

PERFORMANCE ANALYSIS OF HIGHLY FAULT COVERAGE & LOW POWER PRESTO GENERATOR

Submitted in partial fulfillment of the requirements for the award of Masters
of Engineering degree in Embedded Systems and IoT

by

SAIKUMAR GALLA (Register No. 39108001)



**DEPARTMENT OF ELECTRONICS AND
COMMUNICATION ENGINEERING**

SCHOOL OF ELECTRICAL AND ELECTRONICS

SATHYABAMA

**INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)**

Accredited with Grade "A" by NAAC

JEPPIAAR NAGAR, RAJIV GANDHI SALAI, CHENNAI - 600 119

APRIL - 2021



SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)

Accredited with Grade "A" by NAAC

JEPPIAAR NAGAR, RAJIV GANDHI SALAI, CHENNAI - 600 119

www.sathyabama.ac.in



DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

BONAFIDE CERTIFICATE

This is to certify that this Project Report is the bonafide work of **Mr. GALLA SAIKUMAR (39108001)** who carried out the project entitled 'PERFORMANCE ANALYSIS OF HIGHLY FAULT COVERAGE & LOW POWER PRESTO GENERATOR' under my supervision from December 2020 to April 2021.

Internal Guide

Dr.T.RAVI, M.E.,Ph.D.,

Head of the Department

Dr.T.RAVI, M.E.,Ph.D.,

Submitted for Viva voce Examination held on _____

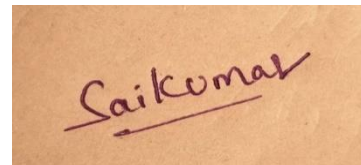
Internal Examiner

External Examiner

DECLARATION

I SAIKUMAR GALLA (39108001) hereby declare that the Project Report entitled '**PERFORMANCE ANALYSIS OF HIGHLY FAULT COVERAGE & LOW POWER PRESTO GENERATOR**' done by me under the guidance of Dr. SUMATHI M is submitted in partial fulfillment of the requirements for the award of Masters of Engineering degree in Embedded Systems and IoT.

DATE : 17.4.2021
PLACE : CHENNAI

A photograph of a piece of brown paper with the name 'Saikumar' written in dark ink. The signature is written in a cursive style and is underlined with a single horizontal line.

SIGNATURE OF THE CANDIDATE

ACKNOWLEDGEMENT

I am pleased to acknowledge my sincere thanks to Board of Management of **SATHYABAMA** for their kind encouragement in doing this project and for completing it successfully. I am grateful to them.

I convey my thanks to **Dr. N.M.Nandhitha, M.E.,Ph.D., Dean, School of Electrical and Electronics Engineering** and **Dr. T. Ravi, M.E., Ph.D., Head of the Department, Dept. of Electronics and Communication Engineering** for providing me necessary support and details at the right time during the progressive reviews.

I would like to express my sincere and deep sense of gratitude to my Project Guide **Dr. T. Ravi** for his valuable guidance, suggestions and constant encouragement paved way for the successful completion of my project work.

I wish to express my thanks to all Teaching and Non-teaching staff members of the Department of Department of Electronics and Communication Engineering who were helpful in many ways for the completion of the project.

ABSTRACT

Attempts to overcome the bottleneck of test data bandwidth between the tester and the chip have made the concept of combining LBIST and test data compression a vital research and development area. In particular, several hybrid BIST schemes store deterministic top-up patterns (used to detect random pattern resistant faults) on the tester in a compressed form, and then use the existing BIST hardware to decompress these test patterns.

Some solutions embed deterministic stimuli by using compressed weights or by perturbing pseudorandom vectors in various fashions. If BIST logic is used to deliver compressed test data, then underlying encoding schemes typically take advantage of low fill rates, as originally proposed in LFSR coding which subsequently evolved first into static LFSR reseeding and then into dynamic LFSR reseeding. Thorough surveys of relevant test compression techniques can be found in the literature.

The goal of the work is to design a low power programmable pseudo random pattern generator using bit swapping technique which can generate pseudo random test patterns with toggle selection. It provides improved fault coverage and power optimization compared to conventional BIST with LFSR based pseudo random pattern generator. It contains a BS-LFSR which produces binary sequences with TOSE (toggle selection) activity. A new BIST configuration with proposed PRPG is designed and used for testing some standard circuits.

TABLE OF CONTENTS

CHAPTER NO.	TITLE	PAGE NO.
	ABSTRACT	v
	LIST OF FIGURES	viii
1	INTRODUCTION	1
1.1	VLSI Design Flow	1
1.1.1	<i>Specification</i>	2
1.1.2	<i>High Level Design</i>	2
1.1.3	<i>Micro Design/Low Level Design</i>	2
1.1.4	<i>RTL Coding</i>	3
1.1.5	<i>Simulation</i>	3
1.1.6	<i>Counter Simulation</i>	4
1.1.7	<i>VLSI Testing Problems</i>	5
1.1.7.1	<i>Test generation</i>	5
1.1.7.2	<i>I/P combination</i>	5
1.1.7.3	<i>Door to I/O stick proportion issues</i>	6
1.2	VERLOG HDL	6
1.2.1	<i>History</i>	6
1.2.2	<i>Verilog – 2001</i>	6
1.2.3	<i>Constants</i>	7
1.2.4	<i>Synthesizable Constructs</i>	7
1.2.5	<i>Initial VS Always</i>	10
1.2.6	<i>Race Condition</i>	11
1.2.7	<i>System Tasks</i>	12
1.3	SOFTWARE TOOL-XILINX	13
1.3.1	<i>Version ISE 14.7</i>	14

CHAPTER NO.	TITLE	PAGE NO.
1.4	FPGA ARCHITECTURE	27
1.4.1	FPGA Design Flow	27
1.4.2	Complex Programmable Logic Design	31
1.4.3	Field-Programmable Entryway Exhibits	32
1.5	OVERVIEW	34
2	LITERATURE SURVEY	37
2.1	BASIC STRUCTURE OF PRESTO GENERATOR	39
3	PROJECT DESCRIPTION	41
3.1	INTRODUCTION	41
3.2	ARCHITECTURE OF LFSR	43
3.3	OBSERVATION OF BS-LFSR	44
3.3.1	PROPOSED TOSS GENERATOR WITH LFSR	45
4	RESULTS AND DISCUSSIONS	47
4.1	BASIC PRESTO GENERATOR RESULT	47
4.1.1	<i>Basic Presto Simulation Result</i>	47
4.1.2	<i>Basic Presto Synthesis Result</i>	47
4.2	PROPOSED PRESTO GENERATOR RESULT	48
4.2.1	<i>Proposed Presto Simulation Result</i>	48
4.2.2	<i>Proposed Presto Synthesis Result</i>	48
4.3	PRESTO BASED BIST FOR CARRY SELECTOR	49
4.4	PROPOSED PRESTO BASED	
	BIST FOR CARRY SELECTOR	50
4.4.1	<i>Power Result</i>	50
5	SUMMARY AND CONCLUSION	61
5.1	SUMMARY	61
5.2	CONCLUSION AND FUTURE SCOPE	61
	REFERENCES	62

LIST OF FIGURES

FIGURE NO.	TITLE	PAGE NO.
1.1	Typical design flow	1
1.2	High level block diagram	2
1.3	Sample low level design	3
1.4	Simulation	4
1.5	Counter wave form	4
1.6	Architecture of FPGA	27
1.7	Design flow	28
1.8	General architecture of CPLDs	32
1.9	General architecture of FPGA	33
2.1	Basic structure of presto generator	39
3.1	Presto generator	41
3.2	Architecture of bit swapping LFSR	44
3.3	Architecture toss generator with BS-LFSR	46
4.1	Basic Presto Generator simulation results	47
4.2	Basic Presto Generator synthesis results	47
4.3	Proposed Presto Generator simulation results	48
4.4	Power Consumption results of Presto based CSA	49
4.5	Power analysis of proposed presto BIST for CSA	50

LIST OF TABLES

4.1	Synthesis Results for PRESTO architecture	48
4.2	Synthesis Results for basic Presto CSA	49
4.3	Synthesis Results for proposed presto CSA	50

CHAPTER 1

INTRODUCTION

1.1 VLSI DESIGN FLOW

VHDL is a genuinely broadly useful dialect, and it doesn't require a test system on which to run the code. There are a great deal of VHDL compilers, which construct executable doubles. It can read and compose records on the host PC, so a VHDL program can be composed that creates another VHDL program to be joined in the outline being produced (Fig 1.1).

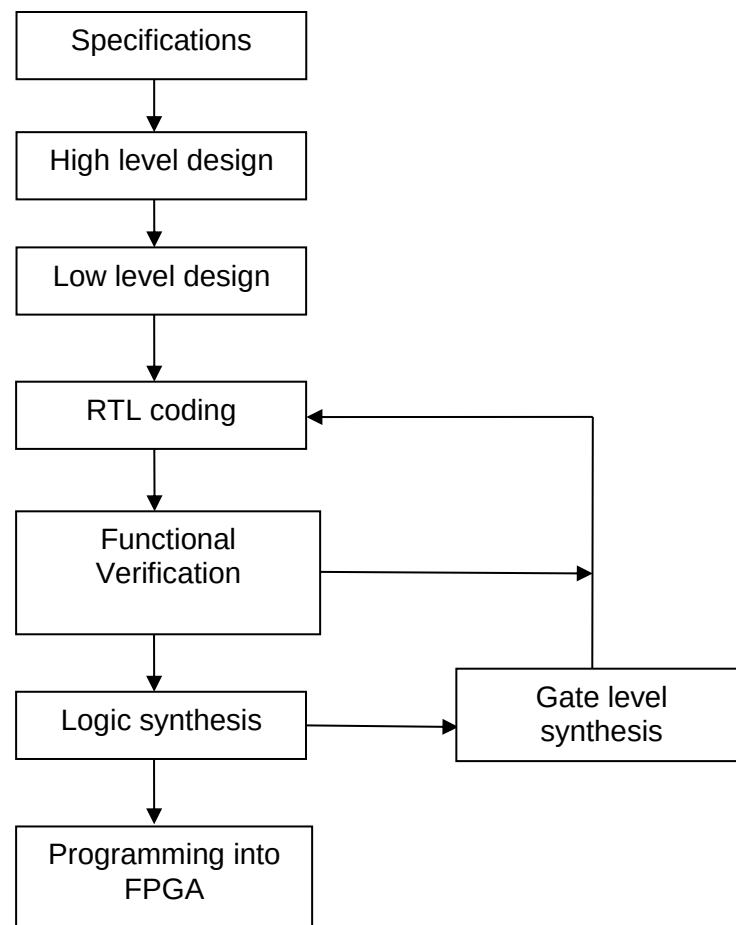


Fig 1.1: Typical design flow

1.1.1 Specification

This is the stage at which we define what are the important parameters of the system design that you are planning to design. A simple example would be: I want to design a counter; it should be 4 bit wide, should have synchronous reset, with active high enable; when reset is active, counter output should go to "0" .

1.1.2 High Level Design

This is the stage at which you define various blocks in the design and how they communicate. Let's assume that we need to design a microprocessor: high level design means splitting the design into blocks based on their function; in our case the blocks are registers, ALU, Instruction Decode, Memory Interface, etc. (Fig 1.2).

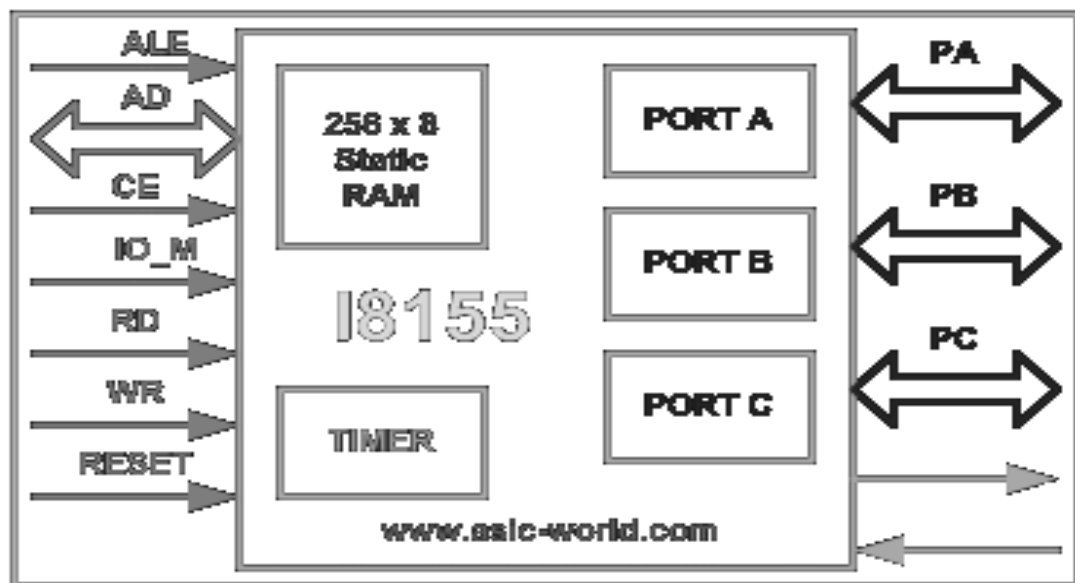


Fig 1.2: High level block diagram

1.1.3 Micro Design/Low Level Design

Low-level design or Micro design is the phase in which the designer describes how each block is implemented. It contains details of State

machines, counters, Mux, decoders, internal registers. It is always a good idea to draw waveforms at various interfaces. This is the phase where one spends lot of time (Fig 1.3).

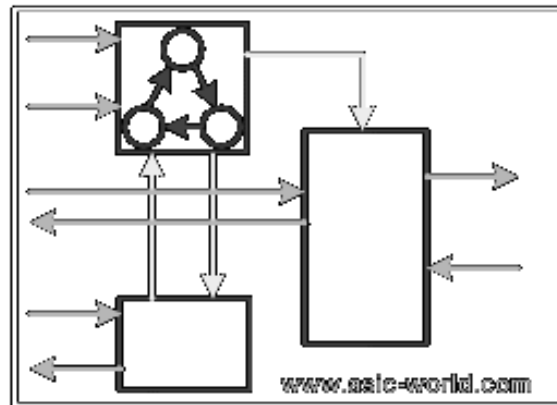


Fig 1.3: Sample Low-level design

1.1.4 RTL Coding

In RTL coding, Micro design is converted into Verilog/VHDL code, using synthesizable constructs of the language. Normally we like to lint the code, before starting verification or synthesis.

1.1.5 Simulation

Simulation is the process of verifying the functional characteristics of models at any level of abstraction. We use simulators to simulate the Hardware models. To test if the RTL code meets the functional requirements of the specification, we must see if all the RTL blocks are functionally correct. To achieve this we need to write a test bench, which generates clock, reset and the required test vectors. A sample test bench for a counter is shown below. Normally we spend 60-70% of time in design verification(Fig 1.4).

