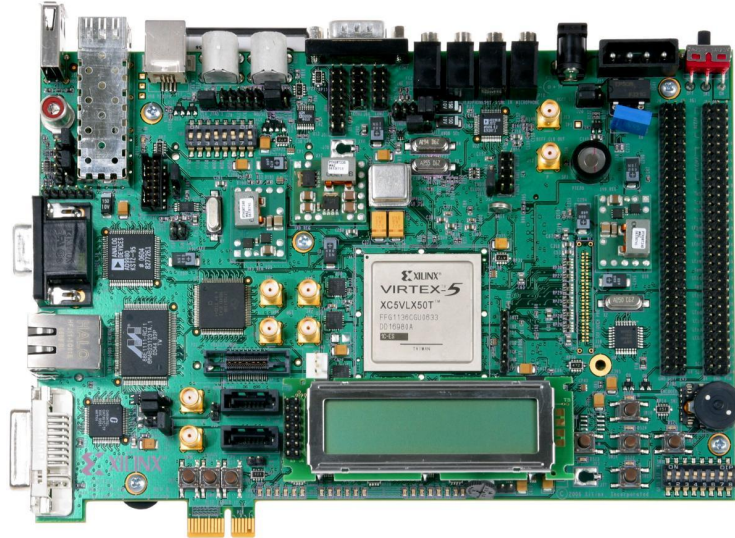


Electronics Club

FPGA AND HDL

WHAT IS AN FPGA

FPGA or field programmable gate arrays are integrated circuits similar to CPUs and other microcontrollers only difference being that the circuit inside the FPGA can be modified any number of times. The circuit inside FPGA are defined using Hardware description language or HDL. There are two major HDLs, VHDL and verilog. We will look into them today.



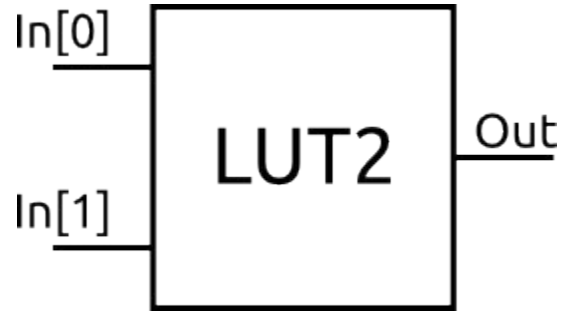
HOW DOES AN FPGA WORK?

FPGAs consist mainly of three elements: Look up tables (LUTs), flip-flops, and the routing matrix.

Look up Tables:

A LUT consists of a block of RAM that is indexed by the LUT's inputs. The output of the LUT is whatever value is in the indexed location in its RAM.

A 2-input LUT:

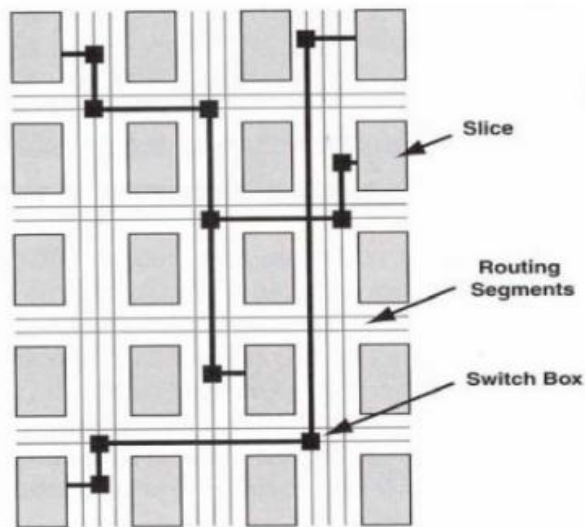


Address (In[1:0])	Output (Value Stored at that Address)
00	0
01	0
10	0
11	1

ROUTING MATRIX

FPGA Routing Matrix and Global Signals

21



FPGA signal Routing

SO WHY FPGA?

The main reason why FPGAs are used is because of parallel processing. A microprocessor performs any given task sequentially, while an FPGA can be reconfigured to perform a very specific task. This makes it faster than any microprocessor.

SOFTWARE PROGRAMMING LANGUAGE

Let us first see how a software programming language works. A software programming language executes in a sequence i.e. the code runs from top to bottom unless of course you use go-to, loop etc. But the important point is in a software programming language, only one line is executed at a time.

