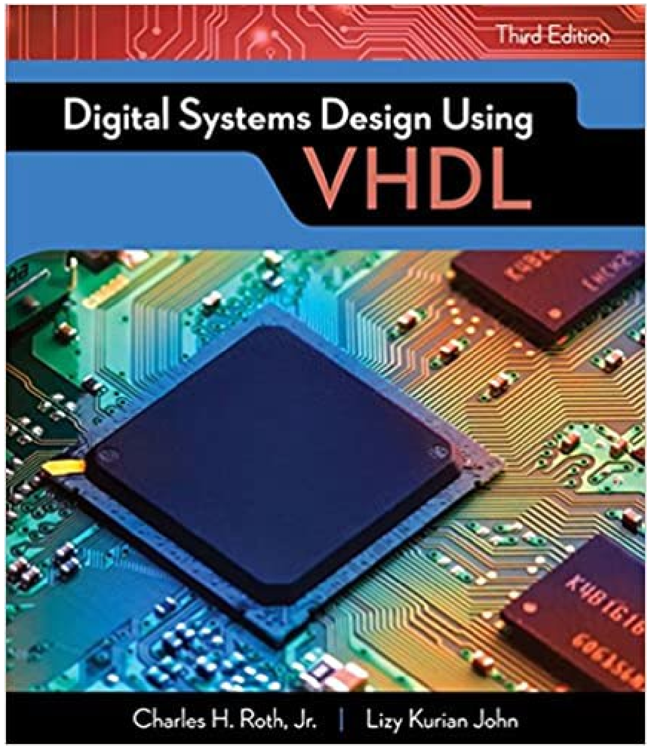
EE2000 Logic Circuit Design

Lecture 5- VHDL

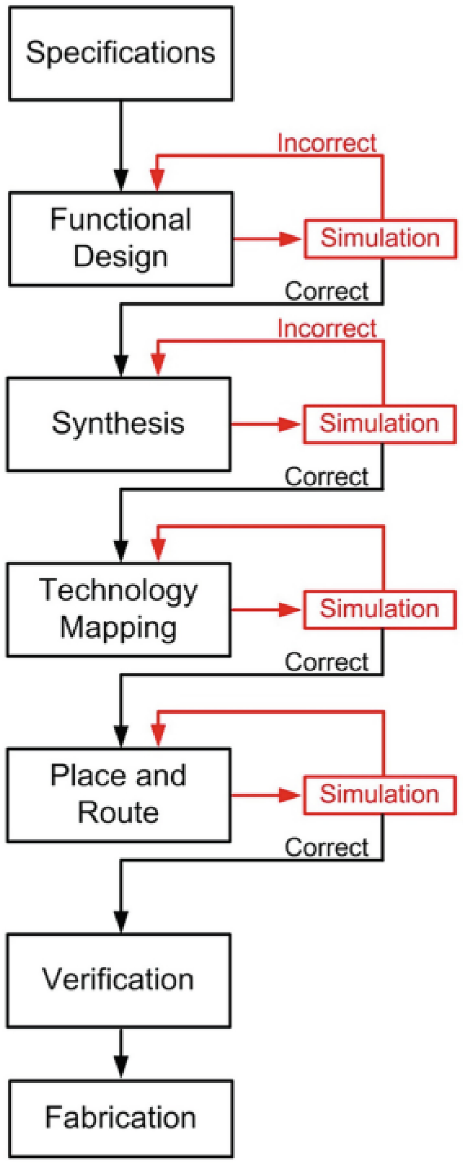


Outline

5.1 Hardware Description Language

- Hardware Description Languages (HDLs) is a software programming language used to **describe a digital system**
- Behavioral description: The general function of the design at the **algorithmic level** without specifying physical components, gates, *etc*.
- Structural description: **Specific components, gates, and their interconnections** are associated with the design

Modern Digital System Design Flow



State the desired behavior of the design

Design the system using HDL. The design is simulated to verify its functionality

Convert the design in HDL to gate-level connection

Map design to specific logic technology (physical components)

Arrange components to minimize area needed, wiring length and crossings

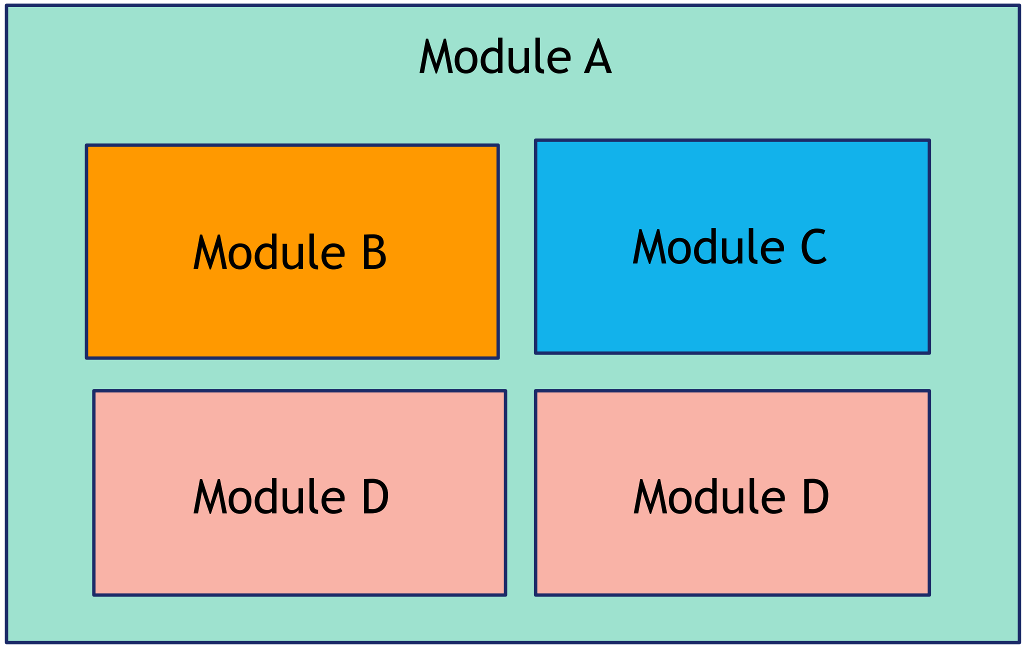
Final design is analyzed (gate and propagation delays, power consumption, etc.)

FPGA, ASIC, board-level, discrete parts

VHDL

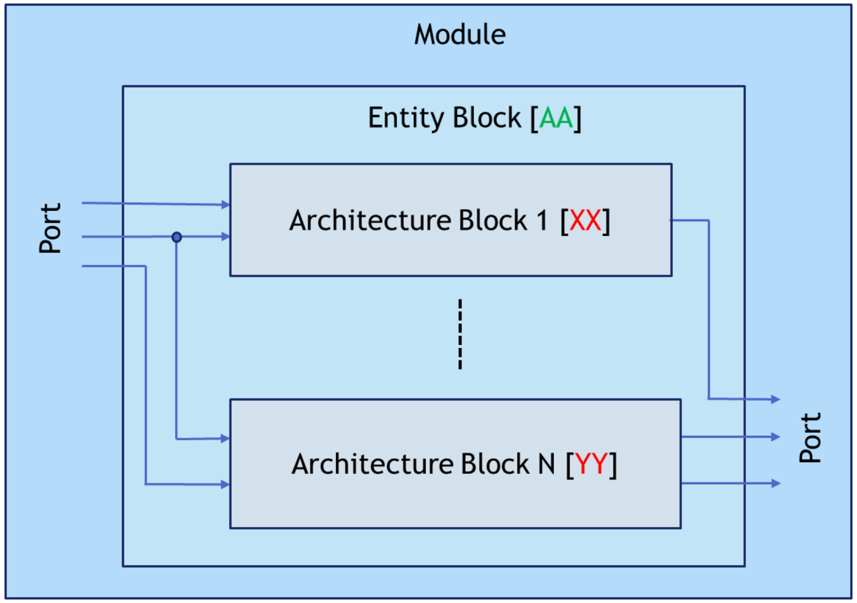
- Two popular HDLs (IEEE standard)
  - Verilog: Verifying Logic
  - VHDL: Very High Speed Integrated Circuit HDL
- In this lecture, we learn VHDL
- In the lab, we use Vivado design tool from Xilinx to design, simulate and synthesize the logic circuit/system
- A top-down design methodology: System is specified and tested using simulator; then refined to structural description and led to actual hardware implementation

5.2 VHDL Code Structure



- Like breaking down complicated circuits into various smaller circuits
- Each module describes a **smaller circuit**
- Modules are connected through **ports (inputs/outputs)** to form the final circuit

