A NOVEL FAULT TOLERANT ARCHITECTURE ON A RUNTIME RECONFIGURABLE FPGA

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

BY

İBRAHİM AYDIN COŞKUNER

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

NOVEMBER 2006

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

Assist. Prof. Dr. Cüneyt BAZLAMAÇCI	(METU, EE)	_____
Prof. Dr. Hasan Cengiz GÜRAN	(METU, EE)	_____
Assist. Prof. Dr. İlkay ULUSOY	(METU, EE)	_____
Dr. Şenan Ece SCHMIDT	(METU, EE)	_____
M.Sc. Alper ÜNVER	(TÜBİTAK – SAGE)	_____

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.

İbrahim Aydın COŞKUNER

ABSTRACT

A NOVEL FAULT TOLERANT ARCHITECTURE ON A RUNTIME RECONFIGURABLE FPGA

COŞKUNER, İbrahim Aydın
M.S., Department of Electrical and Electronics Engineering
Supervisor: Prof. Dr. Hasan Cengiz Güran

November 2006, 128 Pages

Due to their programmable nature, Field Programmable Gate Arrays (FPGAs) offer a good test environment for reconfigurable systems. FPGAs can be reconfigured during the operation with changing demands. This feature, known as Runtime Reconfiguration (RTR), can be used to speed-up computations and reduce system cost. Moreover, it can be used in a wide range of applications such as adaptable hardware, fault tolerant architectures.

This thesis is mostly concentrated on the runtime reconfigurable architectures. Critical properties of runtime reconfigurable architectures are examined. As a case study, a Triple Modular Redundant (TMR) system has been implemented on a runtime reconfigurable FPGA. The runtime reconfigurable structure increases the system reliability against faults. Especially, the weakness of SRAM based FPGAs against Single Event Upsets (SEUs) is eliminated by the designed system. Besides, the system can replace faulty elements with non-faulty elements during the operation. These features of the developed architecture provide extra safety to the system also prolong the life of the FPGA device without interrupting the whole system.

Keywords: Runtime Reconfiguration, Partial Reconfiguration, Fault Tolerant Reconfigurable Systems

ÖZ

ÇALIŞIRKEN YENİDEN BİÇİMLENDİRİLEBİLİR FPGA ÜZERİNDE HATAYA DAYANIMLI YENİ BİR YAPI

COŞKUNER, İbrahim Aydın
Yüksek Lisans., Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Hasan Cengiz Güran

Kasım 2006, 128 Sayfa

Alan Programlanabilir Kapı Dizinleri (FPGA) programlanabilir yapıları sayesinde, yeniden biçimlendirilebilir sistemler için uygun bir yapı sunarlar. FPGA gelen değişik ihtiyaçlara göre çalışma esnasında yeniden biçimlendirilebilir. Çalışırken Yeniden Biçimlendirme (RTR) olarak bilinen bu özellik sayesinde işlemler daha hızlı yapılabilir ve toplam sistem maliyeti düşürülebilir. Ayrıca RTR uyarlanabilir donanımlar ve hataya dayanımlı yapılar gibi birçok alanda kullanılabilir.

Bu tez çalışırken biçimlendirilebilir yapılar üzerine yoğunlaşmıştır. Çalışırken biçimlendirilebilir yapıların önemli özellikleri incelenmiştir. Örnek olarak Üçlü Modüler Yedekleme (TMR) sistemi, çalışırken biçimlendirilebilir bir yapı üzerinde uygulanmıştır. Çalışırken yeniden biçimlendirilebilir yapı sistemin hatalara karşı güvenilirliğini artırmıştır. Özellikle FPGA'lerin Tekli Hata Oluşumlarına (SEU) karşı olan zaafları tasarlanan sistem sayesinde giderilmiştir. Ayrıca sistem hatalı elemanları hatasız olanlarla çalışma sırasında değiştirebilmektedir. Geliştirilen mimarinin bu özellikleri sayesinde sistem daha güvenilir olmuş ve FPGA'in kullanım ömrü sistem durdurulmadan uzatılabilir hale gelmiştir.

Anahtar Kelimeler: Çalışırken Yeniden Biçimlendirme, Kısmi Yeniden Biçimlendirme, Hataya Dayanıklı Yeniden Biçimlendirilebilir Sistemler

To My Family

ACKNOWLEGMENTS

I would like to thank my supervisor Professor Hasan Güran, for his guidance and advices throughout the preparation of this thesis. Thanks to his advices and helpful criticisms, this thesis is completed.

I also thank to my family for their great encouragement, for their great support, and for their endless love.

I am very grateful to Türkmen Canlı, for the boards he brought at a critical time. I am also grateful to Salih Zengin for his ideas, suggestions, and technical support.

Special thanks goes to Yüksel Subaşı for the moral support he has given to me when I was near to give up, I also thank to Yiğiter Yüksel for his guidance and my colleagues for their help and friendship during this period. I also greatly appreciate Ahmet Coşar for the PCB he produced.

TÜBİTAK-SAGE who supported this work is greatly acknowledged.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVATIONS

ALU	Arithmetic Logic Unit
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
CAD	Computer Aided Design
CRC	Cyclic Redundancy Check
DSP	Digital Signal Processing
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GUI	Graphical User Interface
HDL	Hardware Description Language
I/O	Input-Output
IP	Intellectual Property
LUT	Look-up Table
PCB	Printed Circuit Board
PE	Processing Element
PROM	Programmable Read Only Memory
RA	Reconfigurable Architecture
RAM	Random Access Memory
RTR	Runtime Reconfiguration
SDR	Software Defined Radio
SEU	Single Event Upset
SoC	System on Chip
TMR	Triple Modular Redundancy
UART	Universal Asynchronous Receiver and Transmitter
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

CHAPTER I

INTRODUCTION

1.1 OVERVIEW

The microprocessors provide a flexible environment for the programmers. Any type of algorithm can be computed on a general-purpose microprocessor. However, this flexibility has a significant cost on computation time. The calculations are done on the same hardware resources for all type of applications (i.e. one instruction is handled at a time). Calculating algorithms in such serial structures results in performance degradation.

If the computations can be done in parallel, a significant speed-up can be achieved. *Reconfigurable architectures* provide enough hardware resources that can be used to make computations in parallel. Moreover, their flexible structure allows constructing different hardware configurations.

Reconfigurable architectures contain configurable connections and a plenty of logic resources. An application specific hardware can be formed by configuring these connections. These configurations can be stored by SRAM or Flash based switches. If SRAM based architecture is used on the reconfigurable device, infinite number of configurations can be loaded at different times. Loading a different configuration is called *reconfiguration*.

The most popular reconfigurable architecture is the Field Programmable Gate Array (FPGA). It is commercially available and used for high performance applications. FPGA is the ideal component for low volume products and it is used for prototyping Integrated Circuits (IC). With continuously increasing capacities and falling prices, they are also used in mass products now.

Normal usage of reconfigurable architectures such as FPGAs is as follows; all the demands is ready before the device runs. Then according to these

demands, only one final configuration is prepared and loaded to the reconfigurable device. Only this configuration runs on the device until a power-down occurs.

However, SRAM based reconfigurable devices enable changing configuration data whenever required. Some devices use this property to change configuration data during the device is running. Therefore, changing demands during the operation can be satisfied by reconfiguring these devices. This type of reconfiguration is called Runtime Reconfiguration (RTR). RTR introduced “Virtual Hardware” concept. It allows same hardware sources to be used for different purposes at different times by reconfiguring hardware. Therefore, a runtime reconfigurable architecture enables using unlimited circuits in only one chip by time multiplexing them.

RTR can be used in adaptable hardware applications, in-field upgrade of hardware. Other advantages of time multiplexing sources by RTR are reduced cost and reduced power of the system. Most importantly, speed-up can be obtained for different types of computations. Consequently, adding RTR property to the reconfigurable architectures offer new opportunities for digital systems.

1.2 OBJECTIVE OF THE THESIS

The main aim of the thesis is to investigate Runtime Reconfigurable architectures and to design one such architecture. In order to design a reconfigurable system, capabilities of a Field Programmable Gate Array (FPGA) are examined. Afterwards, a fault tolerant architecture is designed that use runtime reconfiguration to eliminate the faults. This design is implemented and tested on a runtime reconfigurable FPGA.

1.3 TOOLS USED

In order to implement a runtime reconfigurable system, some hardware and software tools were used. The tools are the following:

Hardware Tools

- D2SB Board from Digilent Inc.

- Personal Computer (PC)
- DIO1 Board from Digilent Inc.
- Custom made RS232 to TTL Converter Card
- Xilinx Parallel Cable III

D2SB Board, which is at the heart of the reconfigurable system, contains a Xilinx Spartan 2 - 200E FPGA on it. Personal Computer (PC) is responsible for the reconfiguration processes of the FPGA. DIO1 Board is used to display real-time information. An RS232 to TTL converter board is used for the communication of PC and FPGA. The configuration data of the FPGA is downloaded from the PC using Xilinx Parallel Cable III. Detailed description of the hardware configuration will be given in Chapter 5.

Software Tools

- Xilinx ISE 6.3i SP2
- VHDL
- Borland C++ Builder 5

Xilinx ISE is a CAD tool that is necessary to generate FPGA designs for Xilinx FPGAs. It has a Graphical User Interface (GUI) that can be used for standard FPGA designs. However, the GUI is not enough to achieve a runtime reconfigurable design. The command line tools of ISE such as NgdBuild, MAP, PAR, and BitGen are used in this design.

VHDL is a language that can describe hardware. It is used to generate circuits on FPGA. Files written in VHDL are synthesized using Xilinx Synthesis Tool (XST).

Borland C++ Builder 5 is used to generate a visual PC program. This program communicates with FPGA board and manages reconfiguration processes. The program also provides a user interface that enables user manipulation and shows the status of the system.

1.4 ORGANIZATION OF THE THESIS

The thesis is composed of six chapters. The chapter contents are the following:

In Chapter 2, a literature survey is done on reconfigurable computing Basic terms and concepts of reconfigurable architectures are explained. The application areas of the reconfigurable architectures are also given. Alternative reconfigurable FPGAs from different vendors are discussed and their critical characteristics are compared.

In Chapter 3, Xilinx FPGA and its features that enable runtime reconfiguration are discussed. Some properties of Xilinx FPGAs are explained from this viewpoint.

In Chapter 4, a simple reconfigurable application is mapped on Xilinx FPGA. The steps of designing a reconfigurable system are explained using that simple application. All tools and their batch files are described in detail.

In Chapter 5, a runtime reconfigurable TMR system that is designed to be highly fault tolerant is presented.

In Chapter 6, a conclusion of this thesis is given. Moreover, planned future works are given in this chapter.

CHAPTER II

BACKGROUND

In this chapter, basic concepts about reconfigurable architectures will be explained. In addition, some applications based on reconfigurable architectures will be emphasized.

2.1 RECONFIGURABLE COMPUTING

In the last few decades, *Reconfigurable Computing* has become popular in the area of computer architectures. Reconfigurable systems arise to compensate the differences of flexible microprocessors and high-speed ASIC circuits. A reconfigurable architecture takes advantages of both systems. It is more flexible than ASIC circuits since it can be reconfigured with changing computing needs. In addition, it has better performance than processors since it implements the desired algorithm on a dedicated hardware. As seen in Figure 2-1, reconfigurable architectures take place in between microprocessors and ASICs according to the flexibility and speed.

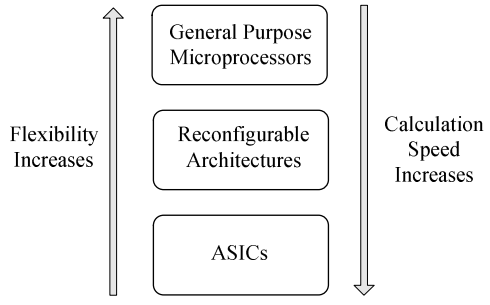

Figure 2-1: Comparison of Microprocessors, ASICs, and Reconfigurable Architectures

FPGAs are the first reconfigurable devices introduced as a commercial product. The first vendor Xilinx has produced FPGAs at mid-1980s with a very limited capacity. The capacity improvement of FPGAs has nearly followed Moore's Law [1]. Today FPGAs have millions of logical gates. Hence, it is possible to implement more than one medium-sized processor inside one FPGA. Xilinx MicroBlaze, Altera Nios are examples of such processors. The improvement of these reconfigurable devices leads to raise academic research on reconfigurable architectures.

2.1.1 The Aim of Reconfigurable Architectures

The hardware on reconfigurable architectures can be reconfigured if the demands are changed. This flexibility allows reusability of the hardware resources. Therefore, reconfigurable architectures can be used for all applications that can benefit from hardware reusability. Some general benefits of this flexibility are speeding-up calculations and resource saving.

2.2 GRANULARITY OF RECONFIGURABLE ARCHITECTURES

Reconfigurable architectures generally composed of array of reconfigurable unit blocks and routing sources that connect these blocks. The size of these unit blocks reflects granularity of the architecture. The granularity of these devices ranges from fine to coarse grain. They can be mainly classified as

- Fine-Grained,
- Coarse-Grained and
- Heterogeneous Architectures.

Fine-grained architectures are suitable for bit-level manipulations and contain elements such as LUT. On the other side, coarse grain architectures have elements such as ALU or small processor, which makes them suitable for word level computations. Heterogeneous architectures also become available to use advantages of both architectures.

Fine Grained Architectures

Fine-grained architectures are intended to implement bit level logic circuits. Calculations that have arbitrary bit width can be done by using fine-grained architectures. The advantage of fine-grained architectures is that it can map any logical circuit on the hardware. However, the overhead of routing resources increases as a cost of this flexibility.

The well-known example for a fine-grained architecture is FPGA. FPGAs are commercially available reconfigurable devices and most of reconfigurable computing researches are done on them.

Fine-grained reconfigurable architectures are generally composed of configurable Logic Cells (LC), configurable Routing Sources, and Input-Output (I/O) Sources. The general structure of a fine-grained architecture is shown in Figure 2-2. The Logic Cells are connected to other ones using routing resources. There are switch matrices that determine how these cells and routing lines will be connected. I/O cells are also used to connect internal resources to the outside world.

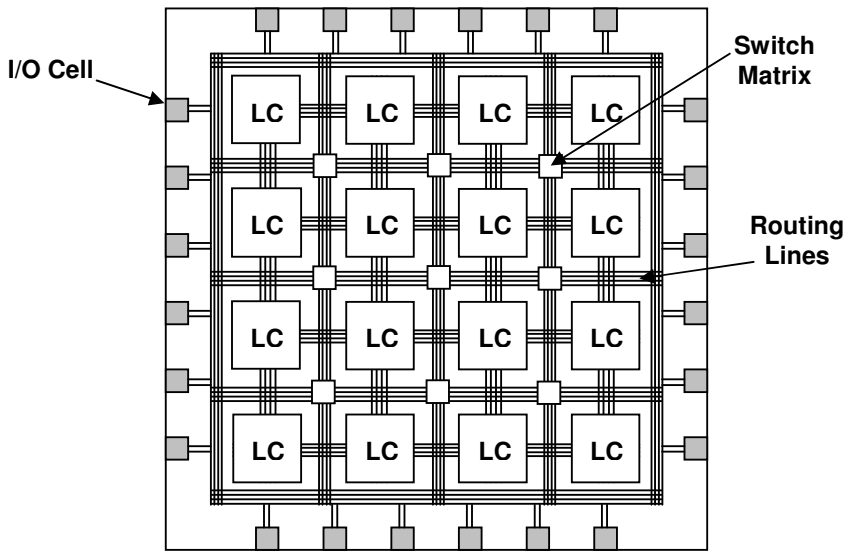

Figure 2-2: General Structure of a Fine-Grained Architecture

Logic Cells (or Logic Tiles) are used to implement logical functions. Most of the FPGA vendors use Lookup Table (LUT) to implement bit-level combinational logic functions on Logic Cells. For example, a LUT takes four input signals, gives one output signal on Virtex Family devices of Xilinx. The combinational function (4 inputs, 1 output) of LUT is encoded to 16 Bit and stored on configuration memory of FPGA. In addition to LUT, a Flip-flop (FF) is placed on same logic cell to generate synchronous circuits. Logic Cell structure of an SRAM based FPGA is shown in Figure 2-3

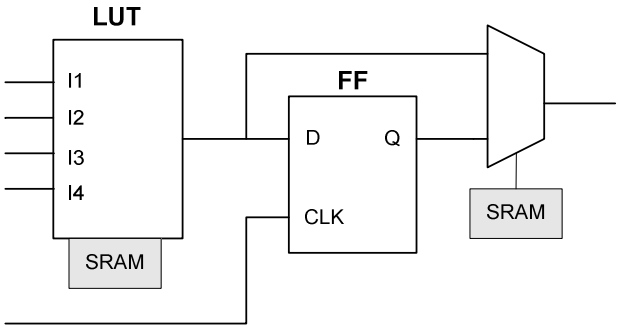

Figure 2-3: Basic Structure of a Fine-Grained Logic Cell on an FPGA

Fine-grained architectures can be used for a very broad range of applications since fine granularity allows mapping almost all types of applications. However, efficiency will decrease for some applications because of fine granularity. Therefore, only some applications can be classified as suitable for fine-grained architectures. The well-fitted applications such as image processing, data encryption need bit-level data handling [2]. In addition to these applications, finite state machines (FSMs) can be good candidates for mapping on fine-grained architecture (since state transitions of FSMs mostly depend on single bit values).

Coarse Grained Architectures

Coarse Grained architectures are composed of array of Processing Elements (PEs). Processing Elements are designed to compute word-level computations. They contain coarse grain structures such as an ALU or a small processor. Therefore, a datapath calculation can be easily mapped on coarse

grain architectures. The word length of PE differs on different types of architectures. It ranges from 2 bit to 128 bit while most of them are 16 bit [3]. In Figure 2-4, the PE of KressArray is shown. It is called reconfigurable Datapath Unit (rDPU), and it has a 32-bit ALU and registers.

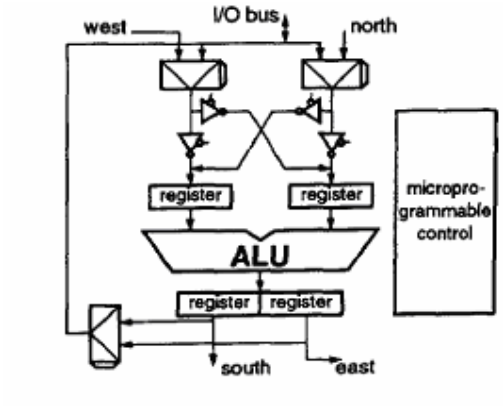

Figure 2-4: Reconfigurable Data Unit of KressArray [6]

The elements of the array are connected with a configurable routing. I/O ports connect the PEs to the outside world. The arrangement of the array differs according to the target application. Different array structures are available such as Mesh, Crossbar, Linear array, 2-Dimensional Array. In Figure 2-5, these structures are shown.

Linear arrays are designed as a pipeline with reconfigurable connections. Rapid and PipeRench are the popular linear array designs. *Mesh arrays* arrange PEs in two-dimension and they are connected with nearest neighbor. Popular mesh based course grained structures are MorphoSys, CHESS, Matrix, RAW and Garp. Some mesh structures add global connections to increase the performance of the array. These structures are also called *2-Dimensional arrays* and enables connection of arbitrary PEs. *Crossbar structures* connect all PEs with each other. However, this results in increased cost for the routing resources. PADDI-1 and PADDI-2 are the crossbar structures, which are intended to prototype datapath for Digital Signal Processing (DSP) Algorithms [4].

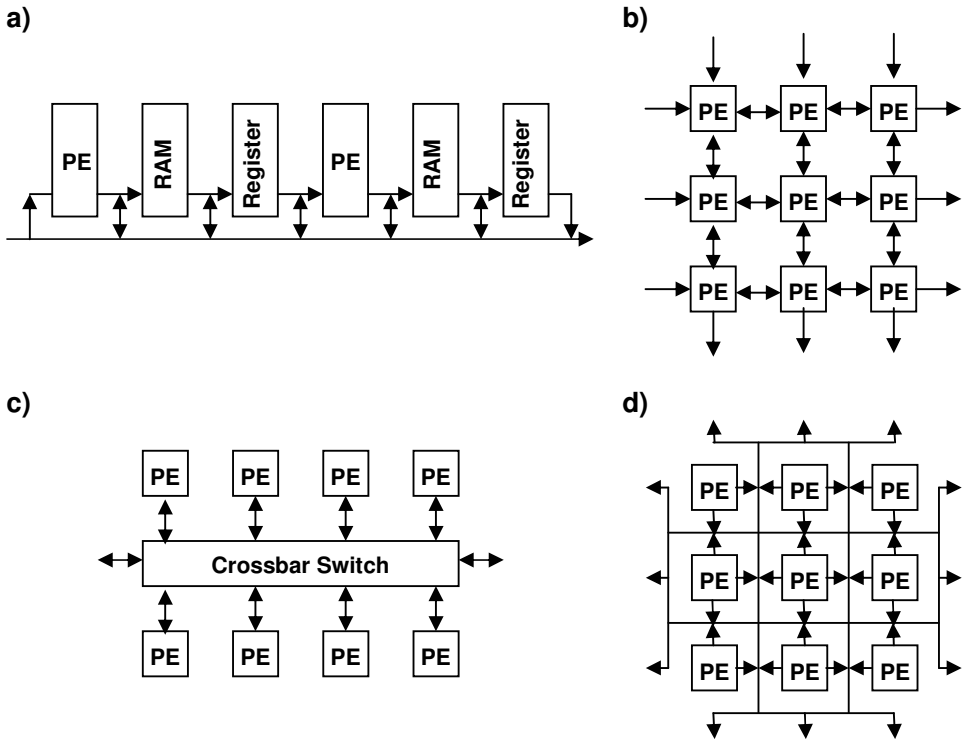

Figure 2-5: Array Structures of Coarse Grain Architectures a) Linear Array b) Mesh c) Crossbar d) 2-Dimensional Array

Some coarse grain architectures have also embedded routing structures and/or memory inside the PE. For example, KressArray-3 [5] has rDPU that contains an ALU and routing structure at the same time.

Datapath calculations can be easily mapped on coarse grain architectures. For instance, mapping of **y = a * b + c * (d + e)** on KressArray is shown in Figure 2-6.

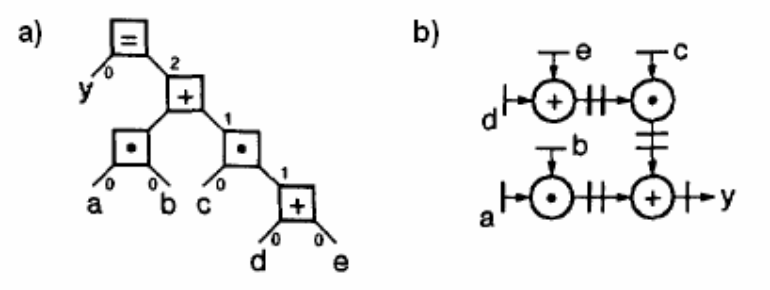

Figure 2-6: A Datapath Equation and Hardware Mapping [6] a) Equation mapped to the node levels b) Hardware mapping of the equation

Fine vs. Coarse Granularity

Both structures have their own advantages and disadvantages. Fine-grained architectures can implement any logic function in one clock cycle, which is impossible on coarse grain architectures. However, this flexibility is obtained by using high number of routing resources. The increase of routing sources results in some drawbacks. First, the area needed for routing will be much higher than logical elements in a fine-grained architecture. Power consumption increase and frequency decrease are other disadvantages of fine-grained structures. Routing sources of fine-grained architectures also need more configuration data than the coarse grain architectures. Because of higher configuration data, reconfiguration time of fine-grained architectures is higher than coarse-grained architectures.

Then why fine-grained FPGAs are extensively used instead of coarse-grained architectures? The reason may be flexibility dominates the other advantages of coarse-grained architectures. If an application can be mapped to coarse grain architecture, it can get high speed-up. However, another application cannot get considerable speedup, if it is not well suited on the same coarse grain architecture. This factor limits usage of one coarse grain architecture for different applications. Therefore, a unique coarse grain architecture is not available that can be used for all type of applications. Such a universal coarse grain structure does not seem to be available also in the future [5].

In addition, the compiler support of coarse grain architectures is still in its start stage. Current mapping tools cannot utilize the full potential of coarse-grained architectures due to the hardware complexity [7].

Heterogeneous Architectures

Heterogeneous architectures contain both fine and coarse grain elements to take advantage of both worlds. Usage of coarse grain elements results in an increase of the system performance. By using fine grain elements flexibility is maintained. Therefore, newer reconfigurable architectures are designed heterogeneously. Generally, arithmetic functions that occupy large space on fine grain blocks are moved to coarse grain blocks in heterogeneous architectures.

For example, Xilinx has embedded multiplier blocks into their FPGA devices starting from Virtex-II family. In newer devices, such as Virtex-4, there are multiply-accumulate (MAC) units, which are well fitted to Digital Signal Processing (DSP) filter implementations. These embedded units occupy less area, consume less power, and work with higher frequencies since they have a fixed routing inside. Therefore, embedded multipliers are much more efficient than implemented multipliers with fine grain elements.

2.3 RECONFIGURATION APPROACHES

Dynamic (Run-Time) Reconfiguration

If device is reconfigured according to the changing demands during the operation then it is called *dynamically reconfigurable* architecture. In such architectures, same hardware sources can be used for different purposes at different times by reconfiguring hardware. Therefore, the hardware becomes a *virtual hardware*, which looks like using infinite hardware resources on a system. In Figure 2-7, a dynamically reconfigurable system is shown.

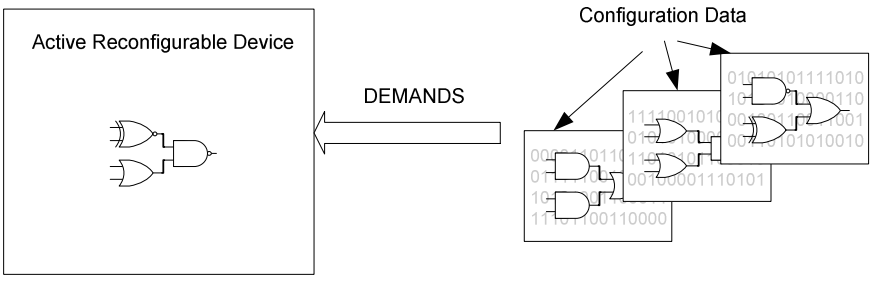

Figure 2-7: Dynamic Reconfiguration of Hardware

Note that, *runtime reconfiguration* term is also used instead of dynamic reconfiguration.

Partial Reconfiguration

Partial reconfiguration is a sub-class of runtime reconfiguration. According to the coming demands, only a part of these devices is reconfigured instead of reconfiguring whole device. In addition, while reconfiguring some parts of the device, remaining parts still operate in such partially reconfigurable devices. Therefore, different functions can be loaded to partially reconfigurable part while the other parts are working, as seen in Figure 2-8.

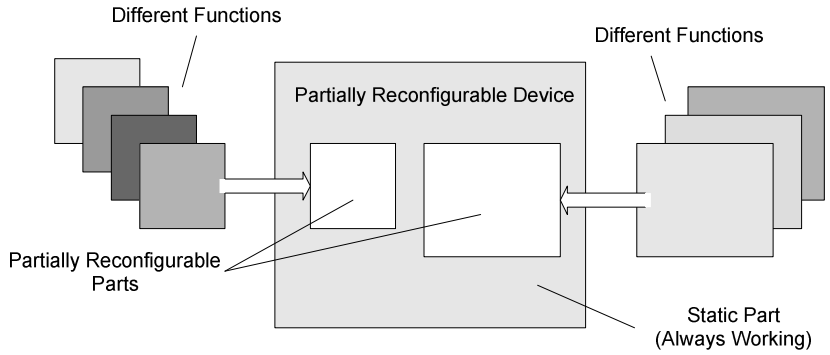

Figure 2-8: A Partially Reconfigurable Device and its Configurations

Partial reconfiguration has many benefits. For instance, the hardware on partially reconfigurable parts can be shared by different applications at different times. The other parts can be maintained as fixed parts that always remain active. The fixed parts can manage scheduling operations of reconfigurable parts. Therefore removing unnecessary hardware and inserting necessary ones to the system, results in reduced cost and power. In addition, system can operate without interrupting by keeping fixed part in contact with the outside world.

Partial reconfiguration property of reconfigurable devices is also used for speeding up the applications in some researches. For example, in [8] a CPU is placed on the fixed part and coprocessors are placed on reconfigurable parts of

the FPGA. Different coprocessor configurations are prepared off-line and they are loaded to the reconfigurable parts with changing demands.

Another advantage of partial reconfiguration is reduced reconfiguration time. Since reconfiguration of full device is not needed, size of reconfiguration data also decreases. In other words, reconfiguration times are directly proportional with the reconfigured modules size. For example, if reconfiguration time of the entire device is 4 ms then quarter of the device can be reconfigured at 1 ms.

Self Reconfiguration

If the reconfigurable device reconfigures itself without any aid from the outside world then it is called *self-reconfigurable* system. Data required for different configurations are generally stored on standard storage mediums. A part of the device is responsible for taking data from the storage medium and sending this data to the configuration port of the device. The configuration of the device changes after port takes the data.

The main advantage of such reconfiguration is elimination of the need for external configuration controller. This results in reduction of the total system cost. Moreover, configuration data can be compressed at the storage side, and it can be decompressed by the configuration controller. Therefore, the size of the configuration data will decrease.

Different configuration port types can be used for self-reconfiguration. For example, if the device has only a configuration port available at external pins, then it can be used as shown in Figure 2-9. In this structure, configuration data is taken by configuration controller and it is sent to the external configuration port of the device. However, this approach has some drawbacks. Firstly, pins used by configuration controller cannot be used for different purposes. Secondly, the configuration data sent from configuration controller to configuration port cannot be secure since data signals must go through PCB.

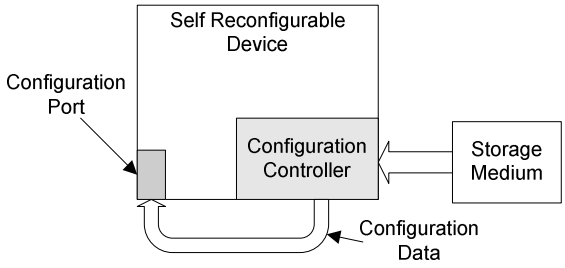

Figure 2-9: Self-Reconfiguration from External Configuration Port

Some devices (such as Xilinx Virtex-II FPGA) have integrated configuration port inside the fabric of the device. The configuration controller can access this port internally (without going through pins) as shown in Figure 2-10. As a result, pins are not wasted for reconfiguration purpose and reconfiguration can be done securely.

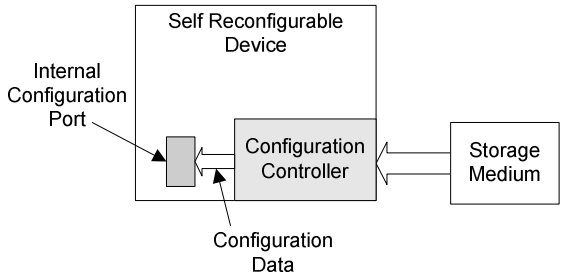

Figure 2-10: Self-Reconfiguration using Internal Configuration Port

In some works such as [9] [10], this structure is used to implement a secure runtime reconfiguration. An initial configuration is loaded to the device that includes configuration controller and decryption hardware. The other parts are reserved for user applications and loaded by a partial reconfiguration. The partial configuration data is encrypted with a known key. This key is also stored on decryption circuit. Flow of secure partial reconfiguration occurs as follows: Encrypted configuration data is taken from an external source such as a storage medium or a radio link. Then it is decrypted by decryption circuit using the known key and passed to the configuration controller. Configuration controller writes

configuration data to the internal configuration port of the device and user application switches to another one. As a result, reconfiguration of user application becomes secure with this method since raw configuration data cannot be monitored from the outside world.

2.4 RECONFIGURATION TIME

Reconfiguration time is an important criterion on runtime reconfigurable architectures. Especially the applications that use runtime reconfigurable architectures to speed up calculations need fast reconfiguration. The logic circuit inside reconfigurable part must be replaced with another logic circuit in a limited time for such applications. In Figure 2-11, distribution of different applications according to the reconfiguration frequency is shown. The overhead of this reconfiguration time must be compensated by speeding up the calculations by hardware.

