A DEPENDABLE COMPUTING APPLICATION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

UĞUR GÜNGÖR

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

APRIL 2005

Approval of the Graduate School of Natural and Applied Sciences

Prof. Dr. Canan ÖZGEN
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr. İsmet ERKMEN
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Prof. Dr. Hasan Cengiz GÜRAN
Supervisor

Examining Committee Members

Doç. Dr. Gözde BOZDAĞI AKAR	(METU,EEE)
Prof. Dr. Hasan Cengiz GÜRAN	(METU,EEE)
Yrd. Doç. Dr. Cüneyt BAZLAMAÇCI	(METU,EEE)
Dr. Şenan Ece SCHMIDT	(METU,EEE)
Gökhan GÖKSÜGÜR (M.Sc.)	(ASELSAN)

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Uğur GÜNGÖR

ABSTRACT

A DEPENDABLE COMPUTING APPLICATION

Güngör, Uğur
M.S., Department of Electric and Electronics Engineering
Supervisor : Prof. Dr. Hasan Cengiz Güran

April 2005, 129 pages

This thesis focuses on fault tolerance which is kind of dependable computing implementation. It deals with the advantages of fault tolerance techniques on Single Event Upsets (SEU) occurred in a Field Programmable Gate Array (FPGA). Two fault tolerant methods are applied to floating point multiplier. Most common SEU mitigation method is Triple Modular Redundancy (TMR). So, two fault tolerance methods, which use TMR, are tested.

There are three printed circuit boards (PCBs) and one user interface software in the setup. By user interface software running on a computer, user can inject fault or faults to the selected part of the system, which uses TMR with voting circuit or TMRVC TMR with voting and correction circuits on floating point multiplier. After inserting fault or faults, user can watch results of the fault injection test by user interface software. One of these printed circuit boards is called as a Test Pattern Generator. It is responsible for communication between the Fault Tolerant Systems and the user interface software running on a computer. Fault Tolerant Systems is second PCB in the setup. It is used to implement fault tolerant methods on fifteen bits floating point multiplier in the FPGA. First one of these methods is TMR with voter circuit (TMRV) and second one is TMR with voter and correction circuits (TMRVC). Last PCB in the setup is Display PCB. This PCB displays fault tolerant test result and floating point multiplication result. All the functions on Test Pattern Generator and Fault Tolerant Systems are implemented through the use of a Field Programmable Gate Array (FPGA), which is programmed using the Very High Speed IC Description Language (VHDL).

Implementation results of the used methods in FPGA are evaluated to observe the performance of applied methods for tolerating SEU.

ÖZ

YÜKSEK GÜVENİLİRLİKLİ BİLGİSAYAR DONANIM UYGULAMASI

Güngör, Uğur
Yüksek Lisans, Elektrik-Elektronik Mühendisliği Bölümü
Tez Yöneticisi : Prof. Dr. Hasan Cengiz Güran

Nisan 2005, 129 sayfa

Bu tez, yüksek güvenilirlikli bilgisayar uygulamalarından birisi olan hata toleransı uygulamasının üzerinde durmaktadır. Bu tezde Alan Programlanabilir Kapı Dizini (FPGA) içinde oluşan Tekli Hata Oluşumlarına (SEU) karşı uygulanan hata tolerans metodlarının avantajları ile ilgilenilmiştir. Bunun için, 2 tane hata tolerans metodu, kayan noktalı çarpıcı üzerine uygulanmıştır. En yaygın SEU azaltma metodu Üçlü Modüler Yedekleme'dir (TMR). Bu nedenle, TMR kullanan iki çeşit hata tolerans metodu test edilmiştir.

Düzenekte 3 tane Baskı Devre Kartı (PCB) ve 1 tane kulanıcı arayüz yazılımı bulunmaktadır. Bilgisayar üzerinde çalışan kullanıcı arayüz yazılımı ile kullanıcı, oylama devreli yada oylama ve düzeltme devreli TMR kullanan systemin seçilen bir bölgesine hata veya hatalar enjekte edebilir. Hata veye hataları enjekte ettikten sonra , kullanıcı hata enjekte etme testinin sonuçlarını kullanıcı arayüz yazılımından izleyebilir. Düzenekteki PCB'lerden birisi Test Örüntüsü Yaratıcı'dır. Bu PCB, Hata Tolere Edebilir Sistemler ve kullanıcı arayüz yazılımı arasındaki konuşmadan sorumludur. Düzenekteki ikinci PCB, Hata Tolere Edebilir Sistemler'dir. Bu PCB, hata tolere edebilir metotları, kayan noktalı çarpıcı üzerine FPGA'de gerçekleştirmekle sorumludur. Bu metodlardan birincisi oylama devreli TMR (TMRV), ikincisi ise oylama ve düzeltme devreli TMR'dır (TMRVC). Düzenekteki son PCB Gösterge PCB'sidir. Bu PCB, test sonucunu ve çarpma isleminin sonucunu gösterir.

Test Örüntüsü Yaratıcı ve Hata Tolere Edebilir Sistemler üzerindeki fonksiyonlar, Çok Yüksek Hızlı Entegre Devre Tanımlama Dili (VHDL) kullanılarak programlanan Alan Programlanabilir Kapı Dizinleri içine gerçekleştirilmiştir.

FPGA içerinde gerçekleştirilen metotların sonuçları değerlendirilerek, uygulanan metotların performansları irdelenmiştir.

To My Family

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervisor Prof. Dr. Hasan Cengiz Güran for his supervision and guidance.

Special thanks to ASELSAN Inc. Microelectronics, Guidance and Electro-Optics division for providing technical support and laboratory environment in which I could develop my design.

Also, I am thankful to all my family and my friends for their continued patience, understanding, and encouragement throughout the preparation of this study.

TABLE OF CONTENTS

CHAPTERS

ABBREVIATIONS

ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
BGA	Ball Grid Array
CLB	Configurable Logic Blocks
CMOS	Complementary Metal Oxide Silicon Technology
DSP	Digital Signal Processor
PROM	Programmable Read Only Memory
EEPROM	Electrically Erasable Programmable Read Only Memory
FP	Floating Point
FPGA	Field Programmable Gate Array
GRM	General Routing Matrix
IC	Integrated Circuit
IOB	Input/Output Block
I2C	Inter Integrated Circuit Control
LED	Light Emitting Diode
LUT	Look-up Table
NMR	N-Modular Redundancy
PCB	Printed Circuit Board
PGA	Pin Grid Array
PROM	Programmable Read Only Memory
QFP	Quad Flat Package
SEU	Single Event Upset
TMR	Triple Modular Redundancy
TTL	Transistor-Transistor Logic
VHDL	Very High Speed Integrated Circuit Description Language
TMRV	TMR with voter circuit
TMRVC	TMR with voter and correction circuits

LIST OF TABLES

TABLE

LIST OF FIGURES

FIGURE

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION TO FAULT TOLERANCE

A Fault tolerant system can perform its specified tasks in the presence of hardware faults and software errors. Fault tolerance tries to prevent negative effects of these faults on the system operation. Fault tolerance is very important in mission critical applications. Most popular applications where fault tolerance is used are listed below. This list shows only most popular applications but there are many other systems which need fault tolerance.

- Space based applications
- Process control systems
- Missile guidance systems
- Medical applications

Fault tolerance can be classified into sub-classes which are listed below **[1]**.

- Fault Detection
- Fault Diagnosis
- Fault Containment
- Fault Masking
- Fault Compensation

- Fault Repair

Fault Detection determines the occurrence of a fault.

Fault Diagnosis determines the reason of a fault. It also specifies which subsystem is faulty.

Fault Containment prevents the propagation of faults from one subsystem to other subsystems.

Fault Masking ensures that only correct values get passed to the system boundary in spite of a failed component.

Fault Compensation provides a response to compensate for output of the faulty subsystem.

Fault Repair is a process, which removes faults in the systems and repair faulty modules.

Redundancy is the addition of resources, time, or information beyond what is needed for normal system operation. There are some forms of redundancy, which are listed below.

- Hardware redundancy
- Software redundancy
- Information redundancy
- Time redundancy

Hardware redundancy is the addition of extra hardware. Software redundancy is the addition of extra software, beyond what is needed to perform a given function. Information redundancy is the addition of extra information beyond that required to implement a given function. Time redundancy is the usage of additional time to perform the functions of a system. Hardware redundancy and software redundancy are most common forms of redundancy **[1]**.

1.2 SHORT DESCRIPTION OF THE THESIS APPLICATION

Field Programmable Gate Arrays (FPGAs) are high performance data processing devices. They provide to construct highly parallel architectures for processing data. They are reprogrammable. This decreases design period. Functional changes can be implemented in a short time. As a result, they are preferred in most of the applications today.

Charged particles can provoke a transient pulse, when it hits the silicon. This pulse can change the state of a memory cell on integrated circuits. This phenomenon is known as a Single Event Upset (SEU). SEU is a very serious problem in mission critical applications. FPGAs are also potentially sensitive to SEU. As a result, fault tolerant algorithms must be implemented into FPGAs in mission critical applications **[2] [3]**.

SEU is very serious problem in FPGAs. There are some fault tolerant methods to decrease the probability of a system failure due to these SEUs. Most popular fault tolerant method for FPGA applications is Triple Modular Redundancy (TMR) with voter circuit. TMR uses three identical circuits, which perform the same task in parallel. Their outputs are compared through a majority voter circuit **[4]**.

Fault injection is a widely used technique for dependability estimation of fault tolerant systems. It is an intentional activation of faults in order to observe the system under fault behavior.

In this thesis, there are two methods applied to floating point multiplier. One of them is TMRV. This method is used for detecting faults and correcting output of a system. It does not correct the faulty module. Second one is TMRVC. It is used for detecting faults, correcting output of a system and correcting the faulty module. Second one is more powerful than first one. But, it uses extra logic for correction circuit.

The aim of this work initially was to design a fault tolerant floating point multiplier circuit with correction property. But the algorithm to be used was not suitable for a parallel multiplication operation with correction. So, our application multiplies input and the previous value of the floating point multiplier output. By this way, we can apply fault tolerant methods with correction circuit to our circuit.

In our application, faults are injected into sub-modules, which are implemented in XC4010 family FPGA, and responses are observed.

Fault injection test composes of 4 main parts.

1.2.1 Software Part:

By the software user interface running on a computer, user can inject fault or faults to the selected part of the system, which uses TMRV or TMRVC. After inserting fault or faults, user can watch the result of the fault injection test by the software user interface.

1.2.2 Test Pattern Generator Part:

There are three PCBs in this thesis setup. One of them is called as Test Pattern Generator. This PCB is responsible for communication between the Fault Tolerant Systems and the software user interface running on a computer. Fault Tolerant Systems is second PCB in the system and responsible for fault tolerant system implementations.

Test Pattern Generator communicates with the software via RS232 port of the computer. Test Pattern Generator transfers fault injection commands, which come from RS232 port, to the Fault Tolerant Systems by a specified protocol. Then, it takes responses from the Fault Tolerant Systems and sends them to the software part of the setup.

1.2.3 Fault Tolerant Systems Part:

Second PCB in the setup is called as Fault Tolerant Systems. This PCB is used to implement 2 fault tolerant methods on floating point multiplier in the FPGA. First one of these methods is TMRV and second one is TMRVC. As said, there are two systems but they don’t operate in parallel. By changing FPGA configuration data, which is stored in EEPROM, of Fault Tolerant Systems, operating fault tolerant method on the system is selected. Fault injection commands, which come from Test Pattern Generator, are applied to selected fault tolerant method on the system, and test results send back to Test Pattern Generator. These test results are also sending to Display PCB, which is the last PCB on the thesis setup.

1.2.4 Display PCB:

Third PCB in the setup is called as Display PCB. This PCB displays fault tolerant test result and floating point multiplication result.

Finally, advantages and disadvantages of the applied fault tolerant methods on floating point multiplier are explained. These explanations are proven by synthesis and implementation reports of the thesis application.

1.3 ORGANIZATION OF THE THESIS

This thesis is composed of 7 chapters.

Chapter 2 provides general information about fault tolerance. Fundamental definitions on fault tolerance are given. Design goals and methodologies on fault tolerance are explained in this chapter.

Chapter 3 focuses on the FPGA fault tolerance. Single Event Upset (SEU) is specified. FPGA fault tolerance techniques for tolerating SEU are explained.

Chapter 4 explains high speed PCB design rules, which are considered during the design of PCBs in this setup. PCB components used in these PCBs are also explained in this chapter.

Chapter 5 is focused on FPGA design in Test Pattern Generator and Fault Tolerant Systems.

Chapter 6 presents advantages and disadvantages of the applied fault tolerant methods. These explanations are proven by implementation reports of Fault Tolerant Systems FPGA.

Chapter 7 gives the conclusion.

CHAPTER 2

FAULT TOLERANCE

A **fault tolerant system** is one that can continue to correctly perform its specified tasks in the presence of hardware failures and software errors. Fault tolerance is an attribute that is designed into a system to achieve some design goals. These design goals are, reliability, availability, safety, performability, maintainability, testability and dependability.

Below, we explain each term separately **[1] [13]**,

Reliability, ***R(t):*** The conditional probability that a system performs correctly throughout an interval of time [t0, t], given that the system was performing correctly at time t0.

Availability, ***A(t):*** The probability that a system is operating correctly and is available to perform its functions at the instant of time t.

Safety, ***S(t):*** The probability that a system will either perform its functions correctly or will discontinue its functions in a well defined, safe manner. Discontinue its functions in a manner that does not disturb the operation of other systems or compromise the safety of any people associated with the system.

Performability, *P(L, t):* The probability that a system is performing at or above some level of performance L at the instant of time t.

Maintainability, *M(t):* The probability that an inoperable system will be restored to an operational state within the time t.

Testability: The ability to test for certain attributes within a system.

Dependability: The quantity of service that a particular system provides. Reliability, availability, safety, maintainability, performability and testability are measures used to quantify the dependability of a system.

2.1 FUNDAMENTAL DEFINITIONS

Three fundamental terms in fault tolerant design are fault, error and failures. There is a cause and affect relationship between them.

A fault is a physical defect, imperfection, or flaw that occurs within some hardware or software component. An error is an occurrence of an incorrect value in some unit of information within a system. A failure is a deviation in the expected performance of a system. The relationships between fault, error and failure are shown in Figure 2.1 **[13]**.

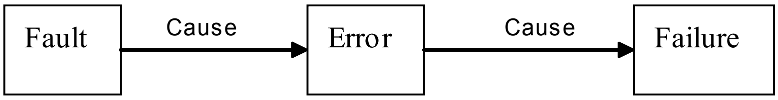


Figure 2.1 Relationships between fault, error and failure

2.1.1 CHARACTERISTICS OF A FAULT

Faults can be characterized by five attributes, which are cause, nature, duration, extend and value. Figure 2.2 illustrates each of these basic characteristics of faults.

Possible **fault causes** can be associated with problems in four areas.

Specifications mistakes:
These include incorrect algorithms, architectures, or hardware and software design specifications.

Implementation mistakes:
The implementation can introduce faults due to poor design, poor component selection, poor construction, or software coding mistakes.

Components defects:
These include random device defects, manufacturing imperfections, and component wear-out.

External disturbance:
These include operator mistakes, radiation, electromagnetic interference, and environment extremes.

The **fault nature** specifies the type of fault, which is hardware fault or software fault. If hardware faults, it specifies analog fault or digital fault.

The **fault duration** specifies the length of time that a fault is active. There are three types of fault durations.

Permanent fault:
It is a fault, which remains in existence indefinitely if no corrective action is taken.

Transient fault:

It is a fault, which can appear and disappear within a very short period of time.

Intermittent fault:

It is a fault, which appears, disappears, and reappears repeatedly.

The **fault extend** specifies whether the fault is localized to a given hardware or software module or whether it globally affects the hardware, the software, or both.

The **fault value** can be either determinate or indeterminate. A determinate fault is one whose status remain unchanged throughout time unless external action upon. An indeterminate fault is one whose status at some time t may be different from its status at another time **[1]**.

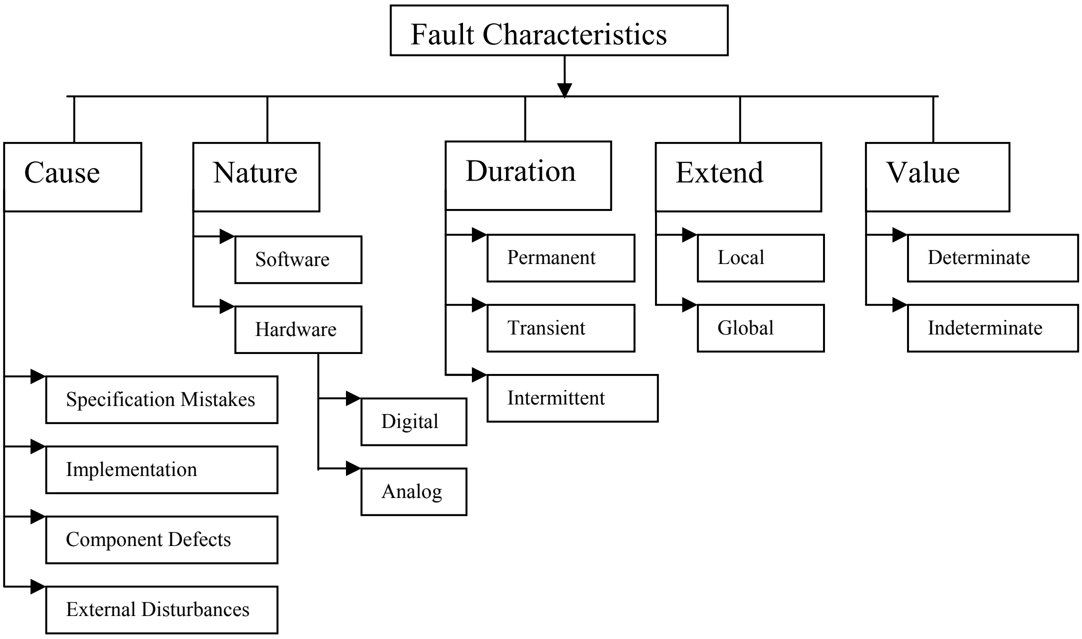


Figure 2.2 Fault Characteristics

2.1.2 PRIMARY SYSTEM IMPROVEMENT TECHNIQUES

There are three primary techniques for attempting to improve or maintain a system's normal performance. They are fault avoidance, fault masking and fault tolerance.

Fault avoidance is any technique that attempts to prevent the occurrence of faults. It can include design reviews, component screening, testing and other quality control methods. Figure 2.3 illustrates the barriers that are constructed by each of the available techniques.

Fault masking is the process of preventing faults from introducing errors.

Fault tolerance is the ability to correct performance of functions in the presence of faults. (In fact, failure is directly related to error tolerance but we can use fault tolerance instead of error tolerance.)

Fault tolerance can be achieved by many techniques. Fault masking is one approach to tolerating faults. Another approach is to detect and locate the fault that has occurred and reconfigure the system to remove the faulty component.

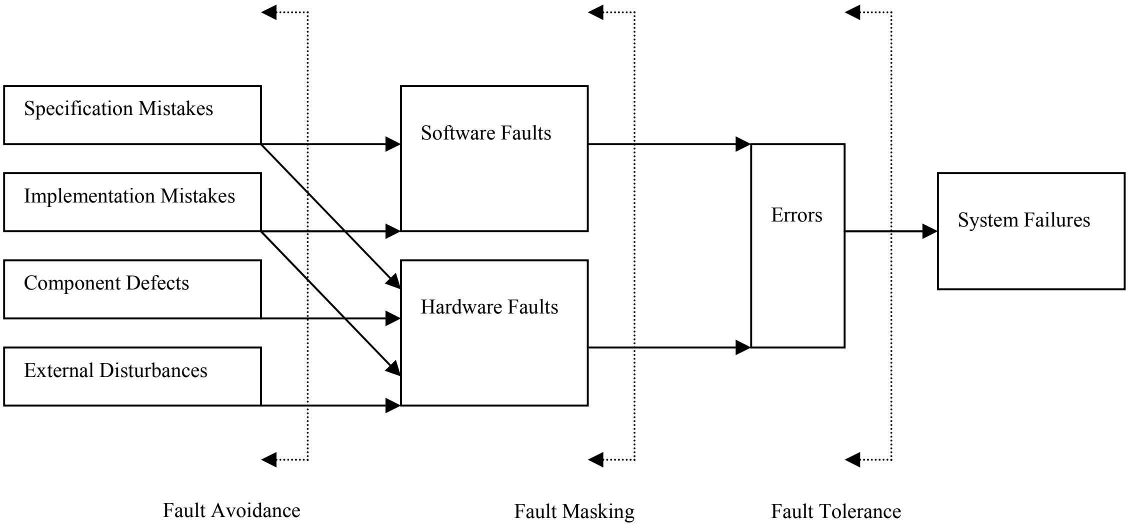


Figure 2.3 System performance improvement techniques barriers

As shown in Figure 2.3, by fault avoidance occurrence of faults, which is caused by specification mistakes, implementation mistakes, component defects and external disturbances is tried to be prevented. Fault masking tries to prevent error occurrence due to software and hardware faults. Fault tolerance tries to prevent system failure due to errors.

2.2 DESIGN TECHNIQUES TO ACHIEVE FAULT TOLERANCE

Fault masking achieves fault tolerance by hiding faults that occur. Systems that use fault masking do not require fault detection before tolerating them. Fault masking makes faults local. It prevents the effects of faults from spreading throughout the system. Fault masking is a method of achieving fault containment.

Systems that do not use fault masking require **fault detection**, **fault location** and **fault recovery** to achieve fault tolerance. Fault detection is essential to fault location and fault recovery processes. Fault location is required to identify exactly which component is faulty. Fault recovery involves some form of reconfiguration that is usually accomplished by disabling, either physical or logically, a faulty component and enabling, again either physically or logically, a replacement component.

Before dealing with techniques for achieving fault detection, fault location, fault recovery and fault masking, we are interested in redundancy. Some forms of redundancy are required for all of these techniques.

2.3 REDUNDANCY

Redundancy is an addition of resources, time, or information beyond what is needed for normal system operation. There are some forms of redundancy, which are listed below.

Hardware redundancy
Software redundancy

Information redundancy
Time redundancy

Hardware redundancy is an addition of extra hardware beyond what is needed to perform a given function. Software redundancy is the addition of extra software, beyond what is needed to perform a given function. Information redundancy is an addition of extra information beyond that required to implement a given function. Time redundancy is the usage of additional time to perform the functions of a system **[1]**.

2.3.1 HARDWARE REDUNDANCY

There are three forms of hardware redundancy that are **passive**, **active**, and **hybrid [1]**. Passive techniques use fault masking. Passive approaches are designed to achieve fault tolerance without requiring any action on any part of the system.

Active techniques (dynamic methodologies) achieve fault tolerance by detecting the existence of faults and performing some actions to remove the faulty hardware from the system. In other words, active hardware redundancy uses fault detection, fault location, and fault recovery to achieve fault tolerance.

Hybrid techniques combine the attractive features of both active and passive techniques.

2.3.1.1 PASSIVE HARDWARE REDUNDANCY

Most of the passive approaches are developed around the concept of majority voting. In passive approaches, fault detection and system reconfiguration is not required.

Triple Modular Redundancy (TMR)

TMR uses triplication of the hardware and it performs majority vote to determine the system output. Block diagram of TMR is shown in Figure 2.4.

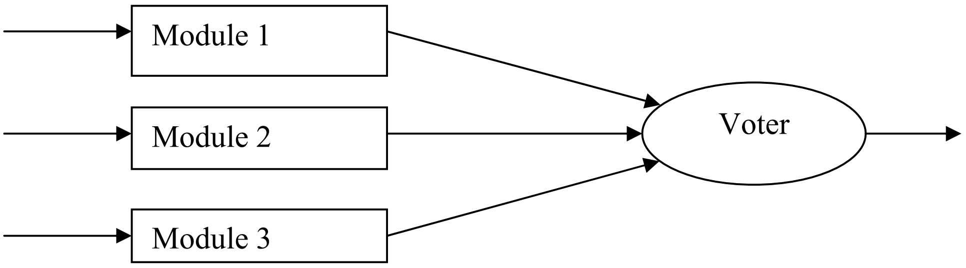


Figure 2.4 Block diagram of TMR

If one of these modules becomes faulty, other two modules mask the fault by a majority voting mechanism.

If the voter fails, the complete system fails. This must be prevented. This can be achieved by increasing number of voters as shown in Figure 2.5. If there are 3 voters, receiver side must have 3 input channels to get information from fault tolerant module. Receiver side needs voter circuit to get information.

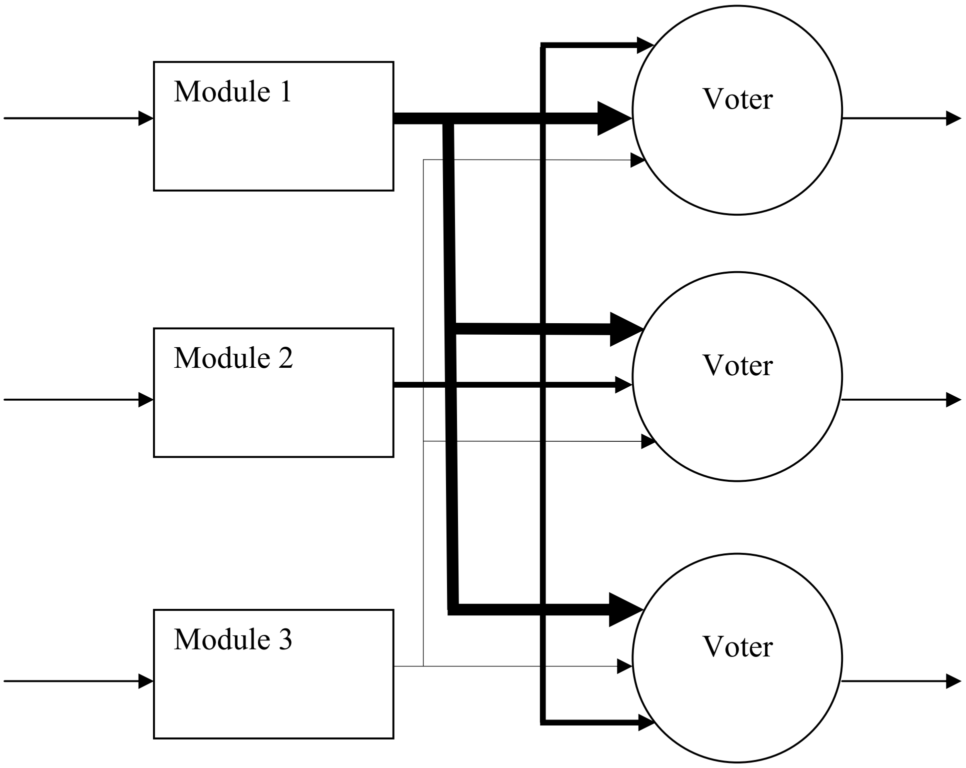


Figure 2.5 Block diagram of TMR with three voter circuit

N-Modular Redundancy (NMR)

NMR is the generalization of the TMR approach. It uses same principles but number of modules is not three. N is chosen as an odd number to use majority vote arrangement.

By using NMR, more module faults can be tolerated. Block diagram of NMR is shown in Figure 2.6.

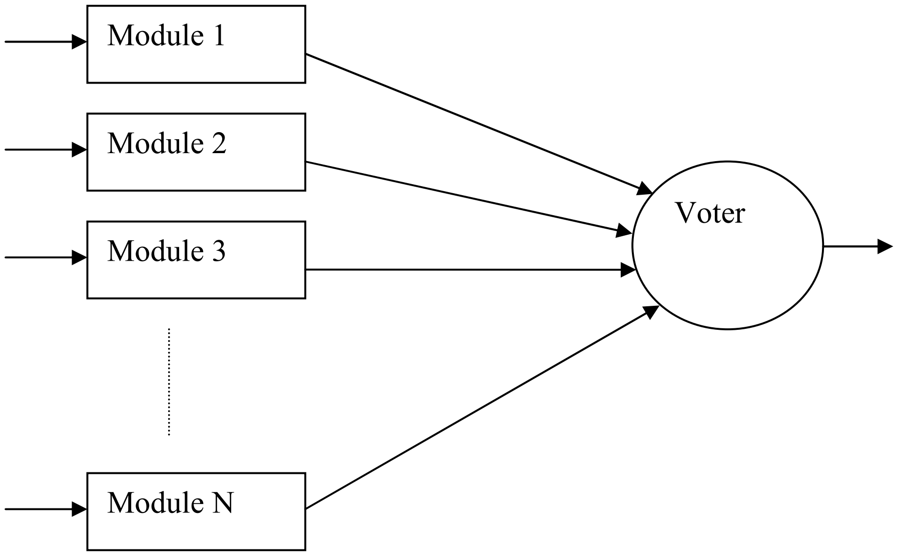


Figure 2.6 Block Diagram of NMR

2.3.1.2 ACTIVE HARDWARE REDUNDANCY

Active hardware redundancy techniques achieve fault tolerance by fault detection, fault location and fault recovery. This approach does not try to prevent faults from producing errors. So that, active hardware redundancy is common in applications that can be tolerated temporary.

Duplication with Comparison

In this scheme, two modules operate in parallel and their results are compared. If they are different, an error message is generated. This is fundamental fault detection technique in an active redundancy approach. Block diagram of duplication with comparison is shown in Figure 2.7.

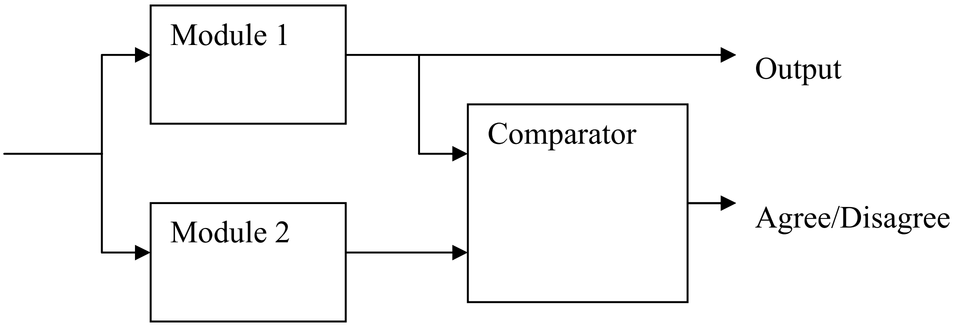


Figure 2.7 Block diagram of duplication with comparison

Standby Sparing (Standby replacement)

In standby sparing, one of n modules is used to provide the system's output, and the remaining n-1 modules serve as spares. Error detection techniques identify faulty modules so that a fault-free module is always selected to provide the system's output. Block diagram of standby sparing is shown in Figure 2.8.

