

Electronic Theses and Dissertations, 2004-2019

2007

A Multi-layer Fpga Framework Supporting Autonomous Runtime Partial Reconfiguration

Heng Tan
University of Central Florida

 Part of the [Computer Engineering Commons](#)
Find similar works at: <https://stars.library.ucf.edu/etd>
University of Central Florida Libraries <http://library.ucf.edu>

This Doctoral Dissertation (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Electronic Theses and Dissertations, 2004-2019 by an authorized administrator of STARS. For more information, please contact STARS@ucf.edu.

STARS Citation

Tan, Heng, "A Multi-layer Fpga Framework Supporting Autonomous Runtime Partial Reconfiguration" (2007). *Electronic Theses and Dissertations, 2004-2019*. 3377.
<https://stars.library.ucf.edu/etd/3377>

**A MULTI-LAYER FPGA FRAMEWORK SUPPORTING AUTONOMOUS
RUNTIME PARTIAL RECONFIGURATION**

by

HENG TAN

B.S. Dalian University of Technology, 2000

M.S. University of Central Florida, 2004

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the School of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term
2007

Major Professor: Ronald F. DeMara

© 2007 Heng Tan

ABSTRACT

Partial reconfiguration is a unique capability provided by several Field Programmable Gate Array (FPGA) vendors recently, which involves altering part of the programmed design within an SRAM-based FPGA at run-time. In this dissertation, a *Multilayer Runtime Reconfiguration Architecture (MRRA)* is developed, evaluated, and refined for Autonomous Runtime Partial Reconfiguration of FPGA devices. Under the proposed MRRA paradigm, FPGA configurations can be manipulated at runtime using on-chip resources. Operations are partitioned into Logic, Translation, and Reconfiguration layers along with a standardized set of Application Programming Interfaces (APIs). At each level, resource details are encapsulated and managed for efficiency and portability during operation. An MRRA mapping theory is developed to link the general logic function and area allocation information to the device related physical configuration level data by using mathematical data structure and physical constraints. In certain scenarios, configuration bit stream data can be read and modified directly for fast operations, relying on the use of similar logic functions and common interconnection resources for communication. A corresponding logic control flow is also developed to make the entire process autonomous.

Several prototype MRRA systems are developed on a Xilinx Virtex II Pro platform. The Virtex II Pro on-chip PowerPC core and block RAM are employed to manage control operations while multiple physical interfaces establish and supplement autonomous reconfiguration capabilities. Area, speed and power optimization techniques are developed based on the developed Xilinx prototype. Evaluations and analysis of these prototype and techniques are performed on a number of benchmark and hashing algorithm case studies. The results indicate

that based on a variety of test benches, up to 70% reduction in the resource utilization, up to 50% improvement in power consumption, and up to 10 times increase in run-time performance are achieved using the developed architecture and approaches compared with Xilinx baseline reconfiguration flow.

Finally, a Genetic Algorithm (GA) for a FPGA fault tolerance case study is evaluated as a ultimate high-level application running on this architecture. It demonstrated that this is a hardware and software infrastructure that enables an FPGA to dynamically reconfigure itself efficiently under the control of a soft microprocessor core that is instantiated within the FPGA fabric. Such a system contributes to the observed benefits of intelligent control, fast reconfiguration, and low overhead.

To my parents

ACKNOWLEDGMENTS

I would like to thank NASA, for their financial support and the opportunity to work with such advanced and exciting technology. I would like to thank Dr. DeMara, for his many technical and editorial suggestions that have helped shaping this work and his constant encouragement and support during the research. I would like to thank the committee members who have taken the time to review and comment on this dissertation. But most of all I would like to thank my parents, who raised me, give me unconditional love and constantly support and encourage me to pursue my dream even when I am far way from home, from my country.

TABLE OF CONTENTS

TABLE OF CONTENTS.....	vii
LIST OF FIGURES	x
LIST OF TABLES.....	xii
LIST OF TABLES.....	xii
CHAPTER 1: INTRODUCTION.....	1
1.1 Overview.....	1
1.2 Partial Reconfiguration Technology.....	4
1.3 Need for Autonomous Partial Reconfiguration	6
1.4 Dissertation Outline	9
1.5 Contribution of Dissertation	11
CHAPTER 2: PREVIOUS WORK	13
2.1 Commercial Partial Reconfiguration Tools	13
2.2 Partial Reconfiguration Tools and Framework.....	16
2.3 Hardware Optimization Technique.....	18
2.4 Software Control Optimization.....	21
2.5 Chapter Summary	25
CHAPTER 3: MULTI-LAYER RUNTIME RECONFIGURATION ARCHITECTURE	26
3.1 Architecture Design Definition.....	26
3.2 Hierarchical Architecture Design	28
3.3 Dynamic Control Flow	34
3.3.1 Adopted Module Based Partial Reconfiguration Flow	36

3.3.2	Frame Based Partial Reconfiguration Flow	39
3.3.3	Physical Area Management	42
3.4	Logic Layer Application	45
3.4.1	Low-Level Crossover Operator	47
3.4.2	Low-Level Mutation Operator	47
3.5	Chapter Summary	48
CHAPTER 4: RECONFIGURATION AND VERIFICATION METHODOLOGY		51
4.1	SelectMAP Based Methodology	52
4.1.1	SelectMAP Interface	52
4.1.2	System Design	53
4.1.3	Testing Strategy and Communication Channel	56
4.2	JTAG Based Methodology	59
4.2.1	JTAG Interface	59
4.2.2	Reconfiguration Control	62
4.2.3	Testing Strategy and System Design	64
4.3	ICAP Based Methodology	70
4.3.1	ICAP Interface	70
4.3.2	System Design	72
4.3.3	Bit Manipulation and Reconfiguration Control	74
4.4	Chapter Summary	77
CHAPTER 5: RESOURCE UTILIZATION AND TIMING ANALYSIS		79
5.1	Basic Application Case Study	79
5.2	Resource Optimization Analysis	83

5.3 Timing Analysis.....	91
5.3.1 Fundamental Timing Parameters	91
5.3.2 Translation Engine Evaluation.....	92
5.3.3 Timing Analysis.....	96
5.4 Chapter Summary	98
CHAPTER 6: RECONFIGUARTION OPTIMIZATION TECHNIQUES	101
6.1 Area and Bitstream Optimization	101
6.2 Application Analysis.....	111
6.2.1 Data Structure Mapping.....	111
6.2.2 Flexible Routing.....	113
6.3 GA Execution MRRA.....	118
6.3.1 Performance Evaluation.....	120
6.3.2 Results Analysis.....	122
6.4 Chapter Summary	124
CHAPTER 7: CONCLUSION	126
7.1 Summary.....	126
7.2 Future Work	130
LIST OF REFERENCES.....	133

LIST OF FIGURES

Figure 1: FPGA architecture.....	2
Figure 2: Design Layout with Two Reconfigurable Modules	5
Figure 3: Multi-layer Runtime Reconfiguration Architecture	28
Figure 4: LUT Representation at Logic Layer.....	29
Figure 5: Translation Process Flow Diagram	31
Figure 6: Logic Control Flow	34
Figure 7: Module-Based Flow	37
Figure 8: A Simple Logic Modification Example	39
Figure 9: Physical Area Management.....	42
Figure 10: SelectMAP based System.....	54
Figure 11: IPIF Template.....	55
Figure 12: JTAG Control State Machine [80]	61
Figure 13: Programming Flow under JTAG Control.....	62
Figure 14: An .SVF Example [20].....	63
Figure 15: ChipScope Logic Analyzer [20].....	64
Figure 16: Loosely Coupled System for JTAG [20].....	66
Figure 17: ChipScope Core Insertion Procedure	67
Figure 18: GNAT Based System	69
Figure 19: System On Chip Example	71
Figure 20: ICAP Based Testing System	73
Figure 21: Bitstream Mapping Process.....	76

Figure 22: Bus Macro Placement [20]	84
Figure 23: Partial Reconfiguration Module Routing	85
Figure 24: MRRA with SelectMAP Interface Placement and Routing	86
Figure 25: Block Diagram of ChipScope Cores and Associated Interconnection.....	87
Figure 26: Placement and Routing of ChipScope Cores and Associated Interconnection [20] ...	88
Figure 27: GNAT Placement and Routing [52]	89
Figure 28: B02 and C17 Mapping and Placement	93
Figure 29: B03 and C1908 Mapping and Placement	94
Figure 30: Column Level Configuration Memory Map	102
Figure 31: Optimized Design Layout for Case 1 and 2	106
Figure 32: Case 3 Before and After Optimization	108
Figure 33: Case 4 Before and After Optimization	109
Figure 34: Flexible Routing Example.....	115
Figure 35: Inter-connectivity Nets for Flexible Routing	116
Figure 36: Unseeded Design GA Runs [52]	122
Figure 37: Seeded Design GA Runs [52]	123
Figure 38: Repair GA Runs [52].....	124

LIST OF TABLES

Table 1: Recent Frameworks for Partial Reconfiguration	17
Table 2: High-Level Partial Reconfiguration Optimization	24
Table 3: Communication APIs on PowerPC	57
Table 4: Communication APIs on Host PC	58
Table 5: Testing APIs for JTAG.....	70
Table 6: Step Function Resource Utilization And Power Evaluation.....	82
Table 7: Resource Utilization	90
Table 8: Basic Timing Evaluation	92
Table 9: Translation Engine Evaluation	96
Table 10: Area and Bitstream Optimization	110
Table 11: Intermediate Translation APIs.....	113
Table 12: GA Parameters [52]	121

CHAPTER 1: INTRODUCTION

1.1 Overview

Field reconfigurable devices have been available for almost two decades [50]. The evolution of these systems has been considerably impacted by the development of Field-Programmable Gate Array (FPGA) technology. The basic architecture of a modern FPGA, as shown in Figure 1, consists of an array of Configurable Logic Block (CLB) that can be programmed to implement different design logics and a routing architecture that interconnects the CLB logic. Current commercial FPGA's CLBs can be based on one or more following technologies [32]: Transistor pairs, Basic simple gates such as two-inputs NANDs, Multiplexers, SRAM based Look-up tables, and Wide fan-in AND-OR structures. These CLBs can be implemented into both combinational and sequential logic functions.

The routing architecture of an FPGA is the physical network that makes connections between each individual CLB so that the basic logic functions can be formed into larger algorithms and applications. The routing architecture could be as simple as a nearest neighbor mesh or as complex as the perfect shuffle used in multiprocessor [32]. Most of the time, the FPGA incorporates various lengths of segments that can be interconnected for different needs, such short line, long line and global clock line. The number and the length of the wire segments directly affect the density and performance achieved by a FPGA. Along the path of the routing lines, there are programmable switches that can change the interconnections between different lines. There are three major types of technologies currently in use for the programmable switch implementation: SRAM, Antifuse, and EEPROM. Each has its own advantages and

disadvantages in the term of area and reprogramming ability issues. SRAM-based switches, working just as normal memory elements, can be readily reprogrammed and refreshed. However, they require more transistors to be implemented thus more area is occupied. On the other hand, the Antifuse based switches use the minimum area among the three. Yet they only allow to the device to be programmed once, and after that the interconnections are fixed. EEPROM-based switches stand in the middle among these three in both the area and reprogramming ability issues. Their required implementation area is larger than Antifuse based, but smaller than the SRAM based. They can be reprogrammed multiple times but may require higher voltage and the speed and flexibility is not as good as SRAM.

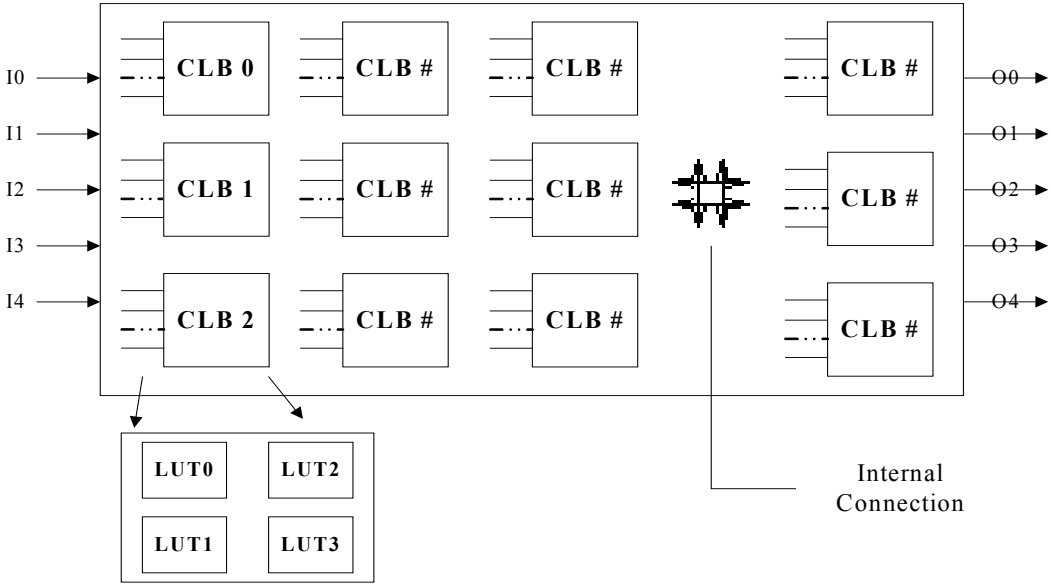


Figure 1: FPGA architecture

Modern FPGAs have evolved from simple Programmable Logic Devices (PLDs) to fully integrated System on Chip (SOC) architectures containing microprocessors, embedded memory, and optimized datapaths connected to a high capacity, dynamically reconfigurable fabric. As a

case in point, the high-end Virtex FPGAs family offered by Xilinx contains a multi-million gate-equivalent reconfigurable fabric in which several PowerPC processors, a number of RAM blocks, and multipliers are provided. Hence there is considerable interest in using these powerful customizable platforms for a wide range of high-performance scientific computation and embedded applications.

In particular, a unique aspect of flexibility provided by FPGAs is the capability for dynamic reconfiguration, which involves altering the programmed design within an SRAM-based FPGA at run-time [42]. With the capability of partial reconfiguration from device manufacturers [72] and availability of powerful on-chip CPU cores and block RAM, more and more research interests have been focused on this area and demonstrate applications benefiting from use of this reconfiguration paradigm, such as mobile systems [14], [15] operating system frameworks [22], [23] and artificial intelligence applications [4]. On the other hand, although FPGA architectures have advanced considerably, dynamic reconfiguration capabilities are only introduced in quite recent year and have not kept pace with architectural improvements. Currently, only limited FPGA hardware, provide and support partial and dynamic reconfigurable ability to some extent. As more and more applications involve runtime reconfiguration, *autonomous dynamic reconfiguration*, which automates the partial reconfiguration and/or testing/verification process by combining the capability of partial reconfiguration from the device manufacturers and the availability of powerful on-chip CPU cores, has become an important research topic.

1.2 Partial Reconfiguration Technology

Currently multiple vendors offer devices with various partial reconfiguration abilities including Altera, Atmel, Lattice, and Xilinx. The partial reconfiguration capability from Altera only includes certain components such as the divider of the Phase Locked Loop [65], instead of the general reconfigurable resources. The AT40K family from Atmel demonstrates some promising general partial reconfiguration performance with literature describing a 50K maximum gate-equivalent capacity [66], which comparatively restricts the framework and possible applications. On the other hand, Xilinx FPGAs, which provide multi-million gate-equivalent capacities, are the most widely used chips with partial reconfiguration capability. Therefore, most of the current research work focused on these hardware platforms. Work here has also chosen Xilinx as the major research platform. To support their FPGA architectures, Xilinx has proposed two standard reference flows for partial reconfiguration process: *Difference-based flow* and *Module-based flow* [72].

With a Difference-based flow, the designer must manually edit a design with only small changes. After the changes are completed, the partial bitstream, which contains information only regarding the differences between the two designs, is generated and stored in a file. Switching the configuration of a module from one implementation to another is very quick, as the bitstream differences can be significantly smaller than the entire device bitstream [72].

There are two main ways a design can be altered to be utilized with Difference-Based Partial Reconfiguration. The design can be changed either at the front-end in HDL or Schematic format or at the back-end in Native Circuit Description (NCD) file format. For front-end changes, the design must be re-synthesized and re-implemented to create a newly placed and

routed NCD file. For back-end changes to the NCD files, sections of a design can be modified using the FPGA Editor tool directly. Bit stream generation tool then can produce custom partial bitstreams that only modify small sections of the device.

Xilinx Modular Design methodology is another flow proposed by Xilinx, which allows a team of engineers to independently work on different pieces, or “modules,” of a design and later merge these modules into one FPGA design [72]. For this flow, the full design is partitioned into modules, some of which can be fixed while others can be reconfigurable. The reconfigurable fabric of the FPGA is partitioned into column-based rectangular regions with the width ranging from a minimum of four slices to a maximum of the full-device width in four-slice increments, in which the fixed and reconfigurable modules will be arranged based on specified area constraints of the design.

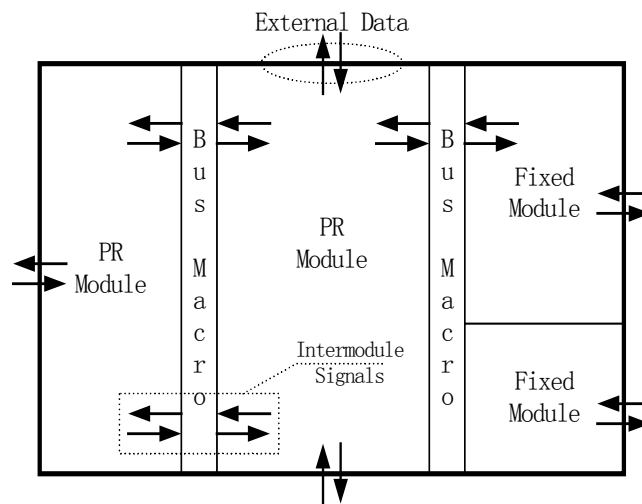


Figure 2: Design Layout with Two Reconfigurable Modules

There should be no common I/O between fixed and reconfigurable modules except the clock. If signals are needed to cross over a partial reconfiguration area boundary, a *bus macro*, a fixed "bus" of inter-design communication, should be used to maintain correct connections

between the modules by spanning the boundaries of these rectangular regions. Each time when partial reconfiguration is performed, the bus macro is used to establish unchanging routing channels between modules, guaranteeing correct connections. Figure 2 shows the basic concept of this reconfiguration flow methodology. Such arrangement makes the Module-based flow much more flexible than the Difference-based flow and suitable for possible full automation, particularly when it is considered for integration with high-level user applications. Therefore, the Module-based flow has been chosen as the primary basic partial reconfiguration technique for the research.

1.3 Need for Autonomous Partial Reconfiguration

As mentioned earlier, since the introduction of the partial reconfiguration technique, more and more high-level applications and algorithms in different areas have been attempting to incorporate this concept into their design to boost their performance and decrease the human-related control requirements. NASA deep space mission is one of the typical examples of such scenarios, which requires high reliability involving mission safety or other critical tasks. Such applications rely increasingly on FPGAs to support their computing requirements. For example, NASA's Stardust probe carries on board over 100 FPGA devices [27]. As the number of FPGAs increases in the computing systems supporting these missions, fault detection and repair becomes critical to these missions. While a probe is traveling in deep space, permanent failures can degrade the functionality of Configurable Logic Blocks (CLBs) or programmable interconnects of the FPGAs. To quickly reconfigure many alternatives in an attempt to recover from these failures is necessary to keep the on-board computing system of the probe in normal operation.

However, under such circumstances, human intervention is very difficult or even impossible to carry out and will definitely slow down the performance of the original task significantly. Hence, an automatic mechanism is crucial to carry out such process. At the same time, such high throughput applications also require the reconfiguration operations to be processed as fast as possible to avoid long time system waiting and severe performance degeneration.

To meet such requirements, the partial reconfiguration management should be automatic, seamless, and completely transparent to the application. The management architecture needs to determine:

- (i) *Partitioning*: Which computational resources to initialize as component,
- (ii) *Placement*: How to determine the target location of the component on the reconfigurable fabric of the device,
- (iii) *Routing*: How to properly interface the component to its surrounding resources,
- (iv) *Generation*: How to generate the new bitstream of the component at the target location,
- (v) *Configuration*: When and how to write the generated bitstream to the appropriate portions of the underlying reconfigurable infrastructure of the reconfigurable fabric, and
- (vi) *Verification*: How to communicate with the hardware platform to test and validate the new downloaded bitstream.

However, current partial reconfiguration flow suggested by Xilinx mainly focused on the 4th step and thus only provides an incomplete solution to it. To use this basic flow, design tools with GUI are involved. Manual adjustments of design are unavoidable. Full automation control

process, as required by most of the current partial reconfiguration applications, is almost impractical.

Furthermore, using this basic solution, the reconfiguration generation delay alone is already on the order of tens to hundreds of milliseconds [54]. In some situations, configuration overhead can comprise over 98.5% of execution time [36]. Clearly, this extremely slow process already becomes one of the major barriers, especially for applications based on redundancy and spare availability when using partial reconfigurations such as [39], [57]. Some research work has been carried out at various levels to suggest solutions for separate steps. But so far there is no framework that has been proposed to accomplish all of the above steps and provides a general-purpose solution.

A more sophisticated partial reconfiguration framework would be useful to integrate and optimize existing reprogrammable technologies, as well as refine theories of operation in light of the feasibility of current and near-term hardware implementations. Ideally, this approach would provide a standardized set of APIs and abstracted data structures for a variety of high-level applications. It would facilitate algorithm mapping via uniform access to heterogeneous logic and communication resources. Such an approach would also improve flexibility and enhance portability across hardware reconfiguration interfaces requirements, and enable more sophisticated applications based on autonomous reconfiguration.

1.4 Dissertation Outline

The remainder of the document is divided into the following Chapters. In Chapter 2, *PREVIOUS WORK*, a review of the State of the Art in partial reconfiguration tools, architectures and high-level control algorithms is given. The section *Partial Reconfiguration Tools and Framework* introduces the current toolsets for partial reconfiguration published by industry companies and some of most recent framework and toolsets for partial reconfiguration developed at academic area. The section *Hardware Optimization Technique* refers to the proposed optimization techniques that addressed the basic partial reconfiguration hardware components or the bitstream generation process. The chapter ends with the section *Software Control Optimization* that contrasts the investigated techniques at the software algorithm level for partial reconfiguration research.

