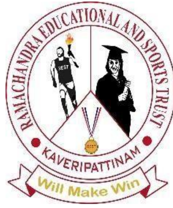
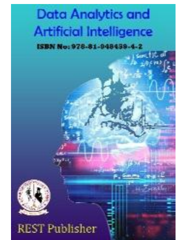


Data Analytics and Artificial Intelligence
Vol: 3(6), 2023
REST Publisher; ISBN: 978-81-948459-4-2
Website: http://restpublisher.com/book-series/daai/



Efficient Fault Detection Using Voter Logic Circuit in VLSI Testing

Vinoka M*, Durai Raj S
Madurai Institute of Engineering and Technology,Pottapalayam, Sivagangai,, Tamil Nadu, India
Corresponding Author Email: vinoselvi21@gmail.com

Abstract: *Defect rate in Nanoelectronics is much higher than conventional CMOS circuits. Hardware redundancy can be a suitable solution for fault tolerance in nano level. A voter circuit is a part of a redundancy-based fault tolerant system that enables a system to continue operating properly in the event of one or more faults within its components. Robustness of the voter circuit defines the reliability of the fault tolerant system. This work provides simulation results and analysis of a fault tolerant voter circuit. In a Triple Modular Redundant (TMR) system, the robustness of the voter circuit has been improved. For this purpose, redundancy at logic level has been added. Extra hardware is added to override the effects of a failed component. Static Hardware Redundancy for immediate masking of a failure. Use three processors and vote on the result. The wrong output of a single faulty processor is masked. Dynamic Hardware Redundancy Spare components are activated upon the failure of a currently active component. Hybrid Hardware Redundancy is a combination of static and dynamic redundancy techniques. In this technique a new voter for multiple fault detection is proposed with minimum area cost. The propose voter coded in VHDL and simulated using Xilinx 12.1.*

1. INTRODUCTION

Since the electronics technology accomplished higher levels of integration into a single Silicon chip that led to Large Scale Integration (LSI), which preceded VLSI, the applications of electronic systems have experienced an almost unlimited expansion. However, despite the many advantages provided by VLSI, the inherent high integration level started to necessitate very sophisticated testing strategies in order to verify the correct device operation. As the electronics market stimulated the use of VLSI in a variety of tasks from critical military applications to consumer products, the reliability of the products, functioning gained an escalating importance. The expanding demand for ASIC applications led to development of more sophisticated Computer Aided Design (CAD) tools; which have shown most significant progress in layout and simulation, with yet more inferior improvement in testing. This consequently leads to designs with superior complexity, but which are in contrast extremely difficult to test effectively. Moreover, due to the low volume attribute of ASICs, the high-test costs cannot be retaliated with large amounts of mass production. Circuit modules are easy to test when only a few test vectors are needed to be sure that the module is fault free. Generating tests for circuit modules is beyond the scope of this class. We instead will take the point of view that a circuit structure that gives easy access to the internal nodes of the circuit can usually be tested with only a few test vectors. Access to internal nodes is easy when you construct the circuit out of discrete components on a prototyping board. This is not physically possible inside an integrated circuit. We are restricted to using the signals on the bonding pads of the circuit. Thus, the designer must take testing into account when deciding what signals will be on the pads. Often, additional test pads are added to the circuit which can be used for testing purposes before packaging.

Evolution of Testing: In the early times of electronics engineering, when systems were constituted from discrete components, testing of digital systems comprised three distinct phases:
1. Each discrete component was tested for concordance to its specifications.
2. The components were assembled into more complex digital elements (i.e. flip flops etc.), and these were tested for correct functionality Testing and Built in Self-Test.

Copyright@ REST Publisher

3. The higher-level system was built up and was tested for functionality. As the systems acquired higher complexity, the 3rd phase began to become increasingly difficult to accomplish and other means of system verification were begun to be sought. In R.D. Eldred suggested another way, which is well-known and used as structural test at present, in order to test the hardware of a system instead of the burdensome functional test.

The first applications of the proposed structural test was to discrete components on Printed Circuit Boards (PCBs), which then began to be applied to ICs as the electronics technology developed into higher levels of integration. Though the problems of IC testing was not very much different from that of the PCBs, the objective of testing had then changed to discard the faulty units rather than locating the defective components and replace them. In the case of Small Scale Integration (SSI) and Medium Scale Integration (MSI), the problems were relatively as simple as PCB testing, since:

1. Internal nodes of the devices were easily controlled and observed from the primary inputs and outputs of the devices.
2. The simplicity of circuit functions permitted the use of exhaustive testing.

Functional and Structural Testing: Before structural testing was proposed, digital systems were tested to verify their compliance with their intended functionality, i.e., in this philosophy, a multiplier would be tested whether it would multiply and so forth. This testing philosophy is termed as functional testing, which can be defined as, applying a series of determined meaningful inputs to check for the correct output responses in terms of the device functionality. Although this methodology imparts a good notion of circuit functionality, under the presence of a definitive fault model, it is very difficult to isolate certain faults in the circuits in order to verify their detection. With the proposal of structural testing, which might be defined as, consideration of possible faults that may occur in a digital circuit and applying a set of inputs tailored for detecting these specified faults. . As obvious structural testing relies on the fault Testing and Built in Self-Test models described for the Device Under Test (DUT), and any result obtained in this manner is unworthy without a proper description of used fault models. Fault models and fault simulation techniques developed as a result of the above-described technique will be elaborated.

Controllability and Observability: These terms, were introduced in 1970s in order to describe the ease – or difficulty – of testing the nodes of a digital circuit. Controllability is a measure of how easily – or hardly – can a node of a digital circuit can be driven from the accessible (primary) inputs. Observability, is a measure of how easily the logical value of a given node can be propagated to the observable (primary) outputs. In general, controllability decreases as the distance between the to be controlled node and primary inputs – i.e. the number of internal gates between primary inputs and the node – increases, and observability decreases as the distance from primary outputs increases. A representative plot for observability and controllability can be demonstrated as in figure.