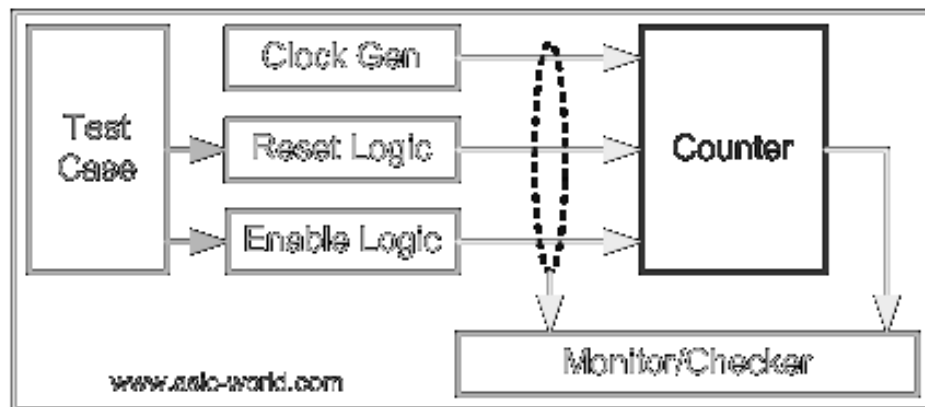


Fig 1.4: Simulation

We use the waveform output from the simulator to see if the DUT (Device Under Test) is functionally correct. Most of the simulators come with a waveform viewer. As design becomes complex, we write self checking test bench, where test bench applies the test vector, then compares the output of DUT with expected values.

There is another kind of simulation, called timing simulation, which is done after synthesis or after P&R (Place and Route). Here we include the gate delays and wires delays and see if DUT works at rated clock speed. This is also called as SDF simulation or gate level simulation.

1.1.6 Counter Simulation

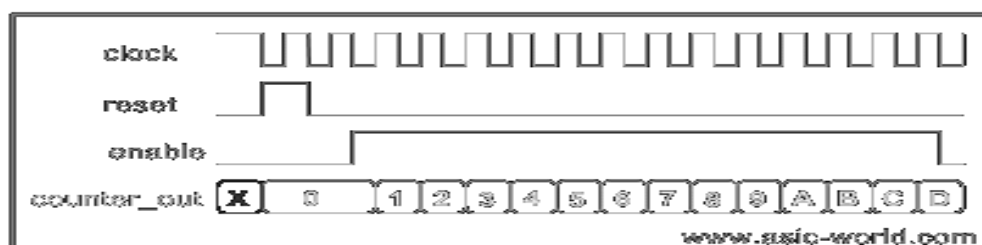


Fig 1.5: Counter waveform

1.1.7 VLSI Testing Problems

VLSI – consequently physically testing an IC is almost inconceivable.

1. Test Generation
2. I/P combinatorial issue
3. Gate to I/O stick proportion issue

1.1.7.1 Test generation

1. VLSI has extensive number of doors, henceforth pushed programmed test age times to weeks and months.
2. It is about incomprehensible for an outer analyzer to deal with the experiments, and subsequently brought about high cost as far as both Money and Time.
3. Another test age issue is, ATPG function admirably for combinatorial rationale.

Yet, sometimes falls short for successive rationale (as it requires more space and comp time – as it needs to assess every one of the states).

1.1.7.2 I/p combinatorial issue

1. A combinatorial circuit, with N number of data sources will have an aggregate arrangement of 2^N number of test vectors for comprehensive testing.
2. How about a microchip (VLSI) chip, which has somewhere in the range of 32 pins (and again obviously in various mixes).
3. Selective no. of sources of info can be connected (however it isn't a thorough testing).

1.1.7.3 Door to I/O stick proportion issue

1. Typically, a VLSI IC will have 32 pins – 40 pins; yet the quantity of doors inside the chip will be around > 50,000.
2. To test every one of the doors inside, it is exceedingly unthinkable; bringing about poor perceptibility and controllability.

Limit Scan, JTAG, IEEE 1149.1 Tutorial

1.2 VERILOG HDL

1.2.1 History

Verilog was the first modern hardware description language to be invented. It was created by **Phil Moorby** and **PrabhuGoel** during the winter of 1983/1984. The wording for this process was "Automated Integrated Design Systems" (later renamed to **Gateway Design Automation** in 1985) as a hardware modeling language. Gateway Design Automation was purchased by **Cadence Design Systems** in 1990. Cadence now has full proprietary rights to Gateway's Verilog and the Verilog-XL, the HDL-simulator that would become the de-facto standard (of Verilog **logic simulators**) for the next decade. Originally, Verilog was intended to describe and allow simulation; only afterwards was support for synthesis added.

1.2.2 Verilog 2001

Extensions to Verilog-95 were submitted back to IEEE to cover the deficiencies that users had found in the original Verilog standard. These extensions became **IEEE** Standard 1364-2001 known as Verilog-2001.

Verilog-2001 is a significant upgrade from Verilog-95. First, it adds explicit support for (2's complement) signed nets and variables. Previously, code authors had to perform signed operations using awkward bit-level manipulations (for example, the carry-out bit of a simple 8-bit addition required an explicit description of the Boolean algebra to determine its correct value).

The same function under Verilog-2001 can be more succinctly described by one of the built-in operators: +, -, /, *, >>>. A generate/endgenerate construct (similar to VHDL's generate/end generate) allows Verilog-2001 to control instance and statement instantiation through normal decision operators (case/if/else). Using generate/endgenerate, Verilog-2001 can instantiate an array of instances, with control over the connectivity of the individual instances. File I/O has been improved by several new system tasks. And finally, a few syntax additions were introduced to improve code readability (e.g. always @*, named parameter override, C-style function/task/module header declaration).

Verilog-2001 is the dominant flavor of Verilog supported by the majority of commercial [EDA](#) software packages.

1.2.3 Constants

The definition of constants in Verilog supports the addition of a width parameter. The basic syntax is:

<Width in bits>'<base letter><number>

Examples:

- 12'h123 - Hexadecimal 123 (using 12 bits)
- 20'd44 - Decimal 44 (using 20 bits - 0 extension is automatic)
- 4'b1010 - Binary 1010 (using 4 bits)
- 6'o77 - Octal 77 (using 6 bits)

1.2.4 Synthesizable Constructs

There are several statements in Verilog that have no analog in real hardware, e.g. \$display. Consequently, much of the language can not be used to describe hardware. The examples presented here are the classic subset of the language that has a direct mapping to real gates.

```
// Mux examples - Three ways to do the same thing.
```

```
// The first example uses continuous assignment
```

```
wire out;
```

```
assign out = sel?a : b;
```

```
// the second example uses a procedure
```

```
// to accomplish the same thing.
```

```
reg out;
```

```
always@(a or b or sel)
```

```
begin
```

```
case(sel)
```

```
1'b0: out = b;
```

```
1'b1: out = a;
```

```
endcase
```

```
end
```

```
// Finally - you can use if/else in a
```

```
// procedural structure.
```

```
reg out;
```

```
always@(a or b or sel)
```

```
if(sel)
```

```
out= a;
```

```
else
```

```
out= b;
```

The next interesting structure is a [transparent latch](#); it will pass the input to the output when the gate signal is set for "pass-through", and captures the input and stores it upon transition of the gate signal to "hold". The output will remain stable regardless of the input signal while the gate is set to "hold". In the example below the "pass-through" level of the gate would be when the value of the if clause is true, i.e. gate = 1. This is read "if gate is true, the din is fed to latch_out continuously." Once the if clause is false, the last value at latch_out will remain and is independent of the value of din.

EX6: // Transparent latch example

```
reg out;  
always@(gate or din)  
if(gate)  
out= din;// Pass through state  
// Note that the else isn't required here. The variable  
// out will follow the value of din while gate is high.  
// When gate goes low, out will remain constant.
```

The [flip-flop](#) is the next significant template; in Verilog, the D-flop is the simplest, and it can be modeled as:

```
reg q;  
always@(posedgeclk)  
    q <= d;
```

The significant thing to notice in the example is the use of the non-blocking assignment. A basic [rule of thumb](#) is to use <= when there is a **posedge** or **negedge** statement within the always clause.

A variant of the D-flop is one with an asynchronous reset; there is a convention that the reset state will be the first if clause within the statement.

```
reg q;  
always@(posedgeclkorposedge reset)  
if(reset)  
    q <=0;  
else  
    q <= d;
```

The next variant is including both an asynchronous reset and asynchronous set condition; again the convention comes into play, i.e. the reset term is followed by the set term.

```

reg q;
always@(posedgeclkorposedge reset orposedge set)
if(reset)
    q <=0;
else
if(set)
    q <=1;
else
    q <= d;

```

1.2.5 Initial V/S Always

There are two separate ways of declaring a Verilog process. These are the **always** and the **initial** keywords. The **always** keyword indicates a free-running process. The **initial** keyword indicates a process executes exactly once. Both constructs begin execution at simulator time 0, and both execute until the end of the block. Once an **always** block has reached its end, it is rescheduled (again). It is a common misconception to believe that an initial block will execute before an always block. In fact, it is better to think of the **initial**-block as a special-case of the **always**-block, one which terminates after it completes for the first time.

//Examples:

```

initial
begin
    a =1;// Assign a value to reg a at time 0
    #1;// Wait 1 time unit
    b = a;// Assign the value of reg a to reg b
end

```

```

always@(a or b)// Any time a or b CHANGE, run the process
begin
if(a)

```

```

    c = b;
else
    d =~b;
end// Done with this block, now return to the top (i.e. the @ event-control)
always@(posedge a)// Run whenever reg a has a low to high change
    a <= b;

```

These are the classic uses for these two keywords, but there are two significant additional uses. The most common of these is an **always** keyword without the @(..) sensitivity list. It is possible to use always as shown below:

```

always
begin// Always begins executing at time 0 and NEVER stops
clk=0;// Set clk to 0
#1;// Wait for 1 time unit
clk=1;// Set clk to 1
#1;// Wait 1 time unit
end// Keeps executing - so continue back at the top of the begin

```

The **always** keyword acts similar to the "C" construct **while(1) {..}** in the sense that it will execute forever.

The other interesting exception is the use of the **initial** keyword with the addition of the **forever** keyword.

1.2.6 Race Condition

The order of execution isn't always guaranteed within Verilog. This can best be illustrated by a classic example. Consider the code snippet below:

```

initial
    a =0;
initial
    b = a;

```

```

initial
begin
#1;
$display("Value a=%b Value of b=%b",a,b);
end

```

What will be printed out for the values of a and b? Depending on the order of execution of the initial blocks, it could be zero and zero, or alternately zero and some other arbitrary uninitialized value. The \$display statement will always execute after both assignment blocks have completed, due to the #1 delay.

1.2.7 System Tasks

System tasks are available to handle simple I/O, and various design measurement functions. All system tasks are prefixed with \$ to distinguish them from user tasks and functions. This section presents a short list of the most often used tasks. It is by no means a comprehensive list.

- \$display - Print to screen a line followed by an automatic newline.
- \$write - Write to screen a line without the newline.
- \$swrite - Print to variable a line without the newline.
- \$sscanf - Read from variable a format-specified string. (*Verilog-2001)
- \$fopen - Open a handle to a file (read or write)
- \$fdisplay - Write to file a line followed by an automatic newline.
- \$fwrite - Write to file a line without the newline.
- \$fscanf - Read from file a format-specified string. (*Verilog-2001)
- \$fclose - Close and release an open file handle.
- \$readmemh - Read hex file content into a memory array.
- \$readmemb - Read binary file content into a memory array.

- \$monitor - Print out all the listed variables when any change value.
- \$time - Value of current simulation time.
- \$dumpfile - Declare the VCD ([Value Change Dump](#)) format output file name.
- \$dumpvars - Turn on and dump the variables.
- \$dumpports - Turn on and dump the variables in Extended-VCD format.
- \$random - Return a random value

1.3 SOFTWARE TOOL-XILINX

Xilinx ISE is a software tool produced by Xilinx for synthesis and analysis of HDL designs, which enables the developer to synthesize ("compile") their designs, perform timing analysis, examine RTL diagrams, simulate a design's reaction to different stimuli, and configure the target device with the programmer.

Xilinx was founded in 1984 by two semiconductor engineers, Ross Freeman and Bernard Vonderschmitt, who were both working for integrated circuit and solid-state device manufacturer Zilog Corp.

While working for Zilog, Freeman wanted to create chips that acted like a blank tape, allowing users to program the technology themselves. At the time, the concept was paradigm-changing. "The concept required lots of transistors and, at that time, transistors were considered extremely precious – people thought that Ross's idea was pretty far out", said Xilinx Fellow Bill Carter, who when hired in 1984 as the first IC designer was the company's eighth employee.

Xilinx is a software tool, which is used to run the programs in **VHDL/Verilog HDL** language. It has various versions like Xilinx 92.1, Xilinx 10.1, Xilinx 13.2 and 14.7 etc.

1.3.1 Version ISE 14.7

The design example used in this tutorial demonstrates many device features, software features, and design flow practices you can apply to your own design. This design targets a Spartan™-3A device; however, all of the principles and flows taught are applicable to any Xilinx® device family, unless otherwise noted. The design is composed of HDL elements and two cores. You can synthesize the design using Xilinx Synthesis Technology (XST), Synplify/Synplify Pro, or Precision software.

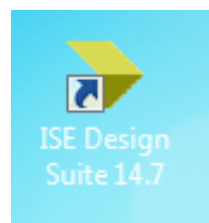
WHY XILINX ONLY?

We have many software tools to run the VHDL/Verilog programs like cadence .But compared to all software tools Xilinx is cost effective.

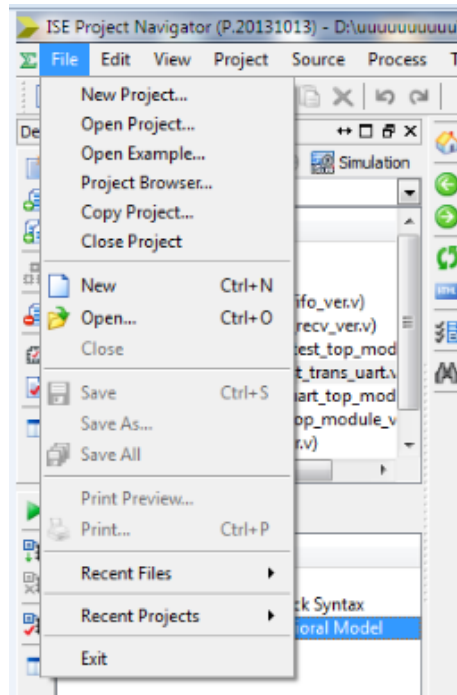
A BRIEF TUTORIAL: IMPLEMENTING VHDL DESIGNS USING XILINX ISE.

This tutorial shows how to create, implement, simulate, and synthesize VERILOG designs for implementation in FPGA chips using Xilinx ISE 14.7 and ModelSim: Xilinx.