In Chapter 3, *MULTILAYER RUNTIME RECONFIGURATION ARCHITECTURE*, the design considerations for the autonomous reconfiguration architecture are first analyzed. The detailed architecture design is then discussed, including the 3 layers of the *Hierarchical Architecture* and the *Dynamic Control Flow*, which provide the *Adopted Module Based Flow* at design phase, the *Frame Based Flow* at runtime phase and the *Physical Area Management* control. Finally the basic *Genetic Algorithm* Operators for FPGA high-level fault tolerance application based on the MRRA structure and control flow are illustrated.

In Chapter 4, *RECONFIGURATION AND VERIFICATION METHODOLOGY*, the system design considerations for the reconfiguration and verification methodology are discussed. Three different types of reconfiguration interface, including SelectMAP, Joint Test Action Group (JTAG) and Internal Configuration Access Port (ICAP) are introduced. Difference

reconfiguration and verification methodologies and system design based on the characteristics of these three interfaces are explored. Details of bit manipulation and function mapping equations are also studied and discussed in details.

In Chapter 5, *RESOURCE UTILIZATION AND TIMING ANALYSIS*, three groups of experiments are described. First group of tests analyzed the resource utilization and power consumption by using two Hashing algorithms as top-level application case study. The data are compared and contrast through the criteria of slice occupancy, dynamic power consumption, and core power consumption. The utilization data are analyzed in three groupings of the design, including the traditional baseline design, module-based design, and frame-based design. The second group of experiments demonstrates the detailed FPGA resource utilization for each MRRA platform with a different reconfiguration interface and testing methodology. Six different MRRA prototype platforms are tested. In addition to the slice utilizations, the routing flexibility, I/O arrangement, resource placement and communication overhead are also used as metrics to quantify these platforms' performance. The last group of experiments evaluates the timing performance of the MRRA system, including the basic reconfiguration and testing time and the Translation Engine overhead. A brief timing analysis is also presented based on the collected data to determine the bottleneck of the system speed and possible optimization methodology through pipelining.

In Chapter 6, *HIGH-LEVEL RECONFIGURATION OPTIMIZATION TECHNIQUES*, the proposed bitstream optimization strategy is tested and its behavior is analyzed by applying it to five different types of experimental circuits. Comparisons against the tool automation approach are given. The design technique of linking the high-level data structure with the physical circuit design is then demonstrated. The flexible routing strategy by using LUT-based switch box is also

presented. The resource expense of this strategy is analyzed as well. The final part of this chapter shows the experimental details of GA applications running on MRRA using these advanced techniques.

In Chapter 7, *CONCLUSION*, the results of the experiments are summarized and general conclusions about MRRA architecture design, the reconfiguration and verification methodology employed, and high-level application design using the MRRA platform are drawn. Future work is identified for the continuation of the project. Several areas are proposed for further research, related to new hardware architecture, better area management algorithms, and the application of these techniques at general data structure level, especially for GAs.

1.5 Contribution of Dissertation

A summary of the major contributions made by this dissertation includes:

- 1) An original thorough solution providing a clear framework with standard interfaces between partial reconfiguration physical implementations and logic designs, which allows the general high-level algorithms to execute at an abstract level without the knowledge of the low-level hardware details. Other COTS tools may be easily integrated into this framework and customized for their own applications. This enables a real hardware-related Genetic Algorithm to execute directly on a FPGA in real-time.
- 2) Layering model that shields the reconfiguration logic and higher-level application logic from the hardware details. So when later a different FPGA is selected for use, the bottom reconfiguration layer can change accordingly. The algorithm layer

can remain untouched. The only modification needed is carried out on the hardware-dependent part of the APIs at the translation layer.

- 3) This is the first system design that explores multiple mainstream reconfiguration interfaces and provides correspondent reconfiguration and verification methodologies. Their performance is also contrasted and compared between each other. Advantages and disadvantages are analyzed in details under a variety of scenarios.
- 4) The traditional spatial and temporal algorithms for fast configuration are also advanced. A practical strategy is formed based on current hardware technology. The strategy now is implemented distributively in multiple layers as independent hardware or software modules. Therefore these algorithms can be used respectively or combined depending on user's high-level application. Special modifications can also be easily carried out on these traditional algorithms for specific user requirements without affecting the normal major working flow.

The benefits of this MRRA approach include increased design productivity, portability and resources utilization. On the other hand, these advantages require new tradeoffs for extra hardware complexity, software capability, and resource overhead, as quantified in the analysis herein.

CHAPTER 2: PREVIOUS WORK

Many different industry efforts and research works aimed at different aspects of partial reconfiguration have been studied in the literature, including partial reconfiguration generation tools, layered partial reconfiguration frameworks, hardware size and performance optimizations and software control optimizations, etc. Some of the most recent and relevant works from each category are reviewed in this Chapter.

2.1 Commercial Partial Reconfiguration Tools

Currently, the most widely used FPGA chips with partial reconfiguration capability are from Xilinx in the Virtex, Virtex II, and Virtex Pro families [69]. Yet only very few preliminary toolsets commercially have been reported supporting the partial reconfiguration paradigm. JBits [53], an earlier research toolset presented by Xilinx, provides dynamic reconfiguration capabilities, allowing an application to instantiate a component, generate its corresponding bitstream, and download it to a reconfigurable device. The JBits Application Programming Interface (API) is implemented in the Java programming language and permits programmatic access to all of the configurable elements in Xilinx Virtex-II FPGAs. It supports partial configuration and modification of the bitstream. It is a low level tool that not only changes the logic content of LUTs, but also dynamically modifies specific logic interconnections. JBits communicates with the hardware through a generic hardware interface called Xilinx Hardware Interface or XHWIF. It can be very useful for designers who want to design run-time reconfigurable logic on FPGAs, which will help in the future revolution of online upgradeable hardware. Since the tool is Java based, the execution speed can be comparatively slow. This

toolset has been made only available for the Virtex family primarily as a research tool for Virtex II series, instead of the bigger Virtex family such as Virtex II Pro and Virtex 4. Xilinx had dropped all the technical support and further development several years ago, even though the toolsets still have significant academic interest.

The Xilinx Partial Reconfiguration Toolkit (XPART) [5] is another tool that has been proposed by Xilinx engineers in 2003. XPART is derived from the JBits API efforts. It provides a lightweight, minimal set of JBits API features implemented in the C language instead of Java. Similar to the JBits API, it abstracts the bitstream details providing seemingly random access to select FPGA resources. The XPART API enables fine grain reconfiguration control over select FPGA resources. This allows actions such as tuning off Multi-Gigabit Transceiver (MGT), or constant folding achieved by modifying LUTs. XPART also provides some basic functionality for supporting relocatable modules. A locatable module is a partial bitstream that can be relocated to multiple places on the FPGA. XPART provides two methods for dealing with relocatable modules. The two methods are `setCLBModule` and `copyCLBModule`. The `setCLBModule` method works on regular partial bitstreams that contains information about all of the rows in the included frames. The `copyCLBModule` function copies any sized rectangular region of configuration memory and writes it to another location. The copied region contains just a subset of the rows in a frame. This allows the designer to define dynamic regions that have static regions above or below it. The `copyModule` function employs a read/modify/write strategy like the re-source modification functions. This technique enables changing select bits in a frame and leaving the others bits to their current configured state. This self-reconfiguring platform enables embedded applications to take advantage of dynamic partial reconfiguration without requiring external circuitry. However this toolset has never been officially released.

The most recent released platform for partial reconfiguration paradigm by Xilinx is called PlanAhead [68]. This software is based on the Module Based flow paradigm introduced in Chapter 1. PlanAhead streamlines the design step between synthesis and place and route and attempts to reduce in both the number and the length of design iterations. This methodology allows designers to divide a larger design up into smaller, more manageable blocks and focus efforts toward optimization of each module, improving performance and quality of the entire design. PlanAhead software is the first graphical environment for partial reconfiguration. Using PlanAhead design tools as a platform for partial reconfiguration applications can simplify the complexities of the constraints of the dynamic operating environment of applications, allowing a single device to operate in applications that previously required multiple FPGAs. It provides a single environment to manage the Module Based flow guidelines. Using PlanAhead design tools to implement a partial reconfiguration design, users can carry out the netlist import, floor planning for partial reconfiguration, design rule checks, netlist export, and implementation flow management. The methodology offered by PlanAhead software can potentially increase productivity and decrease time-to-solution for designers using partial reconfiguration. However this tool is a graphical environment for design time use only instead of runtime control. Moreover, it supports the Module Based flow considerations. Currently, the partial reconfiguration function part of this tool is also not open for public use yet and requires a special request to Xilinx for access.

2.2 Partial Reconfiguration Tools and Framework

Since the commercial works have placed emphasis on components and methods for the step of partial reconfiguration file generation, significant challenges have remain with creating an autonomous environment for dynamic reconfiguration. Significant recent work in academia and industry has been focused on this area.

Some representative research approaches are listed in Table 1. Early work by Moraes et al had developed a set of tools for remote and partial reconfiguration for Virtex XCV300 [11]. With this toolset remote reconfiguration is enabled to update and/or fix hardware cores in the field. By using this toolset, parameter reconfiguration can be used to customize a circuit, avoiding extra devices as external microcontrollers and ROMs, and saving internal control logic in the FPGA. However, some steps of the approach have to be carried out with manual user intervention and the described technique does not intrinsically support core relocation.

Raghavan and Sutton's tool called JPG was developed for Xilinx Virtex devices [1]. This JPG tool is based on the Xilinx Java-based JBits API. Jbits allows an application to instantiate a component, generate its corresponding bitstream, and download it to a reconfigurable device such as the Virtex FPGA. Therefore JPG is able to generate partial bitstreams for Xilinx Virtex devices based on data extracted from the standard Xilinx CAD tool flow. Yet due to the use of JBits and its Java interpretation overheads, the tool has some speed and scalability limitations. Since Xilinx had already dropped the support for JBits, further upgrading and expansion of this framework may be uncertain.

Table 1: Recent Frameworks for Partial Reconfiguration

APPROACH	DEVICE SUPPORTED	ON-CHIP SYSTEM	BITSTREAM REUSE	POTENTIAL CHALLENGES
Mesquita et al.	Virtex XCV300	N	N	Area Relocation
Raghavan, Sutton	Virtex	N	N	Supporting CAD flow
Blodget, McMillan	Virtex II	Partial	Y	Direct bitstream reuse
Williams et al.	Virtex II	Y	Y	Large User application
Kalte et al.	Virtex E	N	Y	Dynamic Routing
Bobda et al.	Virtex	N	N	Communication and Control Overhead

To avoid some of these limitations and more fully encapsulate the higher layers from low-level device specifics, a two-layer framework for Virtex II devices had been separately suggested by Blodget et al [6] and also Fong et al [51]. These systems enable self-reconfiguration under software control within a single FPGA. The system enables self-reconfiguration through the reconfiguration hardware interface *Internal Configuration Access Port (ICAP)* inside the Xilinx FPGA. The reconfiguration subsystem has a two-layer hardware and software architecture that permits a variety of different interfaces. The system enables self-reconfiguration under software control within a single FPGA, minimizes the need for external hardware and provides a highly integrated, lightweight approach to dynamic reconfiguration for embedded systems. However, because of the operations of ICAP, the bitstream has to be processed directly instead of high-level netlists.

Experimental Generic Reconfigurable Embedded Target (EGRET) [33] [45] is another similar framework proposed by Williams et al. This framework is also focusing on full system-on-chip (SOC) solution by using ICAP and an embedded Linux system on a Xilinx Virtex II

chip. More emphasis has been drawn to the software control side using embedded Linux. But on the other hand, currently available CPU core speed and RAM size can impact the complexity of the high-level applications that can be implemented into such solution.

Bobda et al also presented a framework named Erlangen Slot Machine (ESM) [9]. In this platform, each module can access its periphery independent from its location through a programmable crossbar, which gives the potential of unrestricted one-dimensional relocation of modules on the device. Different inter-module communication channels, including a bus macro, shared memory, reconfigurable multiple bus and crossbar have also been proposed. As a variety of communication channels are available, multiple external control hardware and boards can be involved. Communication and control overheads using such approaches may vary.

The most recent framework developed by Kalte et al [16] is called *REPLICA (Relocation per online Configuration Alteration)*, which mainly focused on the step of downloading and relocating the modules. The REPLICA parses the bitstream during the normal download process and replaces the column addresses within the bitstream according to the desired location of the module. Next, it uses the SelectMAP interface to perform bitstream manipulation to carry out the relocation process on Xilinx Virtex Series to achieve the maximum possible throughput. Other similar proposed frameworks and tools for partial reconfiguration process also include [12], [26], [33], [51], and others.

2.3 Hardware Optimization Technique

One of the other common research interest areas is at the downloading step, by using different hardware optimization techniques to achieve possible timing and performance

improvement. In previous early stages of research, bit file compression [25] [56] is one of the direct approaches used to accelerate the reconfiguration process. In such an approach, customized decompression hardware is required. Different general-purpose compression algorithms for reconfiguration may be carried out. It had been reported that using this approach might bring an overall reduction of up to a factor of four in total bandwidth required for reconfiguration under Hank's approach [56].

With the appearance of the partial reconfiguration technology in recent years, it has observed that the configuration overhead can be improved by just over a factor of 7 over the serially programmed FPGA [36]. This led to an extra series methods addressing from different hardware aspects to improve its efficiency and thus increase the reconfiguration speed even more. Compton, Li, Knol and Hauck [36] developed an algorithm for configuration relocation and defragmentation. A new custom designed architecture FPGA as well as software algorithms for controlling this hardware is presented, which has different features from the normal commercial FPGA chips. With an extra hardware area specially designed for controlling the relocation and defragmentation, which is comparatively negligible, it is reported that as much as 35% improvement in reconfiguration times in these devices for realistic run-time algorithms may be gained, compared with basic partial reconfiguration architecture and multi-context device [36]. However this architecture modification is specifically based on the Xilinx 6200 series, which is not available any longer in the market.

For more recent practical solutions, Raghuraman, Wang, Tragoudas [37] on the other hand studied the configuration data size at the logic level. In this approach, the LUTs' inputs are fixed to appropriate order based on a heuristic algorithm in polynomial time. With such LUT input orders, memory locations that need to be changed during partial reconfiguration process are

relocated into common frames as many as possible. By relating the number of frames that need to be downloaded into FPGAs to the number of minterms of a specially constructed logic function, the required number of configuration frames can be reduced. Since the number of the configuration frames is directly related to the size of the partial reconfiguration bitstream, the final hardware bit file size should be able to be reduced. It is reported that the size of the reconfiguration data could be reduced by around 15% by this approach.

Regarding to other aspects of the partial reconfiguration hardware bitstream generation, Upegui and Sanchez [3] recently discussed possible methodologies to generate the partial reconfiguration bitstreams. Besides the standard module-based and difference-based approaches suggested by Xilinx, the technique for low-level direct bitstream modification is suggested. By calculating the physical location of specific LUT, the logic of this LUT can be directly located in the bitstream. Thus, the logic content of LUTs can be modified directly inside the old bitstream with much faster speed instead of generating a new one based on the cumbersome standard CAD flow. However, this paper only discussed the full bitstream format without exploring the partial reconfiguration bitstream pattern, which is quite different from the format of a complete device file. This limits the usage of such approach. Moreover, this direct bitstream modification can only apply to the logic contents, instead of the routing part of the bitstream, which is far too complicate to be manipulated at the bitstream level.

Sedcole et al [48] also presented a new partial reconfiguration flow to generate the hardware bitstream, called the *merge partial reconfiguration method*. The merge method prepares modules to be allocated arbitrary areas in FPGA with a customized tool required to be involved in the place and route step of the process. In this method, by using an XOR function to combine the new partial bitstream and the current configuration read back from the device,

existing configuration information is preserved, and the module can be removed by repeating the XOR operation. Only the difference then needs to be downloaded to the device. Modules can be allocated in any rectangle region in device and static routes can pass through reconfiguration area. To avoid conflicts, some of the routing resources are reserved for the static routes. It is reported that speed measurements have revealed an increase in configuration times of between 2.4-fold to 4.0-fold, with a baseline overhead of at least 1.58-fold.

Hardware routing-related issues with partial reconfiguration are also addressed. Two types of special designed communication bus for partial reconfiguration modules research independently by Krasteva et al [62] and group of Bobda et al [8] to take the place the bus macro suggested by Xilinx. By using LUT-based or slot-based communication bus macro structure, physical partial reconfiguration resource region size may be modified based on the logic modules size. Large regions may be split. Adjacent small ones may be combined together. Thus, it provides possibility of module relocation and area re-partition. This can bring potential for optimization of the hardware performance.

2.4 Software Control Optimization

Besides the low-level hardware related research, a lot of approaches had also been proposed from theoretical control algorithm angles, including both design phase and runtime phase of the partial reconfiguration flow. In Shirazi, Luk and Cheung's approach, two successive circuit configurations are matched to locate the components common to them, so that reconfiguration time can be minimized [46]. Two stages had to be carried out for this approach. In the first stage, possible components for reconfiguration are identified, and a sequence of

conditions for activating an appropriate component at a particular time is found. This step has to be carried out manually. In the second stage, successive configurations will be optimized to achieve the desired trade-offs in reconfiguration time, operation speed and design size. Components and connections common to two or more successive configurations will be identified automatically and will not be reconfigured.

Another approach suggested by Diessel, Middendorf and Schmeck is dynamic scheduling when there are multiple independent tasks to be reconfigured [47]. Rearranging a subset of executing tasks takes two steps. The first step identifies a rearrangement of the tasks executing on the FPGA that frees sufficient space for the waiting task, and the second schedules the movements of chosen tasks so as to minimize the delay to their execution. Three methods for have been tested, including *local repacking* (deterministic heuristics), *ordered compaction* and *GA*. However the result varies depending on the reconfiguration delay $T_{DRT}(i)$ and the processing time of tasks $T_{TAT}(i)$. The GA is advantageous when $T_{DRT}(i) \ll T_{TAT}(i)$. While for the large reconfiguration delays $T_{DRT}(i)$ and smaller task processing times $T_{TAT}(i)$, faster methods carrying out less rearrangement perform better. When $T_{DRT}(i) \gg T_{TAT}(i)$, the simple first-fit method without rearrangement tends to perform well.

For large reconfiguration tasks, Mak and Young introduced a temporal logic replication method named hierarchical bipartitioning [60]. In this method, an application has to be partitioned into multiple stages. The configuration will be switched continuously to implement each stage one by one in order to perform the function of the original circuit. They applied this to effectively exploit the slack capacity of a stage to reduce the communication cost. For the case when there is a tight area bound that limits the amount of replication, they also presented a flow-

based replication heuristic. In addition, they showed a correct network flow model for partitioning sequential circuits temporarily and proposed a new hierarchical flow-based performance-driven partitioner for computing initial partitions without replication [60].

Pipelining is another temporal partitioning approach discussed by Ganesan and Vemuri [55]. By using the processors partial reconfiguration capability, overlapping execution of one temporal partition with the reconfiguration of another, reconfiguration overhead L_{tot} was reduced from $\sum_{i=1}^n (T_{DRT}(i) + T_E(i))$ to $T_{DRT}(1) + \sum_{i=1}^{n-1} \max(T_{DRT}(i+1) + T_E(i)) + T_E(n)$, where $T_{DRT}(i)$ and $T_E(i)$ are the reconfiguration time and execution time respectively. They also incorporated block processing in the partitioning framework for reducing overhead of partitioned with the ability to handle loops and conditional constructs in the input specification.

Herbert, Christoph and Macro presented placement methods that rely on efficient algorithms for the partitioning of the reconfigurable resource in the temporal level and a hash matrix data structure to maintain the free space in the spatial level at the same time [22]. Given n as currently placed tasks, previously known placers find a feasible location in $O(n)$ time. Additionally, simulations show that the methods improve the placement quality by up to 70% compared to Bazargan's Partitioner [35].

However, these proposed methods are only focused on the logic algorithm level with simplified FPGA architecture and partial reconfiguration operation assumptions, which current hardware may not be able to provide. Most of them remain untested on real hardware platforms or even sophisticated commercial simulation tools. Therefore each may have its own implementation limits. Some may appear to be even impractical to implement based on current partial reconfiguration technology. The brief comparison of these methods is listed in Table 2.

Table 2: High-Level Partial Reconfiguration Optimization

APPROACH	METHOD	PARTIAL RECONFIGURATION	SPATIAL REALLOCATION	TEMPORAL PARALLELISM	AREA SHAPE	RUN-TIME	POTENTIAL LIMITATIONS
Hauck, Li, Schwabe	Bit file compression	No	N/A	N/A	N/A	No	Full reconfiguration required
Shirazi, Luk, Cheung	Identifying common components	Yes	N/A	N/A	N/A	No	Design time work required
Compton, Li, Knol, Hauck	Relocation and Defragmentation with new FPGA architecture	Yes	Yes	No	Row-based	Yes	Special FPGA architecture required
Diessel, Middendorf, Schmeck, Schmidt	Task Remapped and Relocated	Yes	Yes	No	Rectangle	Yes	Overhead for remapping calculations
Mak, Young	Dynamic Partitioning	Yes	No	Yes	N/A	Yes	Only desirable to large design
Ganesan, Vemuri	Pipeline	Yes	No	Yes	N/A	Yes	Limited pipeline depth
Herbert, Christoph, Macro	Partitioning and 2D Hashing	Yes	Yes	Yes	Rectangle	Yes	Rigid task modeling assumptions

Now with the appearance of partial reconfiguration technique from Xilinx and more powerful FPGA chip equipped with on-chip CPU core, a more sophisticated and thorough approach may be applied to the reconfiguration process. In this dissertation, a 3-layer architecture, named Multilayer Runtime Reconfiguration Architecture (MRRA) is proposed to establish a general-purpose framework for a wide variety of practical applications and algorithms that require reconfiguration during operation. This framework seeks to integrate and optimize existing technologies and theories as well as fill in all the missing pieces to thoroughly fulfill the tasks listed in Section 3 of Chapter 1. All the concepts and architectures proposed in this

dissertation have been validated with on a real hardware platform along with detailed performance analysis.