VHDL Code Structure - Illustration



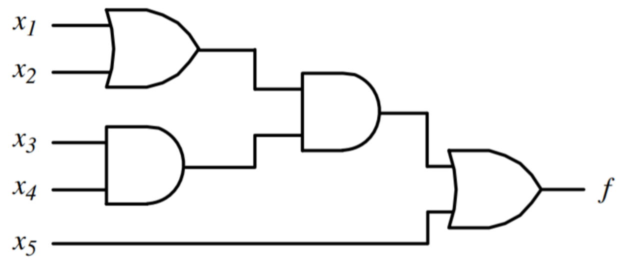
Library declaration
Entity declaration
Architecture declaration

Library declaration: IEEE Standard library, e.g., STD_LOGIC_1164 contains STD_LOGIC types & related functions

Entity declaration: defines the input or output ports to the current module

Architecture declaration: describes the logic function of the corresponding entity

VHDL Format and Synthax



Library declaration

```
library IEEE;                       -- load to access a library
use IEEE.std_logic_1164.all;        -- Use a package in the library.
```

Entity declaration

```
entity logic_c is                          -- define the name of the entity
    port (x1, x2, x3, x4, x5 : in bit;     -- inputs and data type
          f: out bit);                     -- outputs and data type
end logic_c;
```

Architecture declaration

```
architecture behavior of logic_c is   -- define the architecture and
begin                                             -- specify the entity
        f <= ((x1 or x2) and                 --define the logic operation
              (x3 and x4)) or x5);
end behavior;
```

Entity Synthax

```
entity entity-name is
    port (interface-signal-declaration);
end entity-name;
```

Interface-signal-declaration

```
 list-of-signals: mode data-type := initial value;
```

Example

```
port (A, B: in integer := 2; C, D: out bit );
```

- A and B are input signals of type integer that are initially set to 2
- C and D are output signals of type bit (default ‘0’)

Entity Synthax

```
entity two_gates is
    port (A, B: in integer := 2; C, D: out bit );
end two_gates;
```

- Port name and entity name (identifiers)
  - ➢ may contain letters, numbers and underscore character (_)
  - ➢ Must start with a letter
  - ➢ Cannot end with underscore or double underscores
- Mode: **in**, **out**, **inout, buffer (output + feedback)**
- VHDL is case **insensitive**!
  - ➢ CLK <= A and B
  - ➢ Clk <= a AND b

Question

Which of the following identifier is illegal to be used as an entity name in VHDL?

• TwO_gaTE	Legal
• 2_gate	Illegal – cannot start with a number
• T2-gate	Illegal – cannot have other symbol
• _2gate	Illegal – cannot start with underscore
• AND	Illegal – Reserved word

Data Objects

Data objects: A container for data values; a place to store and retrieve data values

Constant:

- Hold unchangeable values
- Can be declared anywhere that allows declaration
- Declaration synthax

  constant name: data_type := initial value;

  e.g. **constant** data_bus: integer := 4 ns;

Data Objects

```
entity AA is
port (x1, x2, x3, x4, x5 : in bit;
        f: out bit);
end AA;
```

Signal:

- Represent **wires** in schematics
- Declare in **port** of **entity** as inputs/outputs
- Declare in **architecture** before **begin** as internal signals
- Declaration synthax (without/with initial value)

  signal name: data_type := initial value;

  e.g. **signal** count: bit := ‘1’;

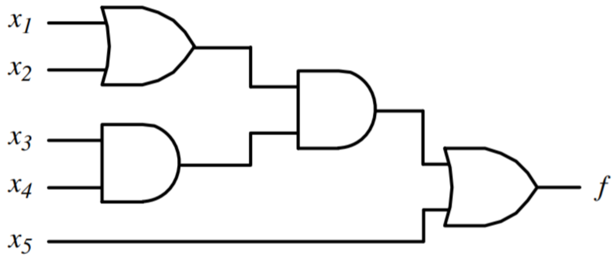
  signal count: bit;

- Assignment synthax

  count <= ‘0’;

```
architecture AA of BB is
signal count: bit := ‘1’;
begin
  count <= ‘0’;
end AA;
```

Architecture declaration



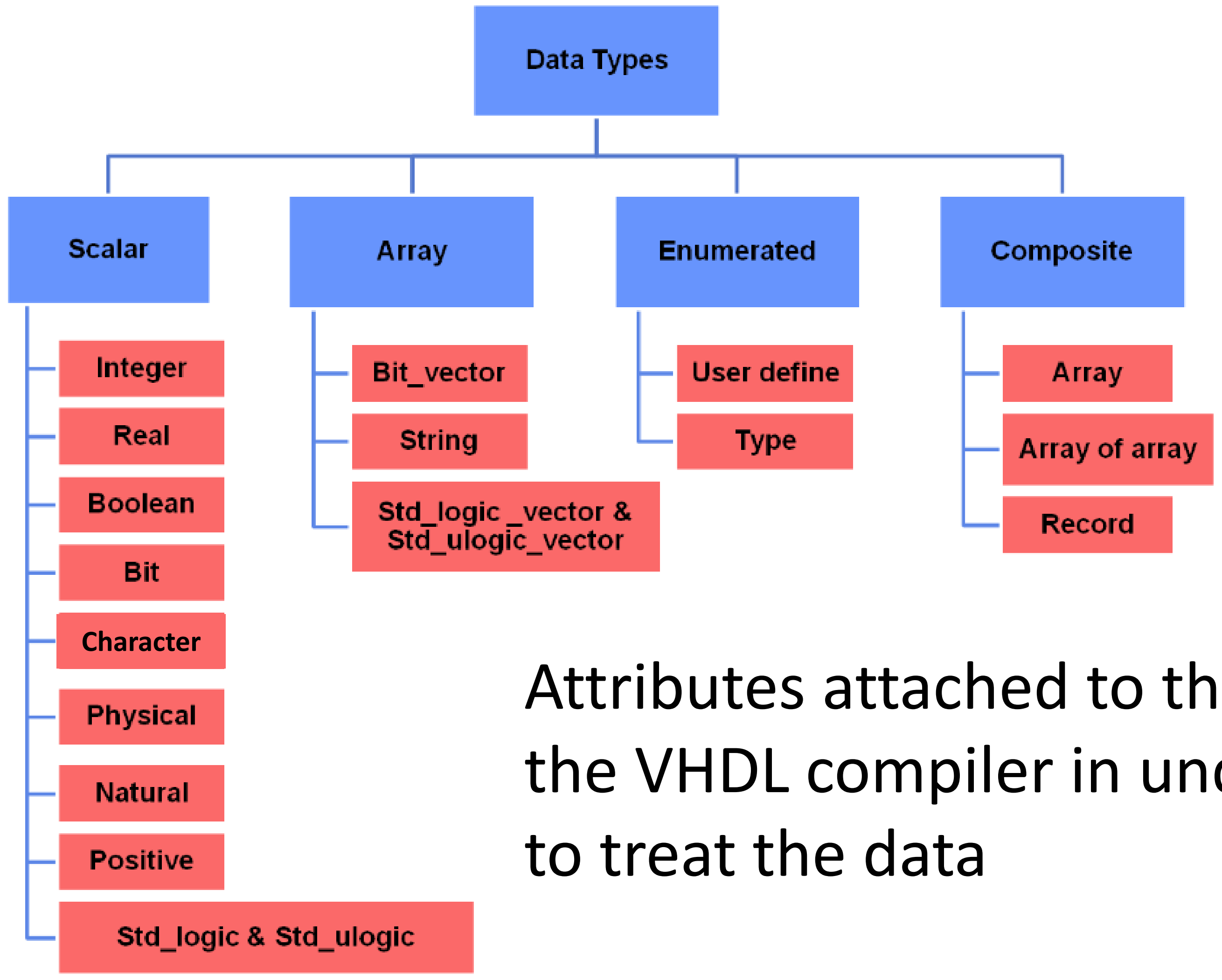
Version 1

```
architecture behavior of logic_c is   -- define the architecture and
begin                                        -- specify the entity
      f <= ((x1 or x2) and                  --define the logic operation
           (x3 and x4)) or x5);
end logic_c;
```

Version 2

```
architecture behavior of logic_c is   -- define the architecture and
signal y1, y2, y3: bit;                  -- declare internal signals
begin                                        -- specify the entity
     y1 <= x1 or x2;
     y2 < = x3 and x4;
     y3 <= y1 and y2;
     f <= y3 or x5;                         --define the logic operation
end logic_c;
```

5.3 VHDL Data Types



Attributes attached to the data that help the VHDL compiler in understanding how to treat the data