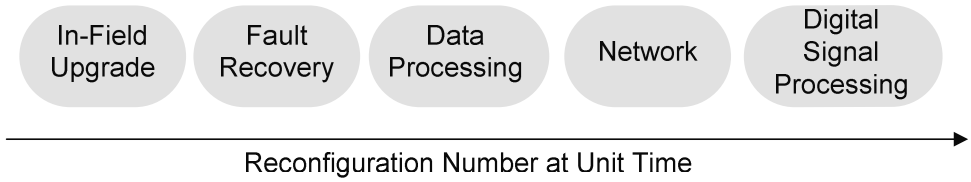

Figure 2-11: Required Reconfiguration Times for Different Application Types

Reconfiguration time of commercially available FPGAs still takes around milliseconds. Therefore, the applications that take more than milliseconds at least can obtain a speedup by reconfiguring FPGAs. Generally, data processing applications are in this range. For example, encryption/decryption or sorting algorithms are good candidates to run on a runtime reconfigurable FPGA.

Some other devices such as DPGAs have been proposed to reduce the reconfiguration time to nanoseconds. However, they did not become commercially available due to their high costs (due to large configuration memory requirements) [11].

Nevertheless, the overhead of reconfiguration time can be reduced by dividing reconfigurable device into multiple parts and using scheduling algorithms.

Reducing reconfiguration time overhead allows mapping highly dynamic applications onto reconfigurable hardware [12]. Two types of scheduling algorithm can be used. These are runtime scheduling and design time scheduling.

Scheduling of applications at runtime brings a new concept called *Hardware Operating System*. The hardware operating system work *online,* which means decisions are made during the system is running. Hardware operating systems also try to find solution for online placement of tasks onto different parts of the reconfigurable hardware. In Figure 2-12, elements of hardware operating system is shown.

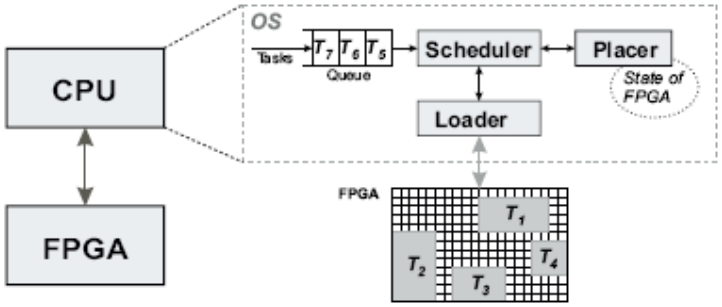

Figure 2-12: An Example of Hardware Operating System [13]

Some works also try to reduce the reconfiguration delay by using offline-scheduling algorithms. For example, [14] assumes the sequence of the tasks is already known before running the system (i.e. at design time) and it reduces the reconfiguration overhead up to 40%.

2.5 PARTIALLY RUNTIME RECONFIGURABLE FPGAS

FPGAs are widely used devices on reconfigurable computing applications since most of them are inherently reconfigurable. A combination of a CPU and reconfigurable FPGA can be used as a reconfigurable platform. CPU can manage reconfiguration processes of the FPGA and map different hardware configurations to FPGA at different times. However, this structure is not so efficient since two devices are needed for this system. Instead, a *partially runtime reconfigurable* FPGA can do the tasks of both CPU and non-partially reconfigurable FPGA as a

System on Chip (SoC). FPGA can be divided into two parts in which one part is static and the other one is reconfigurable. Then a soft CPU can be mapped on the static part and it can manage reconfiguration processes of the reconfigurable part.

On a partially reconfigurable FPGA, more than one area can be reconfigured at an instance. Therefore, multiple tasks can be loaded at the same time and they can be reconfigured independent from the others .This is another advantage of using partial reconfiguration of FPGA.

Altera, Atmel, Lattice, QuickLogic, and Xilinx are the major FPGA vendors in the world. About half of them have FPGA products that offer partial runtime reconfiguration. These partially reconfigurable FPGA devices are listed below:

- Atmel AT6K
- Atmel AT40K
- Atmel AT94K
- Lattice ORCA
- Xilinx Virtex
- Xilinx Spartan

Xilinx Virtex and Spartan FPGA families can be partially reconfigured in a column-based approach. FPGA can be divided into columns and any of the columns can be reconfigured while the others are still running. There are also some restrictions to achieve partial reconfiguration. For example, the column boundaries must be determined at design time, the boundaries cannot change during execution. In addition, modules must communicate through special structures. Partial reconfiguration of Xilinx FPGAs will be discussed in further depth in Chapter 4.

Atmel AT6K, AT40K, and AT94K series FPGA can achieve runtime partial reconfiguration. The technology of reconfigurable logic inside FPGA is called *Cache Logic* by Atmel. The reconfigurable part can be any rectangle inside FPGA. AT94K series FPGA includes an AVR microcontroller embedded on FPGA. This microcontroller can change the logic inside the FPGA.

Lattice ORCA FPGA's can be partially reconfigured. For partial reconfiguration, the address is written with "Explicit" mode. Indeed every address frame is written into the bitstream, followed by the data frame for each address. Partial reconfiguration is done by setting a bitstream option in the previous

configuration sequence that tells the FPGA not to reset the entire RAM configuration during a reconfiguration [15].

2.5.1 Reconfiguration Times of FPGAs

Full reconfiguration time of Xilinx XCV50 is 1.2 ms with SelectMAP 8 bit parallel mode at 60 MHz with handshaking, where XCV50 is the smallest device of Virtex series FPGAs. Reconfiguration time for Atmel FPGA AT40K40 is 631 µs in parallel mode, with writing 16-bit wide words at 33 MHz [16]. Full reconfiguration of ORCA OR4E06 takes 5.94 ms [17]. Note that, these devices are smallest devices of the vendors. Newer and higher capacity FPGAs will have bigger configuration data. However, they also speed-up the configuration ports, which maintain reconfiguration times almost in the same order. For example, Xilinx Virtex-4 has a 32-bit SelectMAP configuration port, which can reach up to 100 MHz clock rates.

2.6 APPLICATION AREAS OF RECONFIGURABLE ARCHITECTURES

A wide range of applications can benefit from reconfigurable architectures. Some applications areas of the reconfigurable architectures are listed below.

- Easy Prototyping, Low Volume Products
- Field Upgrade of Hardware

2.6.1 Easy Prototyping – Low Volume Products

A digital Application Specific Integrated Circuit (ASIC) can be prototyped using a reconfigurable architecture. To accomplish this, different hardware configurations are mapped on a reconfigurable architecture at design time. After verifying correct operation of the designed circuit, an ASIC can be produced. If this circuit is not a mass product, reconfigurable device can also be used as a final product. Hence using a reconfigurable device will eliminate costly processes of producing an ASIC device.

2.6.2 In-Field Upgrades

Being a reconfigurable architecture also provides some other unique properties. Reconfigurable devices provide an opportunity to change hardware on the fly. In other words, the device can be reconfigured easily by writing configuration data to the configuration memory. This feature can be used on systems that need upgrade of hardware structure during operation. In such systems, reconfigurable device can be used as a heart of the system. A remote computer can connect to the system and send configuration data. Then hardware structure can be changed by reconfiguring the device with the new configuration data. Since hardware components are generally base of a system, reconfiguration can almost replace whole architecture with a new one. This type of upgrade can save time and money for the producer.

Even there may be conditions such that it may be impossible to upgrade device without in-field upgrade. For example, servicing or replacing components physically is impossible on a satellite system. In such architectures, using reconfigurable architecture that can be reconfigured with a remote connection is inevitable. As a result, reconfigurable devices are ideal components for systems that need in-field upgrade operations. Some works [10] deal with partial reconfiguration of hardware that eases in-field upgrades.

2.7 APPLICATION AREAS OF RUNTIME RECONFIGURABLE ARCHITECTURES

Changing the hardware on a running system is possible by using Runtime Reconfigurable architecture. This feature enables using runtime reconfigurable architecture as a *virtual hardware* source. In other words, different hardware configurations can be used at different times by RTR. Many applications can benefit from this feature to save cost, power, and resource usage on digital circuits. Moreover, applications can get speedup by using RTR, since it provides a flexible dedicated hardware for different functions. As a result, RTR can be used for the following purposes:

- Cost and power reduction
- Designing an Adaptable Computing Platform

- Designing Fault Tolerant Circuits
- Speeding-up Computations

2.7.1 Cost and Power Reduction

RTR can reduce needed resource size if the required hardware can be divided into multiple parts. These smaller parts can be mapped to the hardware by generating configurations. Then these configurations can be loaded to the device at different times by using RTR. A scheduler arranges the reconfiguration operations according to the demands. Therefore, a smaller capacity device can be enough to map a bigger circuit on it. This results in cost and power reduction of the system.

For example, Lianos et al. proposed a space efficient method for calculating Fast Fourier Transform (FFT) by using a dynamically reconfigurable architecture [18]. One reconfigurable vector calculates a column of FFT then feeds the outputs into the reconfigurable vector again to calculate consecutive stages of the FFT. Therefore, only one reconfigurable vector is enough to calculate FFT on a dedicated hardware by using RTR.

In another work [19], a reconfigurable architecture is implemented that behave as Programmable Logical Controller (PLC). Designed architecture utilizes Temporal Petri Net language to describe applications. The sequential structure of Petri Nets allows splitting applications into multiple parts. Then these parts are mapped to same FPGA and used sequentially by reconfiguring it. This architecture can divide whole application up to 40 parts. Therefore, using 40 times smaller capacity FPGA can be enough instead of using a big one. This can reduce the cost of device from $317 to $38.

Widespread usage of mobile systems increased the demand for low power consumption while maintaining high performance. Some works deals with mobile systems that use dynamic reconfiguration to reduce the total power of the system. In [20], control units of an automobile are implemented on a runtime reconfigurable FPGA. The user area is divided into four smaller parts. High number of control units (e.g. 20 units) that cannot fit to one-device shares available sources by time multiplexing. A scheduler determines reconfiguration

processes of control units. As a result, the system only consumes power of four control units for implementing much higher number of control units. In addition, a part of FPGA is always kept in contact with the outside world since only necessary parts reconfigured. This eliminates a need for external controller of reconfiguration process, which contributes power and cost reduction.

2.7.2 Adaptable Computing

Some types of applications require adaptation of hardware to changing demands. In such applications, implementing circuits on a static device is impossible, even a highest capacity one is used. The ultimate solution of this problem is using a reconfigurable hardware. Infinite number of configurations can be prepared and reconfigurable hardware can be reconfigured with new demands.

Furthermore, many applications can benefit from reusability of hardware on reconfigurable architectures. Computations can be divided into multiple parts and they can be computed one after another with a parallel processing structure. If the gain obtained on area usage compensates the latency, the reconfigurable architecture can be preferred. For example, a matrix multiplication method proposed by L. Jianwen et al. [21] can do matrix multiplication with 80% less area than linear array structure. It have also used approximately 50% less area than linear array structure in terms of AT Metric (product of area and latency)

Some of the adaptable-computing applications absolutely need reconfigurable architectures are the following:

Evolvable Hardware

Evolvable Hardware is the application of Genetic Algorithms on circuits. Evolvable algorithms can find a circuit from its behavioural description [22]. There are two methods available to achieve this goal. One of them, known as Extrinsic Evolvable Hardware, simulates alternative circuit configurations and selects the best one. The other method, known as Intrinsic Evolvable Hardware, directly tests alternative circuit configurations on hardware. Then best of the configuration is selected [23]. It is necessary to use a reconfigurable hardware to test large number of alternative configurations. Therefore, RTR is necessary to implement Evolvable Hardware with the second method.

Hardware implementations of Robotics or Artificial Neural Networks also require such evolvable structures. Therefore, they are the candidates of RTR applications.

Software Defined Radio

Software Defined Radio (SDR) is another concept that involves adaptable hardware sources inside. SDR is a wireless platform that can work with different communication protocols. It can adapt to a communication protocol just by downloading and changing the configuration on the platform as a software module. SDR requires a large amount of digital signal processing operations. For this reason, SDR systems generally use a Digital Signal Processor (DSP) and an FPGA as a coprocessor [24]. DSP makes software operations whereas FPGA implements different filters and reconfigured with changing necessities. However, it is possible to use only one runtime reconfigurable FPGA to do operations of both DSP and FPGA. This runtime reconfigurable FPGA can be divided into two parts where one part is static and the other one is dynamic. Static part can be loaded by a soft processor core. Dynamical part can be reconfigured to run alternative coprocessor cores. Some researches (such as [25] and [26]) deal with such single chip systems that can reconfigure themselves with changing demands.

2.7.3 Speeding-up Computations

Reconfigurable Architectures (RAs) provide a flexible structure as microprocessors. Microprocessors allow changing the software and RAs allow changing hardware. Dedicated hardware on RA enables parallel computing while software on microprocessor allows only serial operations. Therefore, implementing a computational task on a dedicated hardware on RA is much faster than executing on a processor as software.

Reconfigurable architectures can be used to accelerate computational tasks by mapping algorithms or parts of them to the dedicated hardware. For each different computational task, hardware can be reconfigured to map calculations on hardware. The rate of computations changes also affects the reconfiguration period of the hardware. If reconfiguration overhead is less than the gain obtained by mapping calculations on hardware, a considerable speed-up can be achieved.

Moreover, it is known that more than 90% of time is consumed on 10% of code in most of the software programs [27]. These codes are generally nested loop statements, which intend to take longer time than other structures. If the statements inside a loop can be mapped directly on hardware, execution time will decrease. The hardware on the reconfigurable architectures can be used for such loop statements. For each loop statement, an alternative configuration is created. Then by using runtime reconfiguration, infinite number of loop statements can be mapped on hardware. Therefore, the software can be executed more parallel, and it can be accelerated more.

Many algorithms such as image processing, image compression /decompression, data encryption/decryption may benefit from the parallelism of reconfigurable architectures. The only necessity to get a speedup is reconfiguration time cost must be lower than the gain obtained with parallelism.

2.7.4 Fault Tolerant Systems

Fault tolerance on hardware generally requires reserving spare sources and replacing faulty sources with spare ones. Reserving spare sources is a trivial issue on reconfigurable devices since they are composed of array of identical elements. Many researches such as [28], [29] and [30] use inherent reconfiguration property of the FPGAs in order to tolerate faults on them. In Chapter 5, researches dealing with this topic will be discussed in more detail.

2.8 APPLICATION IN THIS WORK

A fault tolerant hardware was also designed in this work, which uses RTR property of an FPGA. Faults were eliminated using reconfiguration of the hardware. Furthermore, fault injection was done with the help of RTR. In Chapter 5, working principle of designed architecture will be explained in more detail.

CHAPTER III

XILINX FPGA ARCHITECTURE AND TOOLS

In this chapter, the general architecture of Xilinx FPGAs will be explained. At necessary points, examples will be given from Virtex-E or Spartan-2E series of FPGAs.

3.1 MAIN STRUCTURE OF XILINX FPGAS

Xilinx FPGA's are composed of Configurable Logic Blocks (CLB), Input Output Blocks (IOB), BlockRAM's (internal RAM), and the configurable routing matrix. Array of CLBs forms the FPGA structure. They are connected using routing lines and they implement logic functions. For example, the device used in this work, XC2S200E has 28 rows and 42 columns of CLBs. The structure of Spartan 2E FPGA is shown in Figure 3-1.

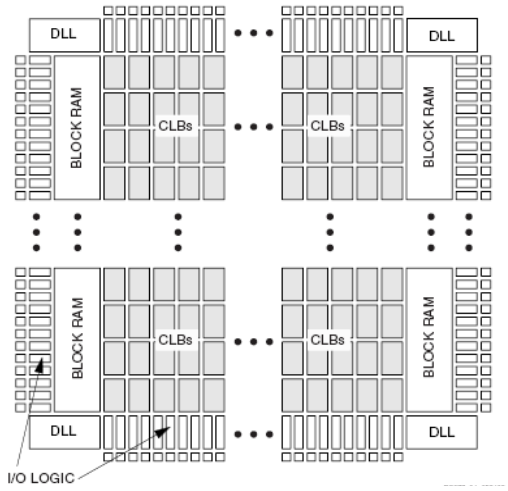

Figure 3-1: General Structure of Spartan 2E FPGAs [31]

3.1.1 Configurable Logic Block Structure

Each Configurable Logic Block (CLB) has two identical slices each of which have two Logic Cells (LCs). These logic cells are the basic building block of the FPGA. There is one flip-flop as storage elements and one look-up table which implements combinational logic in a LC. Also, carry logic elements are inserted to speed-up arithmetic operations. A CLB structure of Virtex-E or Spartan 2E device is shown in Figure 3-2. Note that CLB architectures of Virtex-E and Spartan 2E are same.

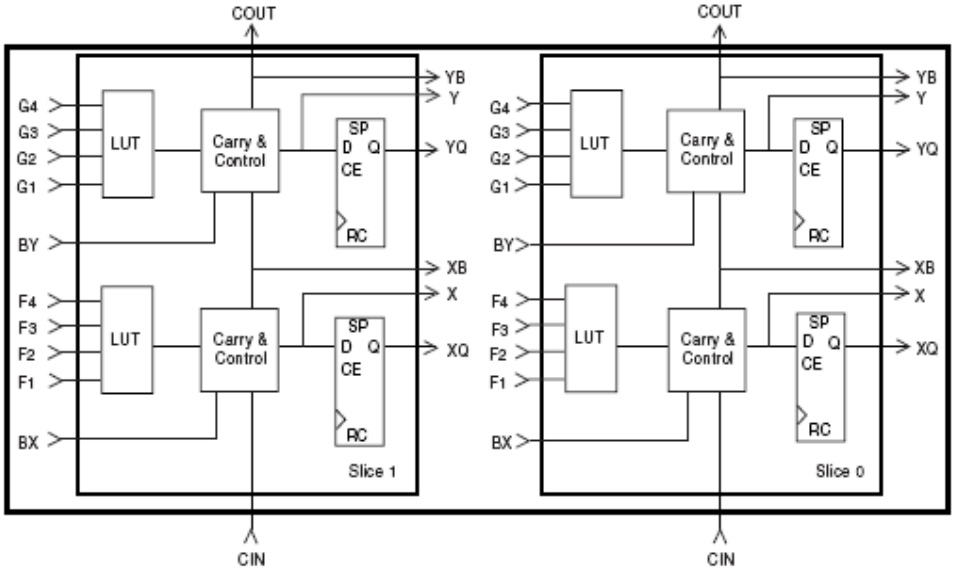

Figure 3-2: A CLB of a Virtex-E (or Spartan 2E) device

3.1.2 Input Output Block Structure

FPGAs are connected to the outside world using programmable Input Output Blocks (IOBs). As shown in Figure 3-3, an IOB include flip-flops (FF) for input, output and tri-state enable signal. These FFs can be used to obtain minimum FF to pin delay. In addition, a number of IOBs are grouped to form a bank. Voltage levels of banks can be selected from different types of I/O standards.

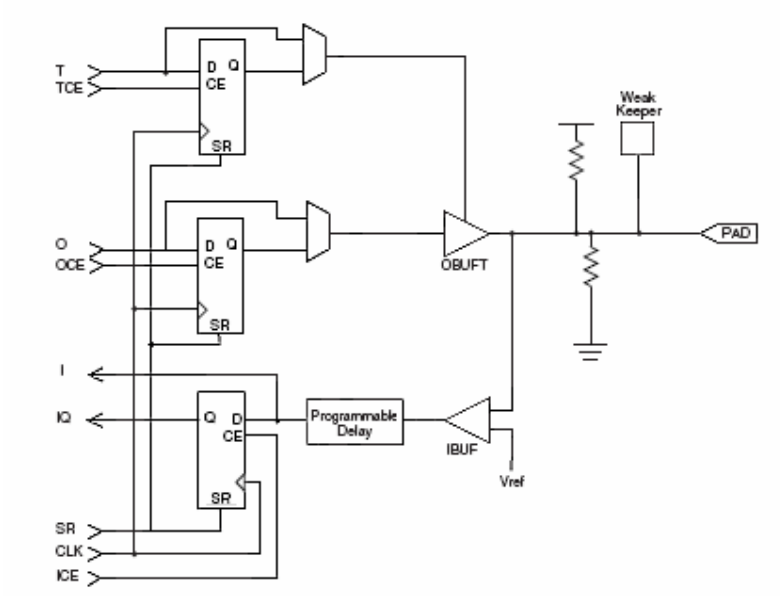

Figure 3-3: Input/Output Block Structure of Virtex-E Device

3.1.3 Routing Structure

Routing structure is reconfigurable on Xilinx FPGAs, which is one of the necessities to be a reconfigurable device. It is also adjusted in a hierarchical manner to make it area efficient. There are mainly four types of routing resources:

- *Local Routings* are used to make connections inside the CLB, between CLB and General Routing Matrix (GRM), and between two CLBs.
- *General Purpose Routing* connects most of the signals on the FPGA. CLB's are connected to other resources using GRM switch. In addition, a GRM is connected to adjacent six GRMs. GRM connections are shown on Figure 3-4. These switches also connect horizontal and vertical lines. These vertical and horizontal long lines span the full height/width of the FPGA.

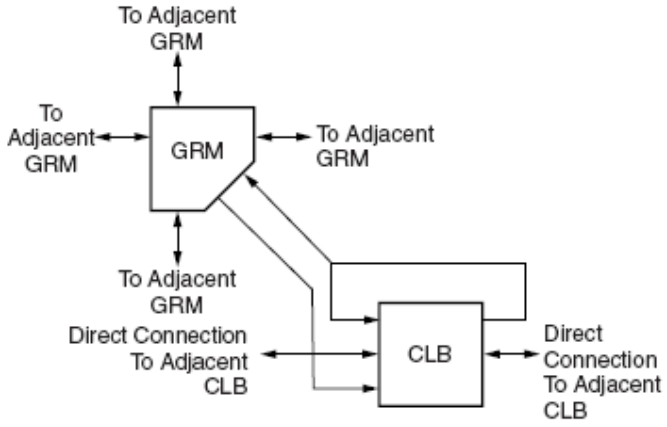

Figure 3-4: General Routing Matrix and its Connections [31]

- *Dedicated Routing* sources connect special signals on the FPGA. For example, there are four signal lines horizontally placed on the FPGA for each CLB row as shown in Figure 3-5. These lines can be used for tri-state bus implementation. In this work, tri-state lines were used to implement a bus inside the FPGA. This bus is called bus-macro and will be described in detail in Chapter 4.

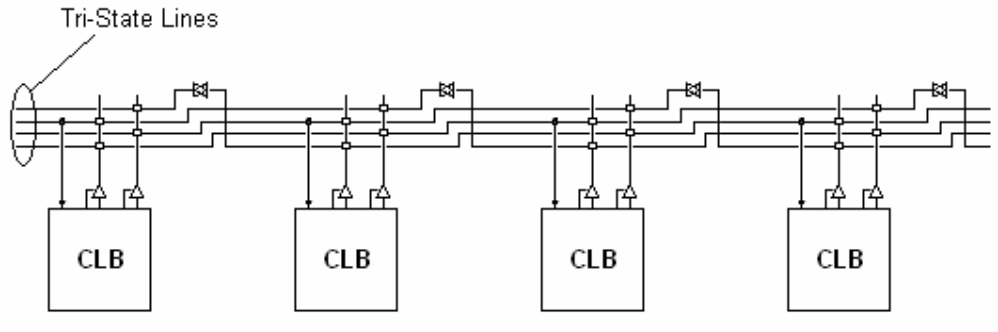

Figure 3-5: Horizontal Longlines that traverse all along the FPGA

- *Global Routings* are used for low skew and high fanout signals such as clock signals

3.2 CONFIGURATION ARCHITECTURE OF XILINX FPGAS

Xilinx FPGAs have SRAM based configuration memory, which provides unlimited reprogramming feature. The configuration file of a Xilinx device is called *bitstream*. A host device sends this bitstream file to one of the configuration ports of the FPGA. Then internal state machines of the FPGA device evaluate if the bitstream file has correct Cyclic Redundancy Check (CRC) value or not. If the CRC value is correct then it programs the configuration memory (SRAM) of the device with the bitstream data.

The configuration data of FPGA has divided into frames. A *frame* is the minimum segment of configuration memory that can be reconfigured. A frame includes configuration information of full height of device with one bit wide. Since a frame includes the configuration data of full height of the device, minimum reconfigurable unit must occupy full height of the device.

Since configuration bitstream is divided into frames in a column-based order, at least a column of CLBs can be reconfigured at the same time. Moreover, configuration information of one CLB column is stored on 48 frames on XCV50 device [32]. Therefore, reconfiguration of 48 frames is necessary to reconfigure a column of CLBs. The configuration memory structure of XCV50 device is shown in Figure 3-6.

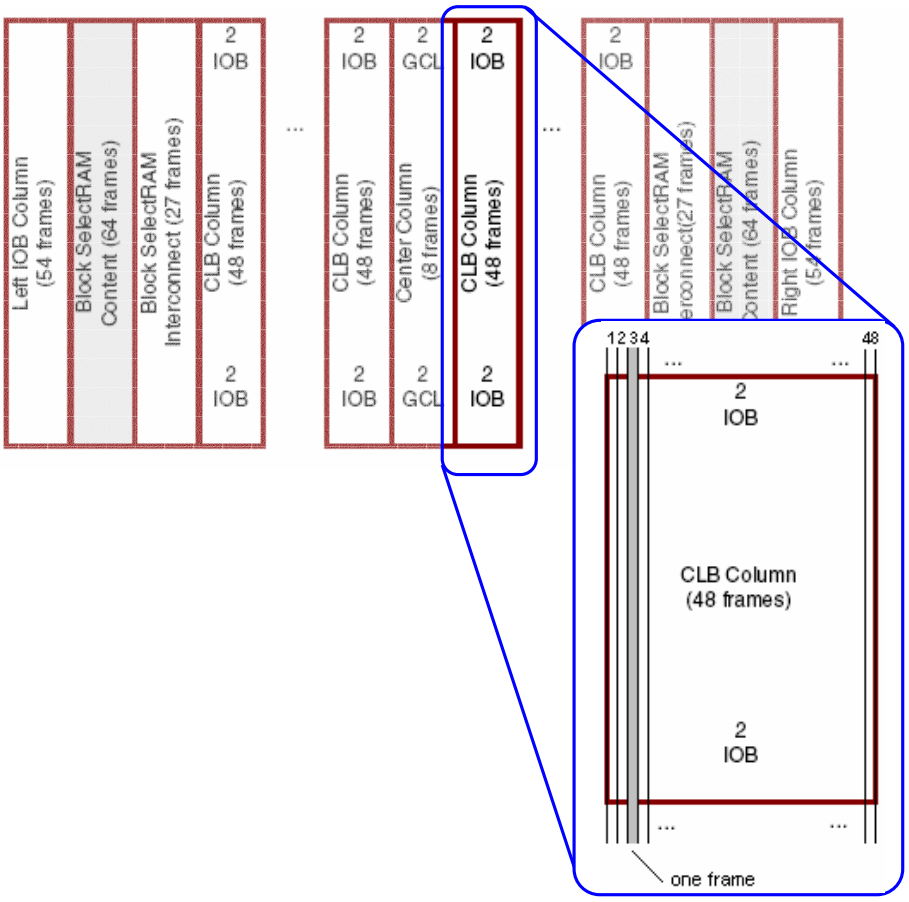

Figure 3-6: Configuration Columns and Frames of Xilinx XCV50 device

3.2.1 Column and Difference Based Reconfiguration

Xilinx FPGAs allows two types of partial reconfigurations; column and difference based reconfigurations. It is possible to reconfigure one or more columns of CLBs using column based reconfiguration flow. On the other hand, difference based reconfiguration allows small changes on the configuration data.

If boundary between two CLB column are defined strictly (i.e. no routing connection between) then reconfiguration of one column does not affect the other. By using this principle, *modules* that occupy integer multiple of CLB columns can be partially reconfigured. This type of reconfiguration is called column-based reconfiguration.

Another possibility for reconfiguration is making small changes on the configuration memory. Internal configurations of a CLB can be changed by reconfiguring them. For example, the function of Lookup Table inside a CLB may be changed from an OR gate to a AND gate. The bitstream generation tools will compare two different bitstreams and generate a bitstream that includes only different frames. The resulting bitstream will be much smaller than the original ones.

3.2.2 Glitchless Reconfiguration

"FPGA memory cells have glitchless transitions, when rewritten, the unmodified logic will continue to operate unaffected" [33]. This glitchless reconfiguration is required for communication channels that pass through from a reconfigurable module. Otherwise, reconfiguration of the module will break the communication channel and connection will be lost.

Glitchless reconfiguration property is supported on Spartan 2, Spartan 2E, Virtex, Virtex E, Virtex 2, Virtex 2 Pro, and Virtex 4 devices of Xilinx. Spartan 3 and Spartan 3E devices do not reconfigure without glitches [34].

3.2.3 Clocking Logic

Same clock can route to all partial modules. However, clocking logic (Clock Routing Paths, Clock IOB) is always separate from the reconfigurable module and clocks have separate bitstream frames [35]. As a result, reconfiguration of a module does not affect synchronous circuits on another module.

3.2.4 Suitable Configuration Options for Runtime Reconfiguration

Xilinx FPGA devices can be configured using different configuration interfaces [36]. These interfaces are

- Master / Slave Serial Mode,
- SelectMAP Interface,
- Boundary Scan (JTAG) port and

- Internal Configuration Access Port (ICAP).

Master Serial Mode is used to configure FPGA from a PROM device. SelectMAP is a parallel bus available at normal I/O pins of the FPGA. Boundary scan port is a standard test port that has dedicated pins on FPGA. ICAP is an internal port that is similar to the SelectMAP interface.

One of these configuration interfaces is selected at power-up according to the configuration mode pins, M0, M1, and M2. Because data pins of the configuration interface must be reserved to one of the interfaces at start-up. However, it is not necessary to make mode selection for boundary scan mode since it is always available for configuration independent of the mode selection [31]. ICAP also does not need any mode selection since it is an internal interface.

To make a runtime reconfigurable system using a Xilinx FPGA, a suitable configuration scheme must be constructed. FPGA must be configured initially and it must be reconfigured while initial configuration is operating on it. It is possible to use different configuration interfaces for these initial and run-time reconfigurations. However, not all of these methods are suitable for run-time reconfiguration. The methods suitable for run-time reconfiguration are

- SelectMAP Interface,
- Boundary Scan (JTAG) port and
- Internal Configuration Access Port (ICAP).

Note that, one of these modes is necessary for only runtime reconfiguration. Loading initial bitstream can be done by any method. For example, the initial bitstream can be loaded using a serial PROM then all reconfigurations can be done using ICAP port. As another example, loading initial bitstream and reconfiguration can be done using JTAG port.

Slave Parallel Mode (SelectMAP)

SelectMAP is a parallel bus, which is driven by an external device to program the FPGA. In normal operation, SelectMAP pins are left to the user after configuration as normal I/O pins. However, in a runtime reconfigurable system they must be always available as a SelectMAP interface to enable runtime reconfiguration. In order to achieve this, when creating bitstream with Xilinx

BitGen tool, *-g Persist:Yes* option must be used. This option ensures that the SelectMAP interface will remain active after first configuration.

Essential signals used for SelectMAP configuration port are given in Figure 3-7. Configuration data is sent or received through DATA pins synchronized with CCLK Clock. BUSY is used for handshaking and not necessary for low clock rates. CS is the Chip Select signal that enables the port for data transfers. WRITE is used to select the operation type, either as write or as read. PROG, INIT, and DONE signals are the SelectMAP protocol commands and acknowledgements such as “reset the configuration logic”, “verify successful operation” etc... More details about the SelectMAP protocol can be found on [37].