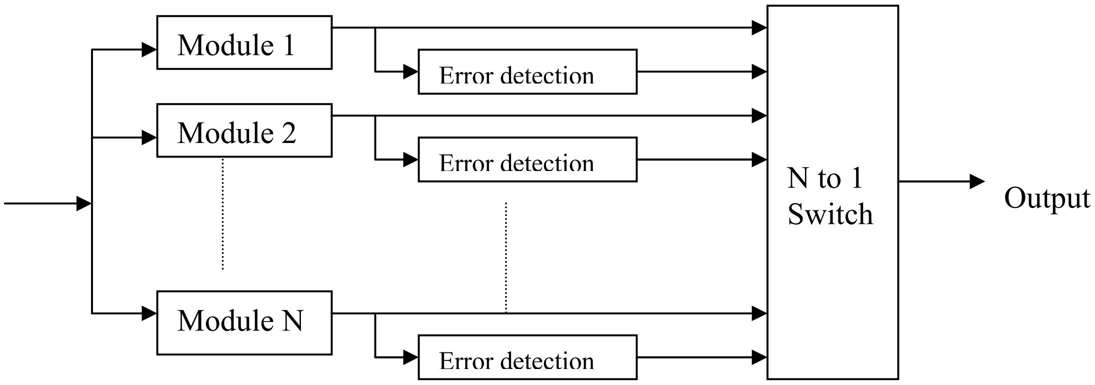


Figure 2.8 Block diagram of standby sparing

Pair–and-a-Spare Technique

It combines the features of both standby and duplication with comparison techniques. Block diagram of pair-and-a-spare technique is shown in Figure 2.9.

In Pair-and-a-Spare Technique, two of n modules are used to provide inputs to the compare circuit, and the remaining n-2 modules serve as spares. If compare circuit gives disagree output, these two modules are identified as faulty modules and another two modules are selected to provide inputs to the compare circuit.

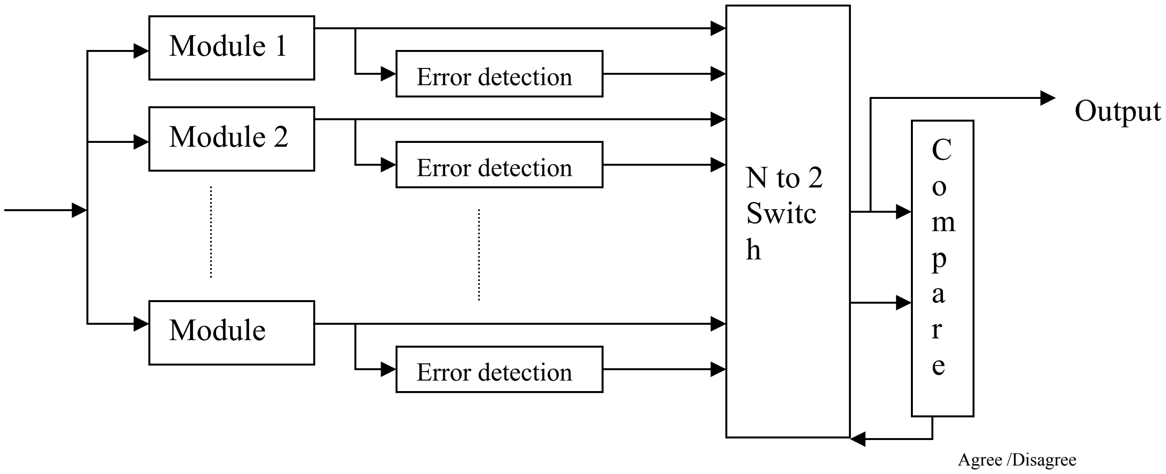


Figure 2.9 Block diagram of pair-and-a-spare technique

2.3.1.3 HYBRID HARDWARE REDUNDANCY

Hybrid hardware redundancy combines the attractive features of both the active and the passive approaches. By fault masking, it tries to prevent erroneous results. Fault detection, fault location, and fault recovery are used to reconfigure the system in the event of a fault. Hybrid redundancy is very expensive in terms of hardware required to implement a system.

2.3.2 SOFTWARE REDUNDANCY

Software redundancy is an addition of extra software, beyond what is needed to perform a given function.

2.3.2.1 CONSISTENCY CHECKS

It uses priori knowledge about the characteristics of information to verify correctness of this information.
Example: In processor system, each sensor's output is in some range. These outputs can be checked for correctness by the software.

2.3.2.2 CAPABILITY CHECKS

It is performed to verify that a system processes the capability expected.
Example: Memory test, ALU test, processor communication test.

2.3.2.3 N-VERSION PROGRAMMING

Up to now, we deal with software to detect faults occur in hardware. Software faults are the result of incorrect software designs or coding mistakes. So, simple duplication and comparison technique will not detect software faults. Design mistake will appear in both modules.

N-version programming is developed for software fault tolerance. Same functional software module is programmed by N different programmers. Hopefully all of them don’t do the same mistake. Then outputs are compared. And fault can be detected easily.

CHAPTER 3

FPGA FAULT TOLERANCE

3.1 FPGA

Field programmable gate array (FPGA) is a general purpose integrated circuit. Application specific integrated circuit (ASIC) performs similar functions but it can not be reprogrammed. FPGA can be reprogrammed after it has been deployed into a system. It is programmed by FPGA system designer.

It is programmed by downloading configuration data (bit stream) into static on-chip random-access memory. This configuration data is the product of compilers. These compilers translate the high level abstractions produced by FPGA system designer into something equivalent but low level and executable code. There are many compilation tools in the industry. Most popular of them are Precision, Leonardo Spectrum and XST **[14]**.

FPGAs are high performance signal processing devices. They provide to construct highly parallel architectures for processing signal. FPGA performance is derived from this ability. Microprocessor or DSP processor performance is tied to the clock rate at which the processor can run, but, FPGA performance is tied to the amount of parallelism to implement algorithms making up a signal processing system. Now, FPGAs can operate up to clock frequencies of 500 mega hertz. It seems to be slow, but FPGAs operate with parallelism. FPGA and DSP represent two very different approaches to signal processing. Each one is good at different things. There are many high sampling rate applications that FPGA can do easily, while DSP can not. Equally, there are many complex software problems that FPGA cannot address. As a result, the ideal system is often splits the work between FPGAs and DSPs **[15]**.

FPGAs are implemented with a regular, flexible programmable architecture of configurable logic blocks (CLBs), interconnected by versatile routing resources (routing channels), and surrounded by programmable input/output blocks (IOBs), as seen in Figure 3.1 **[5]**. This implementation is a basic structure; some FPGA families have extra components such as, dedicated multipliers, dual port memories, digital clock managers, etc…

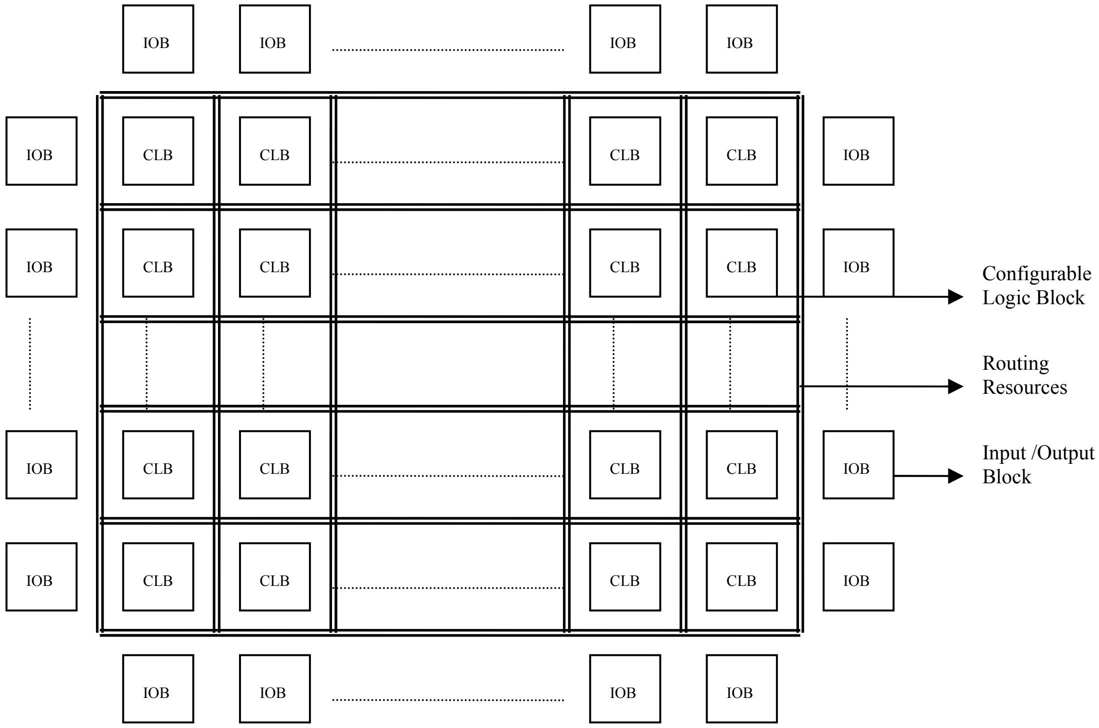


Figure 3.1 FPGA structure

Most of the logic in FPGA is implemented by configurable logic blocks. Internal structure of CLBs changes with FPGA family and FPGA manufacturer. Basic diagram of CLB for XC4000 family Xilinx FPGA that is used in this thesis is shown in Figure 3.2. There are two 4-input function generators (Function generator is called as look-up table (LUT) in some documents.) which are labeled as F and G in Figure 3.2 **[6]**. Third function generator (H) is also provided. H function generator has three inputs as shown in Figure 3.2.

Each CLB contains two storage elements (d type flip-flops (ff)) that can be used to store function generator outputs and direct inputs coming from outside the CLB as shown in Figure 3.2.

Thirteen CLB inputs and four CLB outputs provide access to the function generators and storage elements. These inputs and outputs connect to the programmable interconnect resources outside the CLB.

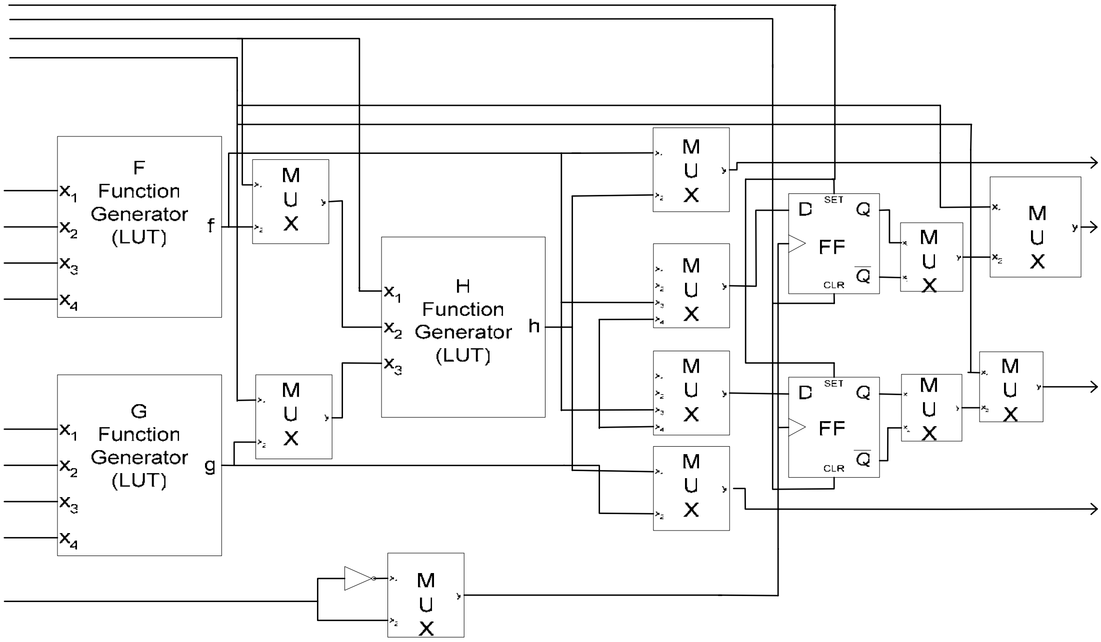


Figure 3.2 Configurable Logic Block Structure

There are 13 FPGA producers today. They are listed below.

Actel Corporation
Altera Corporation
AMI Semiconductor
Amphion Semiconductor, Inc.
Aptix Corporation
Atmel Corporation
Kawasaki LSI U.S.A., Inc.
Nallatech, Inc.
Pentek, Inc.
SiQUEST, Inc.
Tekmos, Inc.
Transtech Parallel Systems
Xilinx, Inc.

3.2 SINGLE EVENT UPSET (SEU)

Fault tolerance on digital circuits has been a meaningful matter since upsets were first experienced in space applications. The digital circuits located in the space environment are affected by the charged particles, which are generated by the solar flares. Charged particles can provoke a transient pulse, when it hits the silicon. This pulse can change the state of a memory cell. This phenomenon is known as a Single Event Upset (SEU). A charged particle can also hit the combinational logic. This generates current pulse. This pulse may be propagated by the combinational logic and latched by memory cells. This event also causes SEU **[2] [3]**.

Integrated circuits are becoming more sensitive to radiation effects. High density devices require smaller feature size, this means less capacitance and hence information

is stored with less charge. Lower voltage or lower power devices mean that less charge or current is required to store information.
Due to these requirements, integrated circuits become much more vulnerable.

FPGAs are composed of an array of configurable logic blocks (CLBs) surrounded by programmable input/output blocks (IOBs). All of them are interconnected by routing resources. The CLBs provide functional elements for constructing logic. The IOBs provide the interface between the package pins and the CLBs. The CLBs are interconnected through a general routing matrix (GRM) that comprises an array of routing switches located at the intersections of horizontal and vertical routing channels.

FPGAs are programmed using a bit stream, which contains all the information to configure the programmable storage elements in the matrix located in the Look-up Tables (LUT) and flip-flops (ff), CLBs configuration cells and interconnections. All these configuration bits are potentially sensitive to SEU. Figure 3.3 shows programmable storage elements.

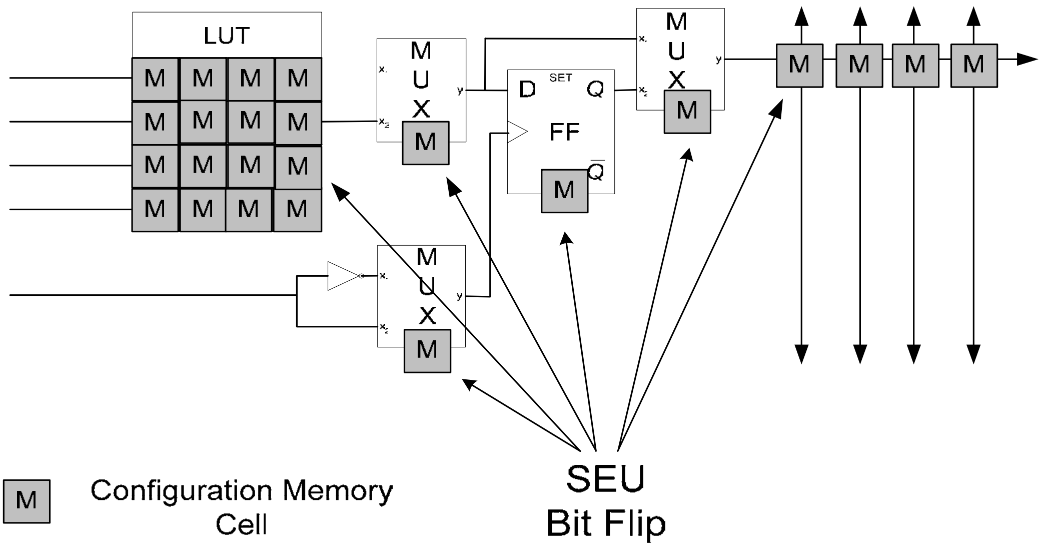


Figure 3.3 Programmable Storage Elements

3.3 FPGA FAULT TOLERANCE TECHNIQUES FOR TOLERATING SEU

There are some FPGA fault tolerance techniques for tolerating SEU. These techniques are explained below.

Most common SEU mitigation method is Triple Modular Redundancy (TMR) with voting circuit. TMR uses three identical circuits, which perform the same task in parallel. Their outputs are compared through a majority voter circuit. TMR can be applied at module level or at device level **[4] [7]**.

Although TMR increases design area and power consumption, full TMR is required in mission critical FPGA designs. So that, device level TMR is preferred for these applications. Device level TMR is shown in Figure 3.4. Device level triple device redundancy has the highest reliability for detecting SEUs. However, this is also the most costly solution.

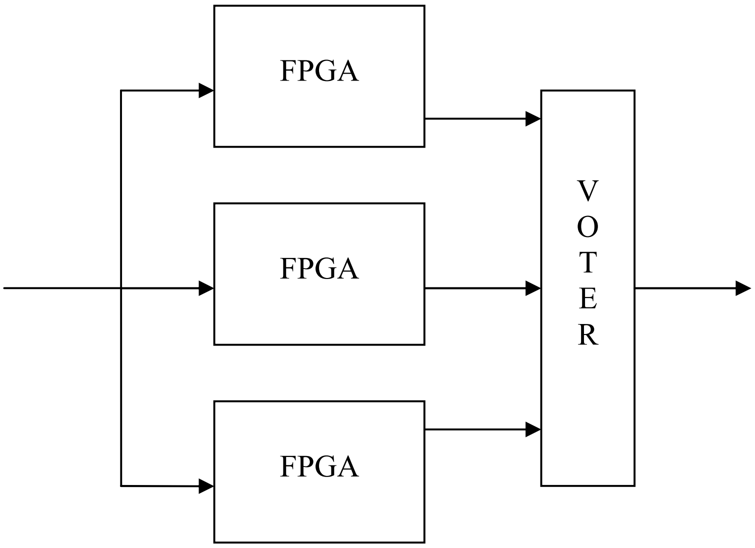


Figure 3.4 Device Level TMR

FPGA design is composed of sub-modules. In most of the applications, some of these modules are mission critical and some of them are not. To decrease design area and power consumption, mission critical modules are implemented with fault tolerance. Other modules are implemented without fault tolerance. For these applications module level TMR can be used. Module level TMR is shown in Figure 3.5.

Module level TMR can not provide a simple recovery mechanism after an error has been detected in one of the modules. Error can be detected when it cause a failure.

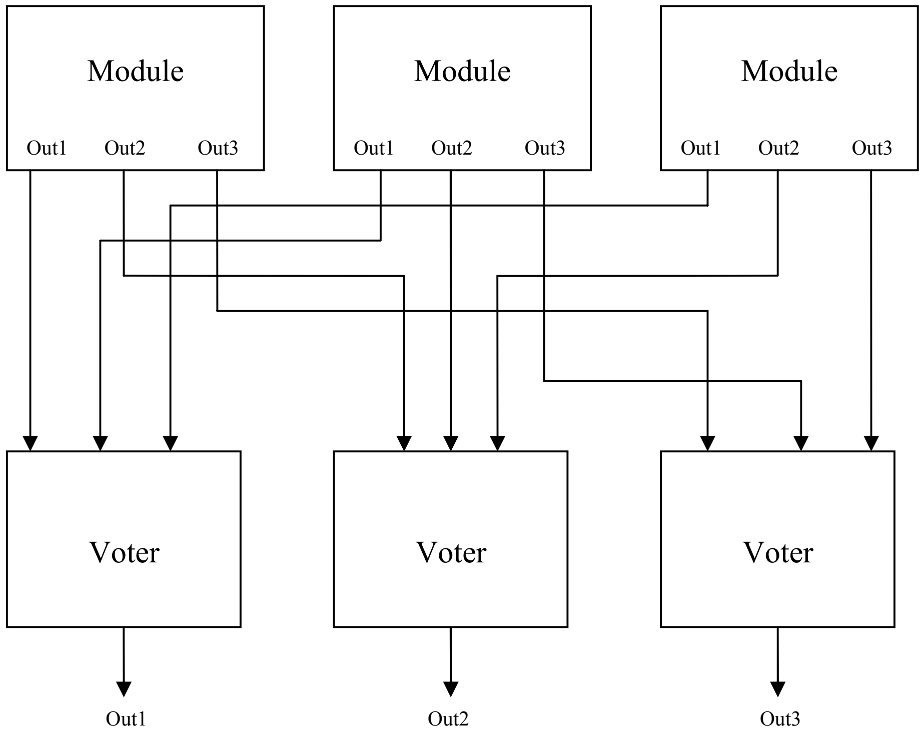


Figure 3.5 Module Level TMR

Module level TMR with feedback (correction) algorithm satisfies all advantages of module level TMR. It also satisfies simple recovery mechanism. When error occurs at one of the modules, state of the faulty module can be corrected by correcting state of the related module. This is achieved by restoring state of all modules with respect to correct result after each operation. Module level TMR with feedback is shown in Figure 3.6.

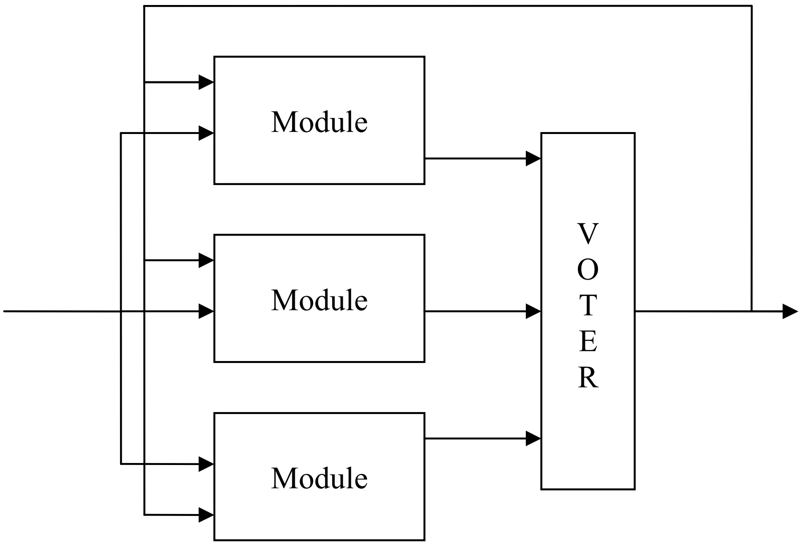


Figure 3.6 Module Level TMR with feedback

Methods mentioned above use three equivalent components to implement function. We need at least three circuits to apply voting algorithm. System designer can implement more than three circuits. This is called as N-Modular Redundancy (NMR), where, n is the number of equivalent components. NMR increases design area and power consumption as the number of equivalent components increases. As a result, TMR is most popular FPGA fault tolerant method.

CHAPTER 4

IMPLEMENTED PCBs IN THE SETUP

4.1 INTRODUCTION

There are 3 PCBs in this thesis setup. One of them is called as Test Pattern Generator. This PCB is responsible for communication between Fault Tolerant Systems and Software User Interface running on a computer. Fault Tolerant Systems is second PCB in the system and responsible for fault tolerant system implementations. Last PCB in the setup is Display PCB. This PCB displays fault tolerance test results and floating point multiplication result. These three PCBs and Software User Interface are shown in Appendix-A

4.2 CRITICAL PCB COMPONENTS

4.2.1 Test Pattern Generator PCB Critical Components

4.2.1.1 Electrically Erasable Programmable Read Only Memory (EEPROM)

Configuration data (bit stream) of the Test Pattern Generator FPGA is stored in AT17C512 Electrically Erasable Programmable Read Only Memory (EEPROM) **[8]**, which is produced by ATMEL Inc. EEPROM loads the FPGA on power up. XC4010E FPGA, used as Test Pattern Generator FPGA in this system, needs a PROM which has at least 178144 bits to store configuration data. AT17C512 has 524288 bits. So that it is chosen as an EEPROM in this PCB.

Test Pattern Generator FPGA is programmed from EEPROM in master serial mode in this design. This mode is selected by MO, M1 and M2 bits of the FPGA **[6]**. In master serial mode, CCLK output of the FPGA drives the CLK input of the EEPROM. AT17C512 internal address counter is incremented at each rising edge of CCLK output of the FPGA. On each address, related data bit is put on DATA_OUT pin of the EEPROM. This pin drives DIN pin of the FPGA. Then, FPGA accept this configuration bit on the subsequent rising CCLK edge. After loading all of the configuration bits, DONE pin of the FPGA goes to logical high, which indicates that configuration of the FPGA is successful. EEPROM program timing diagram is shown in Figure 4.1.

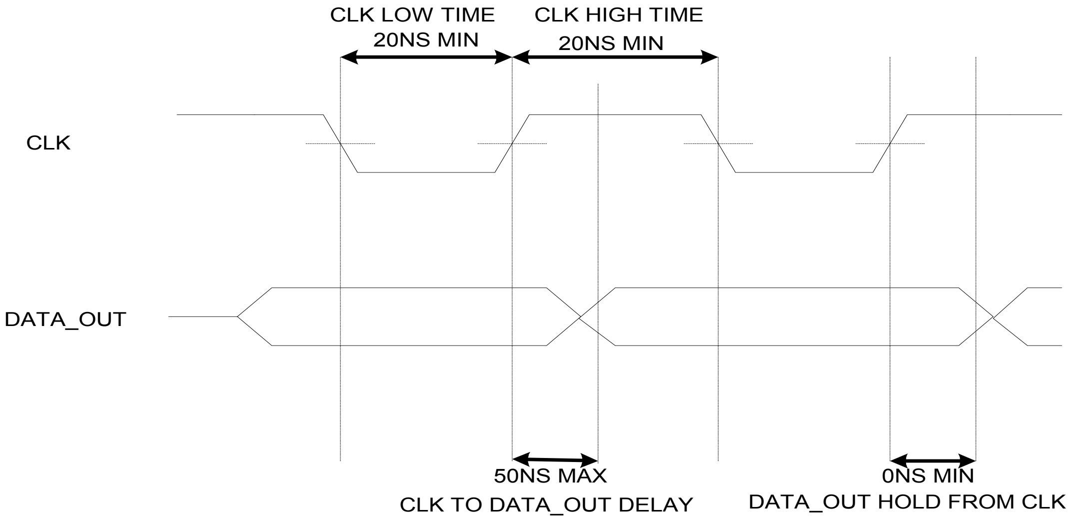


Figure 4.1 EEPROM program timing diagram

4.2.1.2 Field Programmable Gate Array (FPGA)

On Test Pattern Generator PCB, all control and processing options are implemented in the XC4010E FPGA **[6]**, which is produced by Xilinx Inc, Main responsibilities of the Test Pattern Generator FPGA are given below.

It takes user fault injection commands via RS232 transceiver, which is on Test Pattern Generator PCB. If these commands are in the specified protocol, it transfers fault injection signals into the Fault Tolerant Systems.

After transferring fault injection commands, it takes responses from the Fault Tolerant Systems. Then, it sends these responses to the RS232 transceiver in a specified protocol. By this way fault injection test results are displayed on the software user interface running on a computer.

Test Pattern Generator FPGA is also responsible for the generation of a Fault Tolerant Systems FPGA clock.

Implementation of these functions on the Test Pattern Generator FPGA is given in Chapter 5.

4.2.1.3 Oscillator

M55310/26, 39MHz Oscillator **[9]**, produced by Q-Tech Inc, generates clock for Test Pattern Generator FPGA. Clock output of the Oscillator is in HCMOS (high speed CMOS) logic levels. It has a short rise time and fall time. It satisfies input specifications of the FPGA. As a result it is selected as a clock generator in this PCB.

4.2.1.4 RS232 Transceiver

Communication between Software User Interface and Test Pattern Generator FPGA is satisfied with DS232A RS232 Transceiver (Transmitter/ Receiver) **[10]**, which is produced by Dallas Semiconductor Inc. There are two transmitters and two receivers on this chip. In this design, one transmitter and one receiver are used. Other channels are not connected.

Receiver inputs accept RS232 level signals, which come from serial port of the computer. RS232 levels are ±25 volts. These signals are converted to CMOS level signals and send to the Test Pattern Generator FPGA.

Transmitter inputs accept TTL/CMOS level signals, which come from Test Pattern Generator FPGA. These signals are converted to RS232 level signals and send to the serial port of the computer.

4.2.2 Fault Tolerant Systems PCB Critical Components

4.2.2.1 Electrically Erasable Programmable Read Only Memory (EEPROM)

Configuration data of the Fault Tolerant Systems FPGA is stored in AT17C512 Electrically Erasable Programmable Read Only Memory (EEPROM) **[8]**, which is produced by ATMEL Inc. EEPROM loads the FPGA on power up. XC4010E FPGA, which is used as a Fault Tolerant Systems FPGA in this system, needs a PROM which has at least 178144 bits to store configuration data. AT17C512 has 524288 bits. So that it is chosen as an EEPROM in this PCB.

Fault Tolerant Systems FPGA is programmed from EEPROM in master serial mode in this design. This mode is selected by MO, M1 and M2 bits of the FPGA **[6]**. In master serial mode, the CCLK output of the FPGA drives the CLK input of the EEPROM. AT17C512 internal address counter is incremented at each rising edge of the CCLK output of the FPGA. On each address, related data bit is put on the DATA_OUT pin of the EEPROM. This pin drives the DIN pin of the FPGA. Then, FPGA accept this configuration bit on the subsequent rising CCLK edge. After loading all of the configuration bits, DONE pin of the FPGA goes to logical high, which indicates that configuration of the FPGA is successful.

Fault Tolerant Systems FPGA has two configurations. One of them is TMRV and second one is TMRVC. Configuration data (bit stream) of the Fault Tolerant Systems

FPGA changes with selected configuration. So that, EEPROM configuration is changed for selected fault tolerant method test.

4.2.2.2 Field Programmable Gate Array (FPGA)

On Fault Tolerant Systems PCB, all control and processing options are implemented in the XC4010E FPGA **[6]**, which is produced by Xilinx Inc,

Main responsibilities of Fault Tolerant Systems FPGA are given below.

Fault tolerant methods on floating point multiplier are implemented in this FPGA. First one of these methods is TMRV and second one is TMRVC. Fault injection commands, which come from Test Pattern Generator, are applied to selected fault tolerant method on the system, and test results are sent back to Test Pattern Generator. These test results are also sent to Display PCB, which is the last PCB on the thesis setup.

Implementation of these functions on the Test Pattern Generator FPGA is given in Chapter 5.