Standard Data Types

Type	Description
Boolean	FALSE or TRUE
Bit	0 or 1
Character	Any ASCII character
Integer	Integer in the range -2^{31} to $+(2^{31} - 1)$
Natural	Integer in the range 0 to $+(2^{31} - 1)$
Positive	Integer in the range 1 to $+(2^{31} - 1)$
Real	Floating-point number in the range −1.0E38 to +1.0E38
Time	An integer with units fs, ps, ns, us, ms, sec, min, or hr

Bit & Bit Vector

Signal declaration synthax

```
        signal name: type;
```

Examples

```
signal x1: bit;
signal y1: integer := 1;
signal z1: real := 3.14159;
signal x1: bit_vector(3 downto 0);
signal x1: bit_vector(0 to 3);
```

bit: a bit object with value either ‘0’ or ‘1’

bit_vector: multi-bit data in an array of bit objects

Bit Vector

signal x1: bit_vector(3 **downto** 0);

- Elements: x1(3), x1(2), x1(1), x1(0)
- x1(3) holds the most significant bit
- x1 <= “0101”;
- x1(3) = ‘0’, x1(2) = ‘1’, x1(1) = ‘0’, x1(0) = ‘1’

signal x1: bit_vector(0 **to** 3);

- Elements: x1(0), x1(1), x1(2), x1(3)
- x1(0) holds the most significant bit
- x1 <= “0101”;
- x1(0) = ‘0’, x1(1) = ‘1’, x1(2) = ‘0’, x1(3) = ‘1’

Bit Vector Assignment Types

signal x: bit_vector(3 **downto** 0) := “0000”;

- x <= “0101”;
  - x(3) = ‘0’, x(2) = ‘1’, x(1) = ‘0’, x(0) = ‘1’

- x(3) <= ‘1’;
  - x(3) = ‘1’, x(2) = ‘0’, x(1) = ‘0’, x(0) = ‘0’

- x <= (0 | 1 => ‘1’, **others** => ‘0’);
  - x(3) = ‘0’, x(2) = ‘0’, x(1) = ‘1’, x(0) = ‘1’

- x <= (2 **downto** 0 => ‘1’, **others** => ‘0’);
  - x(3) = ‘0’, x(2) = ‘1’, x(1) = ‘1’, x(0) = ‘1’

Exercise

```
signal C: bit_vector (0 to 3);
signal D: bit_vector (3 downto 0);
signal A: bit_vector (7 downto 0);
```

```
C <= "1101";
D <= C;
A(6 downto 3) <= D;
```

Determine the stored value in the following bit object

C(0) =	C(1) =	C(2) =	C(3) =
D(3) =	D(2) =	D(1) =	D(0) =
A(6) =	A(5) =	A(4) =	A(3) =

Array Data Type

Bit Vector

- An array of bits
- **signal** str_1: bit_vector(1 to 3) := “0011”;

String

- An array of character type elements
- **signal** str_1: string(1 to 11) := “Welcome All”;

Physical Data Type

- Physical type allows user to define **measurement units**, like length, time, pressure, capacity, etc.
- The only predefined physical type is time

```
type time is range -2147483647 to 2147483647
    units
        fs;
        ps = 1000 fs;
        ns = 1000 ps;
        us = 1000 ns;
        ms = 1000 us;
        sec = 1000 ms;
        min = 60 sec;
        hr = 60 min;
    end units;
```

```
signal counter: time := 1 ns;
```

* Note: Integer only

User Defined Physical Type

You can define your own physical type,

e.g.

```
type distance is range 0 to 1E7
    units
        um;  -- micrometer
        mm = 1000 um; -- millimeter
        cm = 10 mm; -- centimeter
        m = 100 cm; -- meter
        inch = 25400 um;
    end units;
```

```
signal dis1, dis2: distance;
Dis1 <= 28 mm;
Dis2 <= 2 inch –  1 mm;
```

User Defined Enumerated Data Type

- Predefined Enumeration types
  - **type** BOOLEAN **is** (FALSE, TRUE);
  - **type** BIT **is** (‘0’, ‘1’);

- Defined your own enumerated type,

e.g. **type** colors **is** (RED, GREEN, BLUE);
type state **is** (S0, S1, S2, S4);

- Leftmost value is the least and is the default value.

e.g. **when** my_color **>=** RED

User Defined Enumerated Data Type

- Subtype: Create a custom type from one of the existing types (a constrained version)

subtype integer_8_bit **is** integer **range** 0 **to** 255;

subtype MyBit **is** std_logic **range** ‘0’ **to** ‘1’;

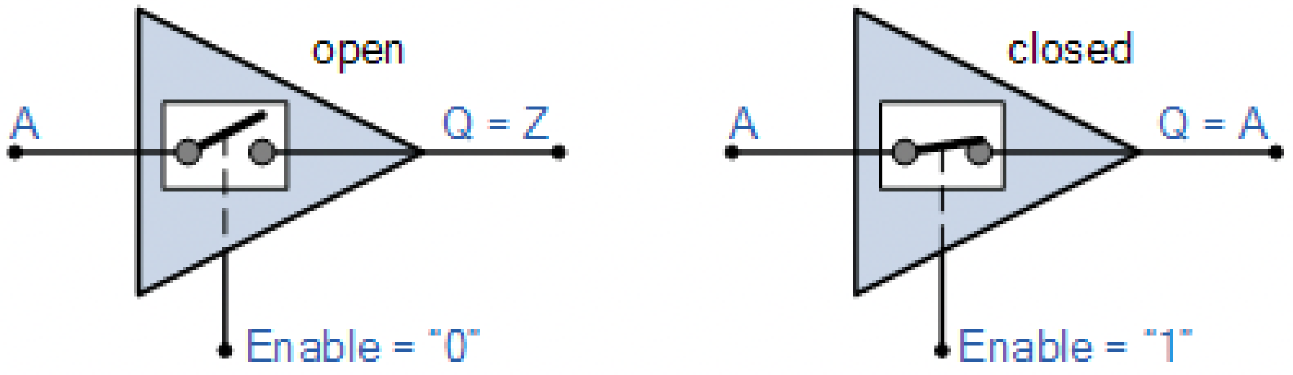
signal x1: integer_8_bit;

signal y1: MyBit;

IEEE std_logic_1164 package

- Bit only provides two outputs ‘0’ or ‘1’
- We might need more states, such as don’t care, uninitialized, etc.

e.g.



Tri-State buffer: When Enable is “0”, the circuit is disabled, High-impedence (High-Z) state

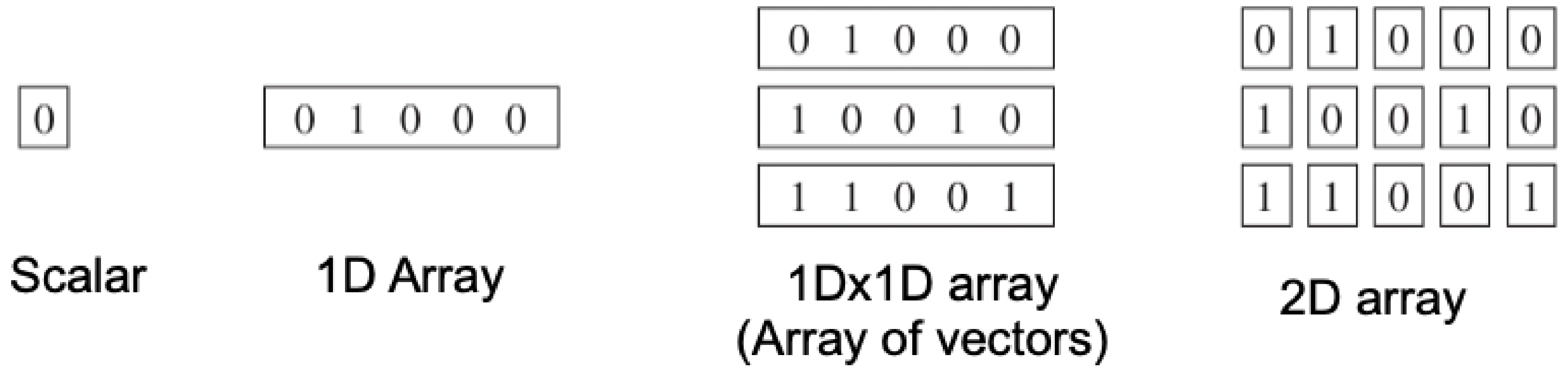
IEEE std_logic_1164 package

Type	Description
U	Uninitialized; Default value
X	Unknown. Cannot determine as 1 or 0
0	Logic 0
1	Logic 1
Z	High Impedance (Tri-state buffer when not enabled)
W	Weak signal, can't tell if it should be 0 or 1
L	Weak signal that should probably go to 0
H	Weak signal that should probably go to 1
-	Don't care

std_logic and std_logic_vector

```
library IEEE;
use IEEE.std_logic_1164.all;

signal x1: std_logic;
signal x2: std_logic;
signal x3: std_logic;
signal x4: std_logic;

signal x: std_logic_vector(0 to 3);
```