Need for Testing: Circuit manufacturers must thoroughly test their products before delivering them to customers. The causes of circuit failure can be divided into two main categories: design errors and manufacturing defects. Design errors are caused by errors in the layout. If the errors can be eliminated by changing the layout, then it is considered a design error. We normally attempt to detect design errors by simulating the circuit and testing the simulation. No simulation can perfectly predict what a real circuit will do. For example, most simulations cannot predict whether or not latch up will occur. Thus, it is necessary to test a real circuit before we can be sure that all design errors have been fixed.

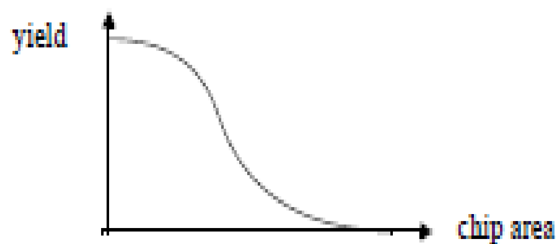


FIGURE 1. chip area vs yield

Manufacturing defects are caused by random variations in the manufacturing process that can cause malfunctions in circuit components. These defects can occur even in circuits that have no design errors.Most modern processes have an average defect density of around 1 defect per cm2. As the chip area increases, the probability of a defective chip increases and the yield (probability of no defects) decreases .Chip is yield for performance reasons,

Copyright@ REST Publisher

we would like to have the chip as large as possible. This means that a considerable fraction of large chips will be defective and we must test each chip to be sure that it is defect.
Before proceeding further, let us be clear on what we mean by testing a circuit. Suppose we have a digital circuit which we wish to test. We will give it a set of inputs (usually called "test vectors") and we will observe the outputs to see if they are correct.

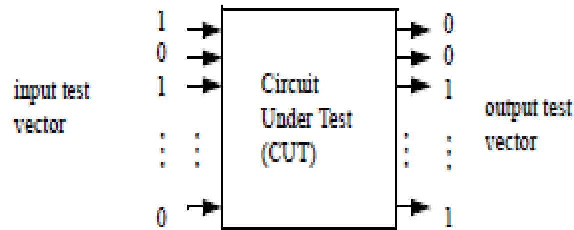


FIGURE 2. CUT

1. Is the output signal equivalent to the correct logic value is it functionally, correct?
2. Does the output reach its correct value in a timely fashion, is it fast enough?3. Does the circuit consume too much power? There might be other test criterion as well, such as susceptibility to temperature variations, radiation or mechanical vibration, but we will not consider them here.

Design for Testability: We can significantly improve circuit testability by decomposing the design into smaller circuit modules that can be tested more easily. Circuit modules are easy to test when only a few test vectors are needed to be sure that the module is fault free. Generating tests for circuit modules is beyond the scope of this class. We instead will take the point of view that a circuit structure that gives easy access to the internal nodes of the circuit can usually be tested with only a few test vectors. Access to internal nodes is easy when you construct the circuit out of discrete components on a prototyping board. This is not physically possible inside an integrated circuit. We are restricted to using the signals on the bonding pads of the circuit. Thus, the designer must take testing into account when deciding what signals will be on the pads. Often, additional test pads are added to the circuit which can be used for testing purposes before packaging.

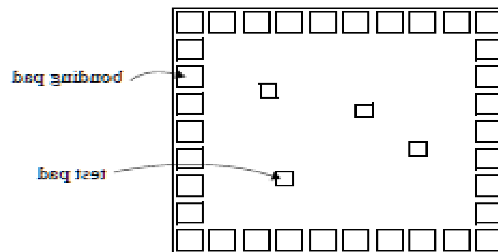


FIGURE 3. Test pads

Even with test pads, the number of pads is going to be far less than the number of nodes on the chip. Thus, we must have a testing strategy that works when we have only a limited number of nodes to use for test vectors.
We want all the nodes in our circuit to be "close" to nodes in the test vector. A node has good controllability if it is close to a test input and a node has good observability if it is close to a test output. There are no precise definitions of "close" at the transistor level.

Fault and Its Classification: A **fault i**n a system is some deviation from the expected behaviour of the system: a malfunction. Faults may be due to a variety of factors, including hardware failure, software bugs, operator (user) error, and network problems. Faults can be classified into one of three categories

Transient faults: These occur once and then disappear. For example, a network message doesn't reach its destination but does when the message is retransmitted.

Copyright@ REST Publisher

Intermittent faults: Intermittent faults are characterized by a fault occurring, then vanishing again, then reoccurring, then vanishing. These can be the most annoying of component faults. A loose connection is an example of this kind of fault.
Permanent faults: This type of failure is persistent: it continues to exist until the faulty component is repaired or replaced. Examples of this fault are disk head crashes, software bugs, and burnt-out power supplies.

2. APPROACHES TO FAULTS

We can try to design systems that minimize the presence of faults. **Fault avoidance** is a process where we go through design and validation steps to ensure that the system avoids being faulty in the first place. This can include formal validation, code inspection, testing, and using robust hardware.
Fault removal is an ex post facto approach where faults were encountered in the system and we managed to remove those faults. This could have been done through testing, debugging, and verification as well as replacing failed components with better ones, adding heat sinks to fix thermal dissipation problems, etcetera.
Fault tolerance is the realization that we will always have faults (or the potential for faults) in our system and that we have to design the system in such a way that it will be tolerant of those faults. That is, the system should compensate for the faults and continue to function.

3. ACHIEVING FAULT TOLERANCE

Information redundancy seeks to provide fault tolerance through replicating or coding the data. For example, a Hamming code can provide extra bits in data to recover a certain ratio of failed bits. Sample uses of information redundancy are parity memory, ECC (Error Correcting Codes) memory, and ECC codes on data blocks.**Time redundancy** achieves fault tolerance by performing an operation several times. Timeouts and retransmissions in reliable point-to-point and group communication are examples of time redundancy. It is of no use with permanent faults. An example is TCP/IP's retransmission of packets.
Physical redundancy deals with devices, not data. We add extra equipment to enable the system to tolerate the loss of some failed components. RAID disks and backup name servers are examples of physical redundancy.