The code that you write is compiled and is converted into simpler instructions which the processor can understand. These instructions are then fed into the processor which then processes them.

The disadvantage of this process is that the processor has to process all the different types of information. This means that it can't be optimised to do a certain process faster.

HARDWARE DESCRIPTION LANGUAGE

There are two major hardware description languages- Verilog and VHDL similar to C. But unlike software programming languages, these do not execute sequentially, but parallelly.

We will explain this in more detail.

VERILOG

File Edit View Project Source Process Tools Window Layout Help

Design

View Implement Simulat

Hierarchy

- fpga_presentation
 - xc5vlx110t-2ff1136
 - test (test.v)

No Processes Running

Processes: test

- Design Summary/Reports
- Design Utilities
- User Constraints
- Synthesize - XST
- Implement Design
- Generate Programming File
- Configure Target Device
- Analyze Design Using Ch...

```
3 // Company:
4 // Engineer:
5 //
6 // Create Date:      17:07:30 10/07/2017
7 // Design Name:
8 // Module Name:      test
9 // Project Name:
10 // Target Devices:
11 // Tool versions:
12 // Description:
13 //
14 // Dependencies:
15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 ///////////////////////////////////////////////////////////////////
21 module test(
22
23 );
24
25
```

test.v* Design Summary

Warnings

Console Errors Warnings Find in Files Results

SAMPLE VERILOG CODE

```
1  `timescale 1ns / 1ps
2
3  module and_gate(
4  input wire A,
5  input wire B,
6  output wire C
7      );
8      assign C = A&B;
9  endmodule
10
```

ANALYZING THE CODE

Module : Modules are simple elements in verilog with inputs and outputs. Instances of a module can be created in the main code.

In this case, the module take A and B as inputs and outputs C.



ANALYZING THE CODE

Wires and Registers : There are two new data-types in Verilog; wires and registers. A wire acts like a physical wire; it connects two points in the circuit. A register is an element that is used to store memory. A register is able to store only one bit

Assign : It creates a connection between wire *C* and *A&B*. Note that since this is not a sequential code, whenever the inputs *A* and *B* are changed, the change will also be reflected in *C*.



CREATING A SIMPLE SWITCH

```
1  `timescale 1ns / 1ps
2
3  module led_switch(
4  input button,
5  output led
6      );
7  reg led_reg = 1'b0;
8  assign led = led_reg;
9  always@(posedge button) begin
10     led_reg = !led_reg;
11 end
12 endmodule
13
```


ANALYZING THE CODE

Initializing register value : The register `led_reg` is initialized with the value `1'b1`. That is, the register is initialized with a one bit binary number of value 1.

The Always block : The always block is executed when the statement inside the parenthesis is true. Here, the statement “*posedge button*” will be true whenever a positive edge of the wire *button* is encountered. In that case, the register `led_reg` will flip its state.



.UCF FILES

Now that we were able to simulate the code for a switch, we would like to actually implement it on an FPGA.

How do we tell the FPGA that the wire *button* corresponds to an actual push-button?

How do we tell the FPGA that the wire *led* should be connected to a physical LED present on the board?

The task of connecting the input/outputs of a module to the GPIO pins/onboard LEDs/pushbuttons is done by the .ucf file.

.UCF FILES

The syntax is: `NET "name" LOC = "pin_number";`

```
1 NET "button" LOC = "U8"; //GPIO North
2 NET "led" LOC = "AF13"; //LED North
```