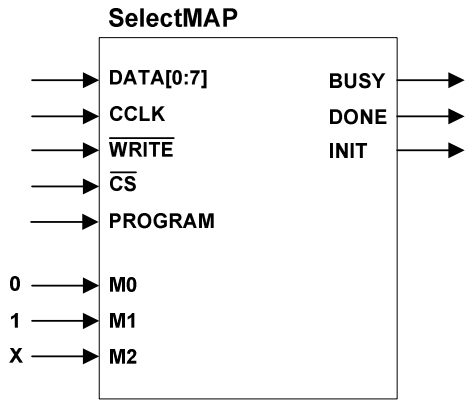

Figure 3-7: SelectMAP Configuration Signals on Xilinx FPGA

The main advantage of the SelectMAP interface is fast configuration opportunity it provides. It is possible to use a SelectMAP up to 50 MHz clock rates without handshaking (Virtex, Virtex-E, and Spartan-2). For Virtex-2, this frequency is 66 MHz [37]. Therefore, SelectMAP can provide bandwidths of higher than 500 Mbit/sec, since it is 8-bit parallel bus.

SelectMAP has also some shortcomings. It requires either an external controller or some parts of FPGA to control the bus. An external controller is an extra cost. When controller logic is implemented on the same FPGA, it limits the reconfigurable areas since controller must access to external pins. Furthermore, it occupies logic and BlockRAM sources, which can be necessary for the user.

Internal Configuration Access Port (ICAP)

Internal Configuration Access Port (ICAP) enables configuring FPGA from logic inside the fabric. It has same protocol with the SelectMAP configuration port. The only difference is the connection points, which are the internal routings on ICAP instead of I/O pins. Therefore, a logic mapped inside the device can reconfigure FPGA by writing configuration data to the ICAP. However, hardware communicating with ICAP port must not be reconfigured since communication can be lost after reconfiguration. Therefore, it is more suitable for partial reconfiguration instead of full reconfiguration [56].

ICAP is a very good solution for self-reconfiguration since it does not require any external hardware sources. It can take advantages of self-reconfiguration such as secure configuration and compressed bitstreams. Unfortunately, it is only available on newer Xilinx devices such as Virtex-II and Virtex-4 FPGAs.

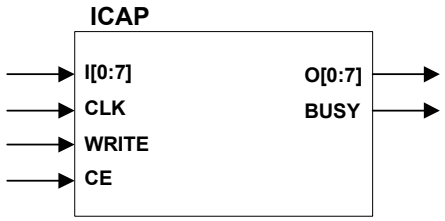

Figure 3-8: ICAP Configuration Signals on Xilinx FPGA

ICAP interface signals are shown in Figure 3-8. The functionalities of CLK, WRITE and BUSY signals are equivalent on ICAP and SelectMAP. In addition, CE has the same function with CS on SelectMAP. The only difference is the data bus, which is divided into two parts on ICAP. One part (I[0:7]) is used for writing configuration data to port, while the other part (O[0:7]) is used for reading back the configuration data.

Boundary Scan (JTAG) Mode

Joint Test Action Group (JTAG) designed a test standard and named JTAG for testing Printed Circuit Boards (PCB). This Boundary Scan architecture is

designed to test the physical connection of I/O pins at the board level. JTAG become a widely used test port with the increase of complicated PCB structures and smaller Integrated Circuits (ICs) [38]. Due to lots of benefits, it has become an IEEE standard (IEEE 1149.1). Most of current ICs contain a JTAG port pins to debug it. Its boundary scan architecture has a four-wire serial interface travels along all the pins of the device forming a chain. Serial data enters to the device with Test Data In (TDI) pin and stored on a shift (instruction) register. The data is send to the output of the device with Test Data Out (TDO) pin. All data shifting on JTAG chain are done with synchronized to Test Clock (TCK). The reserved pins for the JTAG port and their acronyms are listed in Table 3-1.

Table 3-1: JTAG Pins and their descriptions

Pin Name	Description
TDI	Test Data In
TDO	Test Data Out
TMS	Test Mode Select
TCK	Test Clock

JTAG also enables adding vendor specific instructions, instead of standard instructions. Vendors use these instructions to debug software/hardware inside the device. Furthermore, JTAG port can be used for on-board programming. All Xilinx FPGAs contain JTAG port, which enables configuration of the device with JTAG chain. Main advantage of using JTAG port is not wasting any user I/O for configuration since JTAG port has dedicated pins on the device. The JTAG pins and configuration selection is shown in Figure 3-9.

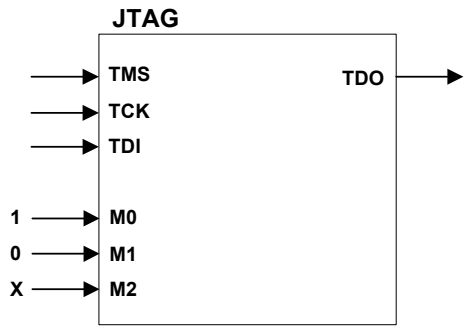

Figure 3-9: JTAG Configuration Signals on Xilinx FPGA

A disadvantage of JTAG Boundary Scan for runtime reconfiguration is high configuration time. Since it sends data from a serial line and PC adapters speed is low, it does not permit fast reconfigurations. Therefore, the selected case study for runtime reconfiguration does not focus on the speedup benefit of the runtime reconfiguration. Instead, it focuses on virtual hardware concept of the runtime reconfiguration.

Used Interface for the Designs

JTAG is used on described designs throughout the thesis. It is a straightforward method since no external pins are required other than test port connections. In addition, software tools are available for JTAG. The other methods require a board that left configuration pins to the user. Generally, PROM loading is provided on most of the commercial boards, which occupy the Data pin of the SelectMAP interface. Therefore, to use SelectMAP port a custom PCB must be designed which is out of the scope of this thesis. Instead, a prototyping board containing Xilinx-Spartan 2E FPGA with JTAG connection is bought to examine RTR.

3.3 CONVENTIONAL DESIGN FLOW FOR XILINX FPGAS

The standard design flow is normally implemented using graphical user interface (GUI) of Xilinx ISE software. The GUI takes the circuit information from the user as a HDL (i.e VHDL, Verilog etc…) or a schematic file. Using these files,

GUI can generate a bitstream to download FPGA device. However, some operations are executed on the back to create this bitstream. The flow of these operations is illustrated in Figure 3-10.

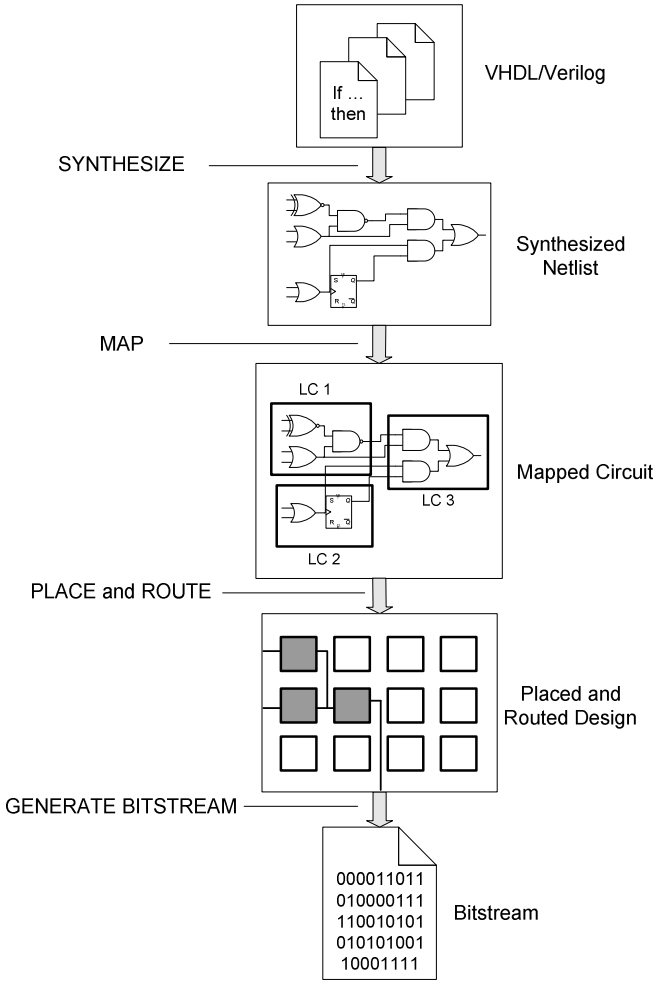

Figure 3-10 Standard Design Flow for an FPGA Design

If an HDL file is used, it is synthesized to create a *netlist*. A netlist contains logic elements and their connections (i.e. circuit description). With schematic files, the creation of a circuit netlist is a trivial issue. After obtaining netlist, the remaining operations are translation, mapping, placing-routing, and lastly creation of configuration file.

The circuit netlist and constraints are combined on a file with a translation operation (not shown in Figure 3-10). In the mapping phase, circuit is partitioned and elements are grouped to map Logic Cells (LCs). Afterward, these logic cells are placed and routed to the FPGA using CLBs, routing sources, IOBs etc…At the last step configuration information is extracted from the placed - routed design and written to the configuration file (i.e. to the bitstream).

The tools used for the operations of standard design flow are given in Table 3-2. Note that, these tools accept additional options that enable for different design flows. This feature is used in creation of runtime reconfigurable designs and explained in Chapters 4 and 5.

Table 3-2: Standard Design Flow Operations and Tools of Xilinx FPGAs

Operation	Used Xilinx Tool
Synthesis	XST
Translation	NgdBuild
Mapping	Map
Placing and Routing	PAR
Creating Bitstream	Bitgen

3.4 TOOLS FOR PARTIAL RECONFIGURATION OF XILINX FPGAS

3.4.1 XAPP290

XAPP290 is an application note published by Xilinx. It includes reference materials for a runtime reconfigurable design. One of the methods explained in this application note is used on designs explained in the thesis. More information about the contents of the application note can be found in Chapter 4.

3.4.2 JBITS

JBits is an Application Programming Interface (API) based on Java. It is developed by Xilinx. This API may be used to construct digital designs and parametrical cores that can be executed on Xilinx Virtex II FPGA devices. It runs on a Java enabled environment (usually a PC). Today it is only published for Virtex II but it can be extended to other devices in the future.

JBits can be used for runtime reconfigurable applications. The circuits can be configured on the fly by executing a Java application that communicates with the circuit board containing the Virtex II device. By using the XHWIF API, it is possible to download the design within the same Java application. This enables run-time configuration and reconfiguration of Virtex II device [39]. The design flow of runtime reconfiguration using JBits is shown in Figure 3-11.

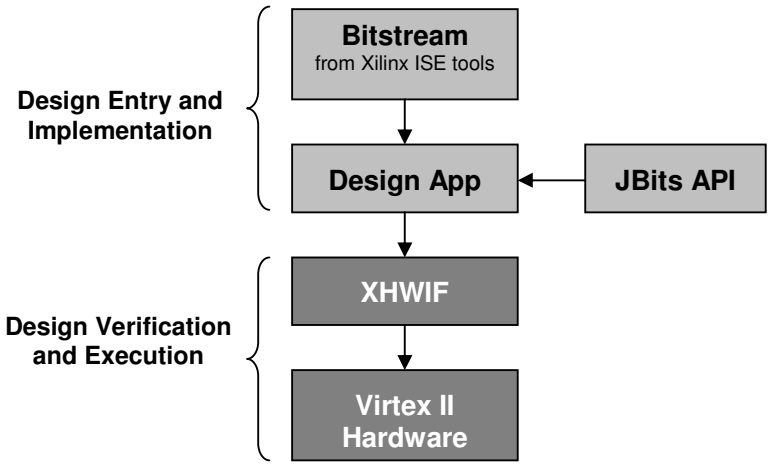

Figure 3-11: Design Flow of Runtime Reconfiguration using JBits [39]

The main steps involved in a JBits application are the object construction, reading bitstream from a .bit file, modifying the bitstream, and writing bitstream to a file again. This application flow on JBits is shown in Figure 3-12.

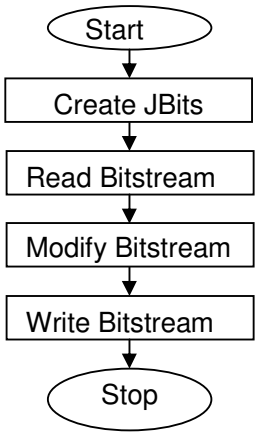

Figure 3-12: JBits Application Flow

An example code that modifies a bitstream is shown below:

"void JBits.setCLBBits(int row, int column, int[][] resource, int[] bits);
jbits.setCLBBits(clbRow, clbCol, F1_B0.F1_B0, F1_B0.X0) ;"

The disadvantage of the JBits is it is too low-level (it changes routing of the device, LUT configurations etc.). Designer must know the entire device architecture to modify bitstreams. Therefore, JBits remain as a research tool and it did not go further to implement complex designs.

CHAPTER IV

MODULE BASED PARTIAL RECONFIGURATION

Xilinx FPGAs supports runtime reconfiguration (RTR). Partial reconfiguration guidelines given in Xilinx Application Note 290 [35] must be considered to realize RTR on FPGA.

The application note covers two different RTR methods. One of them is suitable for making small changes on the logic implemented. This method is called *difference based partial reconfiguration*. In this method, small changes can be made on the design then only frames that have differences are loaded to the FPGA. The recommended changes are restricted to changing I/O standards, BlockRAM contents, and LUT programming. It is also possible to change routing information, however it is not recommended since contention may occur during reconfiguration. The other method is called *module based partial reconfiguration*. In this method, FPGA is divided into multiple columns, which are called modules. Then a new configuration can be loaded to a module while the other parts of the device are still active (working). All the designs in this thesis use module based partial reconfiguration.

In this chapter, the flow of module based partial reconfiguration for a Xilinx FPGA device will be explained. Experiments were done on Virtex-E and Spartan-2E device. Furthermore, to make things simpler an uncomplicated partial reconfigurable architecture was designed and implemented on Xilinx Spartan-2E device. Hence, examples given for the explanation of module based partial reconfigurable design are based on Spartan 2 200E device.

4.1 COLUMN BASED RECONFIGURATION

As mentioned before, Xilinx FPGAs give an opportunity that a column of Configurable Logic Blocks (CLBs) can be reconfigured by writing its belonging frames to a configuration port of the FPGA. This structure and additional features enable creating Runtime Reconfigurable (RTR) architecture on Xilinx FPGAs.

FPGA can be divided into multiple columns to make a RTR system. By using Xilinx map, place and route tools it is possible to generate bitstream for only one reconfigurable column. Then columns may be reconfigured by this bitstream while the other columns are still working. This reconfiguration operation is called *active partial reconfiguration* of FPGA.

In a module based partial reconfigurable design, reconfigurable modules communicate with other modules through *bus macros*. An example partial reconfigurable architecture with two reconfigurable modules is shown in Figure 4-1.

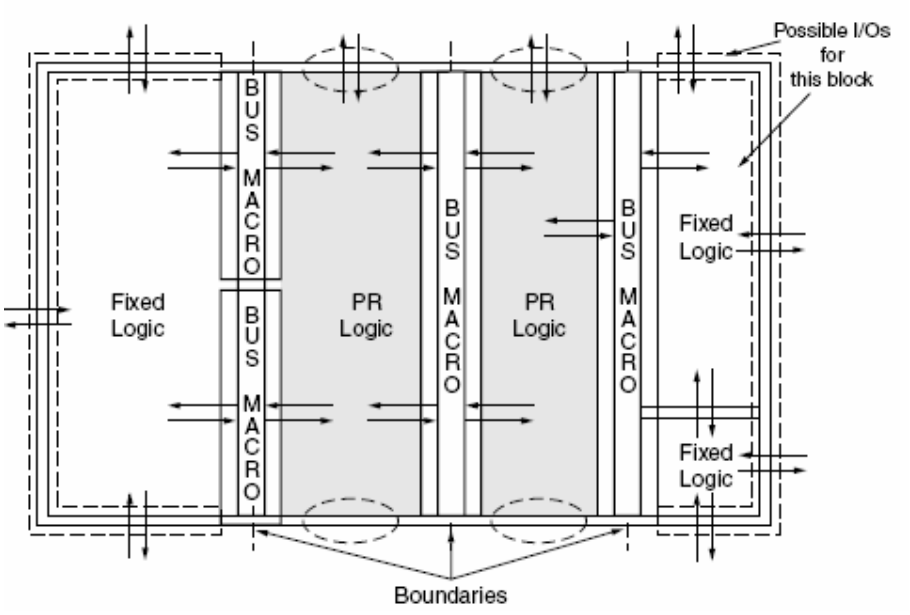

Figure 4-1: Design Layout with Two Reconfigurable Modules [35]

4.1.1 Restrictions of Partial Reconfigurable Design

Some restrictions for runtime partial reconfiguration are listed below **[35]**:

- A reconfigurable module must communicate with other modules only through bus macros.
- Module boundaries cannot be changed at runtime.
- Minimum width of a module must be four slices (2 CLB) width. Additionally modules width must be multiples of four slices.
- Height of a module is the full height of the device.
- A reconfigurable module can use Input Output Blocks (IOBs) that lies on its boundaries only.
- The leftmost slice on a module must be placed at multiple of four slices (i.e. slice numbers 0, 4, 8)

4.1.2 Bus Macros

Reconfigurable modules must communicate with other modules via *bus macros* as seen in Figure 4-2. Bus macros provide a fixed routing for signals that pass to other modules. Therefore, every different configuration of a module uses same path to share signals with other modules. Otherwise, communication will be broken with a reconfiguration.

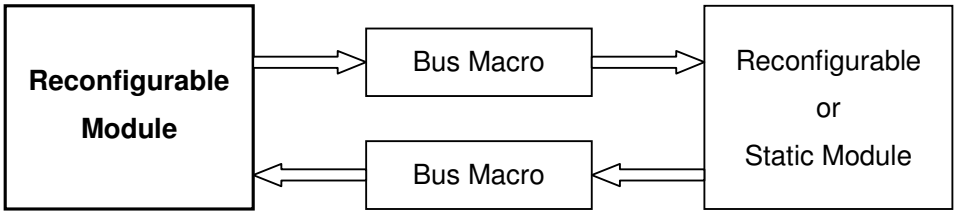

Figure 4-2: Communication with Reconfigurable Modules

Tri-state buffers and horizontal long lines are used to implement bus macros. Physical implementation of bus macro is shown in Figure 4-3. .LO[3:0] and RO [3:0] are the horizontal tri-state long lines and used for tri-state signals.

LI[3:0] can drive these long lines if LT[3:0] enable signals are hold active. Also RI[3:0] can drive these long lines if RT[3:0] enable signals are hold active.

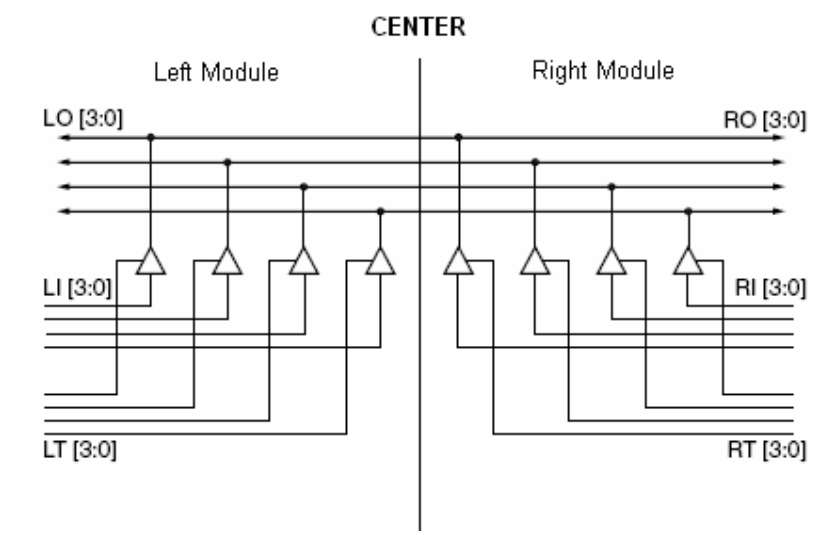

Figure 4-3: Bus Macro connecting two adjacent modules [35]

At an instance, only one side must become active to drive bus to prevent contention (i.e. only RT or LT becomes active at the same time). It is also suggested that a bus macro must be used only in one direction (i.e. not bidirectional). In addition, its direction must not change by reconfiguration.

4.1.3 Clocking Logic

As mentioned before, clocking logic is independent from reconfiguration processes, which enables runtime-reconfiguration on Xilinx-FPGA. Therefore, if a synchronous circuit is used in the design, clock must be distributed from the Global Routing lines. Otherwise, clocks will be disturbed with reconfiguration of a module.

4.2 IMPLEMENTED SIMPLE PARTIAL RECONFIGURABLE ARCHITECTURE

To make module based partial reconfiguration more realistic and to gain experience it is applied on a simple design. In this design, simple arithmetic calculations are performed inside the FPGA. The operands are taken from the outside and the result is given to the external world again. The logic that makes arithmetic operations is changed at runtime by partial reconfiguration of the FPGA. Therefore, different arithmetic operations are done with same logic resources at different times by changing configuration data of the FPGA.

In the designed architecture, there is only one reconfigurable module and one static module. Left side of FPGA is used as reconfigurable module while the other side is used as static module. The data flow between the static and reconfigurable modules are done through bus macros. Basic structure of the designed architecture is shown in Figure 4-4

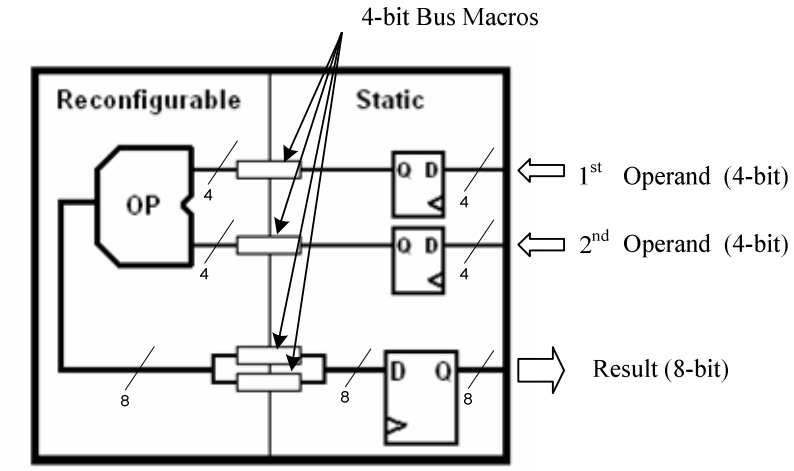

Figure 4-4: Basic Structure of Reconfigurable Design

Static module only stores input operands and the result of the arithmetic operation. Reconfigurable module is used as Arithmetic Logic Unit (ALU) that has three different configurations. These configurations are used to implement

different 4-bit arithmetic operations. The operations are addition, multiplication, and subtraction. The mapping of these alternative configurations on the FPGA is shown in Figure 4-5:

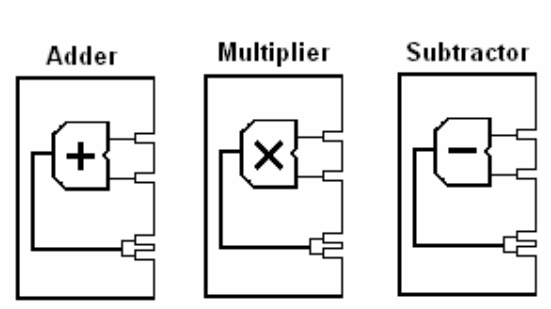

Figure 4-5: Alternative Configurations for Reconfigurable Module

4.3 XILINX TOOLS AND IMPLEMENTATION

To implement a partial reconfigurable design, Xilinx ISE 6.3i tools are used and Xilinx Modular Design Flow [40] is followed. In addition, the restrictions given in Xilinx application note [35] are also taken into the consideration.

4.3.1 Modular Design Flow Overview

Modular design has the following main steps:

- Modular Design Entry and Synthesis
- Modular Design Implementation
  - Initial Budgeting Phase
  - Active Module Implementation Phase
  - Assemble Phase

Figure 4-6 shows flow of these steps.

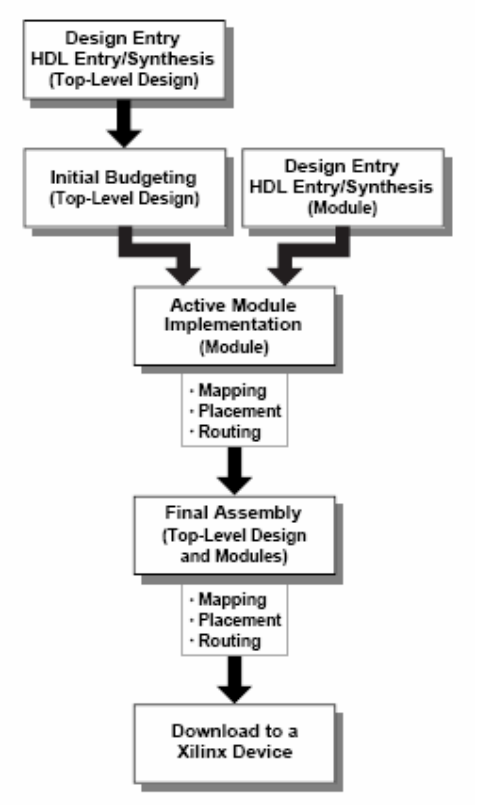

Figure 4-6: Modular Design Flow Overview [40]

Different steps of the flow are implemented on different folders. Implementing design with a good directory structure is important because some files have same name but used for different purposes. The directory structure used in the design is given in Figure 4-7. It is similar to given directory structure in [41]. Note that, all the files used in this design are given in Appendix E (Reconfig-ALU directory and its subdirectories include necessary files).

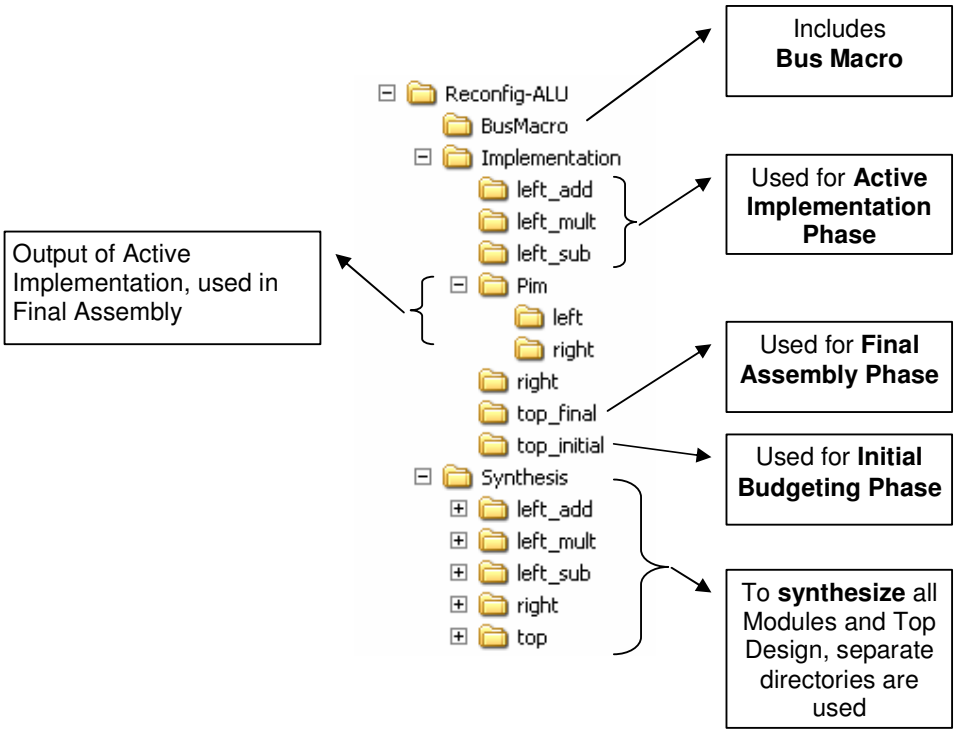

Figure 4-7: Directory Structure Used For A Module Based Partial Reconfigurable Design

- *Synthesis* directory is used to create a netlists (.ngc file) from VHDL designs (.vhd file) of modules.
- The bus macro that will be used in design is put on the *BusMacro* directory. It is taken from the Xilinx application note files [35]. However, some corrections are done in these files for Virtex-E device (will be explained in Section 4.4)
- Three Implementation phases of the Modular Design Flow is done in the *Implementation* directory

Some important file extensions used for reconfigurable design and their descriptions are given in Table 4-1.

Table 4-1: Descriptions of Files that are used for Module Based Partial Reconfiguration

File Extension	Constructed by Program	Description	Used by
.vhd	User	Contains hardware description of the design.	XST (Xilinx synthesis tool)
.ucf	User / Constraints Editor	The User Constraints File (UCF) is an ASCII file that contains timing and layout constraints that affect how the logical design is implemented in the target device.	Ngdbuild command line program
.nmc	N/A	It contains the definition of a physical macro (hard placed and routed macro). In this design it is used as a bus macro.	Ngdbuild command line program
.ngc	XST (Xilinx synthesis tool)	Output of. Synthesized module contains Netlist of the circuit.	Ngdbuild command line program
.ngd	Ngdbuild command line program	Contains both a logical description of the design reduced to Xilinx Native Generic Database (NGD) primitives and a description in terms of the original hierarchy expressed in the input netlist.	MAP command line program
.ncd	MAP command line program	The Native Circuit Description (NCD) is physical representation of the design mapped to the components in the Xilinx FPGA.	PAR command line program
	PAR command line program	PAR command line program takes an NCD file as input, places and routes the design, and outputs an NCD file.	BitGen command line program

Table 4-1 cont'd: Descriptions of Files that are used for Module Based Partial Reconfiguration

.bit	BitGen command line program	The bitstream that is used to load configuration to the FPGA	To load device
.bld	Ngdbuild command line program	The output report of the Ngdbuild command line program. Contains errors, warnings and information.	User
.mrp	MAP command line program	The output report of the MAP command line program. Contains errors, warnings and information.	User
.par	PAR command line program	The output report of the PAR command line program. Contains errors, warnings and information.	User

4.3.2 Module Entry and Synthesis

In the *Module Entry and Synthesis* phase, necessary circuit netlists are created. VHDL is used to describe logic functions and they are synthesized using Xilinx ISE 6.3i XST. There are five different VHDL files. These are

- Top.vhd,
- Right.vhd,
- Left.vhd (Three left.vhd files with different contents are used to implement adder, multiplier, and subtractor) files.

For Top, Right, Left Adder, Left Multiplier and Left Subtractor modules separate projects are created. "Add IO Buffers" option is selected in the "Xilinx Specific Options" tab when synthesizing *top* module, for other modules, this option is deselected. Therefore, I/O buffers are only added to the Top module, which is also done for non-reconfigurable designs.

Also for all modules, Bus Delimiter option is selected as <>. Note that <> sign is called as angle delimiter. Therefore, in the following steps angle delimiter bus macro that is given by Xilinx will be used.

After synthesizing, files with ngc extensions are created. These netlist files will be used in the implementation.

4.3.3 Implementation

In the implementation flow, an initial configuration bitstream that configure whole FPGA will be generated. Also for every different configuration of each reconfigurable module, a partial bitstream will be generated.

In this example, two modules lie on FPGA. One of them is reconfigurable while the other is static. In addition, there will be three different configurations (Adder, multiplier, and subtractor) for the reconfigurable module. In summary, three partial bitstreams for reconfigurable part and one full bitstream for the whole device will be generated.

Implementation flow has following three phases. These are

- Initial Budgeting Phase,
- Active Module Implementation Phase and
- Assemble Phase.

General overview of the flow is shown in Figure 4-8 and Figure 4-9.

