








## What are FPGAs?

Field programmable gate arrays (FPGAs) are digital integrated circuits (ICs) that contain configurable (programmable) blocks of logic along with configurable interconnects between these blocks. Design engineers can configure (program) such devices to perform a tremendous variety of tasks.

## Logic Gates

Name	NOT	AND	NAND	OR	NOR	XOR	XNOR																																																																																																
Alg. Expr.	$\overline{A}$	$AB$	$\overline{AB}$	$A+B$	$\overline{A+B}$	$A\oplus B$	$\overline{A\oplus B}$																																																																																																
Symbol																																																																																																							
Truth Table	<table><tr><th>A</th><th>X</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	X	0	1	1	0	<table><tr><th>B</th><th>A</th><th>X</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	B	A	X	0	0	0	0	1	0	1	0	0	1	1	1	<table><tr><th>B</th><th>A</th><th>X</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	B	A	X	0	0	1	0	1	1	1	0	1	1	1	0	<table><tr><th>B</th><th>A</th><th>X</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	B	A	X	0	0	0	0	1	1	1	0	1	1	1	1	<table><tr><th>B</th><th>A</th><th>X</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	B	A	X	0	0	1	0	1	0	1	0	0	1	1	0	<table><tr><th>B</th><th>A</th><th>X</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	B	A	X	0	0	0	0	1	1	1	0	1	1	1	0	<table><tr><th>B</th><th>A</th><th>X</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	B	A	X	0	0	1	0	1	0	1	0	0	1	1	1
A	X																																																																																																						
0	1																																																																																																						
1	0																																																																																																						
B	A	X																																																																																																					
0	0	0																																																																																																					
0	1	0																																																																																																					
1	0	0																																																																																																					
1	1	1																																																																																																					
B	A	X																																																																																																					
0	0	1																																																																																																					
0	1	1																																																																																																					
1	0	1																																																																																																					
1	1	0																																																																																																					
B	A	X																																																																																																					
0	0	0																																																																																																					
0	1	1																																																																																																					
1	0	1																																																																																																					
1	1	1																																																																																																					
B	A	X																																																																																																					
0	0	1																																																																																																					
0	1	0																																																																																																					
1	0	0																																																																																																					
1	1	0																																																																																																					
B	A	X																																																																																																					
0	0	0																																																																																																					
0	1	1																																																																																																					
1	0	1																																																																																																					
1	1	0																																																																																																					
B	A	X																																																																																																					
0	0	1																																																																																																					
0	1	0																																																																																																					
1	0	0																																																																																																					
1	1	1																																																																																																					

Logic Functions: An elementary processing function in a digital circuit.

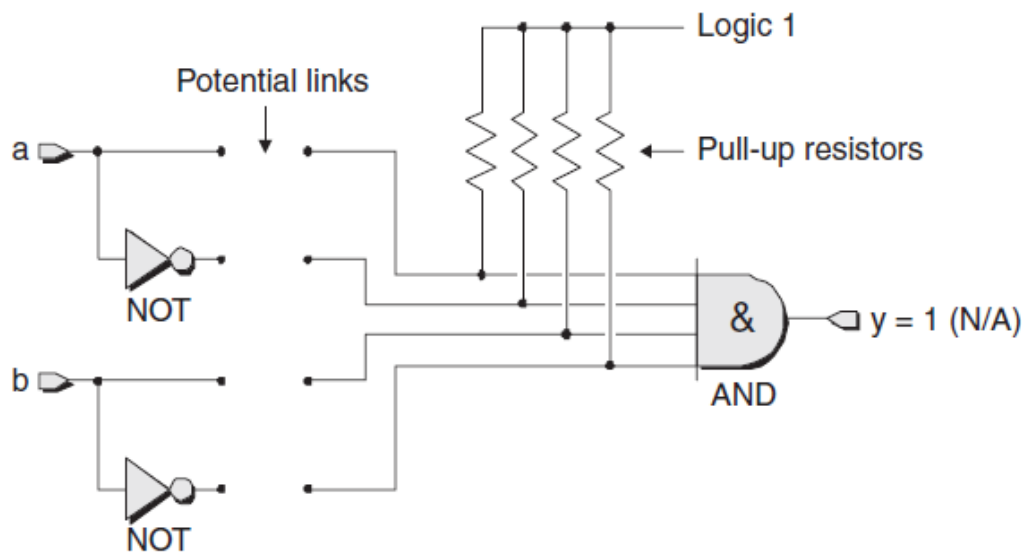
Logic functions can be implemented by logic gates, and fabricated to silicon chip to perform a unique function this design is called Application Specific Integrated Circuit (ASIC).

## The key thing about FPGAs

The thing that really distinguishes an FPGA from an ASIC is ... the crucial aspect that resides at the core of their reason for being is ... embodied in their name:



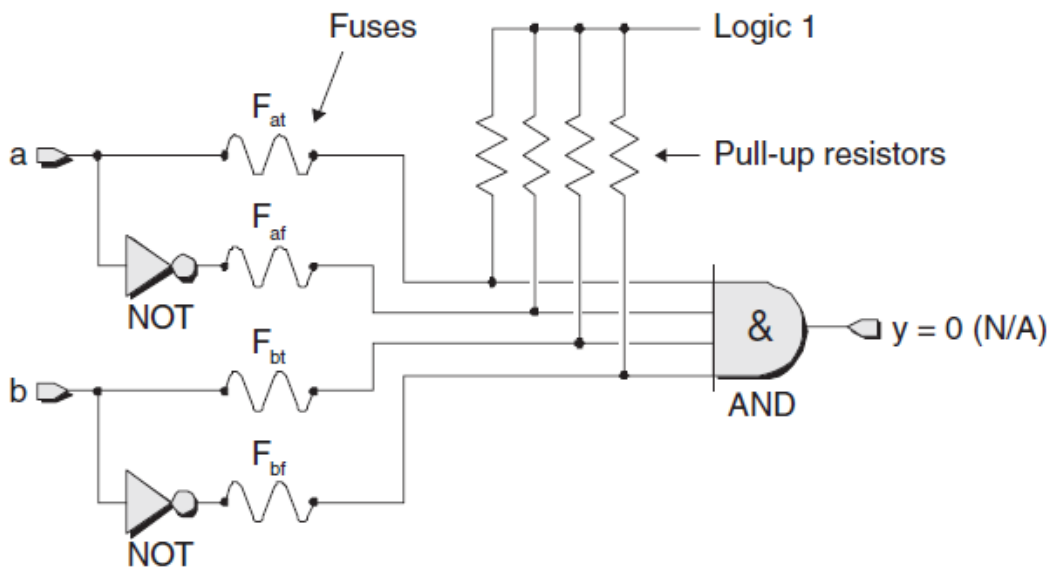
All joking aside, the point is that in order to be programmable, we need some mechanism that allows us to configure (program) a prebuilt silicon chip.



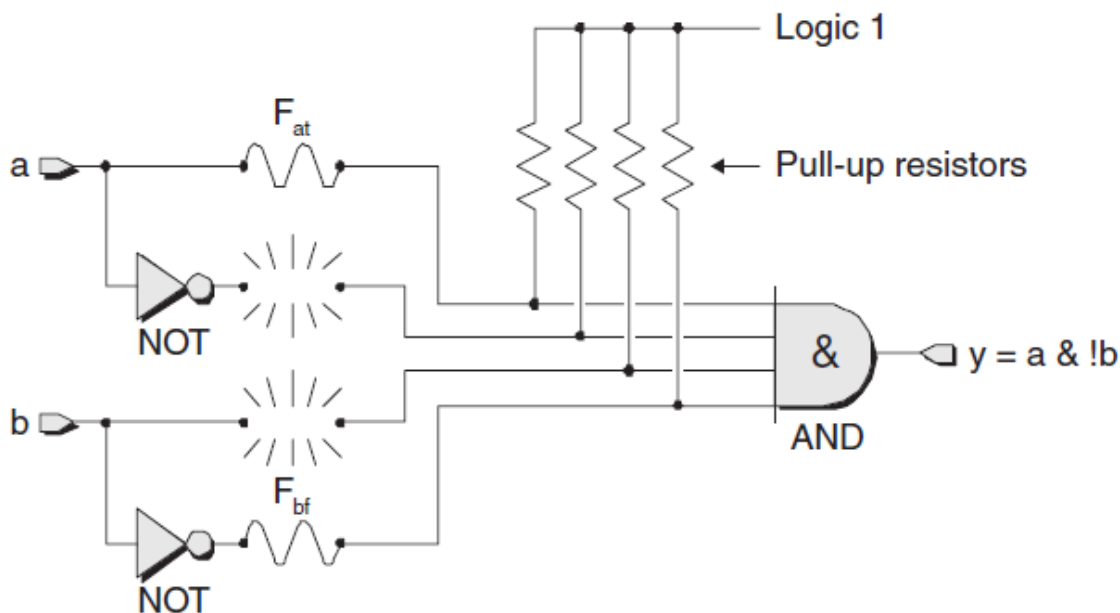
**Figure 2-1. A simple programmable function.**

## Fusible link technologies

One of the first techniques that allowed users to program their own devices was—and still is—known as *fusible-link technology*. In this case, the device is manufactured with all of the links in place, where each link is referred to as a *fuse* (Figure 2-2).



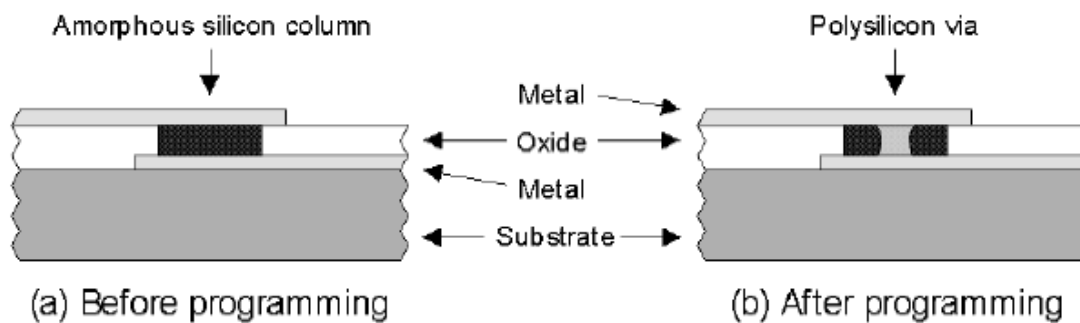
**Figure 2-2. Augmenting the device with unprogrammed fusible links.**



**Figure 2-3. Programmed fusible links.**

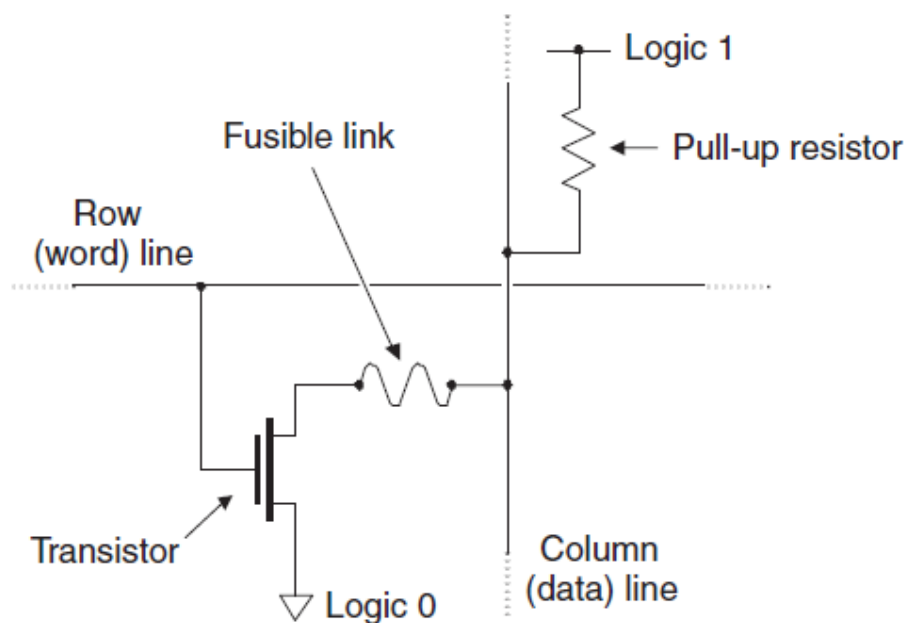
## Antifuse technologies

As a diametric alternative to fusible-link technologies, we have their antifuse counterparts, in which each configurable path has an associated link called an *antifuse*. In its unprogrammed state, an antifuse has such a high resistance that it may be considered an open circuit (a break in the wire), as illustrated in Figure 2-4.



**Figure 2-6. Growing an antifuse.**

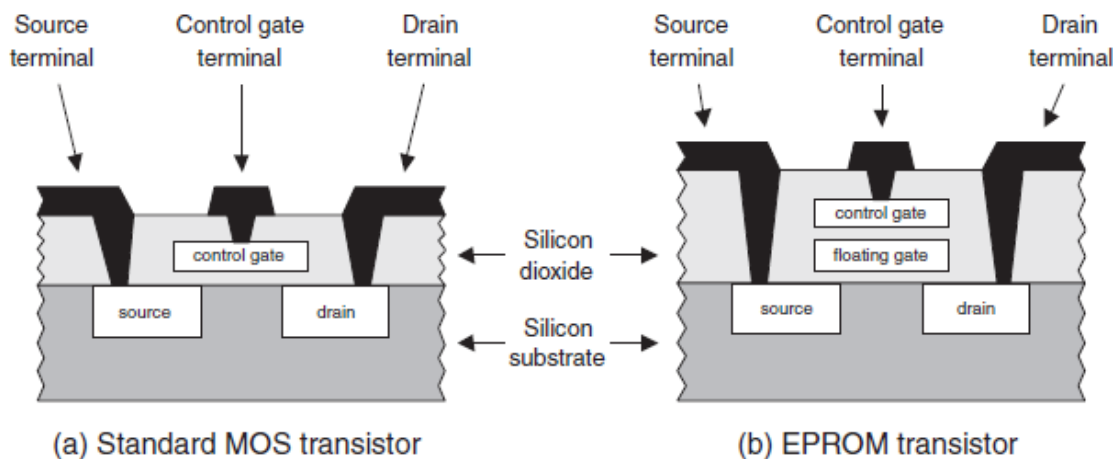
## PROMs



**Figure 2-8. A transistor-and-fusible-link-based PROM cell.**

## EPROM-based technologies

As was previously noted, devices based on fusible-link or antifuse technologies can only be programmed a single time—once you’ve blown (or grown) a fuse, it’s too late to change your mind. (That’s why, for that matter, it’s possible to increment

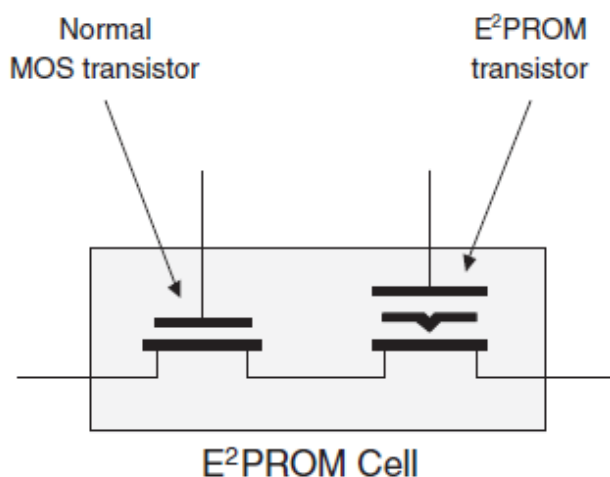


**Figure 2-9. Standard MOS versus EPROM transistors.**

## EEPROM-based technologies

The next rung up the technology ladder appeared in the form of *electrically erasable programmable read-only memories* (EEPROMs or  $E^2$ PROMs). An  $E^2$ PROM cell is approximately 2.5 times larger than an equivalent EPROM cell because it

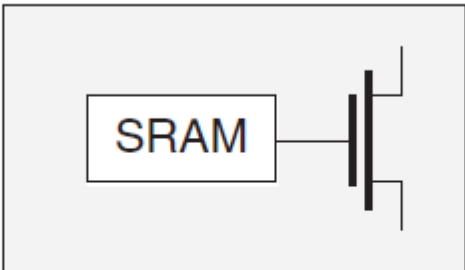
comprises two transistors and the space between them (Figure 2-11).