Here, the pin numbers for the onboard LED and pushbutton are *U8* and *AF13*

CREATING A CLOCK

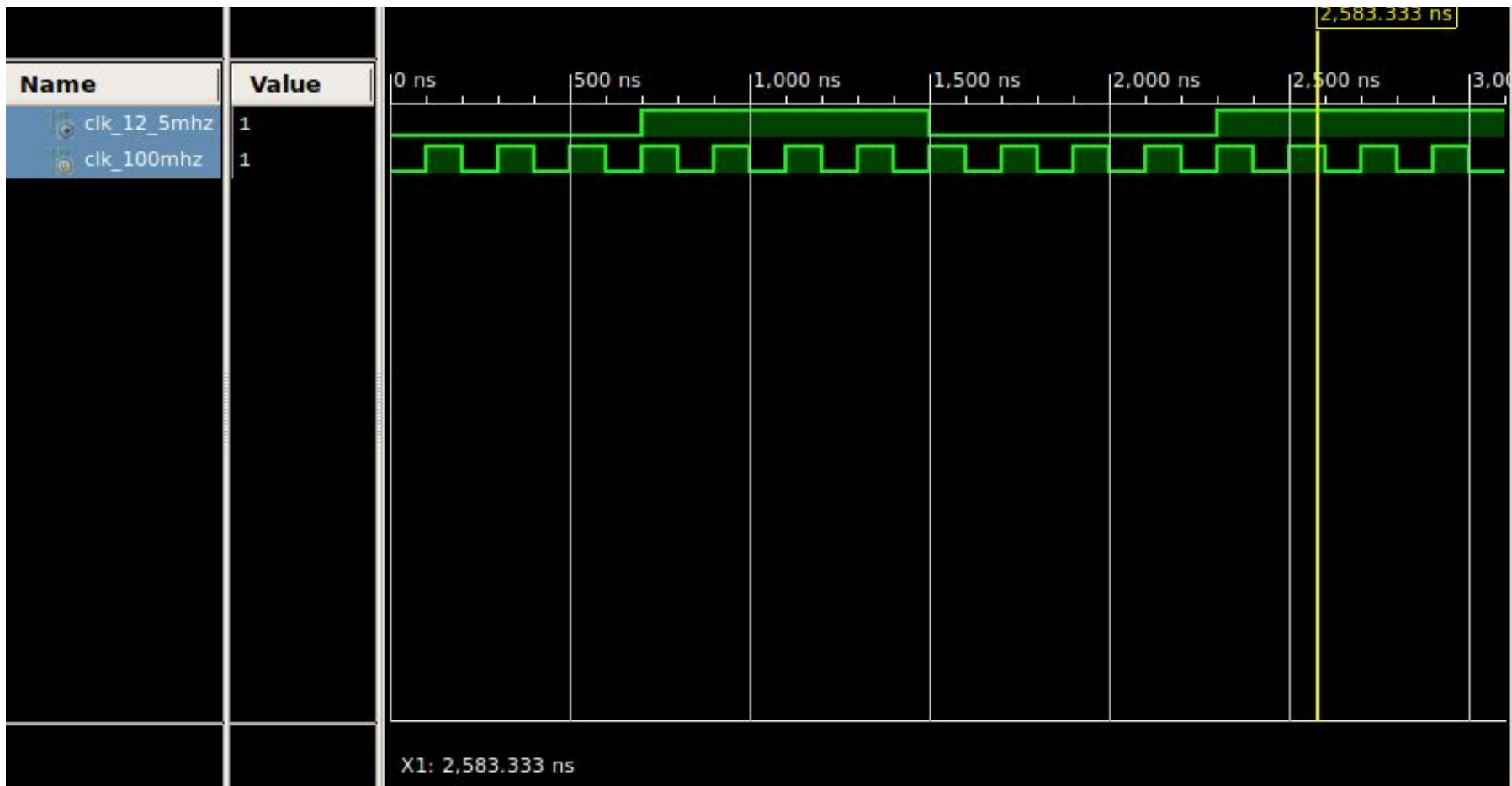
The FPGA board usually has a clock of frequency 100 MHz or 50 MHz. In the following example, we will see how to create a clock of much lower frequency.

```
1  `timescale 1ns / 1ps
2
3  module clkgen(
4  input clk_100mhz,
5  output clk_12_5mhz
6      );
7  reg [2:0] counter = 3'b0;
8  assign clk_12_5mhz = counter[2];
9  always@(posedge clk_100mhz) begin
10      counter = counter + 1'b1;
11  end
12  endmodule
13  |
```

ANALYZING THE CODE

Array of registers : We can create an array of registers by using the following syntax: `reg [length - 1 : 0] array_name.`

Logic: After every clock pulse of the 100 MHz clock, the register *counter* is incremented by one bit. Since the register is three bits long, after 8 such clock pulses, the most significant bit of the register gets flipped. This is connected to the wire `clk_12_5`. Hence, the frequency of this clock is 12.5 MHz.