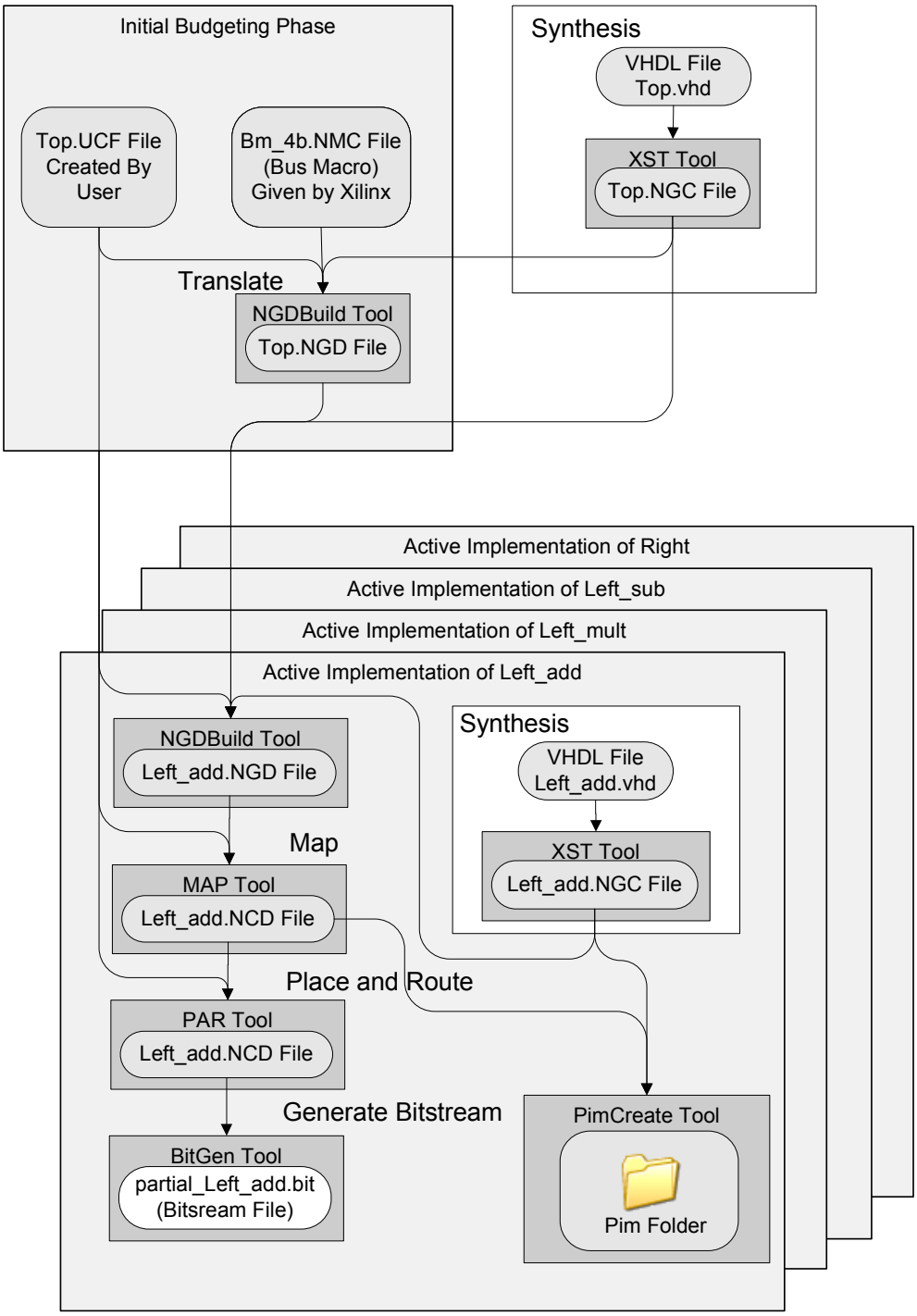

Figure 4-8: Initial Budgeting and Active Implementation Phases of Module Based Partial Reconfiguration Flow.

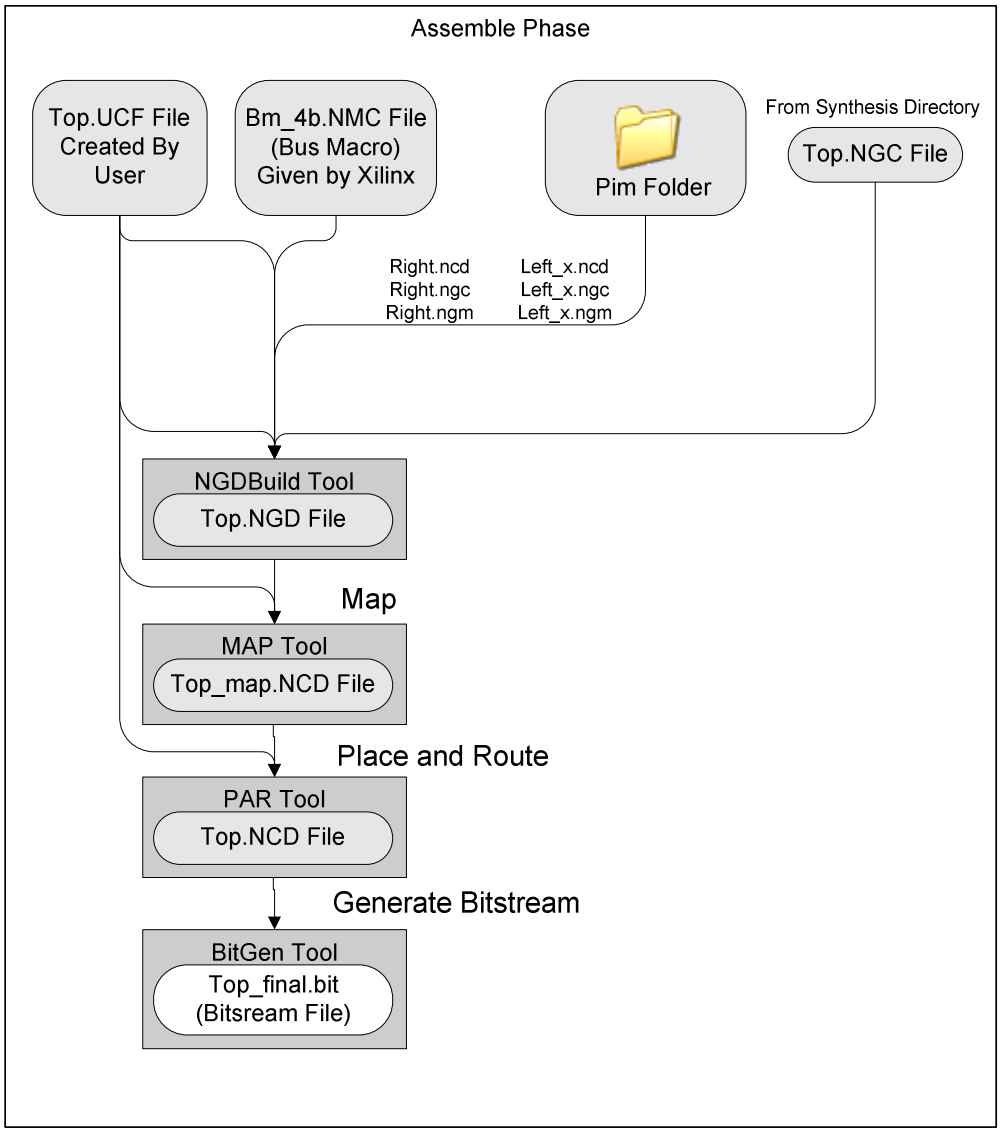

Figure 4-9: Assemble Phase of Module Based Partial Reconfiguration Flow.

4.3.3.1 Initial Budgeting Phase

At the initial budgeting phase, a floor plan and constraints are created for the overall design. A user constraint file is created and used with NgdBuild tool to annotate constraints to the synthesized top design file.

Creating a User Constraint File

Top-level user constraint file contains the following information:

- Physical assignment of pin locations,
- Module area boundaries and
- Locked components (bus macros, LUTs for VCC-GND and Clock buffer).

To adjust module area boundaries and pin locations Xilinx PACE or FloorPlanner tools were used. Instead of using these tools, all constraints can be entered manually with a text editor. However, entering area constraints by graphical interface for the first time and manipulating them manually is much more suitable.

To run PACE and FloorPlanner tools for the design translated file (.ngd) is needed. Therefore, NGD build must be run to obtain a temporary ngd file. The following code is run with Command Prompt in the synthesis directory:

ngdbuild -modular initial top.ngc

Note that, top.ngc is obtained from the synthesis of the top.vhd with XST. The result of this command will be a top.ngd file. Temporarily this ngd file will be used to insert constraints to the ucf file with PACE editor.

Assigning Area Constraints

PACE is opened with the top.ngd file. On the PACE, using logic Tab, Left module is selected. Then using Tools → Assign Area Constraint Mode, an area constraint is drawn. Same operation is done for the Right module. Note that, these constraints can also be entered by using FloorPlanner. Figure 4-10 shows the boundaries of Right and Left Module after adjustment on PACE.

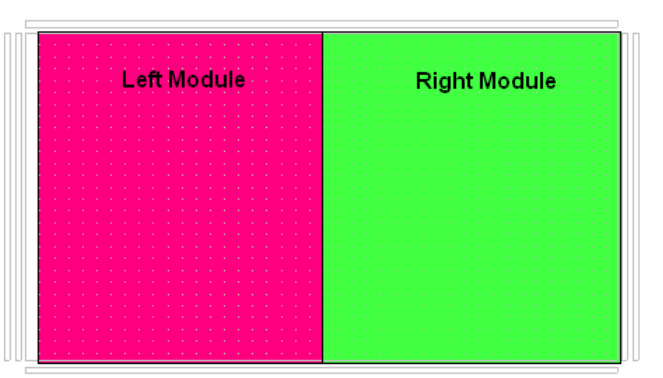

Figure 4-10: Constrained Areas for Modules as seen on PACE

After saving user constraint file, the following constraints are added by PACE:

```
AREA_GROUP "AG_left_module" RANGE = CLB_R1C1:CLB_R28C21 ;
AREA_GROUP "AG_left_module" RANGE = TBUF_R1C1: TBUF_R28C21 ;
INST "left_module" AREA_GROUP = "AG_left_module" ;

AREA_GROUP "AG_right_module" RANGE = CLB_R1C22:CLB_R28C42 ;
AREA_GROUP "AG_right_module" RANGE = TBUF_R1C22:TBUF_R28C42 ;
INST "right_module" AREA_GROUP = "AG_right_module" ;
```

In order to make these modules reconfigurable, RECONFIG property must be added to area constraints. This can be done by manually adding the following constraints to the user constraint file:

```
AREA_GROUP "AG_left_module" MODE = RECONFIG ;
AREA_GROUP "AG_right_module" MODE = RECONFIG ;
```

Bus Macro Placement

Actually, bus macro is a type of hard macro routed on FPGA. A hard macro is a placement constraint for a component. After giving the origin of a hard macro, place and route tools fix the position of logics that belongs to the hard macro.

Dedicated horizontal lines, which have connections to the tri-state buffers (TBUFs) are used to implement bus macros. In each CLB, there are two TBUFs

as shown in Figure 4-11. The upper one is called TBUF 0; the lower one is called TBUF 1.

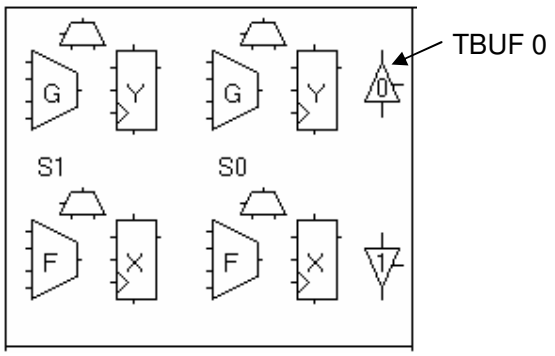

Figure 4-11: Configurable Logic Block (CLB) Contents

A bus macro occupies one row by eight columns of CLBs. The origin of a bus macro is the upper TBUF in the leftmost CLB. The placement and origin of a bus macro is shown in Figure 4-12.

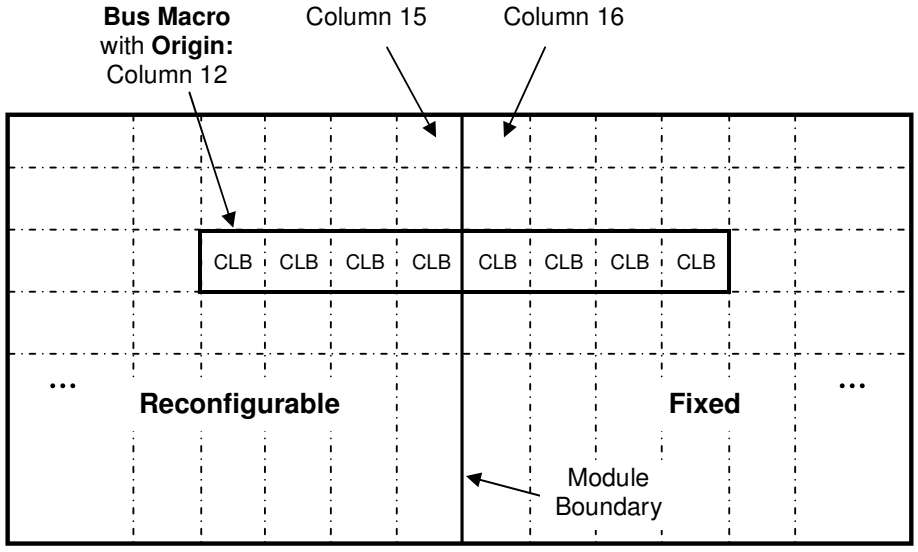

Figure 4-12: Bus Macro placement on FPGA

An example statement in the ucf constraint file to define the origin of the bus macro is the following:

```
INST “busmacroname” LOC = “TBUF_R1C1.0”
```

The position of bus macro is defined by using TBUF’s (tristate buffer) location. In the example R1 means “Row 1”, C1 means “Column 1” and .0 means the upper TBUF (or TBUF 0). In the following figure, a CLB and its TBUF 0 are shown.

The bus macro is placed between two modules. Therefore, the origin of the bus macro must be defined four columns before the intersection of two modules. For example, if the intersection of modules is between column 15 and 16, origin of the bus macro will be on column 12 (16-4) as shown in the following figure.

According to these requirements, the bus macros places are locked with the following constraints: (they are entered manually to the top.ucf file):

```
INST "bus_righttoleft1" LOC = "TBUF_R9C18.0"  ;
INST "bus_righttoleft2" LOC = "TBUF_R10C18.0"  ;
INST "bus_lefttoright1" LOC = "TBUF_R11C18.0"  ;
INST "bus_lefttoright2" LOC = "TBUF_R12C18.0"  ;
```

The following constraints are also entered manually to lock the place of the LUTs used for VCC and GND for each module (Reason for adding VCC and GND will be explained in Section 4.3.4).

```
INST "Internal_Gnd_Left" AREA_GROUP = "AG_left_module" ;
INST "Internal_Vcc_Left" AREA_GROUP = "AG_left_module" ;
INST "Internal_Gnd_Right" AREA_GROUP = "AG_right_module"  ;
INST "Internal_Vcc_Right" AREA_GROUP = "AG_right_module"  ;
```

Initial Budgeting Phase Batch File

Top_initial directory is used for the initial budgeting phase. Created constraint file top.ucf, bm_4b.nmc (in the BusMacro direcory) and synthesized design top.ngc are copied to this directory. Then top.ngd is created with the following ngdbuild command:

ngdbuild *-p xc2s200e-pq208-7* ***-modular initial*** *-uc top.ucf top.ngc*

–uc option ensures that the constraints from the top.ucf file are annotated to the top.ngd file.

-p xc2s200e-pq208-7 option instructs ngdbuild that the device is Xilinx Spartan-2 200E, package is pq208 and speed grade is -7.

4.3.3.2 Active Module Implementation Phase

In this step each of the modules are implemented separately, using top-level constraints. Partial bitstreams are generated for all reconfigurable modules (Left_mult, Left_sub, Left_add) and static module (right) as illustrated in Figure 4-13

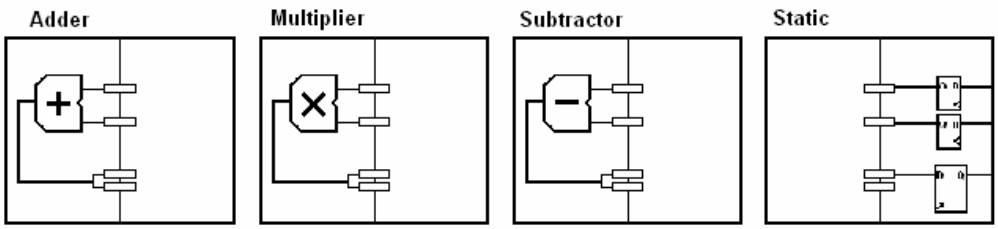

Figure 4-13: Partial Bitstreams for Reconfigurable Modules and Static Module

For all modules, Top.ucf file and associated synthesized .ngc file are copied to the module directories in the implementation directory.

Then for all modules NgdBuild, MAP, PAR and Bitgen commands are executed successively. Also for all modules pimcreate is executed to publish routed and mapped partial module design. Published files are put on Physically Implemented Modules (PIM) folder. Then they will be used in the final assembly phase.

As an example, the following commands are executed for adder configuration of the left module.

- ***ngdbuild*** *-p xc2s200e-pq208-7* ***-modular module -active*** *left ..\top_initial\top.ngc :* NgdBuild is run for the active module implementation phase and uses top.ucf file in working directory and synthesized netlists left.ngc, top.ngc. It creates top.ngd file.

- ***map*** *-pr b top.ngd -o top_map.ncd top.pcf* : MAP takes top.ngd maps the design and creates top_map.ncd

  –pr b option specifies that flip-flops or latches may be packed into input and output registers.

  –o option specifies the name of the output NCD file for the design (top_map.ncd).

- ***par*** *-w top_map.ncd top.ncd top.pcf* : PAR takes mapped design, then it place and route the design and outputs top.ncd file

  –w option instructs PAR to overwrite existing output file top.ncd.

- ***bitgen*** *-d -g ActiveReconfig:yes top.ncd partial_leftadd.bit* : bitgen takes top.ncd as input and produces partial bitstream *partial_leftadd.bit*

  -d option instructs bitGen not to run DRC (Design Rule Check).

  -g ActiveReconfig:Yes switch is required for active partial reconfiguration, meaning that the device remains in full operation while the new partial bitstream is being downloaded.

- ***pimcreate*** *-ncd top.ncd -ngm top_map.ngm ..\Pim :* PimCreate process "publishes" the routed design (and associated files) to the Pim (Physically Implemented Modules) folder.

After these steps, *FPGA-Editor* tool of Xilinx is used to inspect visually whether an unexpected error occurred in the routed design, top.ncd. One possible error is module that does not route at its own boundaries.

4.3.3.3 Final Assembly Phase

In the final assembly phase, partially routed and placed modules are combined to obtain a complete FPGA design. These partial design files are taken from the Pims directory and used for map, place, and route operations to create a full FPGA design.

Only one complete assemble is done. It includes left adder and right modules (another two possibilities are combining left multiplier and right modules or combining left subtractor and right modules).

Top_final directory is used for the final assemble phase. Top.ucf file created in the initial budgeting phase, bm_4b.nmc (in the BusMacro directory) and synthesized design top.ngc are copied to the top _final directory.

After copying files, the following commands are executed successively:

- ***ngdbuild*** *-p xcv100e-pq240-7* ***-modular assemble*** *-uc top.ucf -pimpath ..\Pim -use_pim incrementer -use_pim myRegister top.ngc :* NgdBuild is run for the final assemble phase and uses top.ucf file in working directory, synthesized netlist (top.ngc) and published files in the pims directory. It creates top.ngd file.

- ***map*** *–pr b top.ngd -o top_map.ncd* : MAP takes top.ngd as input, then maps the design and creates top_map.ncd

- ***par*** *-w top_map.ncd top.ncd* : PAR takes mapped design as input, then it place and route the design and outputs top.ncd file

- ***bitgen*** *-w top.ncd top_final.bit :* bitgen takes top.ncd as input and produces partial bitstream top_final.bit

After these steps, *FPGA_editor* tool of Xilinx is used to visually inspect if an error occurred or not in the routed design, top.ncd. In other words, it is checked if modules remain in their own boundaries.

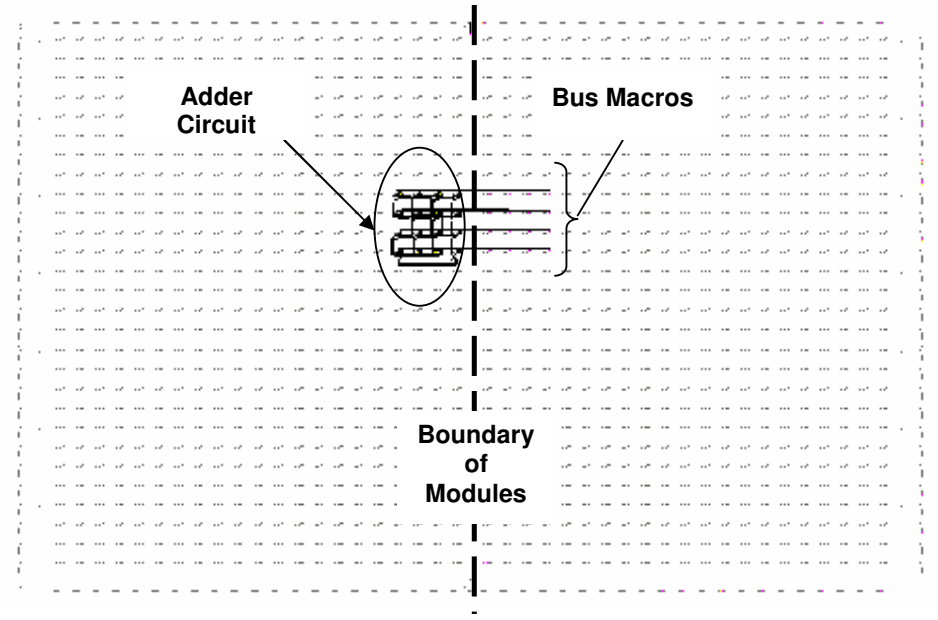

Figure 4-14: Placement of an Adder Circuit and Bus Macro on FPGA

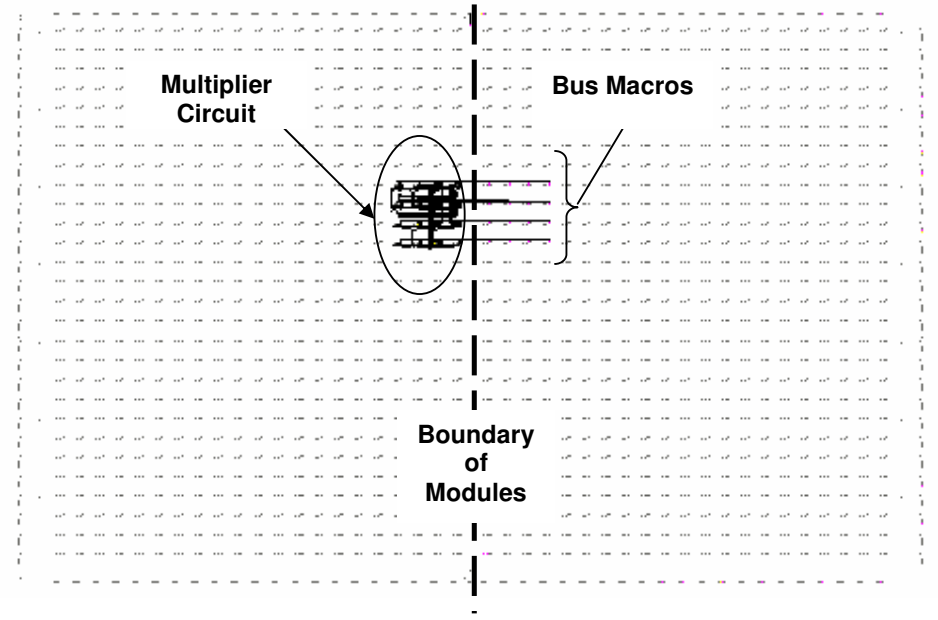

Figure 4-15: Placement of a Multiplier Circuit and Bus Macro on the FPGA

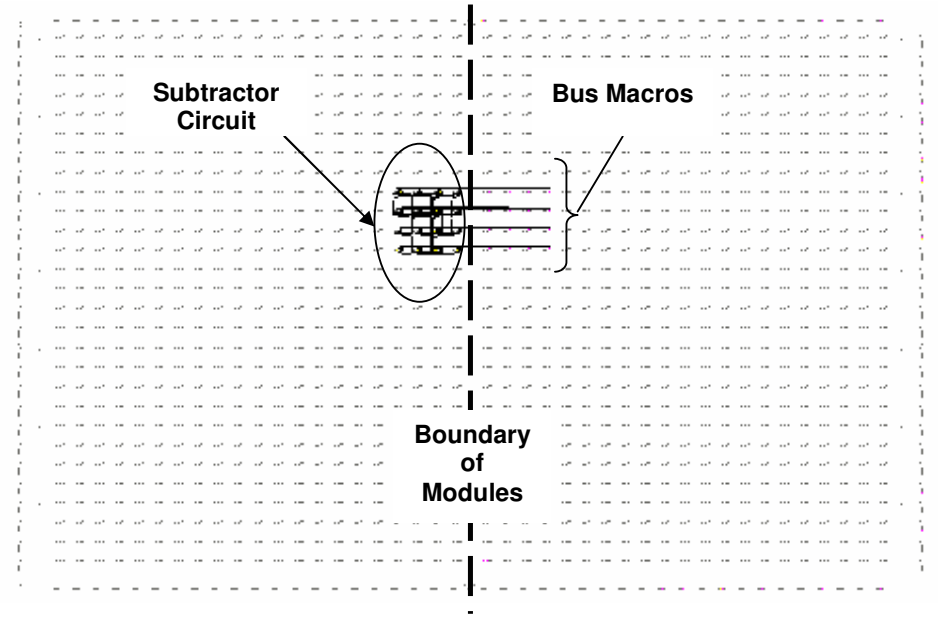

Figure 4-16: Placement of an Subtractor Circuit and Bus Macro on the FPGA

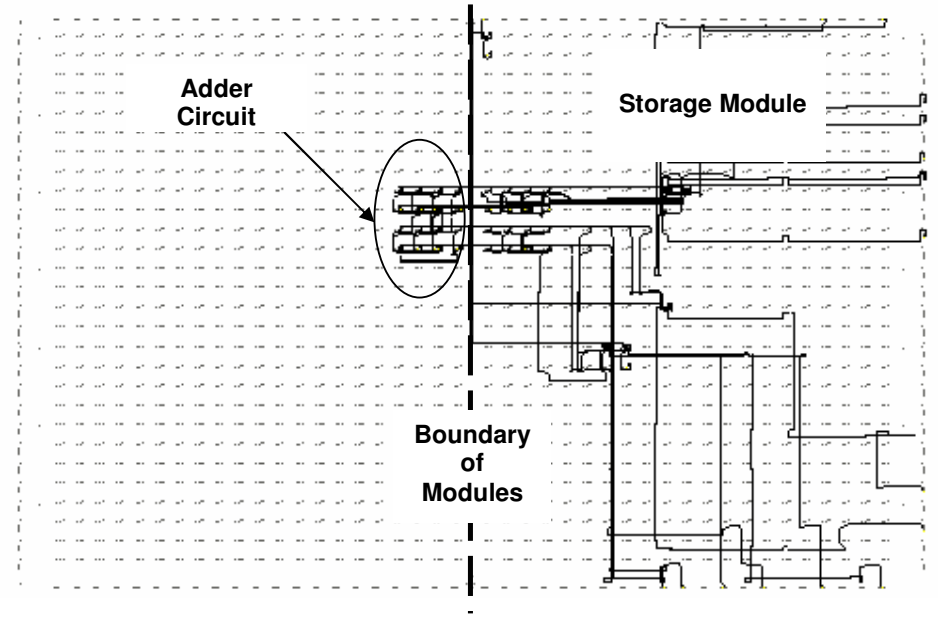

Figure 4-17: Final Layout of the Circuit on the FPGA with Adder Module on the Left Side

4.3.4 Creating “Logic 0” and “Logic 1”s

Reconfigurable modules can connect with each other only through bus macros. In addition, as explained before the direction of bus macros are adjusted by using LT or RT enable ports of the bus macro. These ports are driven by “logic 1” (VCC) and “logic 0” (Ground), therefore one direction is selected for bus macro. These VCC and Ground must also be alive during reconfiguration of a module.

In addition, it is forbidden to use same constant “logic 1” (VCC) and “logic 0” (Ground) signals on different reconfigurable modules. The reason is that it can cause a problem on module that shares these signals with another module while it is reconfiguring. Therefore, instead of sharing “logic 1” and “logic 0” signals, they must be given to the modules separately.

These limitations forces a module to have it is own VCC and Ground signals and must be always available (even if it is reconfiguring). There are two methods for getting “logic 1” and “logic 0” signals to the modules. First one is getting these signals from the outside world by using FPGA pins [41]. The second method is creating dummy Look-up Tables (LUTs) for each module and getting “logic 1” and “logic 0” signals from them [42]. The second method is used in this design.

1-bit LUTs are used and LUT functions are selected so that whatever the input is one LUT creates “logic 0” and another LUT creates “logic 1”. The truth tables of the LUTs are given in Table 4-2 .

Table 4-2: Truth Tables of Dummy Look Up Tables

Truth Table of LUT 0	
Input	**Output**
0	**0**
1	**0**

Truth Table of LUT 1	
Input	**Output**
0	**1**
1	**1**

Two LUTs are connected to each other in order to create dummy inputs. LUT connections are shown in Figure 4-18. LUT on the left side creates “logic 0” and LUT on the right side creates “logic 1”.

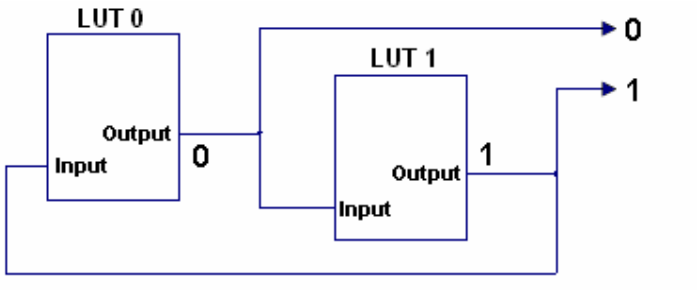

Figure 4-18: Dummy LUTs for creating "Logic 1" and "Logic 0"

The same structure is used for each module (fixed and reconfigurable). The added VHDL codes to the top.vhd file for generating left module's logic 1 and 0 are the following:

```
-- Fake Gnd and Fake Vcc of Left Module
Internal_Gnd_Left: LUT1
      generic map (INIT => b"00")
      port map    (O => Gnd_Left, I0 => Vcc_Left);

Internal_Vcc_Left: LUT1
      generic map (INIT => b"11")
      port map    (O => Vcc_Left, I0 => Gnd_Left);
```

It must be guaranteed that these elements stay inside the corresponding module area. Otherwise, reconfiguration of one module can disturb the outputs of LUTs (this can result in a contention on the bus macro). To overcome the problem LUTs are locked into the modules region. The following constraints are added to the top.ucf file to lock these LUTs:

```
INST "Internal_Gnd_Left" AREA_GROUP = "AG_left_module" ;
INST "Internal_Vcc_Left" AREA_GROUP = "AG_left_module" ;
INST "Internal_Gnd_Right" AREA_GROUP = "AG_right_module" ;
INST "Internal_Vcc_Right" AREA_GROUP = "AG_right_module" ;
```

4.4 ENCOUNTERED PROBLEMS AND SOLUTIONS

This design has been tested on Spartan 2 200E device however; initial test was done on Virtex - 100E. The bus macro provided by Xilinx for Virtex - E device

contains some errors. The errors are corrected to achieve partial reconfiguration on Virtex – E device.

4.4.1 Bus Macro Error and Its Solution

NgdBuild gives the following warnings at the initial budgeting phase:

WARNING:PhysSimExpander:5 - TBUF symbol `t4H_<0>': The following pins were connected on the outside of block "t4H_<0>" but left unconnected within the block: T

WARNING:PhysSimExpander:5 - TBUF symbol `t3G_<1>': The following pins were connected on the outside of block "t3G_<1>" but left unconnected within the block: T

....

These warnings cause the mapping tool (MAP) to fail and give some errors. The reason for these warnings is the error in the bus macro file of the Virtex-E family. The bus macros with extensions nmc must be converted to xdl files in order to make them editable (with a text editor). To convert a command line utility of Xilinx is used. For example, the bus macro bm_4b_ve.nmc is converted to bm_4b_ve.xdl by the following command:

```
xdl -ncd2xdl bm_4b_ve.nmc
```

Converted xdl file is opened with a text editor and there are some lines in the xdl file problematic such as

```
inst "t4H_<0>" "TBUF" , placed R1C16 TBUF_R1C16.1 ,
   cfg "TMUX::0 IMUX::I _SUPERBEL::TRUE";
inst "t3G_<1>" "TBUF" , placed R1C15 TBUF_R1C15.0 ,
   cfg "TMUX::0 IMUX::I _SUPERBEL::TRUE";
```

.....