4.2.3 Display PCB Critical Components

4.2.3.1 LED

90-2841-2832 Dual LED, which is produced by Elma Inc, is used to display fault tolerant test result and 15 bit floating point multiplication result. This LED is dual. One of them is red and the other is green.

4.3 PCB DESIGN CONSIDERATIONS

Designing a PCB, which contains high speed devices, requires special attention to preserve signal quality. When we say high speed, we focus on a clock frequency; we focus on rise time and fall time (slew rates). High speed (fast slew rates) may contribute to crosstalk, and ground bounce. Noise reduction is also very critical in high speed PCB design. To achieve noise free PCBs, we must have noise free power and ground, in other words power system design requires special attention.

Power systems in digital design serve two essential purposes.

- Provide stable voltage references for exchanging digital signals
- Distribute power to all logic devices

What causes noise voltage between the grounds?

Most common cause involves the return current. Whenever one gate send a signal to another gate the outgoing signal current returns to initial gate along the power distribution wiring. The returning signal current acting across inductance of the ground wiring causes noise voltages. Such noise voltages are called common path noise voltages. Common path noise voltage is the product of returning signal current and the ground impedance. To ensure low common path noise, we must have low impedance ground connections between gates. This principle becomes one of the main power system design rules.

There are two types of ground distribution systems, which are listed below.

- Ground Buses
- Ground Planes

Ground Planes present remarkably low inductance to returning signal currents, but Ground Buses design is cheaper than Ground Planes design. So that Ground Planes are preferred in critical designs and Ground Buses are preferred in simple designs.

We use Ground Plane in Test Pattern Generator PCB and Fault Tolerant Systems PCB. Display PCB has a simple design. So that we preferred to use Ground Bus.

Only low ground impedance connection does not solve the common path noise problem. Common path impedance in power wiring can still cause trouble. In the HI state, a gates output voltage depends on the voltage at its power terminal. Any changes in the power voltage caused by returning signal currents flowing in the power wiring directly affect the output voltage. The impedance between power pins on any two gates should be just as low as the impedance between the ground pins.

There are two types of power distribution systems, which are listed below.

- Power Buses
- Power Planes

Power Planes present remarkably low inductance to returning signal currents, but Power Buses design is cheaper than Power Planes design. So that Power Planes are preferred in critical designs and Power Buses are preferred in simple designs.

We use Power Plane in Test Pattern Generator PCB and Fault Tolerant Systems PCB. Display PCB has a simple design. So that we preferred to use Power Bus.

In a system, returning currents flows through the power supply. To maintain stable transmitted signal levels, the impedance of the power supply must be very low as well as the impedance of both the ground and power connections. The only path between the power and ground is the power supply. As a result, there must be a low impedance path between the power and ground. This is satisfied by the bypass capacitors from power to ground.

As a summary, three common power system design rules are listed below **[11]**.

- Use low impedance ground connections between gates.
- The impedance between power pins on any two gates should be just as low as the impedance between the ground pins.
- There must be low impedance between power and the ground.

Any power system that satisfies three power system design rules will have low common path noise and it also distribute the power to every where on the PCB at uniform voltage.

Crosstalk

Crosstalk arises through unwanted coupling of signal from one line to another. Traces which run in parallel for long distances may cause crosstalk problem mainly due to the mutual inductance. Separating the traces or decreasing their distance from the associated reference plane can decrease the crosstalk. Route orthogonally on adjacent traces.

Ground bounce

Ground bounce is very serious problem in high speed design, when multiple outputs change state simultaneously, they cause undesired transient behavior on a non-switching outputs and even inputs. Ground bounce is primarily due to current changes in the combined inductance in the circuit.

To reduce ground bounce **[12]**,

- Design PCBs with ground and power planes connected directly to the IC's supply pins. Place decoupling capacitors very close to these supply pins.
    - Keep the ground plane as undisturbed as possible. Avoid ground plane discontinuities and decrease the number of vias.
    - Minimize the impedance of the system ground distribution network and its connection to the IC pins.
    - Use ICs having low-inductance pins. BGAs are best suited, PGAs are worst, and QFPs are in-between.

- Reduce the number of simultaneous switching outputs and distribute them throughout the device.
    - Perform synchronous designs that will not be affected by momentarily switching pins.
    - Add 10-50Ω resistors in series to each of the switching outputs to limit the current flow through them.
    - Keep the clock inputs physically away from the outputs that create ground bounce, and connect clocks to input pins that are close to a ground pin.

Test Pattern Generator PCB and Fault Tolerant Systems PCB are 6 layers PCBs. Display PCB is 2 layers PCB.

Layer Stack-up of Test Pattern Generator PCB is given in Table 4.1.

Table 4.1 Layer Stack-up of Test Pattern Generator PCB

Layer Order	Layer Type
Layer 1	Signal
Layer 2	Ground
Layer 3	Power
Layer 4	Signal
Layer 5	Signal + Power
Layer 6	Signal

Layer 5 is designed as a signal layer. But most of this plane is filled by a ground. Test Pattern Generator PCB layout and placement are given in Appendix-B.

Layer Stack-up of Fault Tolerant Systems PCB is given in Table 4.2.

Table 4.2 Layer Stack-up of Fault Tolerant Systems PCB

Layer Order	Layer Type
Layer 1	Signal
Layer 2	Ground
Layer 3	Power
Layer 4	Signal
Layer 5	Signal + Power
Layer 6	Signal

Layer 5 is designed as a signal layer. But most of this plane is filled by a ground. Fault Tolerant Systems PCB layout and placement are given in Appendix-B.

Layer Stack-up of Display PCB is given in Table 4.3.

Table 4.3 Layer Stack-up of Display PCB

Layer Order	**Layer Type**
Layer1	Signal
Layer2	Signal

Display PCB does not have any power or ground planes. Power and grounds are distributed as power and ground buses. Display PCB layout and placement are given in Appendix-B.

CHAPTER 5

FPGA DESIGN

5.1 FAULT TOLERANT SYSTEMS FPGA

Floating point numbers are used in Fault Tolerant Systems FPGA design. First of all, floating point representation is given below.

Floating point numbers can be represented by very different notations. General representation is sign * mantissa * 2 $^{\text{exponent}}$. 15 bits floating point numbers are represented by a combination of mantissa bits and exponent bits in this application.

Floating point number contains 2 fields as shown in Figure 5.1. These fields are mantissa and exponent.

In our application floating point number is represented as 15 bits. First 8 bits of it represent mantissa. It is an 8 bits binary signed number expressed in signed magnitute form and fractional. Binary point is assumed at the right of the sign bit.

For example:
If mantissa part is equal to “0110001”, it refers to (1 * 1/4) + (1 * 1/8) + (1 * 1/128) = 0,3828125 in decimal representation.

Last 7 bits of the floating point number represent the signed exponent in 2's complement form.

For example:

If exponent part is equal to "1110100", it refers to 2^{-12} = 0,000244140625 in decimal representation.

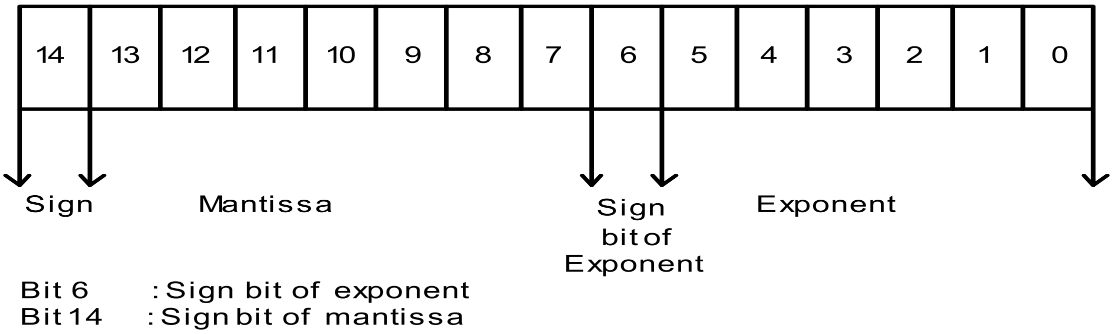


Figure 5.1 15 bits Floating Point Representation

For example, a floating point number "0 1000000 1111001" is equal to + (1 * 1/2) * 2^{-7}. = + 1/256 = 0,00390625 in decimal representation.

The aim of this work initially was to design a fault tolerant floating point multiplier circuit with correction property. But the algorithm to be used was not suitable for a parallel multiplication operation with correction. So, our application multiplies input and the previous value of the floating point multiplier output. By this way, we can apply fault tolerant methods with correction circuit to our circuit.

Floating point multiplier in our design multiplies two numbers. One of them is applied from the software user interface running on a computer. Other one is the previous value of the multiplier output. Initially this value is assigned as "010000000000001" which is equal to 1 in decimal representation.

If we want to multiply two floating point numbers, one of these floating point numbers is applied from the software user interface running on a computer. Multiplier circuit multiplies this number by 1, which is assigned initially to the other input of the multiplier. After first multiplication, output of the multiplier is equal to first one of the floating point numbers. Then, second one of these floating point numbers is applied from the software user interface running on a computer. Multiplier circuit multiplies this number and the previous value of the multiplier output, which is equal to to first one of the floating point numbers. After second multiplication, we can get multiplication of these two numbers on output of the multiplier.

There are 2 fault tolerant methods, implemented in this study. First one of these methods is TMRV and second one is TMRVC. As said previously, there are two systems but they don't operate in parallel. By changing configuration data, which is stored in EEPROM, of Fault Tolerant Systems, operating fault tolerant method on the system is selected. Fault injection commands, which come from Test Pattern Generator, are applied to selected fault tolerant method on the system, and test results are sent back to Test Pattern Generator. These test results are also sent to Display PCB, which is the last PCB on the thesis setup. Implementations of these functions are explained in this section. We divide Fault Tolerant Systems FGPA design into two parts. First of all, TMRV which is applied to floating point multiplier with inner feedback circuit is described. After that second system, TMRVC which is applied to floating point multiplier is described.

5.1.1 TMR WITH VOTER CIRCUIT

TMR with voter circuit which is applied to floating point multiplier with inner feedback is implemented to show fault detection. It has no fault correction property. Floating point multiplier with inner feedback is described in section 5.1.1.2.

This system contains module input fault injection circuit, 3 sub-components (floating point (FP) multiplier inner feedback circuit), and voter circuit. Main diagram of TMRV which is applied to floating point multiplier with inner feedback is shown in Figure 5.2.

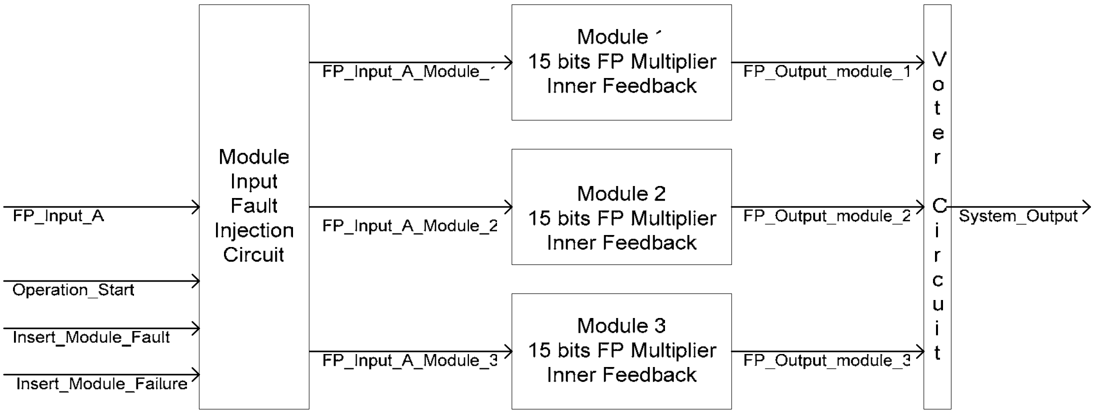


Figure 5.2 Main diagram of TMR with voter circuit

When we insert fault, related module (one of the floating point multiplier inner feedback circuit) gets faulty input. This module produces different multiplication result than other two modules. (It produces faulty output.) Due to inner feedback, this different multiplication result drives FP multiplier input of related module on the next multiplication. So that, any fault injection causes to permanent fault on the related module.

When two of these modules produce same multiplication result, system operates properly due to majority voter circuit. When all of these modules produce different outputs, system fails.

Triple Moduler Redundancy without Correction Block VHDL Code in Appendix-D refers to TMR with voter circuit.

5.1.1.1 Module Input Fault Injection Circuit

Floating point input, Operation_Start, Insert_Module_Fault, and Insert_Module_Failure signals are applied to this circuit. It generates floating point input of 3 sub-components (floating point multiplier). When Insert_Module_Failure is 1, one of these modules (floating point multipliers) gets FP_Input_A as an input. Other two modules get faulty inputs. When Insert_Module_Fault is 1, two of these modules get FP_Input_A as an input. Other module gets faulty input. Floating point multiplication operation starts with the rising edge of the Start_Operation input.

Implementation of Module Input Fault Injection Circuit is shown in Figure 5.3. We use (0111110000000 & Counter_test(1 down to 0)) and (0111000000001 & Counter_test(1 down to 0)) signal bits to insert as a fault to the related module inputs as shown in Figure 5.3.

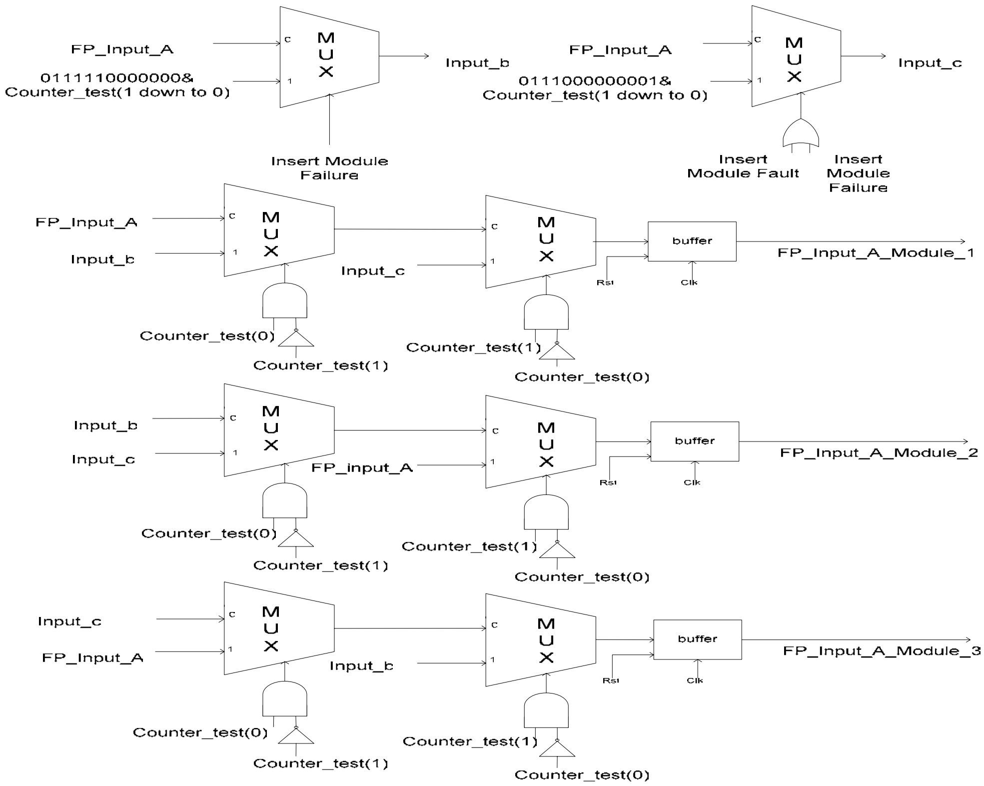


Figure 5.3 Implementation of Module Input Fault Injection Circuit

We use Counter_test signal bits in the implementation of Module Input Fault Injection Circuit. This signal bits are generated as shown in Figure 5.4. Counter_test signal is incremented by one on rising edge of the Operation_Start input. By this way, we can change fault injected module/ modules on each operation.

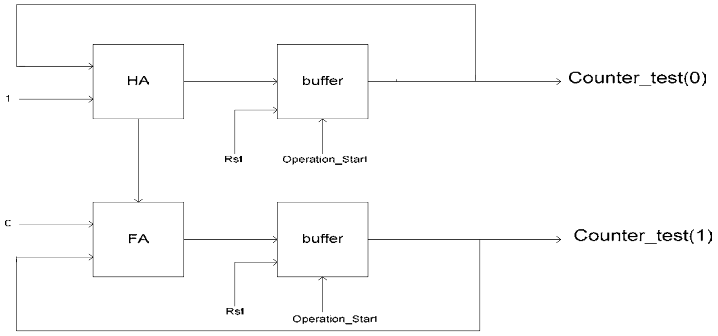


Figure 5.4 Counter_test Generation

5.1.1.2 FP Multiplier Inner Feedback Circuit

Floating Point (FP) Multiplier Inner Feedback Circuit is implemented as shown in Figure 5.5. FP_Input_a is connected to Module Input Fault Injection Circuit. Other multiplier input is connected to output of a FP Multiplier. FP Multiplier output is updated on falling edge of Start_Operation input signal. So that, FP Multiplier multiplies FP_Input_a and previous multiplication result and gets new multiplication result. Initial value of FP Multiplication Output is assigned as 010000000000001, which is equal to 1.

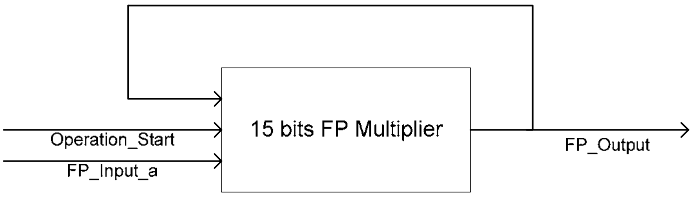


Figure 5.5 15 bits FP Multiplier Inner Feedback Circuit

Floating Point Inner Feedback Block VHDL Code in Appendix-D refers to FP multiplier inner feedback circuit.

5.1.1.3 15 Bits FP Multiplier

Floating Point Multiplier is composed of 6 main blocks as shown in Figure 5.6. Lets explain them seperately.

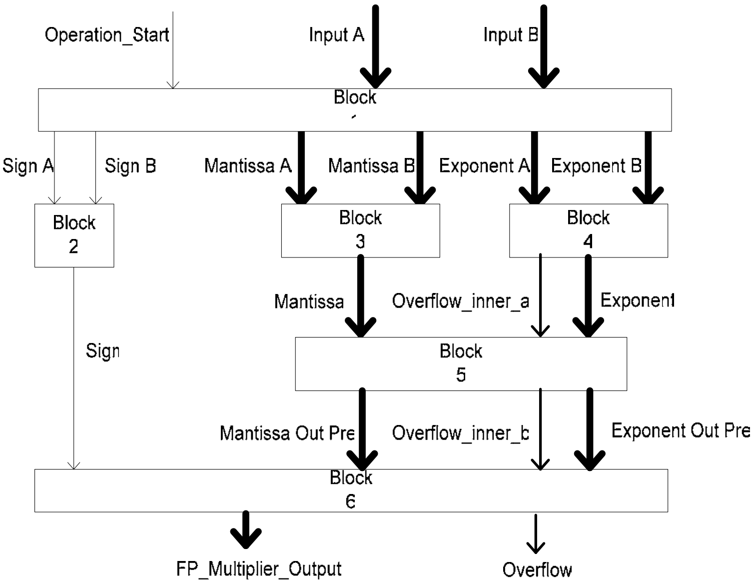


Figure 5.6 Floating Point Multiplier

Floating Point Multiplier Block VHDL Code in Appendix-D refers to 15 bits Floating Point Multiplier.

Block 1:

Operation_Start, Input A and Input B are applied to Block 1. This block generates Sign A, Mantissa A, Exponent A, Sign B, Mantissa B and Exponent B as an output signals. Its operation is given in Figure 5.7.

Input A and Input B are buffered. In other words, Sign A, Mantissa A, Exponent A, Sign B, Mantissa B and Exponent B signals are updated on the rising edge of Operation_Start signal.

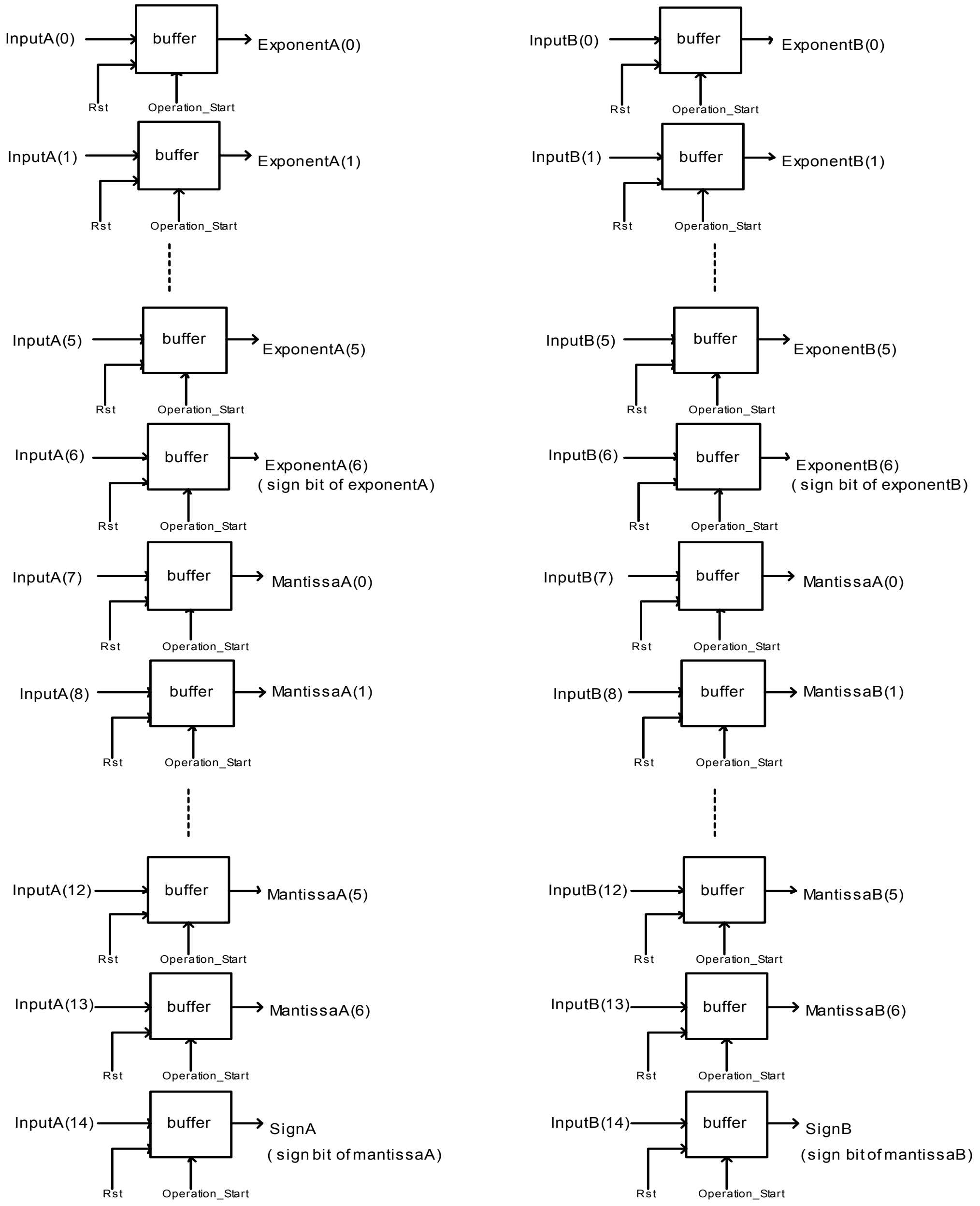


Figure 5.7 Floating Point Multiplier Block 1

Block 2:

Sign bit of input A and sign bit of input B are applied to Block 2. It generates Sign as an output signal. Its operation is given in Figure 5.8.

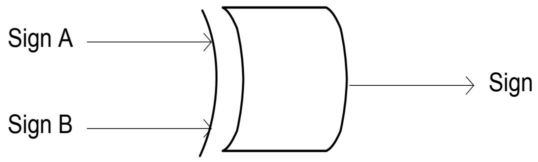


Figure 5.8 Floating Point Multiplier Block 2

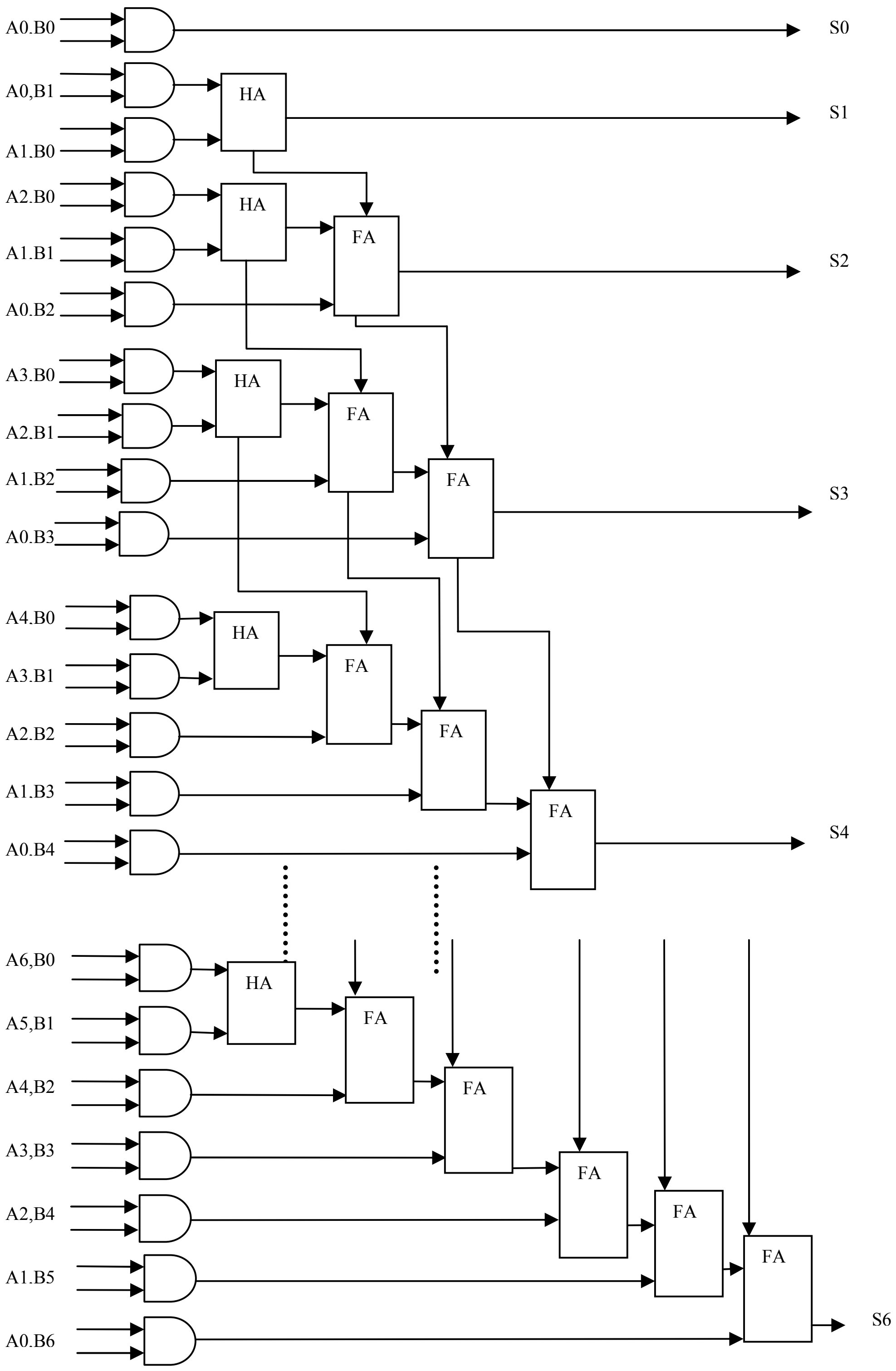
Block 3:

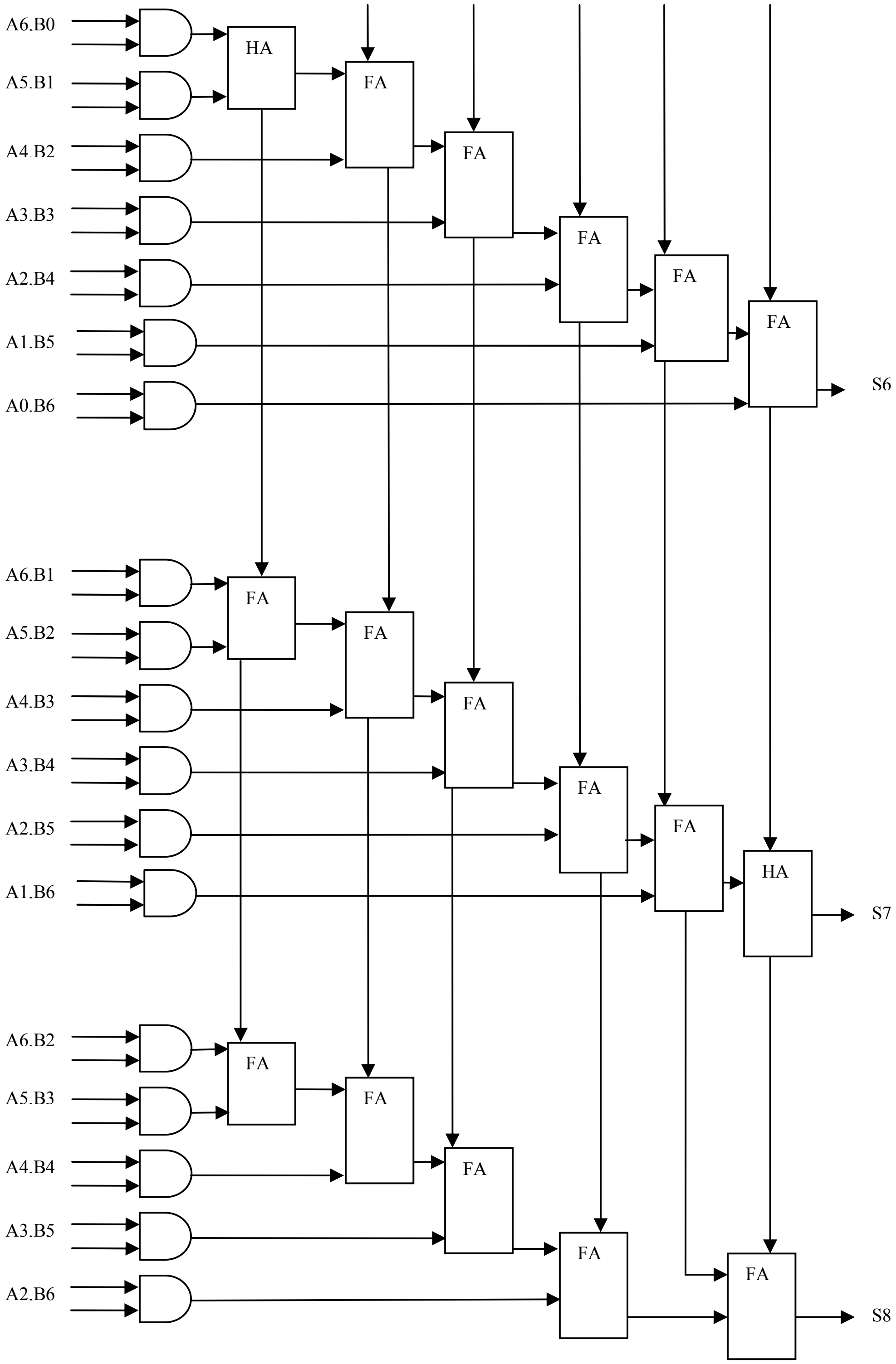
It is 7 bits binary multiplier. 7 bits mantissa part of input A and 7 bits mantissa part of input B are applied to Block 3. It generates 14 bits mantissa as an output.