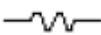




**Figure 2-11. An  $E^2$ PROM—cell.**

# SRAM-based technologies

There are two main versions of semiconductor RAM devices: *dynamic RAM (DRAM)* and *static RAM (SRAM)*.



**Figure 2-12. An SRAM-based programmable cell.**

Technology	Symbol	Predominantly associated with ...
Fusible-link		SPLDs
Antifuse		FPGAs
EPROM		SPLDs and CPLDs
E <sup>2</sup> PROM/ FLASH		SPLDs and CPLDs (some FPGAs)
SRAM		FPGAs (some CPLDs)

**Table 2-1. Summary of Programming Technologies**

# Programmable Logic Devices

- **Digital Electronic systems:**
  - Memory
  - Microprocessor
  - Logic Devices
    - Fixed Function Logic Devices.
    - Programmable Logic Devices.

# Programmable Logic Devices

- **Advantages Programmable Logic Devices:**
  - Less board space.
  - Easy to change with rewiring.
  - Faster.
  - Less cost.



# Programmable Logic Devices

- (PLDs) introduced in the mid 1970s.
- The idea: to construct logic circuits that were programmable.
- Microprocessors: can run a program but possess a fixed hardware.
- Programmability of PLDs was intended at the hardware level (reconfigured).

# Programmable Logic Devices

- **PAL (Programmable Array Logic) PLA (Programmable Logic Array)**
- used only logic gates (no flip-flops), combinational circuits.
- **registered PLDs:** one flip-flop at each output; simple sequential functions.
- In the beginning of the 1980s, additional logic circuitry was added to each PLD output.
- Macrocell, contained (besides the flip-flop) logic gates and multiplexers.

# Programmable Logic Devices

- Moreover, the cell itself was programmable, allowing several modes of operation.
- provided a 'return' (feedback) signal from the output of the circuit to the programmable array, which gave the PLD greater flexibility.
- This new PLD structure was called **generic PAL (GAL)**.
- **PALCE (PAL CMOS Electrically erasable/programmable)**.

# Programmable Logic Devices

- (**PAL, PLA, registered PLD, and GAL/PALCE**) are now collectively referred to as **SPLDs (Simple PLDs)**.
- Today **GAL/PALCE** device is the only still manufactured in a standalone package.

# Programmable Logic Devices

- several **GAL** devices were fabricated on the same chip.
- This approach known as **CPLD (Complex PLD)**.
- **CPLDs** are currently very popular due to:
  - their high density,
  - high performance,
  - and low cost
- (CPLDs under a dollar can be found).

# Programmable Logic Devices

- In the mid 1980s: **FPGAs (Field Programmable Gate Arrays)** were introduced.
- FPGAs differ from **CPLDs** in architecture, technology, built-in features, and cost.
- implementation of large size, high-performance circuits.

# Programmable Logic Devices

PLDs

Simple PLD (SPLD)

Complex  
PLD  
(CPLD)

FPGA

PAL

PLA

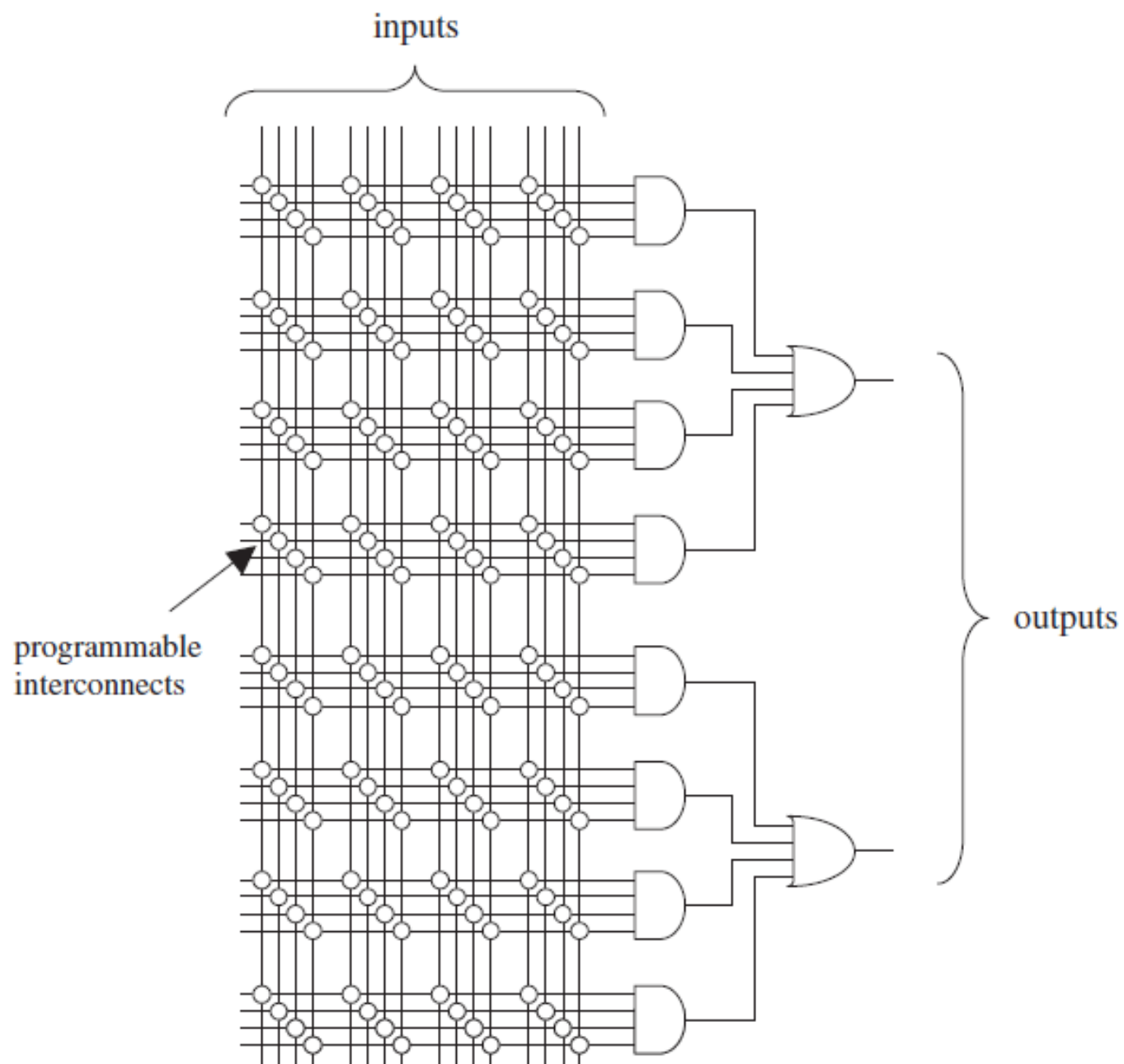
Registered  
PAL/PLA

GAL

# SPLDs (Simple PLDs): PAL Devices

- Introduced by Monolithic Memories in the mid 1970s.
- programmable array of **AND gates**, followed by a fixed array of **OR gates**.
- based on the fact that any combinational function can be represented by a Sum-of-Products (**SOP**).





**Figure A1**  
Illustration of PAL architecture.

# SPLDs (Simple PLDs): PAL Devices

- The main limitation: allowed only the implementation of combinational functions.
- registered PALs launched end of the 1970s.
- These included a flip-flop at each output (after the OR gates in figure A1), allowing the implementation of sequential functions.

# SPLDs (Simple PLDs): PAL Devices **PAL16L8**

- An example : **PAL16L8**
- **16** inputs and **8** outputs
- only **18 I/O** available, because it was a 20-pin DIP package
- 10 IN pins, 2 OUT pins, 6 IN/OUT pins, plus VCC and GND.



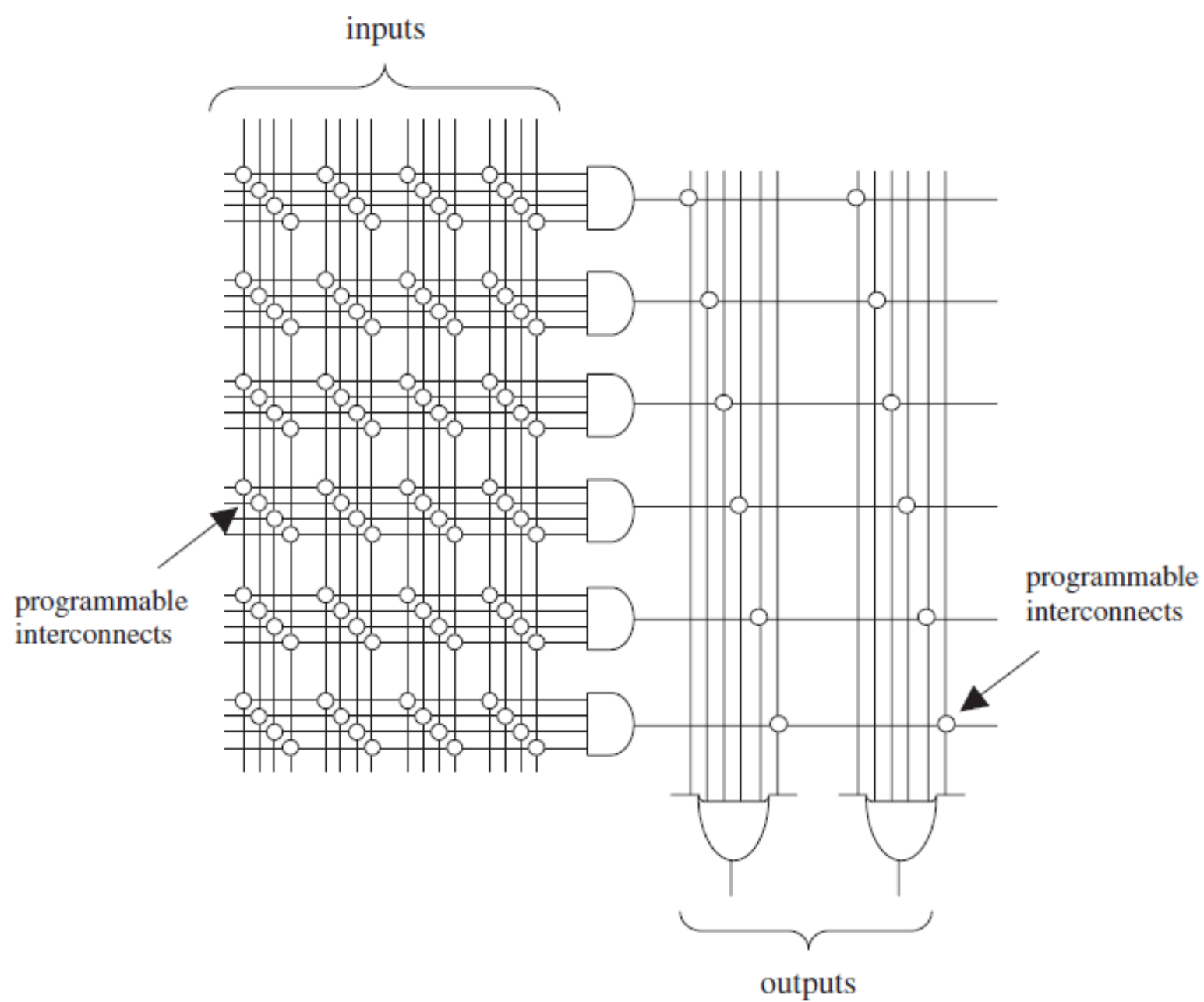
# SPLDs (Simple PLDs): PAL Devices **PAL16L8**

- Its registered counterpart was the 16R8 chip (where R stands for Registered).
- fabrication technology :bipolar.
- 5 V supply.
- current consumption : 200 mA.
- maximum frequency : 100 MHz.
- programmable cells: of PROM (fuse links) or EPROM (20min UV erase time) type.



# SPLDs (Simple PLDs): PLA Devices

- programmable array of **AND gates**, followed by a programmable array of **OR gates**.
- Advantage: was greater flexibility than PAL.
- 
- However, higher time constants at the internal nodes lowered the circuit speed.



**Figure A2**  
Illustration of PLA architecture.

# SPLDs (Simple PLDs): PLA Devices

- example :Signetics PLS161 device.
- 12 inputs and 8 outputs,
- A total of **48** 12-input AND gates, followed by a total of **8** 48-input OR gates.
- At the outputs, additional programmable XOR gates were also available.

# GAL Devices

- **Generic PAL** 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.



# GAL Devices

- **GAL** is the only **SPLD** (Simple PLD) still manufactured in a standalone package.
- serves as the basic building block in the construction of most **CPLDs** (there are exceptions, however, like the CoolRunner **CPLD**, which employs **PLAs** instead).



# Complex PLD (CPLD)

- several PLDs (in general of **GAL** type) fabricated on a single chip.
- With programmable switch matrix.
- JTAG support
- interface to other logic standards (1.8 V, 2.5 V, 5 V, etc.).
- Altera, Xilinx, Lattice, Atmel, Cypress, etc.

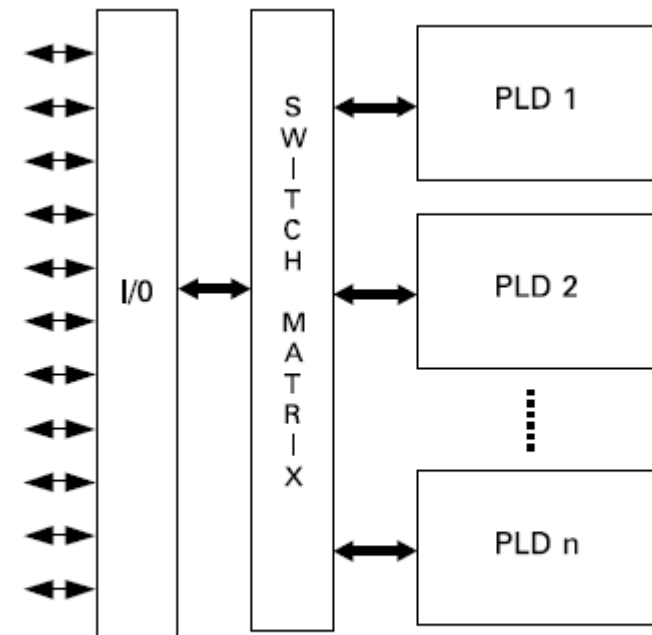


Figure A4  
CPLD architecture.