In these lines

```
cfg "TMUX::0 IMUX::I _SUPERBEL::TRUE"
```

is changed to

```
cfg "TMUX::T IMUX::I _SUPERBEL::TRUE"
```

Explained changes are done on the .xdl file and its again converted to nmc file by the following command:

```
xdl -xdl2ncd bm_4b_ve.xdl
```

This command creates bm_4b_ve.ncd . It is renamed as bm_4b_ve.nmc and again used in the modular reconfiguration flow. This operation removed the warnings in the initial budgeting phase and the mapping errors in the active module phase of partial reconfiguration flow.

4.4.2 Second Bus Macro Error and Its Solution

When the PAR is run for the active module implementation phase of the static module it gives some errors as the following:

```
ERROR:DesignRules:576 - Netcheck: The signal dataR2<1> has a
sigpin on the comp busRegToInc_bus1/t3G_<1> that is not in the
same route area as another sigpin of the same signal. This is not
permited for Modules in partial reconfiguration mode unless the
signal has the property IS_BUS_MACRO.
ERROR:DesignRules:9 - Netcheck: The signal "dataR2<1>" is
only partially routed.
```

Again, the bus macro has problems that need to be corrected. The problematic lines in the converted bus macro file (.xdl) are the following

```
net "TNET<3>" ,
   outpin "t1A_<3>"          O                  ,
   outpin "t1E_<3>"          O                  ,
   pip R1C9 TBUF_OUT0 -> TBUF2 ,
   pip R1C10 TBUF3 =- TBUF_STUB3 ,
   pip R1C13 TBUF_OUT0 -> TBUF2 ,
   # net "TNET<3>" loads=0 drivers=2 pips=3 rtpips=0   ;
```

is changed to:

```
net "TNET<3>" ,
   cfg "_NET_PROP::IS_BUS_MACRO:" ,
   outpin "t1A_<3>"          O                  ,
   outpin "t1E_<3>"          O                  ,
   pip R1C13 TBUF_OUT0 -> TBUF2 ,
   pip R1C10 TBUF3 =- TBUF_STUB3 ,
   pip R1C9 TBUF_OUT0 -> TBUF2 ,
   # net "TNET<3>" loads=0 drivers=2 pips=3 rtpips=0   ;
```

For all nets (net "TNET<3>" , net "TNET<2>" , net "TNET<1>" , net "TNET<0>") the same correction is done. In other words cfg "_NET_PROP::IS_BUS_MACRO:" property is added.

Then bm_4b_ve.xdl file is again converted to bm_4b_ve.nmc file as in the solution of previous error.

CHAPTER V

A TMR SYSTEM ON A RUNTIME RECONFIGURABLE ARCHITECTURE

As a case study for partial reconfiguration, a fault tolerant system is designed to run on a reconfigurable FPGA. It is based on Triple Modular Redundancy (TMR). Runtime reconfiguration property of the FPGA is used to repair modules of the TMR.

The system can eliminate transient faults on routing lines and logical elements of the FPGA. It can also mask the faults encountered on logical elements by replacing them with non-faulty elements. The work does not address permanent faults on routing lines of the FPGA.

In this chapter, designed architecture will be explained. For the sake of completeness, basic terms about the fault tolerance will be given. Moreover, related works about the fault tolerance of FPGAs will be discussed.

5.1 BACKGROUND

Reliability is an important issue for mission and safety critical applications. A high reliability must be maintained on systems where a failure can cost lives and money. For instance, a breakdown of a satellite is unacceptable where total system cost takes a few billions of dollars. Similarly, brake system of an automobile must be highly reliable where people lives depend on. Hence, designers of such systems must take into consideration the faults on hardware that can arise during operation.

A reliable hardware environment is designed in this work, which is mainly built on a run-time reconfigurable FPGA. The fault types seen on such FPGAs and recovery methods of them are discussed in this section.

5.1.1 Fault Tolerance

A fault tolerant system can continue to operate even a fault occurred. Performance degradation can be acceptable in the case of a fault presence however, it is important not to break off the whole system. Fault tolerance requires extra sources to detect and correct the faults. By the help of extra sources, it increases total correctly running time of the system.

5.1.1.1 Redundancy

Reserving extra sources to mitigate the effects of faults is called *redundancy*. Redundant sources are necessary to detect and eliminate faults. Alternative redundancy methods such as hardware, software, and time redundancy can be used on a fault tolerant system. This work only focuses on hardware redundancy to develop a fault tolerant hardware.

On a fault tolerant hardware, faulty elements can be replaced by redundant ones. Another redundancy is required for the error detection circuits since fault recovery can be done after detection.

5.1.1.2 Availability

Availability is a measure of ratio between running time without breakdown and total running time of the system. High availability is the main aim of a fault tolerant architecture. For example, mission critical applications require very high availability. Ideal availability for such systems is 100%.

5.1.2 Triple Modular Redundancy (TMR)

Triple Modular Redundancy (TMR) is widely used approach to mask the faults. TMR is composed of three *redundant modules* and one *voter module* as seen in Figure 5-1. All redundant modules are exact copy of each other. The level of a redundant module can range from only a gate to a complex circuit. A majority voter compares the outputs of these identical modules.

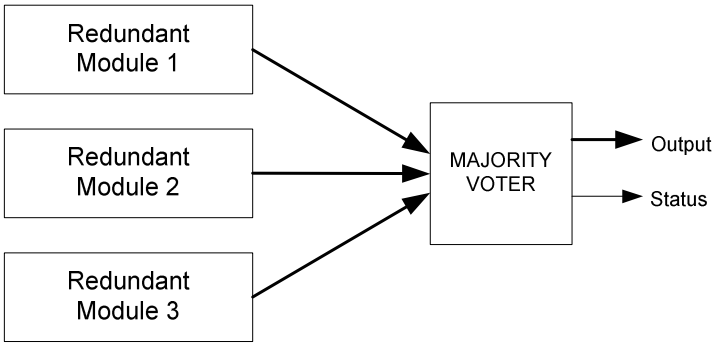

Figure 5-1 Triple Modular Redundancy (TMR) with Simplex Voter

When no error is present in the system, the outputs of all modules agree with each other, then voter use this output. If one of the modules fails, it gives different output from the other modules. Then the outputs of two correct modules agree and voter uses this output to feed forward. If more than one module becomes faulty then none of them agrees with each other and the system breaks down. Therefore, using just TMR can mask the effect of a single failure. If a recovery approach is used on TMR after a failure, system can return to initial state and more than one error can be masked.

The advantage of TMR is high system availability (i.e. 100%) even if an error is present. Another advantage is no extra error detection circuit is required inside a module. Voter immediately detects an error on a redundant module and it can reflect this status to the output.

5.1.3 Rollback and Roll-forward

Duplicate modules can be used to establish Fault Tolerance on a system. If one of the duplicate modules encounters an error, the other can keep the system alive as in TMR. The error must be eliminated as soon as possible since occurrence of an error on correctly operating module can crash the system. After eliminating fault, the state must be recovered to a well-known point. This can be accomplished by either copying state from correctly working module or returning to a checkpoint. Restoring state from a past checkpoint is called *rollback* operation while copying it from a correctly running system is called *roll-forwarding* operation.

5.1.4 Fault Types

Encountered fault types during the operation of a digital circuit can be classified into two groups. One of these types is transient fault and the other type is permanent fault. A transient fault causes the circuit to work incorrectly in a limited time interval; afterward the fault disappears. It is important to be aware of the fault and take precautions such as re-evaluating circuit operations. A permanent fault cannot be corrected without any intervention from the outside world [43]. Spare sources must be reserved in the system, in order to eliminate permanent faulty elements. Whenever a permanent fault occur these spare sources takes the function of the faulty blocks.

5.1.4.1 Transient Faults

Heavy ion and proton particles on space applications may hit a memory element such as latch, flip-flop, RAM etc. This cosmic radiation results in a state change of the memory element, which is called Single Event Upset (SEU). Such events are seen more frequently with continuously decreasing transistor sizes by improving implementation technology [44]. Normally these bit-flips are transient and disappear after storing a new value on the memory cell or resetting it.

However, these transient errors become an important issue on SRAM based FPGAs where circuit behaviours are determined by configuration memory elements. If a configuration memory element of FPGA encounters a SEU, it will change corresponding circuit behaviour. In normal operation of an FPGA (i.e. no reconfiguration is done), this memory will never be refreshed until a power-down – power-up sequence. Hence, these transient errors become permanent errors, if no configuration memory refresh is done. The system can be damaged because of these functional errors. Therefore, they must be corrected by writing the true configuration data after a SEU in order to recover circuits. An example bit-flip or SEU on a Lookup Table (LUT) and resulted functional error is illustrated in Figure 5-2.

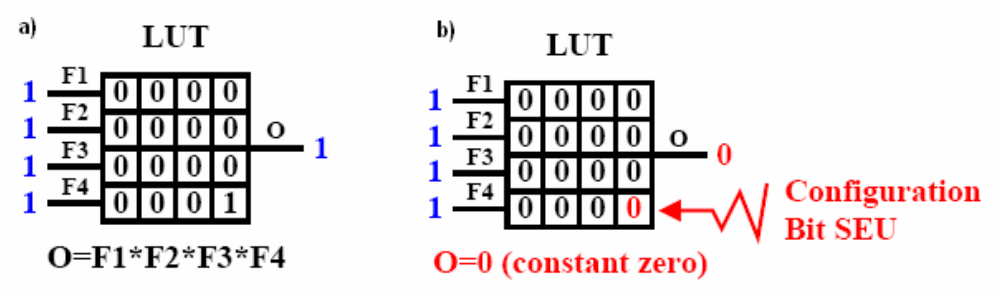

Figure 5-2 Effect of a Single Event Upset (SEU) a) Original Configuration with function AND b) Configuration after a SEU with Function Constant Zero [45]

5.1.4.2 Permanent Faults

Permanent faults can arise during the operation of a circuit due to long life usage or impurities on manufacturing that are not detected with initial tests [28]. The long usage of circuit on high radiation environment can also trigger permanent fault generation by modifying threshold voltage levels of transistors on the circuit [46]. Researches use some models for operational permanent faults and most common model is stuck at error model.

Stuck-At Model

Stuck-At fault models are widely used for permanent error modelling due to their simplicity. The models are stuck-at 0, stuck-at 1, switch stuck open and switch stuck closed. Stuck-at 0 (SA0) means the input or output of a logic gate is locked to 0 and cannot be changed anymore. Similarly, stuck-at 1 (SA1) means the input or output of a logic gate is locked to 1.

Other models are used for the switches. Switches can be locked to a state connecting (stuck closed) or not connecting (stuck open).

Any signal of a CLB can encounter a SA0 or a SA1 error on an FPGA. A connection point on the switch matrix or Programmable Interconnect Points (PIP) can also encounter stuck-at open/closed errors. These elements must be replaced by undamaged ones.

5.2 RELATED WORK

The main research topics about the Fault Tolerance that use Runtime Reconfiguration (RTR) are listed below:

- Detection of Errors
- Recovery of transient errors (such as SEU), which can be permanent if configuration memory of SRAM FPGAs is affected
- Tolerating permanent errors (such as corruption of a CLB)
- Simulating faults

Some researches propose methods for more than one topic at the same time. They are discussed under one subject in the following section.

Error Detection

There are different methods available to test the sources inside an FPGA. Some of them use online methods in which the system continues to operate. Runtime reconfiguration is used to enable uninterrupted operation of user circuit. For example, M. G. Gericota et al. [28] proposed a non-intrusive CLB test method. The method uses a dynamic rotation mechanism to test all CLBs inside the FPGA and rotation is based on RTR of hardware. In order to test a CLB in a non-intrusive manner, its contents are copied to another CLB called *replica*. Then, test is done to the replicated CLB. If no errors found on replicated CLB, the function again copied from the replica CLB. This method is able to detect permanent errors on CLBs and it can recover transient errors on CLBs.

J.Emmert et al. [29] used another method for error detection based on BIST (Built-In Self-Test). BIST structure includes Test Pattern Generator and Output Response Analyzer to test the functionality of the block under test. Their method implements a roving Self-Testing Areas (STARs) that reserve a test area inside the FPGA. A STAR contains vertical and horizontal blocks to be tested. After the test operation of blocks completed, they move to another position. The logic blocks other than STAR are always active inside the FPGA. Partial reconfiguration of the FPGA allows the system working even if the STAR is moving to another place. Moreover, a reconfiguration can eliminate the usage of faulty logic cells.

SEU Mitigation

Single event upsets on configuration memory of SRAM based FPGAs become a permanent error as mentioned before. Two methods can be used to eliminate a SEU on SRAM configuration memory. First method uses a readback, compare, and repair strategy. Configuration memory is continuously read and compared with the original configuration data. If an error found on a frame, it is partially reconfigured again to correct error. Instead of comparing all the configuration bits of a frame, CRC can be generated and can be compared with already prepared CRC to find an error. One restriction of this method is LUT cannot be used as shift-register or RAM, since readback operation can disrupt the data on it [47] [48].

Second method continuously writes correct configuration data to the configuration memory of the FPGA. This method is called *scrubbing*. The reload period is selected according to the rate of SEU events. As a rule of thumb, scrubbing rate must be 10 times higher than average SEU rate.

Both systems must use a fault tolerance method such as TMR in order to increase availability. Otherwise, system can stop working until the repair process correct the fault. Xilinx proposed [49] such SEU mitigation method that uses partial reconfiguration (scrubbing) and TMR.

Another research by R. F. Demara et al. [50] proposed a TMR like solution in which redundant modules are two instead of three. These two redundant modules are called Discrepancy Mirrors. Discrepancy mirrors are exact copy of each other and their output is voted by a discrepancy detection circuit. When a fault arises on one of the redundant modules or detection circuit, the detection circuit indicates unmatched outputs. Therefore, it does not need a golden circuit for the detection circuit. This method enables immediate detection of errors as opposed to test vector method, which requires a high latency for detection of errors. After detecting error, a reconfiguration can eliminate the single event upsets on discrepancy mirrors and detection circuit.

Tolerating Permanent Faults

In a fault tolerant system, extra sources must be reserved in order to eliminate permanent faults. These sources are kept as spare until a fault appears.

Then, in the case of damage occurred, the faulty element is replaced by a spare (non-faulty) one to implement the function of it.

Runtime reconfiguration (RTR) of hardware sources can be very helpful to mitigate the effects of faults on FPGAs. Configuration schemes can be prepared that does not cover faulty sources. Since RTR enables uninterrupted operation, faulty sources can be replaced with spare ones by using such configurations. This is a very useful property since system can stay active even during recover operations. Therefore, researches that deals with Fault Tolerance of FPGA use RTR property of them. In this work, RTR is also used to mitigate effects of faults on FPGAs.

W.J. Huang et al [30] have proposed a column-based precompiled configuration technique that can eliminate permanent faults on the FPGA. In this method, FPGA is divided into multiple columns. One or more columns are reserved as spare and a user design is mapped to the remaining columns. For each function, multiple configuration schemes are prepared offline, and they are used immediately if an error appears. For example, if an error appears in a CLB of a column then the function inside is moved to a neighbour error-free column. The function in neighbour column is also moves by one column and replaces another function. All functions are moved until a spare column is used, then the erroneous column becomes empty. Therefore, each function has a configuration mapped on each column. This method can tolerate permanent faults until the number of faults becomes equal with the number of spare columns.

TMR architecture can be used to tolerate permanent faults. It can tolerate up to one erroneous redundant module if no additional method is used. Its disadvantage is high area overhead. Another work, by S.Y. Yu et al. [51] provide a solution to reduce the area overhead of the TMR. Their technique divides a redundant module into two parts. Then different fault tolerance methods are applied to two distinct parts. For example, in one of the implementations, a part uses a redundant architecture (i.e. TMR) while the other part is strengthened by a Concurrent Error Detection (CED) unit. If TMR part encounters an error, it is reconfigured to eliminate fault and then errors are corrected by roll forwarding. In another method, one part uses TMR while another one uses a duplex scheme. If an error occurs on the parts implemented with duplex schemes, it is reconfigured to eliminate fault and then error is corrected by rollback recovery method.

Simulating Faults

Some researches try to find solutions for simulating Single Event Upsets. This simulation is necessary before launch to analyze the behaviour of the circuits under a real space environment. Two methods can be used to simulate SEUs on the earth. First, SEUs can be directly injected by using radiation (proton beam) test equipments. Second method reconfigures the FPGA with a configuration data that embeds bit-flip errors inside. The second method is a low cost solution since no external equipment is necessary to see a SEU effect.

For instance, Gokhale et al. [48] used RTR to induce SEUs into the configuration memory of the FPGA. P. Kenterlis et al. [52] used JBits to emulate the configuration data corruptions.

5.3 DESIGNED ARCHITECTURE

5.3.1 General Overview of the System

The system is mainly composed of a board containing runtime reconfigurable FPGA and a Personal Computer (PC). A Triple Modular Redundant (TMR) circuit operation is performed on a reconfigurable FPGA. TMR structure includes a fault tolerant user application circuit and a controller circuit (voter of TMR). A PC program organizes reconfiguration processes of the FPGA according to the feedbacks of the controller circuit. The fault tolerance of the user circuit is maintained by an algorithm run on the PC Program with the help of the controller circuit.

Some additional peripherals are used to complete system function. More specifically, a board displays information coming from the FPGA board to the user. Another board enables communication between PC and FPGA. Lastly, a cable from Xilinx is used for downloading configuration data (i.e. bitstreams) from PC to FPGA board. In Figure 5-3, all elements and connections of designed reconfigurable system are shown.

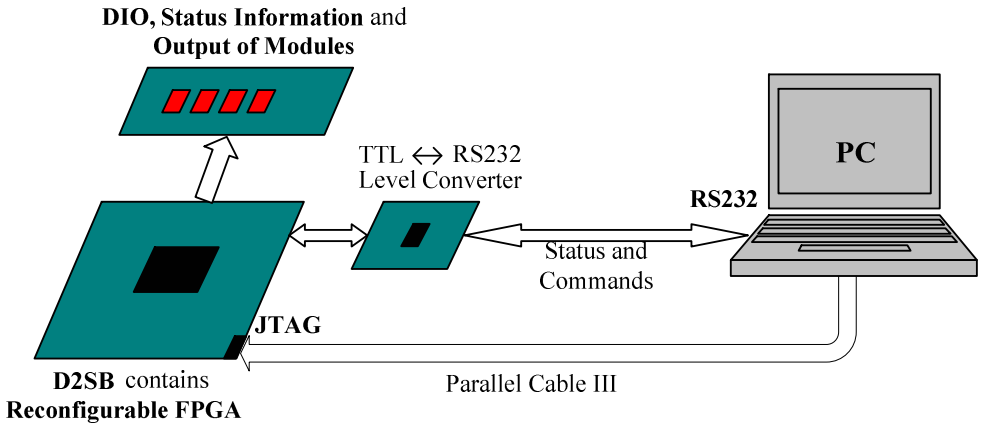

Figure 5-3: Components and Connections of the Reconfigurable System

5.3.1.1 Addressed Error Types

The design can tolerate following faults:

- Corruptions on FPGA configuration memory due to SEU like effect
- Permanent errors on CLB sources of the FPGA
- SEU corruptions on Flip-Flops inside a CLB

5.3.1.2 Partial Runtime Reconfigurable Design

Using only Triple Modular Redundancy (TMR) for Fault Tolerance cannot compensate errors on different modules. In other words, if an error occurs on a redundant module, the system can continue to operate. Nevertheless, if another error comes to another module during this state, system will halt.

However, if the error on a redundant module of the TMR can be eliminated whenever it appears, the system can compensate more than one error. Designed architecture use partially reconfigurable hardware (i.e. FPGA) where parts of circuit can be changed while others are operating. Faulty redundant modules are replaced with repaired ones while continuing the operation of the whole circuit by the help of partial reconfiguration.

The reconfiguration of the FPGA is based on Module Based Partial Reconfiguration, which was described in Chapter 4. Hence, the guidelines of

module based partial reconfiguration are followed for this design and its restrictions are considered.

5.3.2 Hardware Used in the Design

Three boards and two cables are used to implement proposed architecture. These are listed below:

- A board containing a reconfigurable FPGA on it (D2SB)
- A board containing seven segment displays and switches on it (DIO1)
- A board that converts TTL voltage levels to RS232 voltage levels
- A JTAG cable for the PC (Parallel Cable III)
- RS232 to USB converter cable

The hardware components of the reconfigurable system (except PC) and their connections are shown in Figure 5-4. Detailed information about individual hardware components is given in the following sections.

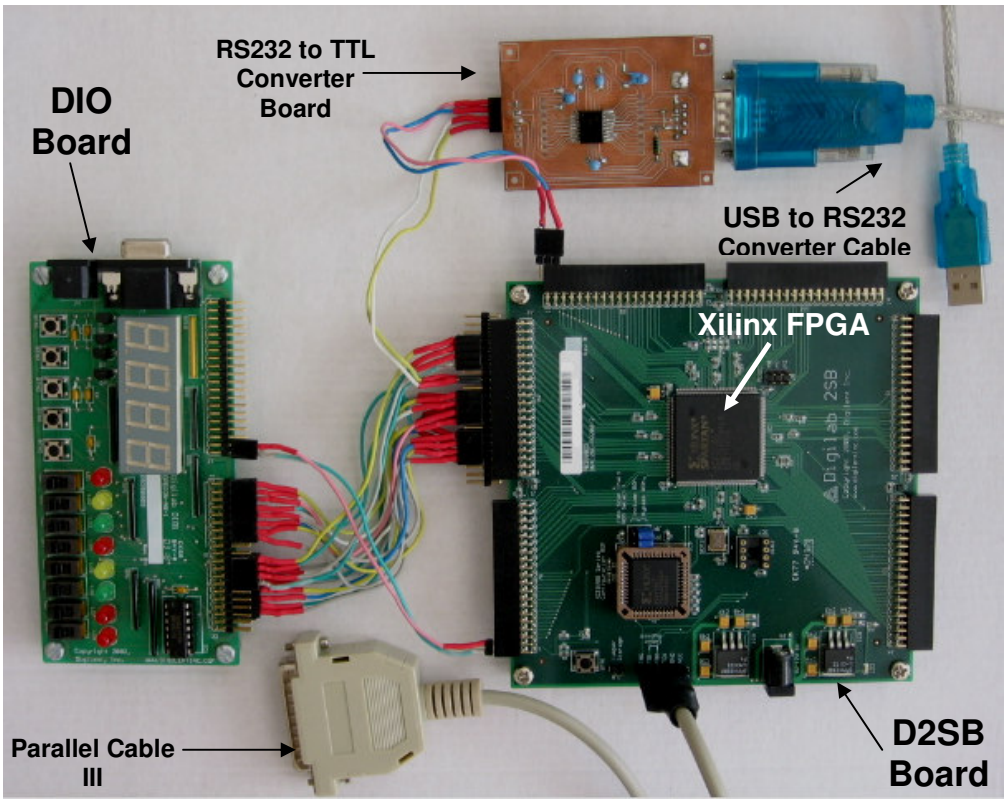

Figure 5-4: Picture of the Reconfigurable System without a PC

5.3.2.1 Digilent D2-SB System Board

Digilent D2SB board contains a Xilinx Spartan 2E XC2S200E FPGA and peripherals to run the FPGA. The block diagram of the board is shown in Figure 5-5. Xilinx XC2S200E is a reconfigurable FPGA as explained in Chapter 4.There is a 1.8V regulator to supply the power of the FPGA core. FPGA I/O voltage is fed by the 3.3V regulator. To make simple tests one LED and one push button is inserted on the board. There is also a socket for a configuration Programmable Read Only Memory (PROM). A JTAG connector is provided to enable configuration of FPGA and PROM by JTAG cable. 143 I/O pins of the FPGA are expanded to the connectors to enable connection with other daughter boards or user circuits [53].

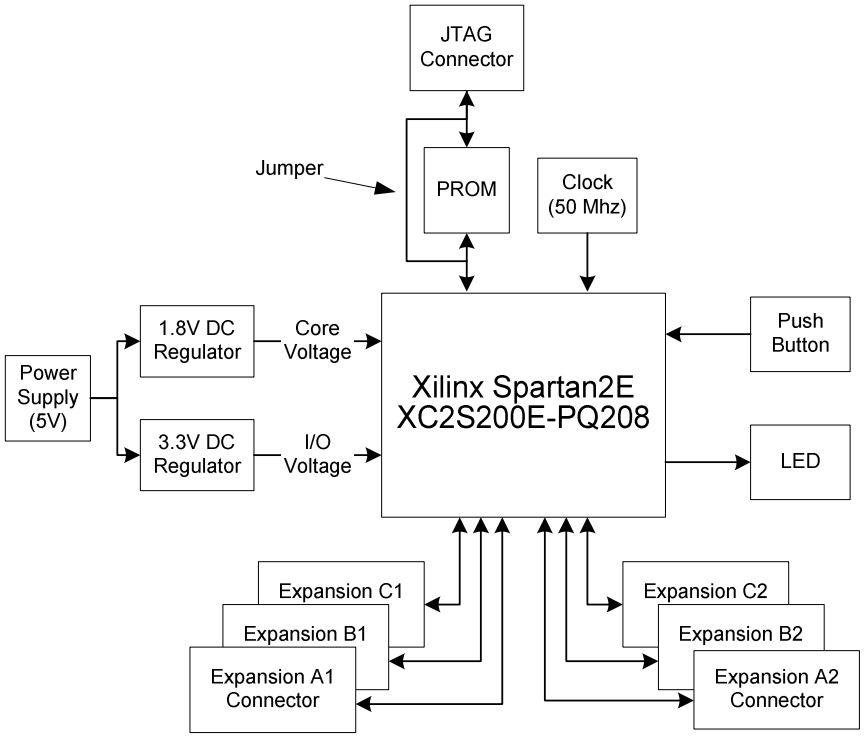

Figure 5-5: Block Diagram of the D2-SB board

5.3.2.2 Digilent DIO Board

Digilent DIO1 board is used to display some information to the user. It takes necessary data and displays on seven segment displays on it. Actually, DIO1 is a daughter board that can be directly connected to the connectors of the D2SB board. However designed architecture implements module based reconfiguration in which a module can access to only I/O's at its boundaries (as explained in Chapter 4 in detail). This limits connection of D2SB board to the DIO1 board to certain pins. Therefore, the daughter board is not connected directly to the D2SB but connected by wiring up the connector pins.

5.3.2.3 RS232 to LVTTL Converter Board

A communication channel is necessary between host computer and FPGA board for the handshaking operations. As a straightforward method, serial port of the computer is selected as the communication channel. However, the signal

levels of the serial port and Xilinx FPGA are incompatible since Serial port uses RS232 voltage levels and Xilinx FPGA is configured to work on TTL voltage levels. In order to convert voltage levels from RS232 to TTL and vice versa, a converter board is constructed. The schematic and PCB figures of the board are given in Appendix A.

Today, some computers such as Laptop PC's do not have a serial port any more. To overcome this problem, a USB to Serial port converter is also added to the board. The design is done normally as using standard serial port.

5.3.2.4 Parallel Cable III

Parallel Cable III is a JTAG cable provided by Xilinx. It is used for loading configuration bitstream to the FPGA using its JTAG port. It is connected to the Parallel Port of the PC and works at 200 kHz.

5.3.3 Working Principle of the TMR

A TMR system is constructed on the FPGA. This system includes three *redundant modules* and a *voter module.* The logic circuits on redundant modules are exact copy of each other and they include user circuit. Voter module compares the outputs of redundant modules. All modules are partially reconfigurable. Alternative partial configurations are prepared for alternative placements of a redundant module.

An initial configuration that contains TMR is loaded to the FPGA. After the initial configuration, redundant modules send their state information to the voter module all the time. Voter module checks if all of them has the same output or not. Then voter sends state information to the PC on fixed time intervals. The connections of Redundant Modules and Voter Module are shown in Figure 5-6.

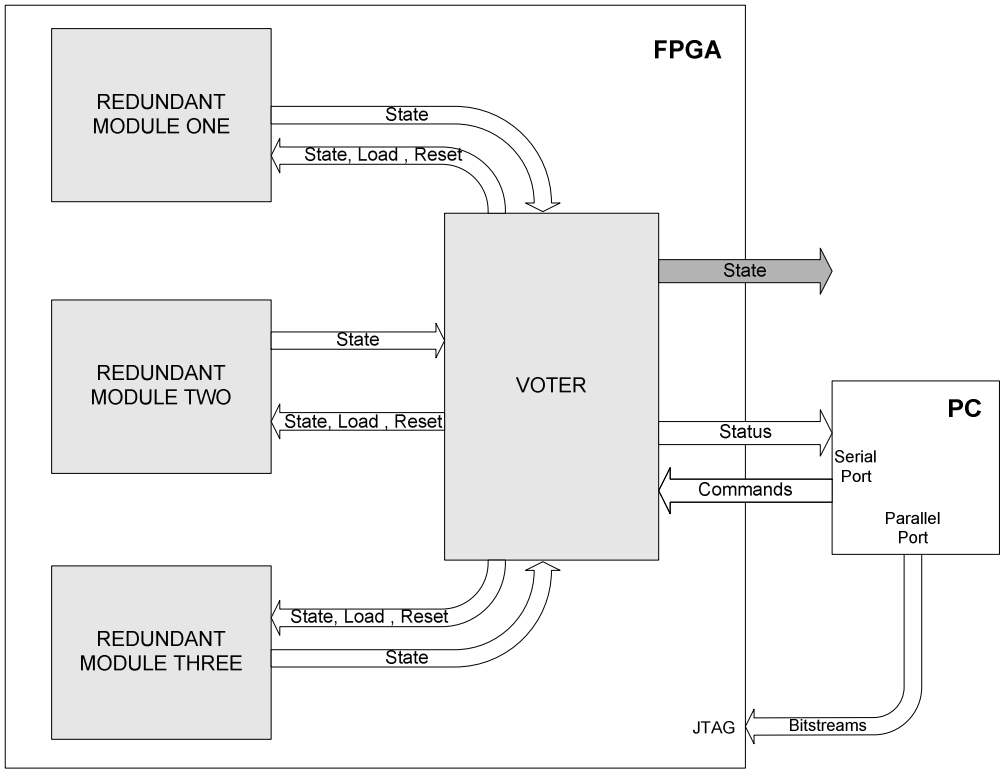

Figure 5-6: General Structure of the System

If an error condition occurs on any redundant module (i.e. one module gives different output than the others), voter also send this information to the PC. Then PC program evaluates how a reconfiguration operation will occur. It starts recovery operations such as reconfiguring a redundant module and instructing the voter module. Hence, it can be said that intelligent part of the system is placed on the PC Program. However, it can be easily moved to the Voter module if a System on Chip (SoC) is required.

5.3.4 VHDL Design of the TMR Circuit

The logic circuits inside FPGA are designed by using VHDL. All the source codes of designed architecture are given in Appendix E (FTArchitecture/Synthesis directory is used for synthesizing these VHDL files). The individual modules of the design will be explained in this section.

5.3.4.1 Voter Module

Voter Module is mainly responsible from controlling whether all modules give same output and informing PC about module states. Other responsibilities are recovering states of redundant modules after reconfiguration, and driving display units. Figure 5-7 shows internal units of the Voter module.