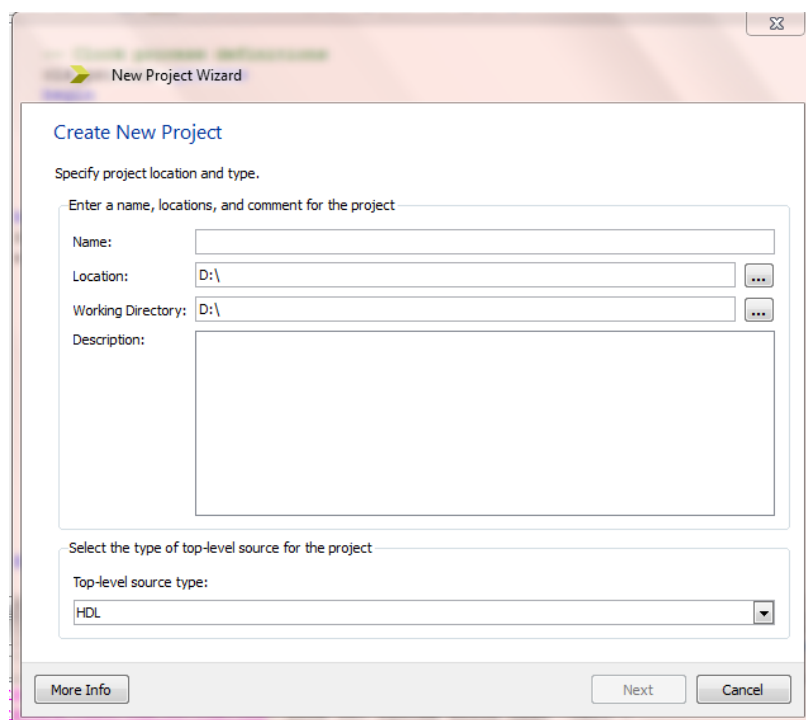
- ✓ Launch Xilinx ISE from either the shortcut on your desktop or from your start menu under Programs → Xilinx ISE 14.7 → Project Navigator.



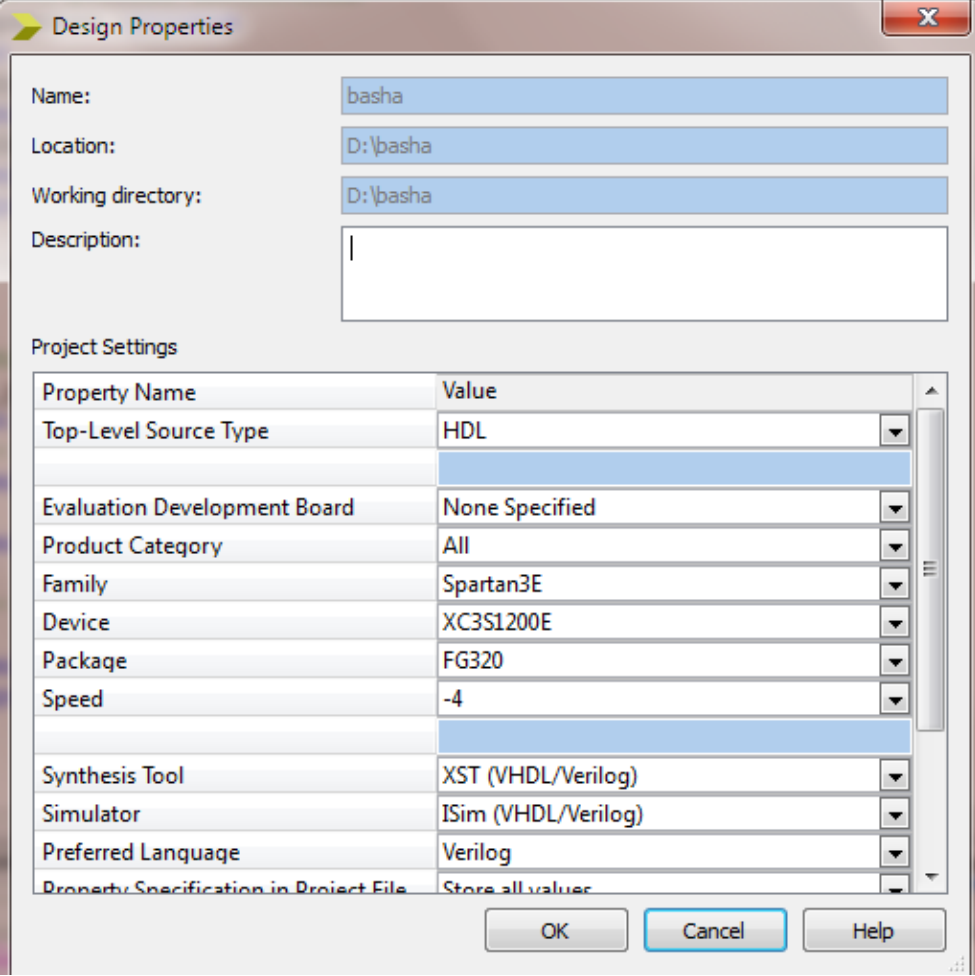
- ✓ Start a new project by clicking File → New Project....



- ✓ In the resulting window, verify the “Top-Level Source Type” is VERILOG. Change the “Project Location” to a suitable directory and give it whatever name you choose, e.g. “lab3”.



The next window shows the details of the project and the target chip. We will be synthesizing designs into real chips so it is important to match the target chip with the particular board/chip you will be using. Beginning labs will be done in a Spartan 3E XC3S1200E chip that comes in a FG320 package with a speed grade of 6 as shown below.



The image shows a 'Design Properties' dialog box with the following fields and settings:

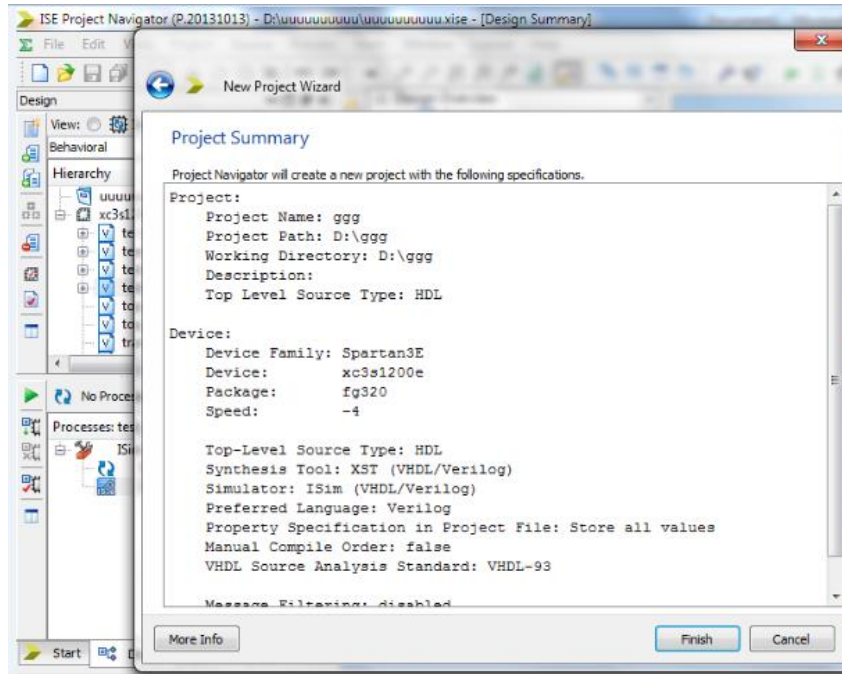
- Name: basha
- Location: D:\basha
- Working directory: D:\basha
- Description: (empty)

Project Settings

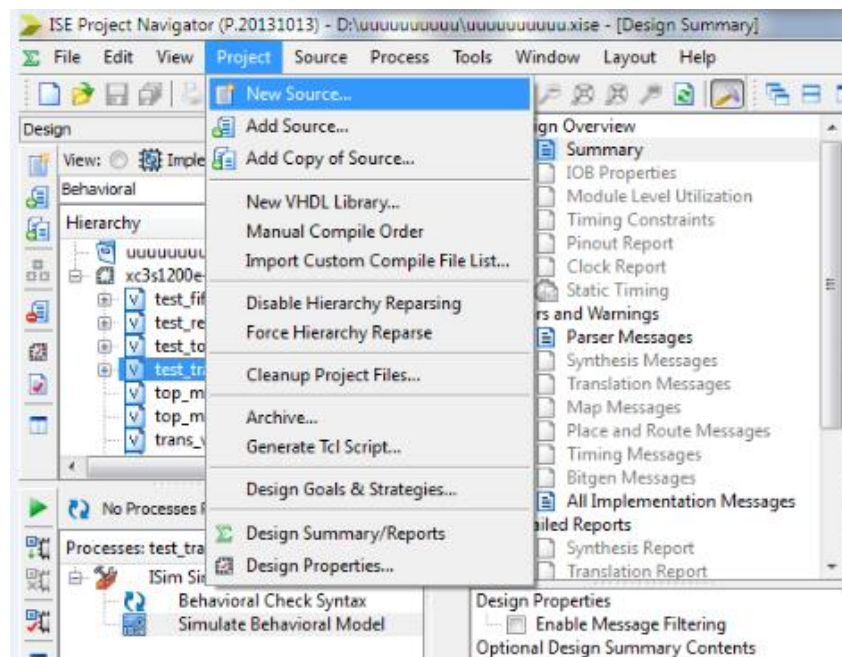
Property Name	Value
Top-Level Source Type	HDL
Evaluation Development Board	None Specified
Product Category	All
Family	Spartan3E
Device	XC3S1200E
Package	FG320
Speed	-4
Synthesis Tool	XST (VHDL/Verilog)
Simulator	ISim (VHDL/Verilog)
Preferred Language	Verilog
Property Specification in Project File	Store all values

Buttons: OK, Cancel, Help

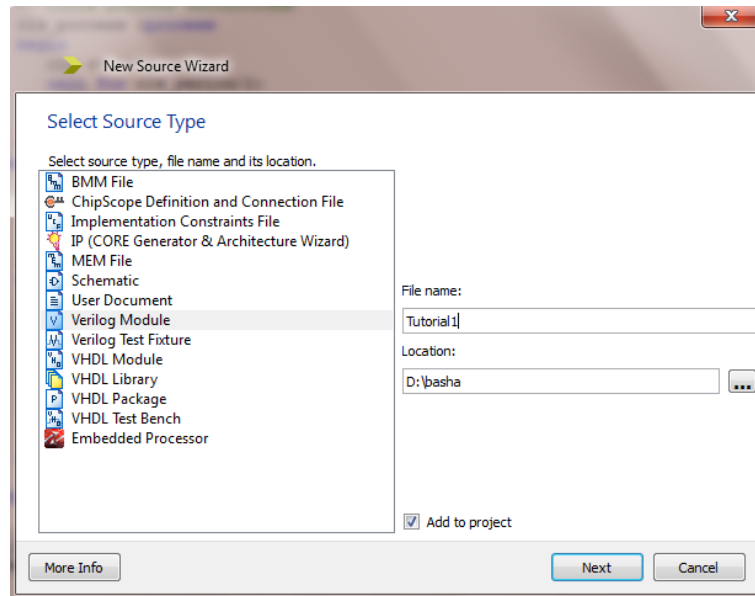
- ✓ Since we are starting a new design the next couple of pop-up windows aren't relevant, just click Next and Next and Finish.



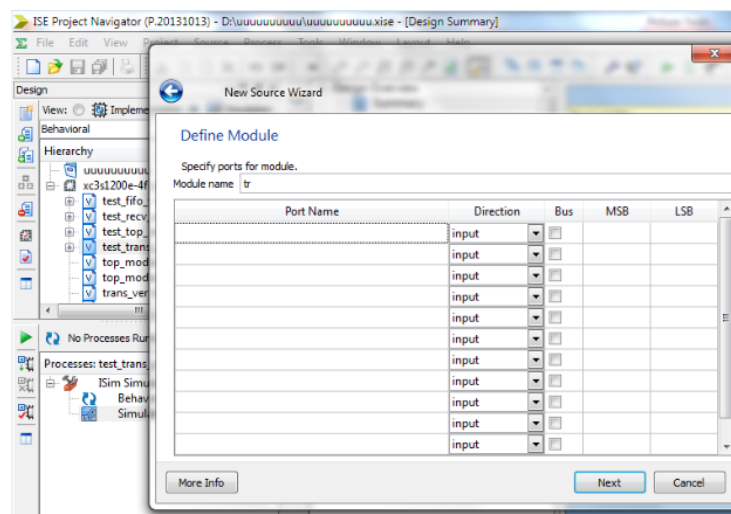
- ✓ You should now be in the main Project Navigator window. Select Project → New Source... from the menu.



- ✓ In the resulting pop-up window specify a VERILOG Module source and give the file a name. I tend to just use the same name as the project itself, e.g. Lab 3. Click Next.



- ✓ The next pop-up window allows you to specify your inputs and outputs through the Wizard if you so desire. In this tutorial we will build a 2 x 1 multiplexer so we can specify the inputs and outputs as shown below. Here, the default entity and architecture names have also been changed. Once all inputs and outputs are entered click Next and click Finish.