In the implementation of this block we use Hall Adders (HA) and Full Adders.

While implement Block 3, 7 bits mantissa part of input A is represented as A (6 down to 0). 7 bits mantissa part of input B is represented as B (6 down to 0). 14 bit Mantissa (result of the mantissa multiplication) is represented by S (13 down to 0).

Implementation of Block 3 is given in Figure 5.9.


A0.B0
S0
A0,B1
HA
S1
A1.B0
A2.B0
HA
FA
A1.B1
S2
A0.B2
A3.B0
HA
FA
A2.B1
A1.B2
FA
S3
A0.B3
A4.B0
HA
FA
A3.B1
A2.B2
FA
A1.B3
FA
A0.B4
S4
A6,B0
HA
FA
A5,B1
A4,B2
FA
A3,B3
FA
A2,B4
FA
A1.B5
FA
S6
A0.B6


A6.B0
A5.B1
A4.B2
A3.B3
A2.B4
A1.B5
A0.B6
HA
FA
FA
FA
FA
FA
S6
A6.B1
A5.B2
A4.B3
A3.B4
A2.B5
A1.B6
FA
FA
FA
FA
FA
HA
S7
A6.B2
A5.B3
A4.B4
A3.B5
A2.B6
FA
FA
FA
FA
FA
S8

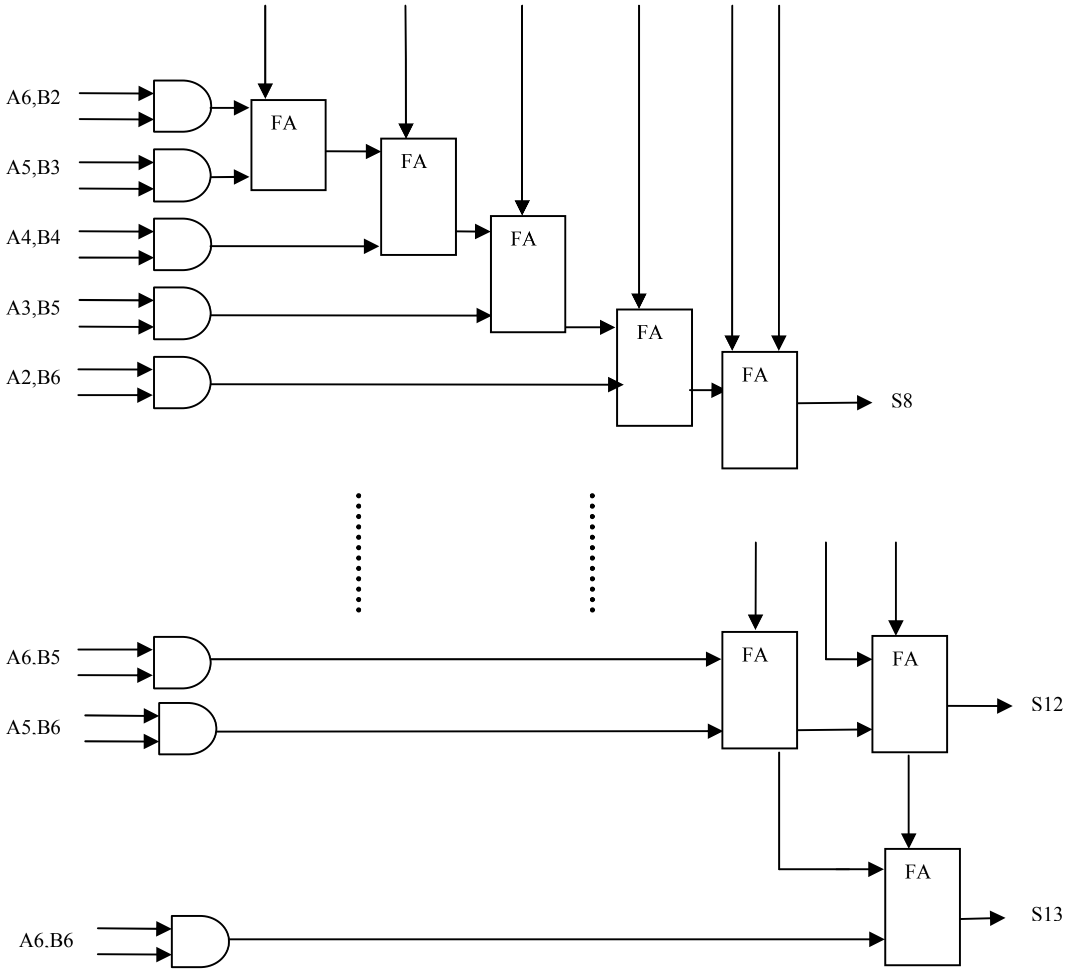


Figure 5.9 Floating Point Multiplier Block 3

Block 4:

It is 7 bits signed adder. 7 bits exponent part of input A and 7 bits exponent part of input B are applied to Block 4. It generates 7 bits Exponent and Overflow_a as an output.

Implementation of Block 4 is given in Figure 5.10.

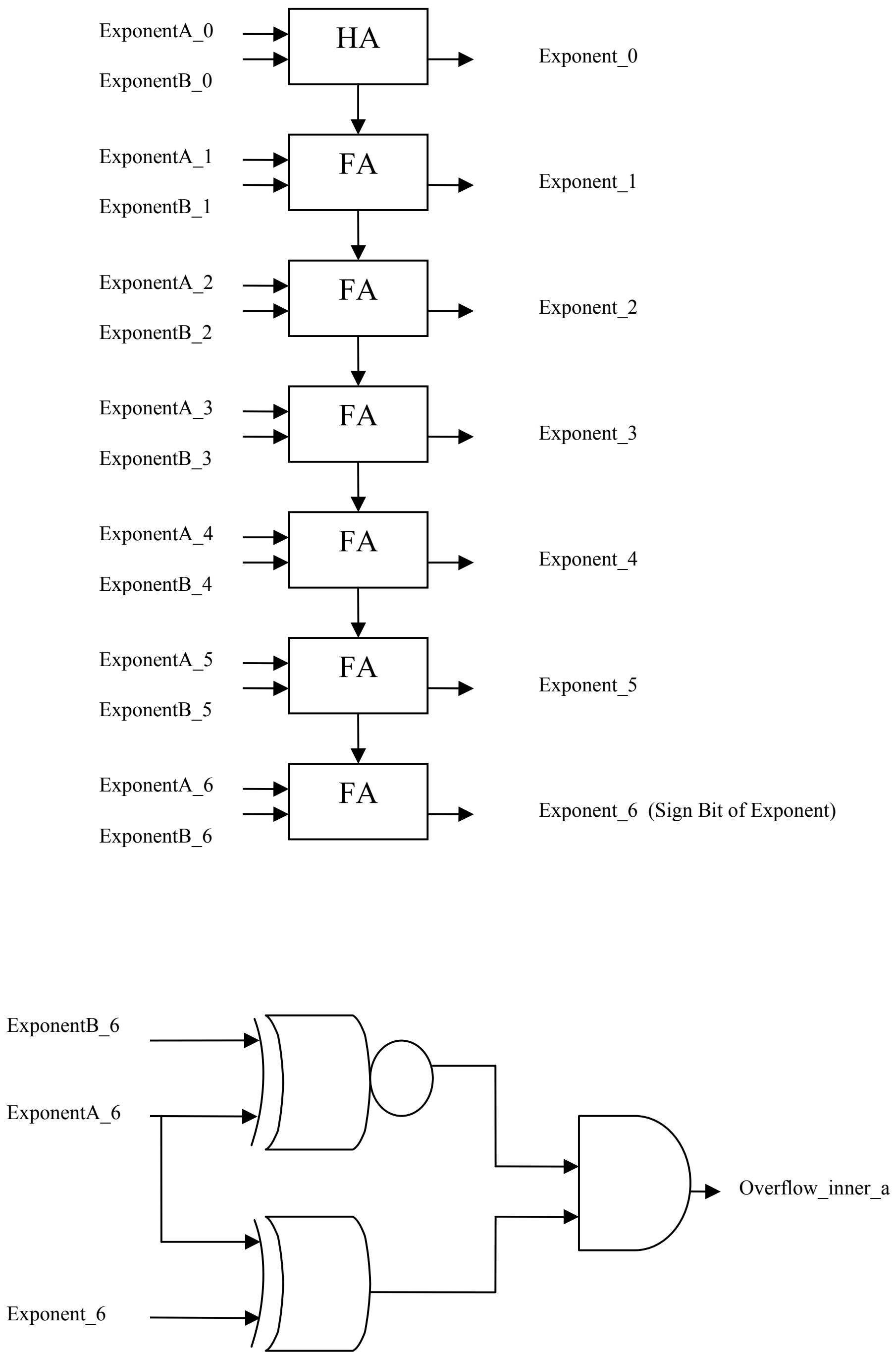


Figure 5.10 Floating Point Multiplier Block 4

Block 5:

It is responsible for normalization of mantissa part; it also adapts exponent part with respect to this normalization.

Floating point numbers must have normalized mantissa. To achieve this, we must normalize the mantissa signal. If we shift mantissa bits one bit position left, we must decrease exponent signal by one.

First of all, mantissa part of the floating point number is normalized. Generation of least significant bit of the Mantissa_Out_Pre is shown in Figure 5.11.

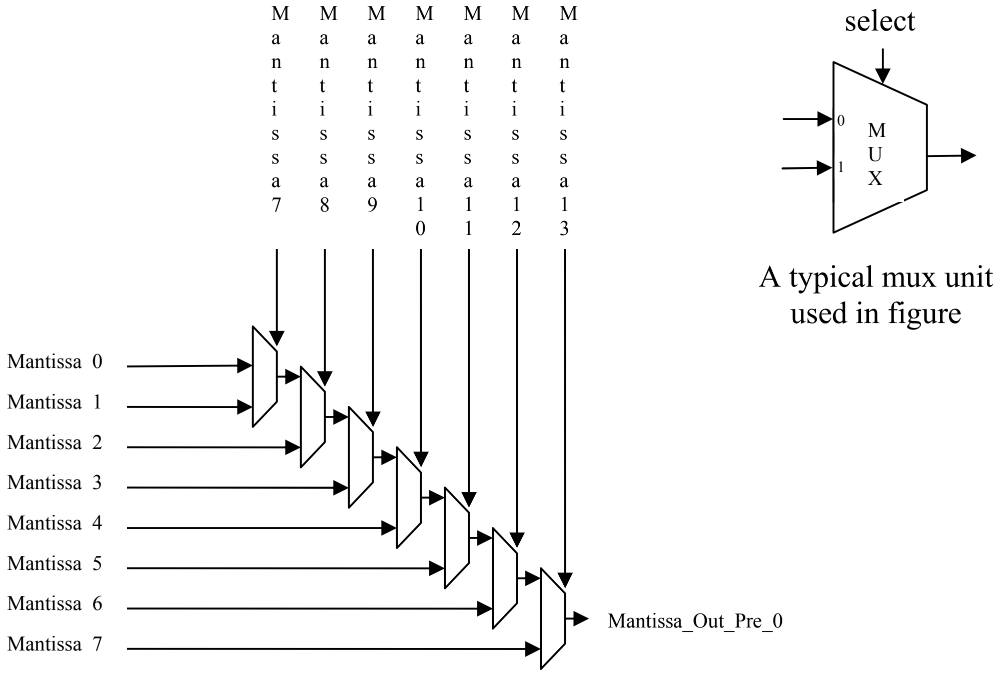


Figure 5.11 Mantissa_Out_Pre_0

Generation of most significant bit of the Mantissa_Out_Pre is shown in Figure 5.12.

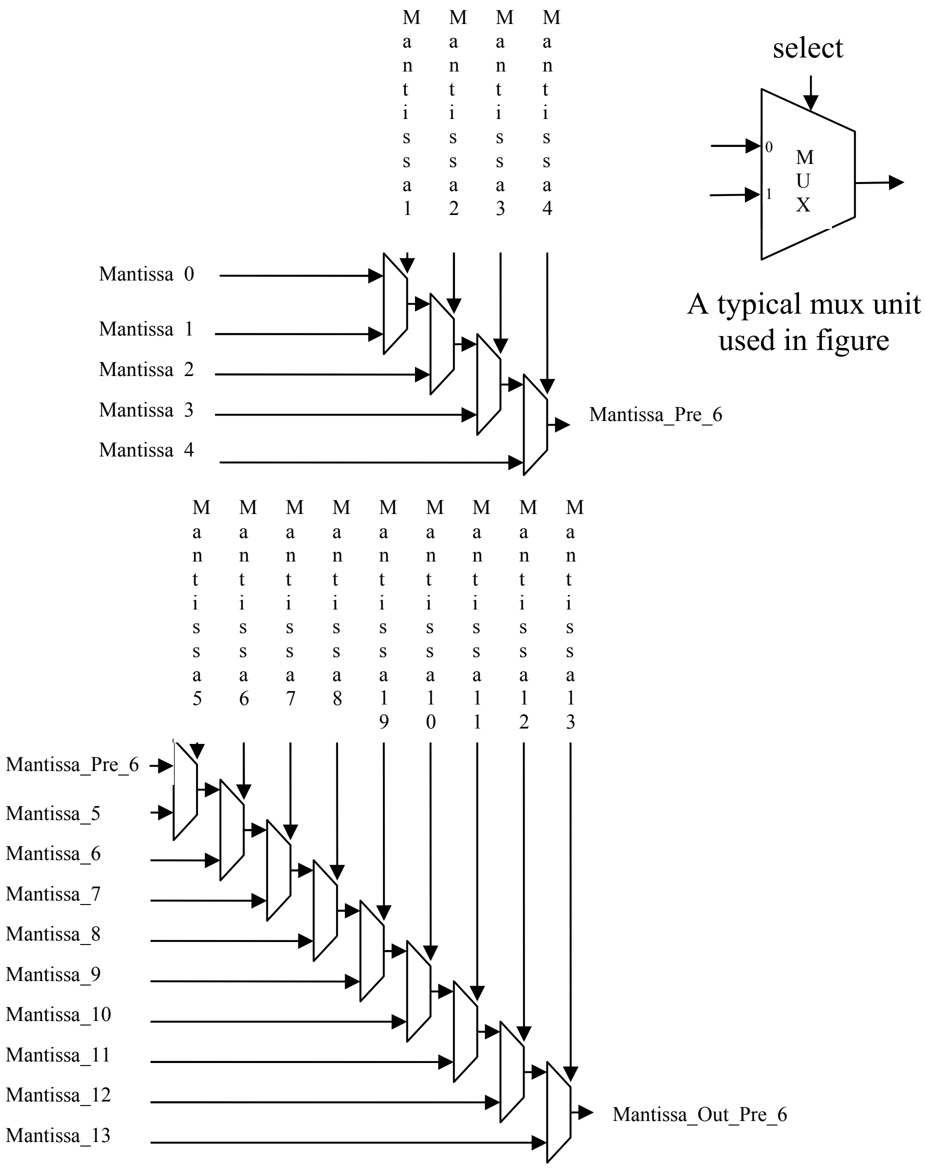


Figure 5.12 Mantissa_Out_Pre_6

Mantissa_Out_Pre is 7 bits signal. Generations of most significant and least significant bits are shown above. Other bits are also generated in the same way.

Number of bits required to shift mantissa left for normalization is calculated. This calculation is shown in Figure 5.13. This number is called as an Exponent_Offset and it represents the number which must be subtracted from Exponent signal.

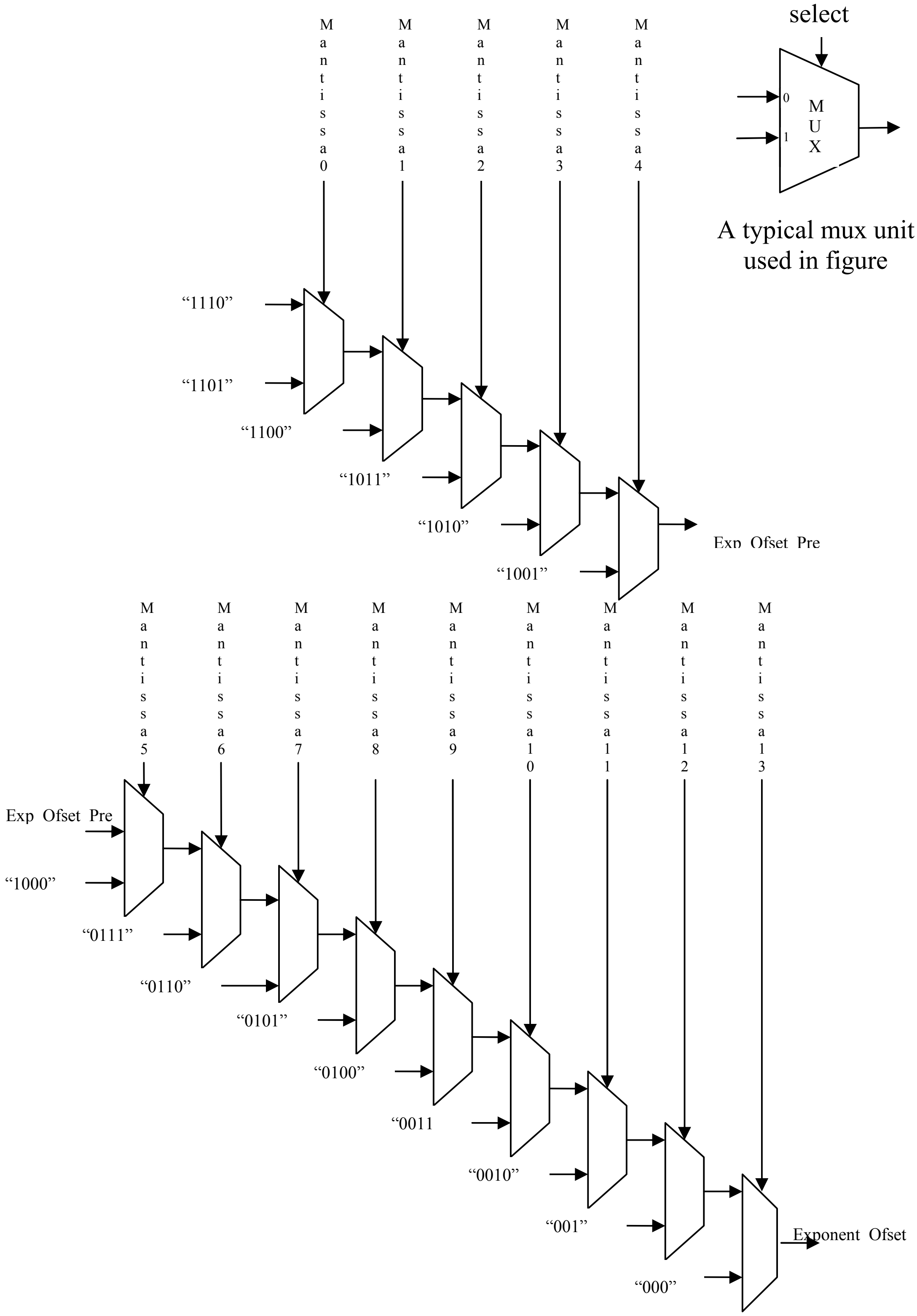


Figure 5.13 Exponent_Offset Generation for Normalization

Exponent_Offset signal is subtracted from exponent signal and Exponent Out Pre is obtained. This operation is shown in Figure 5.14.

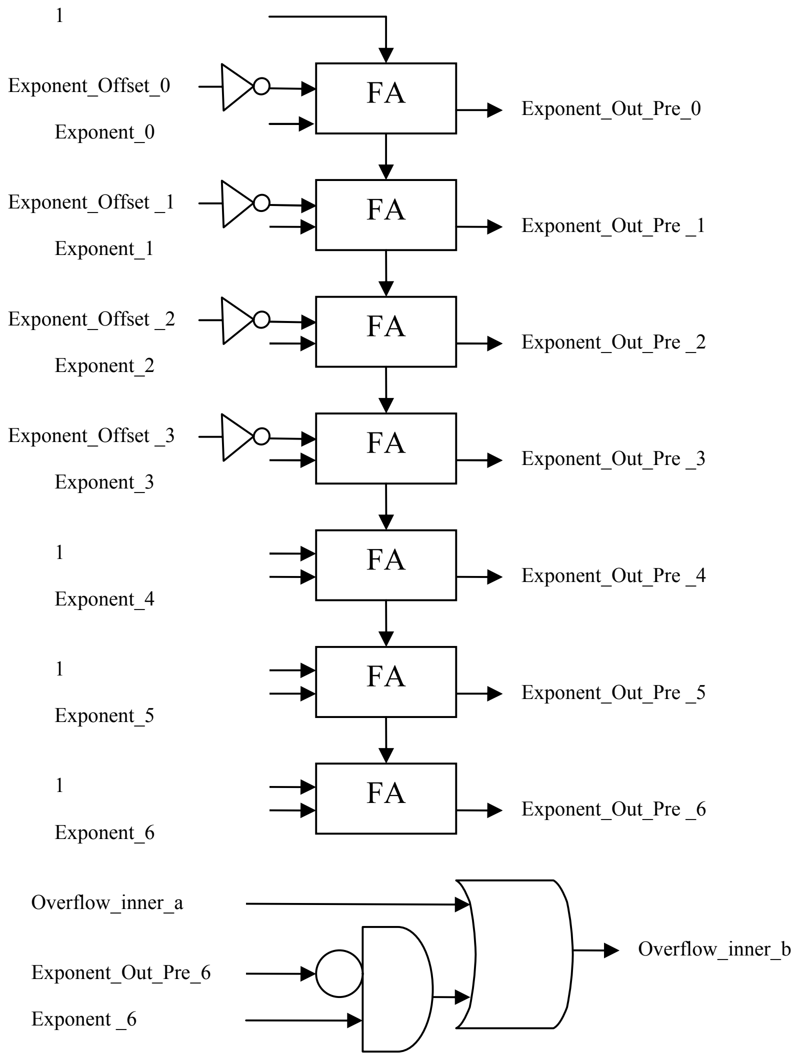


Figure 5.14 Exponent Out Pre Generation

Block 6:

It is responsible for buffering output signals. Mantissa_Out_Pre, Overflow_Inner_b and Exponent_Out_Pre are buffered. FP_Multiplier_Output and Overflow signals are updated on the rising edge of the system clock. This operation is shown in Figure 5.15.

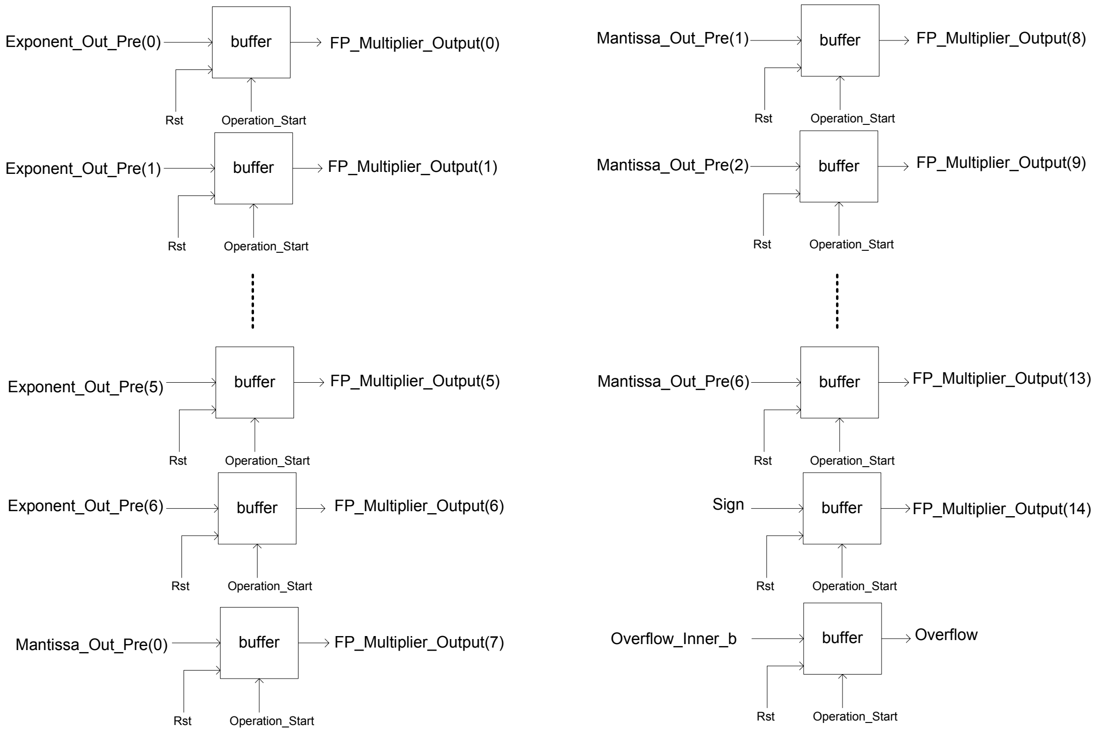


Figure 5.15 Floating Point Multiplier Block 6

5.1.1.4 Voter Circuit

Voter circuit is the last part in the TMRV implementation. Implementation of voter circuit is shown in Figure 5.16.

When two modules outputs (floating point multiplier outputs) are same, System_Output is equal to one of them and Failure_Detection output is equal to 0. When all modules give different outputs, Failure_Detection output is equal to 1. (Failure detected.)

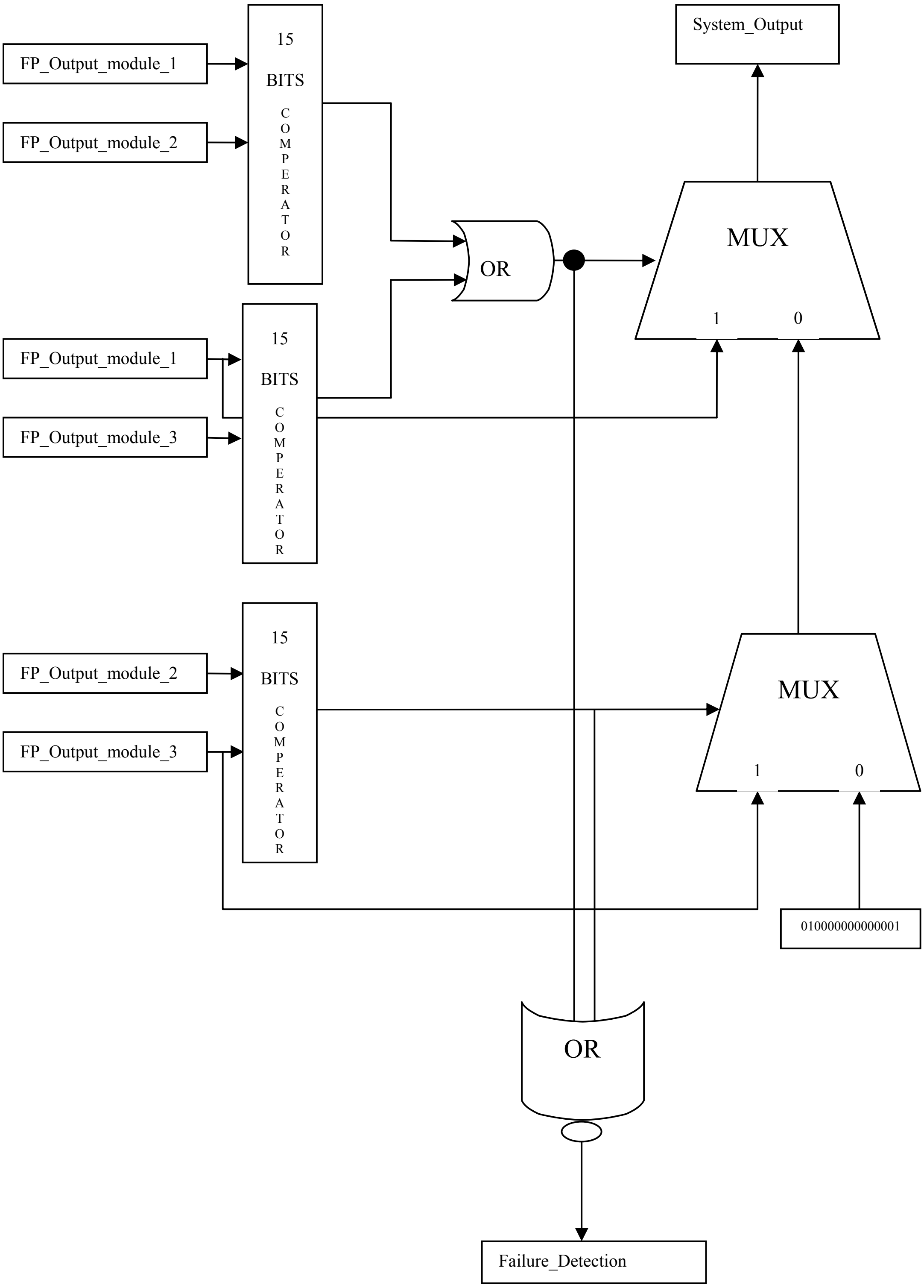


Figure 5.16 Voter Circuit

5.1.2 TMR WITH VOTER AND CORRECTION CIRCUITS

TMRVC which is applied to floating point multiplier circuit is implemented to show fault detection and correction.