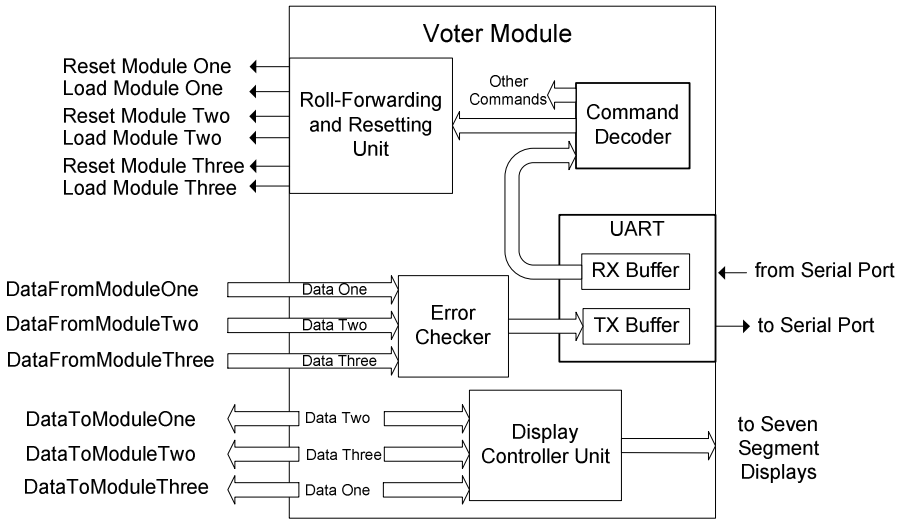

Figure 5-7: Block Diagram of the Voter Module

Error Checker

Error checker unit controls whether all units give same output. To accomplish this goal, a majority voter circuit is implemented on error checker unit. There are three comparators checking the equality of Module One - Module Two, Module One - Module Three, and lastly Module Two - Module Three. If all comparators give 1 to the output, Error Checker generates “All Modules are OK” signal as shown in Figure 5-8a. If comparators belonging to a module give zero output then this module is treated as faulty and reported with a corresponding signal. For example, in Figure 5-8b the logic that generates “Error on Module One” is shown.

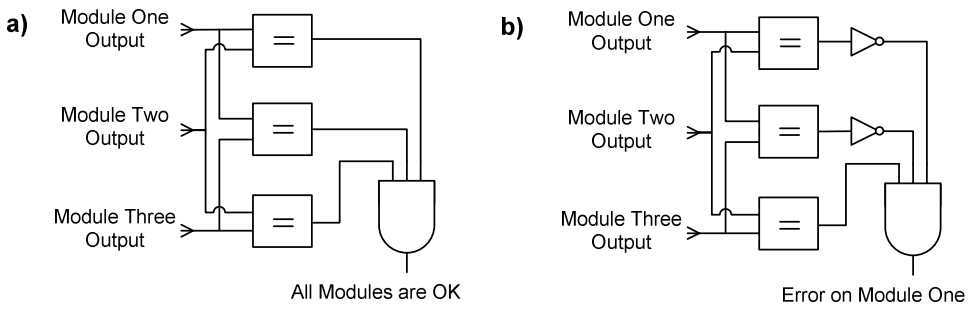

Figure 5-8: Internal Logic Circuits of Error Checker Unit a) Circuit giving "All Modules are OK" signal b) Circuit giving "Error on Module One" signal

Error checker sends status messages to the PC via UART. The messages are shown in Table 5-1.

Table 5-1: Status Descriptions and their corresponding ASCII values

Status Description	ASCII Decimal Number	ASCII Character
Module One gives different output	A	65
Module Two gives different output	B	66
Module Three gives different output	C	67
All Modules give different output	D	68
All Modules give same output	E	69

Universal Asynchronous Receiver Transmitter (UART)

The voter communicates with supervisor program that runs on the PC via the serial port. A Universal Asynchronous Receiver and Transmitter (UART) implements Serial port protocol on the Voter side. Both PC program and Voter use 115200-Baud rate. The UART Intellectual Property (IP) is taken from an application note by Xilinx [54].

Command Decoder

Command Decoder unit decodes the data coming from the PC. It reads receive buffer of the UART whenever a data available. Then it decodes the data and if necessary, it sends commands to the individual units on the Voter Module. A command is one-byte data. It has three fields; *Module Number*, *Module Command* and *Generic Command* as shown in Figure 5-9.

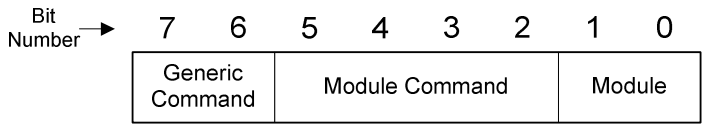

Figure 5-9 A Command Byte sent by the PC

The Module field indicates the recipient of a Module Command. It can take 01 value for Module One, 10 for Module Two and 11 for Module Three. The Module command is sent to the individual Modules according to the Module field. The Module Commands and their codes are listed in Table 5-2.

Table 5-2: Definitions and codes of Module Commands

Command Code	Command Name	Command Definition
0001	Reset	Reset module
0010	Rollforward	Roll forward states of the module from another module
0011	AskDiscrepency _BusMacros	Compare the input data coming from original bus macro and alternative one. Then send this information to the PC. (i.e. request discrepancy information)
0100	UseAlternateBusMacro	Use the data of alternative bus macro
0101	DeleteDiscrepencyInfo _BusMacros	Reset the register that holds discrepancy information
0110	UseOriginalBusMacro	Use the data of original bus macro

Some generic commands are added for debugging purposes. When “00” comes in the *Module* field, the *Generic Command* part of command byte is used. Currently, only two Generic Commands are available, namely *Check Fast* (01) and *Check Slow* (10). If Check Fast command is received by the voter, module errors are reported to the PC all the time. If Check Slow command is received, voter sends status messages on fixed intervals. Therefore, when check slow option is used some errors may be missed. However, it is necessary to prevent locking of the communication channel when a module gives error all the times.

Roll Forwarding and Resetting Unit

When an error occurred on a module, designed architecture corrects the error. If redundant module includes only combinational logic circuits then recovery operation is simple. Reconfiguring redundant module to eliminate faulty elements solves the problem. However correcting faults of a sequential circuit includes two operations. These operations are first eliminating the fault, and then recovering the states of the sequential elements. The state recovery operation is performed by the Roll-Forwarding and Resetting Unit (RFRU). However, PC sends commands to the RFRU to initiate the recovery operations.

Some extra signals are used on the redundant modules to enable recovery operations. A redundant module takes *feedback data* to load its internal registers when an error occurs. This data is taken from the other module’s data outputs. For example, the output data of *Redundant Module Two* is fed to the input data of *Redundant Module One* as seen in Figure 5-7. In the case of an error occurred on Module One and the others are correctly working, recovery operation updates Module One’s internal registers with the Module Two’s data.

RFRU sends other extra signals – load, reset, and Clock Enable (CE) – to all redundant modules individually. CE signal is connected to all synchronous logics (i.e. Flip-Flops inside the CLBs) inside redundant modules. The synchronous logics will run when CE signal is activated. Therefore, control of the clock rate is achieved by CE signal.

Recovery operation occurs as follow: At first, a *Reset* signal is applied to the repaired module and it goes to the initial state. Then a *Load* signal is given to the module under recover operation. At this time, all registers of correctly working modules are deactivated by disabling *Clock Enable (CE)* inputs (this ensures

clock-by-clock equivalence of working and repaired module states). Two different strategies can be applied for CE signal by the roll-forwarding unit.

First method maintains a constant clock frequency rate for all modules at all times. Clock enable signal halves the frequency rate of input clock for all modules. In other words, one clock cycle is used for operations while consecutive cycle is not used (i.e. idle cycle). When roll-forwarding operation is active, the idle cycle is used for loading data from other modules.

On the other hand, second method tries to achieve highest frequency rate if no error is present in the system. In this method, clock frequency of working modules is halved only during roll forwarding and their states are copied to recently repaired module. After reconfiguration process, modules again work at normal frequency rate.

Another important point is the duration of Load signal at recovery operation. The Load signal is applied until a state change is seen at the output of the correctly working modules. Therefore, internal registers of recovered module can be initiated correctly after a state change occurs. The simulations of recovery operations are given in Appendix B.

Display Controller Unit

Display controller manages seven segment displays (SSDs) to display data output of redundant modules. Since three redundant modules exist, three SSDs are used to display their data. Display controller unit takes the output data of all redundant modules. It converts the data output of redundant module to a valid format that will result in a meaningful pattern on a seven segment display. Then it sends converted data to the seven segment displays (SSDs) available on the DIO board. For example, if 4-Bit data is “0010”, SSD displays 2 (i.e. corresponding decimal number). More details about driving SSDs on DIO board are explained in [55].

5.3.4.2 A Redundant Module

A redundant module includes a user circuit. The circuit can be composed of combinatorial and/or sequential logic gates. TMR structure is applied to the final output of the redundant modules. One of redundant modules is shown in the following figure:

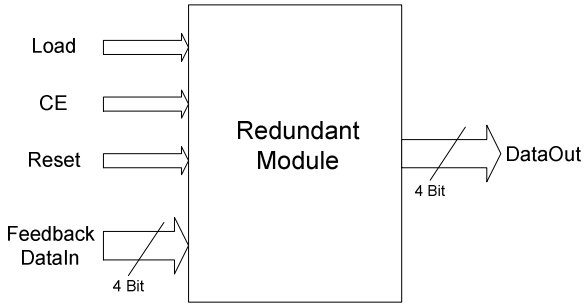

Figure 5-10: A Redundant Module of the TMR System

To test fault tolerance capabilities of the system a Finite State Machine (FSM) is selected as the user circuit. FSMs are formed by using both combinational and sequential logic circuits. Therefore, a feedback data path is used to recover states of the FSM.

A repetitive structure is used on the FSM. In states with prefix stX_load, a counter value 2000000 is loaded and directly passed to another state. On this state, count value is decremented until it reaches to zero. Then it is again passed to another load state and load count 2000000 value. This structure repeats for 16 states then FSM returns to the first state. The outputs of the states are different and used for recover operation. First state sends 0, second one sends 1 and it continues up to 16. This output is encoded to 4 bits (24=16) and send to the output. The FSM state transitions and state outputs are shown in Figure 5-11.

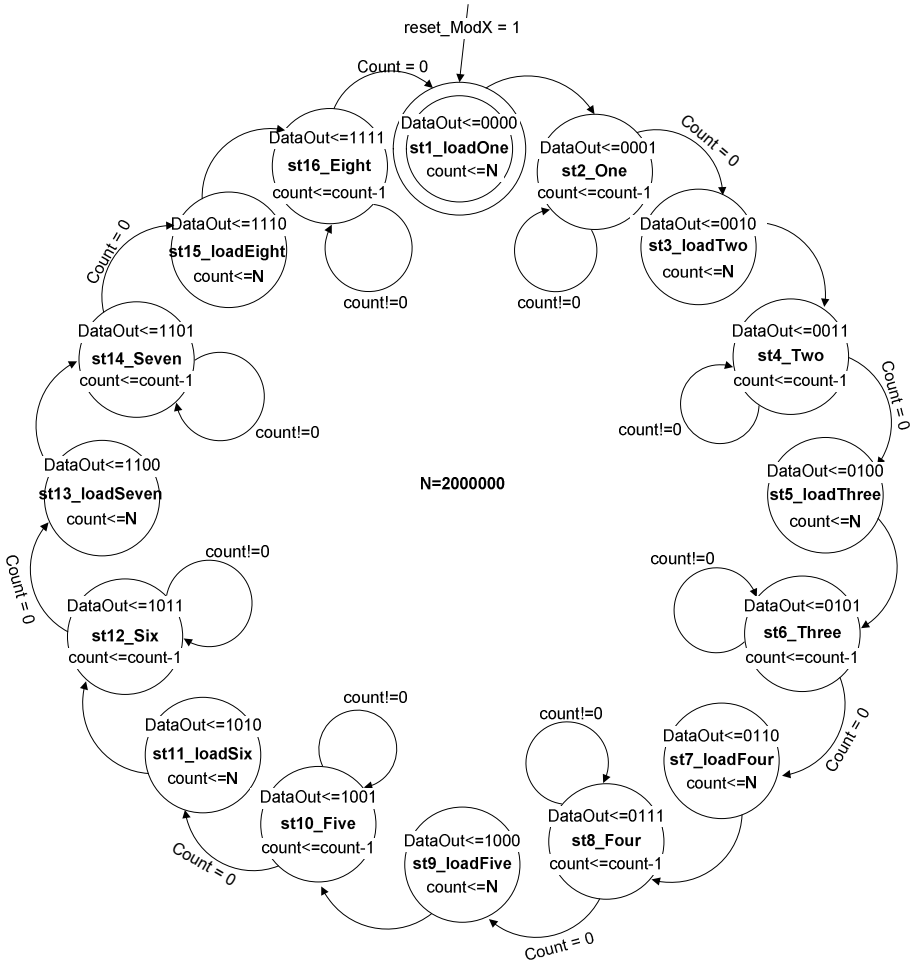

Figure 5-11: Finite State Machine that is implemented on Redundant Modules

User must take into consideration the usage of Clock Enable (CE) and Load signals. These signals are necessary to roll-forward redundant modules in the case of a fault occurs. Any synchronous circuit must use ce_Mod**X** to enable loading input data to a Flip-Flop with the clock. In addition, load_Mod**X** must be used to load data coming from the Voter module. An example VHDL code for Module One is given below:

```
if (reset_ModOne='1') then
      … → Load Initial FF States

elsif (Rising_Edge(clk)) then
            if(load_ModOne='1') then
                  … → FF States Rolled Back

            elsif(ce_ModOne='1') then
                  … → Normal Sequential FF State Transitions

            end if;
end if;
```

5.3.5 Partial Reconfigurable FPGA Design

Module Based Partial Reconfiguration Flow of Xilinx is used in this design to achieve a runtime reconfigurable design. TMR modules can be reconfigured whenever an error appears on them. These reconfigurable modules must occupy full height of the device with this method (More details of restrictions were given in Chapter 4). For this reason, FPGA is divided into columns and modules are placed inside them. Five columns are reserved; four of them are occupied by three redundant modules and a voter module. One column is intentionally left as spare for the future requirements.

The modules on rightmost/leftmost sides can use more pins than the modules that lie on the middle. Therefore, the voter module is put on the right side of the device to use more I/O pins. Figure 5-12 shows the layout of modules inside the FPGA.

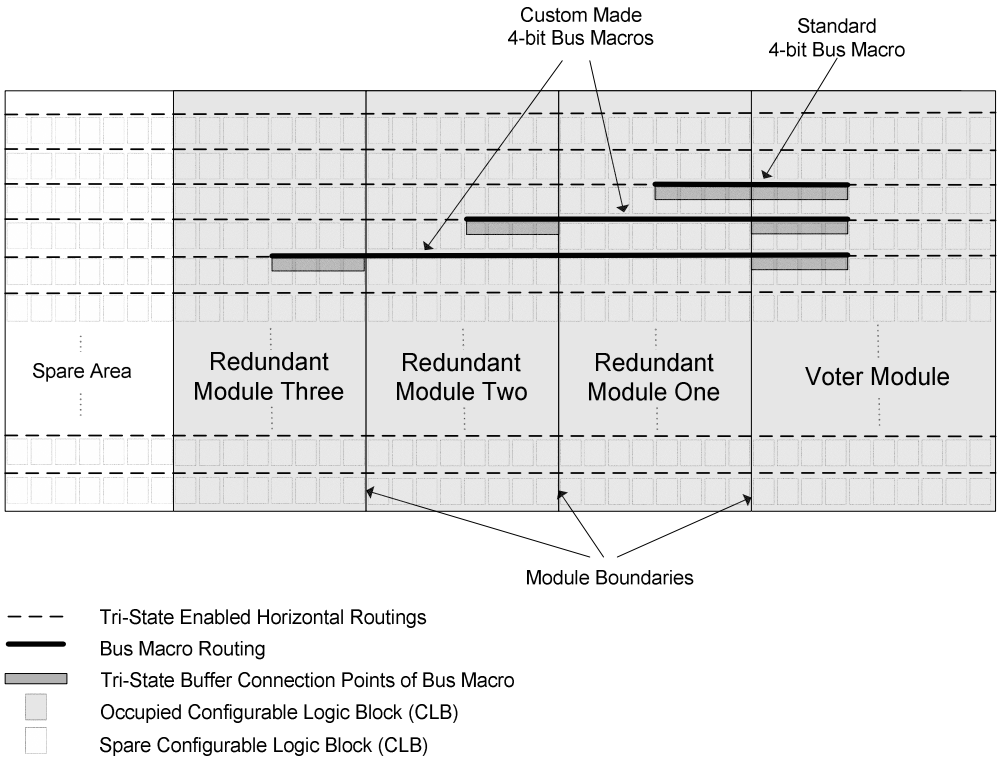

Figure 5-12: Layout of the Modules on the FPGA

Minimum column width of a reconfigurable module must be four CLBs, since a bus macro connection requires four CLB columns. However, it is observed that place and route tools cannot map the logic if modules have a width less than seven CLBs width. Otherwise some routing errors appear. To eliminate these errors minimum width is selected as seven CLB columns. In addition, reconfigurable modules must be put on four slice boundaries (4-8-12...) for partial reconfiguration [35]. Therefore, boundaries between modules must lay on even CLB columns (four slices equals to two CLBs on Spartan-2E). As a result, a module boundary is selected eight CLB away from other boundary of the module. Final placement of modules is given in Table 5-3.

Table 5-3: Occupied Area of the Modules

Module Name	Range of Occupied CLB Columns by the Module
Spare Area	1-7
Module Three	8-15
Module Two	16-23
Module One	24-31
Voter Module	32-42

Module based partial reconfiguration does not allow signals to pass from one module to another except using *Bus Macro* structures. Therefore, bus macros are used for the communication of redundant modules with voter module. However, some extra effort is needed to communicate two non-adjacent modules since Xilinx only gives a bus macro connecting only adjacent modules. Bus macro given by Xilinx is modified to enable communication between two non-adjacent modules.

5.3.5.1 Modified Bus Macro

Standard bus macro given in Xilinx application note [35] only enables communication between two adjacent modules. However to implement our system, bus macros must be able connect modules which are not adjacent. Therefore, it is modified to accomplish communication between two non-adjacent modules as illustrated in Figure 5-13. A Xilinx tool, namely FPGA Editor, is used for this purpose. FPGA Editor Snapshots of standard and modified bus macro is given in Figure 5-14.

Three custom bus macros are created for the connection of voter module to each redundant module. The names of created bus macros and original bus macro are given in Table 5-4. The bus macro files are used in the implementation phase of the Module Based Partial Reconfiguration Flow. The files are given in Appendix E (in FTArchitecture/ BusMacro Directory).

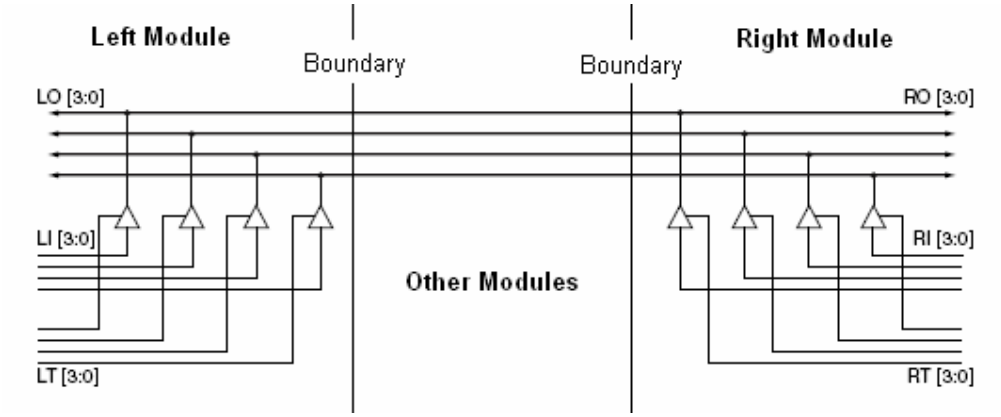

Figure 5-13: Modified Bus Macro that connects Two Non-Adjacent Modules

Working principle of the modified bus macro rely on FPGA cells that can be reconfigured glitchlessly. Writing same configuration data to the configuration cells does not cause a glitch on the cell connection. Furthermore, bus macros are placed exactly same horizontal lines for each configuration of a module. Therefore, while intermediate module is reconfiguring, the bus crossing this module does not corrupted by the help of glitchless configuration of cells. Otherwise, programmable interconnection points (PIPs), which reside in the middle area, will disconnect the bus macro.

Table 5-4: Different Bus Macro Functions and Their Sources

Bus Macro Name	Connecting Modules	Source
bm_one_4b.ncd	“Voter” and “Module One”	Provided by Xilinx (bm_s2e_4b.ncd)
bm_two_4b.ncd	“Voter” and “Module Two”	Edited from bm_one_4b (custom)
bm_thr_4b.ncd	“Voter” and “Module Three”	Edited from bm_one_4b (custom)

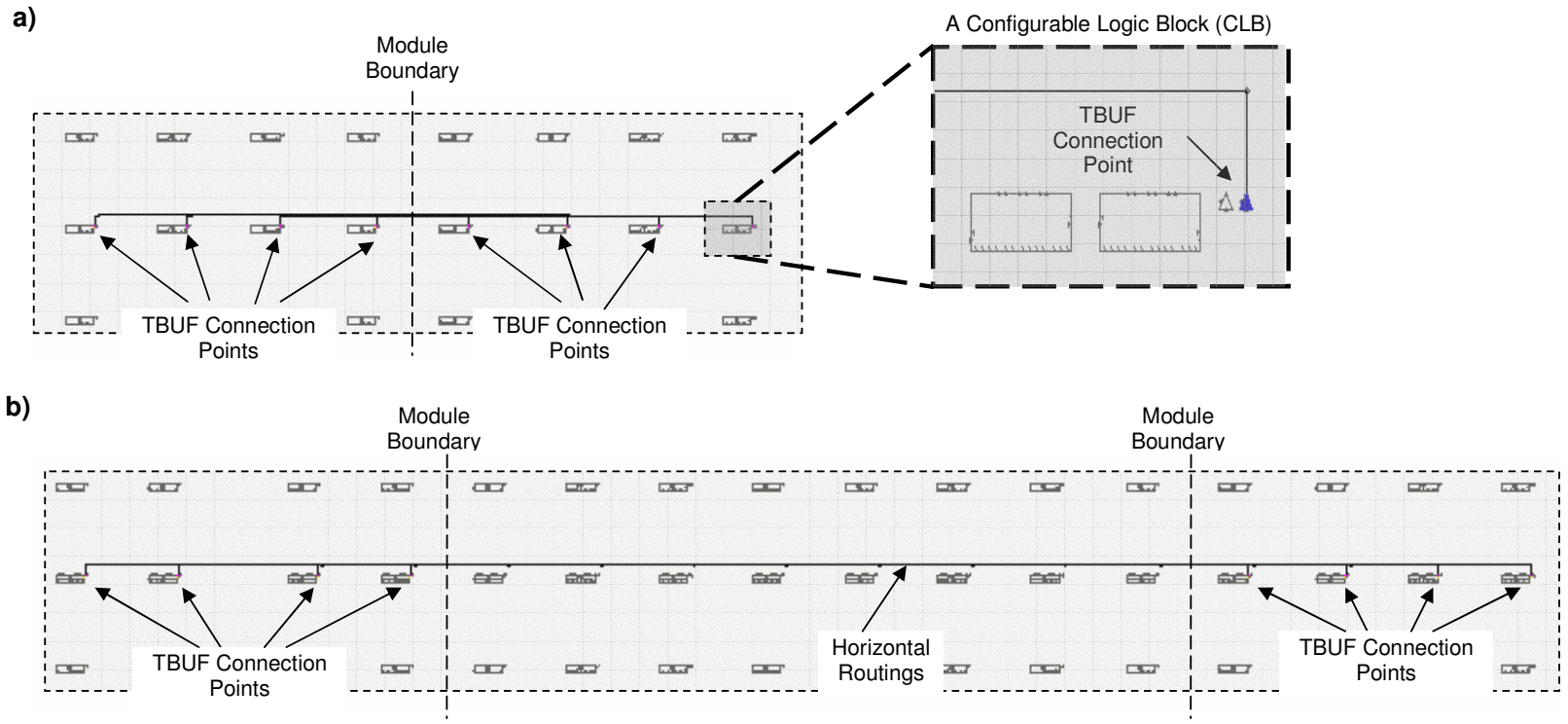

Figure 5-14: FPGA Editor Snapshots of Bus Macros a) Standard Bus Macro connecting Two Adjacent Modules b) Modified Bus Macro connecting Two Non-Adjacent Modules

5.3.5.2 Partial Configurations

To eliminate permanent faults, alternative placements are done for modules. For each alternative placement, a partial configuration (bitstream) is produced. The active implementation phase of the Module Based Partial Reconfiguration Flow is used for generating partial configurations. All batch files for this phase are given in Appendix E (in FTArchitecture/ Implementation/ Module_Name directories).

Partial configurations can be loaded to the corresponding part of the device as shown in Figure 5-15. Placement of logic into different areas is achieved by adding *prohibit* constraint to the User Constraint File (UCF). A UCF example is given in Appendix C. More details of *prohibit* constraint will be given in Section 5.3.6.2 (Eliminating Permanent Faults). For each placement of a module, a corresponding UCF is created and used during the generation of partial configuration. They are given in Appendix E (in FTArchitecture/ UCF directory).

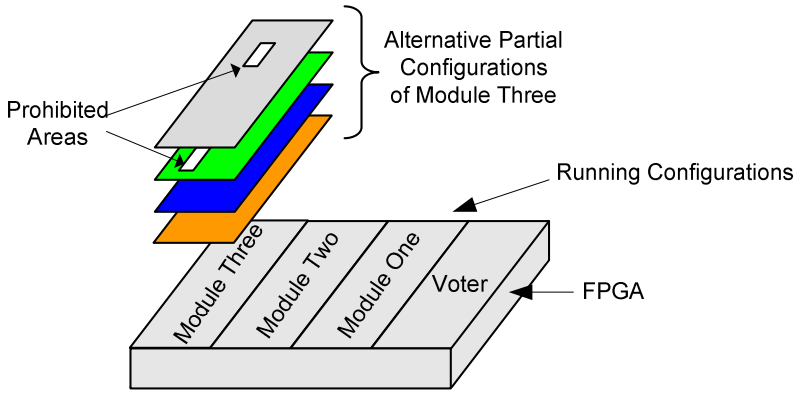

Figure 5-15: Alternative Partial Configurations of Module Three

Therefore, reconfiguring a module with a partial bitstream allows changing the placement of a module. However, during reconfiguration of a module, the other modules must not be affected. For this reason, the bus macros passing through a module (connecting two non-adjacent modules) must remain in all configurations. This requirement is satisfied by locking the bus macros to a fixed position. Positions of the bus macros are locked in the user constraint file (It is

given in Appendix C). Then modular design flow automatically place bus macros in all partial configurations.

5.3.5.2.1 Bus Macro Connections of Redundant Modules

To increase reliability of the TMR system two redundant bus macros are used for each module output. If a redundant module gives erroneous output, Voter can change the output data path from normal bus macro to an alternative one. For this purpose, the output of a redundant module is replicated and passed to the voter by using two bus macros.

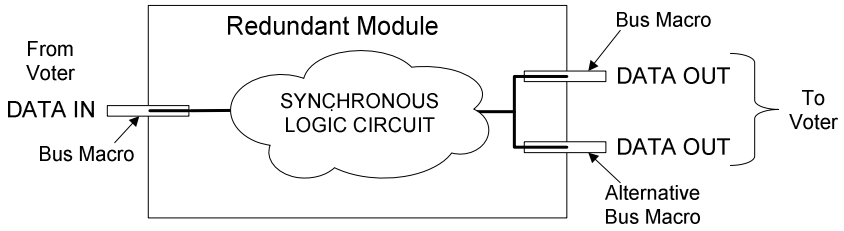

Figure 5-16: Connections of Bus Macros on a Redundant Module

In the case of an error, the voter side checks the equivalence of the bus macros. If a discrepancy seen at the output of them, voter uses alternative one.

5.3.5.3 Batch Files for Modular Design Flow

Batch files are prepared to automate Modular Design Flow. These batch files call necessary Xilinx tools as explained in Chapter 4. Mainly three batch files are prepared for the each step of the modular design flow. These are Initial.bat, Active.bat and Assemble.bat.

Initial.bat copies necessary files to the *topinitial* directory and call initial.cmd. Initial.cmd run the initial phase of the modular design flow with the following command:

```
ngdbuild  -p xc2s200e-pq208-7 -modular initial -uc top.ucf top.ngc
```

Active.bat is used for active implementation phase of the modular design flow. It copies necessary files to individual module directories and call all the *Active.cmd* batch files from these directories. *Active.cmd* generates a partial configuration of a module. For example, *Active.cmd* for the first configuration of module one includes the following commands:

```
ngdbuild  -p xc2s200e-pq208-7 -modular module -active modone
-uc top.ucf ..\topinitial\top.ngc
map  -pr b top.ngd -o top_map.ncd  top.pcf
par  -w top_map.ncd top.ncd top.pcf
bitgen -d -g ActiveReconfig:yes top.ncd partial_modone_1.bit
pimcreate -ncd top.ncd -ngm top_map.ngm ..\Pim
```

Note that pimcreate command is not necessary for the other configurations of module one. It is enough to run pimcreate for the configuration that will be used on the generation of the full bitstream (i.e during Assemble phase).

Assemble.bat creates a final full bitstream file. This file is initially downloaded to the FPGA. It copies necessary files to the topfinal directory and calls assemble.cmd file. Assemble.cmd includes the following commands:

```
ngdbuild -p xc2s200e-pq208-7 -modular assemble -uc top.ucf -
pimpath ..\Pim -use_pim modone  -use_pim modtwo -use_pim modthr -
use_pim voter top.ngc
map -pr b top.ngd -o top_map.ncd
par -w top_map.ncd top.ncd
bitgen -w top.ncd top_final.bit
```

5.3.6 Eliminating Faults

5.3.6.1 Eliminating Single Event Upsets

Single event upsets (SEUs) occurred on the configuration memory of the FPGA can be corrected by only refreshing configuration memory. Therefore, whenever a SEU is detected on a redundant module, its corresponding partial bitstream is reloaded to the FPGA.

5.3.6.2 Eliminating Permanent Faults

The faulty CLBs are eliminated from the system by loading partial configurations that do not use them. For this purpose, empty areas are reserved in alternative configurations. Then if error occurs on a CLB, a configuration that maps the faulty CLB on to empty space is loaded. Reserving spare areas on configurations are done by using the *Prohibit* constraint of Xilinx PAR (Place and Route) tool. Prohibit constraint can be applied to the CLBs that must be discarded during place and route operation. For example, the following constraint is added to User Constraint File to prohibit the usage of two CLBs at the place and route operation:

```
#CONFIG PROHIBIT=CLB_R17C20, CLB_R17C21;
```

It is also allowed to prohibit usage of a column/row of CLBs. The following constraint prohibits the usage of all CLBs on Column 20:

```
#CONFIG PROHIBIT=CLB_R*C20;
```

The granularity of empty spaces can range from fine to coarse grain. Reserving only one CLB to replace faulty a CLB is the finest granularity for the empty space. Such a fine granularity enables effective usage of area on the FPGA while increasing alternative configuration numbers and configuration data. On the contrary, coarse grain elimination lowers the number of alternative configuration bitstreams at the cost of inefficient resource usage on the FPGA.

In this work, it is assumed that FPGA has excessive sources and faults are eliminated in a coarse grain manner. A module is divided into multiple columns, and then one of the columns is left as empty. For each column, a configuration bitstream is prepared offline that left a column empty. In other words, no placement of logic circuits is done on the CLBs that are in the selected empty column. In Figure 5-17, alternative configurations that reserve different empty CLB columns are shown.

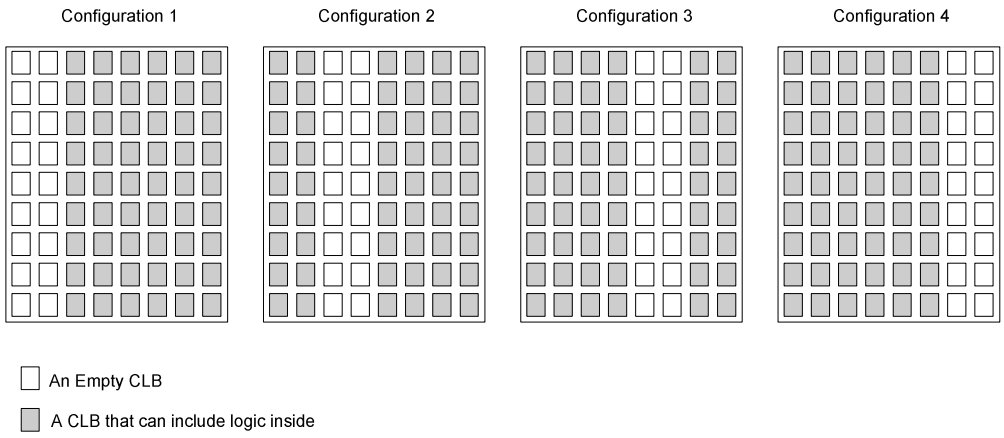

Figure 5-17: Alternative Configurations of a Module

If a permanent-error appears, alternative configurations are loaded to the device until the erroneous CLB is mapped to the Empty Column. Up to eight configurations can be prepared in the design, since the redundant modules occupy eight CLB columns. In this configuration, seven CLB columns can be used by the circuit and one CLB column is reserved empty. Therefore, only 14% (1/7 x 100) of additional CLB sources are required as redundant.