2.5 Chapter Summary

This chapter has explored the recent previous works related to partial reconfiguration from multiple areas. In Chapter 2.1, the available commercial partial reconfiguration tools are introduced. Chapter 2.2 presents and compares the previous work at the partial reconfiguration framework area with their advantages and disadvantages. In Chapter 2.3, the past research work at hardware optimization field is discussed. Different techniques are explained. Chapter 2.4 introduced the previous research at the high-level software control area of the partial reconfiguration process. A quick summary of different algorithms is also listed in the Table provide.

CHAPTER 3: MULTI-LAYER RUNTIME RECONFIGURATION ARCHITECTURE

3.1 Architecture Design Definition

Currently, the most widely used FPGA chips with partial reconfiguration capability are from Xilinx in the Virtex, Virtex II, and Virtex Pro families. Yet, there are no sophisticated toolsets commercially available supporting many aspects of the partial reconfiguration paradigm. In order to accommodate a variety of reconfiguration processes required by different applications, a tiered framework, called the *Multilayer Runtime Reconfiguration Architecture (MRR)* based on the available Xilinx FPGA hardware is designed. This architecture includes two aspects. The first aspect is a Tiered Framework. There are four major design considerations that should be able to conceptually address in this framework:

Autonomous Operation: Provide stand-alone reconfiguration capability on the FPGA device as well as a bi-directional communication channel with the embedded host PC to carry out the partial reconfiguration process and routing without manual intervention.

Task-level Modularity: Provide support at levels down to, and including, task-level granularity. A task is defined as an arbitrary function synthesized to a module that can be dynamically downloaded into the reconfigurable device.

Runtime Scenario Support: Provide the ability to generate and reconfigure task bitstreams at runtime as well as design-time. Runtime scenarios envisioned at design-time may not necessarily know in advance which tasks will arrive nor when they will arrive, and in selected cases, what some of their specific properties will be.

In addition to the framework, the second element of the MRRA paradigm is a Logic Control Flow aimed at increasing capability towards the following attributes:

Flow Coverage: Both the design phase and the runtime phase are automated, so that the partitioning, placement, routing, bitstream generation, and configuration steps can be accommodated.

Encapsulation: Control logic of each layer is self-contained thus exposing only a fixed interface to other layers, so that modification made at one layer has minimal influence on other layers. If new control algorithms are added or the device platform is changed, the system can be ported more readily.

Standardization: A standardized set of APIs is provided for uniform access to heterogeneous logic and communication resources.

Effective provision of these capabilities in MRRA design can accelerate reconfiguration speed, reduce resource inefficiencies, and realize sophisticated range of applications. The benefits of this MRRA approach include increased design productivity, portability and resources utilization. On the other hand, the coming along extra hardware and software resources overhead may need to estimate and compensate.

In this dissertation, some of the existing reprogrammable technologies and theories of operation are investigated and enhanced. Missing components of the steps are explored. A full general-purpose framework MRRA is proposed. A high-level data structure along with a standardized logic control flow is developed in the MRRA framework to enable flexible implementation of user applications and maximize the overall performance. Standardized set of Application Programming Interfaces (APIs) and corresponding hardware platform are demonstrated for uniform access to heterogeneous logic and communication resources. Speed

and area optimization method for reconfiguration is also presented. On the other hand, estimation and compensation techniques are also explored to deal with the additional hardware and software resource demands required to provide such advantages.

3.2 Hierarchical Architecture Design

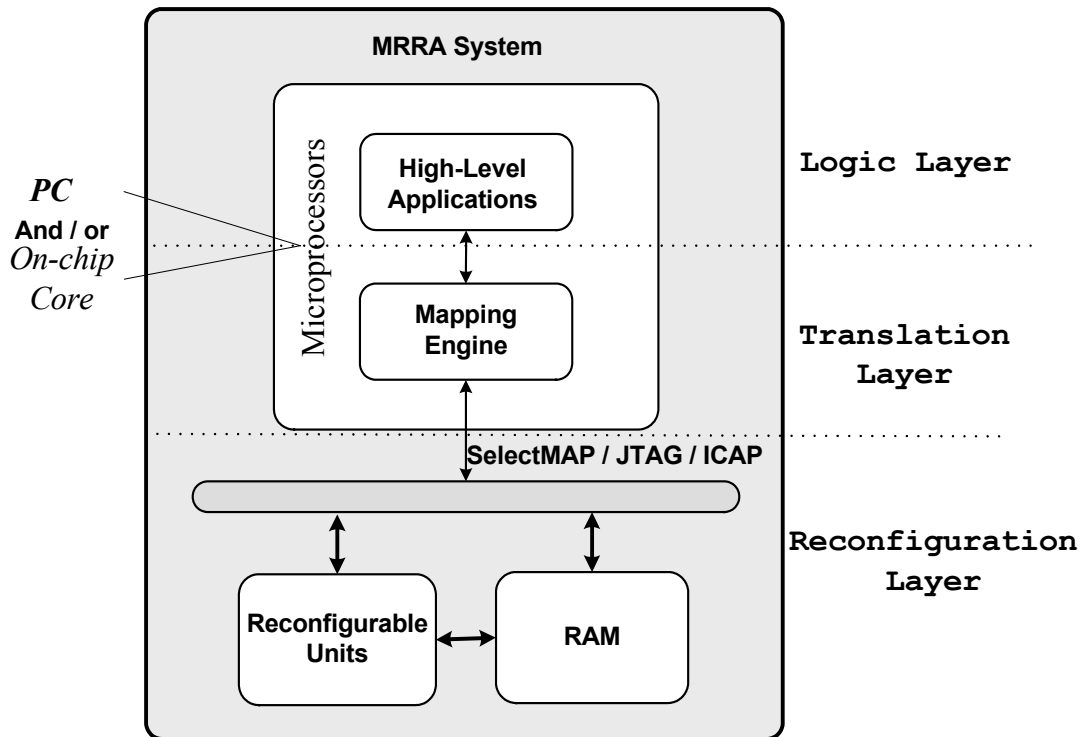


Figure 3: Multi-layer Runtime Reconfiguration Architecture

Figure 3 shows the layered architecture used to encapsulate partial reconfiguration capabilities. The top tier is called the *Logic Layer*. This Layer is the upper tier that supports general user-level applications, carrying out hardware-independent logic control on the tasks running on the FPGA platform. In this layer, task routines are available for invocation by user

applications. Reconfiguration requests can be initiated from this level, based on the requirements of the hardware-independent user logic. These reconfiguration requests, including possible new logic function modification and/or physical resources re-arrangements, are all described in a general logic format at this layer. These are subsequently provided to the translation layer to generate the device-dependent reconfiguration data file.

```

Typedef struct tagLUTinfo
{
    /* LUT status information */
    unsigned short  source[3];          /* The 4 input of the LUT */
    unsigned char   iTruthTable[2];    /* Current output truth table */
    unsigned short  cRow;              /* Current row position */
    unsigned short  cColumn;          /* Current column position */
    unsigned short  destination[255]; /* The output of the LUT */
    char           GorFLUT;           /* 0=G_LUT; 1=F_LUT */

    /* Modification request */
    unsigned short  cFutureRow;        /* Future Row */
    unsigned short  cFutureColumn;    /* Future Column */
    char           SwitchLUTFlag;     /* 0= no change, 1= move position
                                     between G and F LUT */
    unsigned char   iFutureTable[2];  /* Future Truth Table */
    char           PositionFlag;      /* 0=no change; 1=update */
    char           TableFlag;         /* 0=no change; 1=update */

} LUTInfo;

```

Figure 4: LUT Representation at Logic Layer

Figure 4 shows the detailed representation of this logic format. The representation describes the hardware circuit at the Look Up Table (LUT) level. For each LUT, the representation has two parts, the LUT Status Information and the Modification Request. In the LUT Status Information, the LUT inputs and output are labeled. The physical row and column position of the LUT in the FPGA and the logic function inside the LUT are also recorded. The

modification request can be a physical relocation request or a logic function adjustment or both. Besides the details of request information, two modification request flags are also used in this section to advise the translation layer to interpret the request more efficiently. All of the high level applications will only use and modify this device-independent data structure to determine their current state and generate new reconfigurations requests. The reconfiguration requests containing all the LUT information generated at the Logic Layer will generate the device-dependent reconfiguration data file at the Translation Layer. Depending on the complexity of these high-level applications, these can run either in standalone mode on the on-chip CPU core inside the FPGA, or on an external host PC with the on-chip CPU core running simultaneously using a loosely-coupled structure.

The middle tier is referred to as the Translation Layer. In this layer, the general logic descriptions for a palette of tasks are translated into specific physical details as a reconfiguration data file by a hardware-dependent mapping engine. After the partial reconfiguration tasks generation request is made by the user logic from the Logic Layer at runtime, the general information contained in these requests must be translated into a hardware-dependent configuration data file. The original list of partial reconfiguration tasks may include the origin design netlist, physical area allocation, re-allocation and/or direct logic modification. This translation enables the Reconfiguration Layer to execute the reconfiguration requests on the FPGA device. The Translation Layer contains a mapping engine to interpret all of the general representations passed from the upper layer into an actual reconfiguration data file.

Figure 5 shows the details of the translation process. The Translation Layer always stays in the idle state until a new request is sent from the Logic Layer. A new request is always accompanied by an LUT list. Based on the modification request specified in the contents of each

element of the LUT data structure, the status of each LUT is updated. The modification request is then cleared and the corresponding translation engine indicator will be set if necessary. Based on the two translation engine indicators, the corresponding area and logic translation engine will be called to map the general information into device related data. The actions in the dashed boxes in Figure 5 will be processed only when the corresponding flags or indicators are set.

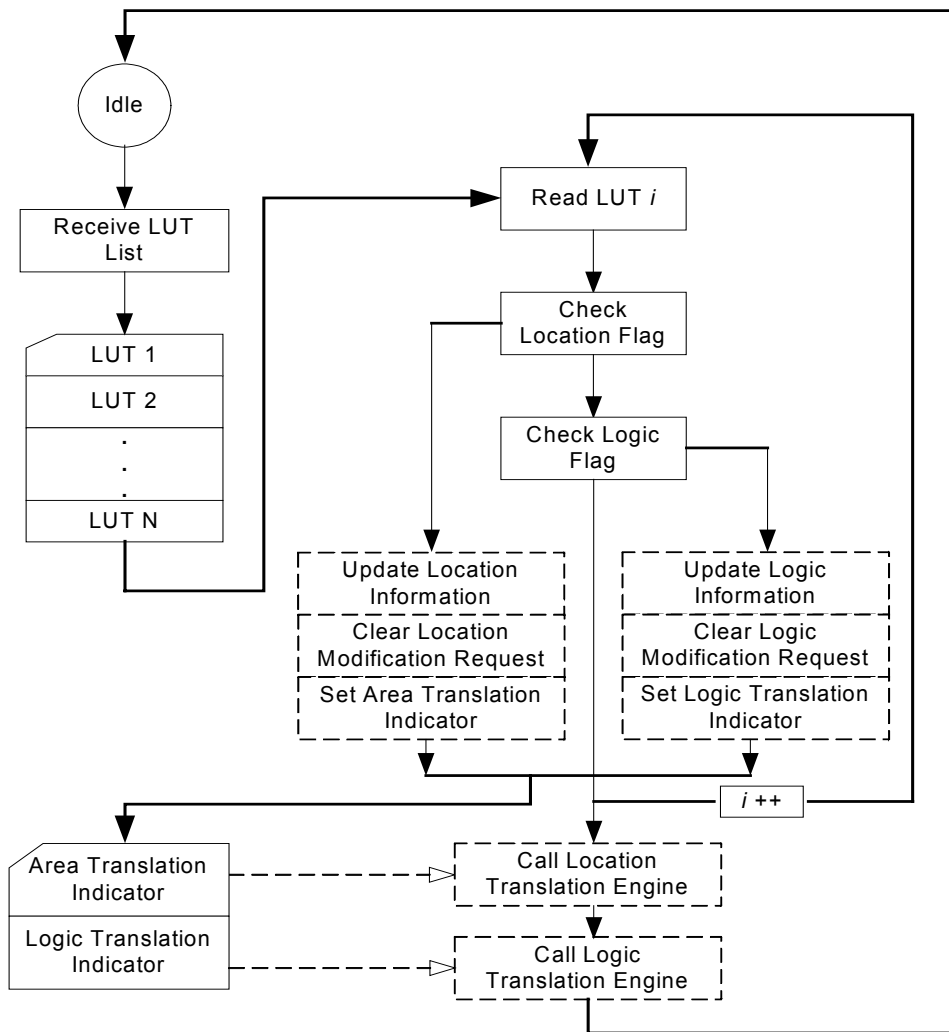


Figure 5: Translation Process Flow Diagram

Currently, in the prototype Translation Layer, both the one-dimensional (1D) and two-dimensional (2D) area management mapping processes still rely on the Xilinx toolset. The

physical resource area management constraints are generated and modified directly by the upper layer logic, and then translated into standard text based constraint inputs by the translation engine in this layer. After the new constraint file is generated, the Xilinx tools are invoked by the translation engine via a shell script. This will automatically run the task in the background to perform the placement and routing for the module without manual input. The Figure 5 shows the details of the translation process. Currently, in the Translation Layer, both the 1D and 2D area management mapping process still rely on the Xilinx toolsets. These physical resource area management constraints are created and modified directly by upper layer logic then translated into standard text based constraint input by the translation engine in this layer. After the new constraint file is generated, the Xilinx toolsets will be called as the other part of translation engine by a shell script. This will run the task at background automatically to perform the placement and routing for the module without manual input.

On the other hand, logic modifications can be translated on either an available partial reconfiguration file or on the currently active configuration data in the device directly without involvement of the Xilinx tools. When the partial reconfiguration file is processed, the Translation Layer will map the top-level logic request directly into the file and then send it to the Reconfiguration Layer interface to be downloaded to the device.

This decouples the bottom layer's hardware-specific considerations from the application's user logic. It also incorporates the online run-time spatial management information into the corresponding partial reconfiguration data file so that when multiple modules need to be reconfigured, the physical area can be reorganized and optimized. With the existence of such a layer, adjustments for changes to the hardware devices or components can be accomplished by

modifications of the mapping engine in the Translation Layer without influencing the top-level Logic Layer.

The bottom tier comprises the *Reconfiguration Layer*. This layer of an autonomous architecture includes the hardware platform and the low-level communication APIs. The hardware platform includes *system resources* and *operational resources*. The system resources include all the external peripherals such as the SRAM, which is a shared memory and can be accessed by both Host PC and FPGA, and the communication interface, such as RS232. The system resources can also contain an on-chip CPU core as the embedded control element and the on-chip Block RAMs, based on design specifications. The operational resources are the actual reconfigurable LUTs inside the FPGA. It can be further conceptually partitioned into two subset region, a *fixed resource subset* that held constant during the entire process and are used to control the on-chip data communications and on-board peripherals, and a *reconfigurable resource subset* that is used for the user-defined partial reconfiguration applications.

The configuration bitstream is downloaded to the targeted FPGA reconfigurable units from the hardware interfaces at this layer when either the initial configuration or the run-time partial reconfigurations are carried out. Input and output data of the FPGA is also passed between the logic control and the bottom FPGA reconfigurable units areas through this path for the functional throughput of the task routines during operation. Block or External RAMs may be used to buffer configuration data in this layer to accelerate the transfer process through pipelining and buffering. The details of the system design and verifications will be discussed in Chapter 4.

3.3 Dynamic Control Flow

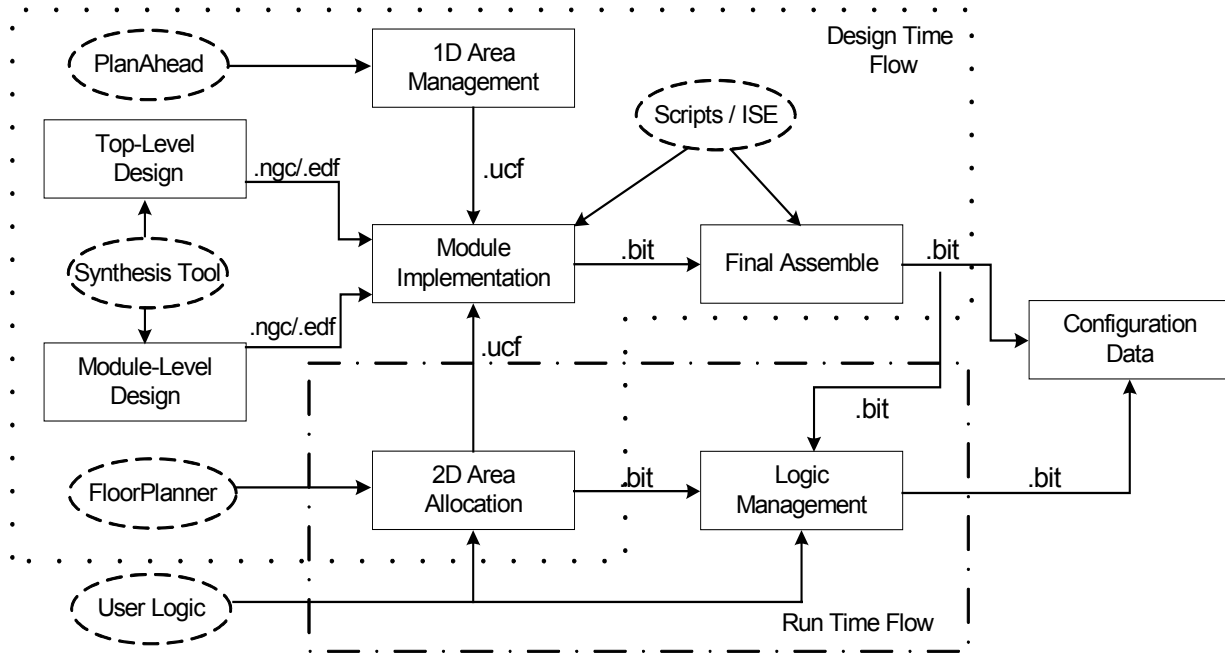


Figure 6: Logic Control Flow

A logic control flow has also been designed for the MRRA as shown in Figure 6. This control flow has integrated a *Module-based Flow* adopted from the standard Xilinx [72] flow with area management ability and the direct bit management process, which we named as a *Frame-based Flow*. This flow allows different pieces, or “modules,” of a design to be independently developed and later merge these modules into one FPGA design and reconfigure and modify them individually later at run-time and it provides the potential full autonomous flexibility by using the translation engine from the lower layer without necessary manual input through GUI interface.

As shown in the Figure 6, a full design needs to carry out first at design time. The full hardware system is partitioned into modules and designed from the top view to the bottom.

Meanwhile, *One-Dimensional Area Management* is performed on the full physical FPGA device by partitioning it into multiple 1D column-based rectangles, in which the fixed and reconfigurable modules will be arranged based on the size of each module and the specified area constraints. Then all these top views and modules are implemented and verified individually. The size of each module will be optimized by additional *Two-Dimensional Area Allocation* placements inside each module. The Optimized partial reconfiguration bitstream for the specific modules are also generated. Finally, all the individual modules are created by *Final Assembly* based on the top-level view and are ready to be downloaded to the FPGA device as Configuration Data bitstreams.

After the initial bitstream is downloaded, based on the user logic control, the precompiled partial bitstream can be monitored by the algorithms in the Logic Layer and updated directly to the device for dynamic reconfiguration when necessary. On the other hand, new modification requests can be generated by the user logic in the form of hardware independent general representation at run-time. Although the boundary of each module is fixed, the physical logic resources inside each module can be re-allocated at runtime. Logic function modification request for each Look UP Table (LUT) inside modules can be generated based on the user requirement as well. Both requests from the Logic Layer will eventually wait for the translation engine from lower layer to interpret to the corresponding configuration data file and reconfigured by the Reconfiguration Layer.

3.3.1 Adopted Module Based Partial Reconfiguration Flow

As delineated by the dashed area in Figure 6, Module-based Partial Reconfiguration Flow is a basic flow primarily used at design time. This flow is adapted from the flow proposed by Xilinx. This flow allows different elements referred to as modules of a design to be independently developed and later merged into one FPGA design. This allows the individual reconfiguration and modification of the modules at run-time. Additionally, it provides the potential for full autonomy and flexibility using the translation engine from the lower layer without the need for GUI-based manual input. For this flow, the reconfigurable fabric of the FPGA is partitioned into column-based rectangular regions in which the fixed and reconfigurable modules will be arranged based on specified area constraints. The reconfigurable modules have the following properties:

- 1) The reconfigurable module height is always the full height of the device. The width ranges from a minimum of four slices to a maximum of the full-device width, in four-slice increments.
- 2) To help minimize problems related to design complexity, the number of reconfigurable modules should be minimized, which means the number of slice columns divided by four is the only real limit to the number of defined reconfigurable module regions.
- 3) A reconfigurable module's boundary cannot be changed. The position and region occupied by any single reconfigurable module is always fixed.
- 4) Reconfigurable modules communicate with other modules, both fixed and reconfigurable, by using a special bus macro.

- 5) The implementation must be designed so that the static portions of the design do not rely on the state of the module under reconfiguration while reconfiguration is taking place. The implementation should ensure proper operation of the design during the reconfiguration process. Explicit handshaking logic may be required.
- 6) The states of the storage elements inside the reconfigurable module are preserved during and after the reconfiguration process. On the other hand, If set/reset initialization is required for the reconfigurable module, user-defined set/reset signals should be defined in the source HDL.

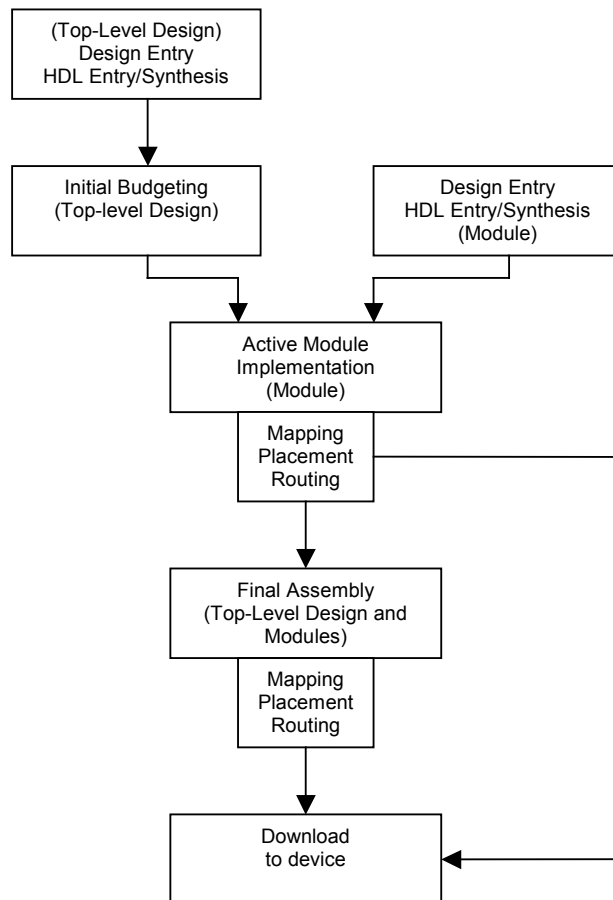


Figure 7: Module-Based Flow

A detailed description of this design flow is shown in Figure 7. A full design is first carried out, implemented and verified from top level to bottom modules, some of which are fixed while others are designated as reconfigurable. This generates the *Top-level Design* and *Module-level Design*. After the device is downloaded with initial power-up configuration, reprogram reconfigurable modules as needed with individual (or partial) bitstreams then can be updated to the device.