This system contains module input fault injection circuit, 3 sub-components (floating point (FP) multiplier), and voter and correction circuits. Main diagram of TMRVC which is applied to floating point multiplier circuit is shown in Figure 5.17.

Triple Moduler Redundancy with Correction Block VHDL Code in Appendix-D refers to TMR with voter and correction circuits.

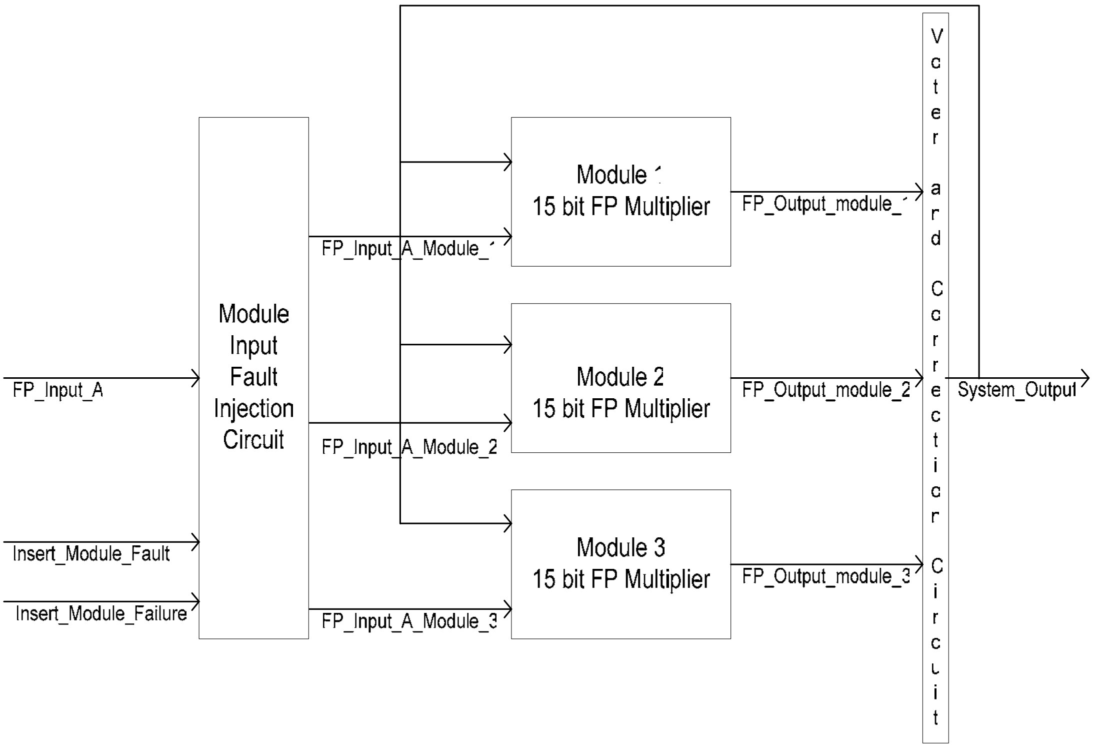


Figure 5.17 Main diagram of TMR with voter and correction circuit

When user inserts fault, related module (floating point multiplier) gets faulty input. This module produces different multiplication result than other two modules. (It produces faulty output.) When two of these modules produce same multiplication result, system operates properly due to majority voter circuit. Correction circuit satisfies correct input to the all modules. So that system repairs itself. When all the modules produce different outputs, (Fault is injected to more than 1 module at the same time.) system fails.

5.1.2.1 Module Input Fault Injection Circuit

Module Input Fault Injection Circuit in TMRVC circuits is same as the Module Input Fault Injection Circuit in TMR with voter circuit. Floating point number, Operation_Start, Insert_Module_Fault, and Insert_Module_Failure signals are applied to this circuit. It generates FP_Input_A of 3 modules. When Insert_Module_Failure is 1, one of these modules gets FP_Input_A as an input. Other two modules get faulty inputs. When Insert_Module_Fault is 1, two of these modules get FP_Input_A as an input. Other module gets faulty input. Floating Point Multiplication operation starts with the rising edge of the Start_Operation input. Implementation of Module Input Fault Injection Circuit is shown in Figure 5.2. (Implementation of Module Input Fault Injection Circuit is same for both methods.)

5.1.2.2 15 Bits FP Multiplier

Block diagram of 15 bits FP Multiplier is shown in Figure 5.18. FP_Input_a is connected to Module Input Fault Injection Circuit. Other multiplier input is connected to System_Output. System_Output is updated on the falling edge of Start_Operation input signal. 15 bits FP Multiplier multiplies FP_Input_b and System Output which comes from Voter and Correction Circuit.

Implementation of 15 Bits FP Multiplier is same for both methods. As a result, we do not repeat this section. It is explained in section 5.1.1.3.

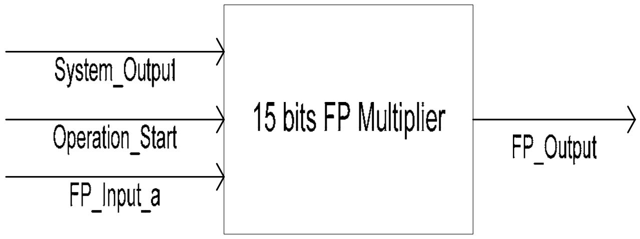


Figure 5.18 15 bits FP Multiplier Block Diagram

5.1.2.3 Voter and Correction Circuits

Voter Circuit:

Voter and Correction Circuit is last part in TMRVC implementation. Implementation of voter circuit is same for both methods. It is shown in Figure 5.16.

When two modules outputs (floating point multiplier outputs) are same, System_Output is equal to one of them and Failure_Detection output is equal to 0. When all modules give different outputs, Failure_Detection output is equal to 1. (Failure detected.).

Correction Circuit:

System_Output drives registered FP_Input_b of three FP Multipliers. This is the correction circuit.

5.2 TEST PATTERN GENERATOR FPGA

5.2.1 Test Pattern Generator FPGA RS232 Interface

There are two input/output pins, assigned for Software communication on Test Pattern Generator FPGA. One of them is called as RS232_in and the other one is called as RS232_out. RS232_in is responsible for receiving fault injection commands from RS232 transceiver and RS232_out is responsible for transmitting fault injection test results to the RS232 transceiver in a defined protocol.

Transfers (transmit/ receive) commands are sending by bytes. Byte transfer is specified as follow:

Initially serial channel is set to 1. Communication starts with the falling edge of this channel. Then, 8 bits (one byte) are transmitted/received with the baud rate of 9600. Finally this channel is set to 1 and new commands are waited for transfer. Byte transfer protocol is shown in Figure 5.19.

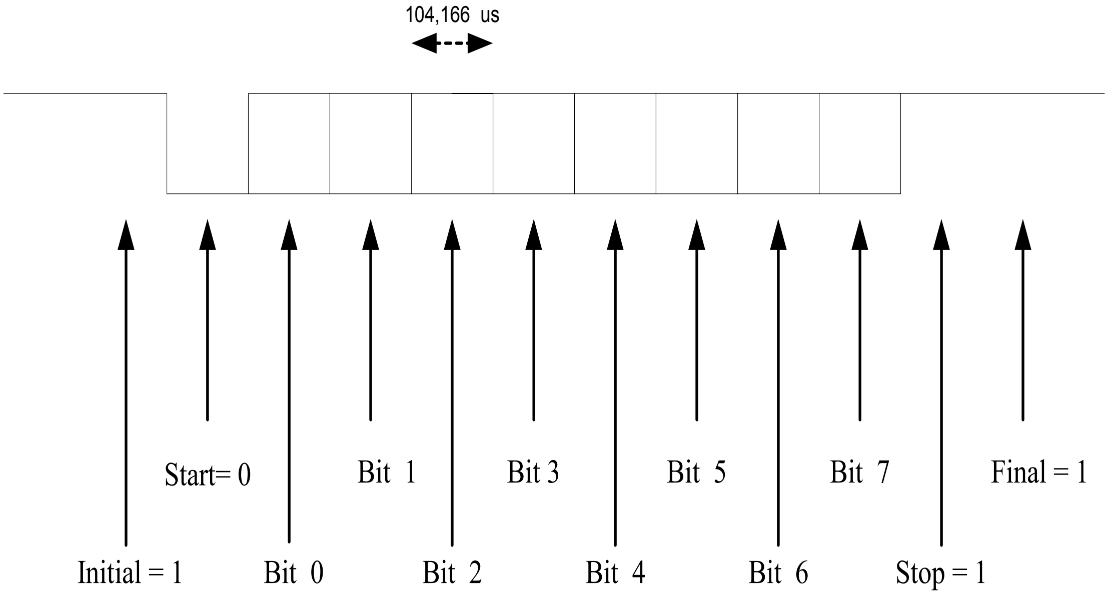


Figure 5.19 Byte transfer protocol

Receiving Fault Injection Commands and Floating Point Multiplier Input:

FPGA waits in the first state. Then, if it takes AA byte from the RS232_in input pin, it gets to second state. If it takes 55 byte from the RS232_in input pin, it gets to fifth state. In other cases, it returns the first state and waits for AA or 55 bytes.

If it gets to second state, it waits for the new byte which includes fault injection commands. When it takes the second byte, these commands are transferred to Fault Tolerant Systems FPGA and Test Pattern Generator FPGA gets to third state, where it finishes the receiving and waits the test results.

If it gets to fifth state, it waits for the new 2 bytes. First one of them includes sign bit and mantissa bits of the floating point multiplier circuit. Second byte contains exponent bits of the floating point multiplier circuit. When Test Pattern Generator FPGA takes these two bytes in fifth state, it returns to first state.

Transmitting Fault Injection Test Results:

FPGA logic drives 1 to RS232_out pin. When FPGA gets to third state, it transmits BB byte from RS232_out pin and it gets to fourth state. In fourth state it transmits fault injection test results in one byte to the computer. Finally, FPGA gets to initial state (first state) and waits new fault injection sequence.

FPGA state diagram for RS232 interface is shown in Figure 5.20.

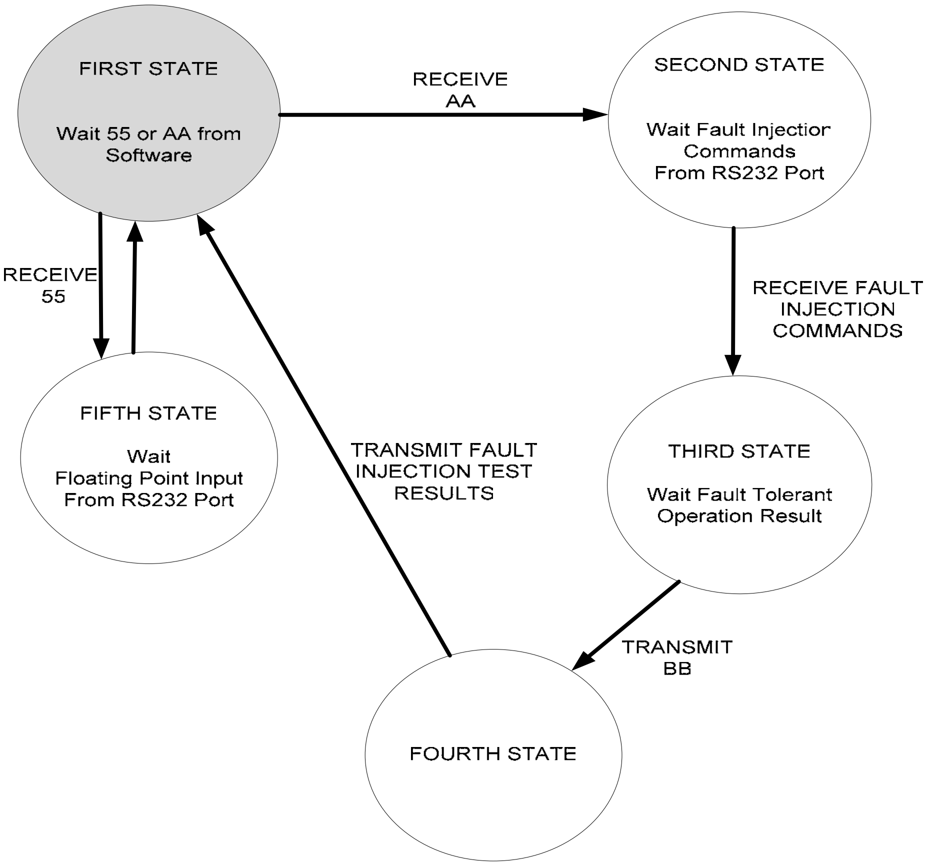


Figure 5.20 State Diagram for RS232 Interface

There are three VHDL source codes for RS232 Interface.

First one of them is called as RS232 Transmitter. This module is responsible for transmitting 8 bits parallel data to the serial port. This VHDL source is given in Appendix-C.

Second one of them is called as RS232 Receiver. This module is responsible for receiving serial data at 9600 baud rate and transfers this signals into 8 bits parallel data. This VHDL source is given in Appendix-C.

Third one of them is called as RS232 Interface. This module is an upper class VHDL code for RS232 Transmitter and RS232 Receiver modules. It coordinates the operation of RS232 Transmitter and RS232 Receiver modules. It generates the state machine, which specifies protocol for software communication. This VHDL source is given in Appendix-C.

CHAPTER 6

IMPLEMENTATION RESULTS OF APPLIED ALGORITHMS

Two fault tolerant methods are tested in this thesis. Both of these methods use triple modular redundancy (TMR) as a fault tolerant method. First one of them uses three identical components (Floating Point Multiplier Inner Feedback Circuit) and voter circuit as said previously. This method is called as TMRV and it is shown in Figure 6.1. Second one uses three identical components (Floating Point Multiplier), voter circuit and correction circuit and it is shown in Figure 6.2.

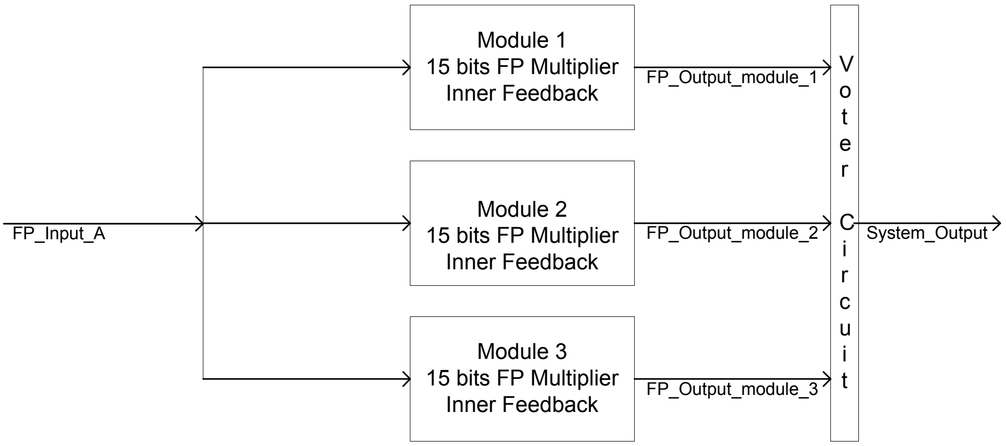


Figure 6.1 TMR with Voter Circuit

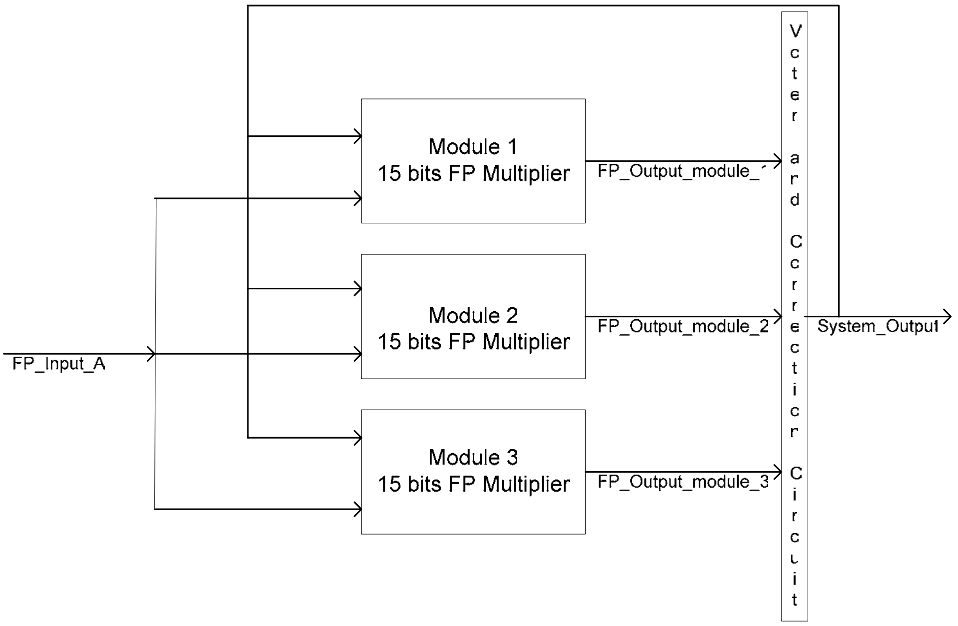


Figure 6.2 TMR with Voter and Correction Circuits

SEU is very serious problem in FPGA applications. When we use three identical components, any error occurred in one of these 3 components can be tolerated. But SEUs, which occurs in the extra logic due to fault tolerance, can not be tolerated.

In TMRV, component is declared as Floating Point Multiplier Inner Feedback Circuit and extra logic due to fault tolerance is declared as a voter circuit.

In TMRVC, component is declared as Floating Point Multiplier and extra logic due to fault tolerance is declared as voter and correction circuits.

If voter circuit in TMRV is smaller than a single component (Floating Point Multiplier Inner Feedback Circuit), TMRV can be preferred to tolerate SEU. Also, if voter and correction circuits in TMRVC are smaller than a single module (Floating Point Multiplier), TMRVC can be preferred to tolerate SEU too. These module sizes are given in the following sections for each method. These values are obtained from FPGA implementation reports.

6.1 Implementation Results of TMR with voter circuit

Main building block of FPGA is called as Configurable Logic Blocks (CLBs). It is composed of Function Generators and Flip Flops.

So that, if the number of flip flops and number of function generators in the voter circuit is smaller than number of these in single component (Floating Point Multiplier Inner Feedback Circuit), applied method is preferable to tolerate SEU. FPGA placement report summary is given in Table 6.1. This report shows used flip flop and function generator quantities in single component (Floating Point Multiplier Inner Feedback Circuit) and in TMRV (in the system).

Table 6.1 TMR with voter circuit FPGA placement report summary

	In system (In TMR with voter circuit)	In single module (In 15 bits Floating Point Multiplier Inner Feedback Circuit)
Number of CLB Flip Flops	261	87
Number of Function Generators	702	217

Number of flip flops used in the voter circuit = ((Total number of flip flops) – 3 x (number of flip flops in the FP multiplier inner feedback circuit)).

Number of flip flops used in the voter circuit = 261 – 3 x 87
Number of flip flops used in the voter circuit = 0

SEU mitigation percentage of flip flops = Number of flip flops used in the voter circuit / Number of flip flops used in the FP multiplier inner feedback circuit x 100

SEU mitigation percentage of flip flops = 0 / 87 x 100

SEU mitigation percentage of flip flops = % 0

Number of function generators used in the voter circuit = ((Total number of function generators) – 3 x (number of function generators in the FP multiplier inner feedback circuit)).

Number of function generators used in the voter circuit = 702- 3 x 217

Number of function generators used in thevoter circuit = 51

SEU mitigation percentage of function generators = Number of flip flops used in the voter circuit / Number of flip flops used in the FP multiplier inner feedback circuit *100

SEU mitigation percentage of function generators = 51 / 217 x 100

SEU mitigation percentage of function generators = % 23, 5

6.2 Implementation Results of TMR with voter and correction circuit

FPGA placement report summary for TMRVC is given in Table 6.2. This report shows used flip flop and function generator quantities in single component and in TMRVC (in the system).

Table 6.2 TMR with voter and correction circuit FPGA placement report summary

	In system (In TMR with voter and correction circuit)	In single module (In 15 bits Floating Point Multiplier)
Number of CLB Flip Flops	336	103
Number of Function Generators	698	213

Number of flip flops used in voter and correction circuits **=** ((Total number of flip flops) – 3 x (number of flip flops in the FP multiplier circuit)).

Number of flip flops used in voter and correction circuits = 336 – 3 x 103
Number of flip flops used in voter and correction circuits = 27

SEU mitigation percentage of flip flops = Number of flip flops used in voter and correction circuits / Number of flip flops used in the FP multiplier circuit x 100

SEU mitigation percentage of flip flops = 27 / 103 x 100
SEU mitigation percentage of flip flops = % 26, 2

Number of function generators used in voter and correction circuits = ((Total number of function generators) – 3 x (number of function generators in the FP multiplier circuit)).

Number of function generators used in voter and correction circuits = 698- 3 x 213
Number of function generators used in voter and correction circuits = **57**

SEU mitigation percentage of function generators = Number of flip flops used in voter and correction circuits / Number of flip flops used in the FP multiplier inner feedback circuit *100

SEU mitigation percentage of function generators = 57 / 213 x 100
SEU mitigation percentage of function generators = % 26, 7

6.3 Advantages and Disadvantages of Applied Algorithms

As shown in the previous section,

Implementation results of TMR with voter circuit:

SEU mitigation percentage of flip flops = % 0
SEU mitigation percentage of function generators = % 23, 5

Voter circuit does not use flip flops. So that, probability of SEU occurrence at these flip flops is zero. Number of function generators used in the voter circuit is almost ¼ of the number of function generators used in the FP multiplier inner feedback circuit. So that, probability of SEU occurrence at function generators in voter circuit is less than the probability of SEU occurrence at function generators in the FP multiplier inner feedback circuit.

So, TMR with voter circuit is a preferable method to tolerate SEU in this application.

Implementation results of TMR with voter and correction circuits:

SEU mitigation percentage of flip flops = % 26, 2
SEU mitigation percentage of function generators = % 26, 7

Numbers of flip flops and function generators used in voter and correction circuits are almost ¼ of the numbers of flip flops and function generators used in the FP multiplier circuit. So that, probability of SEU occurrence at flip flops and function generators in voter and correction circuits is less than the probability of SEU occurrence at flip flops and function generators in the FP multiplier circuit.

So, TMR with voter and correction circuits is a preferable method to tolerate SEU in this application too.

Number of flip flops and function generators in the voter circuit of TMRV is smaller than the number of flip flops and function generators in voter and correction circuits of TMRVC. When we deal with only SEU detection, TMRV seems to be preferable than TMRVC. But don’t forget that, TMRVC has repair property and extra logic due to correction circuit is very small. So, TMRVC can be used as a fault tolerant method, when we need only fault detection or fault detection and correction.

CHAPTER 7

CONCLUSIONS

The aim of this thesis is to investigate the behaviour of hardware fault tolerant methods as the implementation of dependable computing application. In this work, a floating point multiplier unit is selected and implemented in the FPGA to tolerate SEU. Two methods are tested. Both of these methods use triple modular redundancy (TMR) as a fault tolerant method. First one of them uses three identical components (Floating Point Multiplier Inner Feedback Circuit) and a voter circuit. This method is called as TMRV. Second one uses three identical components (Floating Point Multiplier), voter circuit and correction circuit. It is called as TMRVC.

In TMRV, when one of these 3 components (Floating Point Multiplier Inner Feedback Circuit) gets faulty output due to transient error, it is ignored by the voter circuit but after that, faulty component could not operate regularly. In TMRVC, when one of these modules (Floating Point Multiplier) gets faulty output due to transient error, it is ignored by the voter circuit and correction circuit repairs faulty module. After this error, faulty component could operate regularly.

Main consideration in SEU tolerance is area constraint. If extra logic due to fault tolerance is bigger than one of the components in TMR, probability of SEU in the extra logic due to fault tolerance is bigger than the probability of SEU in a single component. In this case, applied fault tolerant technique is not preferred for tolerating SEU.

Extra logic added due to fault tolerance is voter circuit in TMRV and it is smaller than a single component (Floating Point Multiplier Inner Feedback Circuit). So, TMRV is preferable for tolerating SEU.

Extra logic added due to fault tolerance is voter and correction circuits in TMRVC. Extra logic due to fault tolerance is smaller than a single component (Floating Point Multiplier). So, TMRVC is preferable for tolerating SEU too.

When we need fault correction TMRVC is preferred method. Voter circuit in TMRV is smaller than voter and correction circuits in TMRVC. When we only deal with SEU detection, TMRV seems to be preferable. But extra logic added due to correction is very small. So we can use TMRVC when we need SEU detection too.

In our application, we use Leonardo Spectrum as a synthesis program. If we use another synthesis programs (Precision Synthesis, XST ...), implementation outputs may change. But all of these synthesis programs produce similar outputs and our implementation outputs do not change very much with the selected synthesis program.

REFERENCES

[1] Barry W. Johnson, “Design and Analysis of Fault Tolerant Digital Systems”, 1989

[2] M. Nikoladis, “Time Redundancy Based Soft-Error Tolerance to Rescue Nanometer Technologies”, IEEE 17’Th VLSI Test Symposium, April 1999, pp. 86-94

[3] M. Violante, M. Ceschia, M. Sonza Reorda, A. Paccagnella, P. Bernardi, M. Rebaudengo, D. Bortolato, M. Bellato, P. Zambolin, and A. Candelori, “Analyzing SEU Effects in SRAM-based FPGAs”, IEEE International On-Line Testing Symposium, pp. 119-123, 2003

[4] Sandi Habinc, “Functional Triple Modular Redundancy (FTMR): VHDL Design Methodology for Redundancy in Combinatorial and Sequential Logic Design and Assessment Report”, 12/2002

[5] Jamil Khatib, “Introduction to Programmable Logic Devices”, http://www.geocities.com/jamilkhatib75/fpga/FPGA_intro.html, 2002

[6] XILINX, XC4000E and XC4000X Series Field Programmable Gate Arrays Data Sheet, 05/1999

[7] JETTA - Journal of Electronic Testing - Theory and Applications, vol. 17, pp. 163-174, 2001.

[8] Atmel, AT17C512/010 FPGA Configuration EEPROM Memory Data Sheet, 06/2001

[9] Q-Tech Corporation, M55310/26B Data Sheet, 07/1990

[10] Dallas Semiconductor, Dual RS232 Transmitter/Receiver Data Sheet

[11] ASELSAN, High Speed PCB Design Guide: Power System Design, 2004

[12] ASELSAN, High Speed PCB Design Guide: Ground Bounce, 2004

[13] Andrew S. Tanenbaum & Maarten van Steen, Distributed Systems: Principles and Paradigms, 2001

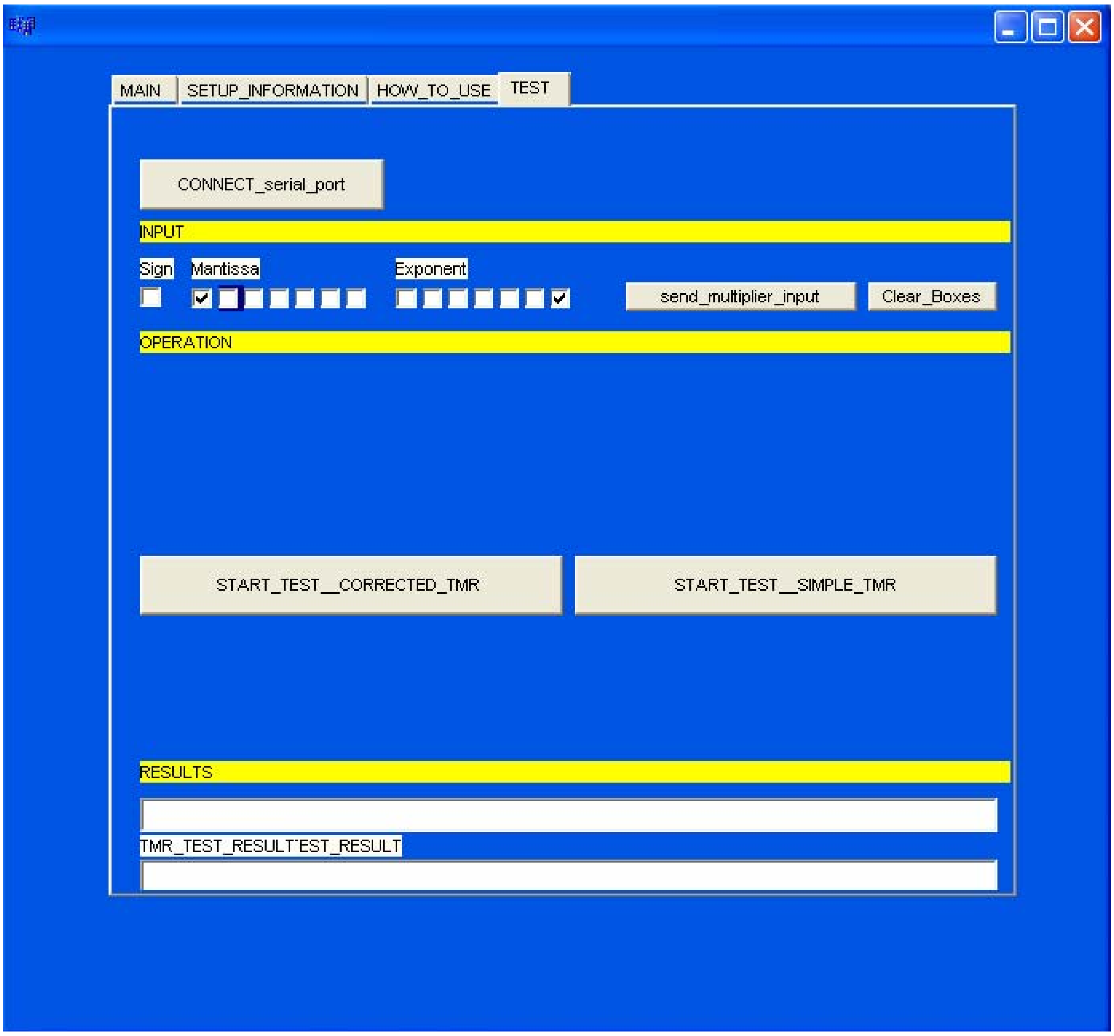
[14] Xilinx, “RTL Synthesis Tools”, http://www.xilinx.com/products/design_resources/design_tool/grouping/synthesis.htm

[15] Xilinx Signal Processing Group, “FPGAs: The High End Alternative for DSP Applications”, 2000

APPENDIX-A

SETUP COMPONENTS

There are 4 components in our setup. They are shown below.