DATA STRUCTURES IN VERILOG

RANDOM ACCESS MEMORY

RAM is a basic data structure which can be configured to have multiple inputs and outputs. A RAM supports 2 functions, Read and write. A RAM takes an address as one argument. The data at this address can be read or it can be overwritten using write enable. Since any address can be accessed at any point in time, hence the name Random Access Memory.

```
1  module rammod (
2    input clk,
3    input we,
4    input [4:0] addr,
5    input [3:0] din,
6    input [3:0] dout
7  );
8
9    reg [3:0] ram [31:0];
10   reg [3:0] dout;
11   always @(clk)
12   begin
13     if (we)
14       ram[addr] <= din;
15   else
16     dout <= ram[addr];
17   end
18   endmodule
```

VHDL

VHSIC Hardware **D**escription **L**anguage

VHDL

VHSIC Hardware Description Language

Purpose of VHDL

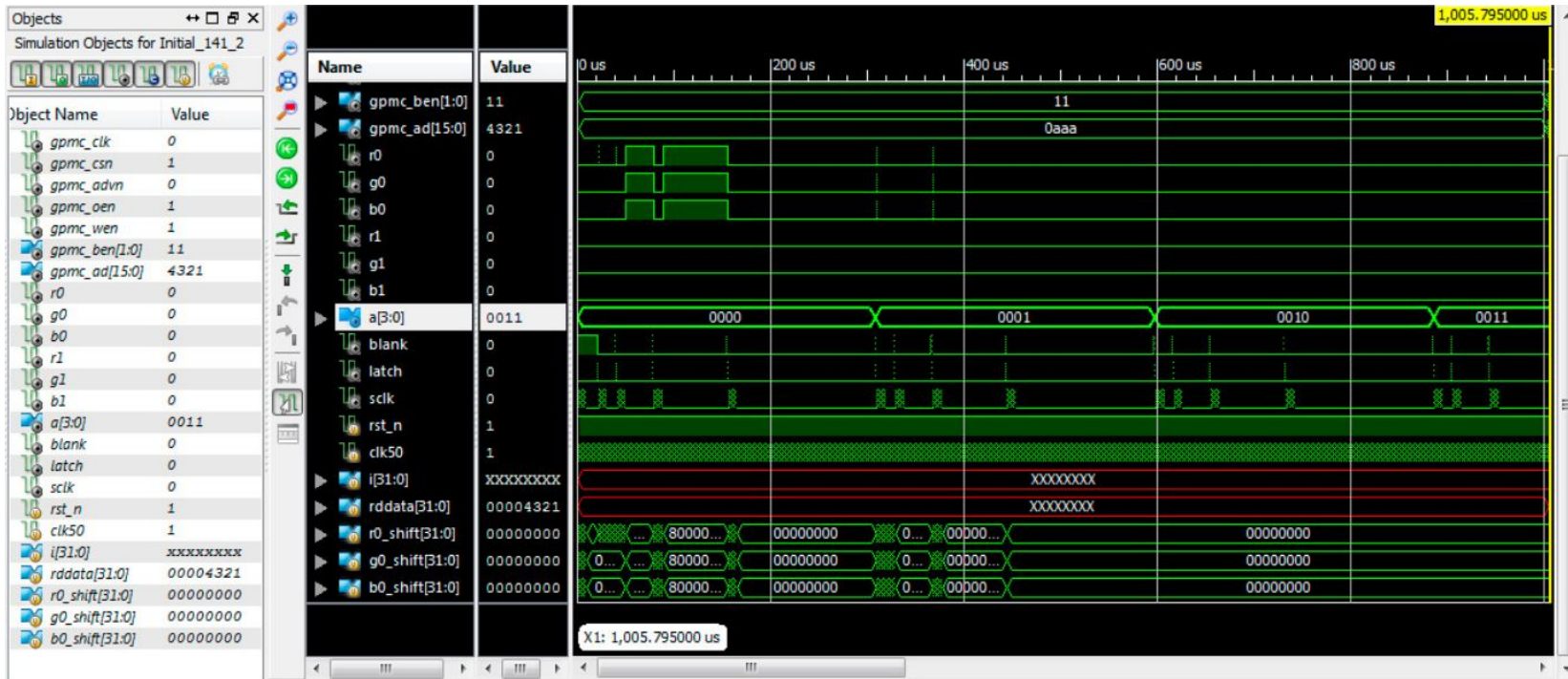
- Java** is a programming language that you can use to create applications on your computer.
- C** Although numerous computer languages are used for writing computer applications, the computer programming language, C.
- HTML** is used for web-pages .
- HDL's** are used for describing a hardware.

WHY TO USE VHDL (or ANY HARDWARE DESCRIPTION LANGUAGE)

- When designing a circuit or system we , directly jumping to the synthesis part and then testing the circuit can prove expensive.
- Hence VHDL provides use a means to observe the behaviour of the system i.e. verify (simulated) before synthesis.
- VHDL (any hardware description language) can be used to represent the characteristics of a real hardware circuit , which is difficult with languages like C,C++.

