D D M M Y Y Y Y
1 4 0 8 2 0 1 7

PREFACE I : Introduction to digital FPGA design flow.

Theory :-

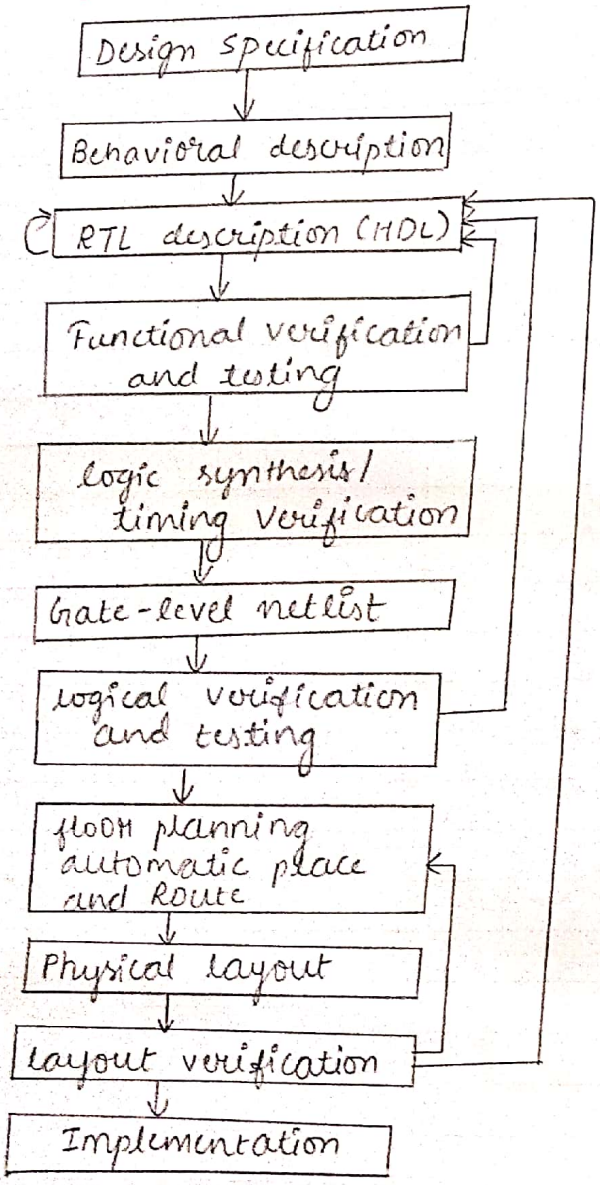
A typical design flow for designing VLSI IC circuits is as shown in figure 1.

Specifications describe abstractly the functionality, interface and overall architecture of the digital circuit to be designed. A behavioral description is then created to analyze the design in terms of functionality, performance, compliance to standards and other high-level issues. Behavioral descriptions are often written with HDLs.

The behavioral description is manually converted to an RTL description in an HDL. The designer has to describe the dataflow that will implement the desired digital circuit. From this point onwards, the design process is done with the assistance of EDA tools.

Logic synthesis tools convert the RTL description to a gate-level netlist. A gate-level netlist is a description of the circuit in terms of gates and connections between them. Logic synthesis tools

Digital Design flow Diagram



D D M M Y Y Y Y

ensure that the gate-level netlist meets timing, area and power specifications. The gate-level netlis is input to an automatic place and Route tool, which creates a layout. The layout is verified and then fabricated on a chip.

Thus, most digital design activity is concentrate on manually optimizing the RTL description of the circuit. After the RTL description is frozen, EDA tools are available to assist the designer in further processes. Designing at the RTL level has shrunk the design cycle times from years to a few months. It is also possible to do many design iterations in a short period of time.