User Defined Array Data Type



```
type type_name is array (specification) of data_type;
signal signal_name: type_name := initial_value;
```

User Defined Array Data Type

type row **is array** (7 downto 0) **of** std_logic;

- Define a 1D array named as **row** with eight STD_LOGIC values

0	**1**	**1**	**0**	**1**	**0**	**0**	**1**

type array1 **is array** (0 to 3) **of** row;

- Define a 1D × 1D array named as **array1** with 4 rows of vectors (eight STD_LOGIC values)

signal u: row;

signal v: array1;

u(0) <= v(1)(2);

0	**1**	**1**	**0**	**1**	**0**	**0**	**1**
0	**1**	**1**	**0**	**1**	**0**	**0**	**1**
0	**1**	**1**	**0**	**1**	**0**	**0**	**1**
0	**1**	**1**	**0**	**1**	**0**	**0**	**1**

User Defined Array Data Type

type array2 **is array** (0 to 3) **of** std_logic_vector(7 downto 0);

- Define a 1D × 1D array named as **array2** with 4 rows of vectors (eight STD_LOGIC values)
- Same as array 1

type array3 **is array** (0 to 3, 7 downto 0) **of** std_logic;

- Define a 2D array named as **array3** with an array of (4, 8) STD_LOGIC values

0	**1**	**1**	**0**	**1**	**0**	**0**	**1**
0	**1**	**1**	**0**	**1**	**0**	**0**	**1**
0	**1**	**1**	**0**	**1**	**0**	**0**	**1**
0	**1**	**1**	**0**	**1**	**0**	**0**	**1**

User Defined Array Data Type

```
type row is array (7 downto 0) of std_logic;
type array1 is array (0 to 3) of row;
type array3 is array (0 to 3, 7 downto 0) of std_logic;

signal u: row;
signal v: array1;
signal x: array3;

u(0) <= v(1)(2);
u(2) <= x(2,1);
```

Record Data Type

- Group a number of common signals together to simplify the port list in entity
- e.g.,

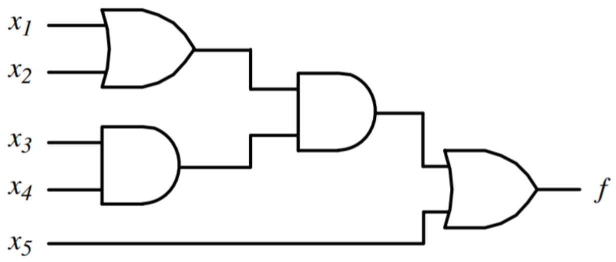
```
type my_record is record
    rx: std_logic;
    cts: bit;
    tx: std_logic;
end record;

signal one_record: my_record;
one_record.tx <= ‘1’;
one_record <= (rx => ‘1’, cts => ‘1’, tx => ‘1’);
```

Exercise

Declare a record type that consists of an 8 bit_vector, an 8-bit integer and a time type

```
type my record is record
    xx: bit vector(7 downto 0);
    yy: integer range 0 to 255;
    zz: time;
end record;
```

VHDL Format and Synthax



Library declaration

```
library IEEE;                       -- load to access a library
use IEEE.std_logic_1164.all;        -- Use a package in the library.
```

Entity declaration

```
entity logic_c is                        -- define the name of the entity
    port (x1, x2, x3, x4, x5 : in bit;   -- inputs and data type
          f: out bit);                   -- outputs and data type
end logic_c;
```

Architecture declaration

```
architecture behavior of logic_c is   -- define the architecture and
begin                                            -- specify the entity
        f <= ((x1 or x2) and                --define the logic operation
              (x3 and x4)) or x5);
end behavior;
```

Architecture

Architecture declaration: describes the logic function of the corresponding entity.

The most basic structure of an architecture

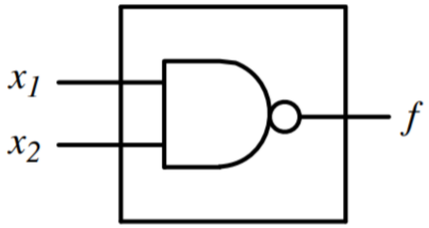
```
architecture arc_name of entity_name is
begin
    functional code
end arc_name;
```

5.4 VDHL Operators

Operators for BIT and BOOLEAN types

Logical operation	Operator	Example
AND	AND	Z <= A AND B;
NAND	NAND	Z <= A NAND B;
NOR	NOR	Z <= A NOR B;
NOT	NOT	Z <= NOT (A);
OR	OR	Z <= A OR B;
XNOR	XNOR	Z <= A XNOR B;
XOR	XOR	Z <= A XOR B;

Example



```
architecture Behavior of nand_gate is
begin
                    f <= not (x1 and x2);
end Behavior;
```

Alternative: `f <= x1 nand x2;`

Concatenation Operator (&)

To combine two bits or bit vectors

e.g.

A1 = “0000”   A2 = “10”   A3 = ‘0’   A4 = ‘1’

A1 & A2 = “000010”

A1 & A3 = “00000”

A2 & A3 = “100”

etc...

Arithmetic Operators

Operators for numeric types

Arithmetic operation	Operator	Example
Addition	`+`	`Z <= A + B;`
Subtraction	`-`	`Z <= A - B;`
Multiplication	`*`	`Z <= A * B;`
Division	`/`	`Z <= A / B;`
Exponentiating	`**`	`Z <= 4 ** 2;` (4^2)
Modulus	`MOD`	`Z <= A MOD B;`
Remainder	`REM`	`Z <= A REM B;`
Absolute value $\lvert\pm A\rvert = A$	`ABS`	`Z <= ABS A;`

REM vs MOD

Remainder: Same as mathematical operation

5 rem 3 = 2 -- 5/3 remainder is 2

(-5) rem 3 = -2

5 rem (-3) = 2

(-5) rem (-3) = -2

$$
\begin{array}{r|l} & 1 \\ \cline{2-2} 3 & 5 \\ & 3 \\ \cline{2-2} & 2 \end{array}
\qquad
\begin{array}{r|l} & -1 \\ \cline{2-2} 3 & -5 \\ & -3 \\ \cline{2-2} & -2 \end{array}
\qquad
\begin{array}{r|l} & -1 \\ \cline{2-2} -3 & 5 \\ & 3 \\ \cline{2-2} & 2 \end{array}
\qquad
\begin{array}{r|l} & 1 \\ \cline{2-2} -3 & -5 \\ & -3 \\ \cline{2-2} & -2 \end{array}
$$

REM vs MOD

Modulus:

(1) (A mod B) has the sign of B

(2) (A mod B) has an absolute value smaller than the absolute value of B

(3) (A mod B) = A – B * N with A, B and N are integers

```
5 mod 3 = 2                 -- 5 - (3*1) = 2
(-5) mod 3 = 1              -- -5 - (3*-2) = 1
5 mod (-3) = -1             -- 5 - (-3*-2) = – 1
(-5) mod (-3) = -2          -- -5 - (-3*1) = – 2
```

REM vs MOD

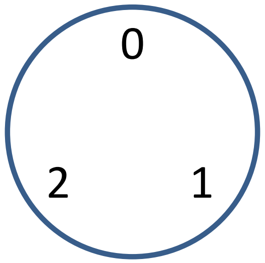
Modulus: Think of a clock!