# Complex PLD (CPLD)

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

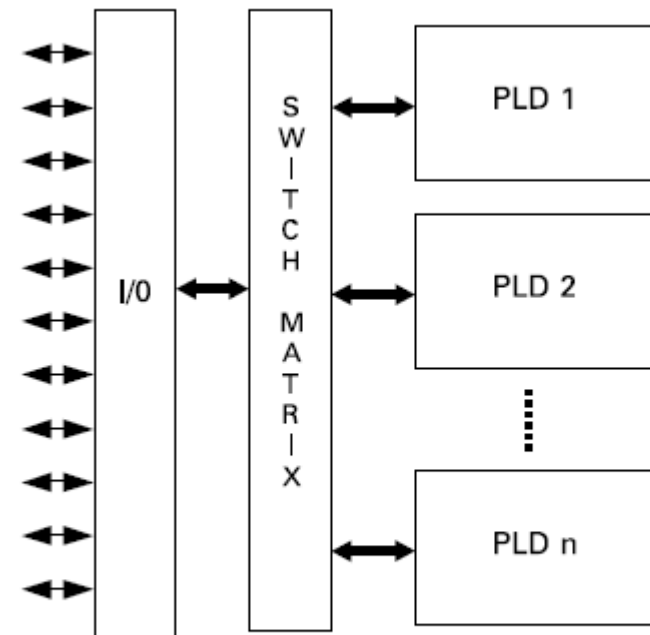
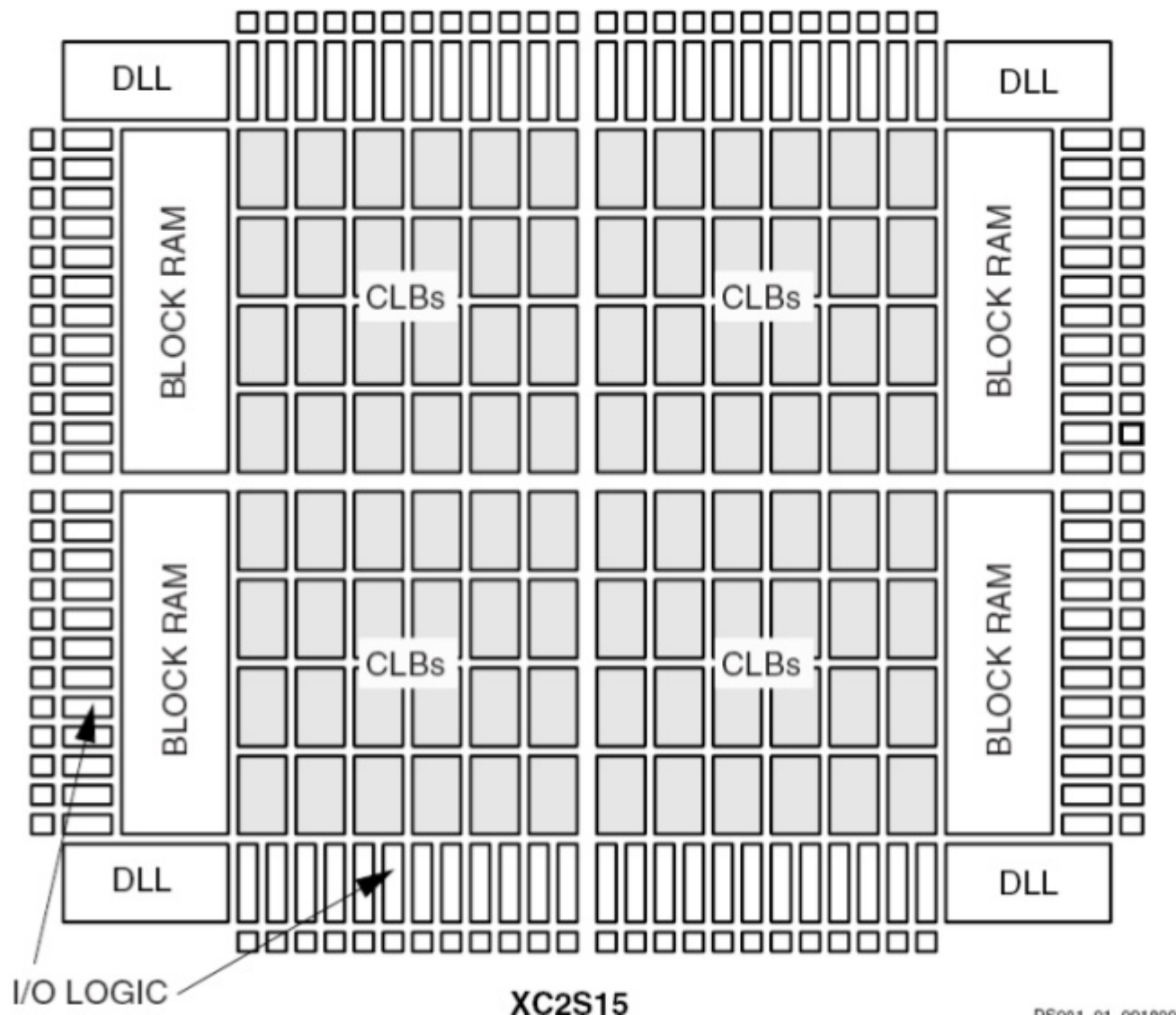


Figure A4  
CPLD architecture.

# Field Programmable Gate Array (**FPGA**)

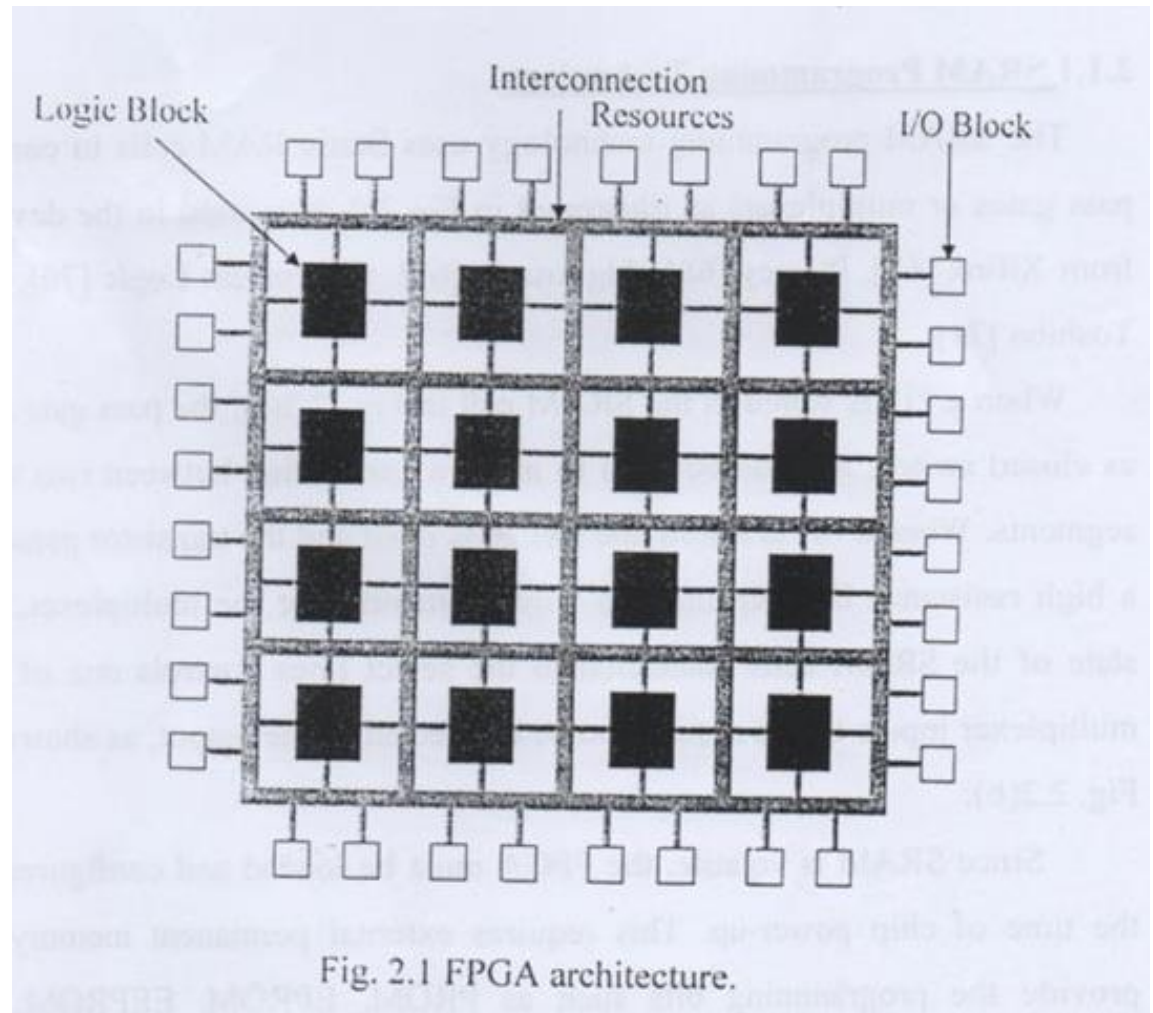
- introduced by Xilinx in the mid 1980s.
- differ from CPLDs in architecture, storage technology, number of built-in features, and cost.
- Aimed at the implementation of high performance, large-size circuits.

# Spartan-II FPGA



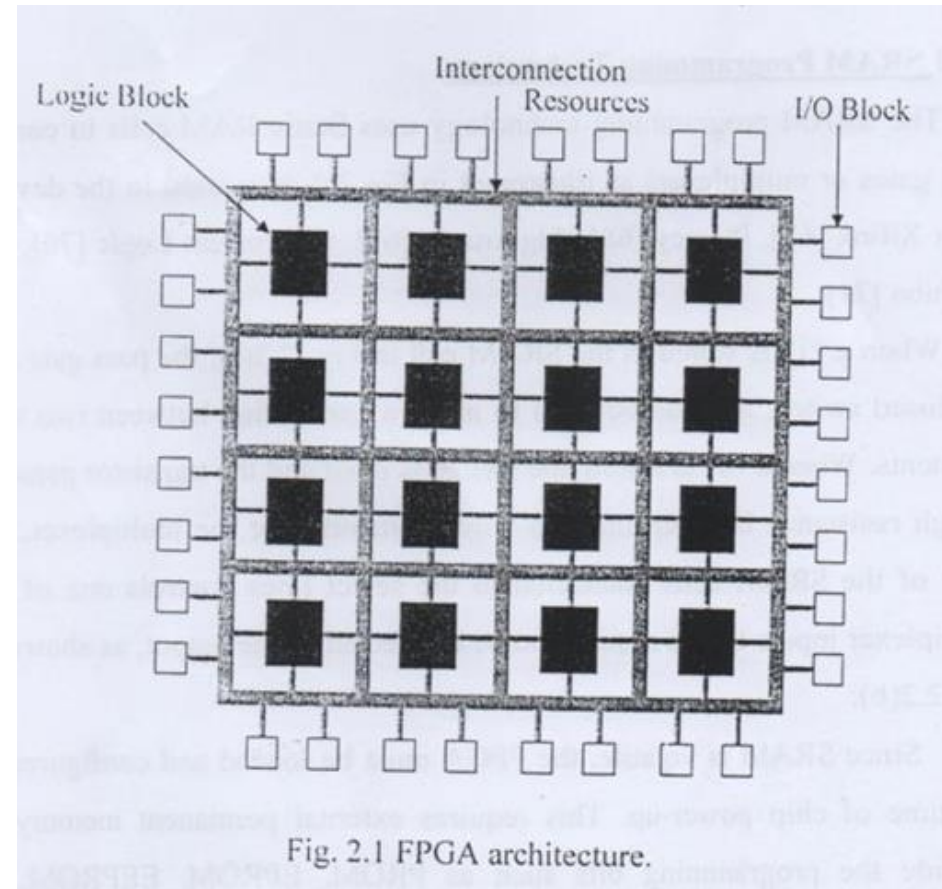
# Field Programmable Gate Array (**FPGA**)

- It consists of a matrix of **CLBs** (Configurable Logic Blocks),
- interconnected by an array of **switch matrices**.
- Array Input/output Blocks.



# Field Programmable Gate Array (FPGA)

- First, instead PAL (in SPLDs), its operation is normally based on a LUT (lookup table).
- The number of flip-flops is much more abundant (more sophisticated sequential Circuits).
- JTAG support.
- interface to diverse logic levels.
- SRAM memory
- clock multiplication (PLL or DLL).
- PCI interface, etc.
- Some chips: multipliers, DSPs, and microprocessors.





# Field Programmable Gate Array (**FPGA**)

- Storage of the interconnects:
  - **CPLDs** are non-volatile (antifuse, EEPROM, Flash, etc.).
  - most **FPGAs** use SRAM, and are therefore volatile.
    - saves space and lowers the cost but requires an external ROM.
  - non-volatile FPGAs (with antifuse), which might be advantageous when reprogramming is not necessary.

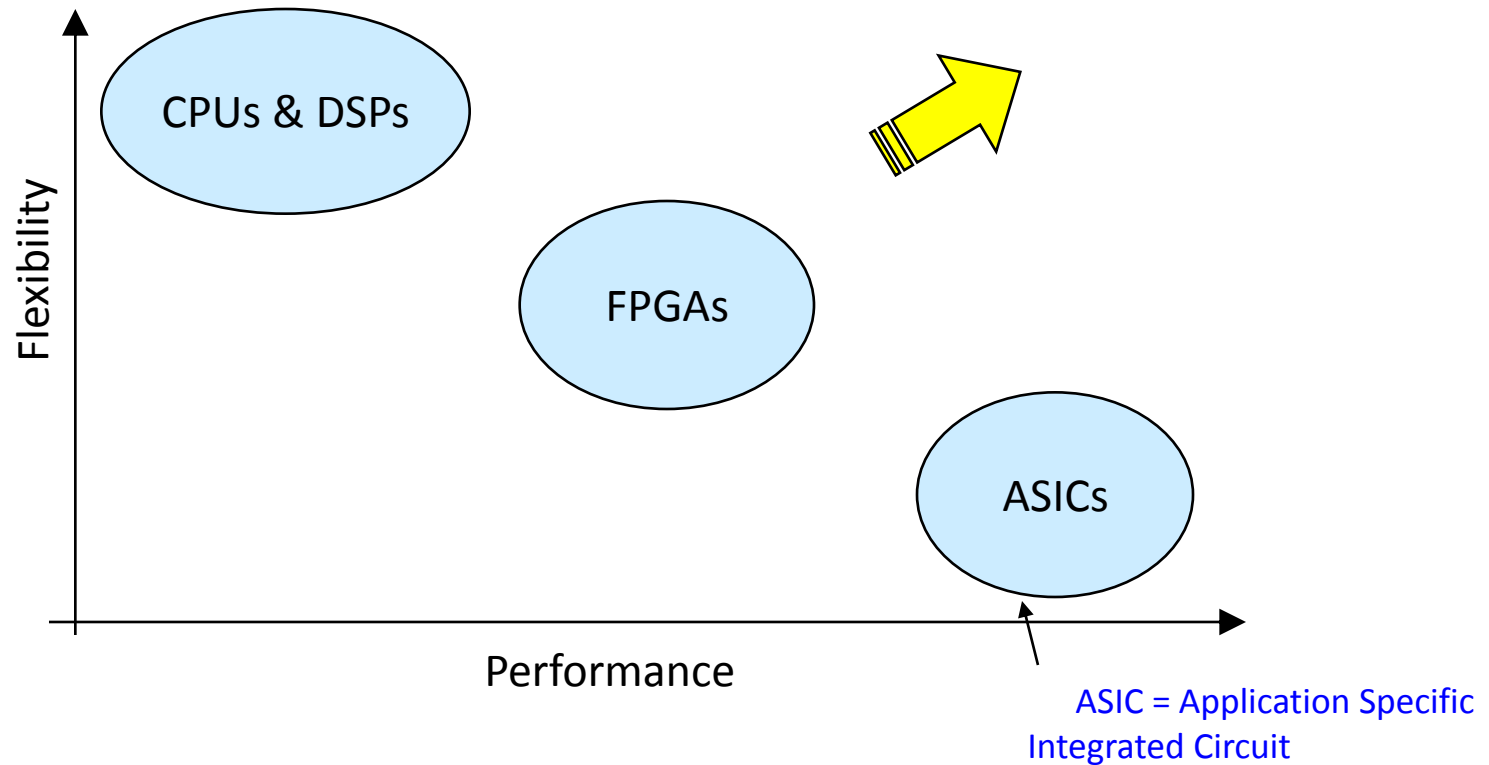
# Field Programmable Gate Array (**FPGA**)

- Several companies manufacture FPGAs, like Xilinx, Actel, Altera, QuickLogic, Atmel, etc.
- Interconnects:
  - all Xilinx FPGAs use **SRAM**, **reprogrammable**, but **volatile** (thus requiring external ROM).
  - Actel FPGAs use **antifuse** , **non-volatile** (they), but are **non-reprogrammable** (except one family, which uses Flash memory).