4. SOFTWARE DESCRIPTION

Introduction to XILINX: Xilinx Tools is a suite of software tools used for the design of digital circuits implemented using Xilinx *Field Programmable Gate Array (FPGA) or Complex Programmable Logic Device (CPLD)*. The design procedure consists of (a) design entry, (b) synthesis and implementation of the design, (c) functional simulation and (d) testing and verification. Digital designs can be entered in various ways using the above CAD tools: using a schematic entry tool, using a hardware description language (HDL) – Verilog or VHDL or a combination of both. In this lab we will only use the design flow that involves the use of Verilog HDL. The CAD tools enable you to design combinational and sequential circuits starting with Verilog HDL design specifications. The steps of this design procedure are listed below:
1. Create Verilog design input file(s) using template driven editor.
2. Compile and implement the Verilog design file(s).
3. Create the test-vectors and simulate the design (functional simulation) without using a PLD (FPGA or CPLD).
4. Assign input/output pins to implement the design on a target device.
5. Download bit stream to an FPGA or CPLD device.
6. Test design on FPGA/CPLD device

A Verilog input file in the Xilinx software environment consists of the following segments:
Header: module name, list of input and output ports.
Declarations: input and output ports, registers and wires.
Logic Descriptions: equations, state machines and logic functions.
End: endmodule. All your designs for this lab must be specified in the above Verilog input format. Note that the *state diagram* segment does not exist for combinational logic designs.

Programmable Logic Device: FPGA

In this lab digital designs will be implemented in the Basys2 board which has a Xilinx Spartan3E – XC3S250E FPGA with CP132 package. This FPGA part belongs to the Spartan family of FPGAs. These devices

come in a variety of packages. We will be using devices that are packaged in 132 pin package with the following part number: XC3S250E-CP132. This FPGA is a device with about 50K gates. Detailed information on this device is available at the Xilinx website.

Creating a New Project

Xilinx Tools can be started by clicking on the Project Navigator Icon on the Windows desktop. This should open up the Project Navigator window on your screen. This window shows (see Figure 1) the last accessed project.

Opening a project

Select **File->New Project** to create a new project. This will bring up a new project window (Figure 2) on the desktop. Fill up the necessary entries as follows:

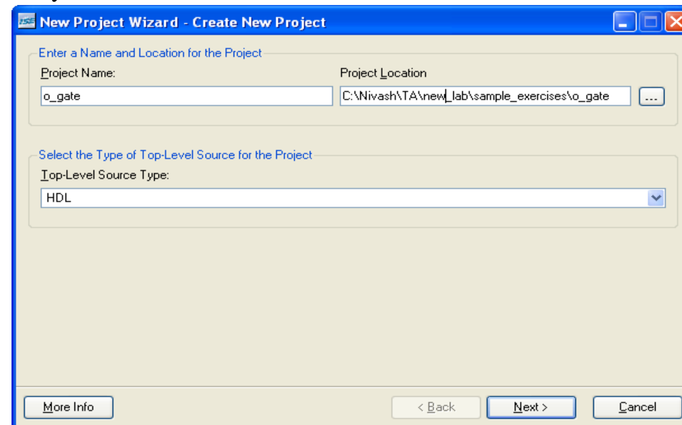


FIGURE 4. New Project Initiation window

Project Name: Write the name of your new project

Project Location: The directory where you want to store the new project (Note: DO NOT specify the project location as a folder on Desktop or a folder in the Xilinx\bin directory. Your H: drive is the best place to put it.

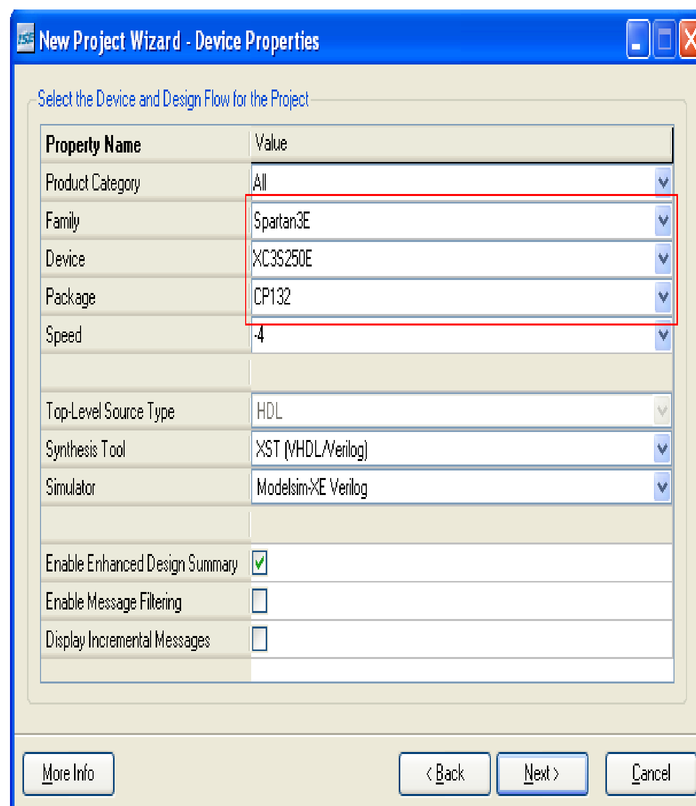


FIGURE 5. Device and Design Flow of Project

Copyright@ REST Publisher

For each of the properties given below, click on the '**value**' area and select from the list of values that appear.

Device Family: Family of the FPGA/CPLD used. In this laboratory we will be using the Spartan3E FPGA's.

Device: The number of the actual device. For this lab you may enter **XC3S250E** (this can be found on the attached prototyping board)

Package: The type of package with the number of pins. The Spartan FPGA used in this lab is packaged in CP132 package.

Speed Grade: The Speed grade is "-4".

Synthesis Tool: **XST** [VHDL/Verilog]

Simulator: The tool used to simulate and verify the functionality of the design. Modelsim simulator is integrated in the Xilinx ISE. Hence choose "Modelsim-XE Veiling" as the simulator or even Xilinx ISE Simulator can be used.

Then click on **NEXT** to save the entries.

All project files such as schematics, net lists, Verilog files, VHDL files, etc., will be stored in a subdirectory with the project name. A project can only have one top level HDL source file (or schematic). Modules can be added to the project to create a modular, hierarchical design

In order to open an existing project in Xilinx Tools, select **File->Open Project** to show the list of projects on the machine. Choose the project you want and click **OK**.