(1) For (A mod B), A clock that count from 0 to abs(B)-1

(2) AB is pos → clockwise for A steps starting from 0

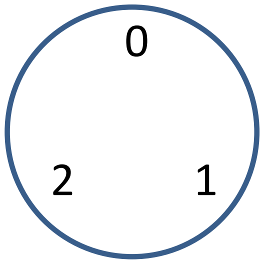
(3) AB is neg → anticlockwise for A steps starting from 0

(4) Sign of (A mod B) follows B



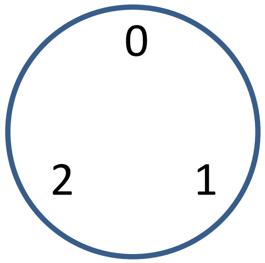
5 mod 3

2



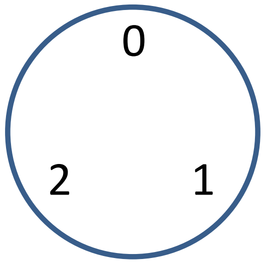
-5 mod 3

1



5 mod -3

-1



-5 mod -3

-2

REM vs MOD

Work out the following

M <= index mod 4;

Index	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9
M																			

M <= index rem 4;

Index	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9
M																			

Relational Operators

Relational operation	Operator	Example
Equal to	=	If (A = B) Then
Not equal to	/=	If (A /= B) Then
Less than	<	If (A < B) Then
Less than or equal to	<=	If (A <= B) Then
Greater than	>	If (A > B) Then
Greater than or equal to	>=	If (A >= B) Then

Shift Operators (Logical)

Shift operation	Operator	Remark
Shift Left Logical	sll	Shift left and fill blank with 0
Shift Right Logical	srl	Shift right and fill blank with 0
Rotate Left Logical	rol	Circular operation
Rotate Right Logical	ror	Circular operation
Shift Left Arithmetic	sla	Shift left and fill blank with LSB
Shift Right Arithmetic	sra	Shift right and fill blank with MSB

A1 = “1011”

1) A1 sll 1 -- 0110
2) A1 sll 3 -- 0110 then 1100 then 1000
3) A1 sll -3 -- A1 srl 3

Shift Operators (Logical)

A1 = “1011”

1) A1 sll 1 -- 0110
2) A1 sll 3 -- 0110 then 1100 then 1000
3) A1 sll -3 -- A1 srl 3
4) A1 srl 3 -- 0101 then 0010 then 0001
5) A1 sla 3 -- 0111 then 1111 then 1111
6) A1 sra 3 -- 1101 then 1110 then 1111
7) A1 rol 3 -- 0111 then 1110 then 1101
8) A1 ror 3 -- 1101 then 1110 then 0111

Precedence of VHDL Operators

Precedence (High to Low)	Operators
1	** abs NOT
2 Multiplying	* / mod rem
3 Unary (Sign)	+ -
4 Adding	+ - &
5 Shift	sll srl sla sra rol ror
6 Relational	= /= < <= > >=
7 Logic	and or nand nor xor xnor

- Operators in the same class are applied from left to right
- Parentheses change the order of precedence; and good coding style

Exercise

Transform the following logic expression to VHDL code

1) $f = ab' + a'b$

2) $f = a(b' + a')b$

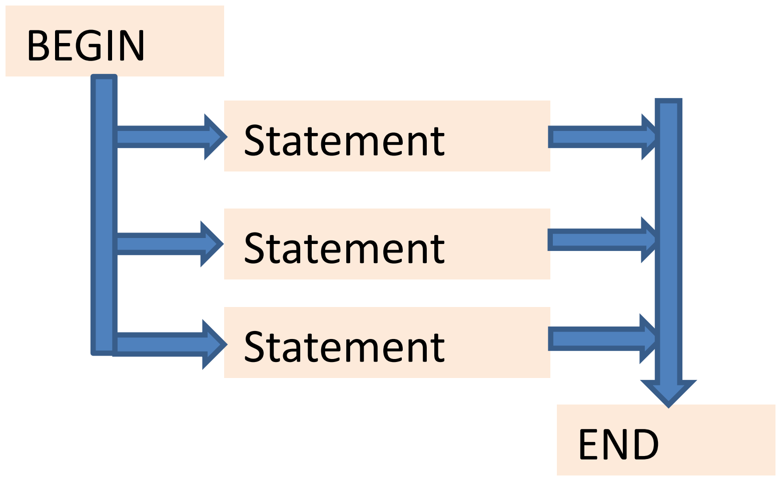
Exercise

Transform the following logic expression to VHDL code

Boolean	VHDL Boolean
$Y = \overline{AB}$	
$Y = \overline{A + B}$	
$Y = A + BC$	
$Y = C\overline{X + D}$	
$Y = A\bar{B}C + \bar{A}\,\bar{B}C + \overline{AB}C$	

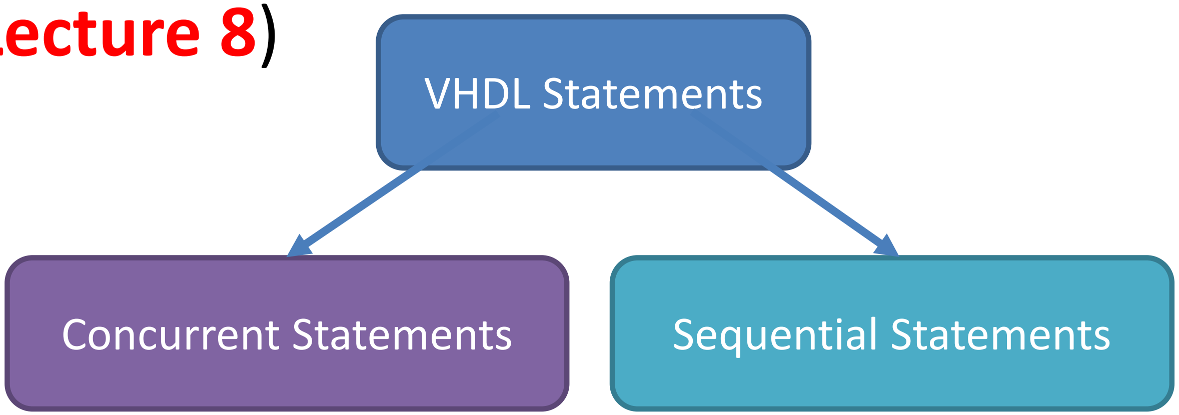
5.5 VHDL Synthax for Logic Circuits

- VHDL code describes the circuit **design with Boolean expressions**
- It is different from a computer program which is composed of a **sequence** of instructions for the CPU to execute
- VHDL Boolean expressions are statements only used for configuring the FPGA chip (no CPU to execute these statements)

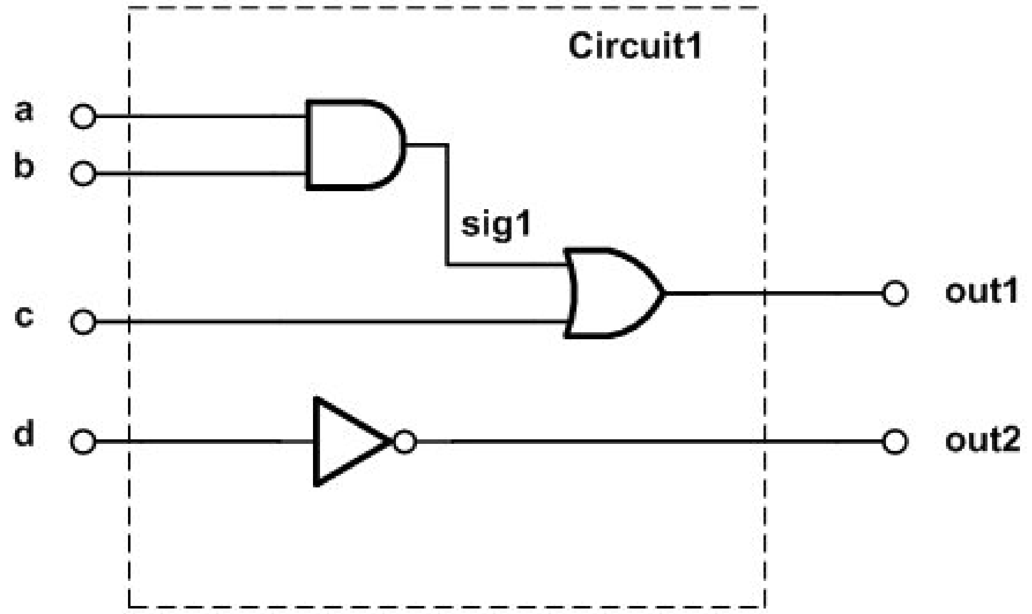


Concurrent vs Sequential Statements

- VHDL models **combinational circuit** using **concurrent statements**
- **Concurrent statements**: All statements written in the **architecture** body will be executed **concurrently** (not in sequence!)
- **Sequential statements**: Statements within a **process** in the architecture body are executed **sequentially** (will be discussed in **Lecture 8**)



Example



Precedence order is NOT important for concurrent statements !!