- ✓ The project will usually open with the design summary tab active in the right hand side of your window. We want to go to the VERILOG code so you need to click the *.v tab for your design.
- ✓ You can see that the Wizard has used the default type for your signals and also filled in the basic entity and architecture details for you.

```

1 | timescale 1ns / 1ps
2 | ///////////////////////////////////////////////////////////////////
3 | // Company:
4 | // Engineer:
5 | //
6 | // Create Date:    13:07:46 09/15/2017
7 | // Design Name:
8 | // Module Name:    orgate
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 or(
22 |     input a,b,
23 |     output y
24 | );
25 | assign y=a|b;
26 |
27 | endmodule
28 |

```

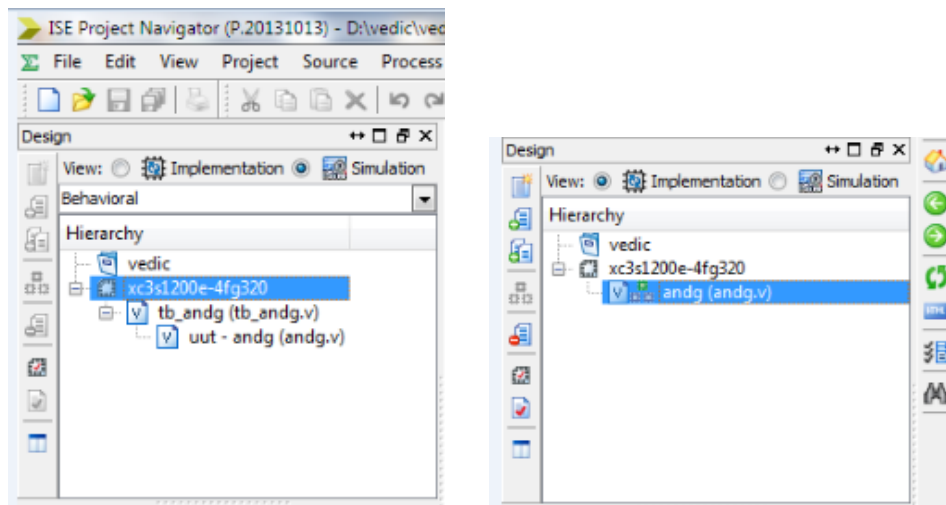
- ✓ Now you can fill in the rest of your code for your design. In this case, we can do the multiplexer as shown below. Make sure to frequently save your code.

```

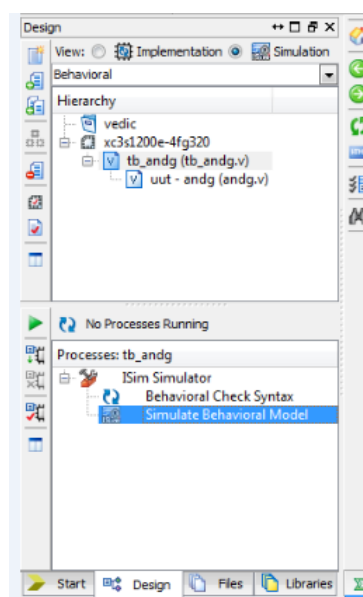
8 | // Module Name:    mux21
9 | // Project Name:
10 | // Target Devices:
11 | // Tool versions:
12 | // Description:
13 | //
14 | // Dependencies:
15 | //
16 | // Revision:
17 | // Revision 0.01 - File Created
18 | // Additional Comments:
19 | //
20 | ///////////////////////////////////////////////////////////////////
21 | module mux21(
22 |     input a,b, s,
23 |     output reg y
24 | );
25 | always@ (a,b,s)
26 | begin
27 |     case (s)
28 |     1'b0:y=a;
29 |     1'b1:y=b;
30 |     default:y=1'bx;
31 |     endcase
32 | end
33 |
34 | endmodule
35 |

```

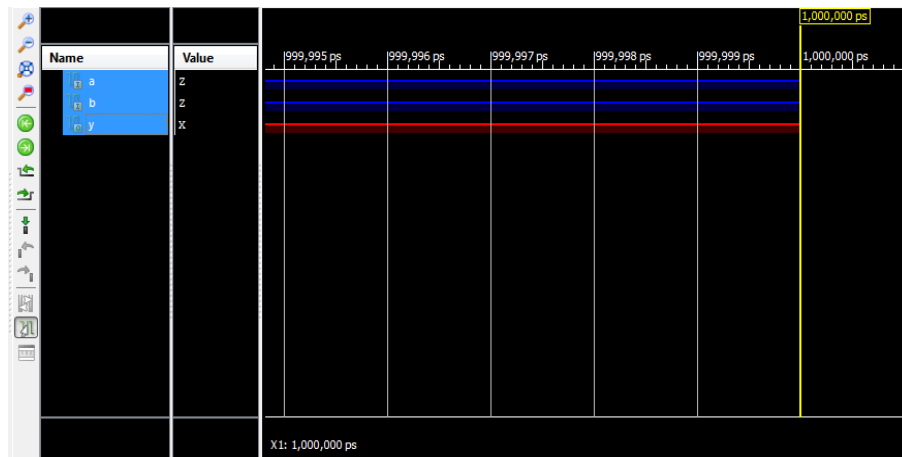
- ✓ Once the code is entered we can proceed with a simulation of the design or we can synthesize the code for implementation and download onto an FPGA. Let us proceed with the simulation first. In the upper left-hand side of the ISE environment there is a Sources sub window which has a drop down box as shown below. Note that the drop down box currently shows “Synthesis/Implementation”. Change this to “Behavioral Simulation”.



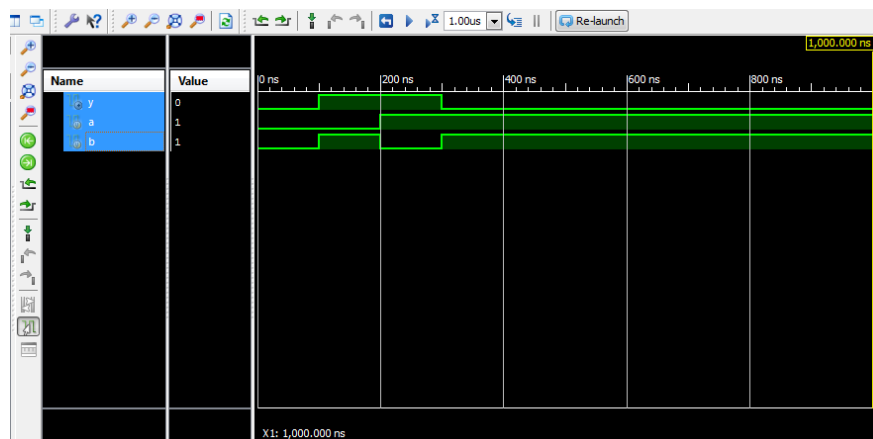
- ✓ Highlight your *.v file in the Sources sub window and then expand the “Model Sim Simulator” selection in the Processes sub window as shown below. Click on “Simulate Behavioral Model” to launch the Model Sim simulator.



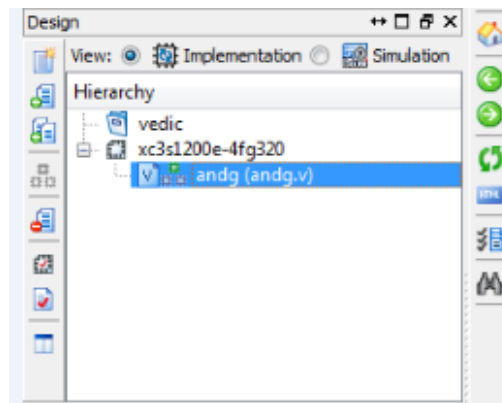
- ✓ Model Sim should successfully launch and will open several sub windows by default. For now we just need the “Wave” and “Transcript” sub windows, so close the other sub windows and you should see the following:



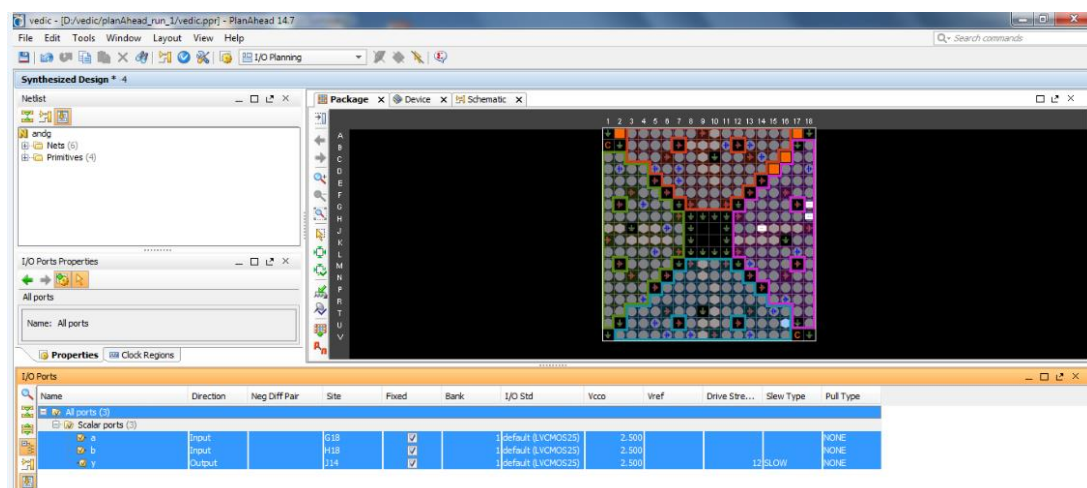
- ✓ To conduct the simulation you basically only need to know two commands, “force” and “run”. Force is used to set the value of any input variable. Then Run the simulation for a specific amount of time. To use a Force command, in the transcript window simply type Force, space, logic variable you wish to set, space, the value you wish to assign (0 or 1). For example, “force b 0”. Then type “run 1000” to run the simulation for 1000 picoseconds. Alternatively, you could type “run 1 ns”. Change the various inputs one or more at a time and do additional run steps.



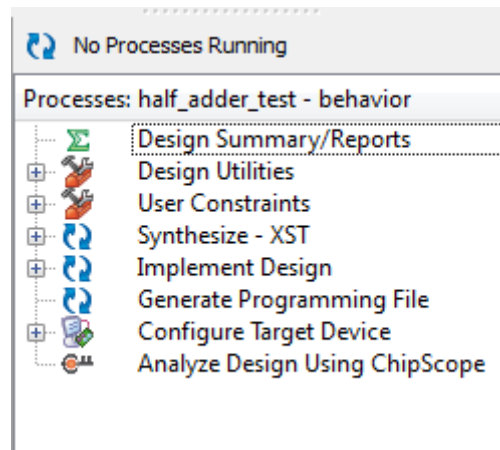
- Now let's look at the flow for actually synthesizing and implementing the design in the FPGA prototyping boards. Close ISim and go back to the Xilinx ISE environment. In the Sources subwindow change the selection in the dropdown box from "Behavioral Simulation" to "Synthesis/Implementation". →



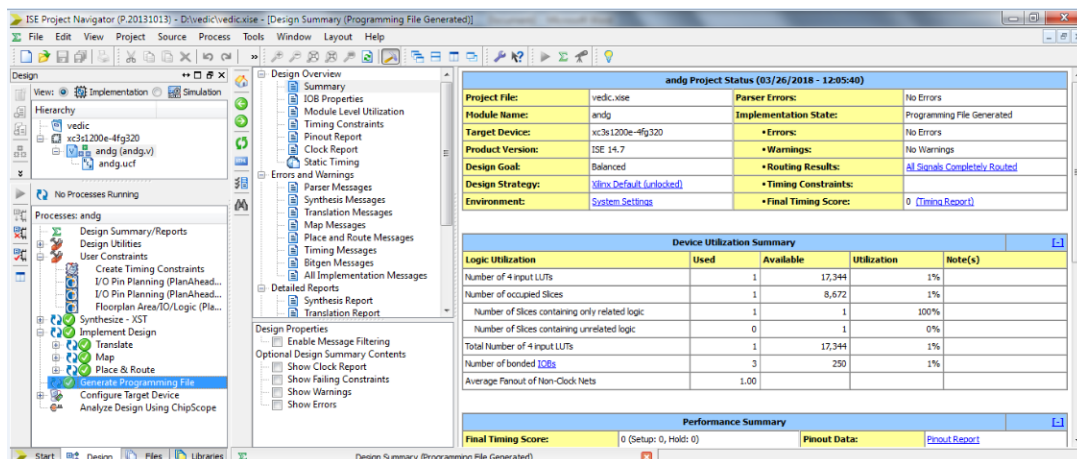
- To properly synthesize the design we need to specify which pins on the chip all the inputs and outputs should be assigned to. In general of course we could assign the signals just about any way we want. Since we will be using specific prototype boards, we need to make sure our pins assignments match the switches, buttons, and LEDs so we can test our design. We will be starting with Digilab 2E boards that are connected to Digilab DIO2 input/output boards. The I/O board has already been programmed and configured to have the following connections:



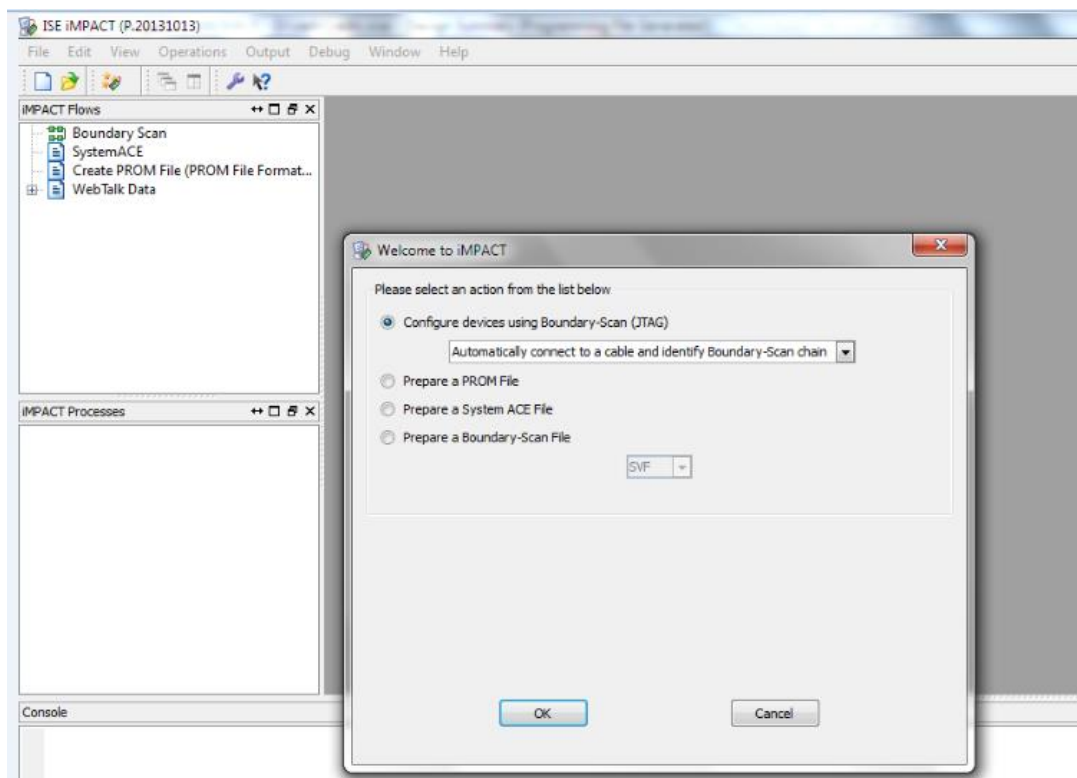
- To assign specific pins, expand the User Constraints selection under the Process subwindow and double-click on Assign Package Pins.