Behavioral synthesis tools have begun to emerge recently. These tools can create RTL descriptions from behavioral or algorithmic descriptions of the circuit. these tools mature, digital circuit design will becor Similar to high-level computer programming. Designe will simply implement the algorithm in an HDL at a very abstract level. EDA tools will help the designer convert the behavioral description to a final IC chi

DD MM YYYY

PREFACE II :-

procedure for xilinx vivado 2015.1 to complete FPGA flow (simulation, synthesis and implementation steps)

CREATING A PROJECT :

1) open vivado 15.1 software.
2) Go to file menu and close any previously opened project if any and then select new project.
3) Enter the project name and location and select the project as RTL project and click next.
4) Select the board as Basys-3 board.
5) And click on the finish button to create a new project with required name.) → Wrong
4) click the '+' button and select add files. click OK to close file browser.
5) set the target language to verilog to define the language of the netlist generated by vivado synthesis.
6) set the simulation language to verilog to define the language required by the logic simulator. click next and skip past and add existing IP page.
7) On the default part page, choose boards and select Basys-3 board
8) click finish and close the New project summary page to create the project
9) vivado IDE opens the project in the default layout.

The VIVADO IDE includes a context sensitive text editor to create and develop RTL sources, constraints files and TCL scripts.

RUNNING BEHAVIOROL SIMULATION

The vivado IDE integrates the vivado simulator which enables one to add and manage simulation sources in the project. Behavioral simulation can be run on RTL sources, prior to synthesis.

1) Click the Run simulation command in the flow navigator, then click the run behavioral simulation in the sub-menu.
2) Force constant is used to force the selected object to a constant value.

SYNTHESIZING AND IMPLEMENTING THE DESIGN:-

1) In flow navigator, click the run synthesis button and wait for the task to complete. The progress bar in the upper-right corner of the vivado IDE, indicates that the run is in progress.
2) After synthesis has completed, the synthesis comp-leted dialog box prompts to choose the next step.
3) Select Run Implementation and click OK. The implementation process is launched and placed into a background process after some initialization.
4) Once implementation is completed, the implementation completed dialog box prompts you to choose the next step.
5) Select open implementation design option from the dialog box. After implemented design has loaded, you can see implementation results in the device window.

After the design has been placed and routed, a timing report can be generated to verify that all the timing constraints are met.

If there are timing problems, the RTL source files or design constraints can be revi-sited to address any of the problems.

GENERATING A BITSTREAM FILE:

with IO standard constraints defined for all of the I/O ports and the logic of the design placed with assigned LOC's, a bitstream can be generated.

1) In the flow navigator, under the program and debug section, click generate bitstream.
2) After the bitstream generates, click OK in the bitstream generation completed dialog box to view the reports from the command.

D D M M Y Y Y Y

Syntax of a module :-

```
module <module_name>(<module_terminal_list>);
---
---
<module internals>
---
---
endmodule
```

PREFACE III :- Verilog Syntax

1) Module.

A module is the basic building block in verilog. A module can be an element or a collection of lower-level design blocks. Typically, elements are grouped into modules to provide common functionality that is used at many places in the design. A module provides the necessary functionality to the higher-level block through its port interface (inputs and outputs), but hides the internal implementation. This allows the designer to modify module internals without affecting the rest of the design.

In verilog, a module is declared by the keyword "module". A corresponding keyword "endmodule" must appear at the end of the module definition. Each module must have a module_name, which is the identifier for the module, and a module_terminal_list, which describes the input and output terminals of the module.

"Initial" syntax:-

```
module module_name < module_terminal_list >;

initial
  --- // single statement ; does not need to be grouped.
initial
begin
  --- // multiple statements ; need to be grouped.
  --
  --
end

endmodule
```

"Always" syntax :-

```
module module_name < module_terminal_list >;

 always
  ---- // executes the statement continuously in a
         looping fashion.
endmodule
```


D D M M Y Y Y Y

2) Initial statement and Always statement.

⇒ Initial statement

All statements inside an initial statement constitute an initial block. An initial block starts at time 0, executes exactly once during a simulation and then does not execute again. If there are multiple initial blocks, each block starts to execute concurrently at time 0. Each block finishes execution independently of other blocks.