Sign ☐
Mantissa ☑ ☐ ☐ ☐ ☐ ☐ ☐
Exponent ☐ ☐ ☐ ☐ ☐ ☐ ☑

Figure A.1 Software user interface of thesis application

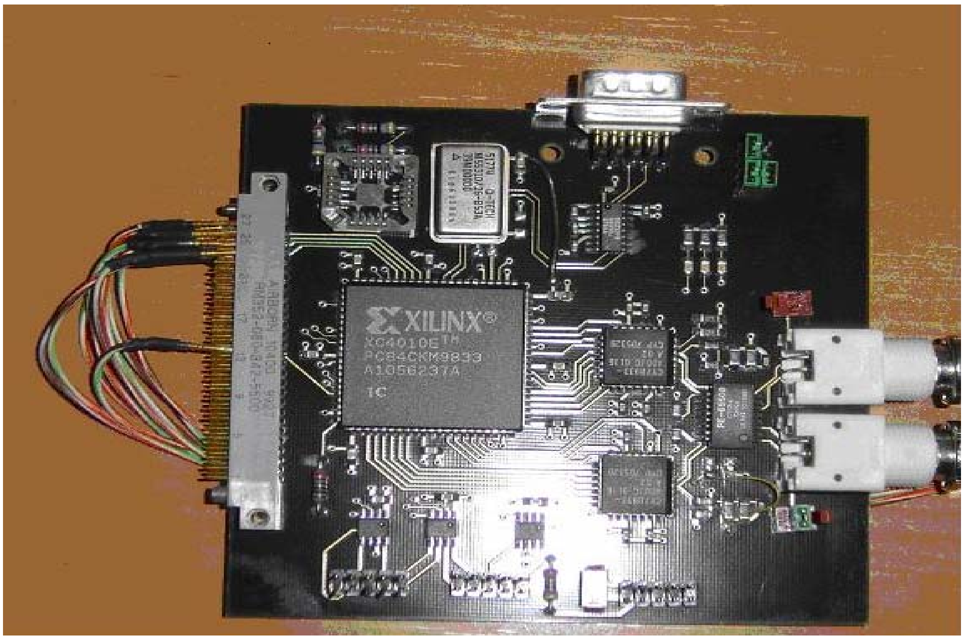


Figure A.2 Test Pattern Generator

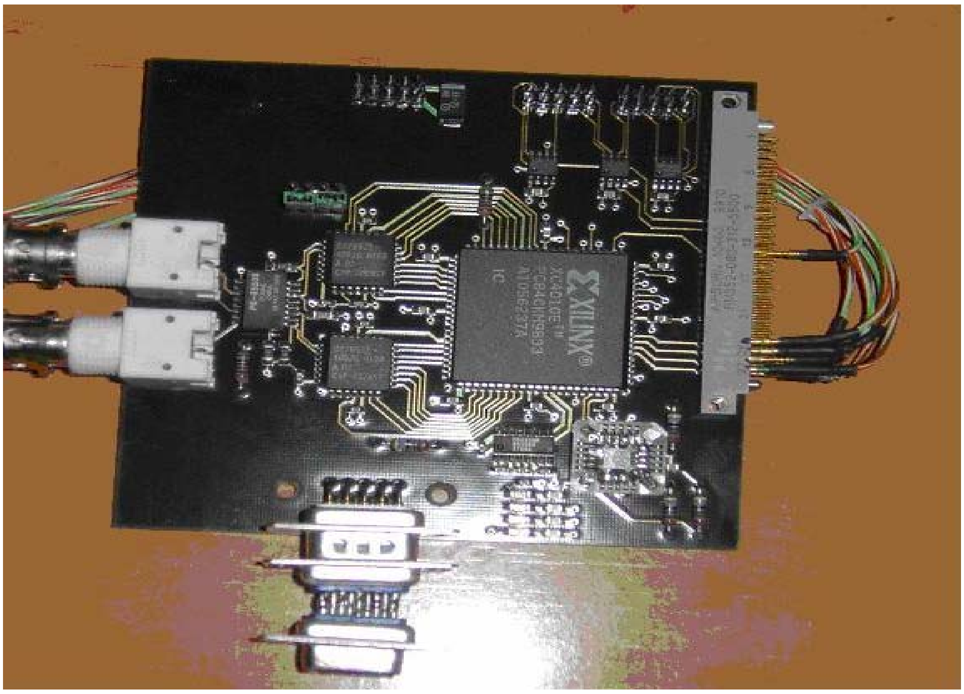


Figure A.3 Fault Tolerant Systems

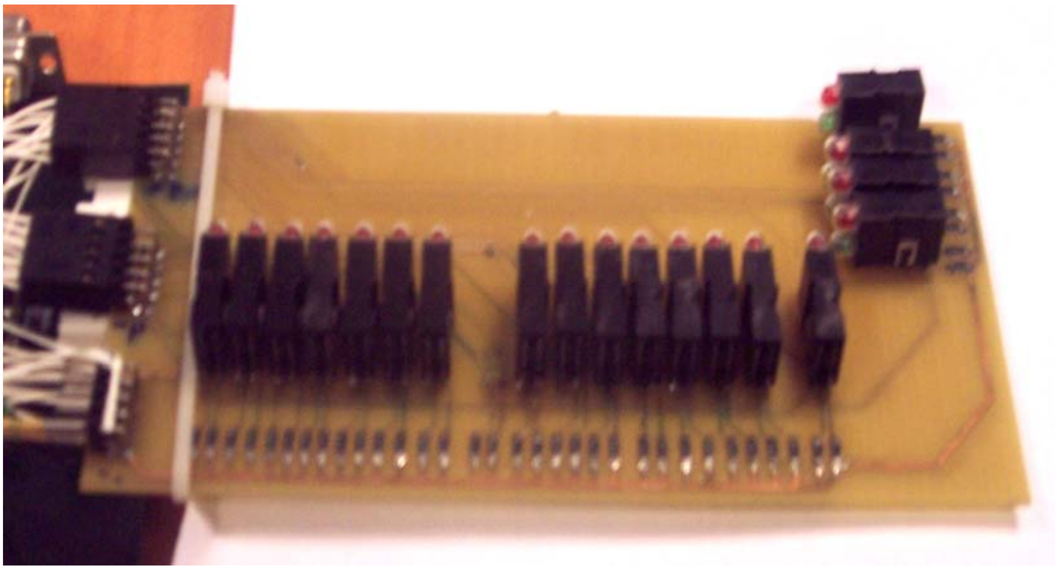


Figure A.4 Display PCB

APPENDIX-B

PCB LAYOUT

TEST PATTERN GENERATOR PCB LAYOUT

Test Pattern Generator is designed as a six layers Printed Circuit Board (PCB). All components are located on the upper side of the PCB. First layer is assigned for component placement and signal routing. Second layer is assigned for a ground. Third layer is assigned for a power. Other three layers are assigned for signal routing. These layers are shown in the following figures.

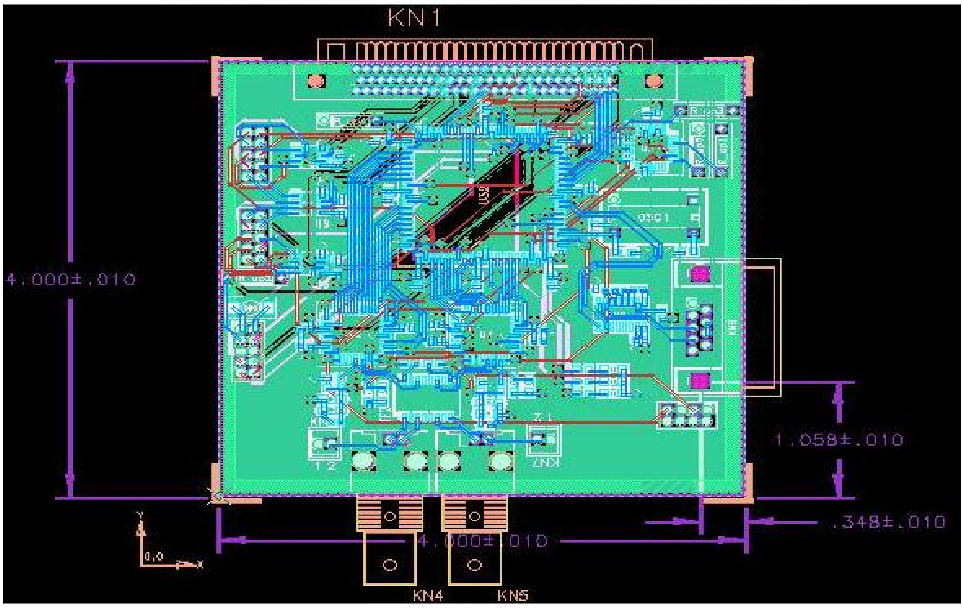


Figure B.1 Test Pattern Generator components, signal routings and dimensions

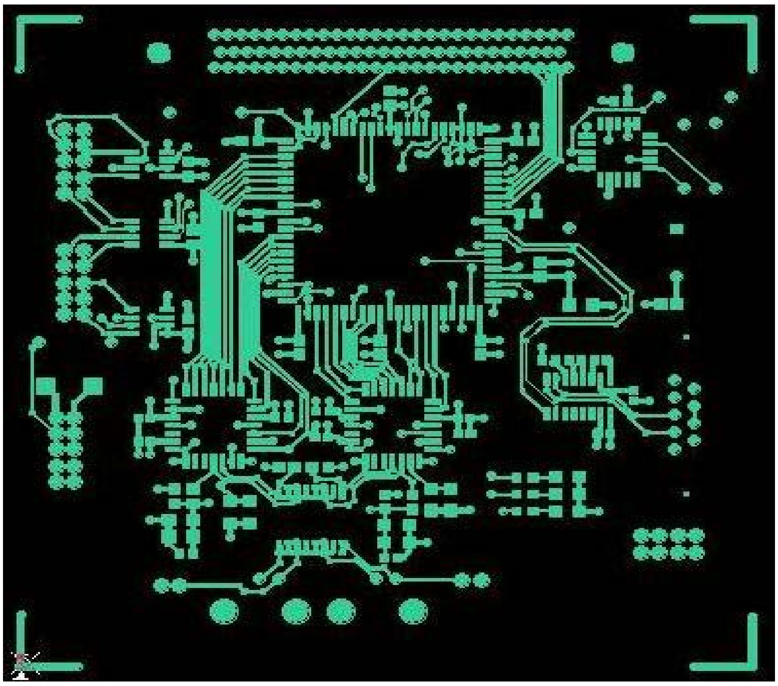


Figure B.2 Test Pattern Generator First Layer

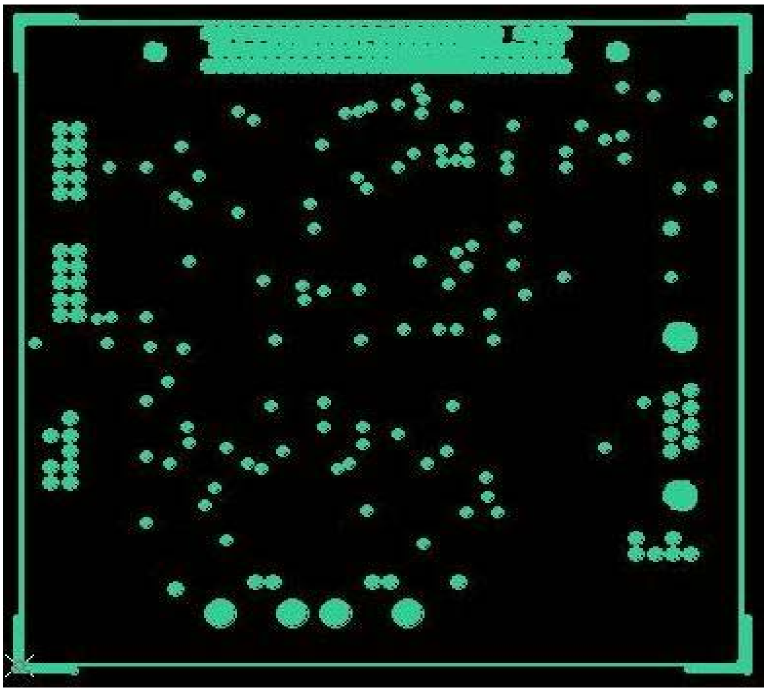


Figure B.3 Test Pattern Generator Second Layer (Ground Layer)

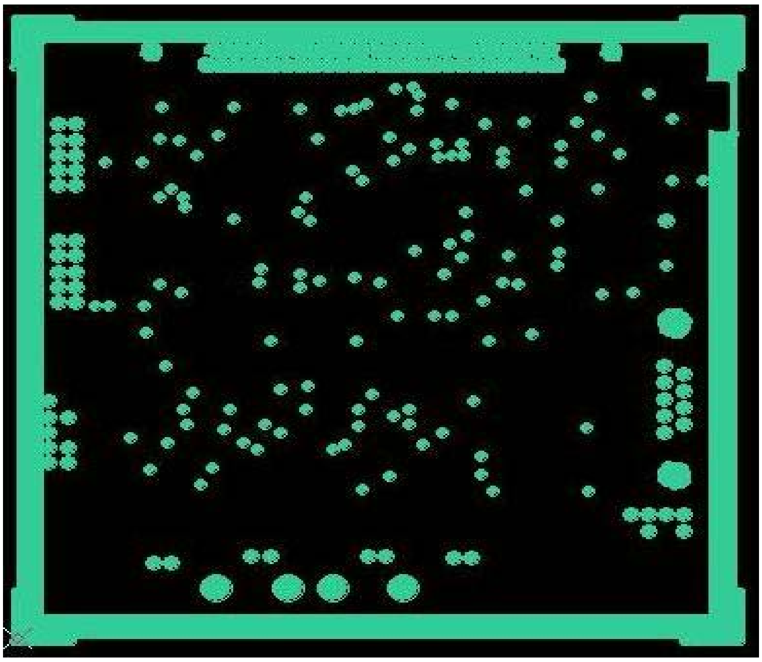


Figure B.4 Test Pattern Generator Third Layer (Power Layer)

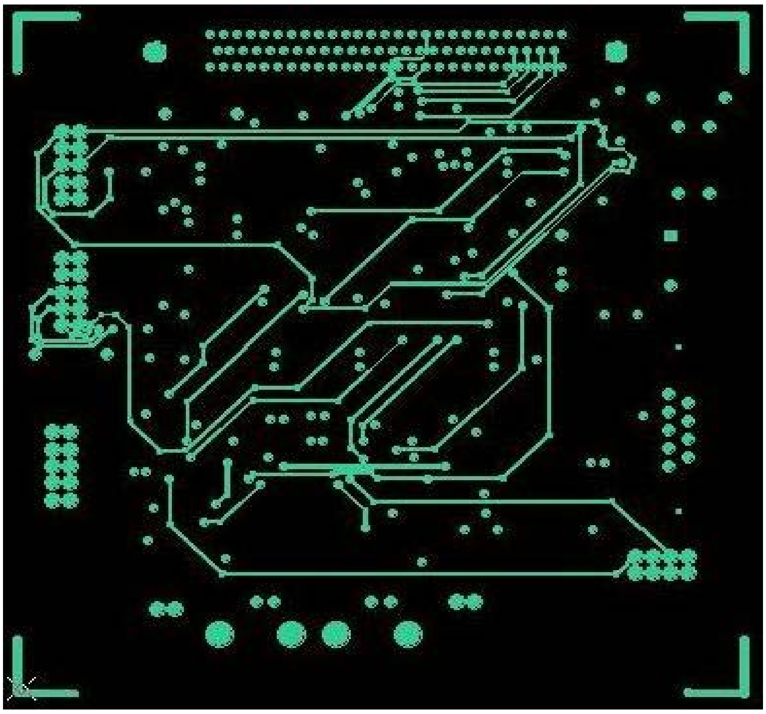


Figure B.5 Test Pattern Generator Fourth Layer

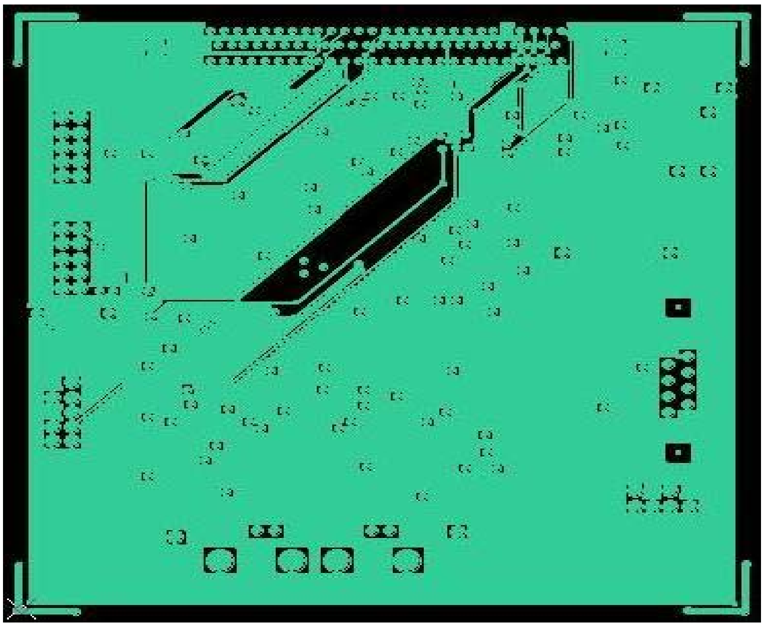


Figure B.6 Test Pattern Generator Fifth Layer

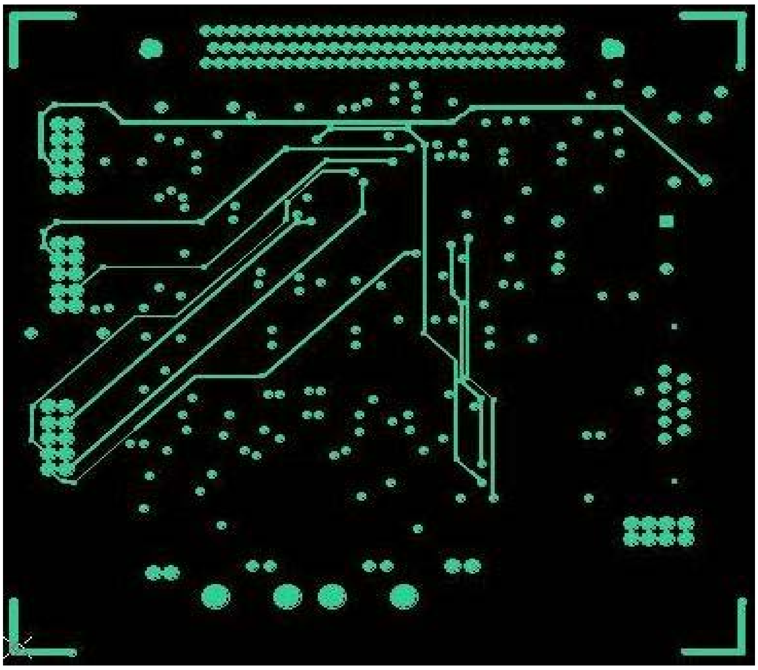


Figure B.7 Test Pattern Generator Sixth Layer

FAULT TOLERANT SYSTEMS PCB LAYOUT

Fault Tolerant Systems is designed as a six layers Printed Circuit Board (PCB). All components are located on the upper side of the PCB. First layer is assigned for component placement and signal routing. Second layer is assigned for a ground. Third layer is assigned for a power. Other three layers are assigned for signal routing. These layers are shown in the following figures.

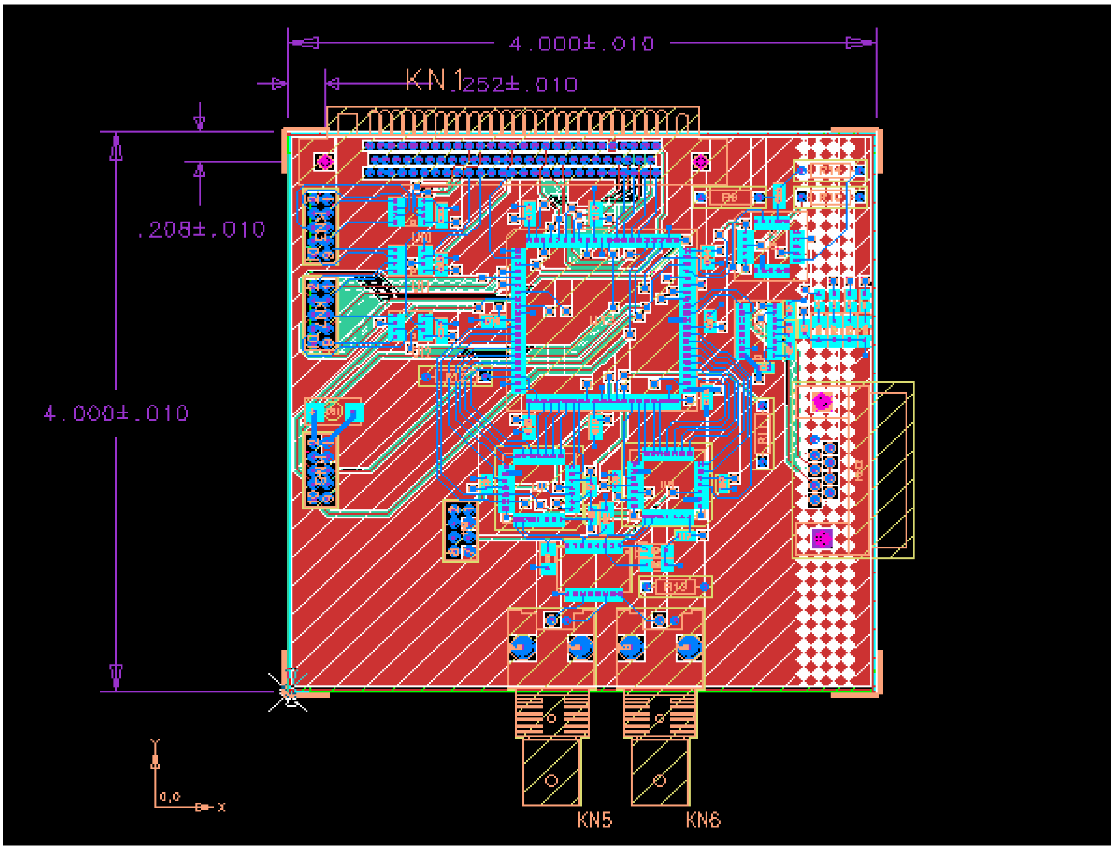


Figure B.8 Fault Tolerant Systems components, signal routings and dimensions

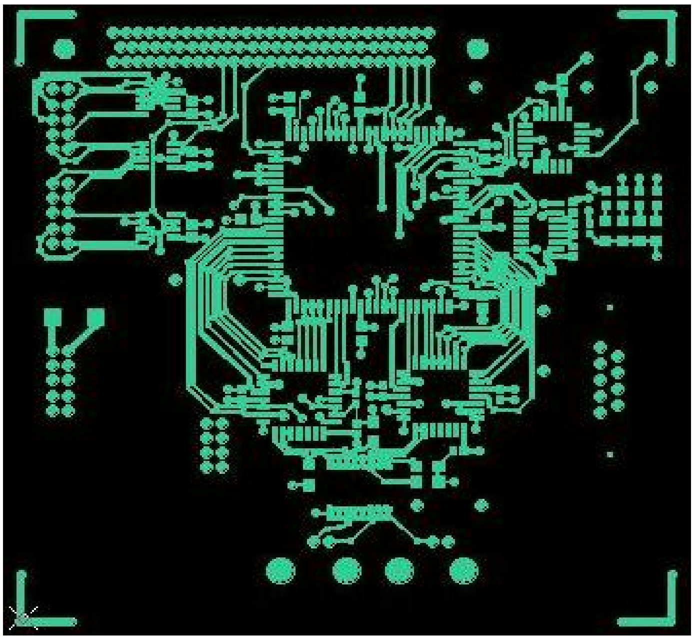


Figure B.9 Fault Tolerant Systems First Layer

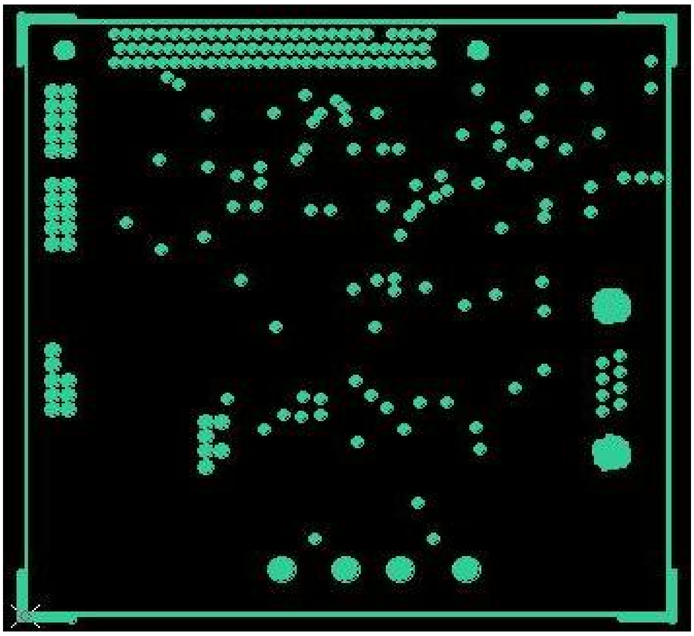


Figure B.10 Fault Tolerant Systems Second Layer (Ground Layer)

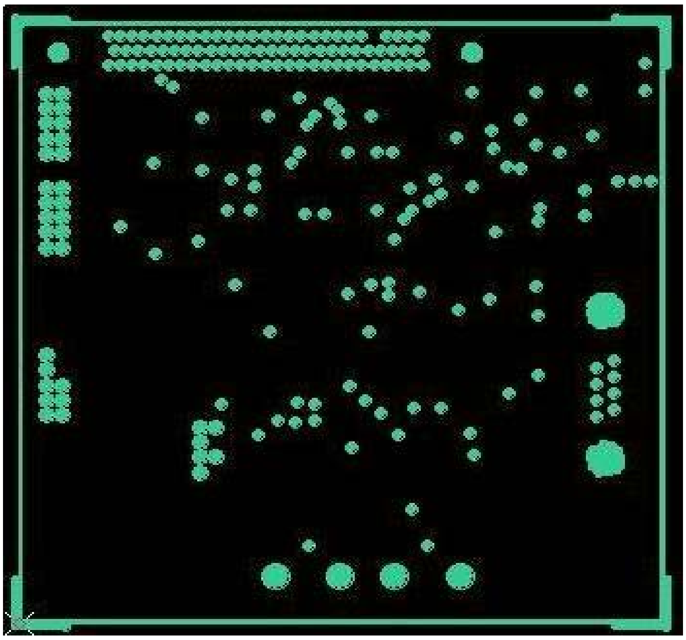


Figure B.11 Fault Tolerant Systems Third Layer (Power Layer)

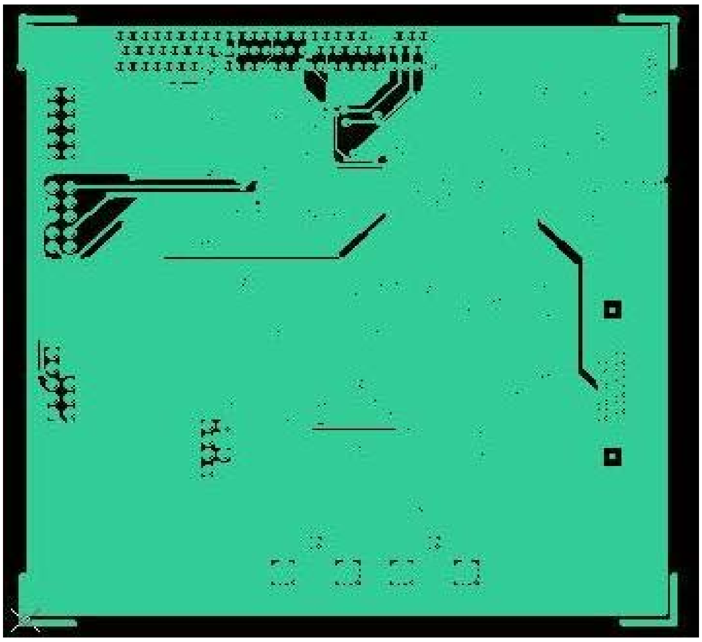


Figure B.12 Fault Tolerant Systems Fourth Layer

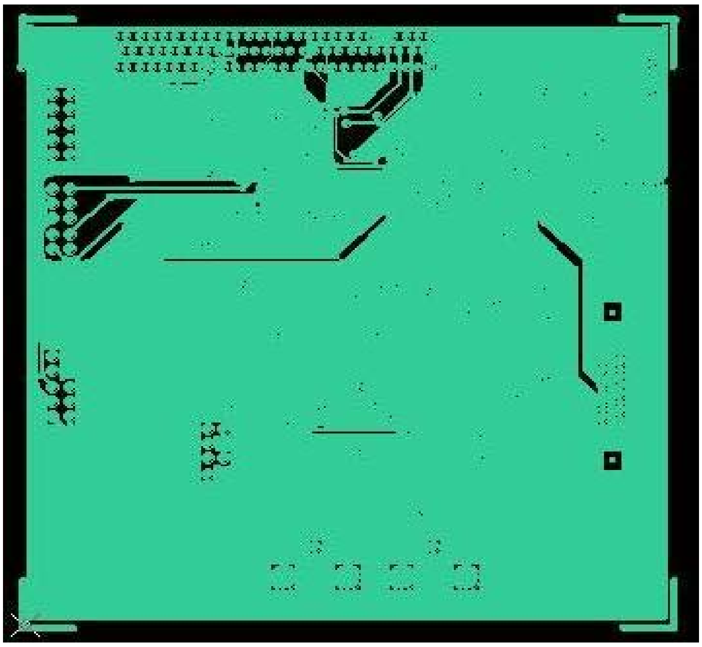


Figure B.13 Fault Tolerant Systems Fifth Layer

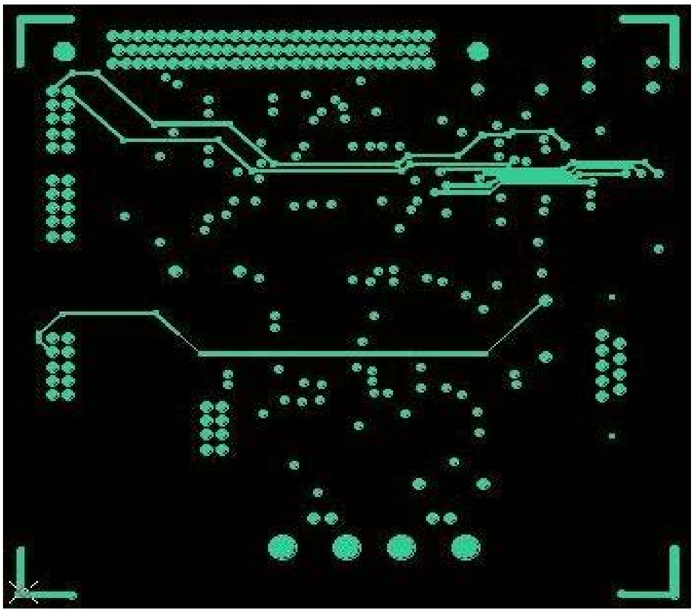


Figure B.14 Fault Tolerant Systems Sixth Layer

DISPLAY PCB LAYOUT

Display PCB is designed as a two layers Printed Circuit Board (PCB). All components are located on the upper side of the PCB. First layer is assigned for component placement and signal routing. Second layer is assigned for a signal routing. This PCB does not has a ground and power planes. Ground and planes are implemented as a signal. These layers are shown in the following figures.