The communication between a reconfigurable module and a fixed module, or between the reconfigurable ones must go through dedicated wires, i.e. in order to maintain correct connections between the modules, must use the same physical wires. This is why the *bus macro* is used. By positioning the bus macro exactly straddling the dividing line between two adjacent modules, the inter-module connection is established. However, the standard bus macro provided by Xilinx can only establish communication between two modules next to each other. When a communication channel is required for two separate modules, i.e. one or more modules physically placed between them, a signal needs to pass through the reconfigurable modules connecting the two modules with standard bus macros used on both sides to make that connection. This effectively requires creation of an intermediate signal that is defined in the reconfigurable module. This signal cannot be actively used during the time the intermediate reconfigurable module is being configured.

To avoid this dilemma, a custom-made bus macro is designed. By extending the dedicated wires, the new bus macro provides communication between modules across multiple module boundaries. The dedicated routing that used by the bus macro going through the modules will be also reconfigured with the intermediate reconfigured module. But since it will be placed

full subtracter have three one-bit inputs and two one-bit outputs. When viewing these two modules as a black box externally, they are reconfiguration-compatible. Specifically, when analyzing the logic structure instantiated inside the black box, these two modules both use 2 LUTs with identical logic interconnections between LUTs and I/O signals. The only difference between them is only one truth table stored inside one LUT, which changes from 0xE8 to 0x8E. There exists a clear overlap between the configuration information for these two modules. When these two similar tasks need to be interchanged, the use of two separate precompiled configuration data files will occupy twice the storage space and twice the reconfiguration time. A more advantageous strategy would be to modify the corresponding logic content directly at runtime when switching between two tasks with similar or even identical logic structures, especially when the logic interconnections are identical. This can also potentially be extended to tasks even at a fine-grained level [61].

In Xilinx Virtex II/-Pro FPGAs, configuration memory is arranged in column-based vertical frames, i.e., one-bit wide extending from the top edge of the device to the bottom. These frames are the smallest addressable segments of the FPGA configuration memory space. Hence, all operations must act on whole configuration frames. Even if only one byte inside a frame is changed, such as the truth table of one LUT, the full frame needs to be rewritten. Configuration memory frames do not directly map to any single piece of hardware; rather, they configure a narrow vertical slice consisting of many physical resources. Therefore, we refer to the direct bit management process as a Frame-based Partial Reconfiguration Flow.

To utilize this flow at runtime, modules have to be implemented at the LUT level at design time when following Module-based Flow procedures. Besides the required 1-dimensional area constraints for the module, all of the logic elements that may require partial reconfiguration

at run-time have to be placed at specific physical locations using 2D area constraints. Thus, the representation scheme shown in Figure 4 is used to describe the module circuit. Since the primary Frame-based Partial Reconfiguration Flow only focuses on the logic modification of the modules without considering the changes of input/output signals or the logic interconnections, there are only two aspects that need to be focused on with the module circuit. The first is the LUT-level functionality and the second is the physical location of the LUTs.

Since there are no logic interconnection changes at runtime, the logic elements, i.e. the LUTs, are labeled with a fixed integer from 1 to N at design time, where N denotes the total number of LUTs used. After receiving the representation scheme from the top tier, the frame address is determined by the translation engine. Based on the calculated address, the corresponding logic function data of the frame can be read back. The mapping engine then continues interpreting the new logic information and loading into the frame. Details of the location calculation process will be discussed in Chapter 4.3. After this process, new frame data will be merged back into the running bitstream. Only the positions of the bitstream containing the user logic request have been substituted. Therefore, configuration outside of the dedicated area is not affected. With frame-based flow alone, bus macro may be eliminated from the design, which can potentially simplify the design from the module-based flow significantly.

3.3.3 Physical Area Management

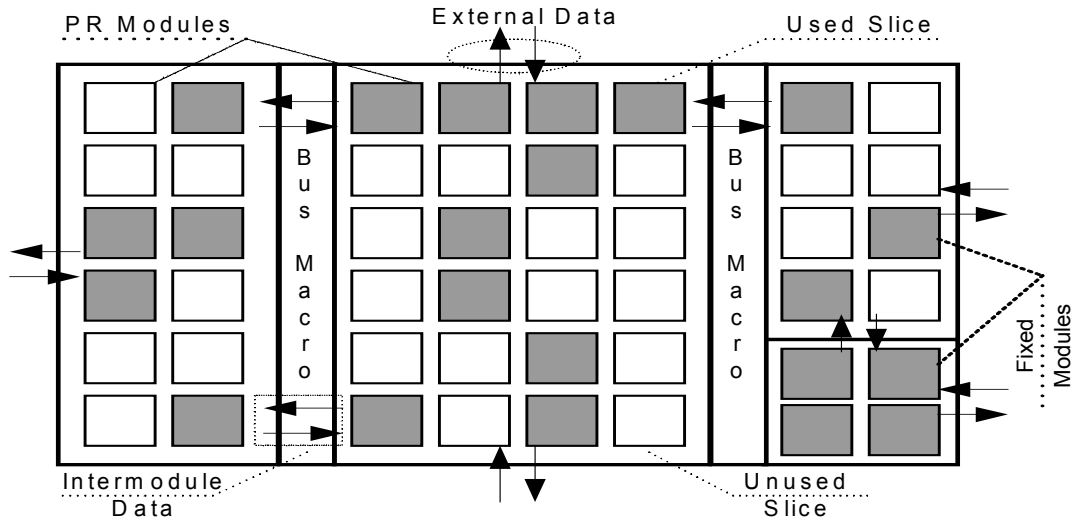


Figure 9: Physical Area Management

Establishing adequate reconfigurable regions and sufficient connectivity at design time is crucial for dynamic partial reconfiguration support. Furthermore, it is necessary to track the occupancy of these regions at run-time to maintain correct module re-allocation operations. MRRA area management strategies address both of these requirements.

As shown in Figure 9, the area management at module level is carried out at a 1-dimensional level. The size of any single occupied reconfigurable module is fixed after design time. Hence these modules can only be re-allocated to other same column-sized reconfigurable regions, given these regions provide identical inter-module interconnections for the external ports of the module. This is a limitation imposed by the module-based flow provided by Xilinx, where the reconfigurable module height is always the full height of the device. The width ranges from a minimum of four slices to a maximum of the full-device width, in four-slice increments. Manipulating the column addresses of a module's bitstream enables a module to be relocated.

When the module spreads across multiple CLB columns the first and leftmost column must be presented at the beginning, then the new CLB column value is automatically incremented internally. During the relocation process, the old column addresses of a module are established. Several FPGA hardware specific parameters are then used to generate the new major column addresses. Hence, the old values of the input bitstream are simply replaced by the newly calculated values. Since checksum data may be generated in the original reconfiguration data, several extra data words may have to be recalculated and updated during the process in order to relocate a module to another CLB column. Detailed calculation process will be discussed in Chapter 4.5.

Since only the column address of the module is changed, the relative position of all the logic resources and routing resources are kept intact and can be quickly shifted to other column positions. Hence, this process also requires that the relative position of inter-module interconnections for the external ports of the module be the same. A related approach for Virtex FPGAs has also been discussed in [16].

On the other hand, inside each module, slices can be placed and adjusted anywhere inside each module's reconfigurable region as shown in Figure 7. These arrangements can be carried out at a two-dimensional level, only limited to the height and the width of the region. These area modifications are translated at the slice level by the mapping engine. Therefore, this requires that the corresponding reconfigurable modules are implemented at least at the RTL-level or the more detailed LUT-level. The 2-dimensional adjustments can be potentially very useful to applications such as fault tolerance or Genetic Algorithms (GAs) [13], [30], [38] that are executed at the top level. Additionally, such adjustments also beneficially influence the size of configuration bitstream.

Based on the current control flow, 1D area management model at module level is a realistic choice, in which tasks can be allocated anywhere along the horizontal device dimension. The width ranges from a minimum of four slices to a maximum of the full-device width, in four-slice increments. The vertical dimension is fixed and spans the total height of the hardware task area. The 1D model leads to a placement problem of reduced complexity.

In this 1D model, an *area manager* is used. The area manager controls the free space on the device and tries to locate feasible placements for the tasks, i.e., tracing the unused area of the reconfigurable device with identical area. When tasks start or terminate execution, the placer updates this data structure to reflect the new allocation. When a task arrives, the placer has to search for matching free areas on the device. A 1D hash table is used to record the free area list.

Hashing is a process during which data items are stored in a data structure called *hash table*. A *hash function* maps a key to the entry in the hash table that holds the data item referenced to by the key. Given a device with W column RCUs, the direct-address hash table H is defined as an array of W elements. Free areas with size of w columns are stored in the element $H[w]$. Every element contains a pointer to a list of free column areas of the correspondent size. Based on the pre-defined inter-module connection, the elements in the same list are also labeled into different group. Clearly, the finding list operation normally only takes one step:

```
Return  $H[\text{size}]$ . Free_entry
```

Extra step may need to locate the area with identical inter-module connection. This searching step can cost up to $O(k)$, where k is the total number of pre-defined inter-module connection group. Therefore, the time complexity of area finding process is only $O(k)$, which is critical to the online performance. Whenever a new task is inserted or deleted, only the corresponding

element in the list may need to be scheduled to update. The time complexity of these updating operations is only $O(1)$.

3.4 Logic Layer Application

As mentioned in Chapter 1, FPGA fault tolerance is one of the typical areas that require the partial reconfiguration ability. Among all the algorithms that have been proposed in this area, Genetic Algorithms (GAs) are one of the most widely discussed and most sophisticated applications, which was selected to drive the development of this solution and used at algorithm layer. The goals of the selected GA algorithm are to *analytically model* the large-scale reconfigurability of on-chip resources and to *iteratively develop* adaptive reconfiguration techniques to occlude failures [49].

In the GA application, all phases of the fault handling process including *Isolation*, *Diagnosis*, and *Recovery* were integrated into a single cohesive approach [49]. An initial population of functionally identical (same input-output behavior), yet physically distinct (alternative design or place-and-route realization) FPGA configurations is produced at design time. At run-time, these individuals compete for selection based on fitness assessment favoring fault-free behavior. Hence, any physical resource exhibiting an operationally significant fault decreases the fitness of those configurations, which use it. Through runtime competition, the presence of the fault becomes occluded from the visibility of subsequent FPGA operations. Meanwhile, new device configurations are created as variations of faulty and viable configurations. This enables regeneration of lost functionality, realized directly as new configurations via the FPGA's normal throughput processing.

Definition 1: For the GA, each *generation* contains a number of P chromosomes. Each chromosome consists of a set of cells C^l , where l is the number of cells in each chromosome.

Definition 2: For each cell,

$$C_i = L^m \cup I^n$$

where, L^m is the set of function units within the cell C_i and I^n is the set of internal connectivity and the inputs/outputs of the function units within the cell C_i .

Definition 3: For each function unit sets L^m ,

$$L^m = \{L_i \mid L_i \in \Omega, i = 1, 2, \dots, m\}$$

where Ω is the logic function space. Currently, Ω contains five dyadic functions OR, AND, XOR, NOR, NAND and one unary-function NOT. The index m is the number of function units in the set.

Definition 4: For each interconnection set I^n ,

$$I^n = \{I_i \mid I_i = \langle L_j, L_k \rangle, i = 1, 2, \dots, n\} \quad (j, k = 0, 1, 2, \dots, m+1, \text{ and } j < k)$$

where $\langle L_j, L_k \rangle$ stands for the interconnection from function unit L_j to L_k , when $j=0$, it means the input from outside of the unit, when $k=m+1$ it means the output of the unit. The outputs of each cell are only allowed being inputs of the array with higher row numbers. Again, the index m denotes the number of function unit in the set and n is the number of connections in the set.

The major GA operations that will directly affect the task result for lower layer translations include crossover and mutation.

3.4.1 Low-Level Crossover Operator

In GA, *Crossover* is a genetic operator that combines (mates) two chromosomes (parents) to produce a new chromosome (offspring). In our GA, this operation performs a probability-driven single point genetic crossover on the two parent chromosomes. Whenever a faulty configuration P_1 has been detected and ready to be repaired, a correct configuration P_2 is also chosen from the pristine pool as another parent.

A Random Select Operator,

$$\text{RSO: } C^l \rightarrow C_i$$

is used to randomly select one cell from the chromosome to carry out the modified single point crossover operation. Then, a crossover point inside of the cell is also need to be selected and only the part of configurations after the select point is replaced.

$$\langle L_j, C_i, P_2 \rangle \rightarrow \langle L_j, C_i, P_1 \rangle$$

3.4.2 Low-Level Mutation Operator

In genetic algorithms, mutation is a genetic operator used to increase genetic diversity from one generation of a population to the next generation. In the high-level algorithm used here, the operation performs a probability-driven single-bit genetic mutation. One randomly selected bit of the binary chromosome content is inverted.

Here, the Random Select Operator first selects the target cell:

$$\text{RSO: } C^l \rightarrow C_i$$

Then the function unit is chosen and the function is also randomly pick up from the function space Ω :

$$\text{RSO: } (L^m, \Omega) \rightarrow L_j$$

Part of the interconnections inside the cell may be able to be randomly modified as well:

$$\text{RSO: } I^n \rightarrow I_k, \dots, I_l$$

With all these modifications

$$\langle I_k, \dots, I_l, L_j, C_i, P_1 \rangle \rightarrow K$$

the new individual chromosome K is generated.

The details of how to map these operations into real FPGA hardware data structures will be discussed in Chapter 6. The performance and limitation of these operations based on the available hardware structure will also be evaluated and analyzed in Chapter 6.

3.5 Chapter Summary

This chapter introduced the MRRA concept in detail. In Chapter 3.1, design considerations for the autonomous reconfiguration architecture were discussed, including both the hardware design requirement and the logic control flow requirement. Chapter 3.2 presents the details of the three-layer paradigm, including the *Logic Layer*, the *Translation Layer* and the *Reconfiguration Layer*. The Reconfiguration Layer is mostly a hardware layer, which provides the basic reconfiguration interface and multiple testing and communication hardware control modules. The Translation Layer shields the details of the hardware from the general control logic by presenting a translation engine with a set of standard APIs. This translation engine can interpret the hardware independent logic, including both the logic functions and the physical

routings, into corresponding hardware dependent digital circuit implementations. A general data structure containing the hardware logic function information as well as physical allocation information has been defined in Logic Layer. High-level algorithms only need to be concerned with these general data structures and performs their normal routines without worrying about the hardware compatibility. The final modification result will be stored in the defined general data structure and pass to the Translation Layer for further interpretation.

Chapter 3.3 discusses the *Dynamic Control Flow*, which provide the *Adopted Module Based Flow* at design phase, the *Frame Based Flow* at runtime phase and the *Physical Area Management* control. At design time, the adopted Module Based Flow is used to generate the full design. The full hardware system is first partitioned into modules and designed from the top view to the bottom. Meanwhile, *One-Dimensional Area Management* is required to perform on the full physical FPGA device by partitioning it into 1D column-based rectangles, in which all the fixed and reconfigurable modules will be arranged based on the size of each module and extra specified area constraints from design specifications. Then, all the individual modules are created by *Final Assembly* based on the top-level view and are ready to be downloaded to the FPGA device as Configuration Data bitstreams.

After the initial bitstream is downloaded, new modification requests can be generated by using the Frame Based Flow at run-time instead of using predefined and precompiled modules. Logic function modification request for each Look UP Table (LUT) inside modules can also be generated based on the user requirement. Requests from the Logic Layer will eventually wait for the translation engine from lower layer to interpret to the corresponding configuration date file and reconfigured by the Reconfiguration Layer.

Finally, Chapter 3.4 illustrated the basic *Genetic Algorithm* conceptions for FPGA high-level fault tolerance application based on the MRRA structure and control flow. The crossover and mutation operators of GA were also defined at conceptual level.

CHAPTER 4: RECONFIGURATION AND VERIFICATION METHODOLOGY

FPGA devices are configured by loading application-specific configuration data into their internal SRAM [5]. Configuration is carried out using a subset of the device pins, some of which are dedicated, while others can be reused as general-purpose inputs and outputs after configuration is complete. When integrating the basic partial reconfiguration flow with high-level algorithms, several basic questions need to be addressed:

- Which hardware interface should be used for the reconfiguration process?
- What fixed hardware modules are required to support control logic?
- How can a communication mechanism be established between the low-level hardware components and the high-level applications through the chosen interface?
- How can the partial reconfiguration modules be verified after being reconfigured?

What strategies and tools can be used in this verification process?

In order to provide a solution to these considerations, three different mainstream reconfiguration interfaces are explored and analyzed in the remainder of the Chapter. Customized system and methodologies are also designed and developed based on the characteristics of each interface.

4.1 SelectMAP Based Methodology

4.1.1 SelectMAP Interface

SelectMAP interface is a parallel reconfiguration interface that provides an eight-bit bidirectional data bus interface to the FPGA configuration logic [73]. It can be used either in Master Mode with the CCLK signal considered as an output from the FPGA, or in Slave mode with the CCLK signal considered as an input. In slave mode, SelectMAP allows for both configuration and readback, while in master mode only configuration is possible.

In SelectMAP slave mode, 14 pins, including DATA pins $D[0:7]$, CCLK, RDWR_B, BUSY, CS_B, PROG_B, DONE, and INIT_B, are required to carry out the reconfiguration process. These pins will need to be placed at specific positions of the general-purpose I/O blocks within the FPGA by specifying user constraints.

SelectMAP configuration data is loaded one byte at a time presented on the $D[0:7]$ bus on each rising CCLK edge with the Most Significant Bit (MSB) of each configuration byte on the D0 pin. Two extra control signals are present for SelectMAP, CS_B and RDWR_B. These signals must both be asserted Low for a configuration byte to be transferred to the FPGA. A third signal, BUSY, is an output from the FPGA. When SelectMAP configuration is running at high frequency, such as greater than 50 MHz for Virtex, Virtex-E, and Spartan-II and greater than for Virtex-II [73], the BUSY line must be monitored to ensure that data was transferred. If BUSY is High, this indicates that the last data byte was not transferred and must remain on the data bus.

Multiple devices can be connected on the same SelectMAP bus. To do so, the DATA pins $D[0:7]$, CCLK, RDWR_B, BUSY, PROG_B, DONE, and INIT_B are connected in common

between all devices. The CS_B Chip Select inputs are kept separate, so that each device can be accessed individually. External control logic is required to arbitrate between devices by asserting and de-asserting the CS_B signals as necessary.

4.1.2 System Design

In this dissertation, all the designs are built to support an Avenet Virtex II Pro Development board. In our development board, only one FPGA device is connected to the reconfiguration interface. The clock is setup to be generated from outside the FPGA itself while readback may be required for user verification purposes. Based on these factors, a slave mode SelectMAP interface is chosen within a Loosely Coupled system, in which all the high-level control logic originates from an external PC host.

In order to communicate with the host PC through the SelectMAP interface in slave mode, a PCI bus interface is normally used to establish the connection. This requires the instantiation of a PCI core either inside the target FPGA chip or inside a dedicated bridge chip on the FPGA board. To complete this setup, SRAM modules may be needed as buffers for the purpose of receiving data from the host PC and reconfiguring the target FPGA.

Figure 10 shows the detailed schematic view for the modular hardware platform of the SelectMAP-based MRRA architecture designed for Xilinx Virtex II/-Pro architecture. This platform has been designed as a full on-chip hardware subsystem. The hardware subsystem includes two subsets comprised of system resources and operational resources [18]. The system resources include an on-chip PowerPC core as the control element, the on-chip Block RAMs and all the external peripherals such as the SRAM, which acts as shared memory and can be accessed

by both the external Host PC and the on-chip PowerPC, and the RS232 interface. The operational resources are the actual FPGA modules instantiated inside the FPGA. It consists of a fixed resource subset that is held constant during the entire process and is used to control the on-chip data communications and on-board peripherals, as well as a reconfigurable resource subset that is used for the user-defined partial reconfiguration applications.