Multiple behavioural statements must be grouped, typically using the keywords begin and end. The initial blocks are typically used for initialization, monitoring, waveforms and other processes that must be executed only once during the entire simulation run.

⇒ Always statement.

All behavioral statements inside an always statement constitute an always block. The always statement starts at time 0 and executes the statements in the always block continuously in a looping fashion. This statement is used to model a block of activity that is repeated continuously in a digital circuit. An example is a clock generator module that toggles the clock signal every half cycle.

D D M M Y Y Y Y

Syntax for conditional statements:-

```
// type 1 conditional statement. No else statement
// statement executes or does not execute

if (<expression>) true_statement;

// type 2 conditional statement. one else statement
// either true_statement or false_statement is
   evaluated.

if (<expression>) true_statement;
else  false_statement;

// type 3 conditional statement. Nested if-else-if
// choice of multiple statements. only one is executed.

if (<expression1>) true_statement1;
else if (<expression2>) true_statement2;
else if (<expression3>) true_statement3;
else  default_statement;

case(expression)
 alternative1 : statement1;
 alternative2 : statement2;
    .
    .
    .
 default : default_statement;
endcase
```

3) Conditional statements - if, if-else, case.

Conditional statements are used for making decisions based upon certain conditions. These conditions are used to decide whether or not a statement should be executed.

⇒ If and else are keywords used for conditional statements.

In the syntax, <expression> is evaluated. If it is true the true-statement is executed. However if it is false or ambiguous false-statement is executed if there is an "else" associated with "if". Otherwise, next instruction will be executed. "else if" is used when multiple expressions are to be evaluated for execution of various statements associated with it.

⇒ case statement :- The case statement is used if there are many alternatives. The expression is compared to the alternatives in the order they are written. For the first alternative that matches, the corresponding statement or block is executed. If none of the alternatives matches, the default-statement is executed. The default-statement is optional.

D D M M Y Y Y Y

Syntax of loop statements:-

```
module  module_name(module_terminal_list);

initial
 begin
 while (<expression>)
  ...          //statement is executed untill expression is true
 end

integer
initial
for (<initialization>;<terminating condition>;<procedural assignments>)


initial
 begin
 repeat(<number>)
  ..
 end

initial
 forever (<statement>);
```

4) Loop statement - while, for, repeat, forever.

There are four types of looping statements in verilog: while, for, repeat and forever. All looping statements can appear only inside an initial or always block.

⇒ while loop - The while loop executes until the while-expression is not true. If the loop is entered when the while-expression is not true, the loop is not executed at all.

⇒ for loop - The for loop contains 3 parts
i) an initial condition ii) A check to see if the terminating condition is true iii) A procedural assignment to change value of the control variable.

⇒ repeat loop - The repeat construct executes the loop a fixed number of times. A repeat construct must contain a number, which can be a constant, a variable or a signal value. However, if the number is a variable or signal value, it is evaluated only when the loop starts and not during the loop execution.

⇒ Forever loop - The loop is equivalent to a while loop with an expression that always evaluates to true. A forever loop can be exited by use of disable statement.

DD MM YYYY
09 08 2017

Experiment no. 1
Basic gates.

Aim :-

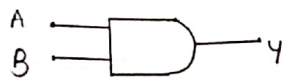
Describing functionality of logic gates using Verilog language and to perform behavioral verification using Xilinx simulator and testing on Artix 7 FPGA.

Hardware and software Requirement:

Sl.NO	Description	specification
1.	FPGA	Artex-7 on Basys-3 board XC7A35T-1CPG236C
2.	CAD TOOL (synthesis, simulation, Implemation)	VIVADO 15.1 (xilinx)
3.	Computer.	OS: Ubuntu 14.04 LTS PROCESSOR: Pentium(R) dual-core CPU E5700@3.4GHz x 2 memory: 3.8GiB

DD MM YYYY

1) AND gate



$Y = A \cdot B$

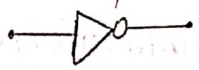
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

2) OR gate

$Y = A + B$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

3) NOT gate



$Y = \overline{A}$

A	Y
0	1
1	0

4) NAND gate

$Y = \overline{A \cdot B}$

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

Theory :-

A logic gate is an elementary building block of a digital circuit. The logic gates are used to implement a boolean function; that is, it performs a logical operation on one or more binary inputs and produces a single binary output.