# Field Programmable Gate Array (**FPGA**)

- the actual **application** will dictate which chip architecture is most **appropriate**.
- Applications:
  - Aerospace and Defense.
  - Medical Electronics.
  - Wired Communications.
  - Wireless Communications.
  - High performance computing.

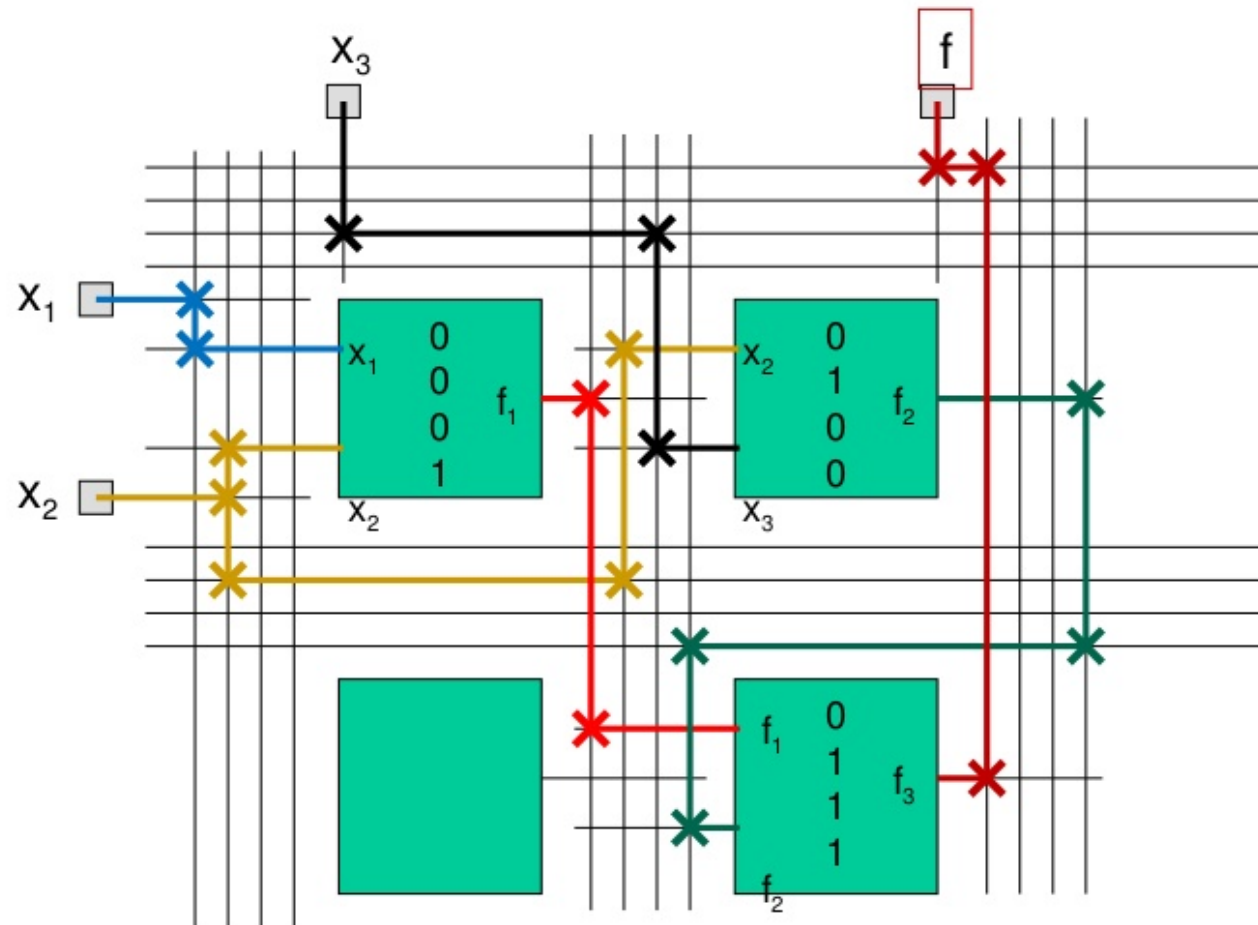
# Performance vs. Flexibility



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

# FPGAs

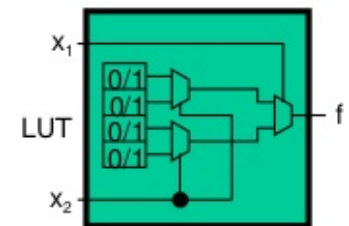
- An example of programming an FPGA



$$f_1 = x_1 x_2$$

$$f_2 = \overline{x_2} x_3$$

$$f = x_1 x_2 + \overline{x_2} x_3$$



# Circuit Design with VHDL

Textbook: Volnei A. Pedroni

Submitted By: Hussein 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.

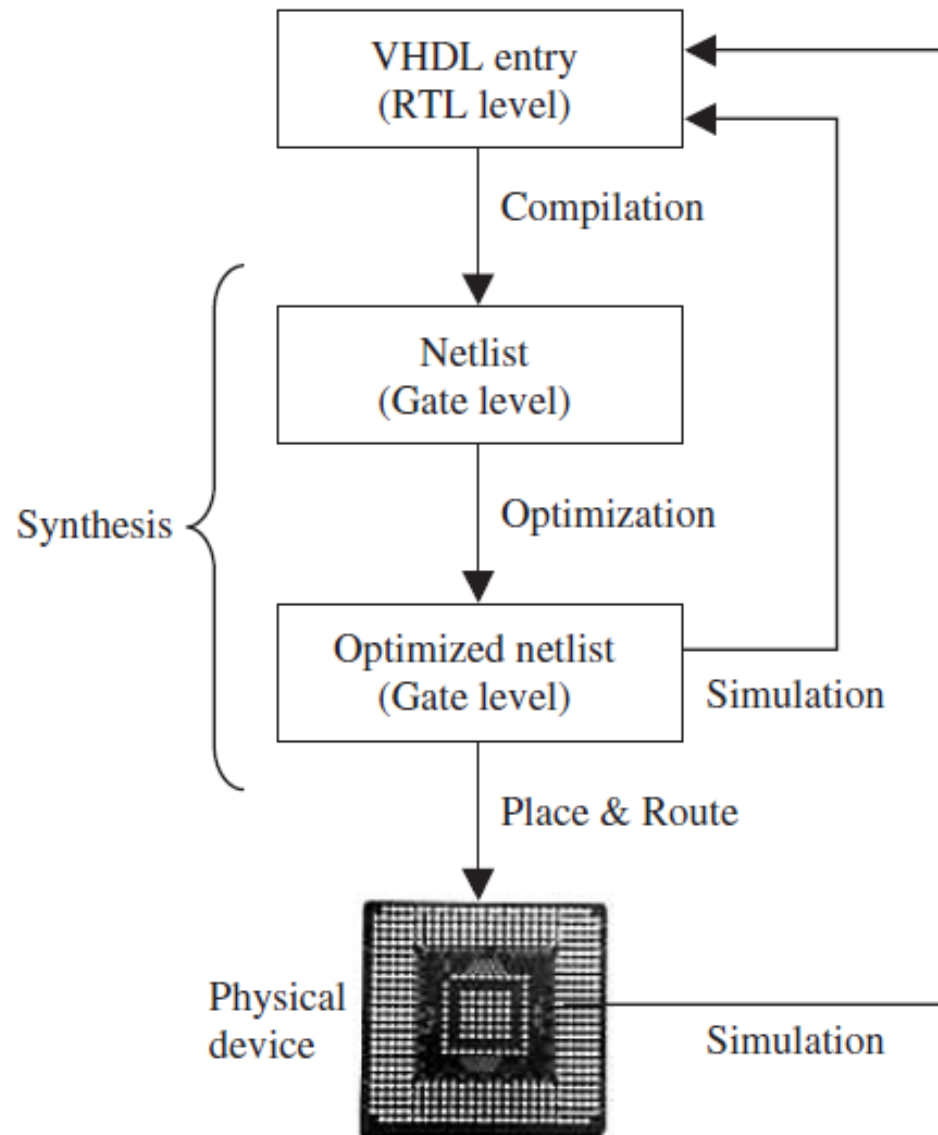
# VHDL> Introduction

---

- 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).

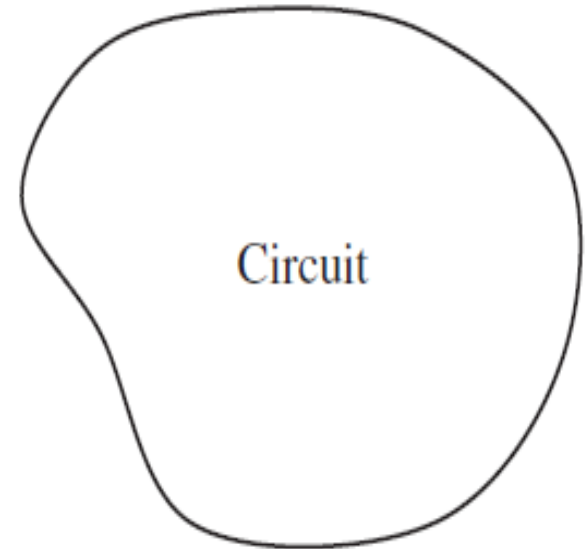


# VHDL> Design Flow

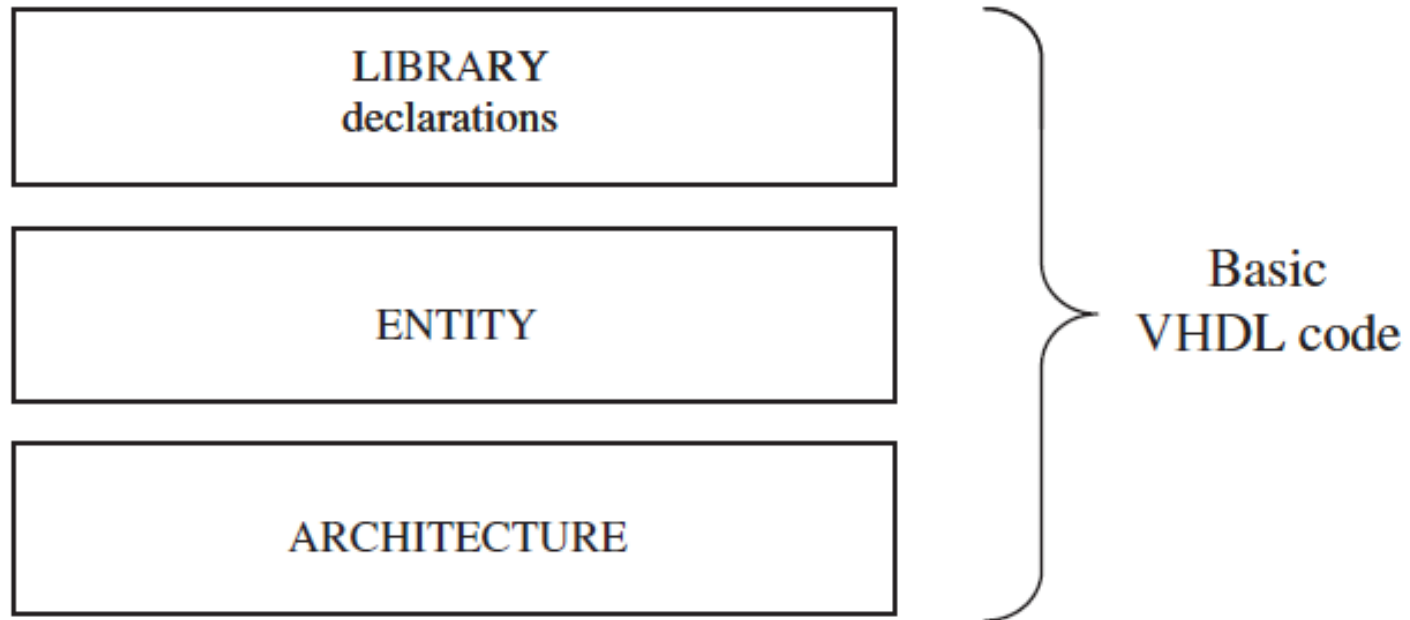


# VHDL> Design Flow

```
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> Code Structure



**Figure 2.1**  
Fundamental sections of a basic VHDL code.

# VHDL> Code Structure

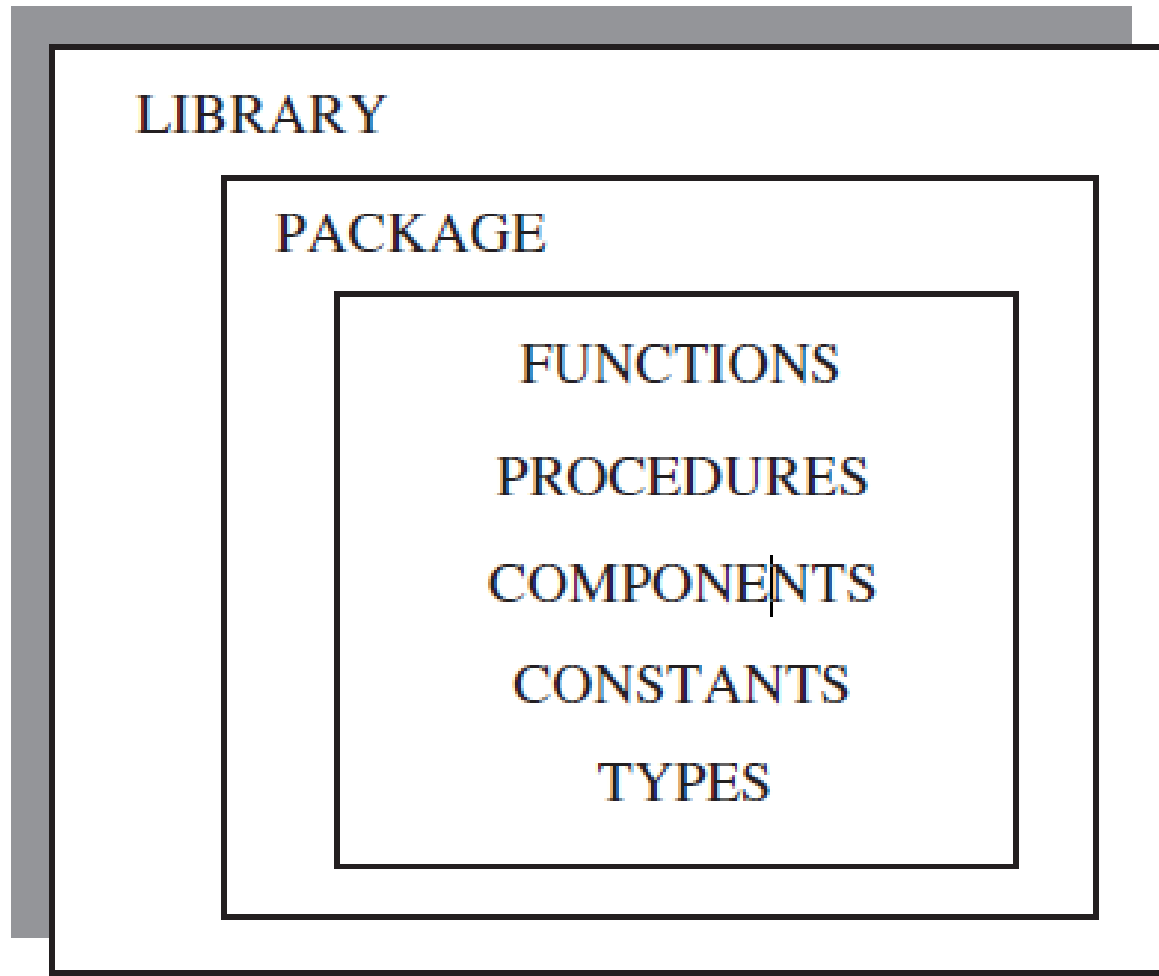
LIBRARY declarations:

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;
```