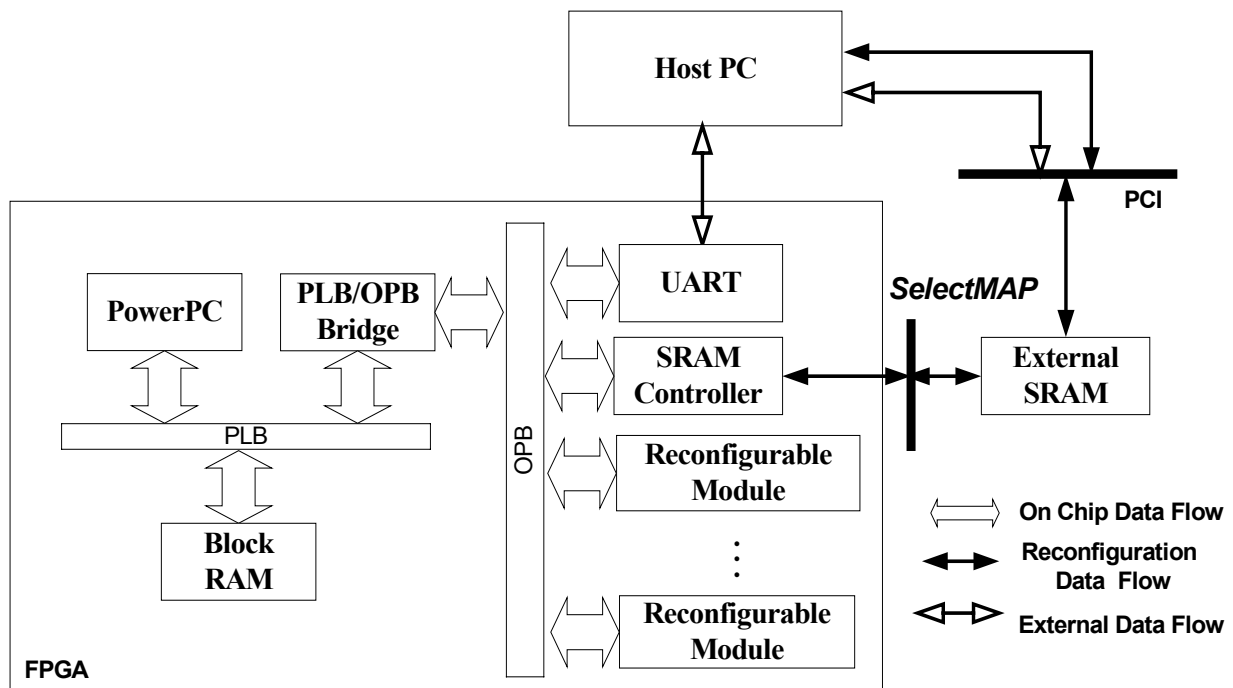


Figure 10: SelectMAP based System

The fixed operational resources include multiple control interfaces for the external on-board SRAM, RS232 resources to communicate with the host PC, and Block RAM controllers serving the PowerPC core and Block RAMs. A three-segment path is also established in the fixed operational resource region to connect all the modules, which includes an On-chip Peripheral Bus (OPB), a Processor Local Bus (PLB) and a bridge core (PLB2OPB) providing access to the OPB from the PLB.

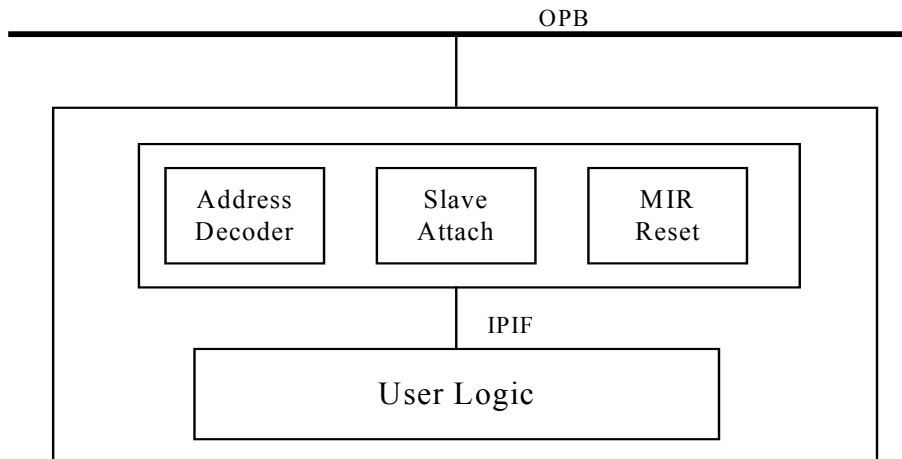


Figure 11: IPIF Template

The remaining modules comprise the reconfigurable subset. In order to maintain uniformed communication with the PowerPC core through the OPB bus with those reconfigurable and relocated modules, a simplified *Intellectual Property Interface and Bus Macro* structures have been adopted. Figure 11 shows the basic template. Each module is attached to the OPB bus in slave mode and each contains 2 components: a component for the user logic and a component for an IPIF. The user logic component carries out the original designed functions. The IPIF provides a simple standard OPB slave interface uses only the slave signal set and bus attachment logic required for a slave interface. Using these structures, all the modules can communicate and be controlled seamlessly.

In addition to the reconfiguration interfaces, the reconfiguration layer also provides two bidirectional communication channels connecting it with the host microprocessor. The first is the standard RS232 serial channel. It is primarily used for debugging and monitoring purposes which normally only require small amounts of data exchanges.

The other communication channel is realized as a custom parallel protocol through the external SRAM and PCI bus. As shown in Figure 10, the input path begins at the PC, which transmits reconfiguration data over the PCI bus to the specific address of an on-board SRAM module, and then sets the data availability flag. Concurrently, the PowerPC polls the flag status, reads data from the addressed SRAM module, and copies data to the hardware component for processing when the data flag is set. On the other hand, the output path data starts from PowerPC reading the data from other hardware modules through the 3-segment bus path. The data width can be varied as 8 bits, 16 bits or 32 bits. This bidirectional channel can be used when a large validation process is required.

4.1.3 Testing Strategy and Communication Channel

Because there are few sophisticated testing tools available for testing the SelectMAP interface at the present time, the communication protocol and testing APIs for this prototype configuration are designed and built from the ground up.

For testing purposes, the communication channel needs to be bi-directional. The on-chip PowerPC is used to control the handshaking between the board and the host PC. To monitor the data exchange and reconfiguration process, the on-board RS232 is subsequently used to connect to the serial port of the host PC. After the board is installed in a PCI slot of the host PC, it is then controlled by the debugging program running on the host PC [21].

As shown in Figure 10, the input path begins at the PC. The PC sends the data over the PCI bus to the specific address of an on-board SRAM module, and then sets the data availability flag. At the same time, the PowerPC core keeps checking the flag, reads data from the addressed

SRAM module, and sends data to the hardware component for processing when the data flag is set. On the other hand, the output path data starts from the PowerPC when the latter reads the data from other hardware modules through the 3-segment bus path, and then writes it to the on-board SRAM by setting the data availability flag. In the meantime, the PC host keeps polling SRAM location for the data availability flag. When the flag is set, the PC host receives the data over the PCI bus, and saves it in its memory for further processing when ready.

Table 3: Communication APIs on PowerPC

API name	Input Parameter	Operation	Data Width
Intc_setup	N/A	Initializes the interrupt controller	N/A
DeviceDriver Handler	*CallbackRef	The corresponding interruption routine	N/A
mem_dump	unsigned start_addr, unsigned end_addr	Reads the on board and on-chip memory®ister	32 bits
mem_write	unsigned wr_addr, unsigned wr_value	Writes the on board and on-chip memory®ister	32 bits
flash_test	unsigned start_addr, unsigned end_addr	Thorough validation test on the flash	32 bits
mem_test	unsigned start_addr, unsigned end_addr	Thorough validation test on the SRAM	32 bits

This bi-directional path is supported by the on-board status register, which maintains the PCI ownership control bit. The combination of the drivers running on the PC and the control program running on the PowerPC in the Virtex-II Pro fabric achieve PCI ownership of the SRAM bus to retrieve the data. After the initial boot up, the PCI ownership is controlled by the host PC until it clears the PCI ownership control bit of the status register. Thus, the PowerPC and the host PC are able to synchronize the exchange of data between each other by polling the data status flag inside the SRAM.

Table 4: Communication APIs on Host PC

API NAME	INPUT PARAMETER	OPERATION	DATA WIDTH
Initial	N/A	Recognizes and Initializes the FPGA board	N/A
WriteBitFile	char Filename[]	Reads configuration file from the board to Host PC	File length
ReadBitFile	char Filename[]	Reads the configuration file from the Host to the FPGA	File length
ByteRead	unsigned long StartAddr, unsigned long EndAddr, int AccessBar	Reads the on board memory	8 bits
WordRead	unsigned long StartAddr, unsigned long EndAddr, int AccessBar	Reads the on board memory	16 bits
DWordRead	unsigned long StartAddr, unsigned long EndAddr, int AccessBar, unsigned long AccessData	Reads the on board memory	32 bits
ByteWrite	unsigned long StartAddr, unsigned long EndAddr, int AccessBar, unsigned long AccessData	Writes to the on board memory	8 bits
WordWrite	unsigned long StartAddr, unsigned long EndAddr, int AccessBar, unsigned long AccessData	Writes to the on board memory	16 bits
DWordWrite	unsigned long StartAddr, unsigned long EndAddr, int AccessBar, unsigned long AccessData	Writes to the on board memory	32 bits

The PowerPC embedded in the FPGA fabric is running in standalone mode by automatically generating the software structure. On top of this basic operating system, the standard communication API interface is then developed. The communication APIs on the

PowerPC side are shown in Table 3. For the on-board memory operations, a 32-bit data width is provided as determined by the on-chip bus width.

The standard communication API interfaces for PC have been developed as listed in Table 4. These API interfaces are categorized into 3 parts: *Initialization Operations*, *Configuration File Operations* and *On-board memory Operations*. For the on-board memory operations, 3 different data width, 8 bits, 16 bits and 32 bits, are provided. The WinDriver routines [67] from Jungo Software for the PCI interface are also used to realize part of this API functionality..

4.2 JTAG Based Methodology

4.2.1 JTAG Interface

The IEEE 1149.1 Test Access Port and Boundary-Scan Architecture is commonly referred to as the *Joint Test Action Group (JTAG)* interface, which is the technical subcommittee initially responsible for developing the standard. This standard provides a means to assure the integrity of individual components and the interconnections between them at the board level.

Devices containing JTAG boundary-scan logic can send data out on I/O pins in order to test connections between devices at the board level [74]. The circuitry can also be used to send signals internally to test the device-specific behavior. These tests are commonly used to detect opens and shorts at both the board and device level. In addition to testing, boundary-scan offers the flexibility for a device to have its own set of user-defined instructions. The added common vendor specific instructions, such as configure and verify, have increased the popularity of boundary-scan testing and functionality.

JTAG is also well suited for downloading configuration bitstreams to FPGAs. The IEEE 1149.1 standard defines a four-wire serial interface (a fifth wire is optional) as designated in the Test Access Port (TAP) to access complex Integrated Circuits (ICs) such as microprocessors, DSPs, ASICs, and CPLDs. In addition to the TAP, a compliant IC also contains shift registers and a state machine to execute the boundary-scan functions. Figure 12 shows the details of this control state machine. Data entering the chip on the Test Data In (TDI) pin is stored in the instruction register or in one of the data registers. Serial data leaves the chip on the Test Data Out (TDO) pin. The boundary-scan logic is clocked by the signal on Test Clock (TCK) while the Test Mode Select (TMS) signal drives the state of the TAP controller. Although it is optional, Test Reset (TRST) can serve as a hardware-reset signal. Multiple scan-compatible ICs may be serially interconnected on the printed circuit board, forming one or more boundary-scan chains, each chain having its own TAP. Each scan chain provides electrical access from the serial TAP interface to every pin on every IC that is part of the chain.

Communication with the JTAG controller is a two-step process. First, an instruction is shifted into the instruction register (IR). This instruction is decoded and selects one of the data registers (DR) for shifting data into (on TDI) and at the same time selecting the same DR as the source of data for TDO. The selected DR is then accessed. Accessing the DR always requires shifting N bits of data into the register on TDI and receiving N bits of data from the register on TDO (where N is the size of the register). Each DR can be read only, write only, or read/write. If a DR is read only, the N bits that are shifted in during an access are discarded and the N bits shifted out are returned on the TDO line. In the case of a write-only data register, the N bits shifted in are captured in the register and the N bits shifted out are meaningless.

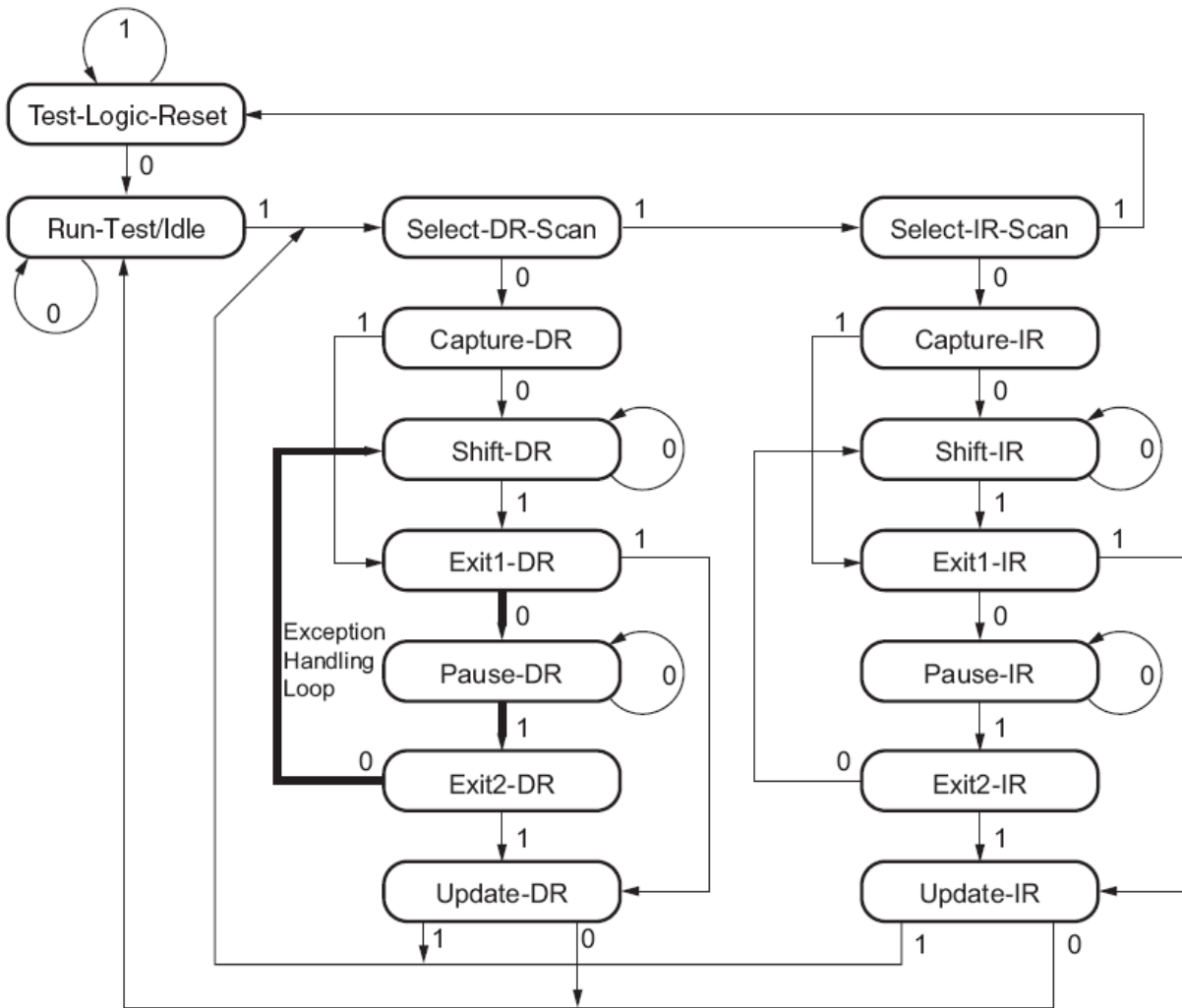


Figure 12: JTAG Control State Machine [80]

While the SelectMAP interface requires occupying the general purpose I/O pins in the FPGA chip, the JTAG pins are the pre-assigned dedicate ones instead. Moreover, SRAM modules can be spared since the JTAG interface does not need them for reconfiguration.

4.2.2 Reconfiguration Control

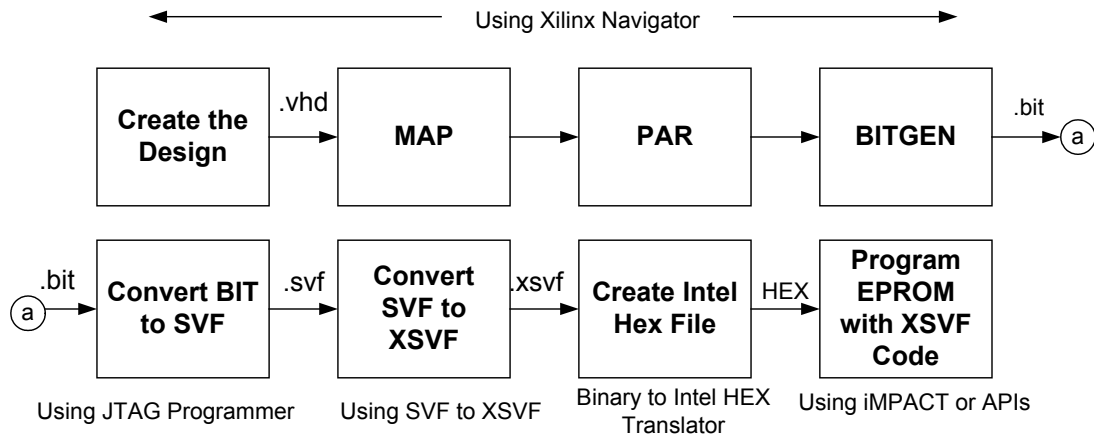


Figure 13: Programming Flow under JTAG Control

As shown in Figure 13, the FPGA programming flow that was used for JTAG-controlled partial reconfiguration begins with design entry in VHDL/Verilog [20]. The design is first simulated using Modelsim for functional verification. Simulation is followed by synthesis, placement, and routing with generation of the `.bit` file using Xilinx Navigator. The JTAG operations are recorded in the Serial Vector Format (`.svf`) file with the iMPACT or JTAG Programmer.

To carry out the autonomous reconfiguration operation, an API from Xilinx application note Xapp058 is adopted [80]. This API, developed in the C programming language, can be ported to any microcontroller including a host PC. Instead of using a `.bit` file directly, this API is adopted into JTAG prototype using the `.xsvf` format, which needs to be converted by the Xilinx tools ahead of time. An `.xsvf` file is a compact binary file based on the `.svf` file. This format consists of scan operations and movements between different stable states on the IEEE 1149.1 state diagram. The `.svf` files were developed by Texas Instruments and have been adopted as a standard for data interchange by JTAG test equipment and software manufacturers.

4.2.3 Testing Strategy and System Design

Because JTAG is a very popular interface, numerous commercial tools are available for testing and debugging purposes. Therefore for JTAG interface based system, testing can be carried out either using commercial COTS software directly or user customized structure and applications.

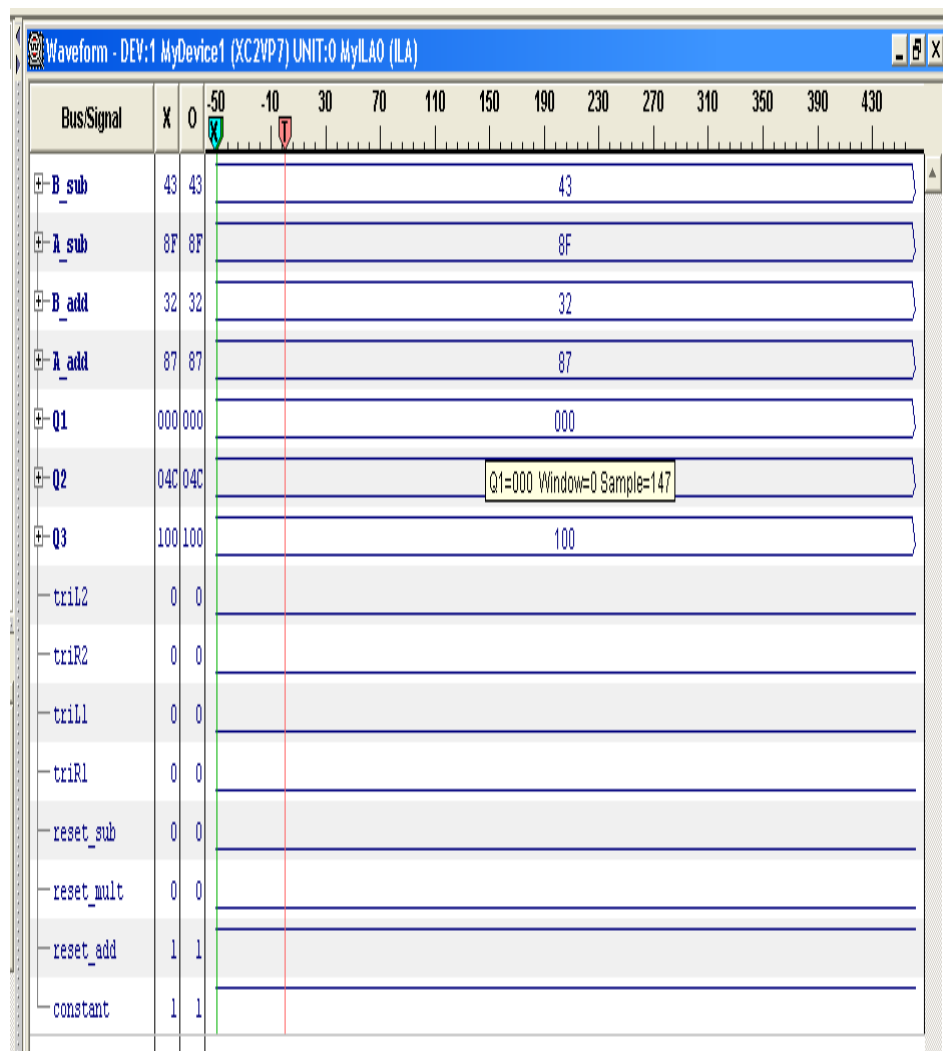


Figure 15: ChipScope Logic Analyzer [20]

ChipScope based testing

ChipScope Pro is such a JTAG based real-time verification tools that provide on-chip debug at or near operating system speed. It inserts logic analyzer, bus analyzer, and Virtual I/O low-profile software cores directly into the design, allowing user to view any internal signal or node, including embedded hard or soft processors. Signals are captured at or near operating system speed and brought out through the programming interface, freeing up pins for the design [75]. Captured signals can then be analyzed through the ChipScope Pro Logic Analyzer as shown in Figure 15.

Choosing such a tool as ChipScope Pro for verification eliminates the requirements for complicated bi-directional communication channels as required with the SelectMAP interface. Therefore, the hardware platform has been greatly simplified. A number of IP cores, such as the PowerPC, the Block RAM controller, the three-segment bus, and the SRAM controller can be removed from the fixed region of the FPGA chip leaving only the basic control logic. The IPIF overhead for each reconfigurable module also becomes unnecessary in the JTAG-controlled design. Only the basic functional modules and bus macros are required in the FPGA device.