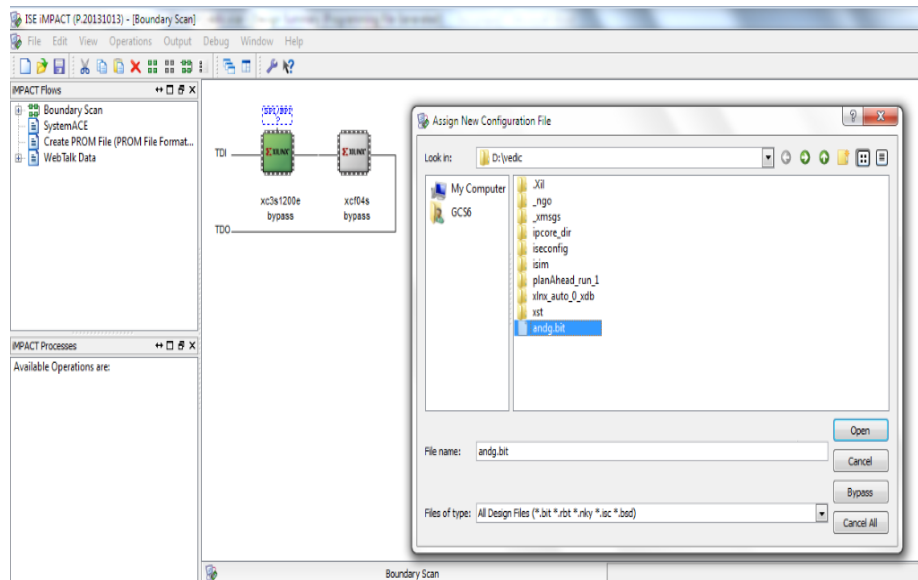
- A new application called Xilinx PACE should be launched. In the Design Object List subwindow you should see a listing of all the input and output signals from our design. Here is where we can specify which pin locations we want for each signal. Simply enter the pins numbers from the tables shown in Step 19 above, making sure to use a capital letter "P" in front of the pin. Once all pins have been assigned, save your constraints by selecting File → Save from the menu bar and exit Xilinx Pace.
- Back in the Xilinx ISE environment window we can now tell the computer to synthesize our design. In the Process subwindow double-click on the Synthesize – XST selection and wait for the process to complete. Then double-click on the Implement Design selection and wait for the process to complete. Then double-click on the Generate Programming File selection and wait for the process to complete. If all goes well, you should have green checks marks for the whole design.



- There is a lot of information you can obtain through all of the objects listed in the Processes subwindow, but let us proceed to downloading the design onto the prototyping board for testing. First make sure the prototyping board is connected to the PC and has power on. Also make sure the slide switch on the FPGA board by the parallel port is set to JTAG (as opposed to "Port"). Then select Configure Device (iMPACT) underneath the Generate Programming File selection. You should see the following window.

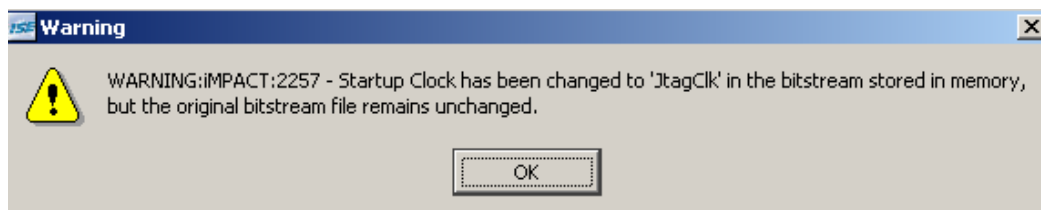


- Now you need to specify which bitstream file to use to configure the device. For this tutorial we want to select the mux.bit file and click Open.

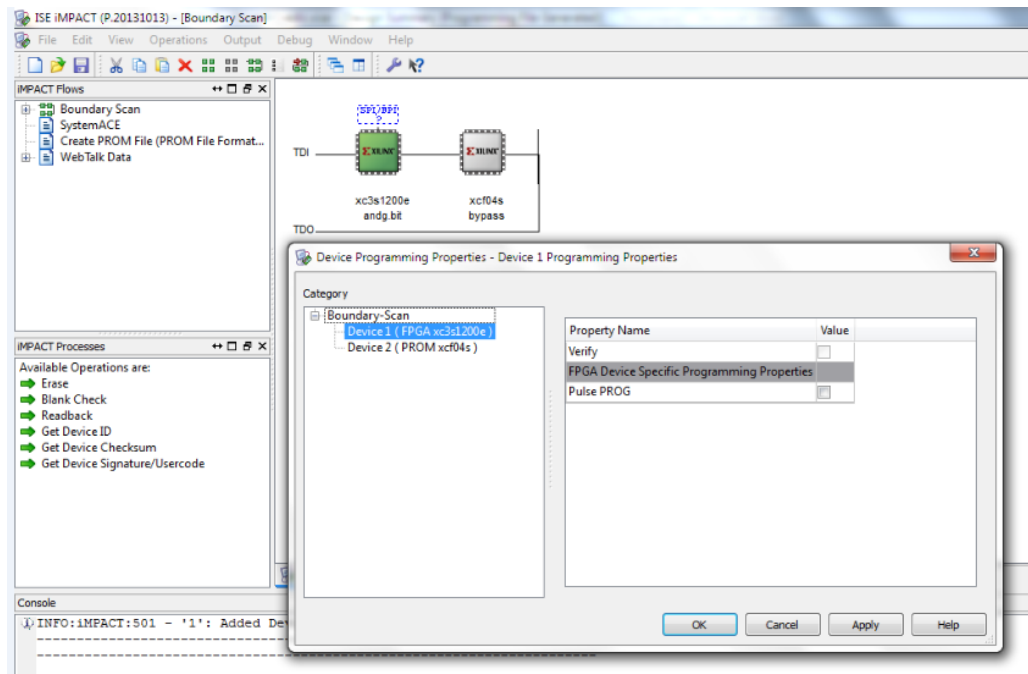


You will probably get the message below. Just click Yes.

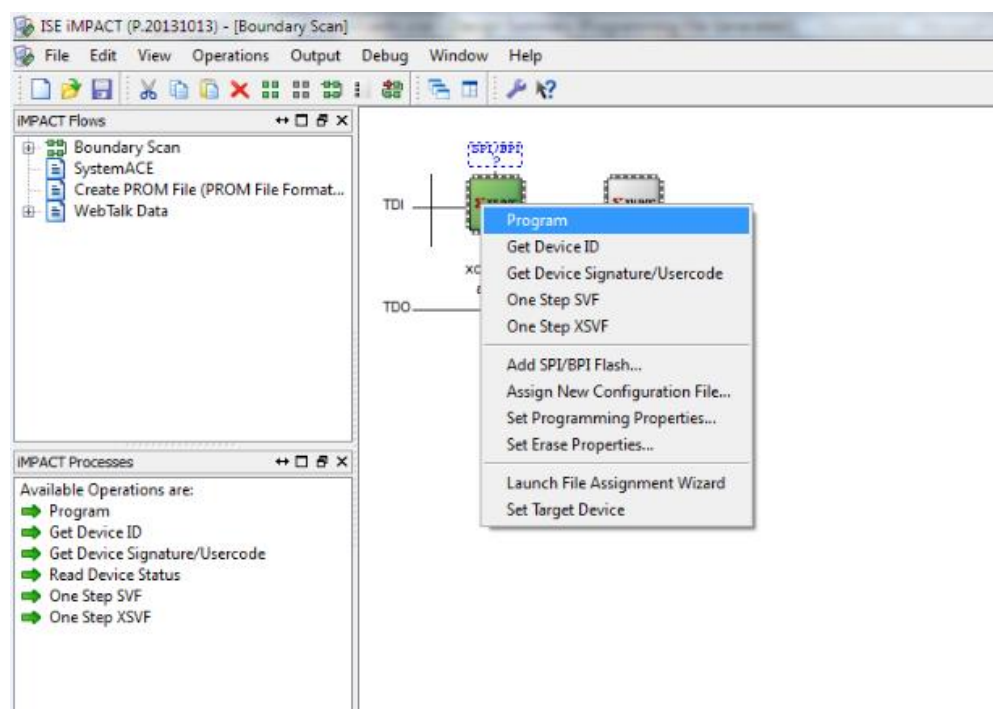
- You will also get a warning message saying the JTAG clock was updated in the bitstream file (which is good) so just click OK. There is a way to correct for that in the original designflow, but Xilinx automatically catches it here so I don't usually bother.



- You should now see the Spartan XC2S200E chip in the main window.
Right click on the chip to prepare for downloading the bitstream file.



Select Program on the resulting window.



- Click OK. If all goes well you should get the Programming Succeeded message

Program Succeeded

- Now just test and verify your design on the actual FPGA board!

1.4 FPGA Architecture

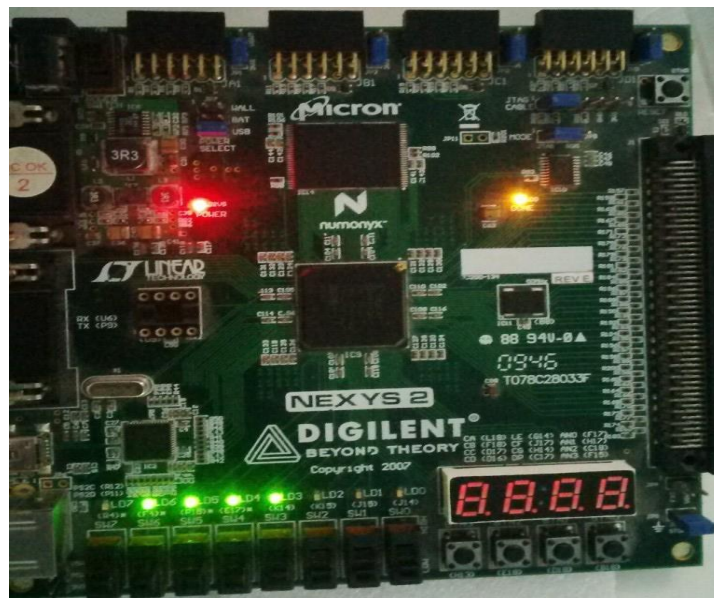


Fig 1.6: Architecture of FPGA

1.4.1 FPGA Design Flow

The ISE™ design flow comprises the following steps: design entry, design synthesis, design implementation, and Xilinx® device programming. Design verification, which includes both functional verification and timing verification, takes places at different points during the design flow.

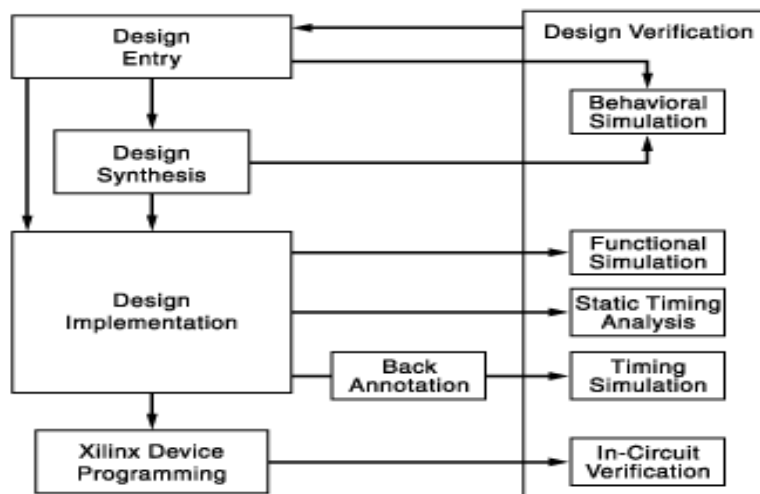


Fig 1.7: FPGA design flow

Plan Entry

Make an ISE venture as takes after

1. Create a venture.
2. Create records and add them to your undertaking, including a client limitations (UCF) document.
3. Add any current records to your venture.
4. Assign requirements, for example, timing imperatives, stick assignments, and zone limitations.

Useful Verification

You can check the usefulness of your plan at various focuses in the outline stream as takes after:

- Before combination, run behavioral reenactment (otherwise called RTL reproduction).
- After Translate, run practical reenactment (otherwise called door level recreation), utilizing the SIMPRIM library.
- After gadget programming, keep running in-circuit confirmation.

Plan Synthesis

Combine your outline.

Plan Implementation

Actualize your outline as takes after:

1. Implement your plan, which incorporates the accompanying advances:
 - Translate
 - Map
 - Place and Route
2. Review reports created by the Implement Design process, for example, the Map Report or Place and Route Report, and change any of the accompanying to enhance your outline:
 - Process properties
 - Constraints
 - Source documents
3. Synthesize and actualize your plan again until outline prerequisites are met.

Timing Verification

You can check the planning of your outline at various focuses in the plan stream as takes after:

- Run static planning examination at the accompanying focuses in the outline stream:
- After Map

- After Place and Route
- Run timing recreation at the accompanying focuses in the outline stream:
- After Map (for a fractional planning examination of CLB and IOB delays)
- After Place and Route (for full planning examination of piece and net deferrals)

Xilinx Device Programming

Program your Xilinx gadget as takes after:

1. Create a programming record (BIT) to program your FPGA.
2. Generate a PROM, ACE, or JTAG petition for investigating or to download to your gadget.
3. Use iMPACT to program the gadget with a programming link.

Execution Overview for FPGAs

After amalgamation, you run plan execution, which involves the accompanying advances:

1. Translate, which blends the approaching net records and imperatives into a Xilinx® configuration document
2. Map, which fits the outline into the accessible assets on the objective gadget
3. Place and Route, which places and courses the outline to the planning imperatives
4. Programming record age, which makes a bit stream document that can be downloaded to the gadget.

In the Sources tab, select Synthesis/Implementation from the Design View drop-down rundown, and select the best module. In the Processes tab, double tap Implement Design to run the usage procedure in one stage, or double tap Translate, Map, and Place and Route to run each of the execution steps independently. To produce the programming record, double tap Generate Programming File. On the other hand, you can choose Process - > Implement - > Top Module to run Implement Design on the best module. For subtle elements, see actualizing the Top Module.

Default property estimations are utilized for the execution procedure, unless you alter them. Properties can be set for the Implement Design process or for each of the different usage forms.

ASIC (Application-Particular Incorporated Circuit)

An ASIC (articulated "a-wiped out"; intense typeface characterizes another term) is an application particular incorporated circuit at any rate that is the thing that the acronym remains for. Before we answer the topic of what that implies we initially take a gander at the development of the silicon chip or incorporate circuit (IC).

1.4.2 Complex Programmable Logic Devices (CPLD)

CPLDs extend the general thought behind SPLDs by giving a large number of them on a solitary chip. Up to many indistinguishable sub-circuits, each of which fits in with a great SPLD, are joined with an expansive programmable interconnect grid or system, see fig 2.1. A trouble with this kind of association is that a parceling into a bundle of collaborating SPLDs must be forced falsely on any given computational errand, which benefits neither equipment nor outline proficiency. Contingent upon the maker, items are known as perplexing programmable rationale gadget (CPLD), programmable

extensive scale coordination (PLSI), erasable programmable rationale gadget (EPLD), and so forth in the business world.