Testbench: To test our system

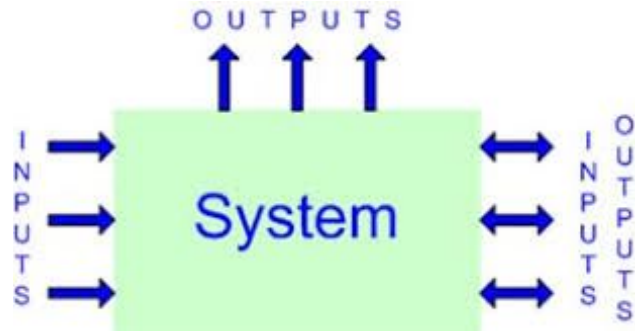
Here it shows the input and output signals.
We verify our logic circuit.



HDLs vs. Software Languages

In VHDL :Concurrent (parallel) Statements
vs.
Sequential Statements

Every hardware system can be described by its inputs , outputs and it's behaviour.

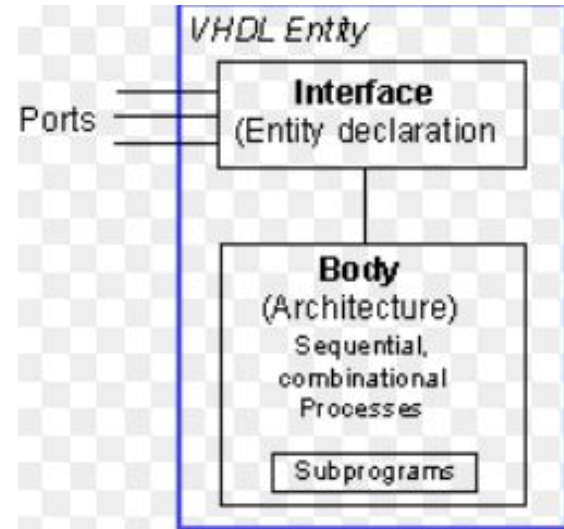


These specifications can also be seen in VHDL design.

VHDL design

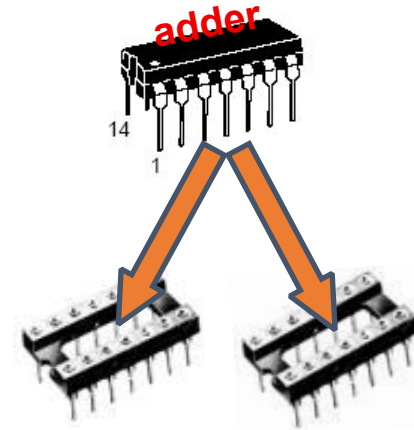
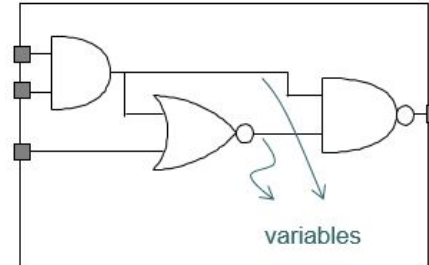
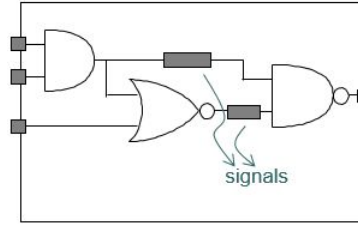
- *Two components to the description:*

1. **ENTITY:** *The interface to the design: **entity** declaration i.e inputs, outputs,*
2. **ARCHITECTURE:** *The internal behaviour of the design: **architecture** construct.*



Other Attribute

1. Signals
2. Variables
3. Component (socket mechanism)
4. Process.
5. The Sensitivity list.
etc.



Signals and variables.

Signal :It is a physical signal (you can think of it like a piece of **wire**).
A signal assignment takes effect only after a certain **delay** (the smallest possible delay is called a “**delta time**”).

Variable : Assignment to variables are scheduled **immediately** (the assignment takes effect immediately) Typically, variables are used as a local **storage mechanism**, visible only inside a process

Example to explain difference between signal and variable

```
architecture bad of logic is
  signal a_or_b : std_logic;
begin
  logic_p: process(a,b,c)
  begin
    a_or_b <= a or b;
    z <= a_or_b and c;
  end process;
end bad;
```

let's assume we
enter process at
time "t".

a_or_b is
cheduled
 $t + \Delta$


this is the value at t
instead of the updated
value (at " $t + \Delta$ ")

Example to explain difference between signal and variable

How to fix this ?

