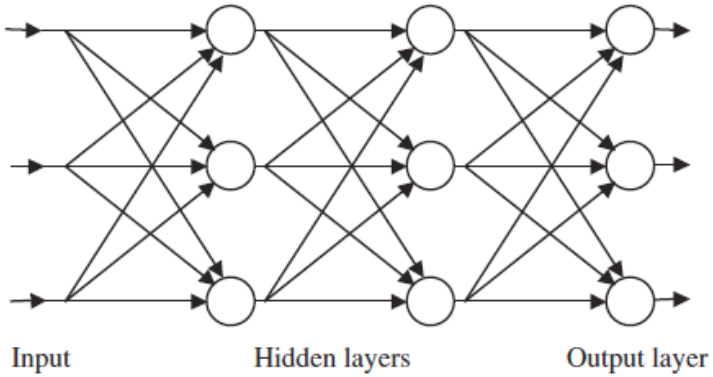
- Digital Filters
  - VHDL code for FIR Filter: (continued)

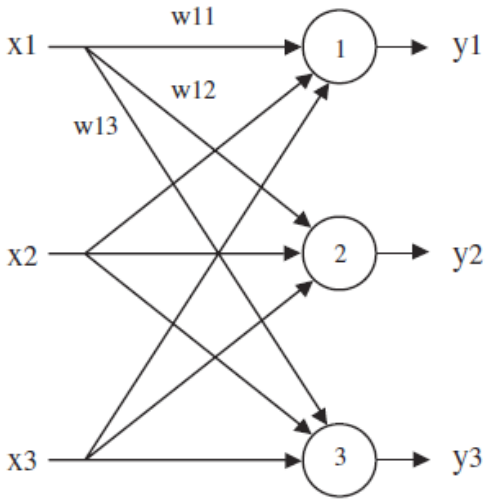
```
36        ----- register inference + MAC: -------
37        ELSIF (clk'EVENT AND clk='1') THEN
38           acc := coef(0)*x;
39           FOR i IN 1 TO n-1 LOOP
40              sign := acc(2*m-1);
41              prod := coef(i)*reg(n-1-i);
42              acc := acc + prod;
43              ---- overflow check: ------------
44              IF (sign=prod(prod'left)) AND
45                    (acc(acc'left) /= sign)
46                 THEN
47                 acc := (acc'LEFT => sign, OTHERS => NOT sign);
48              END IF;
49           END LOOP;
50           reg <= x & reg(n-2 DOWNTO 1);
51        END IF;
52        y <= acc;
53     END PROCESS;
54  END rtl;
```



VHDL> Additional System Design

- Neural Networks
  - Neural Networks (NN) are highly parallel, highly interconnected systems. Such characteristics make their implementation very challenging, and also very costly, due to the large amount of hardware required.
  - 3 layer network layer details