Clicking on NEXT on the above window brings up the following window:

Creating a Verilog HDL input file for a combinational logic design: In this lab we will enter a design using a structural or RTL description using the Verilog HDL. You can create a Verilog HDL input file (**. v**file) using the HDL Editor available in the Xilinx ISE Tools (or any text editor).

In the previous window, click on the

New source:

A window pops up (*Note*: "***Add to project***" *option is selected by default. If you do not select it then you will have to add the new source file to the project manually*.) Select **Verilog Module** and in the "File Name:" area, enter the name of the Verilog source file you are going to create. Also make sure that the option **Add to project** is selected so that the source need not be added to the project again. Then click on **Next** to accept the entries. This pops up the following window.

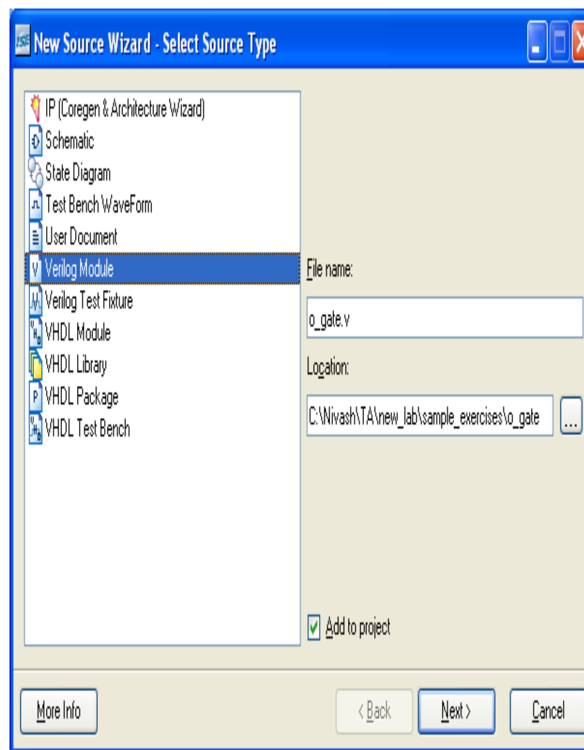


FIGURE 6. Creating Verilog-HDL source file

Editing the Verilog source file

The source file will now be displayed in the **Project Navigator** window (Figure 8). The source file window can be used as a text editor to make any necessary changes to the source file. All the input/output pins will be displayed. Save your Verilog program periodically by selecting the **File->Save** from the menu. You can also edit Verilog programs in any text editor and add them to the project directory using “Add Copy Source”.

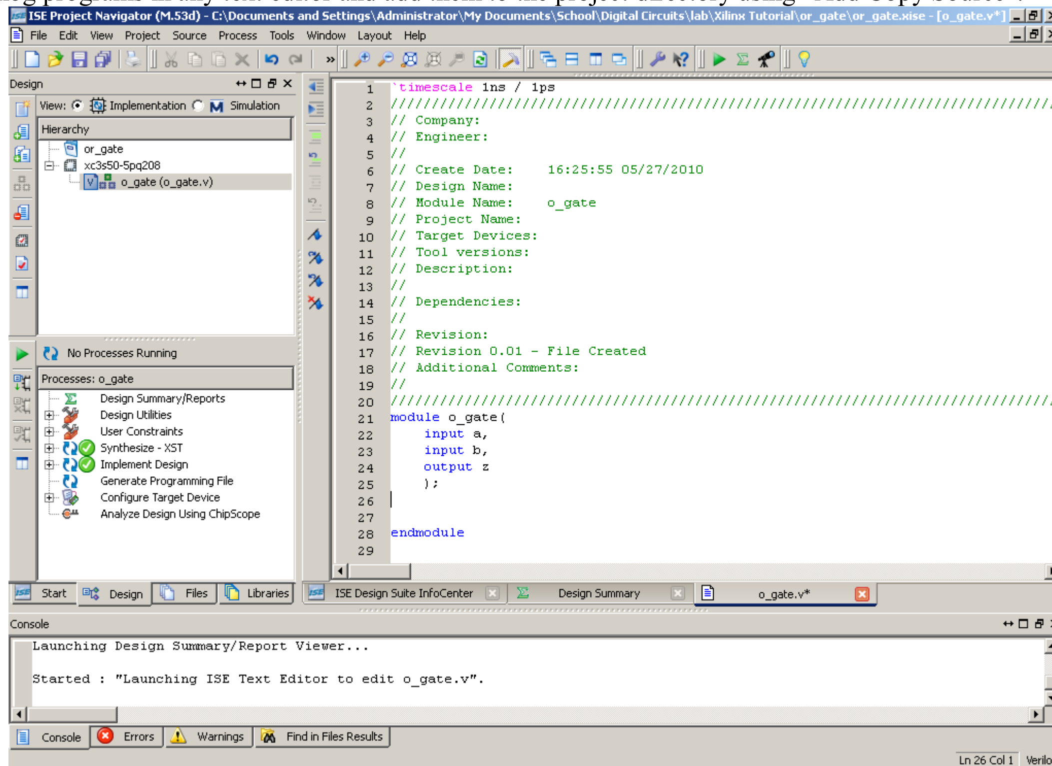


FIGURE 7. Verilog Source code editor window

Adding *Logic* in the generated Verilog Source code template:

A brief Verilog Tutorial is available in Appendix-A. Hence, the language syntax and construction of logic equations can be referred to Appendix-A.

The Verilog source code template generated shows the module name, the list of ports and also the declarations (input/output) for each port. Combinational logic code can be added to the verilog code after the declarations and before the end module line.

For example, an output **z** in an OR gate with inputs **a** and **b** can be described as,

assign *z* = ***a* | *b*;**

Remember that the names are case sensitive.

Synthesis and Implementation of the Design

The design has to be synthesized and implemented before it can be checked for correctness, by running functional simulation or downloaded onto the prototyping board. With the top-level Verilog file opened (can be done by double-clicking that file) in the HDL editor window in the right half of the Project Navigator, and the view of the project being in the **Module view** , the **implement design** option can be seen in the **process view**. **Design entry utilities** and **Generate Programming File** options can also be seen in the process view. The former can be used to include user constraints, if any and the latter will be discussed later. To synthesize the design, double click on the **Synthesize Design** option in the **Processes window**.