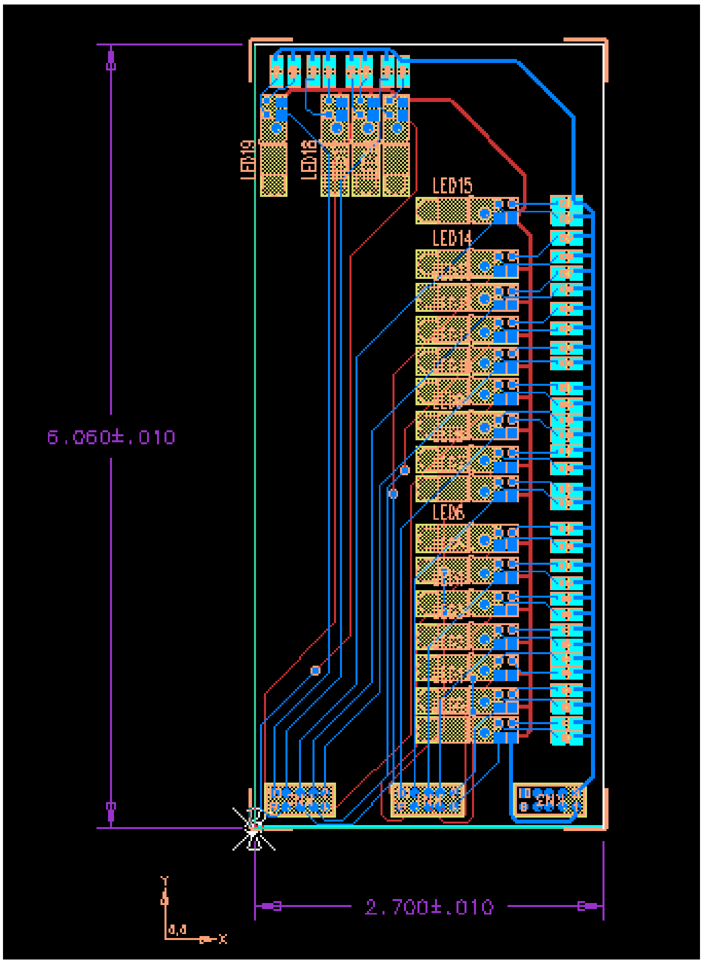


Figure B.15 Display PCB components, signal routings and dimensions

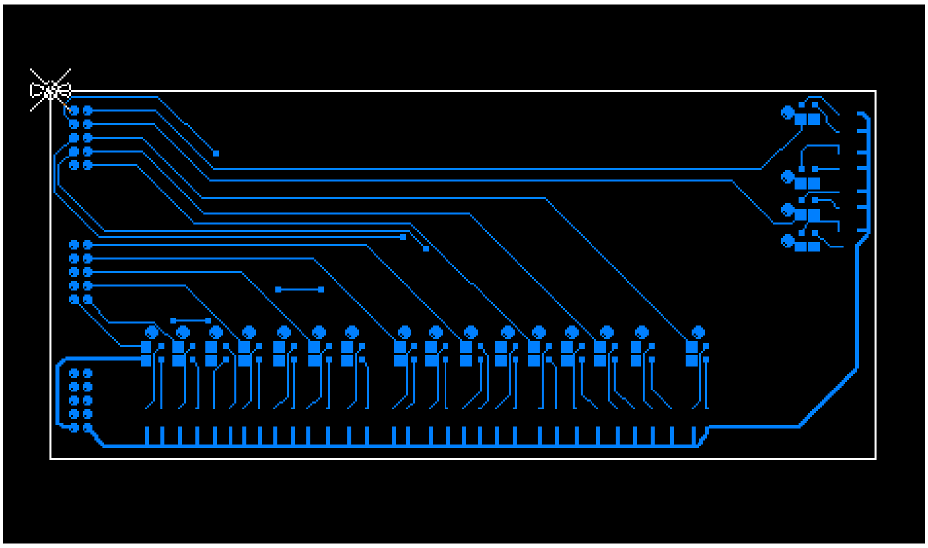


Figure B.16 Display PCB First Layer

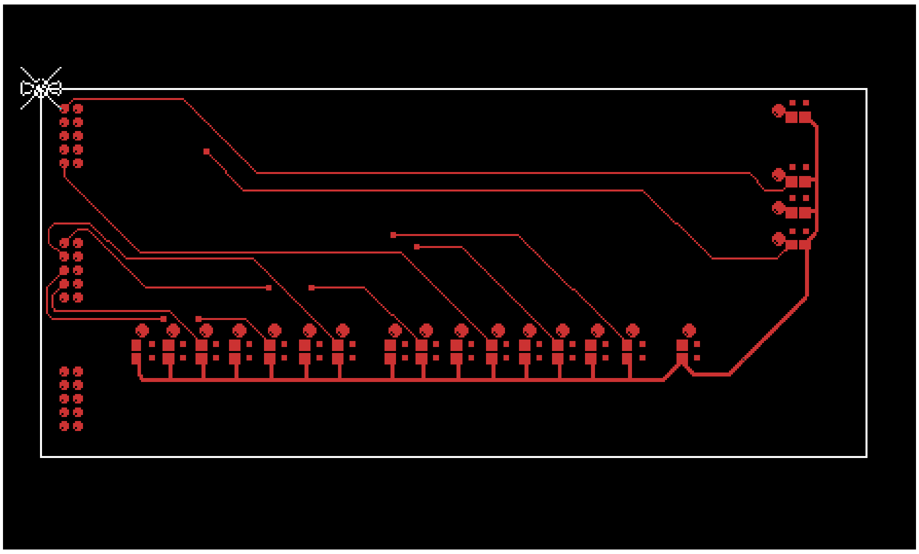


Figure B.17 Display PCB Second Layer

APPENDIX-C

TEST PATTERN GENERATOR VHDL CODES

RS232 Receiver Block VHDL Code

```
--*******************************************************************************
--********************************* METU / EE ***********************************
--*******************************************************************************
--**************************** WRITTEN BY UGUR GUNGOR ***************************
--*******************************************************************************
--********************************* METU / EE ***********************************
--*******************************************************************************
-- BLOCK NAME: RS232 alici blogu
-- ACIKLAMA  :
--******************************************************************************


LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;


ENTITY rs232_receiver IS

PORT(
      baudrate  : IN     std_logic_vector (7 DOWNTO 0);
      clk       : IN     std_logic;
      cs        : IN     std_logic;
      rst       : IN     std_logic;
      sin       : IN     std_logic;
      data_out  : OUT    std_logic_vector (7 DOWNTO 0);
      datavalid : OUT    std_logic;
```

```
      receiving : OUT    std_logic
   );
END  rs232_receiver ;

architecture arch_rs232_receiver of rs232_receiver is


constant wait_first : std_logic_vector(1 downto 0) := "00";
constant count_half : std_logic_vector(1 downto 0) := "01";
constant get_first  : std_logic_vector(1 downto 0) := "10";
constant get_last   : std_logic_vector(1 downto 0) := "11";

signal   counter    : std_logic_vector(7 downto 0);
signal   count_8    : std_logic_vector(2 downto 0);
signal   rcv_data   : std_logic_vector(7 downto 0);
signal   data_flag  : std_logic;
signal   rcv_state  : std_logic_vector(1 downto 0);

begin

  process(clk, rst)

  begin

    if rst = '0' then

       rcv_state            <= wait_first;
       counter              <= (others => '0');
       count_8              <= (others => '0');
       rcv_data             <= (others => '0');
       data_flag            <= '1';
       receiving            <= '0';

    elsif rising_edge(clk) then

      if cs = '0' then
        rcv_state          <= wait_first;
        counter            <= (others => '0');
        count_8            <= (others => '0');
        receiving          <= '0';
        data_flag          <= '1';
      else

        case rcv_state is

        when wait_first =>
```

```
   counter         <= (others => '0');
   data_flag       <= '1';
   if sin = '1' then
      rcv_state    <= wait_first;
   else
      rcv_state    <= count_half;
   end if;

when count_half            =>

   counter         <= counter+1;

   if sin = '0' then
     if counter     = ('0'&baudrate(7 downto 1)) then
        counter    <= (others => '0');
        rcv_state  <= get_first;
        receiving  <= '1';
     else
        rcv_state  <= count_half;
     end if;

   else
     rcv_state     <= wait_first;
   end if;

when get_first             =>

   counter         <= counter+1;

   if counter       = baudrate then
      counter      <= (others => '0');
      rcv_data     <= sin&rcv_data(7 downto 1);
      count_8      <= count_8+1;

      if count_8 = "111" then
         rcv_state <= get_last;
     end if;
   end if;

when get_last      =>
   counter         <= counter+1;

   if counter = baudrate then
      counter      <= (others => '0');
      receiving    <= '0';
      if sin='1' then
         data_flag <= '0';
```

```
               end if;
               rcv_state    <=wait_first;
            end if;

         when others       =>
            null;

        end case;

      end if;

    end if;

  end process;

  process(clk,rst)
  begin

    if rst='0' then

       data_out       <=(others=>'0');
       datavalid      <= '1';

    elsif rising_edge(clk) then

      datavalid       <= '1';

      if data_flag='0' then
         data_out     <= rcv_data;
         datavalid    <= '0';
      end if;
    end if;

  end process;


END ;
--**********************************************************  end of architecture
--*******************************************************************************
--********************************* METU / EE  **********************************
--*******************************************************************************
--**************************** WRITTEN BY UGUR GUNGOR ***************************
--*******************************************************************************
--********************************* METU / EE ***********************************
--*******************************************************************************
```

RS232 Transmitter Block VHDL Code

```
--*******************************************************************************
--********************************* METU / EE ***********************************
--*******************************************************************************
--**************************** WRITTEN BY UGUR GUNGOR ***************************
--*******************************************************************************
--********************************* METU / EE ***********************************
--*******************************************************************************
-- BLOCK NAME: RS232 gonderici blogu
-- ACIKLAMA  : .
--*******************************************************************************

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY rs232_transmitter IS

PORT(
      baudrate     : IN     std_logic_vector (7 DOWNTO 0);
      clk          : IN     std_logic;
      cs           : IN     std_logic;
      data_in      : IN     std_logic_vector (7 DOWNTO 0);
      rst          : IN     std_logic;
      start_xmit   : IN     std_logic;
      sout         : OUT    std_logic;
      transmitting : OUT    std_logic
   );

END  rs232_transmitter ;



architecture  arch_rs232_transmitter of rs232_transmitter is

signal xmitdt        : std_logic_vector(7 downto 0);
signal xmit_stm      : std_logic_vector(1 downto 0);
signal cnt1          : std_logic_vector(7 downto 0);
signal cnt_xmit_stop : std_logic_vector(2 downto 0);

constant WAITSTATE   : std_logic_vector(1 downto 0) := "00";
constant STARTSTATE  : std_logic_vector(1 downto 0) := "01";
constant DATASTATE   : std_logic_vector(1 downto 0) := "10";
```

```
constant STOPSTATE   : std_logic_vector(1 downto 0) := "11";
begin

  process(clk, rst)
  begin

    if rst = '0' then
      xmit_stm         <= WAITSTATE;
      sout             <= '1';
      cnt1             <= (others => '0');
      cnt_xmit_stop    <= (others => '0');
      xmitdt           <= (others => '0');
      transmitting     <= '0';

    elsif falling_edge(clk) then

      if cs = '0' then
         xmit_stm      <= WAITSTATE;
         transmitting  <= '0';
         sout          <= '1';
         cnt1          <= (others => '0');
         cnt_xmit_stop <= (others => '0');
      else

        case xmit_stm is

          --ready to transmit data 1 start,1 stop bit
          when WAITSTATE =>

            if start_xmit = '1' then
               xmit_stm <= STARTSTATE;
               xmitdt   <= data_in;
            end if;

          when STARTSTATE           =>

            transmitting <= '1';
            sout         <= '0';
            cnt1         <= cnt1+1;

            if cnt1 = baudrate then
               cnt1      <= (others => '0');
               xmit_stm  <= DATASTATE;
            end if;

          when DATASTATE                 =>
```

```
            sout               <= xmitdt(0);
            cnt1               <= cnt1+1;

            if cnt1 = baudrate then
              cnt_xmit_stop    <= cnt_xmit_stop+1;
              xmitdt           <= '0'&xmitdt(7 downto 1);

              if cnt_xmit_stop = "111" then
                 sout          <= '1';
                 cnt_xmit_stop <= (others => '0');
                 xmit_stm      <= STOPSTATE;
              end if;
              cnt1            <= (others => '0');
            end if;

          when STOPSTATE =>

            cnt1 <=cnt1+1;

            if cnt1=baudrate then
               cnt1<=(others=>'0');
               xmit_stm<=WAITSTATE;
               transmitting<='0';
            end if;

          when others=>
            null;

        end case;

      end if;

    end if;

  end process;

END ;

--********************************************************************************
--********************************* METU / EE ************************************
--********************************************************************************
--**************************** WRITTEN BY UGUR GUNGOR ****************************
--********************************************************************************
--********************************* METU / EE ************************************
--********************************************************************************
```

RS232 Interface Block VHDL Code

```
--*******************************************************************************
--********************************* METU / EE ***********************************
--*******************************************************************************
--**************************** WRITTEN BY UGUR GUNGOR ***************************
--*******************************************************************************
--********************************* METU / EE ***********************************
--*******************************************************************************
-- BLOCK NAME: RS232 arayuz blogu
-- ACIKLAMA  :
--******************************************************************************

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;


ENTITY  rs232_arayuz IS

PORT
  (
    rst             : IN     std_logic;
    clk             : IN     std_logic;
    rs232_in        : IN     std_logic;
    rs232_out       : OUT    std_logic;
    alinan_data     : OUT    std_logic_vector (7 DOWNTO 0);
    gonderilen_data : IN     std_logic_vector (7 DOWNTO 0);
    mult_input      : out    std_logic_vector (14 DOWNTO 0)
  );
END rs232_arayuz ;

architecture arch_rs232_arayuz of rs232_arayuz is



COMPONENT rs232_receiver
   PORT(
      baudrate     : IN     std_logic_vector (7 DOWNTO 0);
      clk          : IN     std_logic;
      cs           : IN     std_logic;
      rst          : IN     std_logic;
      sin          : IN     std_logic;
      data_out     : OUT    std_logic_vector (7 DOWNTO 0);
      datavalid    : OUT    std_logic;
```

```
      receiving    : OUT    std_logic
   );
END  COMPONENT ;

COMPONENT rs232_transmitter
   PORT(
      baudrate     : IN     std_logic_vector (7 DOWNTO 0);
      clk          : IN     std_logic;
      cs           : IN     std_logic;
      data_in      : IN     std_logic_vector (7 DOWNTO 0);
      rst          : IN     std_logic;
      start_xmit   : IN     std_logic;
      sout         : OUT    std_logic;
      transmitting : OUT    std_logic
   );

 END COMPONENT ;

signal   baudrate        :    std_logic_vector (7 DOWNTO 0);
signal   cs              :    std_logic;
signal   data_out        :    std_logic_vector (7 DOWNTO 0);
signal   datavalid       :    std_logic;
signal   receiving       :    std_logic;
signal   data_in         :    std_logic_vector (7 DOWNTO 0);
signal   start_xmit      :    std_logic;
signal   transmitting    :    std_logic;

signal   state_a         :    std_logic_vector (5 DOWNTO 0);
signal   valid_buf       :    std_logic;
signal   count_a         :    std_logic_vector (9 DOWNTO 0);
signal   alinan_data_mi  :    std_logic_vector (7 DOWNTO 0);
signal   data_check      :    std_logic_vector (7 DOWNTO 0);
signal   ugur            :    std_logic_vector (7 DOWNTO 0);
signal   FP_input_a      :    std_logic_vector (7 DOWNTO 0);
signal   FP_input_b      :    std_logic_vector (7 DOWNTO 0);


BEGIN

baudrate <= "00100000";  -- 9600 baudrate if Tclk= 1,641 us;
cs       <= '1' ;


 process(clk, rst)
  begin
    if rst = '0' then
```

```
   state_a        <= (others => '0');
   valid_buf      <= '1';
   start_xmit     <= '0';
   alinan_data    <= (others => '0');
   alinan_data_mi <= (others => '0');
   count_a        <= (others => '0');
   data_check     <= (others => '0');
   ugur           <= (others => '0');
   FP_input_a     <= (others => '0');  FP_input_a(6) <= '1';
   FP_input_b     <= (others => '0');  FP_input_b(0) <= '1';
   mult_input     <= "010000000000001";

elsif falling_edge(clk) then

   valid_buf      <= datavalid ;
   case state_a is

        when "000000" =>
             if (datavalid ='1' ) and (valid_buf ='0' ) then
                if (data_out = "10101010")  then     --AA
                    state_a  <= state_a  + '1';
                end if;
             end if;
            if (datavalid ='1' ) and (valid_buf ='0' ) then
                if (data_out = "01010101")  then     --55
                    state_a  <= "100000";
                end if;
             end if;
        when "000001" =>

             if (datavalid ='1' ) and (valid_buf ='0' ) then
                alinan_data_mi <= data_out;
                state_a  <= state_a  + '1';

             end if;

        when "000010" =>
              alinan_data  <=  alinan_data_mi ;
              data_in      <= "10111011";  --BB
              state_a      <= state_a  + '1';

        when "000011" =>
              start_xmit   <= '1';
              state_a      <= state_a  + '1';

        when "000100" =>
              start_xmit   <= '0';
```

```
      state_a      <= state_a  + '1';

when "000101" =>
      state_a      <= state_a  + '1';

when "000110" =>
     if transmitting = '0' then
        data_in    <= gonderilen_data;
        ugur       <= gonderilen_data;
        state_a    <= "011000";
        count_a    <= (others => '0');

     end if;

when "011000" =>
      state_a      <= state_a  + '1';

when "011001" =>
      start_xmit   <= '1';
      state_a      <= state_a  + '1';

when "011010" =>
      start_xmit   <= '0';
      state_a      <= (others => '0');


when "100000" =>
    if (datavalid ='1' ) and (valid_buf ='0' ) then
        FP_input_a   <= data_out;
        state_a  <= state_a  + '1';
    end if;
when "100001" =>
    if (datavalid ='1' ) and (valid_buf ='0' ) then
        FP_input_b   <= data_out;
        state_a  <= state_a  + '1';
    end if;
when "100010" =>

    mult_input     <=    FP_input_a   & FP_input_b (6 downto 0);
    state_a  <= state_a  + '1';

when "100011" =>

       state_a    <= (others => '0');

when others=>
     null;
```

```
            end case;n
     end if;
 end process;


U1: rs232_receiver  PORT MAP
    (
      baudrate    => baudrate  ,
      clk         => clk       ,
      cs          => cs        ,
      rst         => rst       ,
      sin         => rs232_in  ,
      data_out    => data_out  ,
      datavalid   => datavalid ,
      receiving   => receiving
    );


U2: rs232_transmitter PORT MAP
    (
      baudrate     => baudrate  ,
      clk          => clk       ,
      cs           => cs        ,
      data_in      => data_in   ,
      rst          => rst       ,
      start_xmit   => start_xmit,
      sout         => rs232_out ,
      transmitting => transmitting
    );

END ;
--*******************************************************************************
--********************************* METU / EE ***********************************
--*******************************************************************************
--**************************** WRITTEN BY UGUR GUNGOR ***************************
--*******************************************************************************
--********************************* METU / EE ***********************************
--*******************************************************************************
```

Test Pattern Generator Block VHDL Code

```
--*******************************************************************************
--********************************* METU / EE ***********************************
--*******************************************************************************
--**************************** WRITTEN BY UGUR GUNGOR ***************************
--*******************************************************************************
--********************************* METU / EE ***********************************
--*******************************************************************************
-- BLOCK NAME: Test pattern generator
-- ACIKLAMA  : FPGA1 testini suren test pattern yaratilir ve sonuclar degerlendirilir.
--******************************************************************************

library ieee                    ;
use ieee.std_logic_unsigned.all ;
use ieee.std_logic_1164.all     ;

library UNISIM                  ;
use UNISIM.VCOMPONENTS.all      ;



entity test_pattern_generator  is

port (
        rst                      :  IN   std_logic ;   -- rst signal, initalize all
registers.
        clk_in                   :  IN   std_logic ;   -- 39MHz input clk
        rs232_in                 :  IN   std_logic ;   -- RS232 port data tranmit
        rs232_out                :  OUT  std_logic ;   -- RS232 port data receive
        insert_module_fault_1    :  OUT  std_logic ;   -- insert one fault to the TMR
circuit
        insert_module_failure_1  :  OUT  std_logic ;   -- insert two fault to the TMR
circuit
        insert_voter_fault_1     :  OUT  std_logic ;   -- insert voter fault to the TMR
circuit
        insert_module_fault_2    :  OUT  std_logic ;   -- insert one fault to the TMR
with feedback circuit
        insert_module_failure_2  :  OUT  std_logic ;   -- insert two fault to the TMR
with feedback circuit
        insert_voter_fault_2     :  OUT  std_logic ;   -- insert voter fault to the TMR
with feedback circuit
        failure_detection_1      :  IN   std_logic ;   -- fault detection signal in TMR
circuit
```

```
        failure_detection_2      :  IN   std_logic ;   -- fault detection signal in TMR
with feedback circuit
        operation_start          :  OUT  std_logic ;   -- start multiplication without
fault injection
        clk_out                  :  OUT  std_logic ;   -- operational clock of fault
tolerant system implementation fpga
        fault_module_a           :  IN   std_logic ;   -- fault injected oodule
specifier_1
        fault_module_b           :  IN   std_logic ;   -- fault injected oodule
specifier_2
      --fault_module_c           :  IN   std_logic ;   -- fault injected oodule
specifier_3
        mult_input               :  OUT  std_logic_vector(14 DOWNTO 0)  --fp multiplier
input
    );
end test_pattern_generator;

architecture arch_test_pattern_generator of test_pattern_generator is


COMPONENT BUFG  --- clock buffer

port
    (
       i : in  std_logic;
       o : out std_logic

    );
end COMPONENT;

COMPONENT  rs232_arayuz  --used to communicate with rs232 port of the computer at 9600
baudrate

PORT
    (
      rst             : IN     std_logic;
      clk             : IN     std_logic;
      rs232_in        : IN     std_logic;
      rs232_out       : OUT    std_logic;
      alinan_data     : OUT    std_logic_vector (7 DOWNTO 0);
      gonderilen_data : IN     std_logic_vector (7 DOWNTO 0);
      mult_input      : out    std_logic_vector (14 DOWNTO 0)

    );
END COMPONENT ;

-- signal declerations
```

```
signal clk_divider                : std_logic_vector (6 downto 0);
signal alinan_data                : std_logic_vector (7 DOWNTO 0);
signal gonderilen_data            : std_logic_vector (7 DOWNTO 0);
signal clk                        : std_logic;
signal response_time              : std_logic_vector (4 DOWNTO 0);
signal insert_module_fault_1_a    : std_logic;
signal insert_module_failure_1_a  : std_logic;
signal insert_voter_fault_1_a     : std_logic;
signal insert_module_fault_2_a    : std_logic;
signal insert_module_failure_2_a  : std_logic;
signal insert_voter_fault_2_a     : std_logic;
signal insert_module_fault_1_b    : std_logic;
signal insert_module_failure_1_b  : std_logic;
signal insert_voter_fault_1_b     : std_logic;
signal insert_module_fault_2_b    : std_logic;
signal insert_module_failure_2_b  : std_logic;
signal insert_voter_fault_2_b     : std_logic;
signal insert_module_fault_1_c    : std_logic;
signal insert_module_failure_1_c  : std_logic;
signal insert_voter_fault_1_c     : std_logic;
signal insert_module_fault_2_c    : std_logic;
signal insert_module_failure_2_c  : std_logic;
signal insert_voter_fault_2_c     : std_logic;
signal operation_start_a          : std_logic;
signal operation_start_b          : std_logic;
signal operation_start_c          : std_logic;
signal send_response              : std_logic;
signal send_response_buf          : std_logic;
--signal mult_input                 : std_logic;

BEGIN

process(clk_in, rst)
 begin
   if rst = '0' then

      insert_module_fault_1_a    <= '0'    ;
      insert_module_failure_1_a  <= '0'    ;
      insert_voter_fault_1_a     <= '0'    ;
      insert_module_fault_2_a    <= '0'    ;
      insert_module_failure_2_a  <= '0'    ;
      insert_voter_fault_2_a     <= '0'    ;
      insert_module_fault_1_b    <= '0'    ;
      insert_module_failure_1_b  <= '0'    ;
      insert_voter_fault_1_b     <= '0'    ;
      insert_module_fault_2_b    <= '0'    ;
      insert_module_failure_2_b  <= '0'    ;
```

```
   insert_voter_fault_2_b     <= '0'    ;
   insert_module_fault_1_c    <= '0'    ;
   insert_module_failure_1_c  <= '0'    ;
   insert_voter_fault_1_c     <= '0'    ;
   insert_module_fault_2_c    <= '0'    ;
   insert_module_failure_2_c  <= '0'    ;
   insert_voter_fault_2_c     <= '0'    ;
   operation_start_a          <= '0'    ;
   operation_start_b          <= '0'    ;
   operation_start_c          <= '0'    ;
   send_response              <= '0'    ;
   send_response_buf          <= '0'    ;
   response_time              <= "00000";

elsif falling_edge(clk) then

   insert_module_fault_1_a    <=  alinan_data (0)           ;
   insert_module_failure_1_a  <=  alinan_data (1)           ;
   insert_voter_fault_1_a     <=  alinan_data (2)           ;
   insert_module_fault_2_a    <=  alinan_data (3)           ;
   insert_module_failure_2_a  <=  alinan_data (4)           ;
   insert_voter_fault_2_a     <=  alinan_data (5)           ;
   operation_start_a          <=  alinan_data (6)           ;

   insert_module_fault_1_b    <=  insert_module_fault_1_a   ;
   insert_module_failure_1_b  <=  insert_module_failure_1_a ;
   insert_voter_fault_1_b     <=  insert_voter_fault_1_a    ;
   insert_module_fault_2_b    <=  insert_module_fault_2_a   ;
   insert_module_failure_2_b  <=  insert_module_failure_2_a ;
   insert_voter_fault_2_b     <=  insert_voter_fault_2_a    ;
   operation_start_b          <=  operation_start_a         ;

   insert_module_fault_1_c    <=  insert_module_fault_1_b   ;
   insert_module_failure_1_c  <=  insert_module_failure_1_b ;
   insert_voter_fault_1_c     <=  insert_voter_fault_1_b    ;
   insert_module_fault_2_c    <=  insert_module_fault_2_b   ;
   insert_module_failure_2_c  <=  insert_module_failure_2_b ;
   insert_voter_fault_2_c     <=  insert_voter_fault_2_b    ;
   operation_start_c          <=  operation_start_b         ;

   insert_module_fault_1      <=  insert_module_fault_1_c   ;
   insert_module_failure_1    <=  insert_module_failure_1_c ;
   insert_voter_fault_1       <=  insert_voter_fault_1_c    ;
   insert_module_fault_2      <=  insert_module_fault_2_c   ;
   insert_module_failure_2    <=  insert_module_failure_2_c ;
   insert_voter_fault_2       <=  insert_voter_fault_2_c    ;
-- operation_start            <=  operation_start_c         ;
```

```
      send_response_buf           <=  send_response             ;
      operation_start             <=  send_response_buf         ;

    if ((insert_module_fault_1_c = '0' )   and  (insert_module_fault_1_b = '1' ))  then
--rising
         send_response <= '1';
    end if;

    if ((insert_module_fault_2_c = '0' )   and  (insert_module_fault_2_b = '1' ))  then
--rising
         send_response <= '1';
    end if;

    if ((insert_module_failure_1_c = '0' ) and  (insert_module_failure_1_b = '1' ))then
--rising
         send_response <= '1';
    end if;

    if ((insert_module_failure_2_c = '0' ) and  (insert_module_failure_2_b = '1' ))then
--rising
         send_response <= '1';
    end if;

    if ((insert_voter_fault_1_c = '0' )    and  (insert_voter_fault_1_b = '1' ))   then
--rising
         send_response <= '1';
    end if;

    if ((insert_voter_fault_2_c = '0' )    and  (insert_voter_fault_2_b = '1' ))   then
--rising
         send_response <= '1';
    end if;

    if ((operation_start_c = '0' )         and  ( operation_start_b = '1' ))       then
--rising
         send_response <= '1';
    end if;


    if ( send_response = '1' ) then
       response_time <=   response_time + '1';
       if (response_time  >= "01100" )then
          send_response <= '0';
          response_time  <= "00000";
          -- gonderilen_data
          -- gonderilen_data test sonuclarini bilgisayara gondermekte kullanilir.
          gonderilen_data(0)          <=  not (failure_detection_1 );
```

```
          gonderilen_data(1)          <=  not (failure_detection_2 );
          gonderilen_data(2)          <=  fault_module_a ;
          gonderilen_data(3)          <=  fault_module_b ;
       -- gonderilen_data(4)          <=  fault_module_c ;
          gonderilen_data(7 downto 4) <=  "0000";
      end if;
    end if;

 end if;
 end process;

 clk_out            <= clk;
-- multiplier_input_a <= mult_input     ;


 process(clk_in, rst)

 begin
   if rst = '0' then
       clk_divider <= "0000000";
   elsif falling_edge(clk_in) then
       clk_divider <=  clk_divider +'1' ;
   end if;
 end process;

U3: BUFG   port map   --clk buffer instantiation
    (
     i => clk_divider(6),
     o => clk
    );

U2: rs232_arayuz PORT MAP   -- rs232 arayuz instantiation
   (
    rst             => rst            ,
    clk             => clk            ,
    rs232_in        => rs232_in       ,
    rs232_out       => rs232_out      ,
    alinan_data     => alinan_data    ,
    gonderilen_data => gonderilen_data,
    mult_input      => mult_input
   );

END;

--*******************************************************************************
--******************************** METU / EE *********************************
--*******************************************************************************
```

```
--**************************** WRITTEN BY UGUR GUNGOR ***************************
--*******************************************************************************
--********************************* METU / EE ***********************************
--*******************************************************************************
```