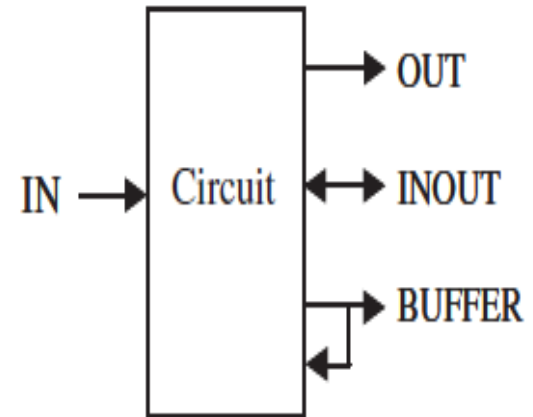
```
ENTITY nand_gate IS
PORT (a, b : IN BIT;
x : OUT BIT);
END nand_gate;
```

```
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)



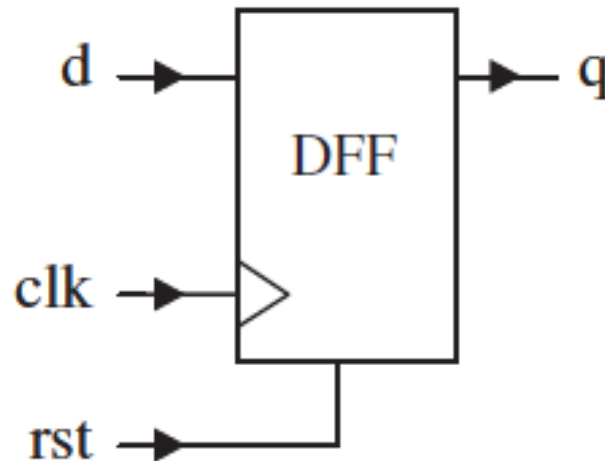
# VHDL> Introductory Examples

- **Process**
- 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

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



# VHDL> Introductory Examples

- DFF

```
entity dff is
  port(
    data :in std_logic; -- Data input
    clk  :in std_logic; -- Clock input
    q    :out std_logic -- Q output
  );
end entity;
```

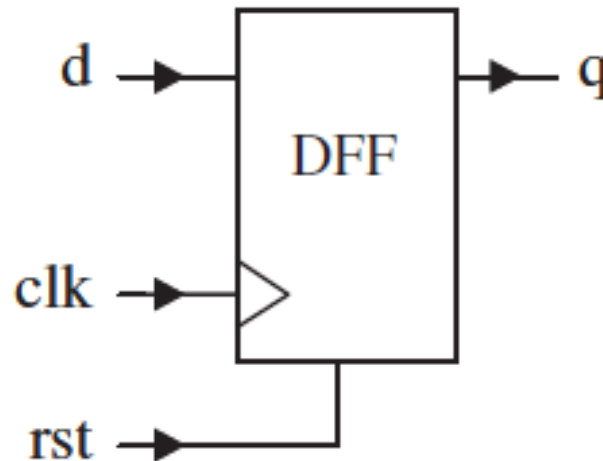
```
architecture rtl of dff is
begin
  process (clk) begin
    if (rising_edge(clk)) then

      q <= data;

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

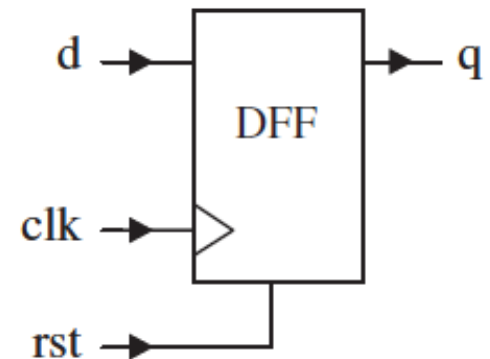
# VHDL> Introductory Examples

- **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

```
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> J-k FF

```
entity JK_FF is
  port( J,K: in std_logic;
        Reset: in std_logic;
        Clock: in std_logic;
        Output: out std_logic);
end JK_FF;
```

```
architecture Behavioral of JK_FF is
  signal temp: std_logic;
begin
  process (Clock)
  begin
    if rising_edge(Clock) then
      if Reset='1' then
        temp <= '0';
```

```
      elsif (J='0' and K='0') then
        temp <= temp;
      elsif (J='0' and K='1') then
        temp <= '0';
      elsif (J='1' and K='0') then
        temp <= '1';
      elsif (J='1' and K='1') then
        temp <= not(temp);
      end if;
    end if;
  end process;
  Output <= temp;
end Behavioral;
```

# VHDL> T FF

```
architecture rtl of tff_async_reset is
    signal t :std_logic;
begin
    process (clk, reset) begin
        if (reset = '0') then
            t <= '0';
        elsif (rising_edge(clk)) then
            t <= not t;
        end if;
    end process;
    q <= t;
end architecture;
```

# Circuit Design with VHDL

2

Hussein Aideen



# 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 DOWNT0 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 DOWNT0 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 DOWNT0 0);
SIGNAL c: STD_LOGIC;
SIGNAL d: STD_LOGIC_VECTOR(7 DOWNT0 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)

```
1  library ieee;
2      use ieee.std_logic_1164.all;
3
4  entity encoder_using_if is
5      port (
6          enable      :in  std_logic;           -- Enable for the encoder
7          encoder_in  :in  std_logic_vector (7 downto 0); -- 16-bit Input
8          binary_out  :out std_logic_vector (2 downto 0)  -- 4 bit binary Output
9      );
10 end entity;
11
12 architecture behavior of encoder_using_if is
13 begin
14     process (enable, encoder_in) begin
15         binary_out <= "00";
16         if (enable = '1') then
17             if (encoder_in = X"02") then binary_out <= "001"; end if;
18             if (encoder_in = X"04") then binary_out <= "010"; end if;
19             if (encoder_in = X"08") then binary_out <= "011"; end if;
20             if (encoder_in = X"10") then binary_out <= "100"; end if;
21             if (encoder_in = X"20") then binary_out <= "101"; end if;
22             if (encoder_in = X"40") then binary_out <= "110"; end if;
23             if (encoder_in = X"80") then binary_out <= "111"; end if;
24
25         end if;
26     end process;
27 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 DOWNT0 0);
SIGNAL b: IN STD_LOGIC_VECTOR (7 DOWNT0 0);
SIGNAL x: OUT STD_LOGIC_VECTOR (7 DOWNT0 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 DOWNT0 0);
SIGNAL b: IN STD_LOGIC_VECTOR (7 DOWNT0 0);
SIGNAL x: OUT STD_LOGIC_VECTOR (7 DOWNT0 0);
...
v <= a + b;      -- legal (arithmetic operation OK), unsigned
w <= a AND b;    -- legal (logical operation OK)
```

# Circuit Design with VHDL 3

Textbook: Volnei A. Pedroni

Submitted By: Hussein Aideen



# 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
<code>conv_integer(p)</code>	INTEGER, UNSIGNED, SIGNED, or STD_ULOGIC	INTEGER
<code>conv_unsigned(p, b)</code>	INTEGER, UNSIGNED, SIGNED, or STD_ULOGIC	UNSIGNED * Where b is number of bits.
<code>conv_signed(p, b):</code>	INTEGER, UNSIGNED, SIGNED, or STD_ULOGIC	SIGNED
<code>conv_std_logic_vector(p, b)</code>	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
<code>unsigned(p)</code>	STD_LOGIC_VECTOR	UNSIGNED
<code>signed(p):</code>	STD_LOGIC_VECTOR	SIGNED

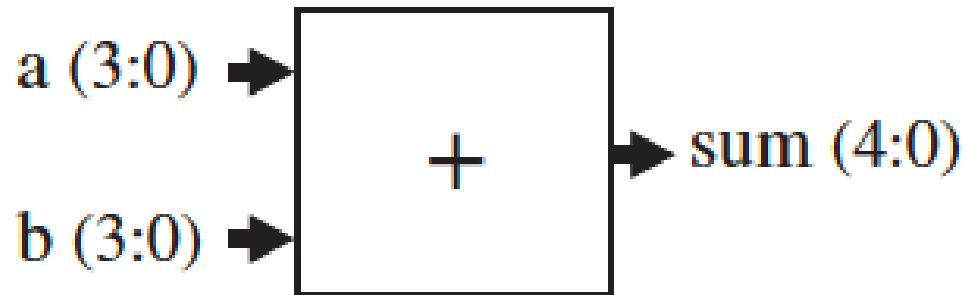
# VHDL> Data Conversion

- 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 DOWNT0 0);  
      sum : OUT SIGNED (4 DOWNT0 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 DOWNT0 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 1..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 DOWNT0 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 DOWNT0 0);
```

# VHDL>Static and non-static data

**Table 7.1**

Comparison between SIGNAL and VARIABLE.

	SIGNAL	VARIABLE
Assignment	<code>&lt;=</code>	<code>:=</code>
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
<code>&lt;=</code>	SIGNAL.
<code>:=</code>	VARIABLE, CONSTANT, GENERIC, initial values.
<code>=&gt;</code>	vector elements or with OTHERS.

# VHDL> Operators

- Assignment operators

```
SIGNAL x : STD_LOGIC;  
VARIABLE y : STD_LOGIC_VECTOR(3 DOWNT0 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'
```



جامعة نينوى  
كلية هندسة الإلكترونيات

# Circuit Design with VHDL 4

Textbook: Volnei A. Pedroni

Submitted By: Hussein Aideen

# 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 temporary 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 DOWNT0 0);
```

Then:

D7	D6	D5	D4	D3	D2	D1	D0
----	----	----	----	----	----	----	----

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 ...
```

D0	D1	D2	D3	D4	D5	D6	D7
----	----	----	----	----	----	----	----

# 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;
```



جامعة نينوى  
كلية هندسة الإلكترونيات

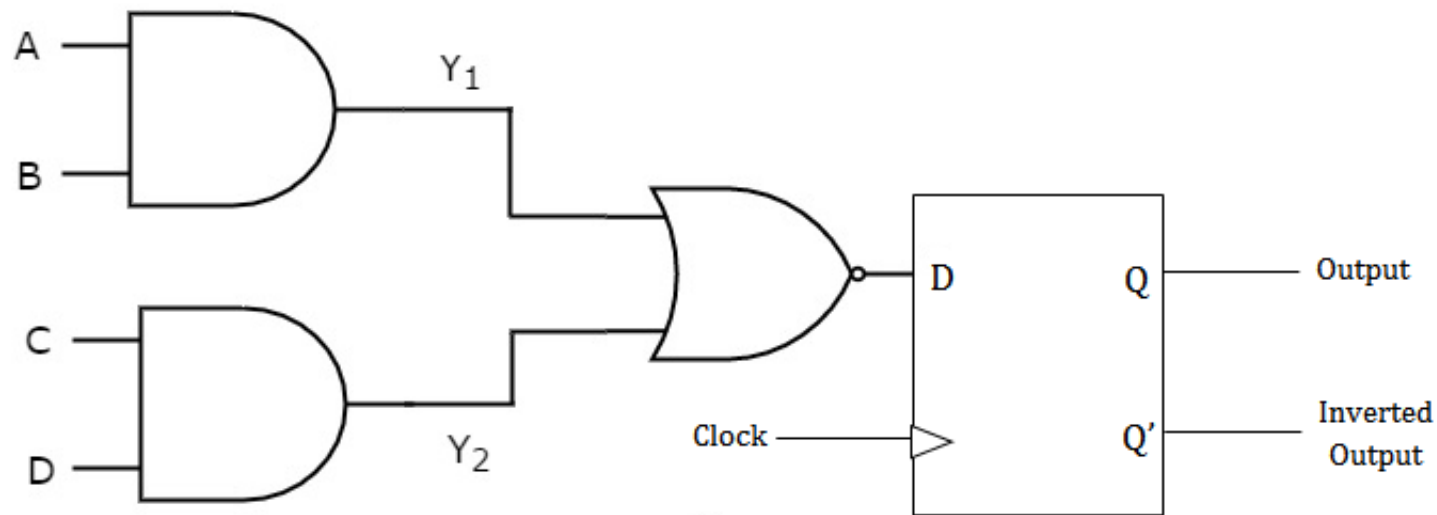
# Circuit Design with VHDL 5

Textbook: Volnei A. Pedroni

Submitted By: Hussein Aideen

# VHDL> Concurrent & sequential Code

- Example: Write VHDL code to implement the following circuit.



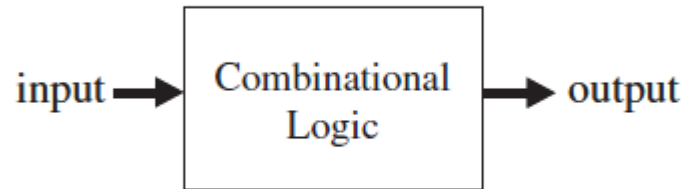
# VHDL> Concurrent & sequential Code

---

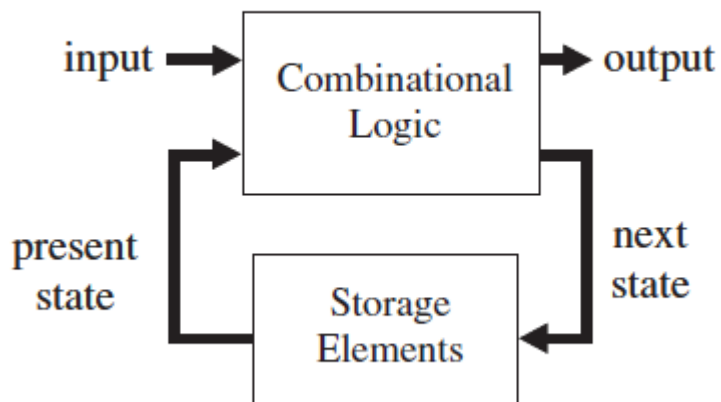
- Concurrent Code:
  - WHEN,
  - GENERATE,
  - Assignments using only operators (AND, NOT, +, \*, sll, etc.),
  - A special kind of assignment, called BLOCK.
- Sequential Code:
  - PROCESSES, FUNCTIONS, PROCEDURES.
  - IF, WAIT, CASE, and LOOP.
  - VARIABLES.

# 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

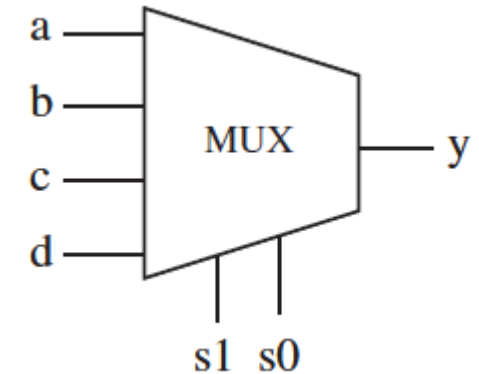
- Operators:

**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

- Multiplexer #1