To implement the design, double click the **Implement design** option in the **Processes window**. It will go through steps like **Translate, Map and Place & Route**. If any of these steps could not be done or done with errors, it will place a **X** mark in front of that, otherwise a tick mark will be placed after each of them to indicate the successful completion. If everything is done successfully, a tick mark will be placed before the **Implement Design** option. If there are warnings, one can see mark in front of the option indicating that there are some warnings. One can look at the warnings or errors in the **Console** window present at the bottom of the Navigator window. ***Every time the design file is saved; all these marks disappear asking for a fresh compilation***.

At the highest level, Verilog contains stochastically functions (queues and random probability distributions) to support performance modelling.

Verilog supports abstract behavioural modelling, so can be used to model the functionality of a system at a high level of abstraction. This is useful at the system analysis and partitioning stage. Verilog supports Register Transfer Level descriptions, which are used for the detailed design of digital circuits. Synthesis tools transform RTL descriptions to gate level. Verilog supports gate and switch level descriptions, used for the verification of digital designs, including gate and switch level logic simulation, static and dynamic timing analysis, testability analysis and fault grading. Verilog can also be used to describe simulation environments; test vectors, expected results, results comparison and analysis.

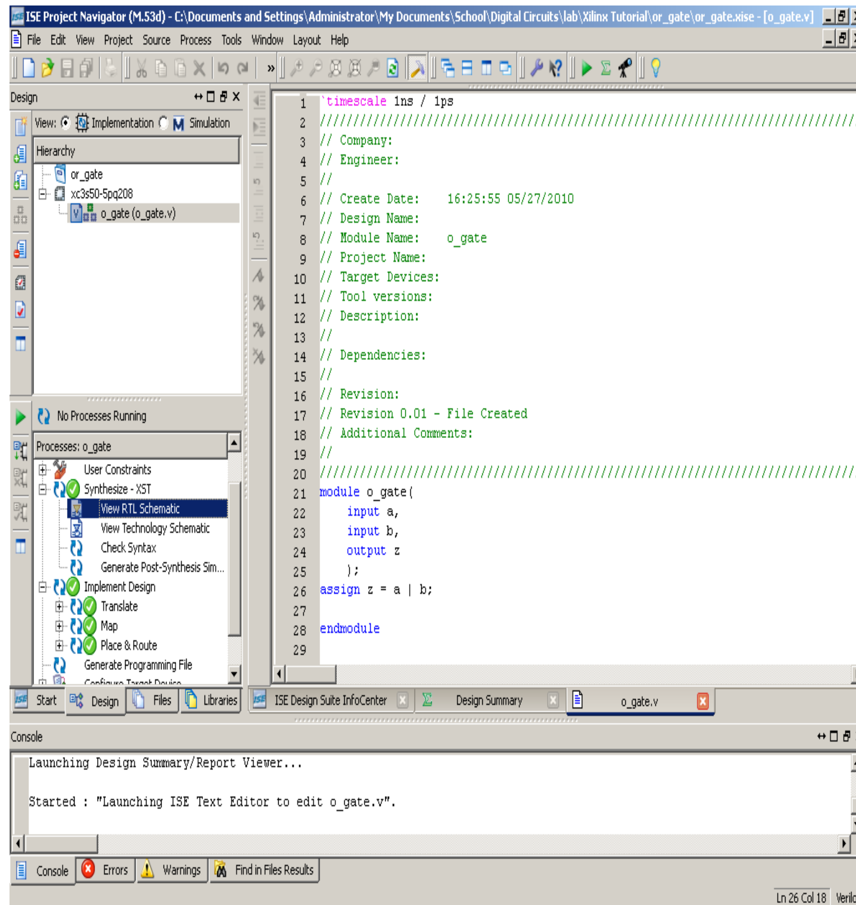


FIGURE 8. Implementing the Design

The schematic diagram of the synthesized verilog code can be viewed by double clicking View RTL Schematic under Synthesize-XST menu in the Process Window. This would be a handy way to debug the code if the output is not meeting our specifications in the proto type board.

Simulating and Viewing the Output Waveforms

Now under the **Processes window** (making sure that the testbench file in the **Sources window** is selected) expand the **ModelSim simulator Tab** by clicking on the add sign next to it. Double Click on **Simulate Behavioral Model**. You will probably receive a complier error. This is nothing to worry about – answer “No” when asked if you wish to abort simulation. This should cause ModelSim to open. Wait for it to complete execution. If you wish to not receive the compiler error, right click on **Simulate Behavioral Model** and select process properties. Mark the checkbox next to “Ignore Pre-Complied Library Warning Check”.

Copyright@ REST Publisher

To save the simulation results, Go to the waveform window of the Modelsimsimulator, Click on File -> Print to Postscript -> give desired filename and location. Note that by default, the waveform is "zoomed in" to the nanosecond level. Use the zoom controls to display the entire waveform.
Else a normal print screen option can be used on the waveform window and subsequently stored in Paint.