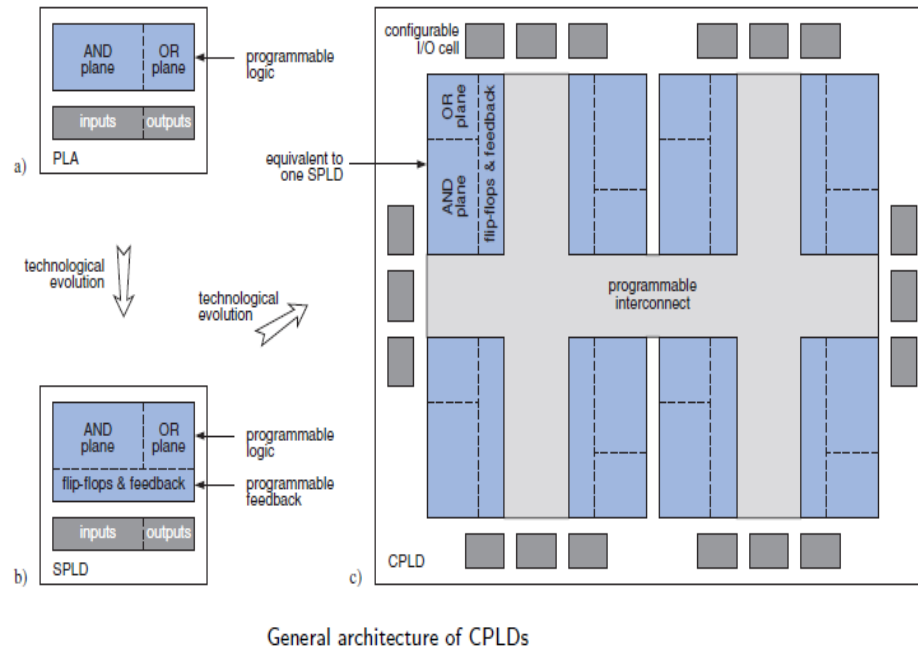


Fig 1.8: General architecture of CPLDs

1.4.3 Field-Programmable Entryway Exhibits

FPGA have their general association designed after that of entryway exhibits. Numerous configurable rationale cells are organized in a two-dimensional exhibit with groups of parallel wires in the middle. A switchbox is available wherever two wiring channels cross, see fig 2.2. Contingent upon the item, every rationale cell can be designed to do some not very unpredictable combinational operation, to store a bit or two, or both.

Instead of conventional door clusters, it is the condition of programmable connections as opposed to creation covers that chooses rationale capacities and flag directing. Parts with this association are being advanced under names, for example, field-programmable entryway cluster (FPGA), rationale cell exhibit (LCA), and programmable multilevel gadget (PMD). The quantity of configurable rationale cells enormously shifts between

items, with commonplace figures extending between a couple of handfuls and many thousands.

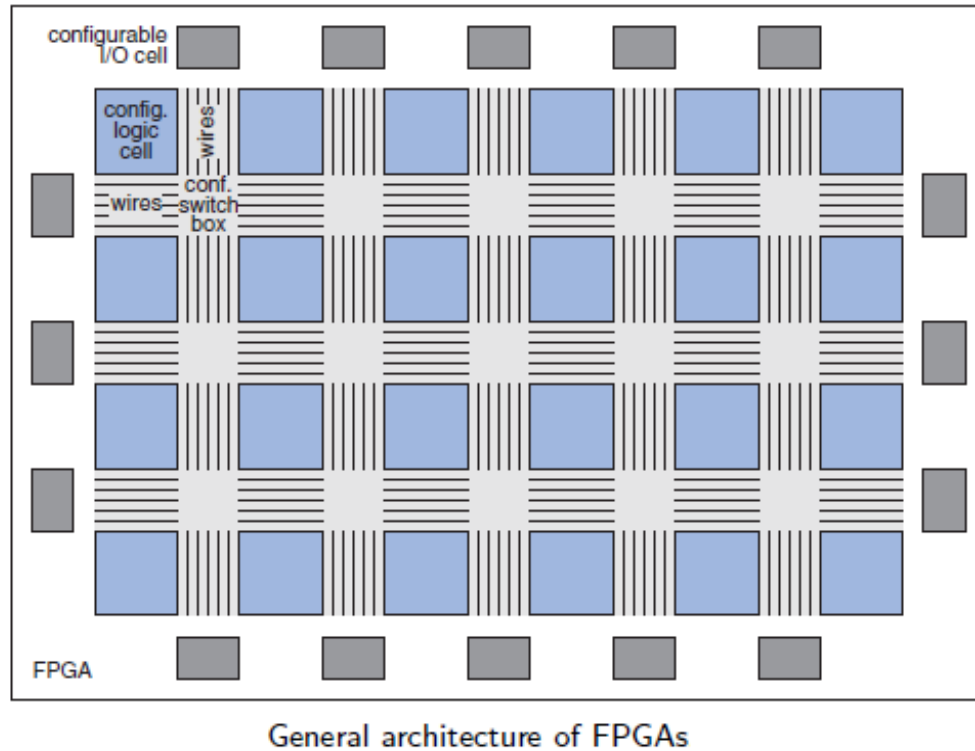


Fig 1.9: General architecture of FPGAs

A field-programmable device (FPGA) is an integrated circuit intended to be arranged by a client or an architect in the wake of assembling—subsequently "field-programmable". The FPGA arrangement is for the most part determined utilizing an equipment depiction dialect (HDL), like that utilized for an application-particular integrated circuit (ASIC) (circuit charts were beforehand used to indicate the design, as they were for ASICs, however this is progressively uncommon). Contemporary FPGAs have vast assets of rationale entryways and RAM squares to actualize complex computerized calculations. As FPGA outlines utilize quick IOs and bidirectional information transports it turns into a test to confirm remedy timing of legitimate information inside setup time and hold time. Floor arranging empowers assets allotment inside FPGA to meet these time limitations. FPGAs can be utilized to actualize any consistent capacity that an ASIC could perform. The capacity to refresh

the usefulness in the wake of delivery, halfway re-arrangement of a part of the plan and the low non-repeating building costs in respect to an ASIC outline (despite the for the most part higher unit cost), offer preferences for some applications.

1.5 OVERVIEW

Attempts to overcome the bottleneck of test data bandwidth between the tester and the chip have made the concept of combining LBIST and test data compression a vital research and development area. In particular, several hybrid BIST schemes store deterministic top-up patterns (used to detect random pattern resistant faults) on the tester in a compressed form, and then use the existing BIST hardware to decompress these test patterns.

Some solutions embed deterministic stimuli by using compressed weights or by perturbing pseudorandom vectors in various fashions. If BIST logic is used to deliver compressed test data, then underlying encoding schemes typically take advantage of low fill rates, as originally proposed in LFSR coding which subsequently evolved first into static LFSR reseeding and then into dynamic LFSR reseeding. Thorough surveys of relevant test compression techniques can be found.

For example, a device presented in employs an LFSR to feed scan chains through biasing logic and T-type flip-flop. Since this flip-flop holds the previous value until its input is asserted, the same value is repeatedly scanned into scan chains until the value at the output of biasing logic (e.g., a k -input AND gate) becomes 1. Depending on k , one can significantly reduce the number of transitions occurring at the scan chain inputs. A dual-speed LFSR consists of two LFSRs driven by normal and slow clocks, respectively. The switching activity is reduced at the circuit inputs connected to the slow-speed LFSR, while the whole scheme still ensures satisfactory fault coverage. Mask patterns mitigate the switching activity in LFSR-produced patterns, whereas a bit swapping achieves the same goal at the primary inputs of CUT. A gated

LFSR clock of allows activating only half of LFSR stages at a time. It cuts power consumption as only half of the circuit inputs change every cycle. Combining the low transition generator of (handling easy-to-detect faults) with a 3-weight pseudorandom test pattern generator (PRPG) (detecting random pattern resistant faults) can also reduce BIST switching activity, as demonstrated. The schemes of suppress transitions in LFSR-generated sequences by either statistical monitoring or injecting intermediate and highly correlated patterns. Finally, a random single-input change generator can produce low power patterns in a parallel BIST environment, As the BIST power consumption can easily exceed the maximum ratings when testing at speed, scan patterns must be shifted at a programmable low speed, and only the last few cycles and the capture cycle are applied at the maximum frequency. In the burst-mode approach presented, typically five consecutive clock cycles are used. The first four cycles serve shifting purposes, whereas the last one is designated for capture. The objective is to stabilize the power supply before the last shift and capture pulses are applied, which are critical for at-speed tests. To reduce the voltage droop related to a higher circuit activity, a burst clock controller slows down some of the shift cycles. It allows a gradual increase of the circuit activity, thereby reducing the di/dt effect. The controller can gate the shift clocks, depending on the needs for gradually warming up of the circuit. Low power (LP) test compression schemes [28], [33], [41], [53] adapt again LFSR reseeding to reduce scan-in transitions as the low fill rates make it possible to deliver identical test data to scan chains for a number of shift cycles directly from the decompress, thereby reducing the number of transitions.

In this project we propose a PRPG for LP BIST applications. The generator primarily aims at reducing the switching activity during scan loading due to its preselected toggling (PRESTO) levels. It can assume a variety of configurations that allow a given scan chain to be driven either by a PRPG itself or by a constant value fixed for a given period of time. Not only the PRESTO generator allows loading scan chains with patterns having low

transition counts, and thus significantly reduced power dissipation, but it also enables fully automated selection of its controls such that the resultant test patterns feature desired, user-defined toggling rates. We will demonstrate that this flexible programming can be further used to produce tests superior to conventional pseudorandom vectors with respect to a resultant fault-coverage-to-test-pattern-count ratio.

This project culminates in showing that the PRESTO generator can also successfully act as a test data decompress, thus allowing one to implement a hybrid test methodology that combines LBIST and ATPG-based embedded test compression. This is the first LP test compression scheme that is integrated in every way with the BIST environment and lets designers shape the power envelope in a fully predictable, accurate, and flexible fashion. As a result, it creates an environment that can be used to arrive at an efficient hybrid solution combining advantages of scan compression and logic BIST. In addition,

both techniques can complement each other to address, for example, a voltage drop caused by a high switching activity during scan testing, constraints of at-speed ATPG-produced test patterns, or new fault models.

CHAPTER 2

LITERATURE SURVEY

LBIST and test compression together has been made a vital research and development area to overcome the obstruction of test data bandwidth between tester and the chip. Other hybrid BIST schemes store deterministic patterns used to detect faults in a compressed form and use hardware to decompress them. In the paper [3], it is stated that as LBIST alone is not sufficient for the fault coverage an additional top up of deterministic test patterns are added for external tester. The combination of LBIST and deterministic ATPG is addressed which forms hybrid test patterns. These patterns can be generated using STUMPS architecture. They reduce external data tester memory and also number of pseudorandom patterns. But it has huge hardware circuitry for that consumes more area. This method also needs initial pattern storage which might not be efficient in general cases. An ATPG to minimize ATE storage requirements and bandwidth. Dynamic Compaction and allowing constraints on inputs of circuits can achieve tailored deterministic test patterns. A partially rotational can register is used in [5] for the n-detection of BIST so that unmodeled faults are detected. For selecting test patterns from ATPG a procedure is presented which reduces number of input vectors from external tester. New programmable deterministic BIST method is developed to lower storage of deterministic patterns than previous methods and increase flexibility of internal and external test. In LFSR seed more care bits can be encoded as the information is stored using compression and decompression in memory.

LFSR reseeding is an effective methodology for lessening test storage. The encoding plan in gives an approach to diminish the test power for LFSR reseeding while as yet protecting or notwithstanding enhancing the pressure that is accomplished. The square size can be effectively conformed to tradeoff test-power lessening versus equipment overhead. The proposed

plan can be utilized as a part of either a BIST situation or in test pressure plans in view of LFSR reseeding to fulfill power imperatives.

Due to high data switching in scan based test may consume more power. Full toggle scan patterns might also draw more power over mission mode. High power consumption may result in voltage noise, power droop or excessive peak power over multiple cycles which inturn leads to device damage, reduced chip reliability, short product lifetime or malfunction of the device etc.

For power reduction BIST is specifically proposed to keep power under the threshold. The test power can be reduced by preventing the transition of data at memory elements during scan shift[10]. Test patterns generated by LFSR can be masked by a test vector inhibiting scheme as not all produced lengthy vectors detect faults. So such tests are eliminated to reduce switching activity with no impact on fault coverage.

A new dimension to power aware BIST was added by low transition test pattern generators. For example, a device in consists of LFSR to feed scan chains through biasing logic and T-type flip flops. These flip flops hold the previous values until their input is asserted, the same value is scanned repeatedly until the output of biasing logic becomes 1[9]. The combination of low transition generator with three weighted pseudorandom pattern generator also reduces BIST switching activity. The BIST power consumption easily exceeds maximum rating when tested at speed, scan patterns are shifted at a programmable low speed, but only last few cycles and the capture cycle are applied at the maximum frequency. LP test compression schemes adapt to LFSR reseeding to reduce number of transitions by reducing scan-in transitions as the low fill rates deliver identical test data scan chains.

The number of transitions at the input of CUT is reduced by 25% by using bit-swapping LFSR as presented in [2]. This BS-LFSR is found to be capable of optimising power up to 63% during test. So after going through these possibilities here in our project we have integrated BS-LFSR in [2] with the Fully operating PRESTO generator which is designed from the basic

architecture of PRESTO in [1]. As the switching activity is controlled by toggle selection (TOSE) technique, by embedding bit swapping concept power is further optimised at a far better level.

2.1 BASIC STRUCTURE OF A PRESTO GENERATOR

An n -bit PRPG connected with a phase shifter feeding scan chains forms a kernel of the generator producing the actual pseudorandom test patterns. A linear feedback shift register or a ring generator can implement a PRPG. More importantly, however, n hold latches are placed between the PRPG and the phase shifter. Each hold latch is individually controlled via a corresponding stage of an n -bit toggle control register. As long as its enable input is asserted, the given latch is transparent for data going from the PRPG to the phase shifter, and it is said to be in the toggle mode. When the latch is disabled, it captures and saves, for a number of clock cycles, the corresponding bit of PRPG, thus feeding the phase shifter (and possibly some scan chains) with a constant value. It is now in the hold mode. It is worth noting that each phase shifter output is obtained by XOR-ing outputs of three different hold latches. Therefore, every scan chain remains in a low-power mode provided only disabled hold latches drive the corresponding phase shifter output.

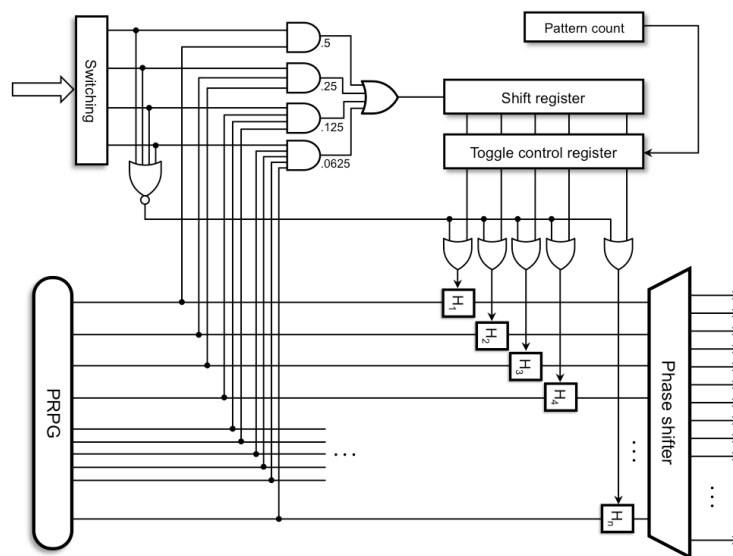


Fig 2.1: The basic structure of a PRESTO generator

As mentioned previously, the toggle control register supervises the hold latches. Its content comprises 0s and 1s, where 1s indicate latches in the toggle mode, thus transparent for data arriving from the PRPG. Their fraction determines a scan switching activity.