Figure 16 shows the verification architecture of the JTAG prototype. All of the ChipScope Pro cores use the JTAG Boundary Scan port to communicate to the host PC via a JTAG downloading cable. The Virtual Input/Output (VIO) core is a customizable core that can both monitor and drive internal FPGA signals in real-time [75]. There are four kinds of signals available in a VIO core: (i) asynchronous inputs, (ii) synchronous inputs, (iii) asynchronous outputs, and (iv) synchronous outputs. In our system, synchronous outputs are used from the VIO cores as the testing pattern generator. Every VIO synchronous output has the ability to

output a static 1, a static 0, or a pulse train of successive values. A pulse train is a 16-clockcycle sequence of 1's and 0's that is driven out of the core on successive clock cycles. Synchronous inputs are also used to read back the corresponding result from the output signals and displayed in the ChipScope Pro Analyzer as virtual LED indicators.

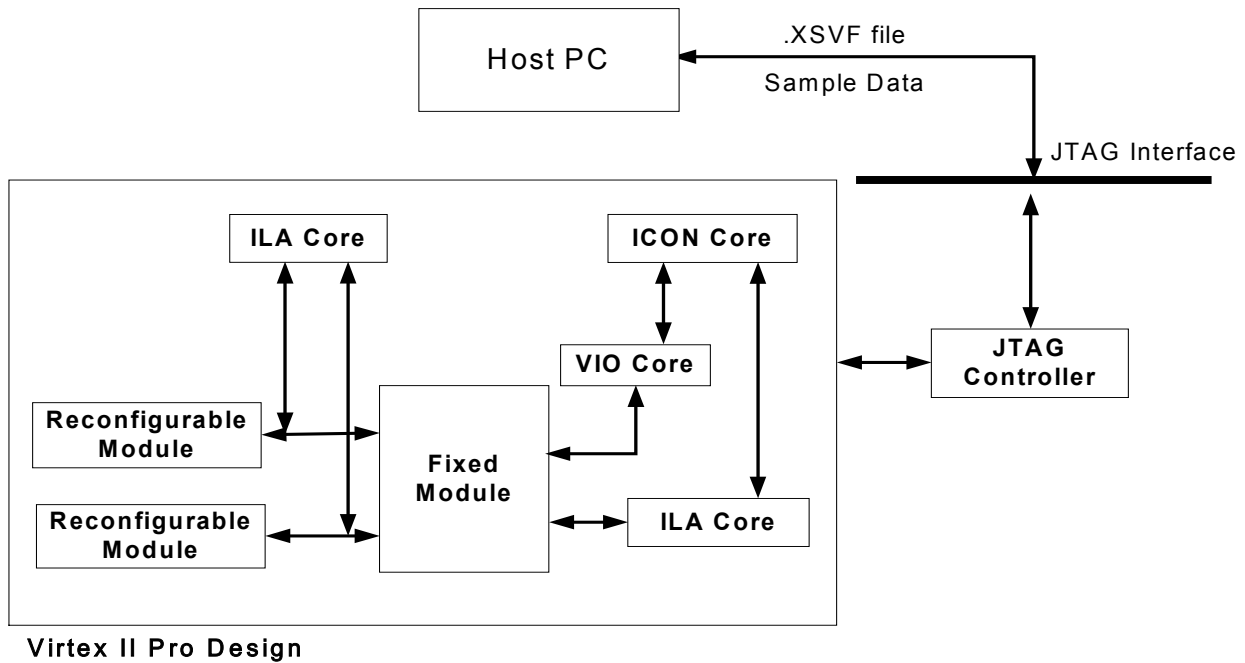


Figure 16: Loosely Coupled System for JTAG [20]

The Integrated Controller (ICON) core is used as a testing control logic core and provides a communications path between the JTAG Boundary Scan port of the target FPGA and the VIO/ILA cores. When necessary, the Integrated Logic Analyzer (ILA) core is another optional module that can be adapted to the testing system, working as customizable logic analyzer to monitor any internal signal of the design. Since the ILA core is synchronous to the monitored design, all design clock constraints that are applied to the design are also applied to the components inside the ILA core. Detailed information about applying these cores can be found in [75].

There are two ways to insert the debugging cores into the user design. First is the Automatic Insertion of Cores, which use ChipScope Core Inserter to incorporate the probe into the user design. This requires less work at the user part. But not every type of cores is supported by the Inserter. Manual Insertion of Cores is the second way. After the ChipScope cores are generated using the Core Generator, users instantiate the core modules, wired and implemented into the design by their code directly. This requires more work. But on the other hand it brings the flexibility and eliminates the limitation of the Inserter tools. Figure 17 demonstrates the details of this process.

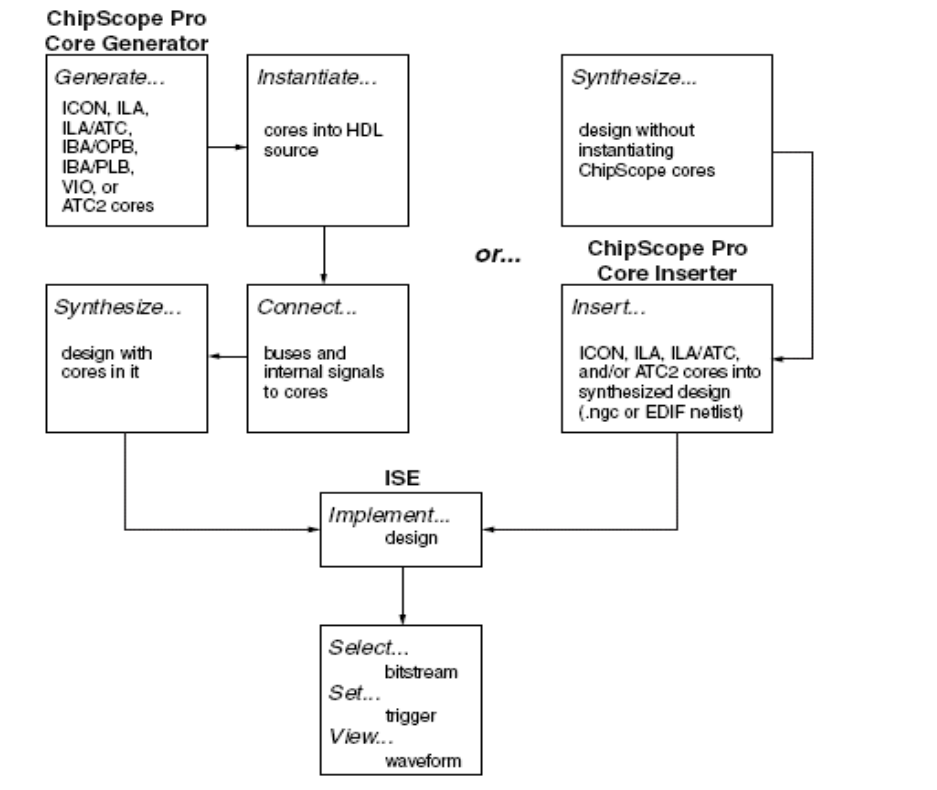


Figure 17: ChipScope Core Insertion Procedure

GNAT based testing

Using JTAG based commercial testing tools can significantly reduce development time. However, these tools were designed to satisfy a variety of testing environments and requirements. Hence they normally have a very sophisticated and complicated structure, which can consume a lot of routing resources. On the other hand, partial reconfiguration design requiring module level design has very strict routing boundary constraints. With an increase in the number of modules, less and less routing flexibility is available for these commercial tools such as ChipScope. Moreover, to debug the signals, commercial tools normally requires using a Graphical User Interface (GUI), which conflicts with the autonomous operation requirement. Alternatively, custom scripts may be created, but can slow down the speed while requiring extra programming interfaces with the other part of the algorithms. Therefore a second user customized verification system, General-purpose Native JTAG Tester (GNAT), is also designed to satisfy the routing flexibility as well as the autonomous operations with simple API interface.

The JTAG controller not only provides reconfiguration ability, but also supports a number of custom JTAG instructions and corresponding data registers. One such example is the USER custom instructions. These instructions allow the JTAG controller to communicate with user logic with FPGA. Figure 18 is a block diagram illustrating how the JTAG controller connects to the FPGA fabric. A dedicated hardware component inside the Xilinx FPGA, named BSCAN is used from the Xilinx UniSim libraries. This hardware component enables the FPGA general logic to communicate with the host PC through JTAG controller. The GNAT module is developed as part of the bitstream on the reconfigurable area of the chip. It connects to the BSCAN block via the TDI, TDO, and Control. Inside the GNAT module, The BSCAN

component provides all of the control signals. There is also a USER Data Register (DR) inside the module. A USER DR is just a shift register with expandable width.

In Figure 18 example, it shows a 10-bit USER DR. The upper 2 bits of the register are used to control access to the GNAT peripherals. Since only 2 peripherals are connected to the GNAT, only 1 bit is needed to identify the peripheral [52]. The other bit is allocated as an opcode (read/write). The lower 8 bits are for data communication. The GNAT component provides a unique Chip Select signal for each peripheral as well as read/write control signals and writedata. A readable GNAT peripheral is responsible for providing read data. GNAT provides the multiplex function with the coming data. For the peripherals that connect to the GNAT, a decoder and register wrapper is required to map the various input and output signals to a uniformed interface signals that connects to the GNAT.

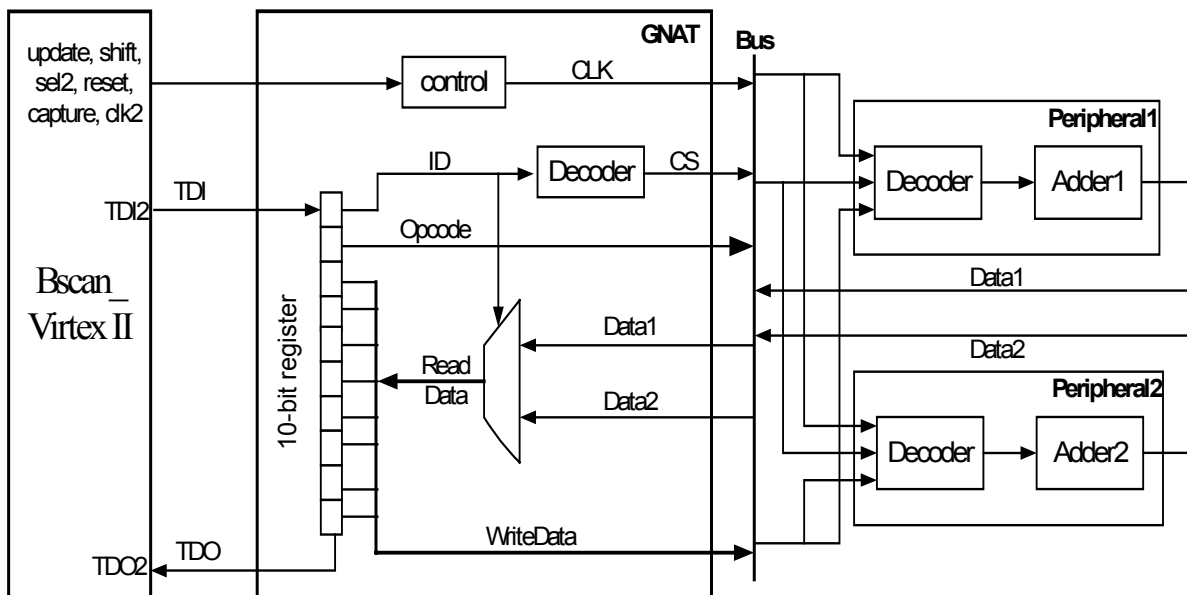


Figure 18: GNAT Based System

The combination of the BSCAN and the GNAT blocks provide a very simple but powerful communication mechanism to the system testing and debugging. The overhead is low

and because it is all module-based configurable, the size of the USER DR and the definition of all bits can be easily expanded and re-defined to meet a different verification requirement.

A set of corresponding control and testing APIs are developed on the Host PC to carry out the debugging procedure. Table 5 shows the details of these functions. Similar to the SelectMAP interface, these functions can be categorized into *Initialization Operations*, *Configuration File Operations*, and *Testing Operations*

Table 5: Testing APIs for JTAG

API name	Input Parameter	Operation	Data Width
xInitialize	N/A	Initialize the JTAG chain	N/A
DownloadDesign	*filename	Download the new configuration data to the board	N/A
xWrite	unsigned long data	Send testing data to the FPGA	
xRead	unsigned int id, unsigned long* result	Get testing result from the FPGA	32 bits
xCleanup	N/A	Close the configuration chain	N/A

4.3 ICAP Based Methodology

4.3.1 ICAP Interface

Modern FPGAs contain ample computational resources consisting of microprocessors and memory, which brings the possibility to establish a complete framework to move reconfiguration management from the outside environment into the SOC.

Figure 19 shows the components of such system. The partial reconfiguration logic resides in the FPGA blockRAM directly. The on-chip CPU core will solely responsible for the runtime algorithm running. The reconfiguration process will be carried out through the ICAP. Such an approach will provide the application with a high degree of self-containment and autonomy. In addition, it will decrease substantially the overhead associated with communication between the reconfigurable device and its environment.

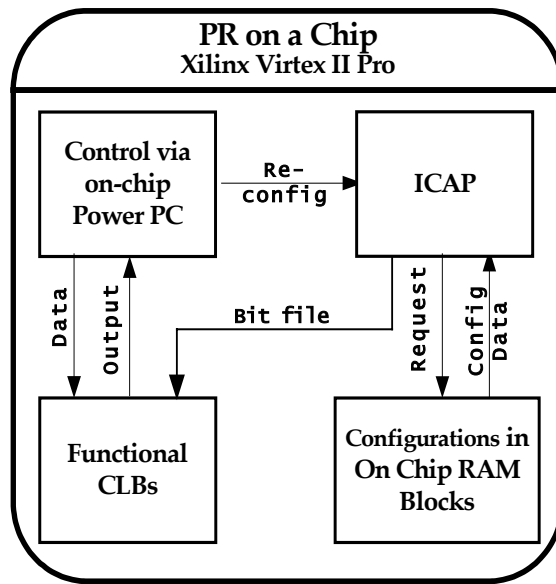


Figure 19: System On Chip Example

The ICAP block is located in the lower right hand corner of the FPGA [70]. It is used to access the device configuration registers as well as to transfer configuration data. The ICAP primitive provides access to device configuration data when connected to user logic in the FPGA fabric. Currently it is only available in the Virtex family of devices, including Virtex II, Virtex II Pro, as well as Virtex IV and V.

The protocol used to communicate with ICAP is a subset of the SelectMAP protocol. It includes pins of $Di[0:7]$, $CCLK$, $RDWR_B$, $Enable$, $BUSY$, and $Do[0:7]$. ICAP, as

indicated by its name, is an internally accessed resource and not intended for full device configuration. The ICAP module supports readback and partial reconfiguration of its own FPGA. This self-reconfiguration capability enables adaptive systems based on the PPC405 and the ICAP module. Consistent with the SelectMAP interface, the ICAP module provides read and write access to all configuration data. In the design, the ICAP module is also a slave IPIF interfaced module on the PLB bus controlled by the PPC405 processor. It has been reported that at optimal speed, the ICAP interface has a throughput of 5 MB/s running at 50 MHz [70].

4.3.2 System Design

Figure 20: shows the Runtime Reconfiguration and Testing System with ICAP. This system is very similar to the SelectMAP systems. However, since there is a PowerPC core in the Virtex-II Pro, most of the control logic can be implemented directly inside the chip to form a tightly coupled on-chip system based on the PPC405, block RAM or external RAM. The SelectMAP or the JTAG interface is mainly for the board initialization only. The host PC is connected to the board with RS232 interface and monitor the test outputs only.

The basic partial reconfiguration process through ICAP proceeds as follows:

After power-up and initial configuration with a base FPGA design through JTAG or SelectMAP interface, the top algorithm then commands the PPC405 processor to modify the reconfigurable module attributes. The PPC405 modifies the module attributes by first reading the contents of the specified module configuration frame through ICAP. The read-in bit stream will be transformed into a shorter bit stream representation or stay unchanged and passed up to the top algorithm layer more further modifications. Then the change is performed on the recently

read and stored frame based on the algorithm such as GA. Lastly, the modified frame, which is held in block RAM or external RAM, is written out to the FPGA reconfigurable area through the ICAP. This flow is described as the Read-Modify-Write of a configuration frame process. No external devices are required to implement this solution.

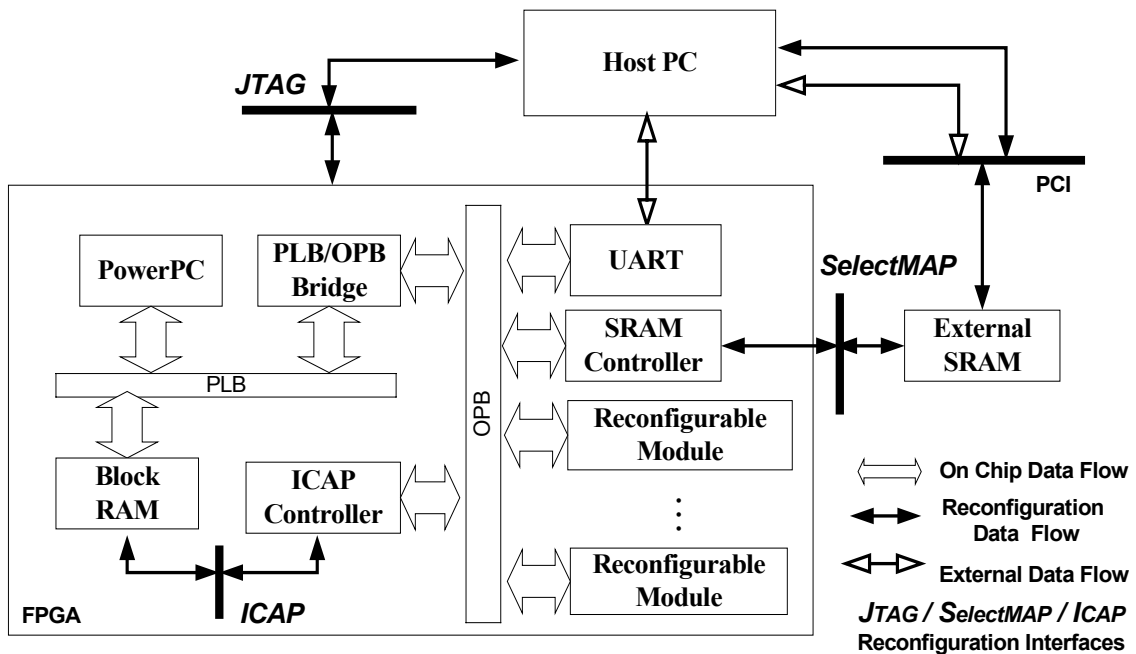


Figure 20: ICAP Based Testing System

The biggest change for this SOC solution from the previous prototype version is that the partial reconfiguration now is carried out by a FPGA module, named ICAP, inside the FPGA instead of the operations through PCI and SRAM. This change means the reconfiguration layer will now only rely on the on-chip hardware instead of any extra software communication APIs. This communication improvement may be critical for large throughput applications as GA.

Another change for the SOC solution is that since there is no host PC gets involved during partial reconfiguration process, no standard Xilinx tool will be available to use either. Bit

stream or pseudo bit stream will have to be used directly for upper layer logic algorithms. Thus, Frame-based workflow has to be used in this situation. Without the bulky CAD tools involved, the workload of the translation layer is greatly reduced. On the other hand, in order to use bit stream or simplified pseudo bit stream directly at algorithm level, many reverse engineering activities need to be carried out in order to obtain an inside understanding of direct the bit stream operations before any sophisticated upper layer algorithms can be developed. This tedious task was completed for Virtex II Pro as described in the subsequent section.

4.3.3 Bit Manipulation and Reconfiguration Control

Based on the data available in [69], [71] and a series of reverse engineering studies performed [17], a set of equations has been deduced to locate the corresponding logic content inside the bitstream file. In the Virtex II/-Pro architecture, each CLB has 4 slices placed in 2 rows and 2 columns. These slices are numerated in the format X_iY_j , where X is the slice column number. i range from 0 to $2N-1$, beginning from the left to the right where N is the number of CLB columns. Y is the row number. The variable j ranges from 0 to $2K-1$, beginning from the bottom to the top where K denotes the number of CLB rows, e.g. for an XC2VP7, there are 40 x 34 CLBs so $N=40$ and $K=34$. Therefore, 80 rows numbered 0 through 79 inclusive, and 68 columns numbered 0 through 67 inclusive of slices are available. Each one of these slices has 2 LUTs called G-LUT and F-LUT, which are normally occupied by the tools in that order.

Xilinx FPGA configuration data are described at frame level. Each configuration frame has a unique 32-bit address that is composed of a Block Address (BA), a Major Address (MJA),

a Minor Address (MNA), and a byte number. The major address identifies a specific column within a block, and the minor address identifies a specific frame within a column.

To locate a specific slice or LUT, the correct frame address has to be located first. As described above, the 32-bit frame address starts with a BA, which is always seven '0's for the CLB column. Next, the following 8 bits are MJA, which can be calculated by:

$$MJA = \lfloor X \div 2 \rfloor + overhead$$

where X is the slice column number and overhead includes the total number of the GCLK columns and the leftmost IOB and IOI columns, which is normally a static number equal to 3 and can be confirmed using the device data sheet if there is any variation.

MNA is represented by the next 8 bits following the MJA inside the frame address. As mentioned earlier, all the logic for each slice column resides only in the second and the third frame. Therefore,

$$MNA = X \bmod 2 + 1$$

where X is the same as in Equation (2). Since the frame number starts from '0 instead of '1', only a value of 1 needs to be added in Equation (3). With extra 9 bits of '0' after the MNA, the frame address is formed.

Inside the logic frame, for each slice, the LUTs logic occupies 5 bytes and there are always 12 bytes at the beginning of the frame as overhead. So the offset in a specific logic frame is calculated as

$$Offset = \lfloor (K - Y - 1) \times 5 + 12 \rfloor \text{ (bytes)}$$

where K is the total slice row number and Y is the slice row number. Since the slice row number starts from '0' instead '1', a value of '1' is subtracted.