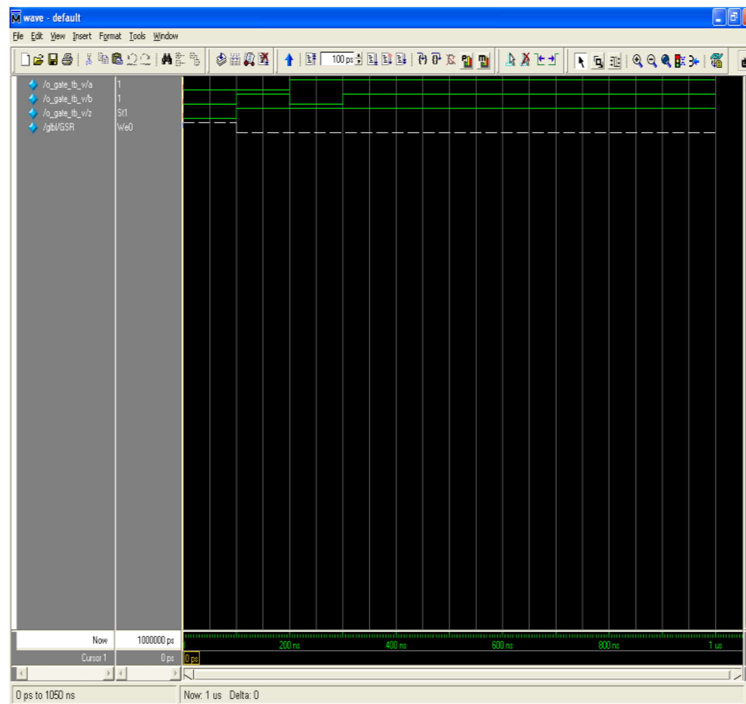


FIGURE 9. Behavioral Simulation output Waveform

5. VERILOG HDL

Verilog is a Hardware Description Language; a textual format for describing electronic circuits and systems. Applied to electronic design, Verilog is intended to be used for verification through simulation, for timing analysis, for test analysis (testability analysis and fault grading) and for logic synthesis.
The Verilog HDL is an IEEE standard - number 1364. The first version of the IEEE standard for Verilog was published in 1995. A revised version was published in 2001; this is the version used by most Verilog users. The IEEE Verilog standard document is known as the Language Reference Manual, or LRM. This is the complete authoritative definition of the Verilog HDL.
A further revision of the Verilog standard was published in 2005, though it has little extra compared to the 2001 standard. System Verilog is a huge set of extensions to Verilog, and was first published as an IEEE standard in 2005. See the appropriate Knowhow section for more details about System Verilog.
IEEE Std 1364 also defines the Programming Language Interface, or PLI. This is a collection of software routines which permit a bidirectional interface between Verilog and other languages (usually C).
Note that VHDL is not an abbreviation for Verilog HDL - Verilog and VHDL are two different HDLs. They have more similarities than differences, however.
Design Flow using Verilog: The diagram below summarises the high-level design flow for an ASIC (ie. gate array, standard cell) or FPGA. In a practical design situation, each step described in the following sections may be split into several smaller steps, and parts of the design flow will be iterated as errors are uncovered.
As a first step, Verilog may be used to model and simulate aspects of the complete system containing one or more ASICs or FPGAs. This may be a fully functional description of the system allowing the specification to be validated prior to commencing detailed design. Alternatively, this may be a partial description that abstracts certain properties of the system, such as a performance model to detect system performance bottle-necks.
Verilog is not ideally suited to system-level modelling. This is one motivation for SystemVerilog, which enhances Verilog in this area.

RTL design and testbench creation: Once the overall system architecture and partitioning is stable, the detailed design of each ASIC or FPGA can commence. This starts by capturing the design in Verilog at the register transfer level, and capturing a set of test cases in Verilog. These two tasks are complementary, and are sometimes performed by different design teams in isolation to ensure that the specification is correctly interpreted. The RTL Verilog should be synthesizable if automatic logic synthesis is to be used. Test case generation is a major task that requires a disciplined approach and much engineering ingenuity: the quality of the final ASIC or FPGA depends on the coverage of these test cases.
For today's large, complex designs, verification can be a real bottleneck. This provides another motivation for System Verilog - it has features for expediting testbench development. See the System Verilog section of Knowhow for more details.

RTL verification:

The RTL Verilog is then simulated to validate the functionality against the specification. RTL simulation is usually one or two orders of magnitude faster than gate level simulation, and experience has shown that this speed-up is best exploited by doing more simulation, not spending less time on simulation.
In practice it is common to spend 70-80% of the design cycle writing and simulating Verilog at and above the register transfer level, and 20-30% of the time synthesizing and verifying the gatesLook-ahead Synthesis
Although some exploratory synthesis will be done early on in the design process, to provide accurate speed and area data to aid in the evaluation of architectural decisions and to check the engineer's understanding of how the Verilog will be synthesized, the main synthesis production run is deferred until functional simulation is complete. It is pointless to invest a lot of time and effort in synthesis until the functionality of the design is validated.

Levels of Abstraction:

Verilog descriptions can span multiple levels of abstraction i.e. levels of detail, and can be used for different purposes at various stages in the design process,

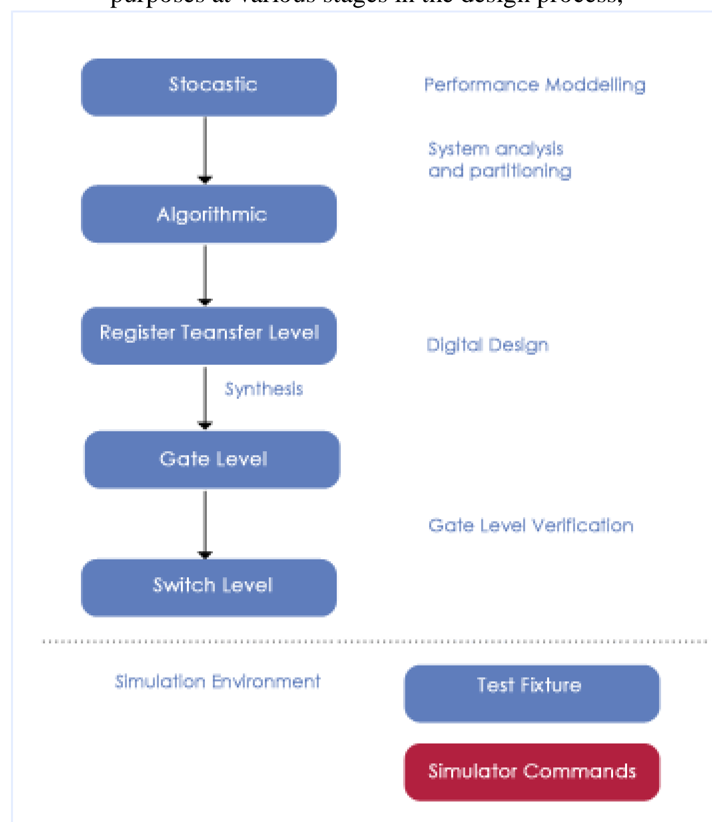


FIGURE 10. Levels of Abstraction

At the highest level, Verilog contains stochastically functions (queues and random probability distributions) to support performance modelling. Verilog supports abstract behavioural modelling, so can be used to model the functionality of a system at a high level of abstraction. This is useful at the system analysis and partitioning

stage. Verilog supports Register Transfer Level descriptions, which are used for the detailed design of digital circuits. Synthesis tools transform RTL descriptions to gate level. Verilog supports gate and switch level descriptions, used for the verification of digital designs, including gate and switch level logic simulation, static and dynamic timing analysis, testability analysis and fault grading. Verilog can also be used to describe simulation environments; test vectors, expected results, results comparison and analysis.
With some tools, Verilog can be used to control simulation e.g. setting breakpoints, taking checkpoints, restarting from time 0, tracing waveforms. However, most of these functions are not included in the 1364 standard, but are proprietary to particular simulators. Most simulators have their own command languages; with many tools this is based on TCL, which is an industry-standard tool language.