As seen on the example constraints, any CLB can be restricted from the place and route operation. Therefore, it is easy to decrease the granularity of the empty spaces by prohibiting smaller number of CLBs in each configuration bitstream. If desired, coarse grain approach can be easily converted to a fine grain approach.

5.3.7 PC Program

The intelligence of the system is put on the PC Program in order to maintain fault free operation. Constructed TMR on the FPGA already provides a fault tolerance however; it is strengthen by the reconfiguration operations. The PC program is responsible for selecting an ideal reconfiguration scheme.

PC Program communicates with the Voter module on the FPGA via serial port of the PC. It sends commands and receives status information. All bitstream files are stored on the computer and downloaded to the FPGA with the JTAG configuration port. PC program manages the bitstream download operations.

Another responsibility of the PC program is providing a user interface. User can see the current situation of the individual modules of TMR and whole system.

To test the behaviour of designed architecture, faults must be injected artificially. The low cost solution is reconfiguring device with incorrect bitstream files. This operation is also done by the PC Program. The user interface enables adding transient fault, permanent fault and bus macro fault. A screenshot of the PC program is shown in Figure 5-18.

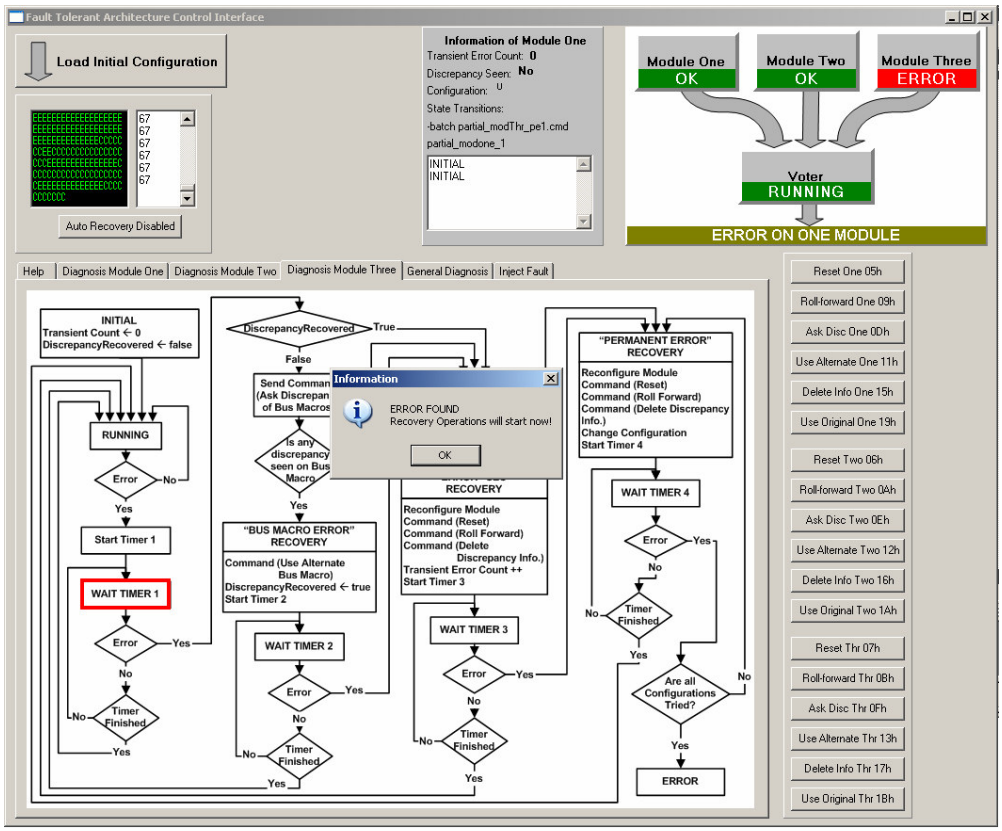

Figure 5-18: Screenshot of the Supervisor PC Program

The PC program has been designed with Borland C++ Builder and written in C++ language. The source codes of the program are given Appendix E (In FTAchitecture / Borland-Project directory).

5.3.7.1 Communication with Serial Port

The UART on the Voter module is adjusted to communicate 115200 Baud Rate. Same baud rate is used on the PC program to synchronize with Voter.

5.3.7.2 Batch Files for Configuration

PC Program must manage the JTAG protocol in order to download bitstream to the FPGA. *Impact*, a Xilinx tool, is used to ease this task. Impact is a PC program that can connect to the JTAG chain with standard Xilinx configuration cables. It can detect devices on the JTAG chain and program them. It has a graphical user interface for manual operations and command line interface for batch operations.

Batch files that use command line interface of the Impact are written. These batch files are executed by the PC program to reconfigure the FPGA. Each batch file has a corresponding bitstream file. For example, the following commands of the batch file reconfigure the FPGA with the *top_final.bit* bitstream file.

```
1) setmode –bscan
2) setCable –p lpt1
3) addDevice –p 1 –file top_final.bit
4) program –p 1
5) quit
```

First line of the batch file instructs Impact to use Boundary Scan (JTAG) interface. Second line selects Parallel Port 1 as the JTAG cable's connection port. Third line selects the first device (i.e. FPGA) on the JTAG chain to configure and assigns top_final.bit as the bitstream file. Fourth line programs the device with the assigned configuration file. The last line exits the command line interface.

The batch files are given in Appendix E (in FTArchitecture/ Borland-Project/ Configurations directory).

5.3.7.3 Running Batch Files from Borland C++ Buider

The batch files of the Impact tool must be called from the Command Prompt. The following command calls the Impact tool and it will run the example_file.cmd batch file.

```
Impact -batch example_file.cmd
```

PC program must be also able to run the above command. This command is executed in Borland C++ Builder by using following class, class variables, and function:

```
SHELLEXECUTEINFO ShExecInfo;

ShExecInfo.cbSize = sizeof(SHELLEXECUTEINFO);
ShExecInfo.fMask = NULL;
ShExecInfo.hwnd = NULL;
ShExecInfo.lpVerb = NULL;
ShExecInfo.lpFile = "impact";
ShExecInfo.lpParameters ="-batch example_file.cmd";
ShExecInfo.lpDirectory = "Configurations";
ShExecInfo.nShow = SW_SHOW; //or SW_HIDE;
ShExecInfo.hInstApp = NULL;

ShellExecuteEx(&ShExecInfo);
```

ShExecInfo is the main class to execute a shell command. *lpFile* parameter define the program that will run. *lpParameters* define options of the program. *lpDirectory* is an additional parameter that enables running command from another directory. Moreover, *nShow* variable is set to *SW_SHOW* to see the outputs of the impact tool. It can be set to *SW_HIDE* in order to hide the outputs of the impact tool. After setting all parameters *ShellExecuteEx* function calls the impact command line interface.

5.3.8 Protocol between PC Program and Voter Module

There is a simple communication protocol between Voter module on the FPGA and PC Program. The errors on module(s) are reported to the PC Program by the Voter. PC makes reconfiguration operations and sends commands to the Voter. An example of communication protocol commands used during fault elimination process is demonstrated in Figure 5-19.

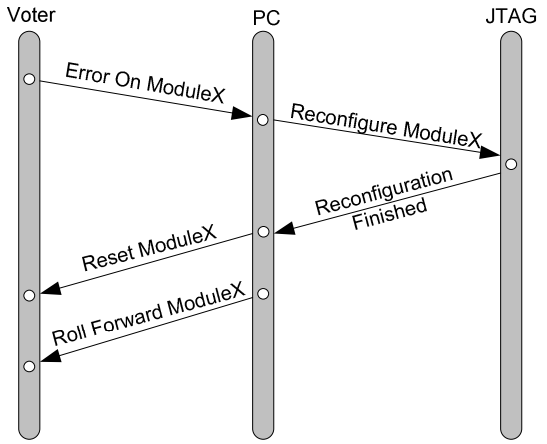

Figure 5-19: An example of Communication Protocol Commands during Error Recovery Operation of a Module

In this example, Voter informs PC Program that ModuleX has erroneous output. Then PC selects a reconfiguration type according to fault elimination algorithm and reconfigures the ModuleX. After reconfiguration process is completed, PC instructs Voter to Reset and Roll Forward corresponding module.

5.3.9 Fault Elimination Algorithm Running on the PC

When an error is found on a redundant module, PC program try to find the source of the error. For each module, the same algorithm runs independently. Briefly working principle of the algorithm is as follows: It first checks if the error is transient. If error is not transient, it successively tries changing bus macro, refreshing configuration memory and loading alternative configurations until error disappears. If error persists, it gives up the tests and passes to the recovery of voter module. Figure 5-20 shows the flow chart of the running algorithm in more detail.

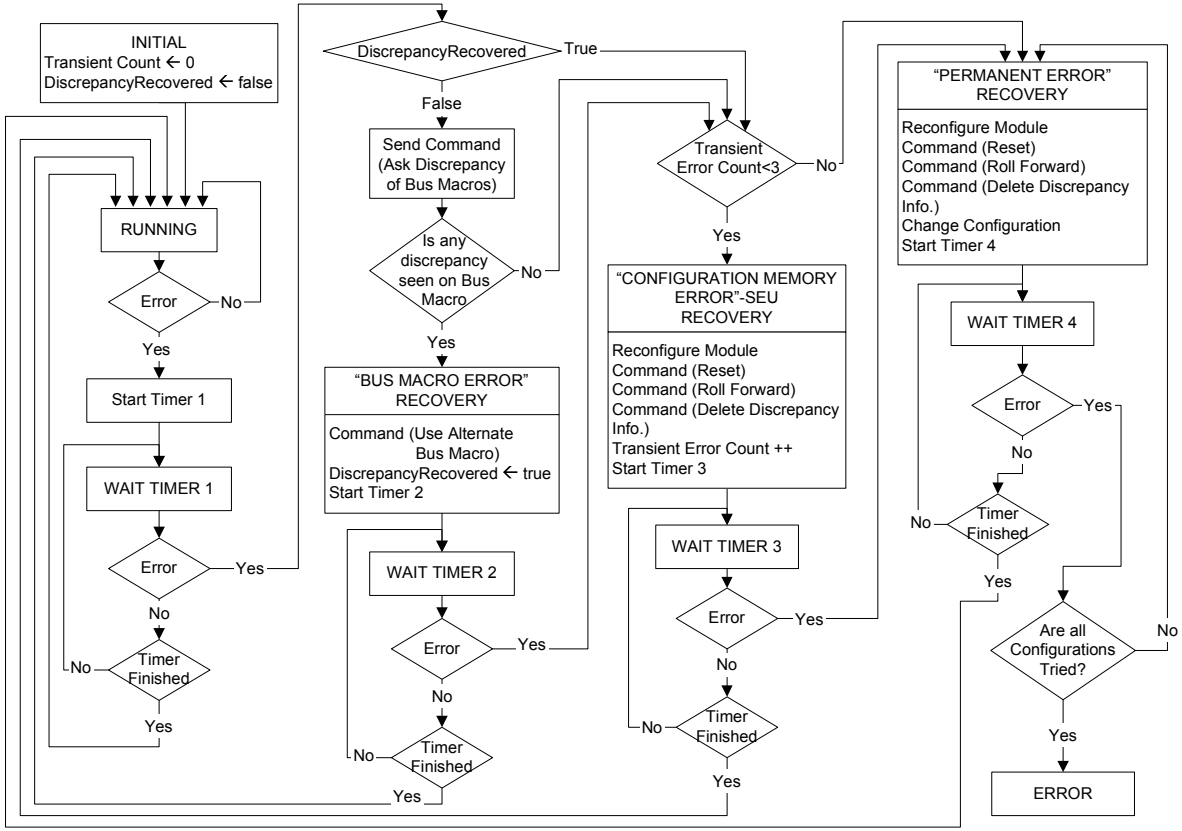

Figure 5-20: Flowchart of Fault Recovery Algorithm that Runs on the PC Program

The algorithm starts with *Initial* state and reset all the variables. Voter periodically sends status output of the module. Algorithm passes to the *Running* state with the first status output. If an error comes at *Running* state, it passes to the *Wait Timer 1* state and waits for second error status. If no more error comes during the Wait Timer 1 state, it returns to the Running state. This timer is necessary to ensure the error is not transient.

If another error comes during Wait Timer 1 state, it starts to check bus macro status. It asks whether a discrepancy is seen between original bus macro and alternative bus macro. If no discrepancy seen, it refreshes the configuration memory (to eliminate SEUs) with the partial bitstream of the module in the *Configuration Memory Error Recovery* state. After refreshing memory, it requests resetting and roll-forwarding operations of the module from the voter. In addition, a counter is incremented that holds number of memory refresh operations. If the counter exceeds three, then further errors are treated as permanent faults.

The errors that cannot be corrected by bus macro altering or memory refreshing are considered as permanent errors. In the *Permanent Error Recovery* state, the FPGA is reconfigured with alternative partial bitstreams of the module. The alternative configuration files reserve empty spaces as described before. When faulty resource falls into the empty space, the error disappears. Therefore, reconfiguration is done with these bitstreams until error disappears. Again, reset and roll-forward operations are done after each reconfiguration process for correct operation. If the fault cannot be eliminated after trying all configurations, algorithm passes to the Error state.

After each recover operation, the algorithm waits on a Timer state (i.e. *Wait Timer 2*, *Wait Timer 3* and *Wait Timer 4* states) to ensure the fault is eliminated. If no error status comes during the Timer states, then algorithm returns to the Running state.

5.3.10 Fault Injection

It is necessary to test the system behaviour in the presence of faults. Faults are artificially injected to test the behaviour of designed system. Fault injection is done by loading an incorrect partial bitstream to the FPGA. Two methods are used to obtain an incorrect partial bitstream. First method directly

modifies a correct partial bitstream. The second method modifies the source VHDL file then synthesized to provide a faulty bitstream.

5.3.10.1 Bitstream Modification

Modifications are done on the functionality of the Configurable Logic Blocks (CLBs). For example, the truth table of Lookup Table (LUT) inside a CLB is changed.

FPGA Editor Tool is used for changing the LUT content. For example, ModOne.ncd file (from Pim directory) is opened with FPGA Editor to inject fault on redundant Module One. ModOne.ncd contains placed and routed design of Module One. A CLB that is configured to use LUT is selected. The attribute of the ncd file is changed to Read-Write from the File → Main Properties menu. Then EditBlock → Begin Editing is selected. The function of the LUT is made visible by selecting Show/Hide Attributes menu as shown in Figure 5-21.

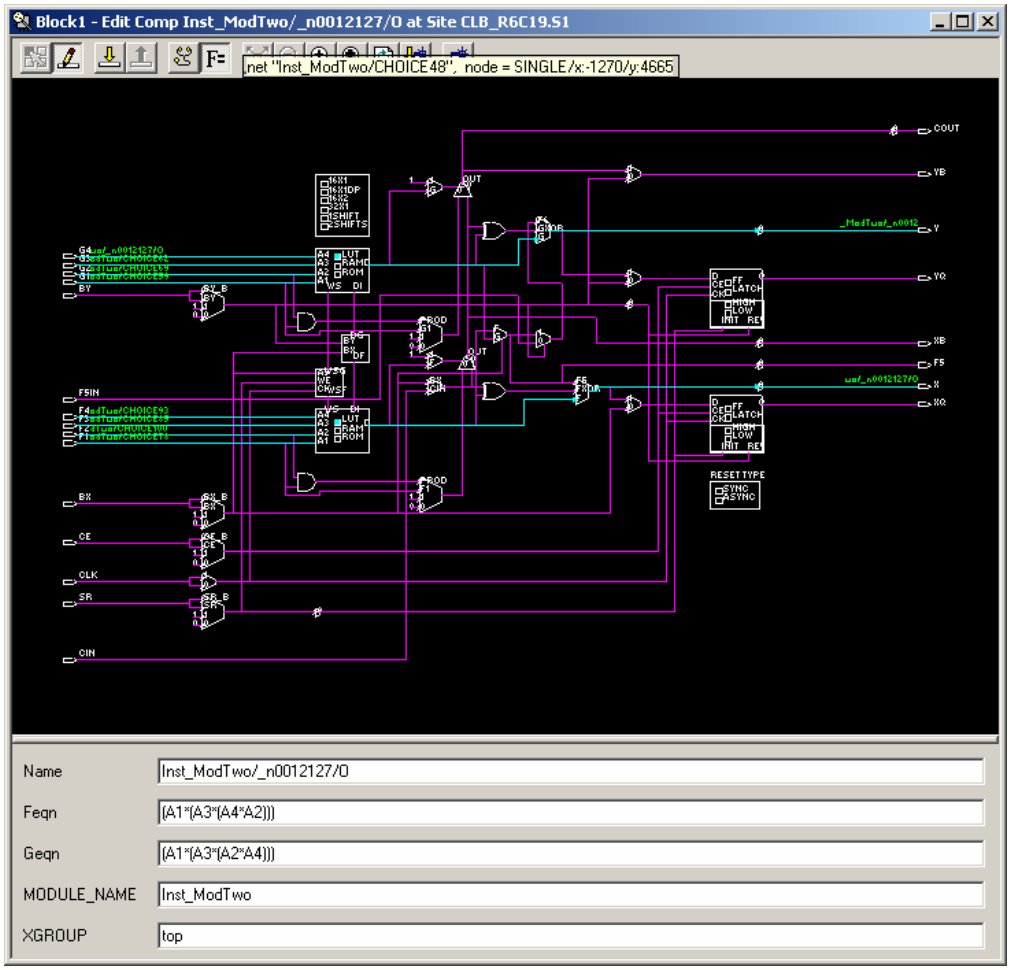

Figure 5-21: Configurable Logic Block in Editing Mode

The functions of the LUTs (i.e. Feqn, Geqn) can be changed to any combination using the input signals. The operators given in Table 5-5 can be used to describe a logical function. For instance, to inject a stuck-at 1 like fault, Geqn (the function of upper LUT) is changed from (A1*(A3*(A4*A2))) to 1. This implies a stuck-at 1 fault on the LUT's function.

Table 5-5: FPGA Editor Symbols and Their Functions

Symbol	Operation
*	Logical AND
+	Logical OR
@	Logical XOR
~	Unary NOT

At last, the ncd file is saved with the “Save Changes and Closes Window” button. At this point, only remaining operation to generate bitstream is running Bitgen tool. For this purpose, the following command is executed to generate a faulty partial bitstream:

```
bitgen -d -g ActiveReconfig:yes ModOne.ncd  modone_faulty.bit
```

5.3.10.1.1 Single Event Upset (SEU) Injection

Bitstream modification is used for injecting SEU like fault on a redundant module. A random CLB that includes logic inside is opened. Then the function of a LUT is changed to reflect a single bit flip. For instance, to inject a SEU like fault Geqn (the function of upper LUT) is modified in R4C28.S0 (Row 4, Column 28 and Slice 0). Geqn is changed from (~A1*(A4*~A3)) to (~A1*(A4*~A3)*~A2). This implies a single bit flip on the LUT’s function. The truth table change of LUT function is shown in Table 5-6.

Table 5-6: Truth Table of LUT Function Before and After a SEU Injection

Inputs				Output Functions	
				Before SEU	After SEU
A1	A2	A3	A4	F1=(~A1*(A4*~A3)*~A2)	F2=(~A1*(A4*~A3))
0	0	0	0	0	0
0	0	0	1	**1**	**0**
0	0	1	0	0	0
0	0	1	1	0	0
0	1	0	0	0	0
0	1	0	1	1	1
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	0	0
1	1	1	0	0	0
1	1	1	1	0	0

5.3.10.1.2 Permanent Fault Injection

Permanent faults are also injected by modification of bitstreams. However, the supervisor program must be aware of permanent fault injection. The reason is the permanent faults are not real; actually, they are only simulation. After a permanent fault injection is done, all reconfigurations must include same permanent fault.

A CLB was selected as victim. Then all configurations that include logic on this CLB were modified. For example in Figure 5-22, first three configurations are modified since the selected CLB include logic inside. However, last configuration is not modified since it maps faulty bitstream in to the empty area (it does not contain logic inside faulty CLB).

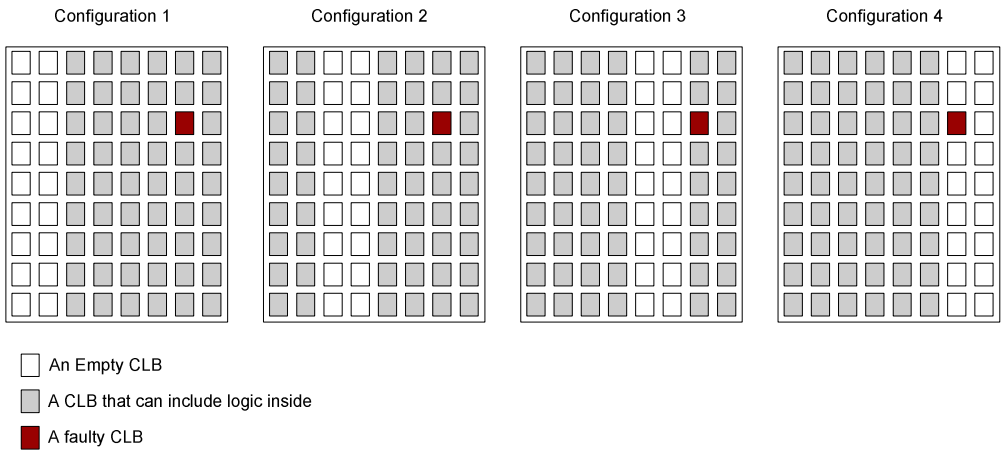

Figure 5-22: A virtual faulty CLB and it is mapping on alternative placements.

5.3.10.2 VHDL Code Modification

A correct VHDL code is changed to include an error. Then it is synthesized to produce a regular netlist file (.ngc). The netlist file is used in active implementation phase of the Modular Design Flow to generate a partial faulty bitstream.

5.3.10.2.1 Bus Macro Fault Injection

Faults were also injected on bus macro connections since they are the only connection path of redundant modules to the Voter. This is achieved by editing VHDL code of redundant modules. Then a partial configuration is produced by using edited VHDL file.

For instance, most significant byte of the state output is inverted as shown in the following code.

```
DataoutModOne(2 downto 0) <= stateOuputModOne(2 downto 0);
DataoutModOne(3) <= not stateOuputModOne(3);
```

After loading generated faulty bitstream, the supervisor program will detect the fault on the bus macro. Then it will try recovery operations such as selecting alternative bus macro.

CHAPTER VI

CONCLUSIONS

6.1 CONCLUSIONS BASED ON THE WORK

The hardware on reconfigurable devices can be used to make computations in parallel. In addition, the versatility of the hardware provides a flexible environment for different applications. Reconfigurable devices achieve high performance with a flexible hardware, which is suitable for all types of digital circuit applications.

In this thesis, the work has been concentrated on runtime reconfigurable architectures. They provide a unique feature, reusability of hardware while system is running. This feature introduces virtual hardware concept similar to virtual memory. Hardware configurations, which are stored on memories, can be loaded to the device whenever needed. Therefore, one device can be used as an infinite hardware source. In this work, application areas that can benefit from runtime reconfiguration (RTR) were surveyed. It was observed that RTR could be also used for speeding up computations and for reducing system costs.

To investigate the feasibility of RTR, a commercially available FPGA (from Xilinx) was used as a runtime reconfigurable platform. The architecture of Xilinx FPGAs was surveyed with a RTR point of view. Then a simple runtime reconfigurable ALU, whose operations can change, was implemented. This design can be used as an initial reference for other runtime reconfigurable designs to implement on Xilinx FPGA.

After achieving RTR with designed simple reconfigurable ALU (explained in Chapter 4), a more complex fault tolerant reconfigurable architecture (explained in Chapter 5) was selected as a case study. The designed architecture is based

on Triple Modular Redundancy (TMR) and it is strengthened by RTR. Triple modular redundancy enables an uninterrupted, fault-tolerant system operation if error occurs on only one module. However, TMR system can breakdown when more than one fault occur on different modules. A system run on FPGA can come across with two different types of faults. First fault type is permanent fault, which may appear due to long life usage. Second fault type is Single Event Upset (SEU), which is encountered frequently on space applications. SEU is a transient fault normally however it may result in a permanent error if configuration memory of the FPGA is RAM based.

Added RTR support has prevented the breakdown of the TMR system. The permanent faults are detected and eliminated on the fly by replacing faulty elements with non-faulty elements. While eliminating the faulty elements, the whole system also remains unaffected by the help of RTR. Furthermore, SEU faults are eliminated by refreshing configuration memory. A high availability is also maintained since faulty modules of the TMR are corrected whenever a fault occurs.

To achieve RTR a PC was used as reconfiguration controller. A PC program was written with Borland C++ Builder for this purpose. The PC Program is also capable of injecting faults to the designed architecture. The faults are injected artificially with the program (by reconfiguration) and the operation of the system is verified.

The design on the FPGA was done with command line tools of Xilinx. The hardware circuits on the FPGA were entered with VHDL. The Xilinx hardware and software tools allowed designing such system. The hardware has some restrictions however; it is possible to design a reconfigurable architecture. The software tools are in their infancy and they tend to improve with the benefits obtained from reconfigurable computing. Later, designed fault tolerant architecture can be adapted to other runtime reconfigurable devices easily.

Consequently, RTR provides significant benefits for digital hardware implementations. In the future, more applications will take advantage of runtime reconfiguration. Therefore, the devices that are capable of making runtime reconfiguration will most probably increase. In this work, it has been proven that a RTR can be achieved with current technology. In addition, a fault tolerant architecture that is highly reliable is provided.

6.2 RECOMMENDED FUTURE WORKS

Self-Reconfiguration

Designed system can be converted to a self-reconfiguring platform. Thus, the PC used as a reconfiguration controller can be removed from the system and replaced by a part of FPGA. This solution requires an embedded memory and embedded configuration controller. ICAP port can be also used by embedded configuration controller. Note that fault tolerant memory architecture is necessary for this system.

New Bus Macro Design

Xilinx did not publish bus macro structures for the new generation devices such as Spartan 3 and Virtex 4 yet. Therefore, current bus macro structure used in the designs is not suitable for these devices. Some researches concentrated for new bus macro architecture [57]. These researches implement slice based bus macros. A new device family with new bus macro architecture can be used in future works.

Automated Design

All VHDL codes are edited three times in the current structure of the system, since fault tolerance is maintained by three identical circuits. There is a need for automation for the generation of TMR structure to decrease intervention of the user. The user must give only the design then the rest of the operations must be made by the batch files. Since generation of such framework is very time consuming, it is left as a future work.

Self-Checking

The errors on the voter module can be detected by Concurrent Error Detection (CED) circuits. Embedding CED circuit on the voter will increase the reliability of the system.

REFERENCES

[1] Gericota M.G.; Alves G.R.; Silva M.L.; Ferreira J.M., “Programmable Logic Devices: A Test Approach for the Input/Output Blocks and Pad-to-Pin Interconnections”, 4th IEEE Latin-American Test Workshop (LATW'2003), pp. 72-77, February 2003

[2] Compton K.; Hauck S., "Reconfigurable Computing: A Survey of Systems and Software", ACM Computing Surveys, Vol. 34, No. 2. pp. 171-210. June 2002

[3] Hartenstein, R., "A decade of reconfigurable computing: a visionary retrospective," Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings, pp.642-649, 2001

[4] Rasmussen S.; Silfverberg T., “Reconfigurable Computing Array”, Master Thesis, Department of Electroscience - Lund Institute of Technology, 2002

[5] Hartenstein, R.W.; Herz M.; Hoffmann T.; Nageldinger U., “Synthesis and Domain-specific Optimization of KressArray-based Reconfigurable Computing Engines”, Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays, pp. 222-232 2000

[6] Hartenstein, R.W.; Kress, R.; Reinig, H., "A dynamically reconfigurable wavefront array architecture for evaluation of expressions," Application Specific Array Processors, 1994. Proceedings., International Conference on , pp.404-414, 22-24 Aug 1994

[7] Hannig, F.; Dutta, H.; Teich, J., "Regular mapping for coarse-grained reconfigurable architectures," Acoustics, Speech, and Signal Processing, 2004. Proceedings. (ICASSP '04). IEEE International Conference on , vol.5, pp. 57-60, 17-21 May 2004

[8] Upegui A.; Moeckel R.; Dittrich E.; Ijspeert A.; Sanchez E., "An FPGA Dynamically Reconfigurable Framework for Modular Robotics", Workshop on Dynamically Reconfigurable Systems at the 18th International Conference on Architecture of Computing Systems, ARCS '05, pp. 83-89, Innsbruck, Austria, March 14-17, 2005

[9] Bossuet, L.; Gogniat, G.; Burleson, W., "Dynamically configurable security for SRAM FPGA bitstreams," Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, pp. 146-153, 26-30 April 2004

[10] Fong, R.J.; Harper, S.J.; Athanas, P.M., "A versatile framework for FPGA field updates: an application of partial self-reconfiguration," Rapid Systems Prototyping, 2003. Proceedings. 14th IEEE International Workshop on , pp. 117- 123, 9-11 June 2003

[11] Tessier, R.; Burleson, W., "Reconfigurable computing for digital signal processing: A survey," Journal of VLSI Signal Processing, vol. 28, no. 1, pp. 7-27, May/June 2001

[12] Resano, J.; Mozos, D.; Verkest, D.; Catthoor, F.; Vernalde S., "Specific scheduling support to minimize the reconfiguration overhead of dynamically reconfigurable hardware" Design Automation Conference, 2004. Proceedings. 41st , pp. 119-124, 2004

[13] Walder, H.; Steiger, C.; Platzner, M., "Fast online task placement on FPGAs: free space partitioning and 2D-hashing," Parallel and Distributed Processing Symposium, 2003. Proceedings. International , pp. 178-185, 22-26 April 2003

[14] Ghiasi, S.; Sarrafzadeh, M., "Optimal reconfiguration sequence management [FPGA runtime reconfiguration]," Design Automation Conference, 2003. Proceedings of the ASP-DAC 2003. Asia and South Pacific , pp. 359-365, 21-24 Jan. 2003

[15] Lattice Semiconductor Corporation, “ORCA Series 4 FPGA Configuration”, TN1013, 2004

[16] Scandaliaris, J.; Moreno, J.M.; Cabestany, J., “Specification of D_FPGA Characteristics”, http://www.reconf.org/ accessed at 2006, RECONF (a European Commission IST Programme) Project Report

[17] Donthi, S.; Haggard, R.L., "A survey of dynamically reconfigurable FPGA devices," System Theory, 2003. Proceedings of the 35th Southeastern Symposium on , pp. 422- 426, 16-18 March 2003

[18] LLanos C.; Jacobi R.P.; Rincón M.A.; Hartenstein R.W., "A Dynamically Reconfigurable System for Space-Efficient Computation of the FFT", Proceedings. International Conference on Reconfigurable Computing and FPGAs 2004 - ReConFig'04, pp 360-369, Colima, Mexico, 2004

[19] Nascimento, P.S.B.; Maciel, P.R.M.; Lima, M.E.; Sant'ana, R.E.; Filho, A.G.S., "A partial reconfigurable architecture for controllers based on Petri nets," Integrated Circuits and Systems Design, 2004. SBCCI 2004. 17th Symposium on , pp. 16-21, 7-11 Sept. 2004

[20] Ullmann, M.; Huebner, M.; Grimm, B.; Becker, J., "An FPGA run-time system for dynamical on-demand reconfiguration," Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International , pp. 135-142, 26-30 April 2004

[21] Jianwen, L.; Chuen, J.C., "Partially reconfigurable matrix multiplication for area and time efficiency on FPGAs," Digital System Design, 2004. DSD 2004. Euromicro Symposium on , pp. 244-248, 31 Aug.-3 Sept. 2004

[22] Upegui A.; Sanchez E., "Evolving hardware by dynamically reconfiguring Xilinx FPGAs", Evolvable Systems: From Biology to Hardware, LNCS, vol. 3637, pp. 56-65, 2005.