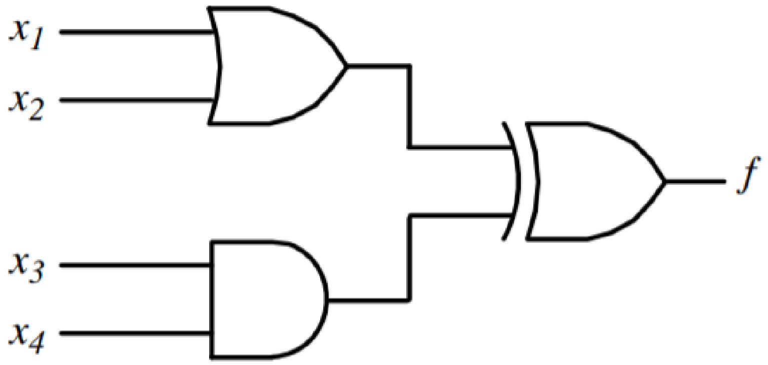
```
sig1 <= (a and b);
out1 <= (sig1 or c);
out2 <= (not d);
```

=

```
out1 <= (sig1 or c);
sig1 <= (a and b);
out2 <= (not d);
```

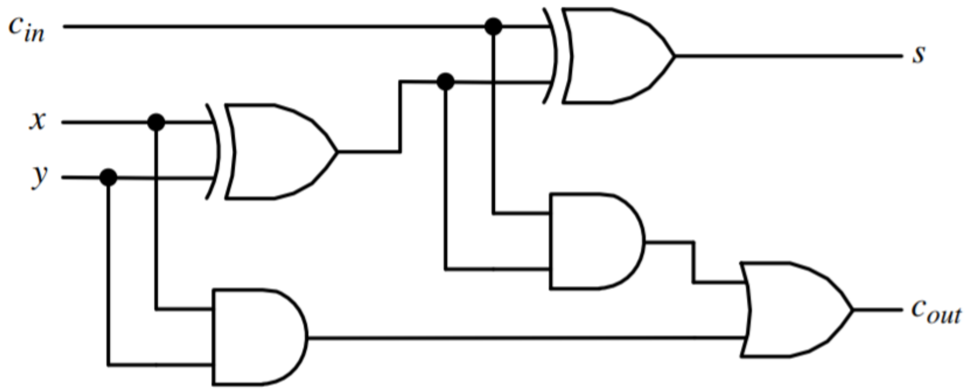
Exercise

Write your own VHDL statement to describe the logic circuit



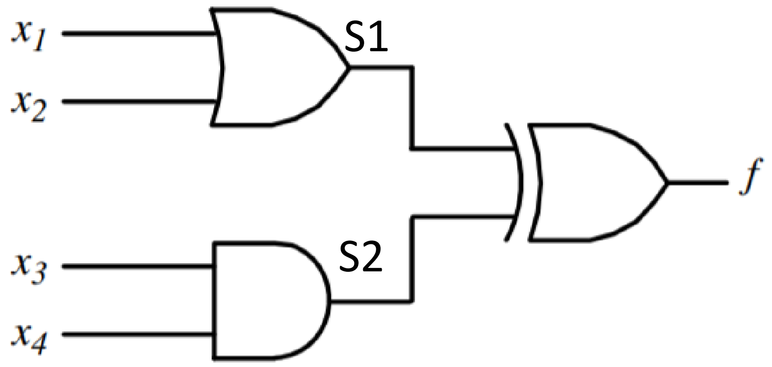
Exercise

Write your own VHDL statement to describe the logic circuit



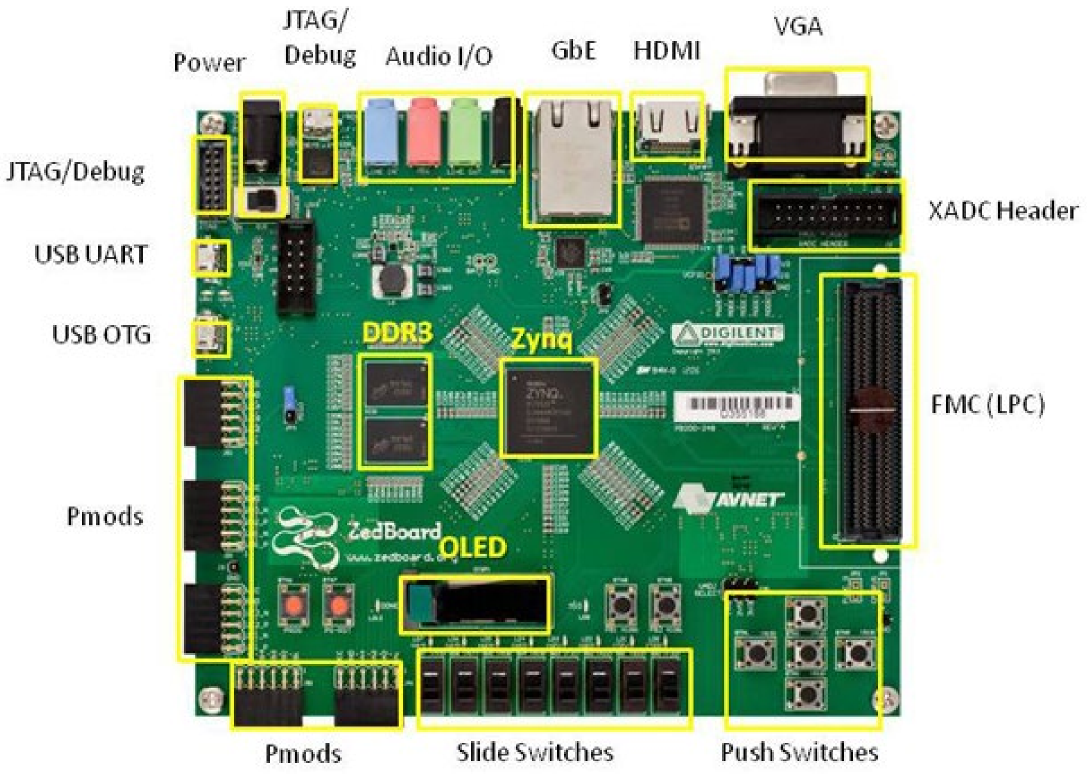
Example of Complete Module

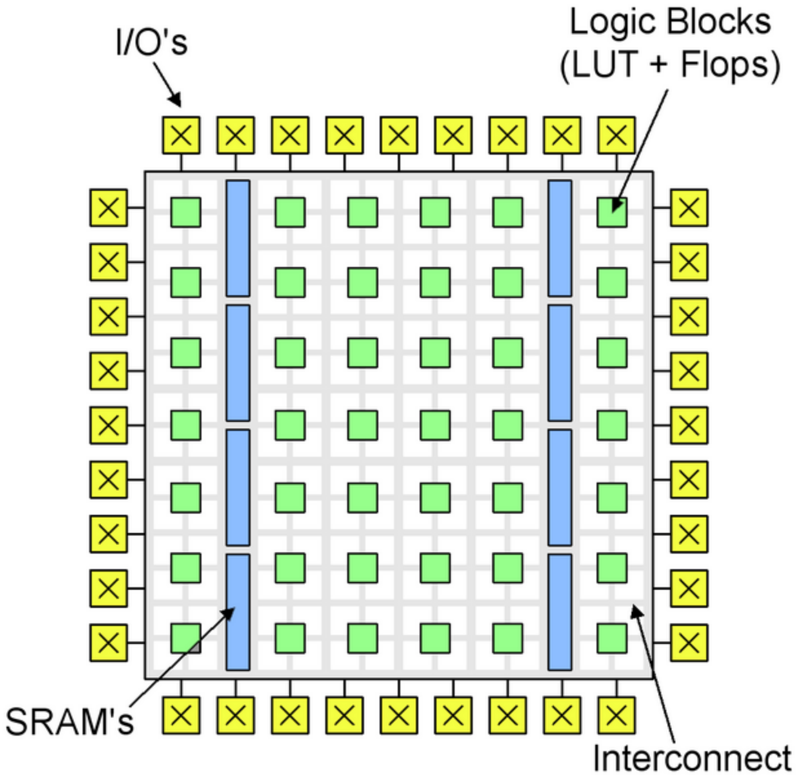
```
library IEEE;
use IEEE.std_logic_1164.all;

entity logic_c is
      port (x1, x2, x3, x4 : in bit;
            f: out bit);
end logic_c;

architecture behavior of logic_c is
signal S1, S2 : bit;
begin
      S1 <= x1 OR x2;
      S2 <= x3 AND x4;
      F <= S1 XOR S2;
end logic_c;
```



5.6 Information for Lab Session 1

To use the Vivado software to design specific logic functions and then program the ZedBoard (with FPGA Zynq chip) and validate the results