Logic gates are classified as
1) Basic gates - AND, OR and NOT
2) Universal gates - NAND and NOR
3) Special/derived gates - XOR and XNOR.

1) Basic gates - These are the basic building blocks of the digital circuit. The boolean expression, truth table etc of these gates are given in figure 1, figure 2 and figure 3.

2) Universal gates - These gates can be used to implement any boolean function. Hence they are called universal gates.

3) Derived gates - These gates are made of simple combinations of the basic gates. But still they have a specific logic symbols as shown in figure 6 and 7.

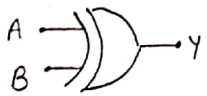
DD MM YYYY

5) NOR gate

$Y = \overline{A+B}$

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

6) XOR gate

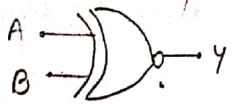


$Y = A \oplus B$

$Y = \bar{A}B + A\bar{B}$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

7) XNOR gate



$Y = \overline{A \oplus B}$

$Y = \bar{A}\bar{B} + AB$

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

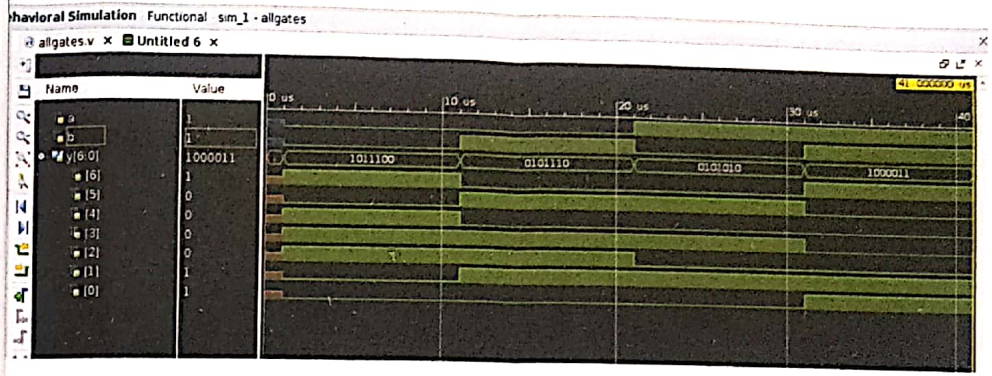
Verilog codes:-

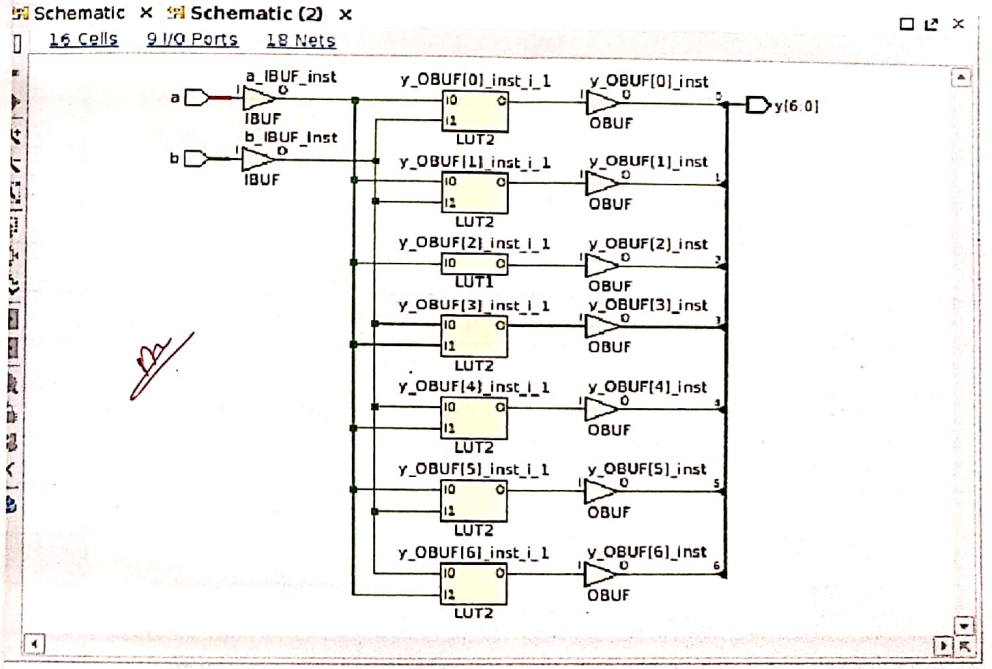
1) Data flow modeling

```
module gates_dataflow (A, B, Y);
input A, B;
output [6:0] Y;
assign Y[0] = A & B;
assign Y[1] = A | B;
assign Y[2] = ~A;
assign Y[3] = ~(A & B);
assign Y[4] = ~(A | B);
assign Y[5] = A ^ B;
assign Y[6] = ~(A ^ B);
endmodule.
```