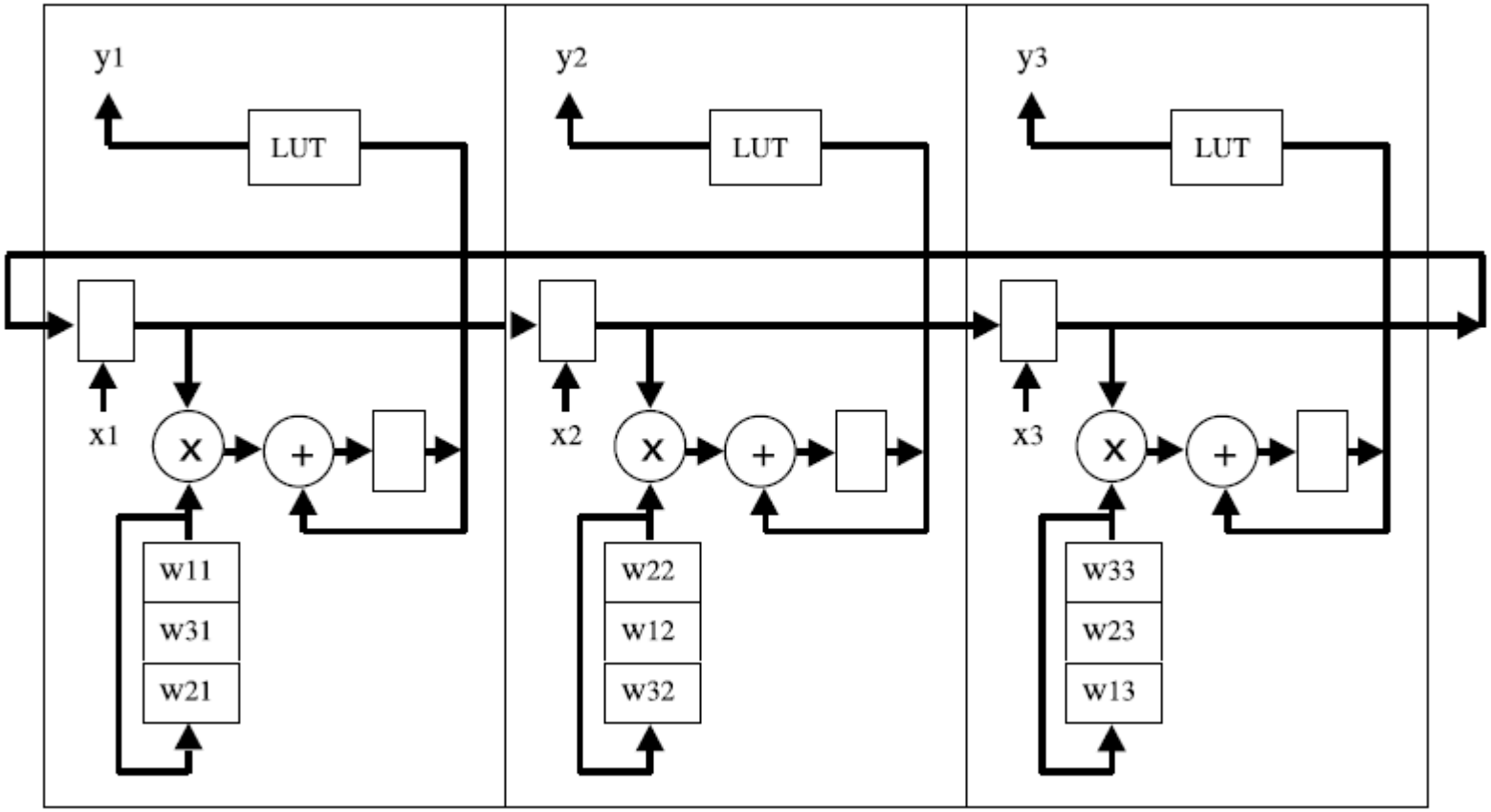


VHDL> Additional System Design

- Neural Networks
- A “ring” architecture for the NN is presented in figure, which implements one layer of the NN.
- Each box represents one neuron.
- There are several circular shift registers, one for each neuron (vertical shifters) plus one for the whole set (horizontal shifter).
- The vertical shifters hold the weights, while the horizontal one holds the inputs (shift registers with ‘data_load’ capability).
- At the output of a vertical shifter there is a MAC circuit, which accumulates the product between the weights and the inputs.

VHDL> Additional System Design

- Neural Networks



VHDL> Additional System Design

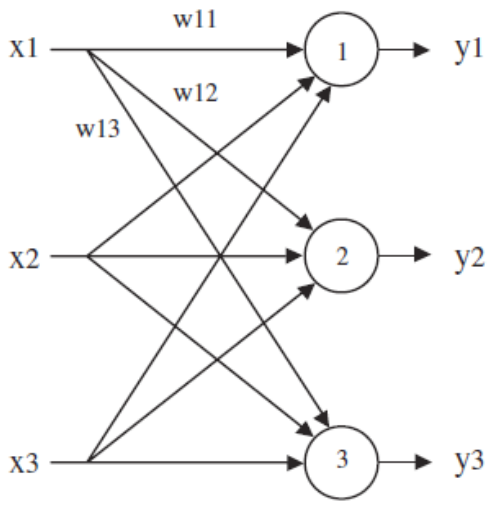
- Neural Networks
- For Small Neural Networks The solution below has the advantage of being simple, easily understandable, and self-contained in the main code. Its only limitation is that the inputs (x) and outputs (y) are specified one by one rather than using some kind of two-dimensional array, thus making it inappropriate for large NNs. Everything else is generic.

```
1  ----------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE ieee.std_logic_arith.all;  -- package needed for SIGNED
5  ----------+-----------------------------------------
```