```
1  -----
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -----
5
6  ENTITY mux IS
7  PORT ( a, b, c, d, s0, s1: IN STD_LOGIC;
8        y: OUT STD_LOGIC);
9  END mux;
10 -----
11
12 ARCHITECTURE pure_logic OF mux IS
13 BEGIN
14   y <= (a AND NOT s1 AND NOT s0) OR
15        (b AND NOT s1 AND s0) OR
16        (c AND s1 AND NOT s0) OR
17        (d AND s1 AND s0);
18 END pure_logic;
19 -----
```

# 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:

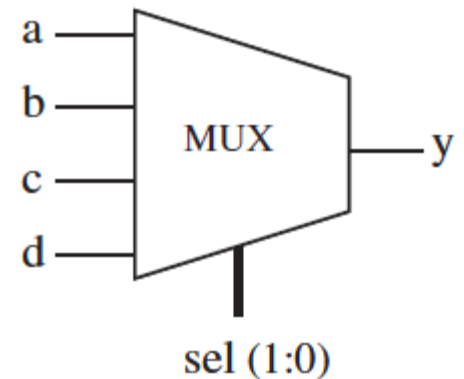
```
----- With WHEN/ELSE -----  
outp <= "000" WHEN (inp='0' OR reset='1') ELSE  
        "001" WHEN ctl='1' ELSE  
        "010";
```

```
---- 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 DOWNT0 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;
```

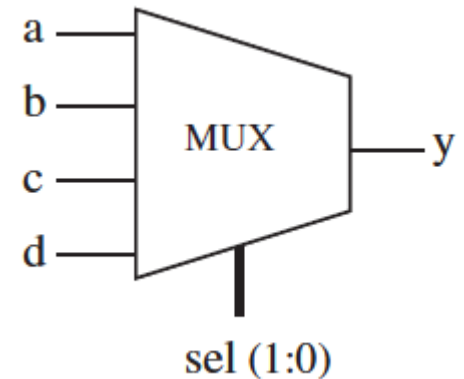


Simple  
when

# 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 DOWNT0 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;
```

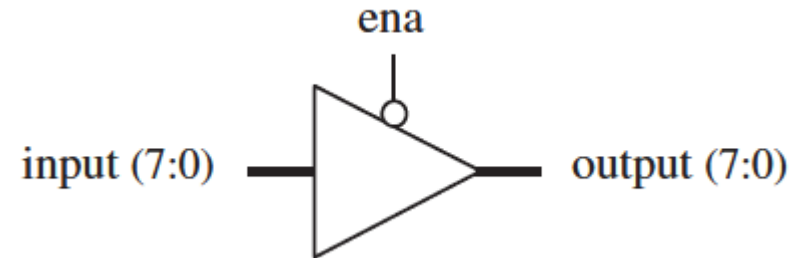


selected  
when



# 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 DOWNT0 0);
7             output: OUT STD_LOGIC_VECTOR (7 DOWNT0 0));
8  END tri_state;
9  -----
10 ARCHITECTURE tri_state OF tri_state IS
11 BEGIN
12     output <= input WHEN (ena='0') ELSE
13         (OTHERS => 'Z');
14 END tri_state;
```

# VHDL> Concurrent Code

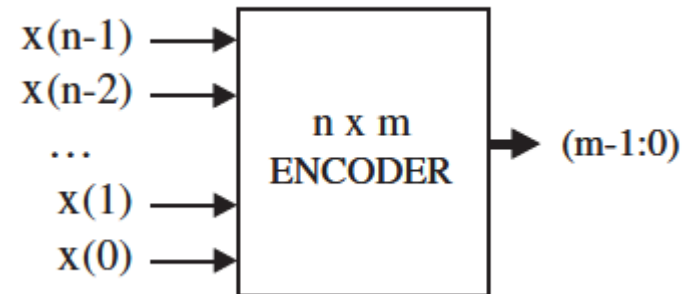
- Home Works: Encoder: page 73:

```
ENTITY encoder IS
    PORT ( x: IN STD_LOGIC_VECTOR (7 DOWNT0 0);
           y: OUT STD_LOGIC_VECTOR (2 DOWNT0 0));
END encoder;
```

-----

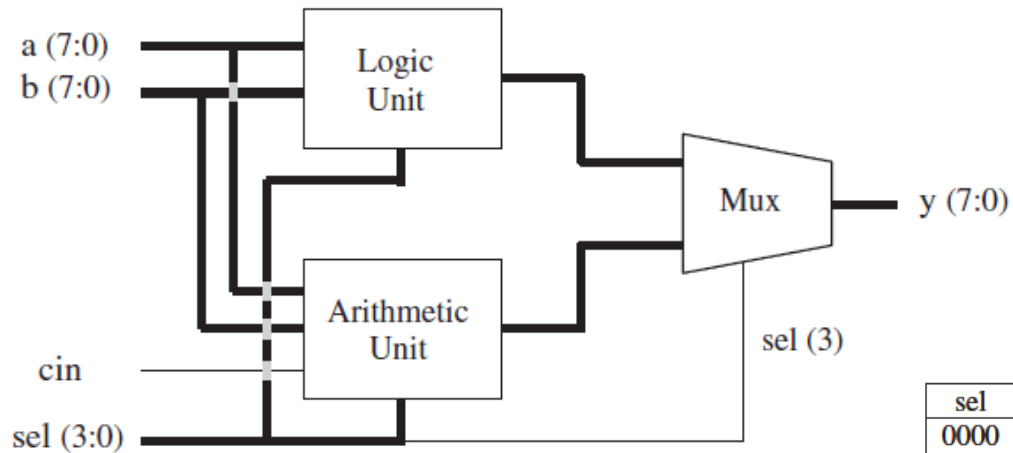
```
ARCHITECTURE encoder1 OF encoder IS
BEGIN
```

```
    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	<code>y &lt;= a</code>	Transfer a	Arithmetic
0001	<code>y &lt;= a+1</code>	Increment a	
0010	<code>y &lt;= a-1</code>	Decrement a	
0011	<code>y &lt;= b</code>	Transfer b	
0100	<code>y &lt;= b+1</code>	Increment b	
0101	<code>y &lt;= b-1</code>	Decrement b	
0110	<code>y &lt;= a+b</code>	Add a and b	
0111	<code>y &lt;= a+b+cin</code>	Add a and b with carry	
1000	<code>y &lt;= NOT a</code>	Complement a	Logic
1001	<code>y &lt;= NOT b</code>	Complement b	
1010	<code>y &lt;= a AND b</code>	AND	
1011	<code>y &lt;= a OR b</code>	OR	
1100	<code>y &lt;= a NAND b</code>	NAND	
1101	<code>y &lt;= a NOR b</code>	NOR	
1110	<code>y &lt;= a XOR b</code>	XOR	
1111	<code>y &lt;= a XNOR b</code>	XNOR	



جامعة نينوى  
كلية هندسة الإلكترونيات

# Circuit Design with VHDL 6

Textbook: Volnei A. Pedroni

Submitted By: Hussein Aideen

# VHDL> Concurrent Code

---

- **COMPONENT:**
- A COMPONENT is simply a piece of conventional code (that is, LIBRARY declarations ENTITY ARCHITECTURE). However, by declaring such code as being a COMPONENT, it can then be used within another circuit, thus allowing the construction of hierarchical designs.
- A COMPONENT is also another way of partitioning a code and providing code sharing and code reuse.

# VHDL> Concurrent Code

- **COMPONENT:**

COMPONENT declaration:

```
COMPONENT component_name IS
    PORT (
        port_name : signal_mode signal_type;
        port_name : signal_mode signal_type;
        ...);
END COMPONENT;
```

COMPONENT instantiation:

```
label: component_name PORT MAP (port_list);
```

# VHDL> Concurrent Code

- **COMPONENT:**

- Example: inverter as component

```
----- COMPONENT declaration: -----  
COMPONENT inverter IS  
    PORT (a: IN STD_LOGIC; b: OUT STD_LOGIC);  
END COMPONENT;  
  
----- COMPONENT instantiation: -----  
U1: inverter PORT MAP (x, y);
```

# VHDL> Concurrent Code

- **COMPONENT:**
- PORT MAP
- There are two ways to map the PORTS of a COMPONENT during its instantiation: **positional** mapping and **nominal** mapping. Let us consider the following example:

```
U1: inverter PORT MAP (x, y);
```

```
U1: inverter PORT MAP (x=>a, y=>b);
```

```
U2: my_circuit PORT MAP (x=>a, y=>b, w=>OPEN, z=>d);
```



# 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;
```

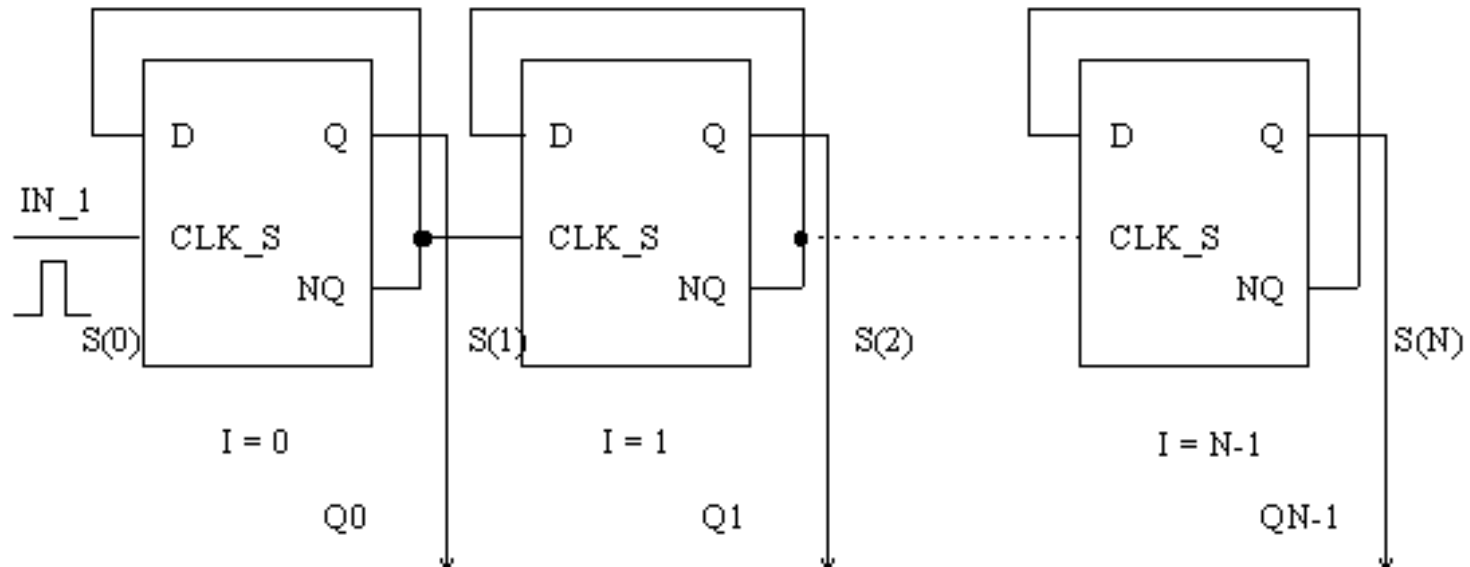
# VHDL> Concurrent Code

- Example:

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

# VHDL> Concurrent Code

- Example: N-bit counter



# VHDL> Concurrent Code

- Example:

```
entity COUNTER_BIN_N is
    generic (N : Integer := 4);
    port (Q : out Bit_Vector (0 to N-1);
          IN_1 : in Bit );
end entity COUNTER_BIN_N;

architecture BEH of COUNTER_BIN_N is
    component D_FF
        port(D, CLK_S : in BIT; Q, NQ : out BIT);
    end component D_FF;
    signal S : BIT_VECTOR(0 to N);

begin
    S(0) <= IN_1;
    G_1 : for I in 0 to N-1 generate
        D_Flip_Flop :
            D_FF port map
                (S(I+1), S(I), Q(I), S(I+1));
        end generate;
    end architecture BEH;
```

# 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

- Simple 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

- 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;
```



جامعة نينوى  
كلية هندسة الإلكترونيات

# Circuit Design with VHDL 7

Textbook: Volnei A. Pedroni

Submitted By: Hussein Aideen

# 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;
```

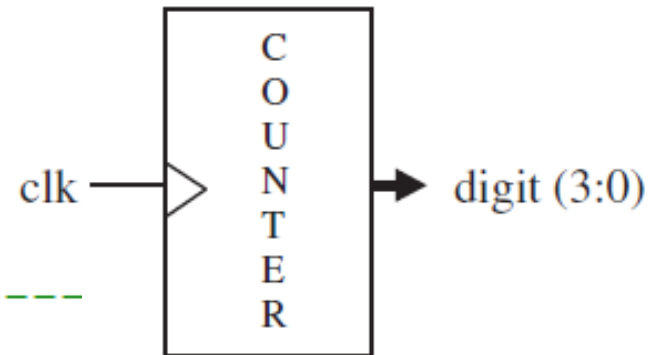
```
IF (x<y) THEN temp:="11111111";  
ELSIF (x=y AND w='0') THEN temp:="11110000";  
ELSE temp:=(OTHERS =>'0');  
end if
```



# VHDL> IF statement

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

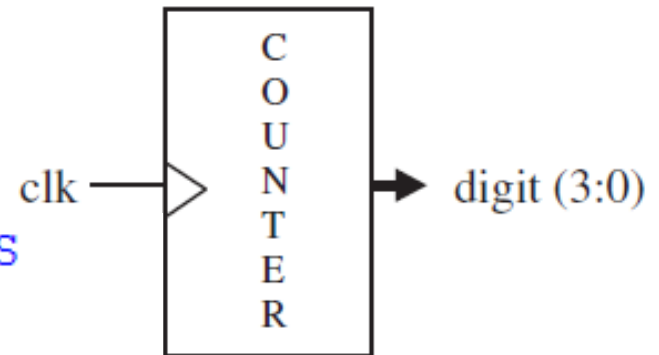
```
1  -----  
2  LIBRARY ieee;  
3  USE ieee.std_logic_1164.all;  
4  -----  
5  ENTITY counter IS  
6  PORT (clk : IN STD_LOGIC;  
7  digit : OUT INTEGER RANGE 0 TO 9);  
8  END counter;  
9  -----
```



# VHDL> IF statement

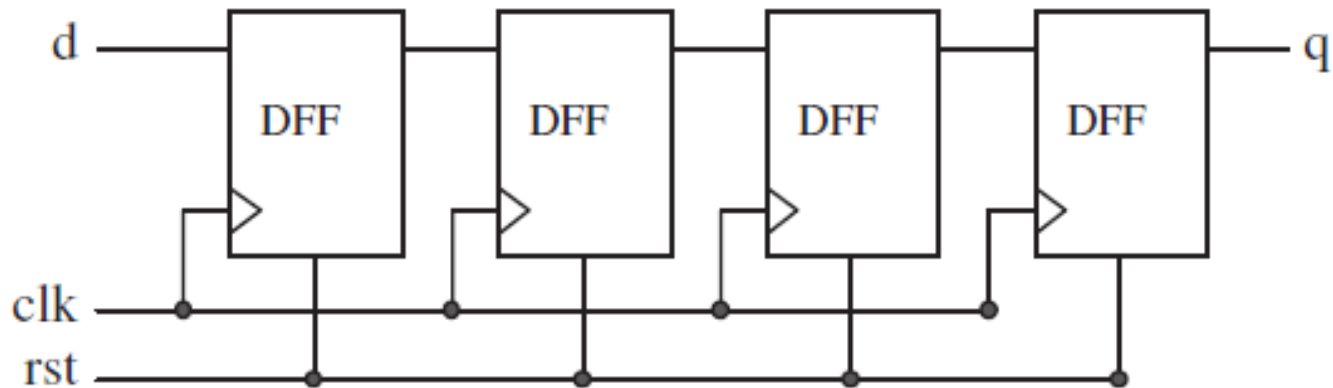
- One-digit Counter #1
  - 1-digit decimal counter ( $0 \rightarrow 9 \rightarrow 0$ ).