The control register is reloaded once per pattern with the content of an additional shift register. The enable signals injected into the shift register are produced in a probabilistic fashion by using the original PRPG with a programmable set of weights. The weights are determined by four AND gates producing 1s with the probability of 0.5, 0.25, 0.125, and 0.0625, respectively. The OR gate allows choosing probabilities beyond simple powers of 2. A 4-bit register *Switching* is employed to activate AND gates, and allows selecting a user-defined level of switching activity. For example, the switching code 0100 will set to 1, on the average, 25% of the control register stages, and thus 25% of hold latches will be enabled. Given the phase shifter structure, one can assess then the amount of scan chains receiving constant values, and thus the expected toggling ratio. An additional 4-input NOR gate detects the switching code 0000, which is used to switch the LP functionality off. It is worth noting that when working in the weighted random mode, the switching level selector ensures statistically stable content of the control register in terms of the amount of 1s it carries. As a result, roughly the same fraction of scan chains will stay in the LP mode, though a set of actual low toggling chains will keep changing from one test pattern to another. It will correspond to a certain level of toggling in the scan chains. With only 15 different switching codes, however, the available toggling granularity may render this solution too coarse to be always acceptable, additional features that make the PRESTO generator fully operational in a wide range of desired switching rates.

CHAPTER 3

PROJECT DESCRIPTION

3.1 INTRODUCTION

Generally to test any circuit, test patterns are generated and are fed to the circuit-under-test. Here also we generate pseudorandom test patterns by using a linear feedback shift register and are applied to the circuit of interest. So, it is very important to choose apt test pattern generator. In most of the cases, LFSR is used as a pattern generator and we have also chosen it here. Test vectors produced by LFSR are directly not applied to CUT. They are processed or modified accordingly, so that non - repetitive patterns are being fed to test CUT for least possible switching activity. The necessity of the proper choice of an LFSR is shown here, by designing a BIST for the ISCAS benchmarks as the complexity of VLSI circuits is constantly increasing, so there is a need to use built-in self-test (BIST). Hence for more power optimization we use Bit Swapping LFSR here. Built-in self-test enables the chip to test itself and evaluate the response of circuit. Thus, the very complex and expensive external ATE (Automatic Test Equipment) can be omitted completely and its complexity can be significantly reduced.

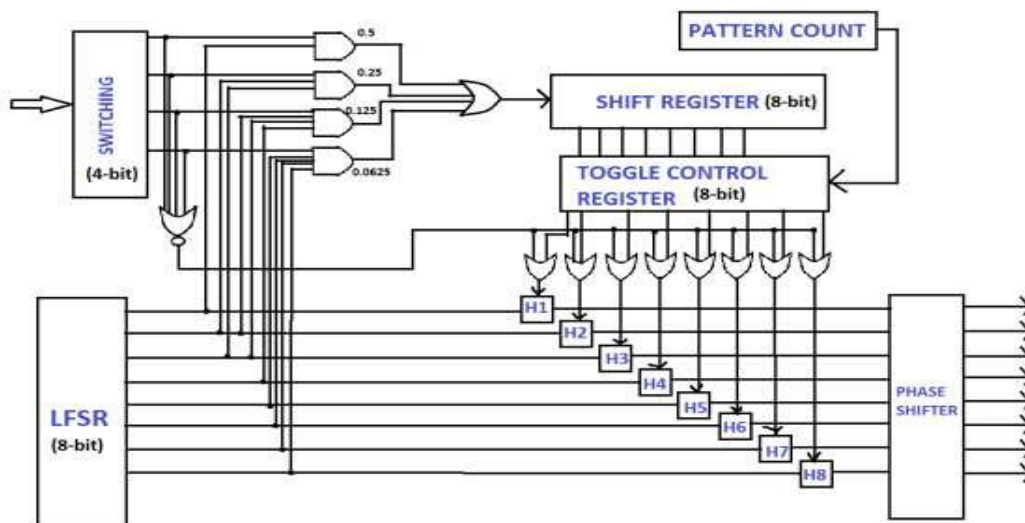


Fig 3.1: Presto generator

The basic architecture of a preselected toggling generator consists of an n-bit PRPG feeding scan chains connected with a phase shifter forms a kernel of the generator producing the pseudorandom test patterns. A linear feedback shift register or a ring generator can be used to implement a PRPG. And n hold latches are placed between the PRPG that is LFSR and the phase shifter. Each hold latch is individually controlled with the corresponding stage of an n-bit toggle control register. As long as its enable input is high, the latch enters into toggle mode and it is transparent for data fed from PRPG to the phase shifter. When the latch is disabled or low, it captures and saves the corresponding bit of PRPG for a number of clock cycles feeding the phase shifter and the same scan chains with a constant value. It will now be in the hold mode. XORed outputs of three different hold latches are given to the phase shifter and the output is NOTed. Therefore, all scan chains remain in low-power mode only when disabled hold latches drive the corresponding phase shifter output the toggle control register controls the hold latches. It comprises of 0s and 1s, where 1 indicates latch is in the toggle mode then transparent for data arriving from the PRPG. Their fractions determine the scan switching activity.

Here we use 8-bit LFSR with shift register and toggle control register. The switching will be fed with a 4-bit input. The control register is loaded once per pattern with the data of an additional shift register. The enable signals inserted into the shift register are produced in a probabilistic fashion by using the original PRPG with a set of weights that can be programmed. The weights are determined by one among four AND gates produce 1s with probabilities 0.5, 0.25, 0.125, and 0.0625, respectively. The OR gate allows choosing probability above powers of 2. A 4-bit switching register is employed to activate AND gates, and allow us to select user-defined level of switching activity. If the phase shifter structure is known then the expected toggling ratio is achieved.

When the switching input is given as 0110, then the LFSR out will be a 8-bit data 11100010. Four AND gates will emit 0100 respectively. That means second AND gate is triggered and only 25% of hold latches will be

activated as its probability is 0.25, remaining will be disabled. Among eight hold latches H1 will be active and other are not. By XORing the outputs of latches properly, the phase shifter output will be for given data 11000000. Additional 4-input NOR gate at switching register detects the switching input 0000, which is used to switch off the Low power functionality of the generator. It is worth noting when it is working in the weighted random mode; the switching level selector maintains statistically stable content of the control register that is the amount of 1s carried. This results the same fraction of scan chains to stay in the LP mode, though a set of actual low toggling chains keep changing from one test pattern to another. It corresponds to the certain level of toggling in those scan chains. The available toggling granularity renders this solution not suitable as it is too rough.

3.2 ARCHITECTURE OF MODIFIED LFSR

Below figure shows the implementation of general architecture of Bit Swapping LFSR, which reduces the average and weighted switching activity during test operation that reduces number of transitions. Design of Bit Swapping LFSR is based on the captions which are used to describe the Transition activity in Bit Swapping LFSR. We can easily define the process of switching and transition of bits that take place in pattern generation. This description will give surprisingly good results in order to optimize power. For the test-per clock and test-per-scan we have different lemmas. There are several techniques to reduced power consumption.

There is direct technique, if we used reduced frequency during testing it will reduced power dissipation. In this direct technique there is no requirement of extra hardware. Second direct technique to reduced power consumption is to apply appropriate testing planning by doing portioning of CUT into block. These direct techniques increase the timing of testing and are not applicable for reduction peak power. In contrast to the other techniques Bit- Swapping LFSR reduce average and peak power dissipated by CUT. Here we have used 8-bit BS-LFSR which consists of eight D flip-flops, two MUX's and an XOR gate.

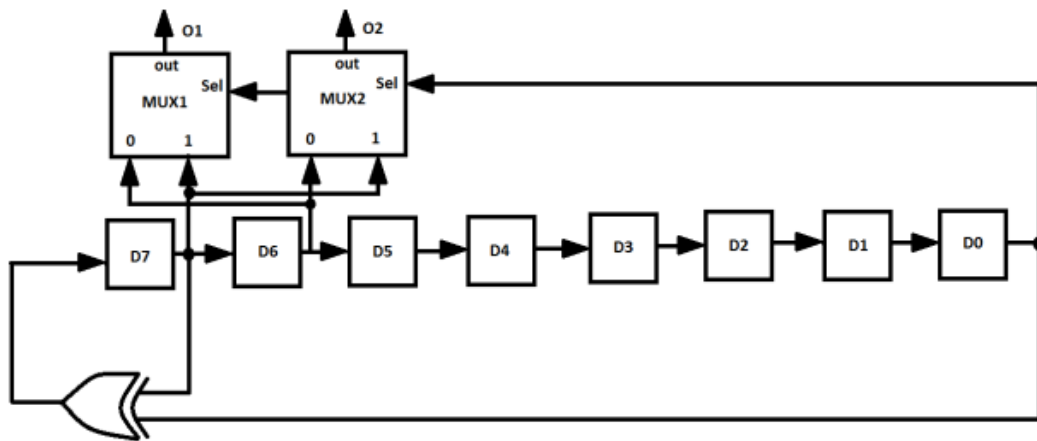


Fig 3.2: Architecture of bit swapping LFSR

3.3 OBSERVATION OF BS-LFSR

Bit-Swapping LFSR is based on some observations depending upon number of transition produced by LFSR at the output. In modified form of normal LFSR; we have used swapping property between every pair of adjacent cells of normal LFSR for designing Bit-Swapping LFSR. General observation for LFSR, For any n -bit maximum length LFSR that starts with any seed and runs for 2^n clock cycles until it returns to the starting seed value, then the total number of transitions T_{total} that occurs is given by the equation $T_{total} = n \times 2^{(n-1)}$. For swapping Bits in BS-LFSR ; If we take any n -bit maximal length LFSR where n is greater than 2 and LFSR is modified with swapping arrangement, we consider one of its outputs O_{th} i.e. first bit is the selection line that will swap two neighboring bits elsewhere in the LFSR at specific value of selection line. If we set the value of selection line at n for swapping then O_{th} is odd and bit $n=0$, bit 7 will be swapped with bit 6, bit 6 with bit 5...bit $n-1$ with bit $n-2$. In all cases the selection line has no effect on swapping operation. If bit $n=1$, then no swapping is performed. In this case, Modified LFSR will generate exactly same as normal (unmodified) LFSR whereas the order of generated test pattern by modified LFSR will be different. Swapping arrangement will save a number of transitions, swapped bits will save number of transition equal to $T_{Saved} = 2(n-2)$. In contrast, general

observation two bits (un-swapped bits) originally produced $2 \times 2^{(n-1)}$ so after swapping the swap bit will therefore save $T_{\text{Saved}} = 2^{(n-2)} / (2 \times 2^{(n-1)}) = 25\%$.

3.4 PROPOSED TOSE GENERATOR WITH BS-LFSR

For the generation of more flexible test patterns we add few blocks to the basic architecture of TOSE generator so that less toggling and low power consumption is achieved. By preserving the operational principles of the basic architecture, proposed architecture uses bit swapping linear feedback shift register and splits up the shifting period of each and every test pattern into a sequence of alternating hold and toggle intervals. To operate the generator back and forth between the two states, a T-type flip-flop is used that switches when there is 1 on its data input. If flip flop is set to 0, the generator enters into hold mode with all latches temporarily disabled irrespective of the content of control register. This can be achieved by placing AND gates onto the control register outputs to freeze all phase shifter inputs. This property is critical in System on Chip design where only a single scan chain crosses a given core, and its abnormal toggling will cause unacceptable heat dissipation locally that is only reduced by temporary hold periods.

If the T flip-flop is in the toggle period, then the enabled latches will be able to pass test data through control register from the PRPG to the scan chains. Two additional 4-bit Hold and Toggle registers are used to determine how long the generator remains either in the hold mode or in the toggle mode, respectively. For terminating either mode, the T flip-flop input should be 1. Then weighted pseudorandom signals are produced in a similar fashion to the weighted logic which is used to feed the shift register.

Four 3.2 input multiplexers routing data from the Toggle and Hold registers are also controlled by T-flip flop. It allows the user to select a source to control data that can be used in the next cycle for changing the operational mode of the generator. For example, when it is in the toggle mode, the input multiplexers observe the Toggle control register (TCR) if the weighted logic output is 1, then flip-flop toggles as a result all hold latches freeze in the

previous state. They will remain in this state until the weighted logic output will change to 1. The random occurrence of this event is related to the content of the Hold register that determines when to terminate the hold mode.

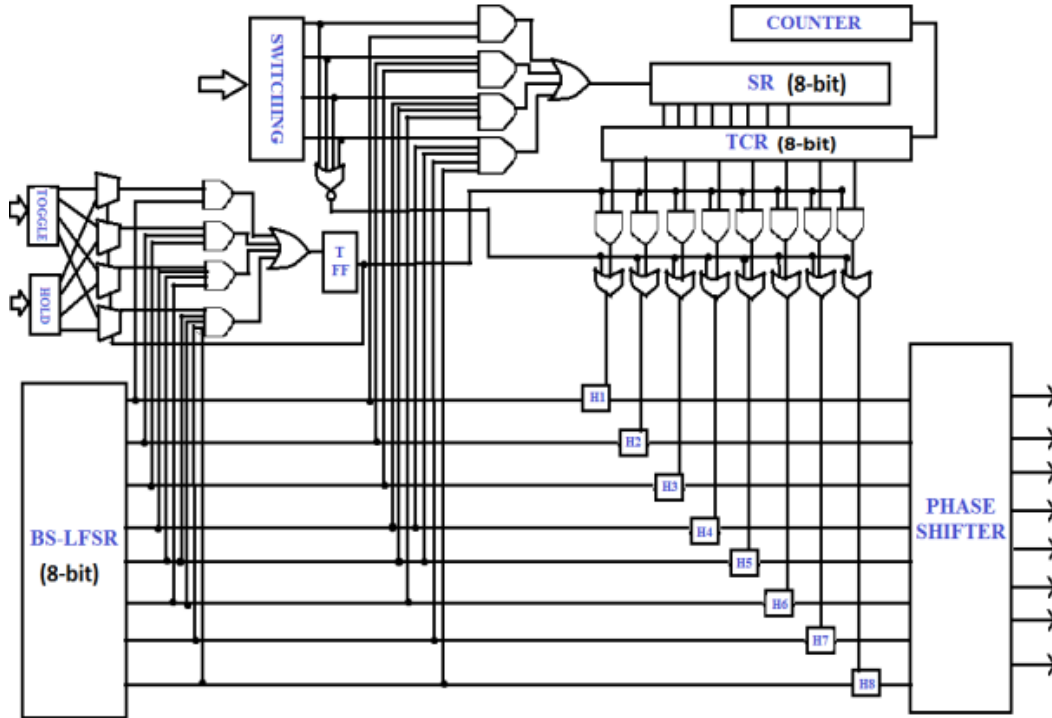


Fig 3.3: Architecture of TOSE generator using BS-LFSR

This fully operational version of the TOSE generator using BS-LFSR reduces power consumption with less switching activity and enhanced fault coverage. In the next section too test circuits; we have used standard benchmark circuits as circuit under test. ISACS benchmark circuits are used for the analysis of fault coverage and power consumption. The ISCAS'89 benchmarks are a set of 31 digital sequential circuits. These benchmark circuits were distributed on tape to participants of the Special Session on Sequential Test Generation. Among 31 circuits we took three s27, s208 and s298. "s" denotes synchronous sequential circuit and the number represents the number of incorrect lines in the circuit primitives. s208 is a frequency divider circuit with 11 primary inputs and 2 primary outputs. Whereas s298 is a traffic light controller circuit with 3 inputs, 6 outputs and 133 gates.