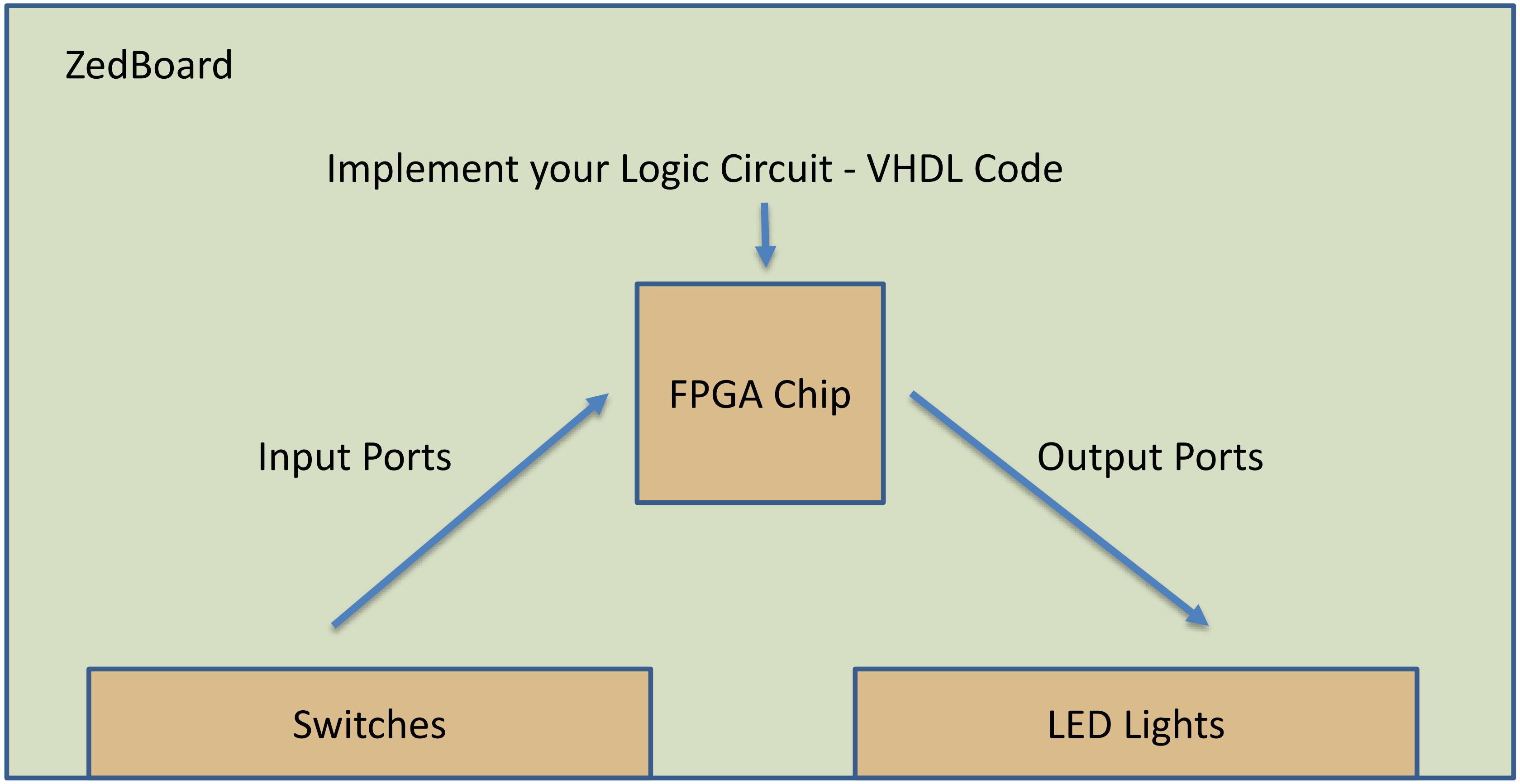




Typical FPGA layout

ZedBoard – Device Interface

- Use switches as inputs and LED lights as outputs
- Lab Session 1: Use switch to control LED (Output = Input)



Step 1: Design Source File (exp01.vhd)

- Design the logic function in this file (Page 8 to 11)

```
entity exp01 is
        Port (
            sw: IN BIT;         switch for input
            led: OUT BIT        led for output
        );
end exp01;


architecture Behavioral of exp01 is
begin
        -- Put your own code here
end Behavioral;
```

Step 2: Simulation Testbench

- To check whether the VHDL perform as we expected
- Create a testbench file (exp01_tb.vhd) (Page 11 to 13)
- Let us take a look at the codes on Page 12 – 13 Line by Line

```
entity exp01_tb is

end exp01_tb;
```

No input or output!!! Because we are doing simulation no physical connections to any device

Step 2: Simulation Testbench

```
architecture Behavioral of exp01_tb is

-- component declaration

component exp01 is

        Port (

            sw: IN BIT;

            led: OUT BIT

        );

end component;
```

- In architecture, we first see the declaration of a component
- Component: Sub-module/circuit to be called
- In this simulation, we call the circuit that we want to test (exp01)
- And specify the inputs and outputs of the component

Step 2: Simulation Testbench

```
-- signal declaration

signal sw: BIT;

signal led: BIT;

begin

exp01_inst: exp01

        port map (

                sw => sw,

                led => led

        );
```

- Then, we declare the internal signals that we use for the simulation (also a switch and an led)

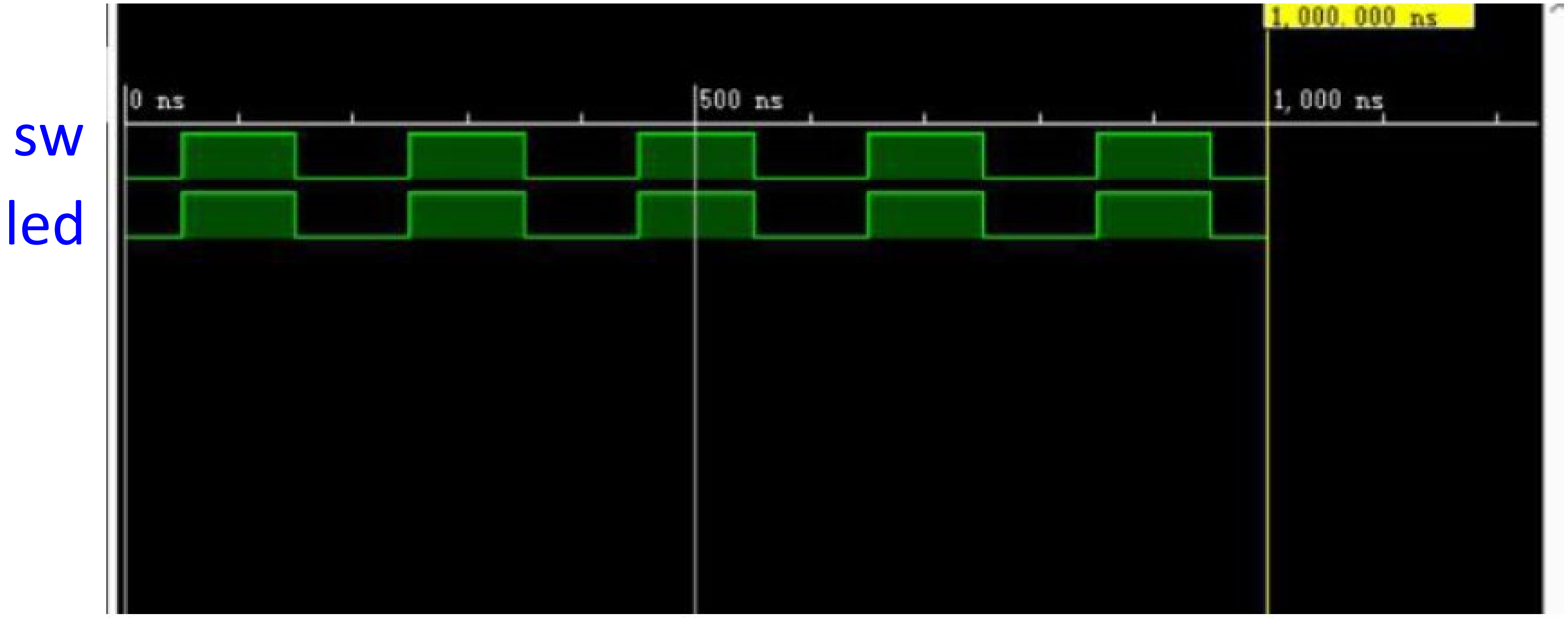
- We map the component input sw to the internal signal sw
- and the component output led to the internal signal led