```
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 DOWNT0 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 DOWNT0 1);  
20  END IF;  
21  END PROCESS;  
22  q <= internal(0);  
23  END behavior;  
24  -----
```

# VHDL> WAIT statement

- WAIT statement: (inside Process)
- 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 test-benches). Example: WAIT FOR 5ns;

# VHDL> WAIT statement

---

- Home Works:
- DFF with Asynchronous Reset #2, Page 99.
- One-digit Counter #2, Page 99-100.



# 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;
```

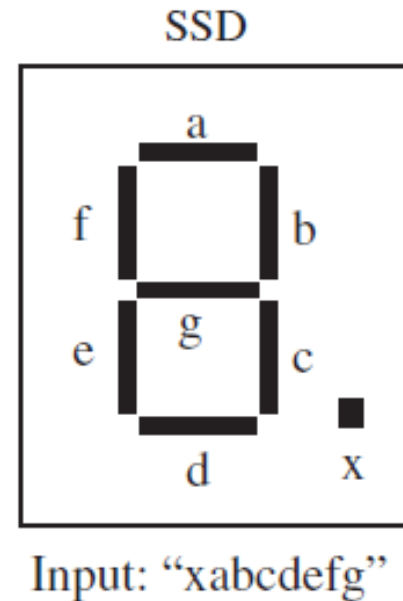
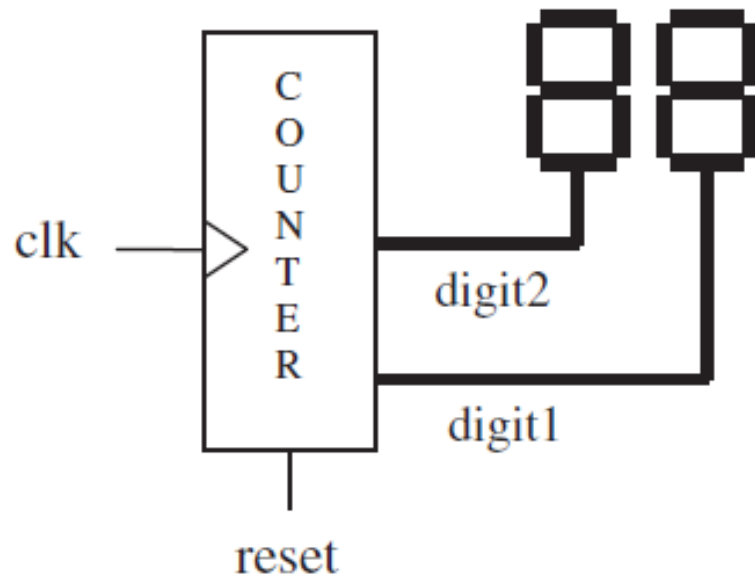
# VHDL> CASE statement

---

- 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;
```

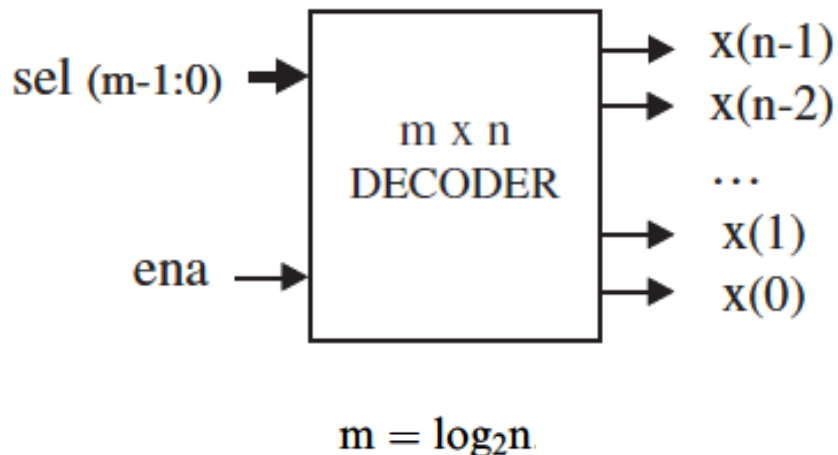
# VHDL> Two-digit Counter with SSD

```
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 -----
```

# VHDL> Examples

- Generic **Decoder**:

- If  $\text{ena} = '0'$ , then all bits of  $x$  should be high; otherwise, the output bit selected by  $\text{sel}$  should be low.



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;
```

# VHDL> Examples

- Gray code counter:

```
entity gray_counter is
  port (
    cout :out std_logic_vector (7 downto 0); -- Output of t
    enable :in std_logic;                    -- Enable counting
    clk :in std_logic;                      -- Input clock
    reset :in std_logic                     -- Input reset
  );
end entity;
```

```
architecture rtl of gray_counter is
  signal count :std_logic_vector (7 downto 0);
begin
  process (clk, reset) begin
    if (reset = '1') then
      count <= (others=>'0');
    elsif (rising_edge(clk)) then
      if (enable = '1') then
        count <= count + 1;
      end if;
    end if;
  end process;
```

```
    cout <= (count(7) &
      (count(7) xor count(6)) &
      (count(6) xor count(5)) &
      (count(5) xor count(4)) &
      (count(4) xor count(3)) &
      (count(3) xor count(2)) &
      (count(2) xor count(1)) &
      (count(1) xor count(0)) );
end architecture;
```

binary	gray
000	000
001	001
010	011
011	010
100	110
101	111
110	101
111	100



جامعة نينوى  
كلية هندسة الإلكترونيات

# Circuit Design with VHDL 8

Textbook: Volnei A. Pedroni

Submitted By: Hussein 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];
```

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

# VHDL> Loop statement

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

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

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

# VHDL> Loop statement

- EXIT: Used for ending the loop.

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

- NEXT: Used for skipping loop steps.

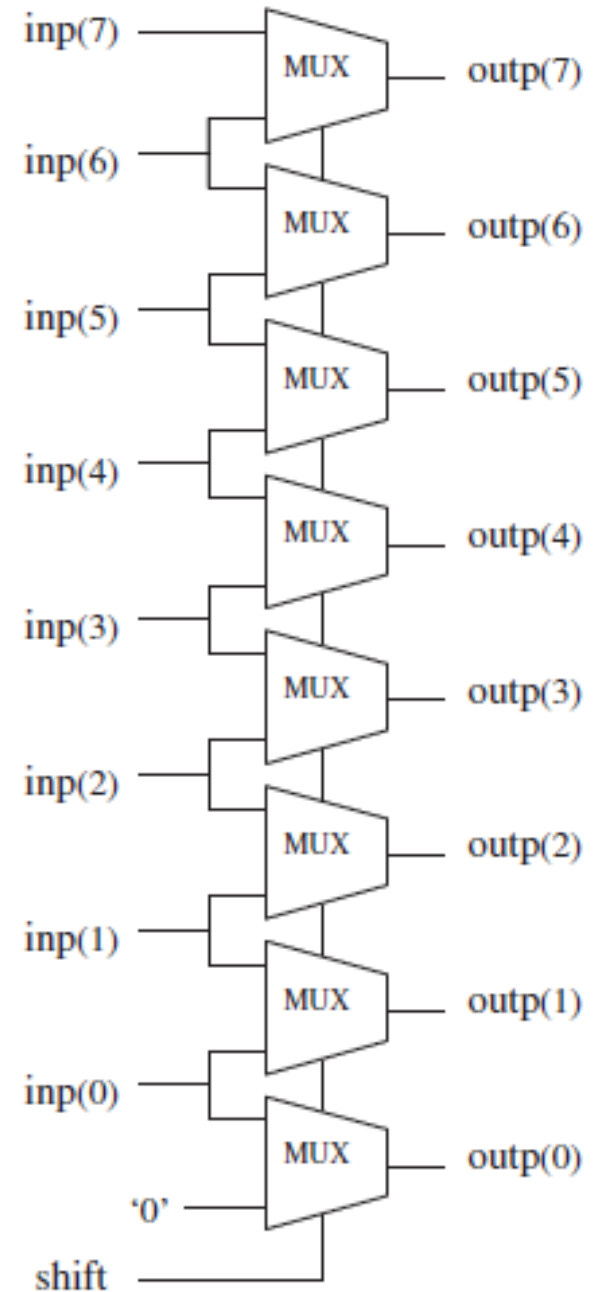
```
[label:] NEXT [loop_label] [WHEN condition];
```

# VHDL> Loop statement

- 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 bottom left corner of the diagram).
- If shift = 0, then  $\text{outp} = \text{inp}$ ;
- if shift = 1, then  $\text{outp}(0) = '0'$  and  $\text{outp}(i) = \text{inp}(i - 1)$ , for  $1 \leq i \leq 7$ .

# VHDL> Loop statement

- If  $\text{shift} = 0$ , then  $\text{outp} = \text{inp}$ ;
- if  $\text{shift} = 1$ , then  $\text{outp}(0) = '0'$  and  $\text{outp}(i) = \text{inp}(i - 1)$ , for  $1 \leq i \leq 7$ .



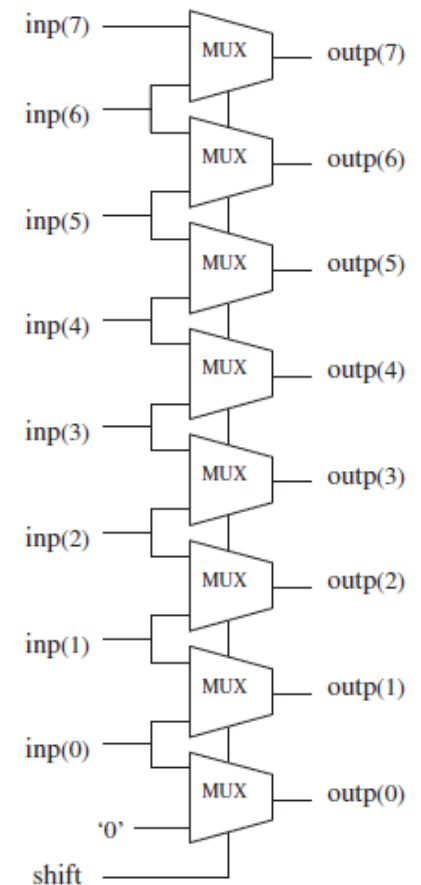


# VHDL> Simple Barrel Shifter

```
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 DOWNT0 0);  
8        shift: IN INTEGER RANGE 0 TO 1;  
9        outp: OUT STD_LOGIC_VECTOR (n-1 DOWNT0 0));  
10 END barrel;  
11 -----
```

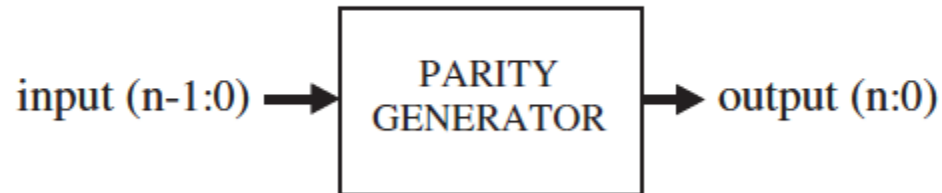
# VHDL> Simple Barrel Shifter

```
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> Examples

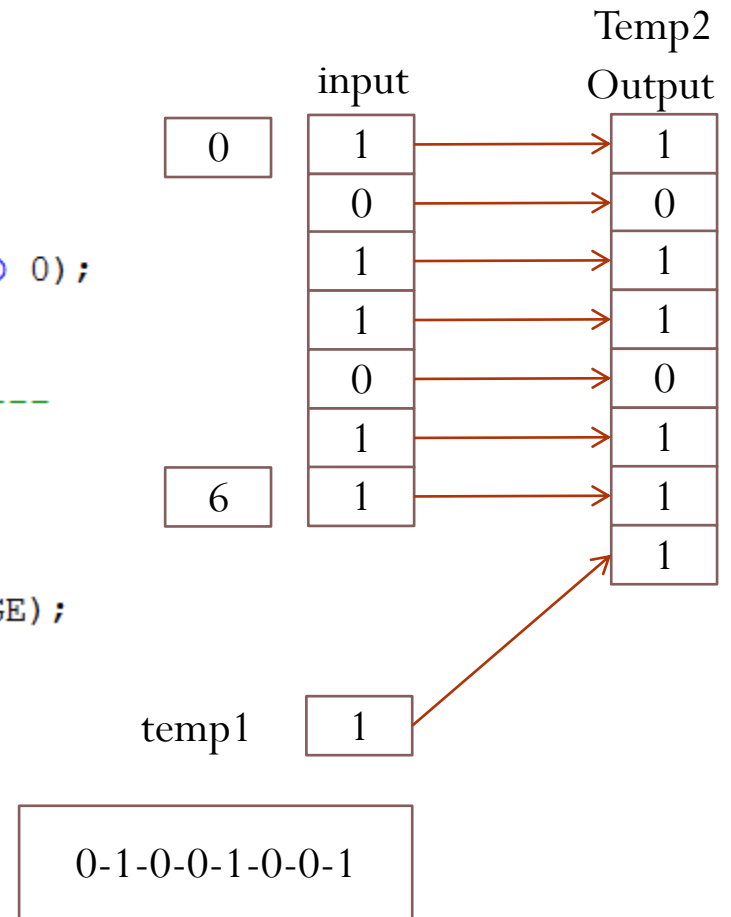
- Generic Parity **Generator**:
  - The circuit must add one bit to the input vector (on its left).
  - Such bit must be a '0' if the number of '1's in the input vector is even, or a '1' if it is odd, such that the resulting vector will always contain an even number of '1's (even parity).