Inside the 5 bytes of each slice, the first 16 bits are for the G-LUT truth table and the last 16 bits are for the F-LUT truth table. The LUT contents are stored in 1's compliment format, i.e. bitstream values are the complement of the truth table entries. The G-LUT truth table contents are stored in sequence from left to right, as MSB to LSB, while the F-LUT contents are stored in reverse order from LSB to MSB. For each LUT, there are maximum 4 inputs with up to 16 truth table elements. When less than 4 inputs are utilized, which means that only 2, 4, or 8 entries in the truth table are used, the remaining unused entries are filled with the duplicated effective values of the used entries instead of arbitrary 0s or 1s. Figure 21 shows an example of the full bitstream mapping process.

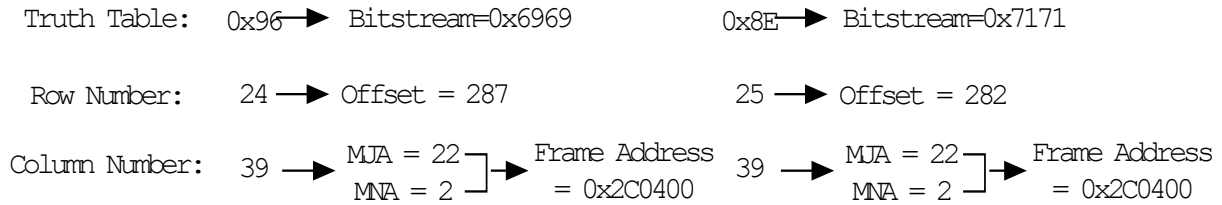


Figure 21: Bitstream Mapping Process

The configuration bitstream data in the full configuration file is organized consecutively frame by frame without labeling each frame address explicitly as partial reconfiguration file. Thus, if the logic in the full bitstream file needs to be modified, the location of each logic slice can also be deduced from:

$$Offset = \left\lfloor \frac{X}{2} \right\rfloor \times 22 \times 424 + 424 \times (X \bmod 2) + (79 - Y) \times 5 + 12 \Big] + (F_{GCLK} + F_{LIOB} + F_{LIOI} + 1) \times 424 + overhead$$

where X is the slice column number; Y is the slice row number; 22 is the total number of frames for each CLB column; 424 is the total number of bytes per frame; FGCLK, FLIOB and

FLIOI are the total frame number of the GCLK, left IOB and IOI column respectively, which can be easily get from the data sheet, e.g. for Virtex II Pro, it is always 30 frames in total. The additional value of '1' added in equation (9) reflects the fact that at the beginning of the configuration bitstream data, there is always an extra frame of '0' used to flush the reconfiguration register. Finally, the overhead of the bitstream file needs to be added on to it in order to obtain the final correct offset. These overheads may vary in length and can be counted manually or through a small API autonomously.

4.4 Chapter Summary

In order to achieve the maximum performance and system design flexibility for MRRA, three different mainstream reconfiguration interfaces have been investigated, including the SelectMAP, JTAG and ICAP in Chapter 4.1, 4.2, and 4.3. SelectMAP is an 8 bits width parallel configuration interface. The reconfiguration pins requires using the general I/O pin from the FPGA device. The interface is connected to the host PC through PCI interface. Since there are few commercial tools available for this interface, the reconfiguration and verification APIs were all built from the ground up. A bidirectional handshaking protocol was designed for this interface.

Chapter 4.2 presents the JTAG interface based system design. JTAG is a 4-pin serial interface. Instead of occupying the general IO pins as SelectMAP, JTAG using dedicated configuration pins only. Two verification strategies for JTAG based system are presented in this section. First is by using the commercial tool ChipScope, which provides a general GUI based solution for the system. The other one is a customized GNAT system dedicated to different user

logics. Compared to SelectMAP system, JTAG based system provides much more flexibility for partial reconfiguration module arrangement due to the pin occupation and the fixed module complexity.

Chapter 4.3 explains the ICAP interface, which is an internal reconfiguration interface for Xilinx Virtex FPGA family. The demands for runtime partial reconfiguration capability in embedded SOC applications are achieved by using this interface due to pin constraints, while combining the on-chip CPU core and Block memory. In order to fully utilize the internal Read-then-Write reconfiguration flow, a bit manipulation methodology was established. Detailed locating and logic transformation equations were demonstrated as well for the most popular partial reconfiguration capable FPGA.

CHAPTER 5: RESOURCE UTILIZATION AND TIMING ANALYSIS

Multiple system prototypes and verification strategies have been designed and developed based on different reconfiguration interfaces in Chapter 4. In this Chapter, in order to evaluate the performance of these systems, a group of test benches are applied to the prototypes. Results are quantitatively analyzed in details using metrics of hardware resource utilizations, power consumptions, translation, reconfiguration, and verification timing. Hashing algorithms are also used as basic application case study to test the ability of running general applications at Logic Layer on MRRA.

5.1 Basic Application Case Study

Hash algorithms [64], also known as message digest algorithms, are frequently used to generate a unique fixed-length bit vector H for an arbitrary-length message M . The vector H is called the hash or the message digest of M . These algorithms are used for encryption in a wide variety of security applications. Two types of the most commonly used hash algorithms, i.e., MD5 [64] and SHA-1 [64], are selected for the preliminary MRRA top-level application case study. Both algorithms are frequently employed in real-time embedded data stream processing applications.

The two algorithms have a very similar general structure. In both algorithms, 32-bit temporary registers are used to derive H . MD5 uses four registers: A , B , C and D . SHA-1 uses an additional fifth register E . These registers are initialized with certain fixed constants. The message M first is padded with '0' to a length of multiple 512 bits and then it is divided into

blocks of 512 bits. After that, each block is processed with several steps. Let i denote the step index. MD5 consists of 64 steps and SHA-1 includes 80 steps, i.e. $0 \leq i \leq 63$ for MD5 and $0 \leq i \leq 79$ for SHA-1. Each step includes a step function and the re-organization of the temporary registers. For each step there are two 32-bit words W and K . The word W is derived from current processing block based on a message schedule. The word K is a constant defined by i . There are four possible functions F . each is used in different round. The MD5 step function is defined as follows:

$$S_{md5} = B + ((A + F_{md5}(B, C, D) + W[g] + K[i]) \lll t)$$

where “ \lll ” means rotate left, g is decided by step index i and F_{md5} is:

$$F_{md5}(X, Y, Z) = \begin{cases} (X \wedge Y) \vee (\neg X \wedge Z) & 0 \leq i \leq 15 \\ (X \wedge Z) \vee (\neg Z \wedge Y) & 16 \leq i \leq 31 \\ X \otimes Y \otimes Z & 32 \leq i \leq 47 \\ Y \otimes (X \vee \neg Z) & 48 \leq i \leq 63 \end{cases}$$

After each step, the values of the registers are re-organized as $A \leftarrow D$, $B \leftarrow S_{md5}$, $C \leftarrow B$ and $D \leftarrow C$.

For the SHA-1 algorithm,

$$S_{sha} = (A \lll 5) + F_{sha}(B, C, D) + W[i] + K[i] + E$$

where if $i < 16$, $W[i]$ is the i th 32-bit word of the message block; if $i \geq 16$ $W[i]$ is calculated by

$$W[i] = (W[i - 3] \otimes W[i - 8] \otimes W[i - 14] \otimes W[i - 16]) \lll 1$$

and the F_{sha} is defined as:

$$F_{sha}(X, Y, Z) = \begin{cases} (X \wedge Y) \vee (\neg X \wedge Z) & 0 \leq i \leq 19 \\ X \otimes Y \otimes Z & 20 \leq i \leq 39 \\ (X \wedge Y) \vee (Z \wedge Y) \vee (Z \wedge X) & 40 \leq i \leq 59 \\ X \otimes Y \otimes Z & 60 \leq i \leq 79 \end{cases}$$

After each step, the values of the registers are re-organized as $A \leftarrow S_{sha}$, $B \leftarrow A$, $C \leftarrow B \lll 30$ and $D \leftarrow C$ and $E \leftarrow D$. When all the steps have been processed, the current values of the temporary registers are added to the values of the registers before the steps. Then, another block is selected for processing which continues until all blocks are finished. In the end, the hash value H of the message M is in the temporary registers, which is 128 bits of length for MD5 and 160 bits for SHA-1. For more detailed information about these two algorithms, see [64].

After analysis, the four step functions have been chosen to be implemented as reconfiguration modules for these two hash algorithms. As mentioned above, there are also many similarities between the MD5 and SHA-1 algorithms. Thus, it is possible for both algorithms to be implemented in a single top-level design so that the required resources are minimized with limited partial reconfiguration. More detailed discussion about combining these two algorithms can be found in [59], [63]. Clearly, the eight step functions in these two hash algorithms have the same type of inputs and outputs with identical bit widths. Therefore as discussed in Chapter 3, besides the Module Based partial reconfiguration flow, the Frame Based reconfiguration flow may offer a more efficient option in such cases.

Table 6 lists the results and compares the resource utilization and power consumption when using different implementation strategies. For each implemented algorithm, the first sub-column lists the result of the original full step function design as a baseline. The results from the Module-based partial reconfiguration flow implementation are listed in the second sub-column. As shown in this sub-column, the resource utilization for each module of the design has been

reduced to one third or less of the baseline design. As far as the power consumption is concerned, two groups of data are listed, including Dynamic Power and Total Core Power, where the latter is the sum of the Quiescent Power and the Dynamic Power consumption obtained by Xilinx XPower average over a test vector of over 211 random inputs. As shown in row 2 and row 3 of Table II, the Dynamic Power consumption has been reduced to just 8.8%, 15.4%, and 29.2% of baseline using the Module-based approach for SHA-1, MD-5, and combined circuits, respectively. The Total Core Power has been reduced to 57%, 58.3%, and 63.8% of baseline, respectively, as well.

Table 6: Step Function Resource Utilization And Power Evaluation

	SHA-1			MD5			Combined		
	Baseline	Module Based	Frame Based	Baseline	Module Based	Frame Based	Baseline	Module Based	Frame Based
Area (slice)	192	65 (33.9%)	32	881	168 (19.1%)	32	1068	324 (30.3%)	32
Dynamic Power (mW)	234.35	20.69 (8.8%)	N/A	255.20	39.32 (15.4%)	N/A	274.12	79.98 (29.18%)	N/A
Total Core Power (mW)	496.85	283.19 (57.0%)	N/A	517.70	301.82 (58.3%)	N/A	536.62	342.28 (63.8%)	N/A

As for the Frame-based design, since all step functions take 3 inputs and generate 1 output of 32 bits width each, only 32 LUTs are required to be updated during partial reconfiguration. Therefore a minimum of 16 and a maximum of 32 slices are needed based on the LUT placement strategy. The truth table representations are stored in the top-level flow control code directly with $2^8=16$ bytes storage consumed. The new bitstream is generated by the translation engine on request. The Xpower tools facilitate estimation of power consumption at the design level and the results reported are the average values across all slices in the design.

The power consumption of individual slices cannot be accurately estimated. Therefore the power data of Frame-Based design is not available.

5.2 Resource Optimization Analysis

The hardware prototype of the MRRA has been developed with a 2GHz Pentium 4 desktop host with 512M bytes of RAM, and on an Avnet Virtex II Pro development board equipped with a Xilinx Virtex II Pro VP7 FPGA. The Virtex-II Pro family provides abundant reconfigurable resources with strong functional elements for both combinatorial and synchronous logic. It also provides an embedded PPC405 cores inside the FPGA device, operating up to 400+ MHz with specially designed interface logic integrates the core with the surrounding CLBs, block RAMs, and general routing resources. In addition, the Avnet development also provides a PCI interface and large amount of on board memory, including SDRAM SODIMM, Mobile SDRAM, Asynchronous SRAM and Flash, which provide huge extension for large calculation and reconfigurations. All these features make it possible to build a MRRA system that meets our original design specifications.

The onboard hardware component and software APIs were initially developed using the Xilinx ISE 6.3 toolset and EDK 6.3, and later extended to support Xilinx ISE 9.1i. WinDriver from Jungo Software is also used to establish the communication APIs on the host PC side. The physical resource area management constraints are entered directly into User Constraint Files (.ucf) as text input. Mapping and routing are accomplished using the Xilinx toolsets. The 1D area management is implemented using the “area group” constraints and the slice-level 2D area

management is defined by using the “LOC” constrains. Details about the syntax of the .ucf file can be found in [76].

Five different partial reconfiguration platforms have been developed based on the different reconfiguration interface requirements and strategy, including one with SelectMAP interface only, one with JTAG and ChipScope only, One with JTAG and GNAT, one with ICAP only and external SRAM controller on chip, and one with all three interfaces. With these five prototypes, complexity and performance tradeoffs for embedded SOC applications can be clearly compared and contrasted.

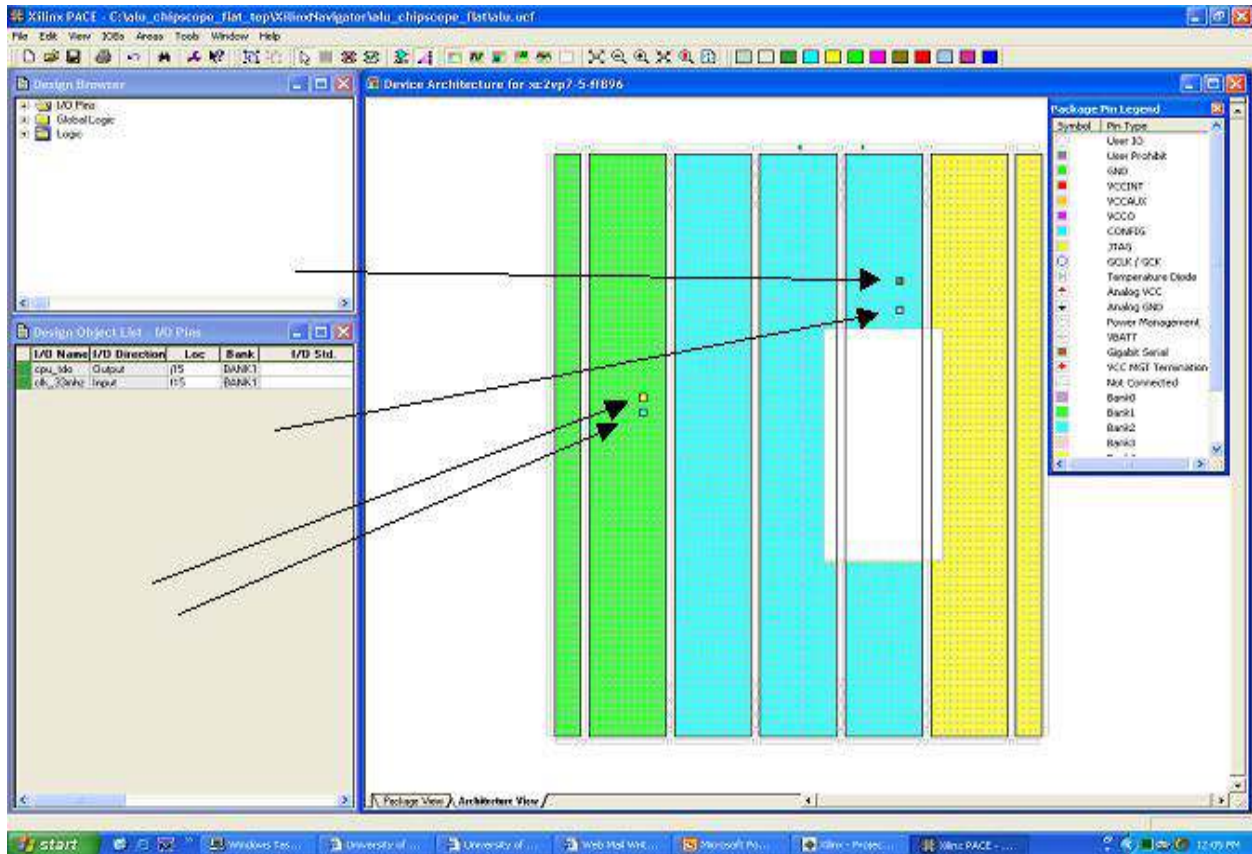


Figure 22: Bus Macro Placement [20]

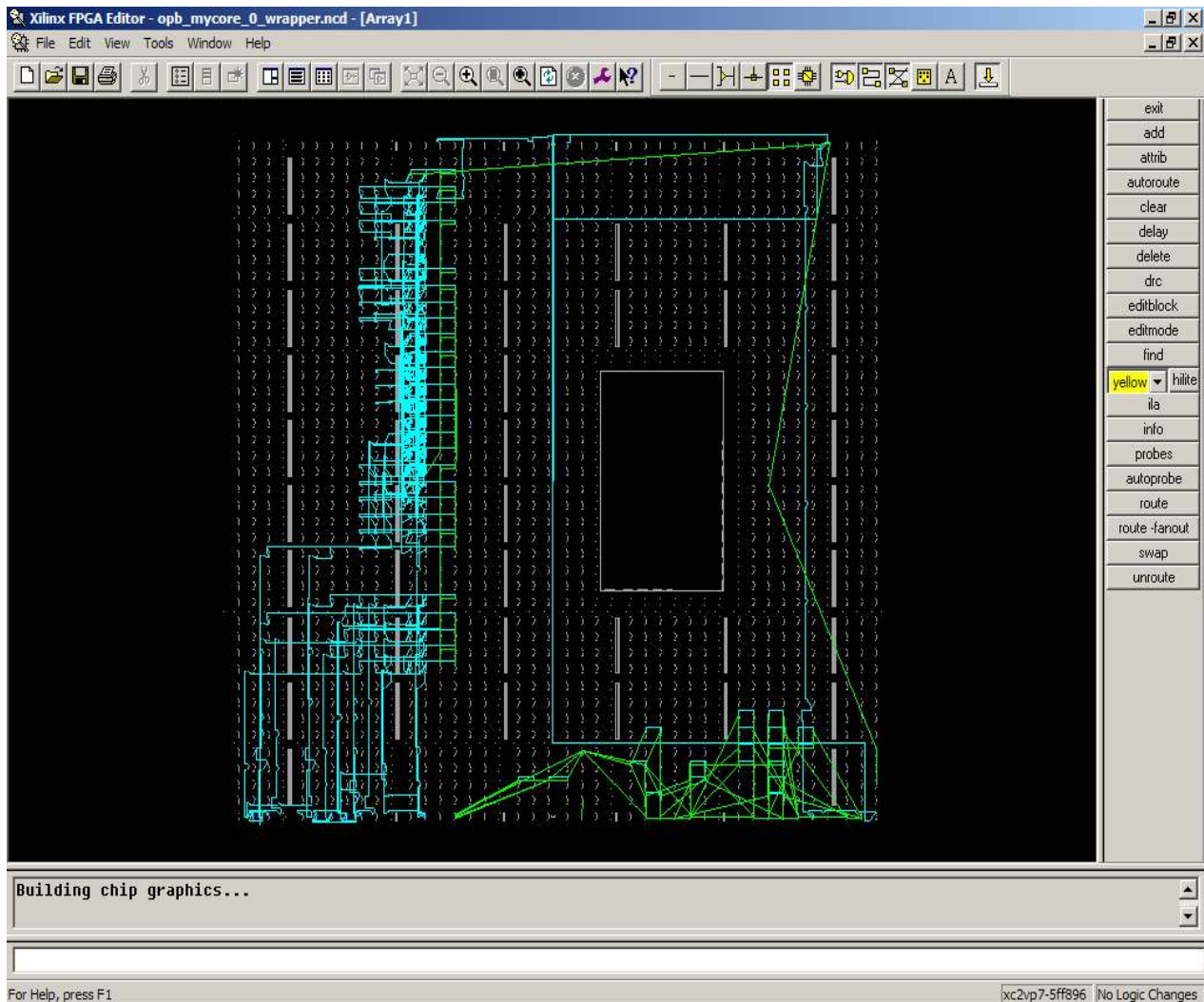


Figure 23: Partial Reconfiguration Module Routing

As mentioned in the previous chapter, to generate an MRRA partial reconfiguration system, bus macro has to be specifically placed between modules. Figure 22 shows a simple example from the developed system. The full FPGA device has been divided into 3 regions, which marked with different colors. Between these regions, bus macros have been placed right on the saddle of the boundary to generate correct connections. Figure 23 shows an example of a routed partial reconfiguration module after the bus macro placement. The module has been placed in the leftmost region. As shown in the figure, the routing of this module has been

specifically constrained inside the defined rectangle area. The inter-module signals are connected through bus macro, which are in green color in the figure. The only exception signal that can cross the boundary is the clock signals, which use the global clock routing net, instead of the local routing nets.