Use variables :

```
architecture good of logic is
  variable a_or_b : std_logic;
begin
  logic_p: process(a,b,c)
  begin
    a_or_b := a or b;
    z <= a_or_b and c;
  end process;
end good;
```



Use variables for
immediate operation

2. Process

Sensitivity list

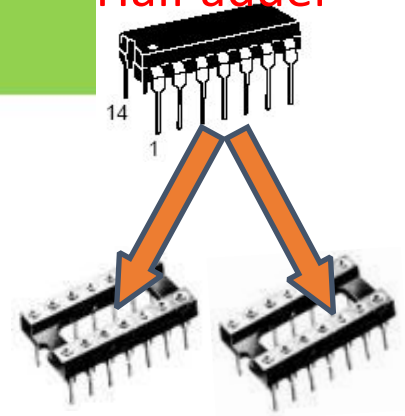


```
process (a)
variable c;
begin
  c:= a or b;
  z <= c;
end process;
```

since b is not in the sensitivity list, when a change occurs in 'b' the process is not invoked, so the value of z is not updated.
(still "remembering" the old value of z)

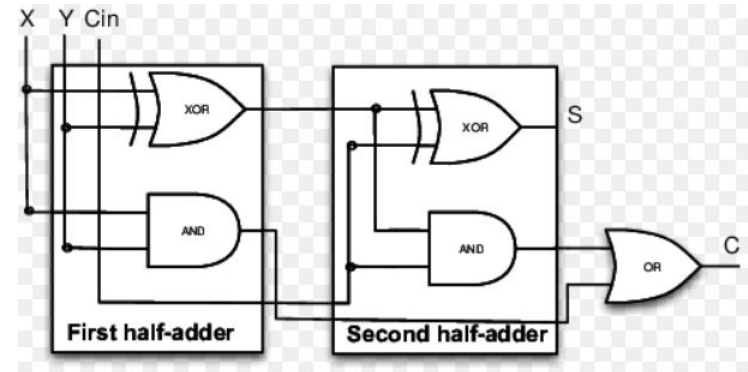
3. Component

Half adder



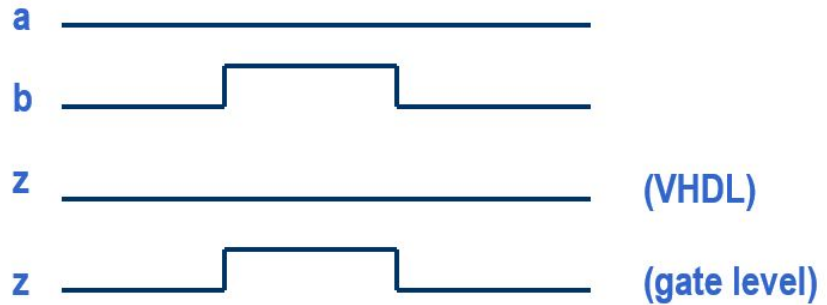
A component is a **small system** in itself. Once the behaviour of the system is determined we can use as many time as we want in creating a complex circuit

Example : **full-adder** using two **half-adders**



Incomplete sensitivity list effect

```
process (a)
variable c;
begin
  c:= a or b;
  z <= c;
end process;
```



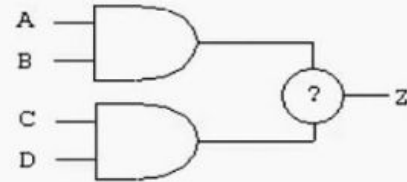
Multiple Drivers

In C

```
int A,B;  
int C,D;  
Z=A & B;  
Z=C & D;
```

IN VHDL

```
architecture CONCURRENT of MULTIPLE is  
    signal Z, A, B, C, D : std_logic;  
begin  
    Z <= A and B;  
    Z <= C and D;  
end CONCURRENT;
```



Consider this code in this the signal is driven by two signals rather than updating the value of Z.

Do not “read” and “write”
a signal at the same time !!!

Assignment operator

IN VHDL

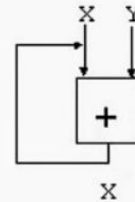
In C

X,Y are variables.

$X = X + Y;$

$X \leq X + Y;$

in hardware:



Think hardware!