VHDL> Additional System Design

- Neural Networks

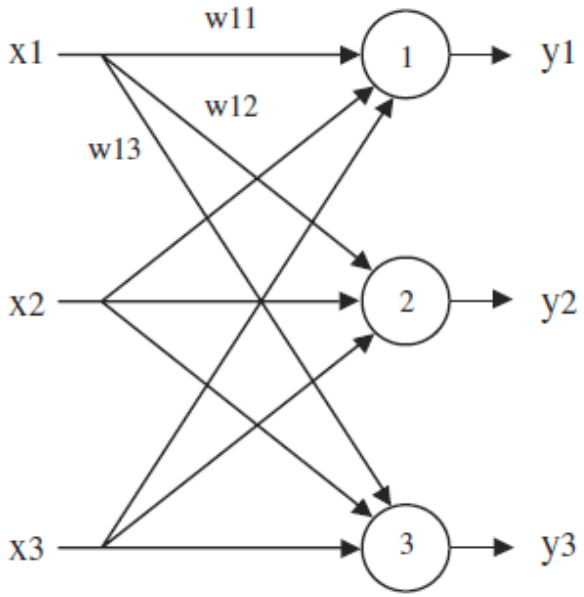
```
6  ENTITY nn IS
7  GENERIC ( n: INTEGER := 3;  -- # of neurons
8            m: INTEGER := 3; -- # of inputs or weights per neuron
9            b: INTEGER := 4); -- # of bits per input or weight
10 PORT ( x1: IN SIGNED(b-1 DOWNTO 0);
11        x2: IN SIGNED(b-1 DOWNTO 0);
12        x3: IN SIGNED(b-1 DOWNTO 0);
13        w: IN SIGNED(b-1 DOWNTO 0);
14        clk: IN STD_LOGIC;
15        test: OUT SIGNED(b-1 DOWNTO 0);  -- register test output
16        y1: OUT SIGNED(2*b-1 DOWNTO 0);
17        y2: OUT SIGNED(2*b-1 DOWNTO 0);
18        y3: OUT SIGNED(2*b-1 DOWNTO 0));
19 END nn;
```



VHDL> Additional System Design

- Neural Networks

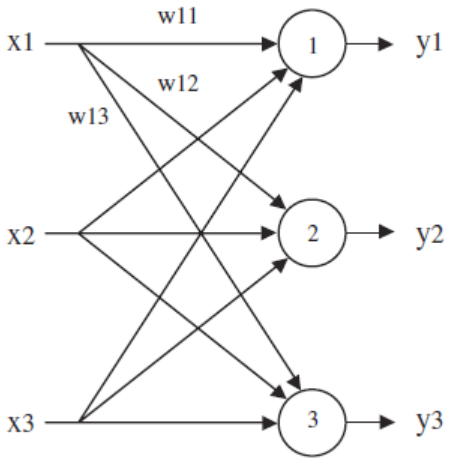
```
20 ------------------------------------------------------------
21 ARCHITECTURE neural OF nn IS
22    TYPE weights IS ARRAY (1 TO n*m) OF SIGNED(b-1 DOWNTO 0);
23    TYPE inputs IS ARRAY (1 TO m) OF SIGNED(b-1 DOWNTO 0);
24    TYPE outputs IS ARRAY (1 TO m) OF SIGNED(2*b-1 DOWNTO 0);
```