CHAPTER 4

RESULTS AND DISCUSSIONS

4.1 BASIC PRESTO GENERATOR RESULTS

In the initial phase, the basic Presto Generator circuits is simulated and timing analysis result is illustrated in figure 4.1.

4.1.1 Basic PRESTO Generator Simulation Results

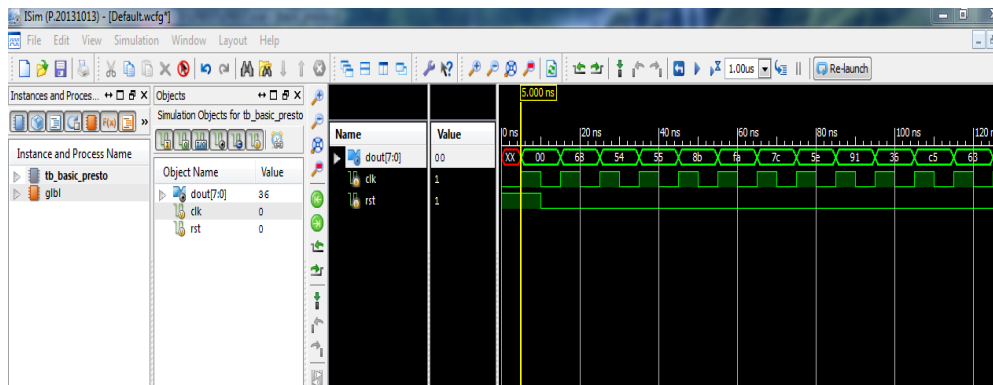


Fig:4.1:Basic Presto Generator results

4.1.2 Basic PRESTO Generator Synthesis Results

Device Utilization Summary (estimated values)				[...]
Logic Utilization	Used	Available	Utilization	
Number of Slices	9	2448	0%	
Number of Slice Flip Flops	16	4896	0%	
Number of 4 input LUTs	8	4896	0%	
Number of bonded IOBs	10	172	5%	
Number of GCLKs	1	24	4%	

Fig:4.2:Basic Presto Generator synthesis results

Timing Summary:

Speed Grade: -5

Minimum period: 2.148ns (Maximum Frequency: 465.593MHz)
 Minimum input arrival time before clock: 2.780ns
 Maximum output required time after clock: 4.040ns
 Maximum combinational path delay: No path found

4.2 PROPOSED PRESTO GENERATOR

The simulation and synthesis results of proposed PRESTO generator architecture are presented in figure 4.3 and table 4.1.

4.2.1 Proposed PRESTO Generator Simulation Results

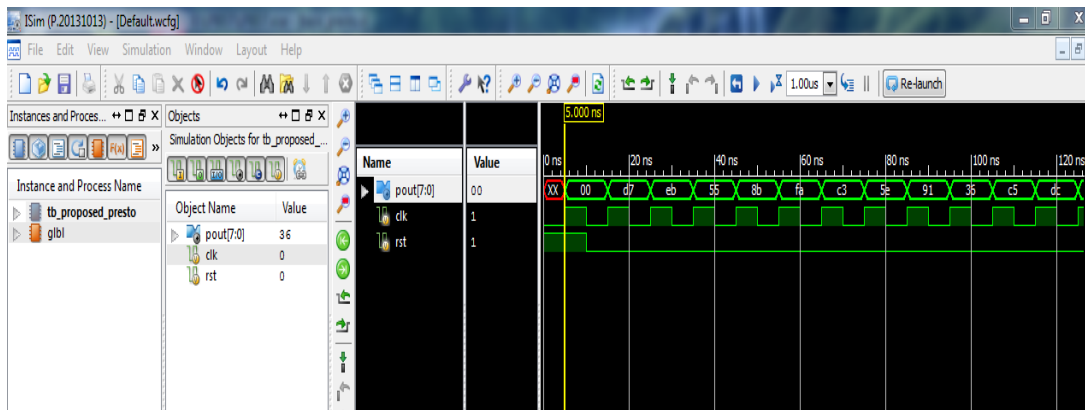


Fig:4.3:Proposed Presto Generator simulation results

4.2.2 Proposed PRESTO Generator Synthesis Results

Table 4.1 Synthesis Results for PRESTO architecture

Device Utilization Summary (estimated values)				
Logic Utilization	Used	Available	Utilization	
Number of Slices	10	2448	0%	
Number of Slice Flip Flops	16	4896	0%	
Number of 4 input LUTs	10	4896	0%	
Number of bonded IOBs	10	172	5%	
Number of GCLKs	1	24	4%	

Timing Summary:

Speed Grade: -5

Minimum period: 3.198ns (Maximum Frequency: 312.695MHz)
 Minimum input arrival time before clock: 2.780ns
 Maximum output required time after clock: 4.040ns
 Maximum combinational path delay: No path found

4.3 BASIC PRESTO BASED BIST FOR CARRY SELECT ADDER

Table 4.2 Synthesis Results for basic Presto CSA

Device Utilization Summary				
Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Flip Flops	104	4,896	2%	
Number of 4 input LUTs	112	4,896	2%	
Number of occupied Slices	89	2,448	3%	
Number of Slices containing only related logic	89	89	100%	
Number of Slices containing unrelated logic	0	89	0%	
Total Number of 4 input LUTs	145	4,896	2%	

Delay results

Timing Summary:

Speed Grade: -5

Minimum period: 5.367ns (Maximum Frequency: 186.333MHz)
 Minimum input arrival time before clock: 8.952ns
 Maximum output required time after clock: 4.114ns
 Maximum combinational path delay: No path found

4.3.1 Power Consumption Requirements

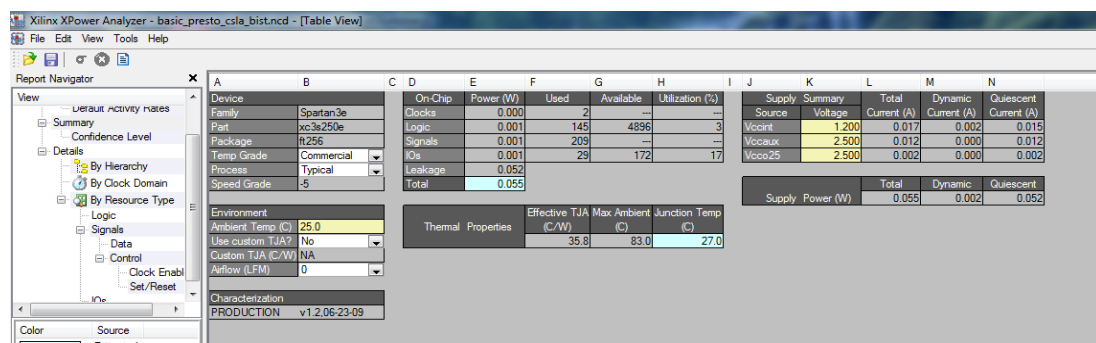


Fig.4.4:Power Consumption results of Presto based CSA

4.4 PROPOSED PRESTO BASED BIST FOR CARRY SELECT ADDER

The device utilization summary is given in table 4.1 for BIST architecture of carry select adder.

Table 4.3 Synthesis Results for proposed presto carry select adder

Device Utilization Summary				
Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Flip Flops	96	4,896	1%	
Number of 4 input LUTs	114	4,896	2%	
Number of occupied Slices	86	2,448	3%	
Number of Slices containing only related logic	86	86	100%	
Number of Slices containing unrelated logic	0	86	0%	
Total Number of 4 input LUTs	147	4,896	3%	

Delay

Timing Summary:

Speed Grade: -5

Minimum period: 5.367ns (Maximum Frequency: 186.333MHz)
 Minimum input arrival time before clock: 8.952ns
 Maximum output required time after clock: 4.114ns
 Maximum combinational path delay: No path found

4.4.1 Power Results for proposed presto BIST carry select order

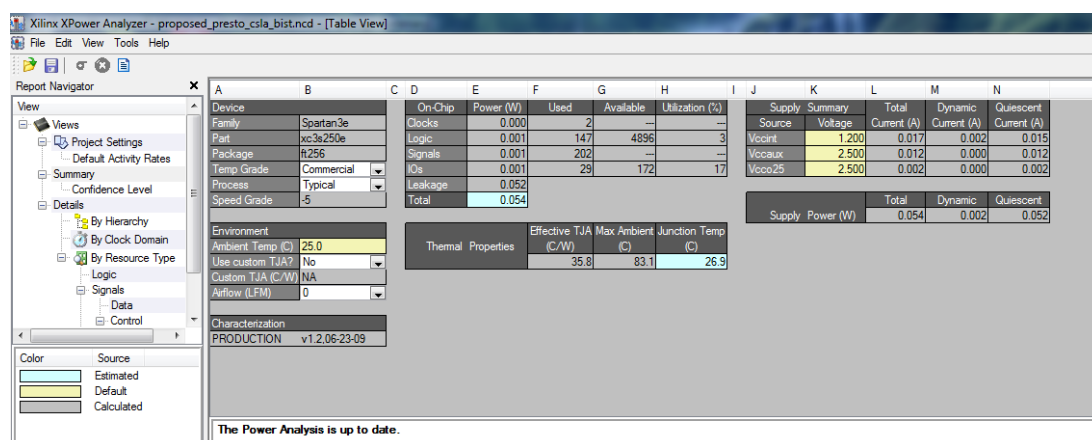


Fig.4.5 Power analysis of proposed presto BIST carry select order

CHAPTER 5

SUMMARY AND CONCLUSION

5.1 SUMMARY

The proposed PRESTO architecture comprised of a linear finite state machine (a linear feedback shift register or a ring generator) driving an appropriate phase shifter and armed with a number of features that allows this device to produce binary sequences with low toggling (switching) rates while preserving test coverage achievable by the best-to-date conventional BIST-based PRPGs with negligible impact on test application time. It is observed from the simulation results that the proposed PRESTO design has better device utilization ration and lower power consumption.

This project presents the implementation with regard to Verilog language. Synthesizing and implementation of source code is carried out on Xilinx - Project Navigator, ISE 14.7 suite.

5.2 FUTURE SCOPE AND CONCLUSION

The proposed approach aims at the concept reduced switching activity and toggling to reduce power consumption using bit swapping technique in LFSR. It is achieved by generation of non recursive test patterns by choosing proper architecture of the PRPG. The basic architecture is added with additional blocks to achieve granularity in the toggling ratio. PRPG consumes less power during testing by taking the benchmark circuit S27. In the future, power Investigation of LFSRS, Random memory access test and System on chip (soc) memory test can be implemented reduced by modifying internal architecture with reduced complexity.

REFERENCES

- [1] Abu-Issa and S. F. Quigley, "Bit-swapping LFSR for low-power BIST," *Electron. Lett.*, vol. 44, no. 6, Mar. 2008, pp. 401-402.
- [2] Abu-Issa and S. Quigley, "Bit-swapping LFSR and scan-chain ordering: A novel technique for peak- and average-power reduction in scan-based BIST," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 28, no. 5, pp. 755–759, May 2009.
- [3] Das D and N. A. Touba, "Reducing test data volume using external/LBIST hybrid test patterns," in *Proc. Int. Test Conf. (ITC)*, 2000, pp. 115-122.
- [4] Dorsch R and H. Wunderlich, "Tailoring ATPG for embedded testing," in *Proc. Int. Test Conf. (ITC)*, 2001, pp. 530-537.
- [5] Feng Liang, Luwen Zhang, Shaochong Lei, Guohe Zhang, Kaile Gao, and Bin Liang "Test Patterns of Multiple SIC Vectors: Theory and Application in BIST Schemes" in *IEEE transactions on very large scale integration (vlsi) systems*, Vol. 21, No. 4, April 2013
- [6] Gerstendorfer S and H. Wunderlich, "Minimized power consumption for scan-based BIST," in *Proc. Int. Test Conf. (ITC)*, 1999, pp. 77-84.
- [7] Girard P, C. Landrault, S. Pravossoudovitch, A. Virazel, and H.-J. Wunderlich, "High defect coverage with low-power test sequences in a BIST environment", *IEEE Design Test*, vol. 19, no. 5, pp. 44–52, Sep. 2002
- [8] Girard P, L. Guiller, C. Landrault, S. Pravossoudovitch, J. Figueras, S. Manich, P. Teixeira, and M. Santos, "Low energy BIST design: Impact of the LFSR TPG parameters on the weighted switching activity," *Proc. IEEE Int. Symp. Circuits Syst.*, vol. 1, Jul. 1999, pp. 110–113.
- [9] Hakmi A.W, S. Holst, H. Wunderlich, J. Schloffel, F. Hapke, and A. Glowatz, "Restrict encoding for mixed-mode BIST," in *Proc. 27th IEEE VLSI Test Symp. (VTS)*, May 2009, pp. 179-184.
- [10] Ichino K, T. Asakawa, S. Fukumoto, K. Iwasaki, and S. Kajihara, "Hybrid BIST using partially rotational scan," in *Proc. 10th Asian Test Symp. (ATS)*, 2001, pp. 379-384.

- [11] Jas, C. V. Krishna and N. A. Touba, "Weighted pseudorandom hybrid BIST," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 12, no. 12, pp. 1277–1283, Dec. 2004.
- [12] Krishna ATV and N. A. Touba, "Reducing test data volume using LFSR reseeding with seed compression," in Proc. Int. Test Conf. (ITC), 2002, pp. 321–330.
- [13] Rajski G, J. Tyszer, G. Mrugalski, and B. Nadeau-Dostie, "Test generator with preselected toggling for low power built-in self-test," in Proc. IEEE 30th VLSI TestSymp. (VTS), Apr. 2012, pp. 1–6.
- [14] Rajski G, J. Tyszer, G. Mrugalski, and B. Nadeau-Dostie, "Test generator with preselected toggling for low power built-in self-test", Proc. IEEE30th VLSI Test Symp. (VTS), Apr. 2012, pp. 1–6.
- [15] Wang S and S. K. Gupta, "LT-RTPG: A new test-per-scan BIST TPG for low switching activity," IEEE Trans. Computer-Aided Design Integrated Circuits Syst., vol. 25, no.8, Aug. 2006, pp. 1565–1574,
- [16] Zorian Y "A distributed BIST control scheme for complex VLSI devices," in 11th Annu. IEEE VLSI Test Symp. Dig. Papers, Apr. 1993, pp. 4–9.