Step 2: Simulation Testbench

```
simgen: process

begin

        sw <= '0';

        wait for 50ns;

        sw <= '1';

        wait for 100ns;

        sw <= '0';

        wait for 50ns;

end process;

end Behavioral;
```

- Begin the process (sequential statements)
- sw will be ‘0’ for 50 ns
- and then change to ‘1’ for 100 ns
- and then change back to ‘0’ for 50 ns

Step 2: Simulation Testbench



Step 3: Synthesis and Implementation

- transfer the VHDL codes to the wire connections and implement them onto the FPGA chip
- tell the program which switch or LED to be used
- We choose switch SW0 with pin number F22
- and LED LD0 with pin number T22

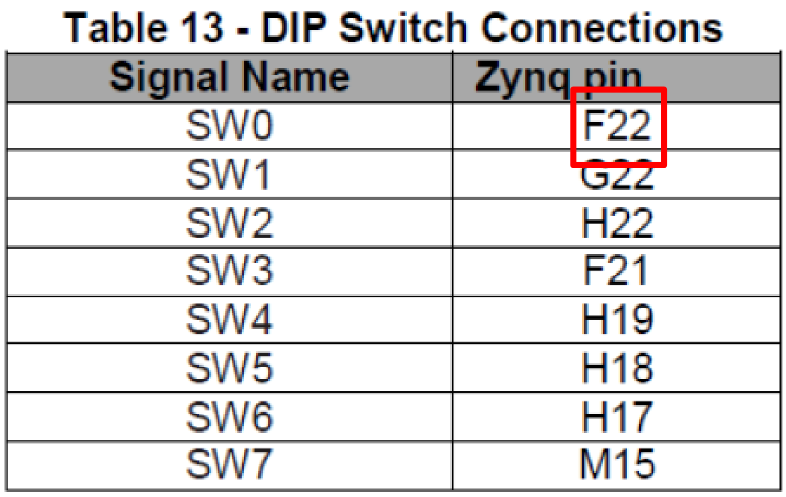


Table 13 - DIP Switch Connections

Signal Name	Zynq pin
SW0	F22
SW1	G22
SW2	H22
SW3	F21
SW4	H19
SW5	H18
SW6	H17
SW7	M15

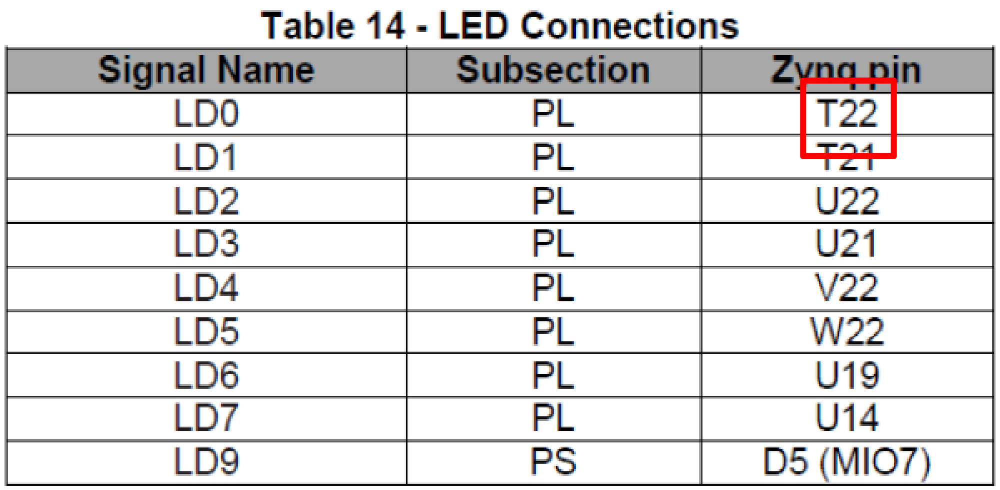


Table 14 - LED Connections

Signal Name	Subsection	Zynq pin
LD0	PL	T22
LD1	PL	T21
LD2	PL	U22
LD3	PL	U21
LD4	PL	V22
LD5	PL	W22
LD6	PL	U19
LD7	PL	U14
LD9	PS	D5 (MIO7)

Step 3: Synthesis and Implementation

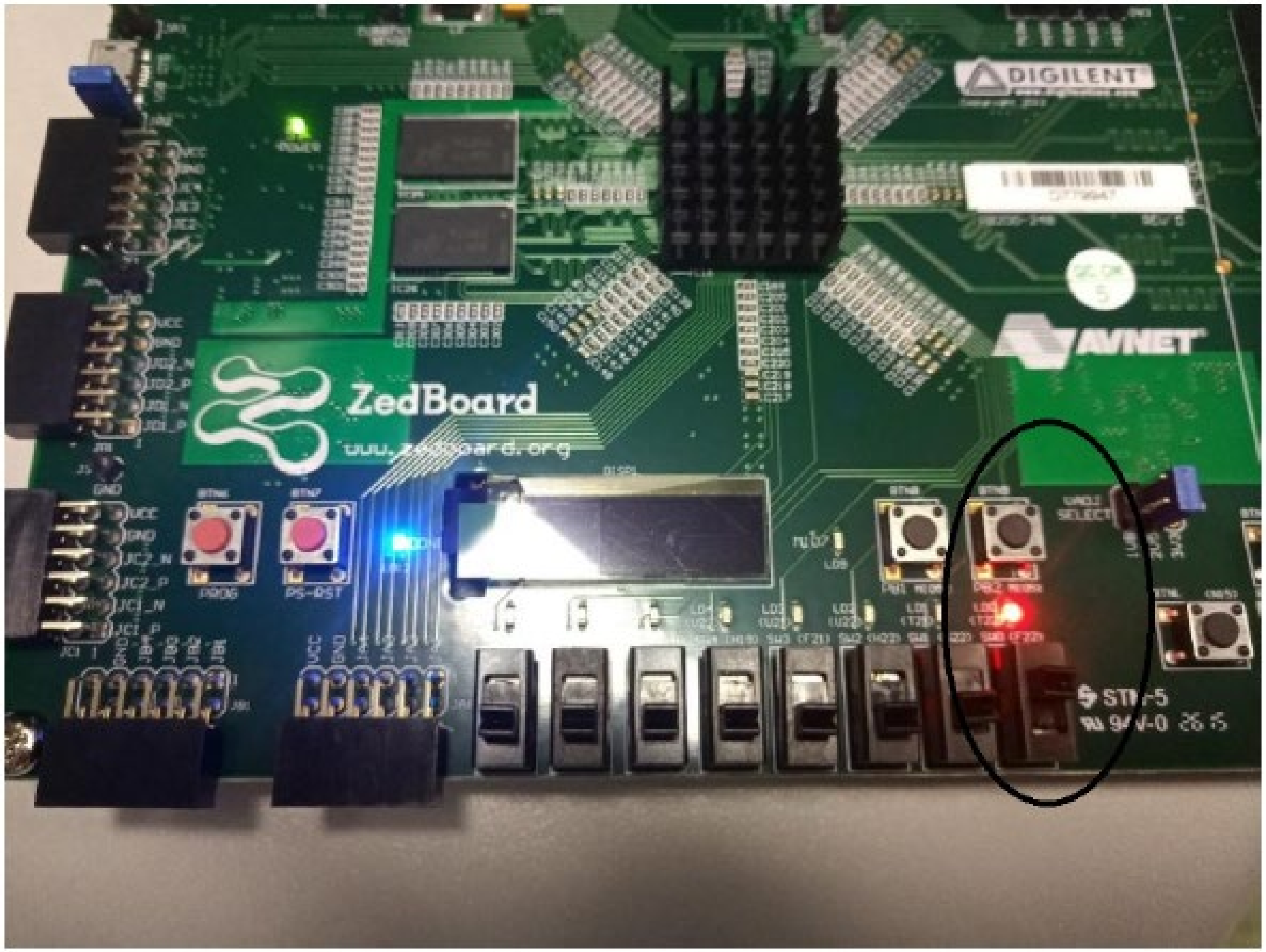
- Create a constrains file to assign them

```
set_property PACKAGE_PIN T22 [get_ports {led}]

set_property IOSTANDARD LVCMOS33 [get_ports {led}]

set_property PACKAGE_PIN F22 [get_ports {sw}]

set_property IOSTANDARD LVCMOS33 [get_ports {sw}]
```

- Then, we can run synthesis, implementation and generate the bitstream files

Step 4: Configure the hardware

- Download the bitstream file to the FPGA and check the function



Example of Verilog

- Design an inverter $Y = \bar{A}$

```
module jinverter (y,a);
        output y;
        input a;

        assign y=~a;

endmodule
```