VHDL> Additional System Design

- Neural Networks

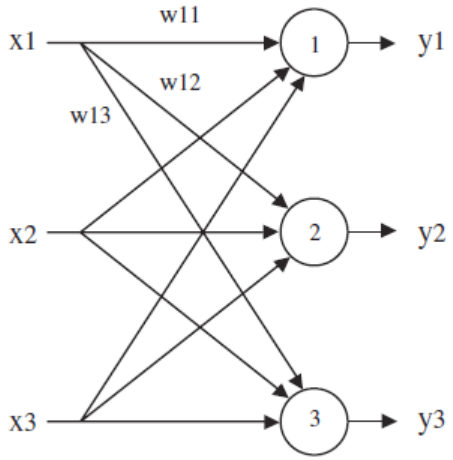
```
25 BEGIN
26    PROCESS (clk, w, x1, x2, x3)
27       VARIABLE weight: weights;
28       VARIABLE input: inputs;
29       VARIABLE output: outputs;
30       VARIABLE prod, acc: SIGNED(2*b-1 DOWNTO 0);
31       VARIABLE sign: STD_LOGIC;
32    BEGIN
33       ----- shift register inference: ------------
34       IF (clk'EVENT AND clk='1') THEN
35          weight := w & weight(1 TO n*m-1);
36       END IF;
37       --------- initialization: ------------------
38       input(1) := x1;
39       input(2) := x2;
40       input(3) := x3;
```

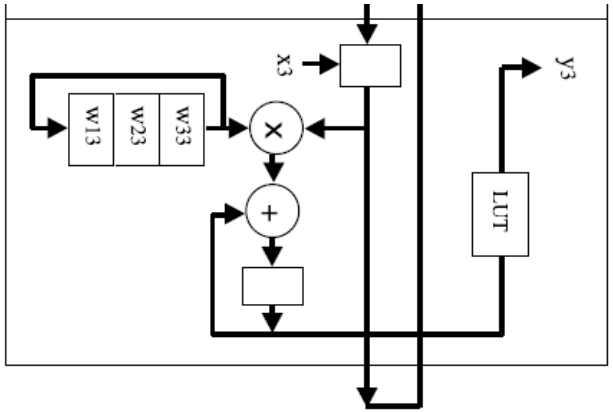


VHDL> Additional System Design

- Neural Networks

```
41        ------ multiply-accumulate: -----------------
42        L1: FOR i IN 1 TO n LOOP
43           acc := (OTHERS => '0');
44           L2: FOR j IN 1 TO m LOOP
45              prod := input(j)*weigth(m*(i-1)+j);
46              sign := acc(acc'LEFT);
47              acc := acc + prod;
48              ---- overflow check: -----------------
49              IF (sign=prod(prod'left)) AND
50                    (acc(acc'left) /= sign) THEN
51                 acc := (acc'LEFT => sign, OTHERS => NOT sign);
52              END IF;
53           END LOOP L2;
54           output(i) := acc;
55        END LOOP L1;
```

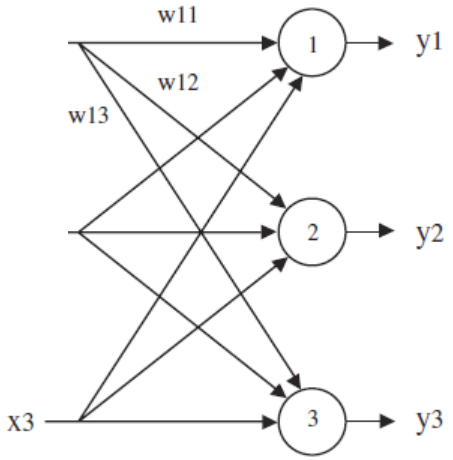




VHDL> Additional System Design

- Neural Networks

```
56        --------- outputs: -------------------------
57        test <= weight(n*m);
58        y1 <= output(1);
59        y2 <= output(2);
60        y3 <= output(3);
61     END PROCESS;
62 END neural;
```



VHDL> Additional System Design

- Neural Networks