# VHDL> Examples

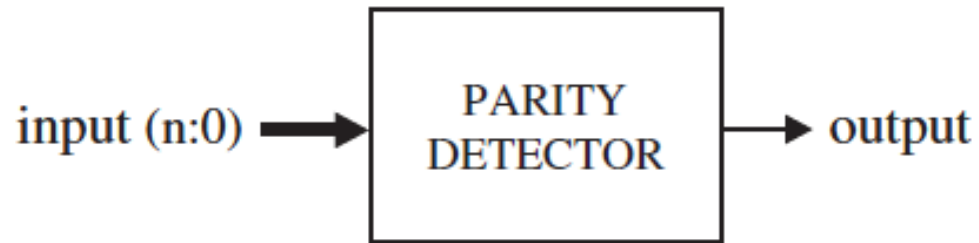
- Generic Parity Generator:

```
-----  
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 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:

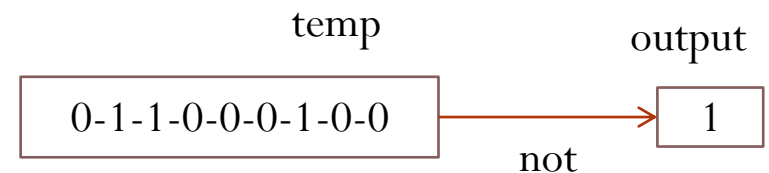
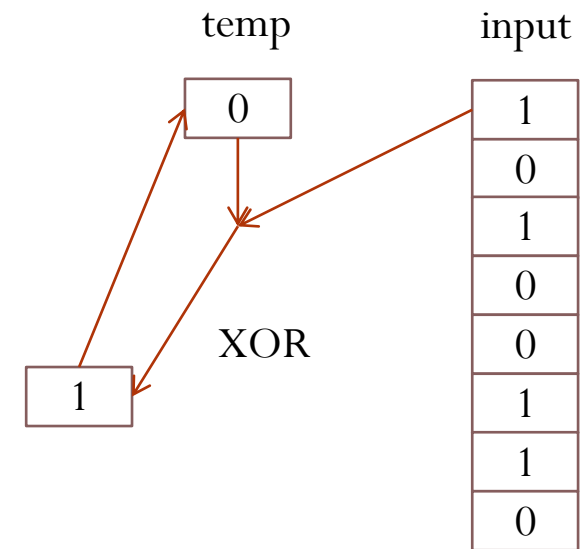
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-----

ENTITY parity_det IS
GENERIC (n : INTEGER := 7);
PORT ( input: IN Std_logic_VECTOR (n DOWNT0 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;
```





جامعة نينوى  
كلية هندسة الإلكترونيات

# Circuit Design with VHDL 9

Textbook: Volnei A. Pedroni

Submitted By: Hussein Aideen

# VHDL> Arrays in VHDL

- Arrays are collections of objects of the same type.

- one-dimensional (1D), 

0	1	0	0	0
---	---	---	---	---

0	1	0	0	0
---	---	---	---	---

- two-dimensional (2D),

1	0	0	1	0
---	---	---	---	---

1	1	0	0	1
---	---	---	---	---

- one-dimensional-by-one-dimensional (1Dx1D).

0	1	0	0	0
---	---	---	---	---

1	0	0	1	0
---	---	---	---	---

1	1	0	0	1
---	---	---	---	---



## 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];

```
TYPE row IS ARRAY (7 DOWNT0 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 DOWNT0 0);
```

# VHDL> Arrays in VHDL

- 2D Array

```
TYPE matrix2D IS ARRAY (0 TO 3, 7 DOWNT0 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)
```

- ROM (Read Only Memory)
- Example: Write a VHDL code to design a ROM which has a size of 64bit i.e. word size=8, number of addresses =8, assume a random data are stored in the 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 DOWNT0 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 DOWNT0 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 -----
```



جامعة نينوى  
كلية هندسة الإلكترونيات

# Circuit Design with VHDL 10

Textbook: Volnei A. Pedroni

Submitted By: Hussein Aideen

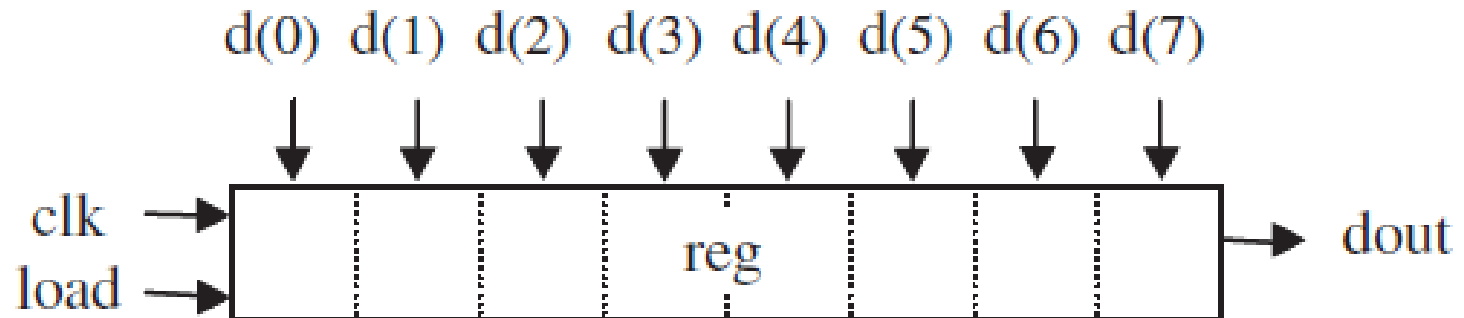
## VHDL> Examples

- 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 DOWNT0 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 -----
```

## VHDL> Examples

- Parallel-to-Serial Converter
- circuit operation:
  - when  $\text{load} = 1$  the data must stored in the register.
  - when  $\text{load} = 0$  the registerd data shifted out to dout each time clk goes from 0 to 1.





## VHDL> Examples

- 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 DOWNT0 0);
7            clk, load: IN STD_LOGIC;
8            dout: OUT STD_LOGIC);
9  END serial_converter;
10 -----
```

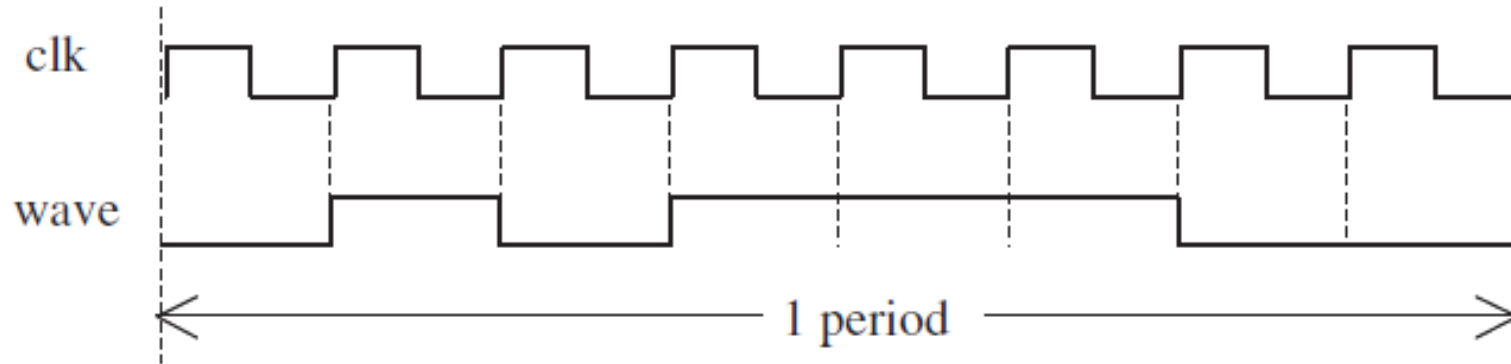
## VHDL> Examples

- Parallel-to-Serial Converter

```
11 ARCHITECTURE serial_converter OF serial_converter IS
12     SIGNAL reg: STD_LOGIC_VECTOR (7 DOWNT0 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 DOWNT0 0) & '0';
19             END IF;
20         END IF;
21     END PROCESS;
22     dout <= reg(7);
23 END serial_converter;
24 -----
```

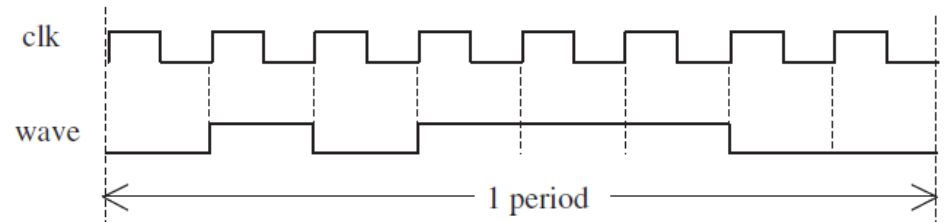
## VHDL> Examples

- Signal Generators



## VHDL> Examples

- 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  -----
```

# VHDL> Examples

- 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 -----
```



جامعة نينوى  
كلية هندسة الإلكترونيات

# Circuit Design with VHDL 11

Textbook: Volnei A. Pedroni

Submitted By: Hussein Aideen

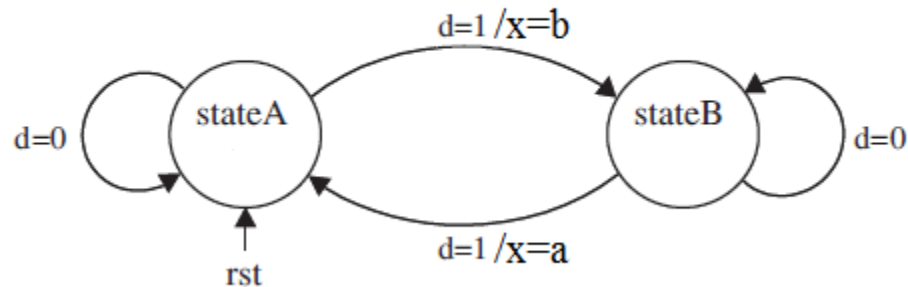
# VHDL> State Machines

---

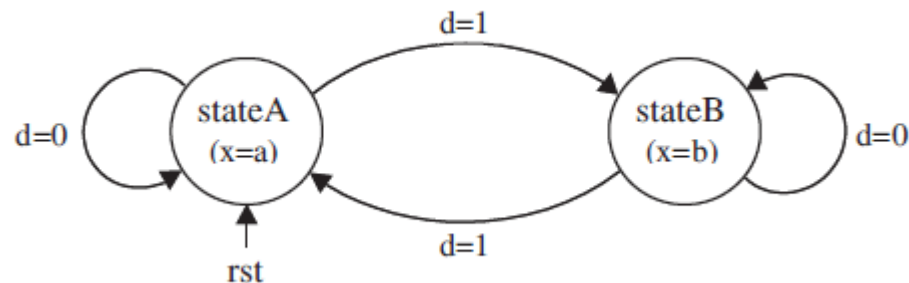
- Finite state machines (FSM) constitute a special modeling technique for sequential logic circuits.
- helpful in the design of certain types of systems, (digital controllers, counters, for example).

# VHDL> State Machines

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

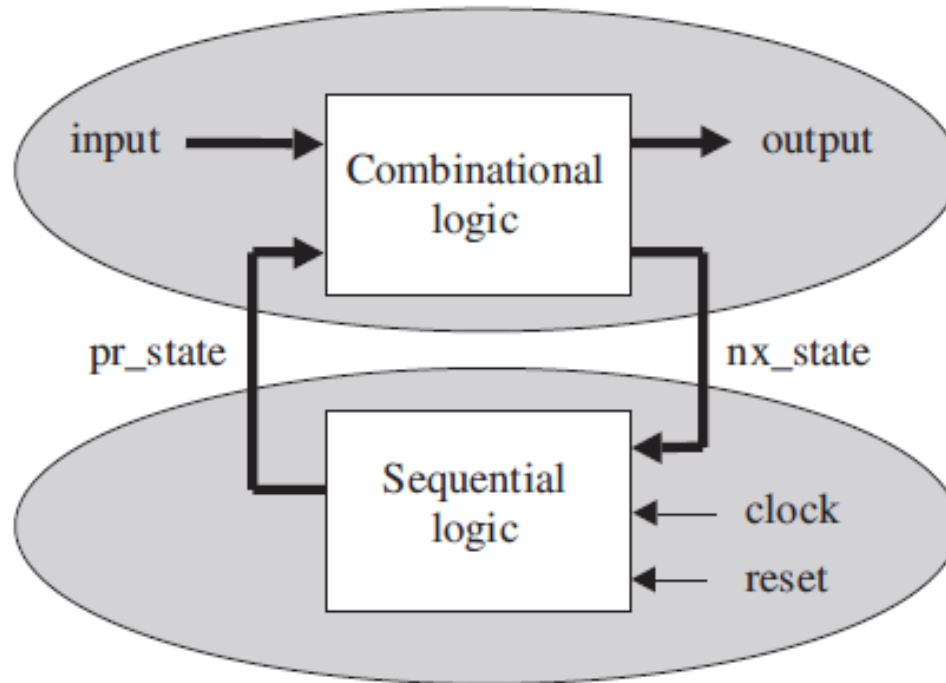


- Moore machine: the output depends only on the current state.





# VHDL> State Machines



**Figure 8.1**  
Mealy (Moore) state machine diagram.

# VHDL> State Machines

## Design Style #1:

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

```
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

---

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:

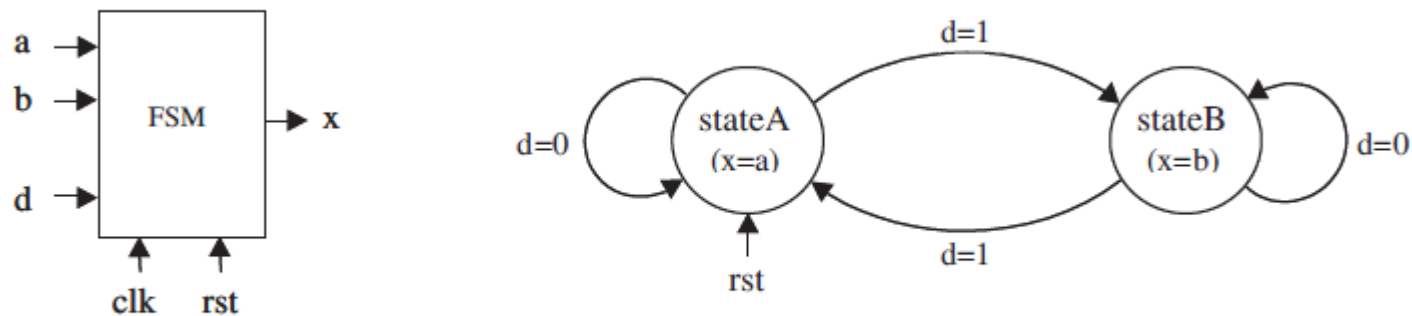
```
PROCESS (input, pr_state)
BEGIN
    CASE pr_state IS
        WHEN state0 =>
            IF (input = ...) THEN
                output <= <value>;
                nx_state <= state1;
            ELSE ...
            END IF;
        WHEN state1 =>
            IF (input = ...) THEN
                output <= <value>;
                nx_state <= state2;
            ELSE ...
            END IF;
        WHEN state2 =>
            IF (input = ...) THEN
                output <= <value>;
                nx_state <= state3;
            ELSE ...
            END IF;
        ...
    END CASE;
END PROCESS;
END <arch_name>;
```

# VHDL> State Machines

## Example

### Example 8.2: Simple FSM #1

Figure 8.4 shows the states diagram of a very simple FSM. The system has two states (stateA and stateB), and must change from one to the other every time  $d = '1'$  is received. The desired output is  $x = a$  when the machine is in stateA, or  $x = b$  when in stateB. The initial (reset) state is stateA.



**Figure 8.4**  
State machine of example 8.1.

# VHDL> State Machines

```
1  -----
2  ENTITY simple_fsm IS
3      PORT ( a, b, d, clk, rst: IN BIT;
4              x: OUT BIT);
5  END simple_fsm;
6  -----
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

```
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

---

## Design Style #2 (Stored Output):

- In #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

## Design Style #2 (Stored Output):

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-----
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

## Design Style #2 (Stored Output):

```
----- 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>;
```