[23] Hollingworth, G.; Smith, S.; Tyrrell, A., "Safe intrinsic evolution of Virtex devices," Evolvable Hardware, 2000. Proceedings. The Second NASA/DoD Workshop on , pp.195-202, 2000

[24] Berthelot, F.; Nouvel, F.; Houzet, D., "Partial and dynamic reconfiguration of FPGAs: a top down design methodology for an automatic implementation," Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International , pp. 436-439, 25-29 April 2006

[25] Berthelot, F.; Nouvel, F.; Houzet, D., "Design methodology for runtime reconfigurable FPGA: from high level specification down to implementation," Signal Processing Systems Design and Implementation, 2005. IEEE Workshop on , pp. 497-502, 2-4 Nov. 2005

[26] Faust, O.; Sputh, B.; Nathan, D.; Rezgui, S.; Weisensee, A.; Allen, A., "A single-chip supervised partial self-reconfigurable architecture for software defined radio," Parallel and Distributed Processing Symposium, 2003. Proceedings. International , pp. 191-196, 22-26 April 2003

[27] J. A. Hennessy, D. L. Patterson, “Computer Architecture: A Quantitative Approach”, Morgan Kauffmann Publishers, 1990

[28] Gericota, M.G.; Alves, G.R.; Silva, M.L.; Ferreira, J.M., "Active replication: towards a truly SRAM-based FPGA on-line concurrent testing," On-Line Testing Workshop, 2002. Proceedings of the Eighth IEEE International , pp. 165-169, 2002

[29] Emmert, J.; Stroud, C.; Skaggs, B.; Abramovici, M., "Dynamic fault tolerance in FPGAs via partial reconfiguration," Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on , pp.165-174, 2000

[30] Wei-Je Huang; McCluskey, E.J., "Column-Based Precompiled Configuration Techniques for FPGA," Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on , pp. 137-146, 2001

[31] Xilinx Inc., “Spartan-IIE 1.8V FPGA Family: Complete Data Sheet”, Xilinx DS077, 2004

[32] Xilinx Inc., “Virtex Series Configuration Architecture User Guide”, Xilinx XAPP 151 v1.7, 2004

[33] Xilinx Inc., “Logicore OPB HWICAP Specification”, Xilinx DS 280, 2004

[34] Xilinx Inc., “Spartan-3 Advanced Configuration Architecture”, Xilinx XAPP 452 v1.0, 2004

[35] Xilinx Inc., “Two Flows for Partial Reconfiguration: Module Based or Difference Based”, Xilinx XAPP 290 v1.2, 2004.

[36] Xilinx Inc., “Virtex FPGA Series Configuration and Readback”, Xilinx XAPP 138 v2.8, 2005

[37] Xilinx Inc., “Using a Microprocessor to Configure Xilinx FPGAs via Slave Serial or SelectMAP Mode”, Xilinx XAPP 502 v1.4, 2002

[38] Xilinx Inc., “Configuration and Readback of Virtex FPGAs Using (JTAG) Boundary Scan”, Xilinx XAPP 139 v1.6, 2003

[39] Xilinx Inc., “JBits SDK 3 for Virtex-II Documentation / JBits Tutorial “, 2003

[40] Xilinx Inc., “Development System Reference Guide - ISE 5”, Xilinx

[41] Mermoud G., “A Module-Based Dynamic Partial Reconfiguration tutorial”, http://ic2.epfl.ch/~gmermoud/files/publications/DPRtutorial.pdf accessed at 2006, Ecole Polytechnique Fédérale de Lausanne, 2004

[42] Braeckman G.; Branden G.V.; Touhafi A.; Dessel G.V. “Module Based Partial Reconfiguration: a quick tutorial”, http://iwt5.ehb.be/typo3/index.php?id=415 accessed at 2006, Erasmushogeschool IWT Department, 2004,

[43] Vigander S., “Evolutionary Fault Repair of Electronics in Space Applications”, Centre for Computational Neuroscience and Robotics (CCNR) at the University of Sussex, Project Report, 2001

[44] Lima, F.; Carro, L.; Reis, R., "Designing fault tolerant systems into SRAM-based FPGAs," Design Automation Conference, 2003. Proceedings , pp. 650-655, 2-6 June 2003

[45] Graham P.; Caffrey M.; Zimmerman J.; Johnson D.E.; Sundararajan P.; Patterson C., "Consequences and Categories of SRAM FPGA Configuration SEUs," Proceedings of the Military and Aerospace Applications of Programmable Logic Devices (MAPLD), Washington DC, September 2003

[46] Pontarelli, S.; Cardarilli, G.C.; Malvoni, A.; Ottavi, M.; Re, M.; Salsano, A., "System-on-chip oriented fault-tolerant sequential systems implementation methodology," Defect and Fault Tolerance in VLSI Systems, 2001. Proceedings. 2001 IEEE International Symposium on , pp.455-460, 2001

[47] Xilinx Inc., “Correcting Single-Event Upsets Through Virtex Partial Configuration”, Xilinx XAPP 216, 2000

[48] Gokhale, M.; Graham, P.; Johnson, E.; Rollins, N.; Wirthlin, M., "Dynamic reconfiguration for management of radiation-induced faults in FPGAs," Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International , pp. 145-150, 26-30 April 2004

[49] Xilinx Inc., “Triple Module Redundancy Design Techniques for Virtex FPGAs”, Xilinx XAPP 197 v1.0, 2001

[50] DeMara, R.F.; Kening Zhang, "Autonomous FPGA fault handling through competitive runtime reconfiguration," Evolvable Hardware, 2005. Proceedings. 2005 NASA/DoD Conference on , pp. 109-116, 29 June-1 July 2005

[51] Shu-Yi Yu; McCluskey, E.J., "Permanent fault repair for FPGAs with limited redundant area," Defect and Fault Tolerance in VLSI Systems, 2001. Proceedings. 2001 IEEE International Symposium on , vol., no.pp.125-133, 2001

[52] Kenterlis P.; Kranitis N.; Paschalis A.; Gizopoulos D.; Psarakis M.,"A low-cost SEU fault emulation platform for SRAM-based FPGAs," On-Line Testing Symposium, 2006. IOLTS 2006. 12th IEEE International , pp. 235-241 , 10-12 July 2006

[53] Digilent Inc., “Digilent D2-SB System Board Reference Manual”, http://www.digilentinc.com, June 2004

[54] Xilinx Inc., “PicoBlaze 8-Bit Microcontroller for Virtex-E and Spartan-II/IIE Devices”, Xilinx XAPP 213 v2.1, 2003

[55] Digilent Inc., “Digilent DIO1 Manual”, http://www.digilentinc.com, May 2004

[56] Bobda C.; Huebner M.; Niyonkuru A.; Bloget B.; Majer M.; Ahmedinia A., “Designing Partial and Dynamically Reconfigurable Applications on Xilinx Virtex-II FPGAs using HandelC”, University of Erlangen-Nuremberg, Germany, Technical Report 03-2004

[57] Sedcole N.P., “Reconfigurable Platform-Based Design in FPGAs for Video Image Processing”, PhD Thesis, University of London, 2006

APPENDIX A

PCB AND SCHEMATICS OF THE RS232 CIRCUIT

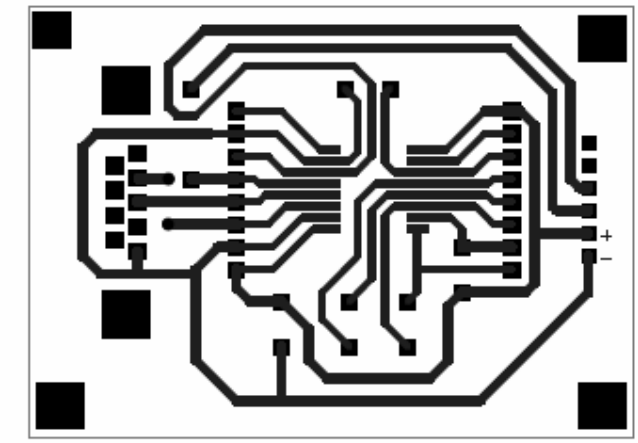

Figure A-1: Top Layer PCB of RS232 Circuit

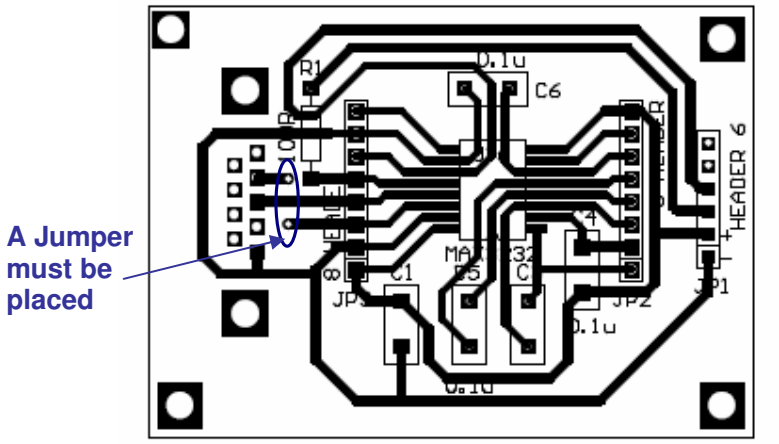

Figure A-2: Top Overlay PCB of RS232 Circuit

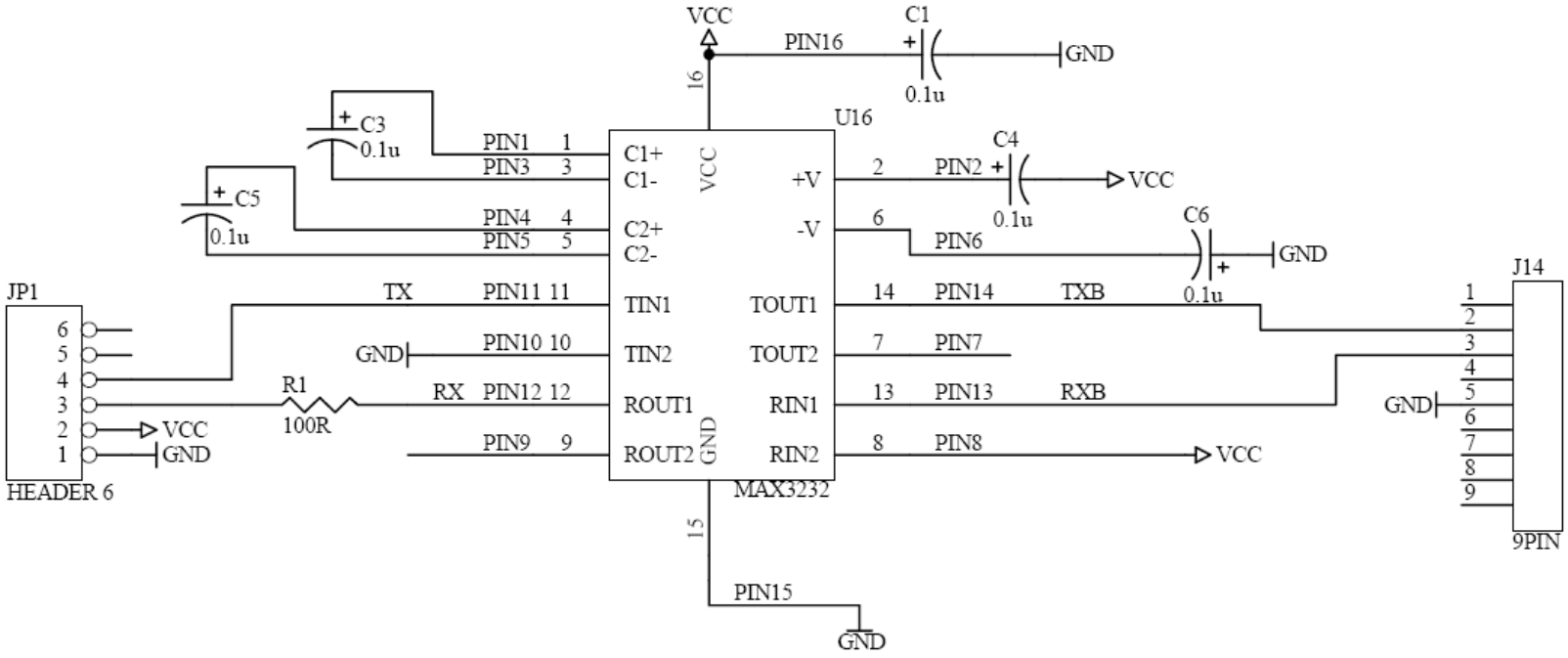

Figure A-3: Schematic of RS232 Circuit

APPENDIX B

SIMULATION OF TWO ROLL FORWARDING METHODS

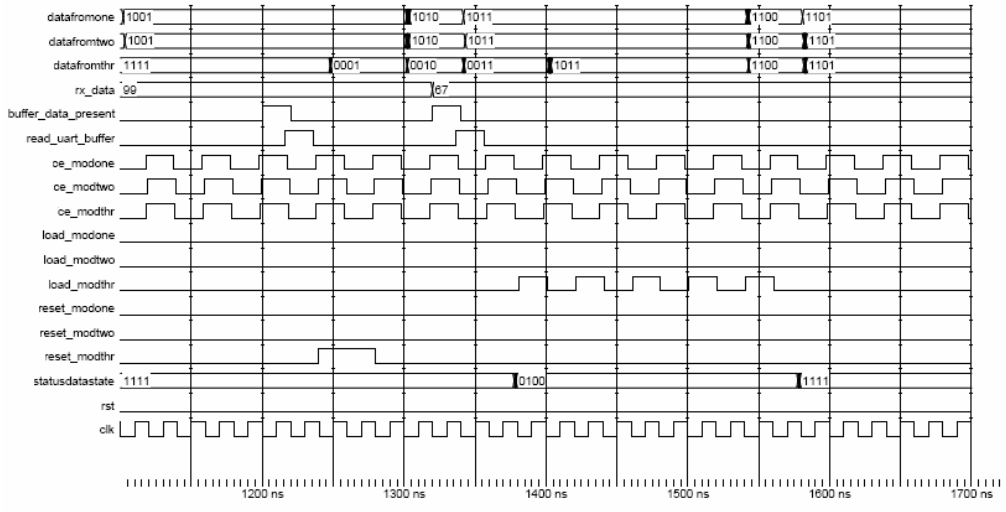

Figure B-1: Simulation of Roll Forwarding Method 1 (Constant Frequency Rate)

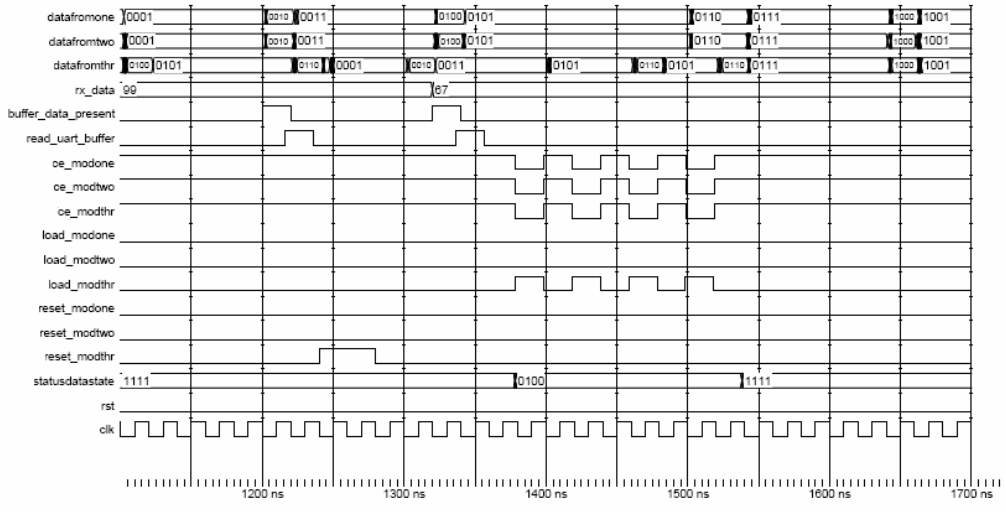

Figure B-2: Simulation of Roll Forwarding Method 2 (Variable Frequency Rate)

APPENDIX C

USER CONSTRAINT FILE OF THE TMR DESIGN

User Constraint File for the First Configuration of Module One

```
# Start of PACE Area Constraints
AREA_GROUP "AG_Inst_Voter" RANGE = CLB_R1C32:CLB_R28C42 ;
AREA_GROUP "AG_Inst_Voter" RANGE = TBUF_R1C32:TBUF_R8C42 ;
INST "Inst_Voter" AREA_GROUP = "AG_Inst_Voter" ;
AREA_GROUP "AG_Inst_Voter" MODE = RECONFIG ;

AREA_GROUP "AG_Inst_ModOne" RANGE = CLB_R1C24:CLB_R28C31 ;
AREA_GROUP "AG_Inst_ModOne" RANGE = TBUF_R1C24:TBUF_R28C31 ;
INST "Inst_ModOne" AREA_GROUP = "AG_Inst_ModOne" ;
AREA_GROUP "AG_Inst_ModOne" MODE = RECONFIG ;

AREA_GROUP "AG_Inst_ModTwo" RANGE = CLB_R1C16:CLB_R28C23 ;
AREA_GROUP "AG_Inst_ModTwo" RANGE = TBUF_R1C16:TBUF_R28C23 ;
INST "Inst_ModTwo" AREA_GROUP = "AG_Inst_ModTwo" ;
AREA_GROUP "AG_Inst_ModTwo" MODE = RECONFIG ;

AREA_GROUP "AG_Inst_ModThr" RANGE = CLB_R1C8:CLB_R28C15 ;
AREA_GROUP "AG_Inst_ModThr" RANGE = TBUF_R1C8:TBUF_R28C15 ;
INST "Inst_ModThr" AREA_GROUP = "AG_Inst_ModThr" ;
AREA_GROUP "AG_Inst_ModThr" MODE = RECONFIG ;

#AREA_GROUP "AG_Inst_Spare" RANGE = CLB_R1C1:CLB_R28C7 ;
#AREA_GROUP "AG_Inst_Spare" RANGE = TBUF_R1C1:TBUF_R28C7 ;
#INST "Inst_Spare" AREA_GROUP = "AG_Inst_Spare" ;
#AREA_GROUP "AG_Inst_Spare" MODE = RECONFIG ;


# Start of PACE Prohibit Constraints

CONFIG PROHIBIT=CLB_R*C24;
CONFIG PROHIBIT=CLB_R*C25;


# Start of Locking Constraints

INST "Internal_Gnd_Voter" AREA_GROUP = "AG_Inst_Voter" ;
INST "Internal_Vcc_Voter" AREA_GROUP = "AG_Inst_Voter" ;
INST "Internal_Gnd_ModOne" AREA_GROUP = "AG_Inst_ModOne" ;
INST "Internal_Vcc_ModOne" AREA_GROUP = "AG_Inst_ModOne" ;
INST "Internal_Gnd_ModTwo" AREA_GROUP = "AG_Inst_ModTwo" ;
INST "Internal_Vcc_ModTwo" AREA_GROUP = "AG_Inst_ModTwo" ;
INST "Internal_Gnd_ModThr" AREA_GROUP = "AG_Inst_ModThr" ;
INST "Internal_Vcc_ModThr" AREA_GROUP = "AG_Inst_ModThr" ;
#INST "Internal_Gnd_Spare" AREA_GROUP = "AG_Inst_Spare" ;
#INST "Internal_Vcc_Spare" AREA_GROUP = "AG_Inst_Spare" ;
```

```
INST "busModOnetoVoter" LOC = "TBUF_R1C28.0"  ;
INST "busVotertoModOne/bus1" LOC = "TBUF_R2C28.0"  ;
INST "busVotertoModOne/bus2" LOC = "TBUF_R3C28.0"  ;

INST "busModTwotoVoter" LOC = "TBUF_R4C20.0"  ;
INST "busVotertoModTwo/bus1" LOC = "TBUF_R5C20.0"  ;
INST "busVotertoModTwo/bus2" LOC = "TBUF_R6C20.0"  ;

INST "busModThrtoVoter" LOC = "TBUF_R7C12.0"  ;
INST "busVotertoModThr/bus1" LOC = "TBUF_R8C12.0"  ;
INST "busVotertoModThr/bus2" LOC = "TBUF_R9C12.0"  ;


INST "busModOnetoVoter_alt" LOC = "TBUF_R10C28.0"  ;
INST "busVotertoModOne_alt/bus1" LOC = "TBUF_R11C28.0"  ;
INST "busVotertoModOne_alt/bus2" LOC = "TBUF_R12C28.0"  ;

INST "busModTwotoVoter_alt" LOC = "TBUF_R13C20.0"  ;
INST "busVotertoModTwo_alt/bus1" LOC = "TBUF_R14C20.0"  ;
INST "busVotertoModTwo_alt/bus2" LOC = "TBUF_R15C20.0"  ;

INST "busModThrtoVoter_alt" LOC = "TBUF_R16C12.0"  ;
INST "busVotertoModThr_alt/bus1" LOC = "TBUF_R17C12.0"  ;
INST "busVotertoModThr_alt/bus2" LOC = "TBUF_R18C12.0"  ;

#INST "busSparetoVoter" LOC = "TBUF_R7C4.0"  ;
#INST "busVotertoSpare/bus1" LOC = "TBUF_R8C4.0"  ;
#INST "busVotertoSpare/bus2" LOC = "TBUF_R12C4.0"  ;

INST "bufg_clk" LOC = "GCLKBUF2" ;


#PACE: Start of I/O Pin Assignments

NET "CathodeOutputs<0>" LOC = "P134";
NET "CathodeOutputs<1>" LOC = "P136";
NET "CathodeOutputs<2>" LOC = "P139";
NET "CathodeOutputs<3>" LOC = "P141";
NET "CathodeOutputs<4>" LOC = "P148";
NET "CathodeOutputs<5>" LOC = "P150";
NET "CathodeOutputs<6>" LOC = "P152";
NET "CathodeOutputs<7>" LOC = "P161";
NET "AnodeOutputs<0>" LOC = "P113";
NET "AnodeOutputs<1>" LOC = "P115";
NET "AnodeOutputs<2>" LOC = "P120";
NET "AnodeOutputs<3>" LOC = "P122";
NET "serialin" LOC  = "P127"  ;
NET "serialout"  LOC = "P126"  ;
NET "clk" LOC = "P182";

#INST "*" IOB=FALSE;
```

APPENDIX D

PACE AND FPGA EDITOR VIEW OF THE TMR DESIGN

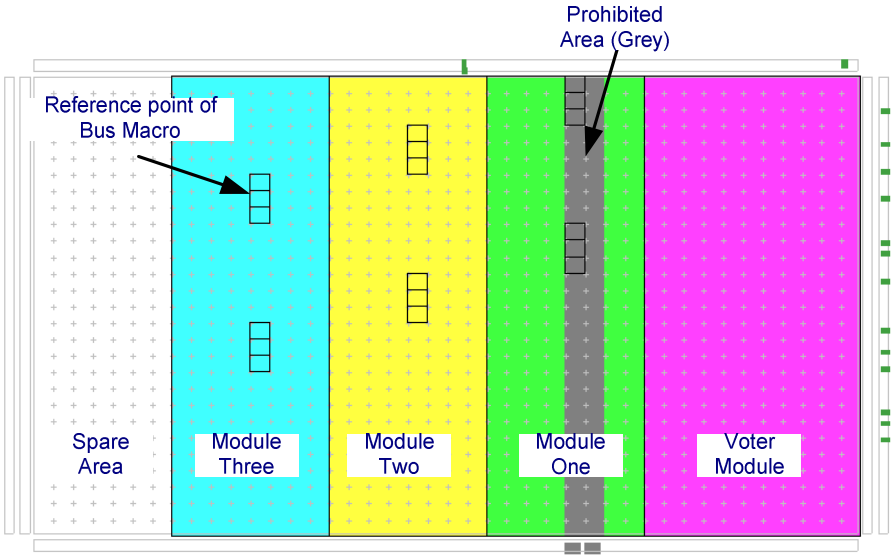

Figure D-1: Module Placements of the TMR Design (Snapshot is taken with PACE)

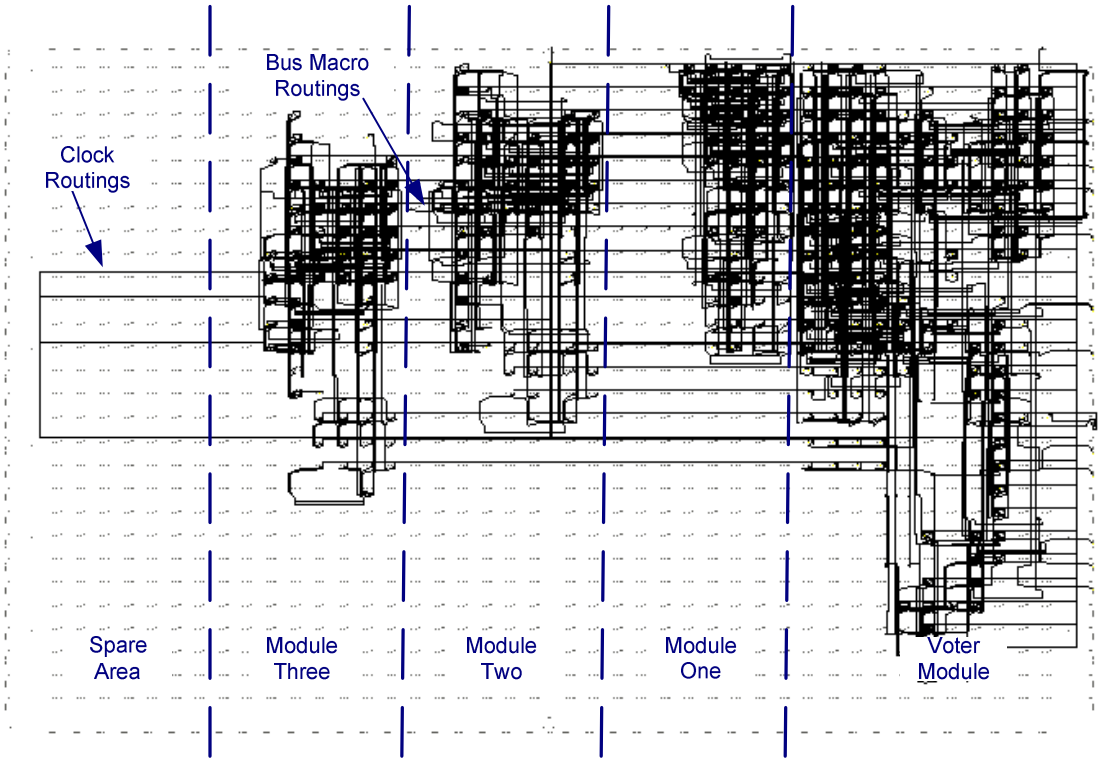

Figure D-2: FPGA Editor View of TMR Design

APPENDIX E

SOURCE FILES OF DESIGNED ARCHITECTURES

A CD-ROM is enclosed to the back cover of the thesis. It contains the source codes, batch files, and generated files of the designed architectures. The contents of the CDROM are given in Table E-1.

Table E-1: The Directories and Files in the CDROM

Reconfig-ALU/	Top level directory of Reconfigurable ALU (Chapter 4)			
	Bitstreams/	Contains Final Partial Bitstreams and a Full Bitstream		
	BusMacro/	Contains angle-delimiter bus macro for Spartan 2E		
	Implementation/	Contains Implementation Flow Files and Folders (Implementation phase of Modular Design Flow (MDF) is done in this folder) Also contains top.ucf and batch files of the MDF.		
		left_add/	Partial implementation of left adder module (Active implementation phase of MDF is done in this folder)	
		left_mult/	Partial implementation of left multiplier module (Active implementation phase of MDF is done in this folder)	
		left_sub/	Partial implementation of left adder module (Active implementation phase of MDF is done in this folder)	
		Pim/	Published placed and routed files of partial configurations	
			left/	Placed and routed file of left module
			right/	Placed and routed file of right module
		right/	Partial implementation of right module (Active implementation phase of MDF is done in this folder)	
		top_final/	Final assembly phase of MDF is done in this folder	
		top_initial/	Initial budgeting phase of MDF is done in this folder	
		Top.ucf	User constraint file for the overall design	
		1-Initial.bat	The batch file for the initial phase of MDF	
		2-Active.bat	The batch file for the active implementation phase of MDF	
		3-Assemble.bat	The batch file for the assemble phase of MDF	

Table E-1 cont'd: The Directories and Files in the CDROM

Reconfig-ALU/	**Synthesis/**	Contains Xilinx ISE projects and VHDL files for partial modules and top module		
		left_add/	Left adder module project and VHDL file for synthesis	
		left_mult/	Left multiplier module project and VHDL file for synthesis	
		left_sub/	Left subtractor module project and VHDL file for synthesis	
		right/	Right module project and VHDL file for synthesis	
		top /	Top module project and VHDL file for synthesis	
	Borland-Project/	Contains Borland C++ Builder Files		
		ReconfigALU.exe	Executable for reconfiguration program	
		Configurations/	Contains Impact batch files and bitstreams	
FTArchitecture/	Top level directory of Fault Tolerant Architecture (Chapter 5)			
	FinalBitstreams/	Contains Final Partial Bitstreams and a Full Bitstream		
	Macros/	Contains angle-delimiter bus macro for Spartan 2E		
	Ucf/	Contains user constraints files for each individual modules		
	Implementation/	Contains Implementation Flow Files and Folders (Implementation phase of Modular Design Flow (MDF) is done in this folder) Also contains top.ucf and batch files of the MDF. In the below folders, X refers to 1,2 ... for alternative configurations. X refers to pe1,pe2 ... for corresponding permanent error including alternative configurations. X refers to SEU for single event upset including configuration. X refers to BME for bus macro error including configuration.		
		Bat/	Contains batch files (for the reset operation)	
		Modone_X/	Partial implementation of Module One (Active implementation phase of MDF is done in this folder)	
		Modtwo_X/	Partial implementation of left multiplier module (Active implementation phase of MDF is done in this folder)	
		Modthr_X/	Partial implementation of left adder module (Active implementation phase of MDF is done in this folder)	
		Voter_1/	Partial implementation of left adder module (Active implementation phase of MDF is done in this folder)	
		Pim/	Published placed and routed files of partial configurations	
			ModOne/	Placed and routed file of Module One
			ModTwo/	Placed and routed file of Module Two
			ModThr/	Placed and routed file of Module Three
			Voter/	Placed and routed file of left module

Table E-1 cont'd: The Directories and Files in the CDROM

FTArchitecture/	**Implementation/**	**Top.ucf**	User constraint file for the overall design
		top_final/	Final assembly phase of MDF is done in this folder
		top_initial/	Initial budgeting phase of MDF is done in this folder
		0-Reset.bat	Deletes all generated files and copies batch files from the /bat directory
		1-Initial.bat	The batch file for the initial phase of MDF
		2-Active.bat	The batch file for the active implementation phase of MDF
		3-Assemble.bat	The batch file for the assemble phase of MDF
	Synthesis/	Contains Xilinx ISE projects and VHDL files for partial modules and top module	
		Modone_1/	Module One project and VHDL file for synthesis
		Modtwo_1/	Module Two project and VHDL file for synthesis
		Modthr_1/	Module Three project and VHDL file for synthesis
		Modone_bme/	Module One project and VHDL file that contains bus macro error for synthesis
		Modtwo_bme/	Module Two project and VHDL file that contains bus macro error for synthesis
		Modthr_bme/	Module Three project and VHDL file that contains bus macro error for synthesis
		Voter_1/	Voter module project and VHDL file for synthesis
		top /	Top module project and VHDL file for synthesis
	Borland-Project/	Contains Borland C++ Builder Files	
		Project1.exe	Executable for reconfiguration management program
		Configurations/	Contains Impact batch files and bitstreams