2) Behvioral modeling

```
module gates_behavioral (A, B, Y);
input A, B;
output [6:0] Y;
reg [6:0] y;
always @(a or b)
begin
Y[0] = A & B;
Y[1] = A | B;
Y[2] = ~A;
Y[3] = ~(A & B);
```





and synthesized using VIVADO software and it is implemented on Basys-3 board.

Index (06)

DD MM YYYY

```
Y[4] = ~(A & B);
Y[5] = ~(A | B);
Y[6] = A ^ B;
Y[7] = ~(A ^ B);
end
endmodule
```

3) Structural modeling

```
module gates_structural (A, B, Y);
input A, B;
output [6:0] Y;
and  x1 (Y[0], a, b);
OR   x2 (Y[1], a, b);
not  x3 (Y[2], a);
nand x4 (Y[3], a, b);
NOR  x5 (Y[4], a, b);
XOR  x6 (Y[5], a, b);
XNOR x7 (Y[6], a, b);
end module.
```

Result :-

The functionality of logic gates is simulated and synthesized using VIVADO software and it is implemented on Basys-3 board.

Index 06

16:4 Encoder using 8:3 Encoder.

Particulars of the Experiments Performed
CONTENTS

Sl. No.	Date	Particulars	Page No.	Marks
01.	14/8/2017	Preface I : Introduction to digital FPGA design flow	01-03	
02.	14/8/2017	Preface II : Procedure for Xilnix VIVADO 2015.1 to complete FPGA flow	04-05	
03.	14/8/2017	Preface III : Verilog Syntax	06-09	
04.	14/8/2017	Preface IV : VHDL Syntax	10-12	
05	9/8/2017	Basic gates	13-16	06
06.	9/8/2017	2 to 4 decoder	17-20	06
07.	16/8/2017	Binary to gray conventon	21-25.	07
08.	23/8/2017	8:1 multiplexer	26-30	07
09.	23/8/2017	8:3 encoder	31-37	07.
10.	6/9/2017	1:4 demux	38-41	07
11.	6/9/2017	comparator	42-47	07
12.	13/9/2017	Full adder	48-51	07
13.	20/9/2017	ALU	52-54	07
14.	21/9/2017	SR flip-flop	55-57	07
15.	21/9/2017	D flip-flop	58-60	
16.	21/9/2017	T flip-flop	61-63	
17.	21/9/2017	JK flip-flop	64-66	
18.	11/10/2017	Binary counter with synchronous CLEAR	67-68	09
19.	11/10/2017	Binary counter (Asynchronous CLEAR)	69-70	
20.	11/10/2017	BCD counter (Synchronous CLEAR)	71-72	
21.	11/10/2017	BCD counter (Asynchronous CLEAR)	73-74	