APPENDIX-D

FAULT TOLERANT SYSTEMS VHDL CODES

Floating Point Multiplier Block VHDL Code

```
--********************************************************************************
--********************************* METU / EE ************************************
--********************************************************************************
--***************************** WRITTEN BY UGUR GUNGOR ***************************
--********************************************************************************
--********************************* METU / EE ************************************
--********************************************************************************
--
-- BLOCK NAME: Floating Point Multiplier
-- ACIKLAMA  : Iki tane 15 bit floating point sayiyi alip bunlari carpar.
--             Sonucu normalize ederek yine 16 bit verir.
--
--********************************************************************************
--
-- 16 bit fp number =>
--                     bit 14    ->  sign bit
--                                           1 ise negatif
--                                           0 ise pozitif
--                     bit 13:7  ->  mantissa
--                                           unsigned representation
--                     bit 6:0   ->  exponent
--                                           signed representation
--
--********************************************************************************

library ieee                    ;
use ieee.std_logic_unsigned.all ;
use ieee.std_logic_1164.all     ;
use ieee.numeric_std.all        ;

library UNISIM                  ;
```

```
use UNISIM.VCOMPONENTS.all      ;

--*****************************************************************************
-- entity decleration
-- bu kisimda giris cikis sinyalleri gosterilmistir.

ENTITY FP_multiplier IS

 port (
        rst                    : in   std_logic ;
        clk                    : in   std_logic ;
        FP_input_A             : in   std_logic_vector (14 downto 0) ;
        FP_input_B             : in   std_logic_vector (14 downto 0) ;
        FP_output              : out  std_logic_vector (14 downto 0) ;
        overflow               : out  std_logic ;
        underflow              : out  std_logic ;
        operation_start        : in   std_logic
      );
END FP_multiplier;

-- end of entity decleration
--*****************************************************************************

-- architecture, kodun implemente edildigi kisim

ARCHITECTURE  Arch_FP_multiplier of FP_multiplier is


-- signal assignments (kod icinde kullanilan sinyaller bu kisimda tanimlanir.)

signal  FP_input_A_sign        : std_logic ;
signal  FP_input_A_exponent    : signed   (6 downto 0) ;
signal  FP_input_A_mantissa    : unsigned (6 downto 0) ;

signal  FP_input_B_sign        : std_logic ;
signal  FP_input_B_exponent    : signed   (6 downto 0) ;
signal  FP_input_B_mantissa    : unsigned (6 downto 0) ;

signal  FP_input_A_exponent_EX : signed   (8 downto 0) ;
signal  FP_input_B_exponent_EX : signed   (8 downto 0) ;
signal  Exponent_addition      : signed   (8 downto 0) ;

signal  multiplier_sign        : std_logic             ;
signal  multiplier_sign_a      : std_logic             ;
signal  multiplier_sign_b      : std_logic             ;

signal  FP_input_A_mantissa_b  : unsigned (6 downto 0) ;
```

```
signal  FP_input_B_mantissa_b  : unsigned (6 downto 0) ;
signal  Mantissa_multip        : unsigned(13 downto 0) ;

signal  output_mantissa        : unsigned( 6 downto 0) ;
signal  output_exponent        : signed   ( 8 downto 0) ;
signal  output_mantissa_b      : unsigned( 6 downto 0) ;
signal  output_exponent_b      : signed   ( 8 downto 0) ;

signal  operation_start_buf    : std_logic             ;
signal  operation_start_buf_b  : std_logic             ;
signal  state_initial          : std_logic_vector( 6 downto 0);
signal  utu                    : std_logic             ;


--*****************************************************************************
BEGIN  -- begin operations



-- asenkron atamalarin yapilmasi.

FP_input_A_sign      <= FP_input_A(14)                    ;
FP_input_A_exponent  <= signed( FP_input_A(6 downto 0))    ;
FP_input_A_mantissa  <= unsigned( FP_input_A(13 downto 7)) ;

FP_input_B_sign      <= FP_input_B(14);
FP_input_B_exponent  <= signed( FP_input_B(6 downto 0))    ;
FP_input_B_mantissa  <= unsigned( FP_input_B(13 downto 7)) ;

process(rst,clk)

begin
  if rst = '0' then

        overflow                  <= '0';                -- NO OVERFLOW
        underflow                 <= '0';                -- NO UNDERFLOW
        FP_output                 <= "010000000000001";
        FP_input_A_exponent_EX    <= "000000001";
        FP_input_B_exponent_EX    <= "000000001";
        Exponent_addition         <= "000000010";
        multiplier_sign           <= '0';                -- POSITIVE
        multiplier_sign_a         <= '0';                -- POSITIVE
        multiplier_sign_b         <= '0';                -- POSITIVE
        FP_input_A_mantissa_b     <= "1000000" ;         -- VALUE1
        FP_input_B_mantissa_b     <= "1000000" ;
        Mantissa_multip           <= "10000001000000" ;
        output_mantissa           <= "1000000";
        output_exponent           <= "000000001";
```

```
        output_mantissa_b          <= "1000000";
        output_exponent_B          <= "000000001";
        FP_output                  <= "010000000000001";
        operation_start_buf        <= '0';
        operation_start_buf_b      <= '0';

  elsif rising_edge(clk) then   -- clk nin rising'i ile alinmisti.

       operation_start_buf_b      <= operation_start_buf ;
       operation_start_buf        <=  operation_start ;

       if (((operation_start_buf_b ='0') and (operation_start_buf = '1')) )then --rising
edge

           FP_input_A_exponent_EX  <= resize ( FP_input_A_exponent, 9);
           FP_input_B_exponent_EX  <= resize ( FP_input_B_exponent, 9);
           multiplier_sign         <= ( FP_input_A_sign xor  FP_input_B_sign) ;
           FP_input_A_mantissa_b   <= FP_input_A_mantissa;
           FP_input_B_mantissa_b   <= FP_input_B_mantissa;
           multiplier_sign_a       <= multiplier_sign    ;
           utu                     <= '0';

        end if;

        multiplier_sign_a       <= multiplier_sign   ;
        multiplier_sign_b       <= multiplier_sign_a ;
        Exponent_addition       <= ( FP_input_B_exponent_EX +   FP_input_A_exponent_EX );
        Mantissa_multip         <= ( FP_input_A_mantissa_b  *   FP_input_B_mantissa_b  );

        if  Mantissa_multip(13) = '1' then

            output_mantissa    <= Mantissa_multip (13 downto 7);
            output_exponent    <= Exponent_addition ;

        elsif Mantissa_multip(12) = '1' then

            output_mantissa   <= Mantissa_multip (12 downto 6);
            output_exponent   <= Exponent_addition - "001"    ;

        elsif Mantissa_multip(11) = '1' then

            output_mantissa   <= Mantissa_multip (11 downto 5);
            output_exponent   <= Exponent_addition  - "010"    ;

        elsif Mantissa_multip(10) = '1' then

            output_mantissa   <= Mantissa_multip (10 downto 4);
```

```
    output_exponent   <= Exponent_addition  - "011"    ;

elsif Mantissa_multip(9) = '1' then

    output_mantissa   <= Mantissa_multip (9 downto 3);
    output_exponent   <= Exponent_addition  - "0100"   ;

elsif Mantissa_multip(8) = '1' then

    output_mantissa   <= Mantissa_multip (8 downto 2);
    output_exponent   <= Exponent_addition - "0101"    ;

elsif Mantissa_multip(7) = '1' then

    output_mantissa   <= Mantissa_multip (7 downto 1);
    output_exponent   <= Exponent_addition - "0110"    ;

elsif Mantissa_multip(6) = '1' then

    output_mantissa   <= Mantissa_multip (6 downto 0);
    output_exponent   <= Exponent_addition - "0111"    ;

elsif  Mantissa_multip(5) = '1' then

     output_mantissa   <= Mantissa_multip (5 downto 0) & '0';
     output_exponent   <= Exponent_addition - "01000";

elsif Mantissa_multip(4) = '1' then

     output_mantissa   <= (  Mantissa_multip(4 downto 0)& "00" );
     output_exponent   <= Exponent_addition - "01001";

elsif Mantissa_multip(3) = '1' then

     output_mantissa   <= ( Mantissa_multip(3 downto 0) &"000" );
     output_exponent   <= Exponent_addition - "01010";

elsif Mantissa_multip(2) = '1' then

     output_mantissa   <=(  Mantissa_multip(2 downto 0)& "0000");
     output_exponent   <= Exponent_addition - "01011";

elsif Mantissa_multip(1) = '1' then

     output_mantissa   <=( Mantissa_multip(1 downto 0)&"00000" );
     output_exponent   <= Exponent_addition - "01100";
```

```
elsif Mantissa_multip(0) = '1' then

     output_mantissa   <= (  Mantissa_multip(0)&"000000" );
     output_exponent   <= Exponent_addition - "01101";

else

     output_mantissa   <=  "0000000";
     output_exponent   <=  "000000000";

end if;

output_mantissa_b      <= output_mantissa;
output_exponent_b      <= output_exponent;

if  (output_exponent < "111000000" ) then  --minimum

    output_mantissa_b  <=  "0000000";
    output_exponent_b  <=  "000000000";
    underflow          <=  '1';

else

    underflow          <=  '0';

end if;


if  (output_exponent > "000111111" ) then     --maximum

     output_mantissa_b  <=  "0000000";
     output_exponent_b  <=  "000000000";
     overflow           <=  '1';

 else
     overflow           <=  '0';
 end if;

 if  ( output_mantissa_b =  "0000000") then

     FP_output(14)      <= '0';

 else

     FP_output(14)      <= multiplier_sign_b  ;
 end if;
```

```
         FP_output(6 downto 0)  <= std_logic_vector( resize (output_exponent_b ,7)) ;
         FP_output(13 downto 7) <= std_logic_vector( output_mantissa_b)  ;

    end if;

end process;

END  Arch_FP_multiplier ;

--***************************** end of architecture *****************************
--*******************************************************************************
--********************************* METU / EE  **********************************
--*******************************************************************************
--**************************** WRITTEN BY UGUR GUNGOR ***************************
--*******************************************************************************
--********************************* METU / EE ***********************************
--*******************************************************************************
```

Floating Point Inner Feedback Block VHDL Code

```
--*******************************************************************************
--********************************* METU / EE  **********************************
--*******************************************************************************
--**************************** WRITTEN BY UGUR GUNGOR ***************************
--*******************************************************************************
--********************************* METU / EE ***********************************
--*******************************************************************************
-- BLOCK NAME: FP_multiplier_with_feedback
-- ACIKLAMA  : Carpma sonucunda cikan sayilardan birisi
--               carpma bloguna input_b olarak yeniden girer.
--*******************************************************************************

library ieee                    ;
use ieee.std_logic_unsigned.all ;
use ieee.std_logic_1164.all     ;
use ieee.numeric_std.all        ;

library UNISIM                  ;
use UNISIM.VCOMPONENTS.all      ;

entity  FP_multiplier_with_feedback is
```

```
port (
        rst                    : in   std_logic ;
        clk                    : in   std_logic ;
        FP_input_A             : in   std_logic_vector (14 downto 0) ;
        FP_output              : out  std_logic_vector (14 downto 0) ;
        overflow               : out  std_logic ;
        underflow              : out  std_logic ;
        operation_start        : in   std_logic
      );

end;

architecture arch_FP_multiplier_with_feedback of  FP_multiplier_with_feedback  is

COMPONENT FP_multiplier IS
 port (
        rst                    : in   std_logic ;
        clk                    : in   std_logic ;
        FP_input_A             : in   std_logic_vector (14 downto 0) ;
        FP_input_B             : in   std_logic_vector (14 downto 0) ;
        FP_output              : out  std_logic_vector (14 downto 0) ;
        overflow               : out  std_logic ;
        underflow              : out  std_logic ;
        operation_start        : in   std_logic
      );
END COMPONENT ;

signal FP_output_a             : std_logic_vector (14 downto 0) ;
signal FP_output_b             : std_logic_vector (14 downto 0) ;
signal overflow_b              : std_logic ;
signal underflow_b             : std_logic ;

begin

overflow  <= overflow_b  ;
underflow <= underflow_b ;
FP_output <= FP_output_a ;

process (rst, clk )
begin
if rst = '0' then
     FP_output_b  <=  "010000000000001";
elsif rising_edge(clk) then
    if ((overflow_b ='1')or (underflow_b ='1' ))then
     FP_output_b <= "010000000000001";
    else
       FP_output_b <= FP_output_a ;
```

```
    end if;
end if;
end process;

-- FP_output_b <= FP_output_a + '1' when  ;

 u1 : FP_multiplier
      port map (
                rst              => rst         ,
                clk              => clk         ,
                FP_input_A       => FP_input_A  ,
                FP_input_B       => FP_output_b ,
                FP_output        => FP_output_a ,
                overflow         => overflow_b  ,
                underflow        => underflow_b  ,
                operation_start  => operation_start
               );
end ;

--*******************************************************************************
--********************************* METU / EE  **********************************
--*******************************************************************************
--**************************** WRITTEN BY UGUR GUNGOR ***************************
--*******************************************************************************
--********************************* METU / EE ***********************************
--*******************************************************************************
```

Triple Modular Redundancy without Correction Block VHDL Code

```
--*******************************************************************************
--********************************* METU / EE  **********************************
--*******************************************************************************
--**************************** WRITTEN BY UGUR GUNGOR ***************************
--*******************************************************************************
--********************************* METU / EE ***********************************
--*******************************************************************************
-- BLOCK NAME:  FP_multiplier_simple_TMR
-- ACIKLAMA  :  Triple moduler redundancy  without correction
--*******************************************************************************
```

```
library ieee                    ;
use ieee.std_logic_unsigned.all ;
use ieee.std_logic_1164.all     ;

library UNISIM                  ;
use UNISIM.VCOMPONENTS.all      ;

--*****************************************************************************
-- entity decleration
-- bu kisimda giris cikis sinyalleri gosterilmistir.

entity  FP_multiplier_simple_TMR is

port (

     rst                         :  in   std_logic ;
     clk_in                      :  in   std_logic ;
     FP_input_A                  :  in   std_logic_vector (14 downto 0) ;
     failure_detection           :  out  std_logic ;
     FP_output                   :  out  std_logic_vector (14 downto 0) ;
     insert_module_fault         :  in   std_logic ;
     insert_module_failure       :  in   std_logic ;
     insert_voter_fault          :  in   std_logic ;
     operation_start             :  in   std_logic ;
     fault_module_a              :  out  std_logic ;
     fault_module_b              :  out  std_logic ;
     failure_detection_display   :  out  std_logic

);

end ;

-- end of entity decleration
--*****************************************************************************

architecture  FP_multiplier_with_feedback_arch of   FP_multiplier_simple_TMR  is

-- component assignments (kodun kullandigi alt moduller bu kisimda implemente edilir.)

component BUFG  --- clock buffer

port (

       i : in  std_logic;
       o : out std_logic

     );
```

```
end component;

component FP_multiplier_with_feedback IS

port (
       rst                    :  in   std_logic ;
       clk                    :  in   std_logic ;
       FP_input_A             :  in   std_logic_vector (14 downto 0) ;
       FP_output              :  out  std_logic_vector (14 downto 0) ;
       overflow               :  out  std_logic ;
       underflow              :  out  std_logic ;
       operation_start        :  in   std_logic

      );

end component ;

-- signal assignments (kod icinde kullanilan sinyaller bu kisimda tanimlanir.)

signal  clk                    :   std_logic ;

signal  FP_output_module1      :   std_logic_vector (14 downto 0) ;
signal  overflow_module1       :   std_logic ;
signal  underflow_module1      :   std_logic ;

signal  FP_output_module2      :   std_logic_vector (14 downto 0) ;
signal  overflow_module2       :   std_logic ;
signal  underflow_module2      :   std_logic ;

signal  fault_detection        :   std_logic ;

signal  FP_output_module3      :   std_logic_vector (14 downto 0) ;
signal  overflow_module3       :   std_logic ;
signal  underflow_module3      :   std_logic ;

signal input1                  :   std_logic_vector (14 downto 0) ;
signal input2                  :   std_logic_vector (14 downto 0) ;
signal input3                  :   std_logic_vector (14 downto 0) ;
signal operation_start_buf     :   std_logic ;

signal inputa                  :   std_logic_vector (14 downto 0) ;
signal inputb                  :   std_logic_vector (14 downto 0) ;
signal inputc                  :   std_logic_vector (14 downto 0) ;
signal counter_test            :   std_logic_vector (7 downto 0) ;

signal operation_start_buf_b   :   std_logic ;
signal operation_start_buf_c   :   std_logic ;
```

```
signal operation_start_buf_e   :   std_logic ;
signal operation_start_buf_d   :   std_logic ;

--*******************************************************************************
BEGIN -- begin operations

-- asenkron atamalarin yapilmasi.

failure_detection_display <= fault_detection ;
failure_detection         <=  fault_detection  ;
operation_start_buf       <= '1' when    (operation_start = '1') else '0';

inputc <=  "0111000000000001"& counter_test(1 downto 0)  when  (( insert_module_failure  =
'1')
           or ( insert_module_fault  = '1') )  else FP_input_A  ;

inputb <=  "0111110000000000"& counter_test(1 downto 0)   when ( insert_module_failure  =
'1')
           else FP_input_A  ;

inputa <=   FP_input_A ;


process(rst,clk)

begin

    if rst = '0' then

       FP_output                    <= "010000000000001";
       fault_detection              <= '0';

     elsif rising_edge(clk) then   -- clk nin rising'i ile alinmisti.

        if insert_voter_fault = '0' then

           fault_detection          <= '0';
           if  ( FP_output_module1 =  FP_output_module2 ) then

               FP_output            <= FP_output_module1;
           elsif( FP_output_module1 =  FP_output_module3 ) then

                FP_output           <= FP_output_module1;
           elsif ( FP_output_module2 =  FP_output_module3 ) then

               FP_output            <= FP_output_module3;
           else
```

```
              fault_detection      <= '1';
              FP_output            <= (others=>'0');

           end if;
        else
              fault_detection      <= '1';
        end if;

     end if;
end process;


process(rst,clk)

begin

 if rst = '0' then

    counter_test           <= (others=>'0');
    fault_module_a         <= '1';
    fault_module_b         <= '1';
    operation_start_buf_c  <= '0';
    operation_start_buf_b  <= '0';
    operation_start_buf_d  <= '0';
    operation_start_buf_e  <= '0';

 elsif rising_edge(clk) then   -- clk nin rising'i ile alinmisti.

    operation_start_buf_c  <= operation_start_buf_b ;
    operation_start_buf_b  <= operation_start_buf;
    operation_start_buf_d  <= operation_start_buf_c ;
    operation_start_buf_e  <= operation_start_buf_d ;

    if ( operation_start_buf_c = '1' ) and  ( operation_start_buf_b = '0' )  then

       counter_test       <=    counter_test   +'1';

    end if;

    case counter_test(1 downto 0)  is

    when "00"=>
             input1          <= inputa ;
             input2          <= inputb ;
             input3          <= inputc ;
             fault_module_a  <= '1';
```

```
                fault_module_b  <= '1';


    when "01"=>

                input1          <= inputb ;
                input2          <= inputc ;
                input3          <= inputa ;
                fault_module_a  <= '0';
                fault_module_b  <= '1';

    when "10" =>

                input1          <= inputc ;
                input2          <= inputa ;
                input3          <= inputb ;
                fault_module_a  <= '0';
                fault_module_b  <= '0';

    when "11"=>

                input1          <= inputa ;
                input2          <= inputb ;
                input3          <= inputc ;
                fault_module_a  <= '1';
                fault_module_b  <= '1';

    when others =>
                null;

    end case;
  end if;

end process;

u1 : FP_multiplier_with_feedback

      port map (
                rst             => rst                  ,
                clk             => clk                  ,
                FP_input_A      =>input1                ,
                FP_output       => FP_output_module1    ,
                overflow        => overflow_module1     ,
                underflow       => underflow_module1    ,
                operation_start => operation_start_buf_e

               );
```

```
 u2 : FP_multiplier_with_feedback

      port map (

                rst             => rst                  ,
                clk             => clk                  ,
                FP_input_A      => input2               ,
                FP_output       => FP_output_module2    ,
                overflow        => overflow_module2     ,
                underflow       => underflow_module2    ,
                operation_start => operation_start_buf_e

               );

 u3 :  FP_multiplier_with_feedback

      port map (

                rst             => rst                  ,
                clk             => clk                  ,
                FP_input_A      => input3               ,
                FP_output       => FP_output_module3    ,
                overflow        => overflow_module3     ,
                underflow       => underflow_module3    ,
                operation_start => operation_start_buf_e

                );
 U4 : BUFG

      port map (

                i => clk_in,
                o => clk

               ) ;

END;

 --**************************** end of architecture *****************************
--******************************************************************************
--******************************** METU / EE  **********************************
--******************************************************************************
--**************************** WRITTEN BY UGUR GUNGOR **************************
--******************************************************************************
--******************************** METU / EE ***********************************
--******************************************************************************
```

Triple Modular Redundancy with Correction Block VHDL Code

```
--********************************************************************************
--********************************* METU / EE  ***********************************
--********************************************************************************
--**************************** WRITTEN BY UGUR GUNGOR ****************************
--********************************************************************************
--********************************* METU / EE ************************************
--********************************************************************************
-- BLOCK NAME:  FP_multiplier_with_correction
-- ACIKLAMA  :  Triple moduler redundancy  with correction of a fault
--********************************************************************************

library ieee                    ;
use ieee.std_logic_unsigned.all ;
use ieee.std_logic_1164.all     ;

library UNISIM                  ;
use UNISIM.VCOMPONENTS.all      ;

--********************************************************************************
-- entity decleration
-- bu kisimda giris cikis sinyalleri gosterilmistir.

entity  FP_multiplier_with_correction is

port (
     rst                       :  in   std_logic ;
     clk_in                    :  in   std_logic ;
     FP_input_A                :  in   std_logic_vector (14 downto 0) ;
     failure_detection         :  out  std_logic ;
     FP_output_port            :  out  std_logic_vector (14 downto 0) ;
     insert_module_fault       :  in   std_logic ;
     insert_module_failure     :  in   std_logic ;
     insert_voter_fault        :  in   std_logic ;
     operation_start           :  in   std_logic ;
     fault_module_a            :  out  std_logic ;
     fault_module_b            :  out  std_logic ;
     failure_detection_display :  out  std_logic
);
end ;

-- end of entity decleration
```

```
--****************************************************************************

architecture FP_multiplier_with_correction_arch of  FP_multiplier_with_correction is

-- component assignments (kodun kullandigi alt moduller bu kisimda implemente edilir.)

component BUFG  --- clock buffer
port (
       i : in  std_logic;
       o : out std_logic
     );
end component;

component FP_multiplier IS
 port (
        rst                   : in   std_logic ;
        clk                   : in   std_logic ;
        FP_input_A            : in   std_logic_vector (14 downto 0) ;
        FP_input_B            : in   std_logic_vector (14 downto 0) ;
        FP_output             : out  std_logic_vector (14 downto 0) ;
        overflow              : out  std_logic ;
        underflow             : out  std_logic ;
        operation_start       : in   std_logic
      );
end component ;

-- signal assignments (kod icinde kullanilan sinyaller bu kisimda tanimlanir.)

signal  clk                   :   std_logic ;
signal  FP_output_module1     :   std_logic_vector (14 downto 0) ;
signal  overflow_module1      :   std_logic ;
signal  underflow_module1     :   std_logic ;


signal  FP_output_module2     :   std_logic_vector (14 downto 0) ;
signal  overflow_module2      :   std_logic ;
signal  underflow_module2     :   std_logic ;

signal  fault_detection       :   std_logic ;
signal  FP_output_module3     :   std_logic_vector (14 downto 0) ;
signal  overflow_module3      :   std_logic ;
signal  underflow_module3     :   std_logic ;
signal  FP_out                :   std_logic_vector (14 downto 0) ;
signal  FP_output             :   std_logic_vector (14 downto 0) ;

signal input1                 :   std_logic_vector (14 downto 0) ;
signal input2                 :   std_logic_vector (14 downto 0) ;
```

```
signal input3                  :   std_logic_vector (14 downto 0) ;

signal operation_start_buf     :   std_logic;

signal inputa                  :   std_logic_vector (14 downto 0) ;
signal inputb                  :   std_logic_vector (14 downto 0) ;
signal inputc                  :   std_logic_vector (14 downto 0) ;
signal counter_test            :   std_logic_vector (7 downto 0) ;

signal operation_start_buf_b   :   std_logic ;
signal operation_start_buf_c   :   std_logic ;
signal operation_start_buf_d   :   std_logic ;
signal operation_start_buf_e   :   std_logic ;

--*************************************************************************
BEGIN -- begin operations

-- asenkron atamalarin yapilmasi.

failure_detection_display <= fault_detection ;
failure_detection         <= fault_detection ;
FP_output_port            <= FP_output ;
operation_start_buf       <= '1' when  (operation_start = '1') else '0';

inputc <=  "0111000000001" &  counter_test(1 downto 0) when  (( insert_module_failure  =
'1')
           or ( insert_module_fault  = '1') )  else FP_input_A  ;

inputb <=  "0111110000000" & counter_test(1 downto 0) when ( insert_module_failure  =
'1')
           else FP_input_A  ;

inputa <=   FP_input_A ;


process (rst, clk )
begin

if rst = '0' then

   FP_out  <= "010000000000001";

elsif rising_edge(clk) then

   if ((overflow_module1  ='1') or (underflow_module1  ='1' )) then

      FP_out  <= "010000000000001";
```

```
    else

       FP_out  <=  FP_output  ;

    end if;

end if;

end process;


process(rst,clk)
begin
    if rst = '0' then

       FP_output             <="010000000000001";
       fault_detection       <= '0';

    elsif rising_edge(clk) then   -- clk nin rising'i ile alinmisti.

        if  insert_voter_fault = '0' then

            fault_detection           <= '0';
            if    ( FP_output_module1  =  FP_output_module2 ) then

                  FP_output           <= FP_output_module1;

            elsif ( FP_output_module1  =  FP_output_module3 ) then

                  FP_output           <= FP_output_module1;

            elsif ( FP_output_module2  =  FP_output_module3 ) then

                  FP_output           <= FP_output_module3;

            else

                  fault_detection     <= '1';
                  FP_output           <=  "010000000000001";

            end if;

       else

           fault_detection          <= '1';
```

```
       end if;

    end if;

end process;

process(rst,clk)
begin

 if rst = '0' then

    counter_test           <= (others=>'0');
    fault_module_a         <= '1' ;
    fault_module_b         <= '1' ;
    operation_start_buf_c  <= '0' ;
    operation_start_buf_b  <= '0' ;
    operation_start_buf_e  <= '0' ;
    operation_start_buf_d  <= '0' ;

 elsif rising_edge(clk) then   -- clk nin rising'i ile alinmisti.

    operation_start_buf_c  <= operation_start_buf_b ;
    operation_start_buf_b  <= operation_start_buf   ;
    operation_start_buf_d  <= operation_start_buf_c ;
    operation_start_buf_e  <= operation_start_buf_d ;

    if ( operation_start_buf_c = '1' ) and  ( operation_start_buf_b = '0' )  then
       counter_test        <= counter_test   +'1';
    end if;

    case counter_test(1 downto 0)  is

    when "00"=>

                 input1          <= inputa ;
                 input2          <= inputb ;
                 input3          <= inputc ;
                 fault_module_a  <= '1';
                 fault_module_b  <= '1';

     when "01"=>

                 input1          <= inputb ;
                 input2          <= inputc ;
                 input3          <= inputa ;
                 fault_module_a  <= '0';
                 fault_module_b  <= '1';
```

```
    when "10" =>

                 input1          <= inputc ;
                 input2          <= inputa ;
                 input3          <= inputb ;
                 fault_module_a  <= '0';
                 fault_module_b  <= '0';

    when "11"=>

                 input1          <= inputa ;
                 input2          <= inputb ;
                 input3          <= inputc ;
                 fault_module_a  <= '1';
                 fault_module_b  <= '1';

    when others =>

                 null;
    end case;

 end if;

end process;



u1 : FP_multiplier

      port map (

                rst             => rst                  ,
                clk             => clk                  ,
                FP_input_A      => input1               ,
                FP_input_B      => FP_out               ,
                FP_output       => FP_output_module1    ,
                overflow        => overflow_module1     ,
                underflow       => underflow_module1    ,
                operation_start => operation_start_buf_e

               );

u2 : FP_multiplier

      port map (
```

```
                rst             => rst                    ,
                clk             => clk                    ,
                FP_input_A      => input2                 ,
                FP_input_B      => FP_out                 ,
                FP_output       => FP_output_module2      ,
                overflow        => overflow_module2       ,
                underflow       => underflow_module2      ,
                operation_start => operation_start_buf_e

               );

 u3 : FP_multiplier

      port map (

                rst             => rst                    ,
                clk             => clk                    ,
                FP_input_A      => input3                 ,
                FP_input_B      => FP_out                 ,
                FP_output       => FP_output_module3      ,
                overflow        => overflow_module3       ,
                underflow       => underflow_module3      ,
                operation_start => operation_start_buf_e

               );

U4 : BUFG

     port map (

               i => clk_in,
               o => clk

              ) ;
END;

--****************************** end of architecture *****************************
--*******************************************************************************
--********************************* METU / EE  **********************************
--*******************************************************************************
--**************************** WRITTEN BY UGUR GUNGOR ***************************
--*******************************************************************************
--********************************* METU / EE ***********************************
--*******************************************************************************
```