Synthesizing Verilog: Synthesis is a broad term often used to describe very different tools. Synthesis can include silicon compilers and function generators used by ASIC vendors to produce regular RAM and ROM type structures. Synthesis in the context of this tutorial refers to generating random logic structures from Verilog descriptions. This is best suited to gate arrays and programmable devices such FPGAs.

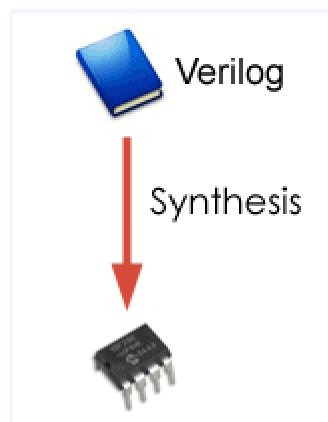


FIGURE 11. Synthesizing Verilog

Synthesis is not a panacea! It is vital to tackle High Level Design using Verilog with realistic expectations of synthesis. The definition of Verilog for simulation is cast in stone and enshrined in the Language Reference Manual. Other tools which use Verilog, such as synthesis, will make their own interpretation of the Verilog language. There is an IEEE standard for Verilog synthesis (IEEE Std. 1364.1-2002) but no vendor adheres strictly to it.

There are currently three kinds of synthesis:

- behavioral synthesis
- high-level synthesis
- RTL synthesis

There is some overlap between these three synthesis domains. We will concentrate on RTL synthesis, which is by far the most common. The essence of RTL code is that operations described in Verilog are tied to particular clock cycles.

6. SIMULATION RESULTS

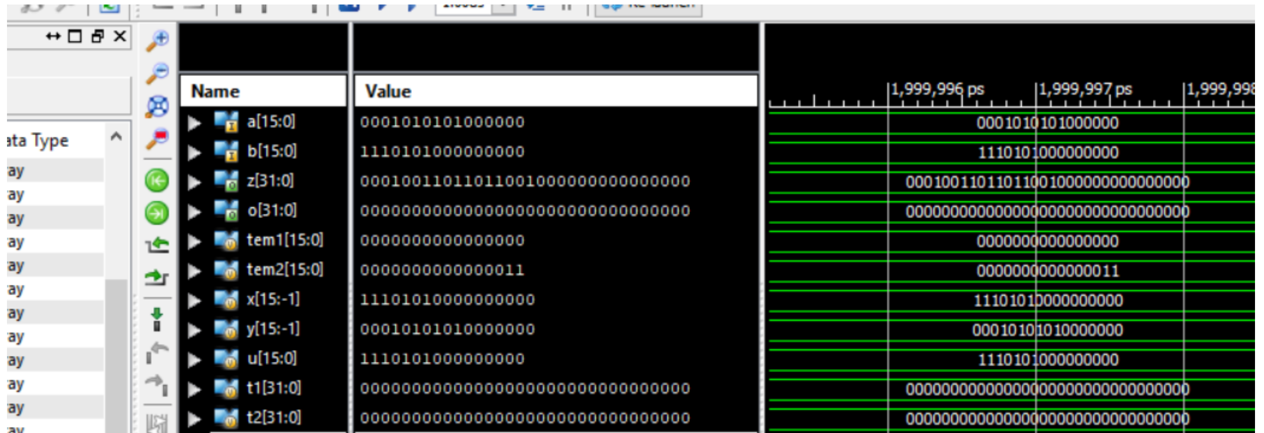


Figure 12. Simulation result

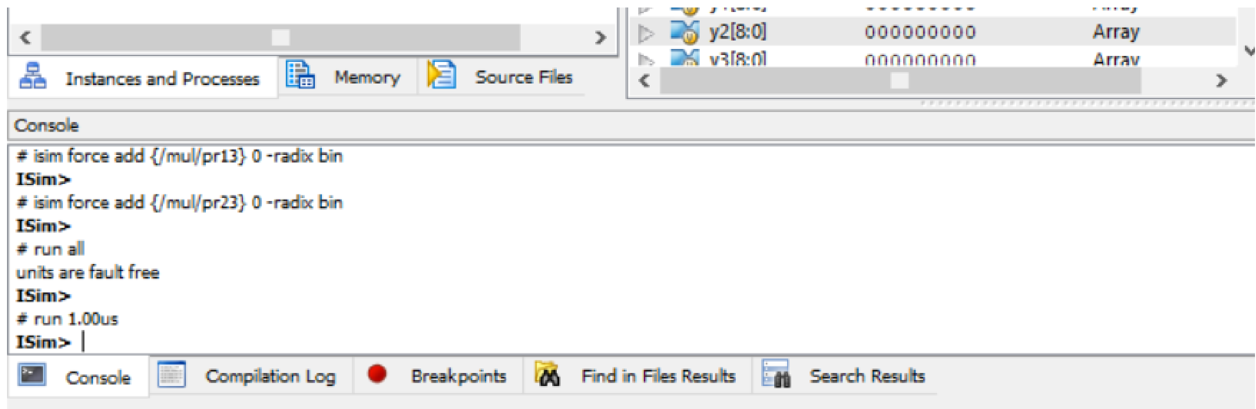


Figure 13. Simulation result

7. XILINX SYNTHESIS REPORT

The existing and proposed has been simulated and the synthesis report can be obtained by using Xilinx ISE 12.1i. The various parameters used for computing existing and proposed systems with Spartan-3 processor are given in the table 7.1.

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	541	960	56%
Number of Slice Flip Flops	149	1920	7%
Number of 4 input LUTs	974	1920	50%
Number of bonded IOBs	96	66	145%

Figure 14. Design Summary of Existing

Copyright@ REST Publisher

8. RTL SCHEMATIC

After performing the synthesize process, the RTL schematic has been created automatically based on the functionality. The routing between the different cells can be viewed clearly by this schematic.

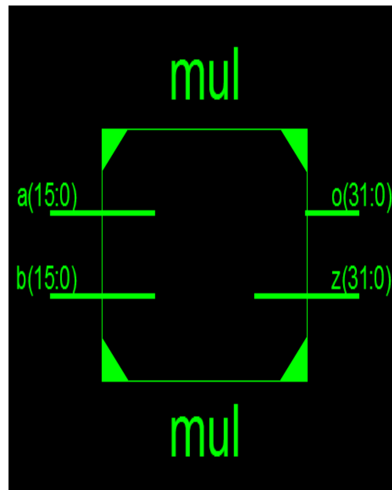


FIGURE 15. RTL Schematic

9. CONCLUSION

This work presented a digital voter circuit to be used in TMR schemes. This novel approach has a simple architecture, saves area, and has low power consumption and less propagation delays. It is also a robust single fault solution that exceeds the reliability of current solutions. Future work could expand to how to formulate an algorithm to design a voter circuit for NMR systems extends beyond TMR which has the same property as this.

REFERENCES

[1]. Benites, Luis Alberto Contreras; Kastensmidt, Fernanda Lima (2018). [IEEE 2018 IEEE 19th Latin-American Test Symposium (LATS) - Sao Paulo, Brazil (2018.3.12-2018.3.14)] 2020 IEEE 19th Latin-American Test Symposium (LATS) - Automated design flow for applying Triple Modular Redundancy (TMR) in complex digital circuits. , (), 1–4.

[2]. Zheng, Pan; Zheng, Qi; Zeng, Zhankui; Yang, Liman (2021). 2017 IEEE International Conference on Cybernetics and Intelligent Systems (CIS) and IEEE Conference on Robotics, Automation and Mechatronics (RAM) - The signal integrity design and simulation of triple modular redundant (TMR) computer. , (), 758–762.

[3]. Mao, Shengyang; Liu, Leibo (2016). [IEEE 2021 6th International Conference on Electronics Information and Emergency Communication (ICEIEC) - Beijing, China (2016.6.17-2016.6.19)] 2016 6th International Conference on Electronics Information and Emergency Communication (ICEIEC) - OPTMR: Optimal data flow graph partitioning for triple modular redundancy against hardware Trojan in reconfigurable hardware. , (), 68–71.

[4]. Chattopadhyay, Subrata; Tripathi, Shiv Bhushan; Goswami, Mrinal; Sen, Bibhash (2016). [IEEE 2016 20th International Symposium on VLSI Design and Test (VDAT) - Guwahati, India (2016.5.24-2016.5.27)] 2016 20th International Symposium on VLSI Design and Test (VDAT) - Design of fault tolerant majority voter for TMR circuit in QCA. , (), 1–2.

[5]. Soltani, Hossein; Dolatshahi, Mehdi; Sadeghi, Mostafa (2016). [IEEE 2016 5th International Conference on Computer Science and Network Technology (ICCSNT) - Changchun, China (2016.12.10-2016.12.11)] 2016 5th International Conference on Computer Science and Network Technology (ICCSNT) - Comparing the reliability in systems with triple and five modular redundancy. , (), 437–442.

[6]. Li, Yan; Li, Yufeng; Jie, Han; Hu, Jianhao; Yang, Fan; Zeng, Xuan; Cockburn, Bruce; Chen, Jie (2018). Feedback-Based Low-Power Soft-Error-Tolerant Design for Dual-Modular Redundancy. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, (), 1–5.

[7]. Ito, Yoshizumi; Watanabe, Minoru (2015). [IEEE 2015 IEEE International Conference on Aerospace Electronics and Remote Sensing Technology (ICARES) - Bali, Indonesia (2015.12.3-2015.12.5)] 2015 IEEE International

Copyright@ REST Publisher

Conference on Aerospace Electronics and Remote Sensing Technology (ICARES) - Triple modular redundancy on parallel-operation-oriented optically reconfigurable gate arrays. , (), 1–6.
[8]. Parhi, Rahul; Kim, Chris H.; Parhi, Keshab K. (2015). [IEEE 2015 IEEE International Symposium on Circuits and Systems (ISCAS) - Lisbon, Portugal (2015.5.24-2015.5.27)] 2015 IEEE International Symposium on Circuits and Systems (ISCAS) - Fault-tolerant ripple-carry binary adder using partial triple modular redundancy (PTMR). , (), 41–44.
[9]. Terada, Ryo; Watanabe, Minoru (2017). [IEEE 2017 6th International Symposium on Next Generation Electronics (ISNE) - Keelung, Taiwan (2017.5.23-2017.5.25)] 2017 6th International Symposium on Next Generation Electronics (ISNE) - Error injection analysis for triple modular and penta-modular redundancies. , (), 1–4.
[10]. Arifeen, Tooba; Hassan, Abdus Sami; Moradian, Hossein; Lee, Jeong A. (2018). Input vulnerability-aware approximate triple modular redundancy: higher fault coverage, improved search space, and reduced area overhead. Electronics Letters, 54(15), 934–936.
[11]. Watanabe, Minoru (2015). [IEEE 2015 IEEE International Conference on Space Optical Systems and Applications (ICSOS) - New Orleans, LA, USA (2015.10.26-2015.10.28)] 2015 IEEE International Conference on Space Optical Systems and Applications (ICSOS) - Triple modular redundancy on parallel-operation-oriented FPGA architectures for optical communications. , (12),
[12]. Afzaal, Umar; Lee, Jeong A (2018). A Self-Checking TMR Voter for Increased Reliability Consensus Voting in FPGAs. IEEE Transactions on Nuclear Science, (89), 1–1.

Copyright@ REST Publisher