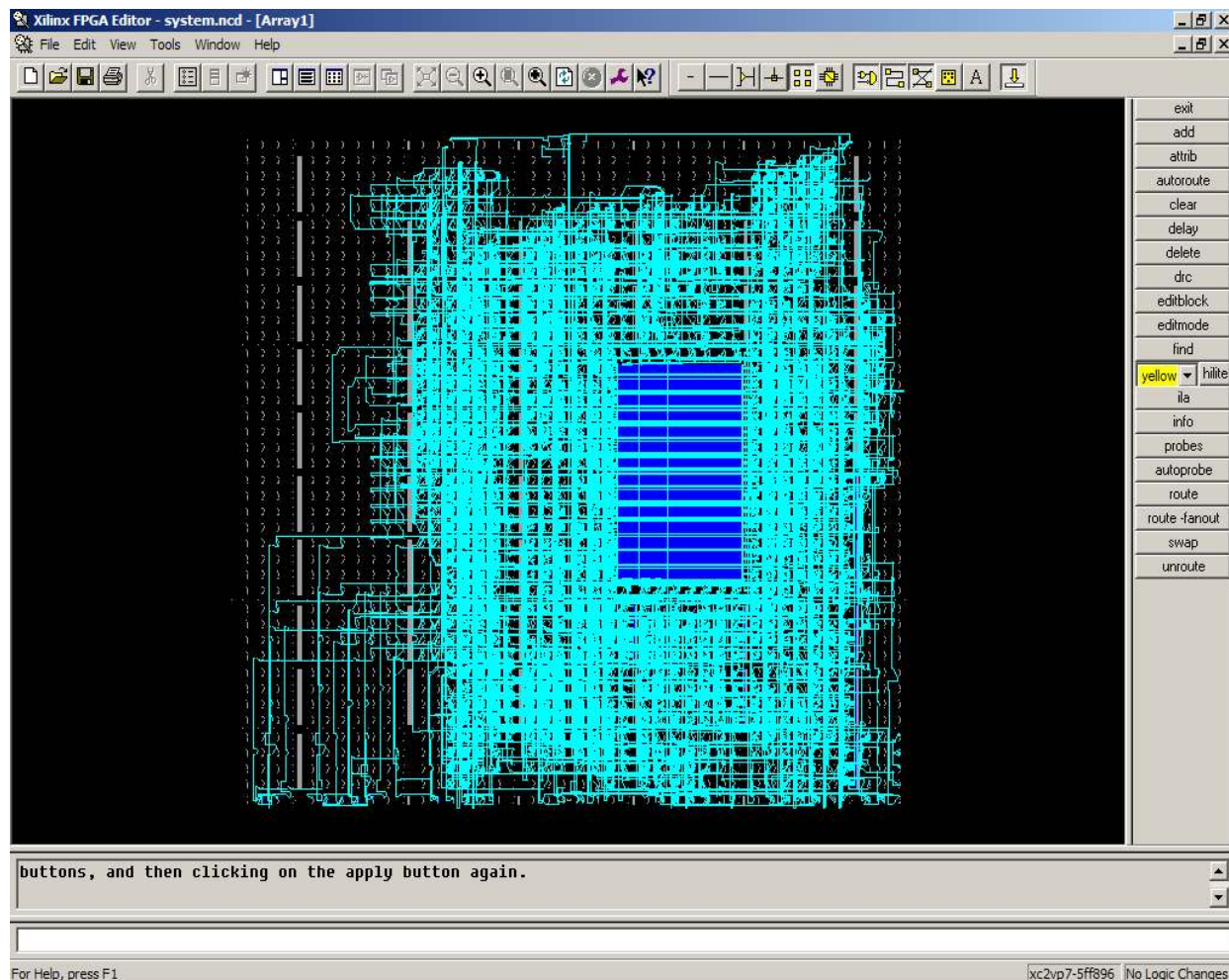


Figure 24: MRRA with SelectMAP Interface Placement and Routing

In order to establish the bi-directional communication channel, external SRAM and on-chip SRAM controller modules are used as data buffers for reconfiguration purposes, which occupy a majority of the 77 external pins and 352 TBUF resources in the fixed region. As

mentioned in the previous chapter, with the PowerPC core involved, 3-segment communication bus with 32 to 64 bits width is required. Therefore, as shown in Figure 24, the routing nets have covered most of the areas even they are limited to their own designated area. In fact, because of the high pin usage dispersed across the fixed region and the complicated communication modules, only 15 out of 68 columns of slices remain available for the reconfigurable modules. Therefore the resource utilization of the SelectMAP prototype, the ICAP with external SRAM prototype, and the multiple interface prototypes show very little variation.

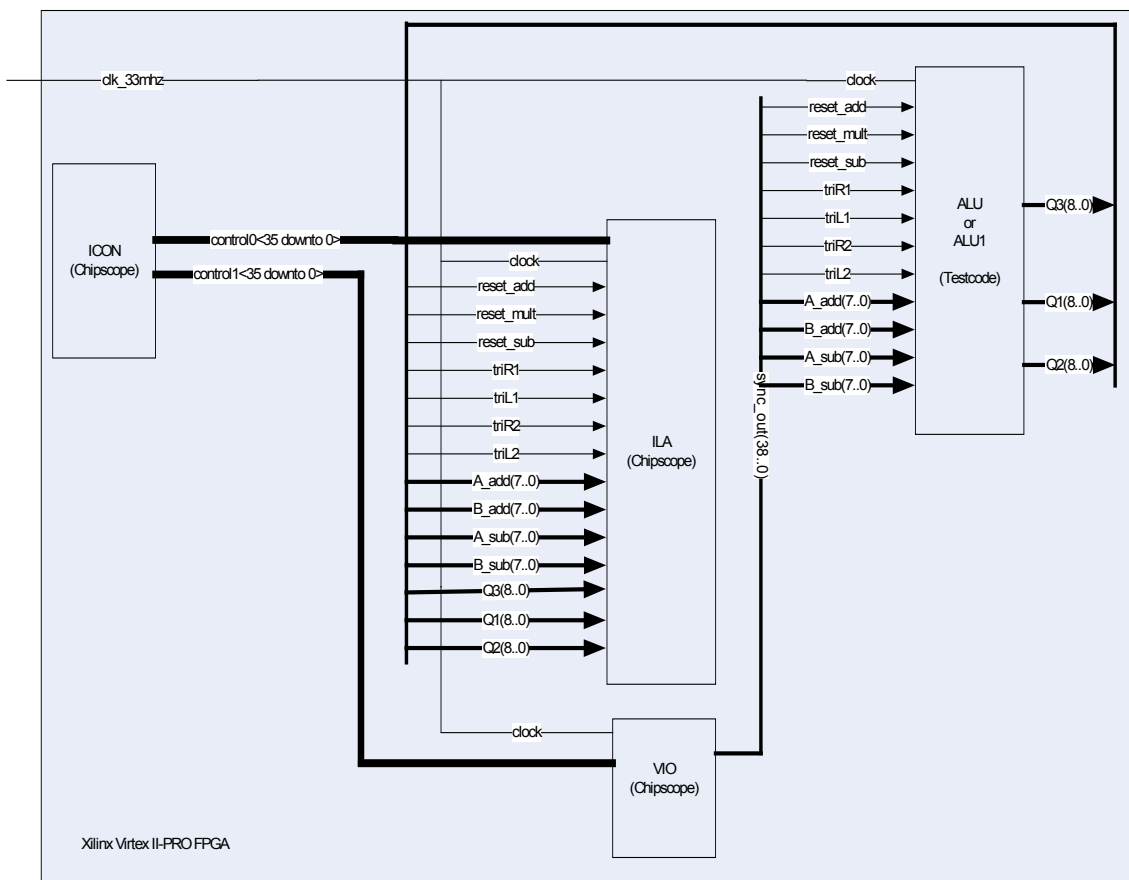


Figure 25: Block Diagram of ChipScope Cores and Associated Interconnection

On the other hand, if the ICAP system is built as an SOC prototype, with no required external parallel communication channel, or using only the standard RS232 port, most of the pin

utilization drops significantly. Furthermore, if only a JTAG interface is used, commercial tools such as ChipScope can be used for validation process, which can eliminate most of the on-chip modules in the fixed region, including the PowePC core. Without the OPB bus interconnection, the IPIF for each module is also not required. Hence the reconfigurable module overhead is reduced from 7 slices to 0, when compared to other platform versions. Figure 25 demonstrates the logic diagram of the instantiate MRRA hardware platform. Figure 26 shows the corresponding physical placement and routing. Clearly, compared to the SelectMAP based platform, such scheme provides much less area and routing occupancy. Hence it provides bigger flexibility for the partial reconfiguration modules.

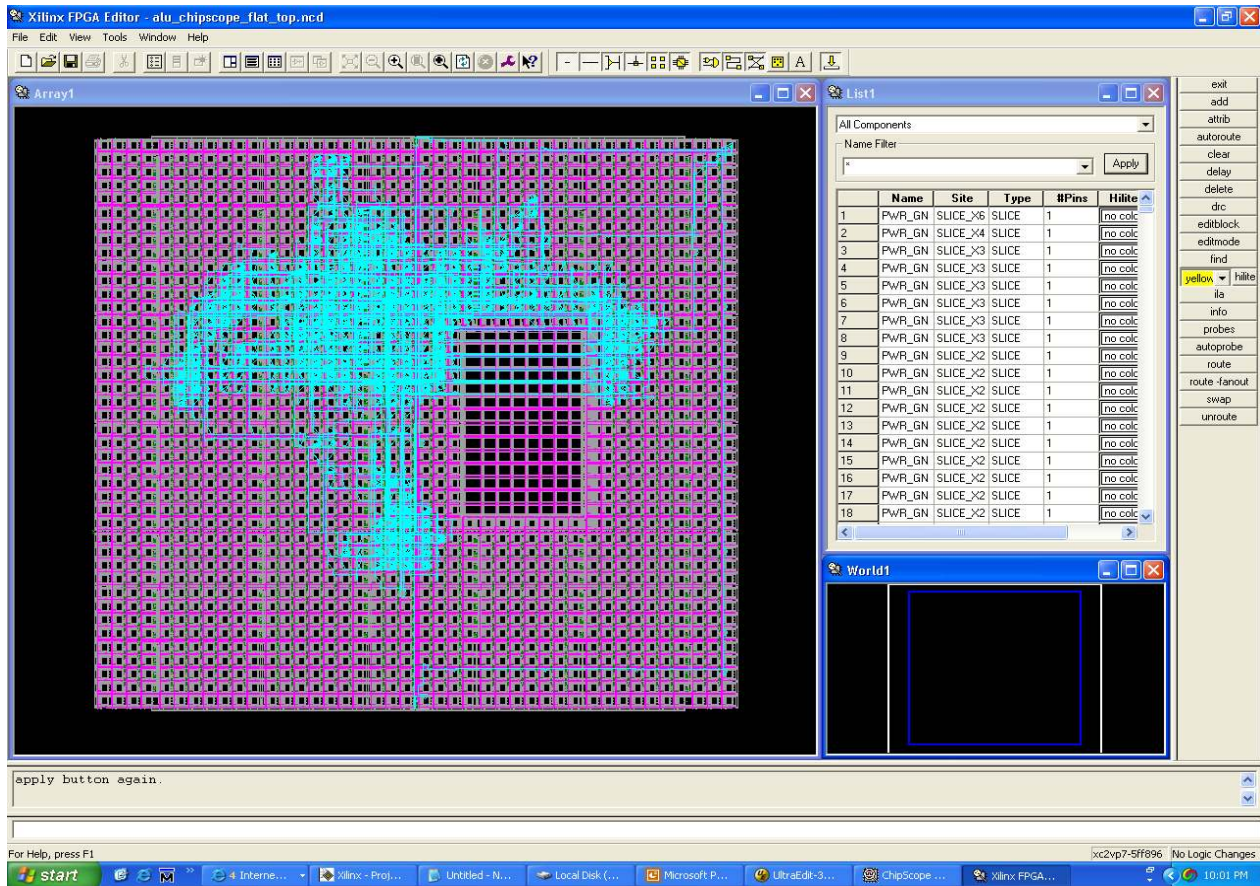


Figure 26: Placement and Routing of ChipScope Cores and Associated Interconnection [20]

Figure 27 shows the implementation of GNAT system. Compared to other platform, this design shows significant advantages in the resource utilizations and routing occupancy. Expandable bus system gives the flexible control on the bit width of the modules and routing arrangement. On the other hand, since the BSCAN module is a dedicate circuit sits at the upper right corner of the device, the gnat needs to be placed on the right most partitioned regions. This may require customized bus macro involved to make correct communications if multiple partial reconfiguration modules need to be connected to the fixed region.

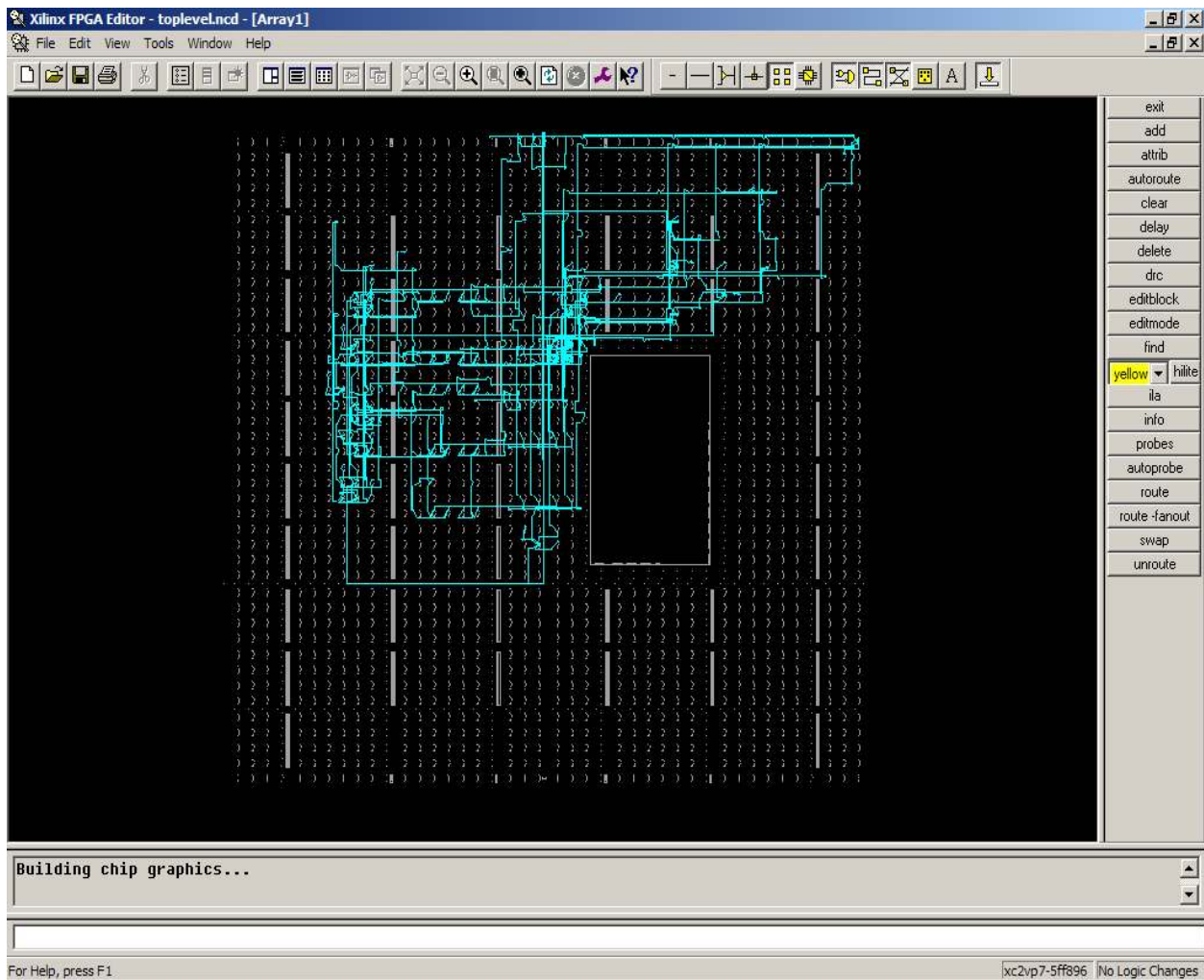


Figure 27: GNAT Placement and Routing [52]

Table 7: Resource Utilization

Interface	# of Fixed Modules	# of Pin of Fixed Modules	Reconfigurable module overhead	Slices for Fixed Modules	BRAM for Fixed Modules	TBUF for Fixed Modules	PPC405
Comprehensive interface	9	77	7 slices	1472	9	352	Y
SelectMAP	8	77	7 slices	1352	8	352	Y
ICAP with SRAM controller	9	77	7 slices	1472	9	352	Y
ICAP without SRAM controller	8	4	7 slices	932	9	42	Y
JTAG with ChipScope	4	25	0	73	0	64	N
GNAT	1	0	5	36	0	0	N

Table 7 shows the detailed comparison of all the MRRA prototype system. In summary, when the SelectMAP interface with external SRAM is used to establish reconfiguration and testing channels, sophisticated hardware logic is involved and excessive pins usage is incurred, which consumes a factor of 5 to 18 times more logic resources in the fabric than the JTAG-based prototype. These costs can limit the size and area placement flexibility of the reconfigurable modules. In this case, large capacity FPGAs, such as the Virtex-II Pro X2VP20 or above, are highly recommended. Furthermore, additional effort is also required for pin assignments and connections with special bus macros thereby resulting in an increase in design complexity.

5.3 Timing Analysis

5.3.1 Fundamental Timing Parameters

Tests have been carried out to measure the performance of the fundamental operations of the MRRA prototype. Table 8 presents the wall clock time for each of these operations. For the JTAG and SelectMAP interfaces, the time taken to download the full bit file, which has a fixed size of 548 KB for the Virtex II Pro XC2VP7, is measured. The time taken by the ICAP reading and writing operations are measured on a per frame basis, each of which contains 424 bytes of data.

Theoretically, the JTAG interface with a parallel cable III can have a download speed of 300Kbps [74] and SelectMap with Virtex II/-Pro can work at a maximum 66MHz clock speed [73]. In our prototype, the measured data transfer rate using JTAG was 216 Kbps. Due to the data-transferring overhead between the host PC and the board, the SelectMap interface requires 536 ms, which roughly translates to 1MB/s throughput. Therefore, when downloading from the host PC, the observed reconfiguration latency of JTAG is 40 times higher than that of the SelectMAP interface, as expected. This indicates the magnitude of benefit achievable by using SelectMAP in terms of low reconfiguration latency.

For the on-chip reconfiguration operation, the ICAP-based technique took 303,425 clock cycles to read a frame and 304,811 cycles to write a frame. Since the current PowerPC core operates at 100MHz, the timing can be easily transformed into milliseconds as listed in Table 8. A single data communication processing cycle, starting from host PC sending the data out to the PCI to reading the data back from the SRAM requires up to 123 ms. This includes the time take

by the hardware and PowerPC to finish processing the data. The time taken to generate a new bitstream file with direct bitstream logic manipulation APIs has an upper-limit of 30 ms for modification of the 32 slices of the hash algorithms on the host PC.

Table 8: Basic Timing Evaluation

Operation	Theoretical Maximum Throughput	Measured Throughput	Measured Transfer Time
SelecMAP Reconfiguration	66 MB/s	1MB/s	536ms (per full file)
JTAG Reconfiguration	300 Kbps	216 Kbps	20.3s (per full file)
ICAP Read	50 Mbps	1.12 Mbps	3.03ms (per frame)
ICAP write	50 Mbps	1.11 Mbps	3.04ms (per frame)
Data Communication	N/A	N/A	123 ms
Direct Bitstream Manipulation	N/A	N/A	30 ms (per 32 slices)

5.3.2 Translation Engine Evaluation

In the Translation Layer, both the 1D and 2D area management mapping process still relies on the Xilinx toolsets. These physical resource area management constraints are created and modified directly by upper layer logic and then translated into standard text-based constraint input by the translation engine in this layer. After the new constraint file is generated, the Xilinx toolsets are invoked as the other part of translation engine via shell scripts. This runs the task in the background automatically to perform the placement and routing for the module without manual input.

The speed of the translation engine for module implementation is also evaluated with a series of circuit with different size. The translation engine is required original partial reconfiguration bitstream and to reallocate the physical resources. In addition to the original

MD5 module, two combinational benchmark circuits from the ISCAS'85 benchmark suite - the C17 and the C1908, which is the SECDEC circuit mentioned in the previous section have been used in this experiment. Two sequential benchmark circuits B02 and B03 from the ISCAS'99 benchmark are also used to gauge performance with sequential circuits.

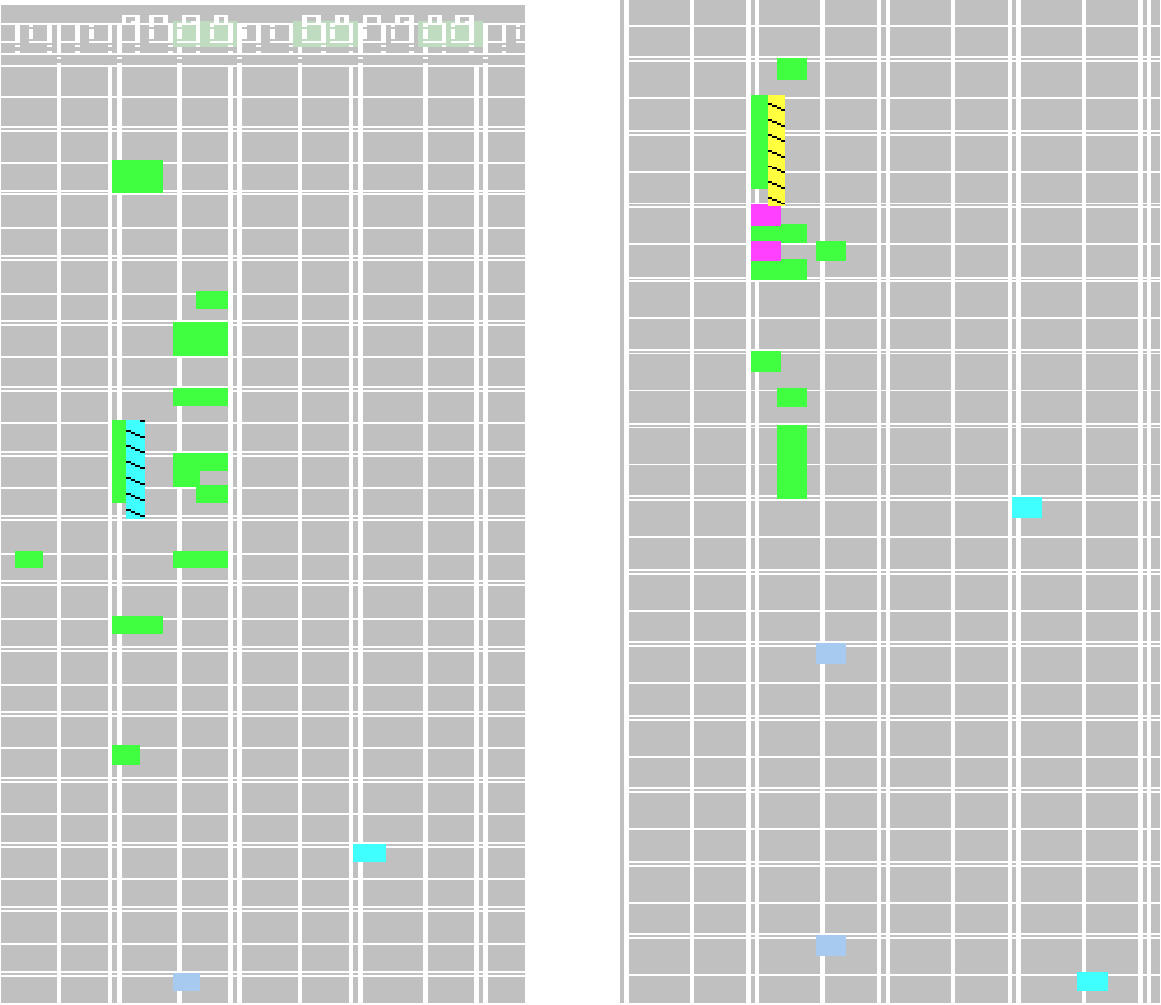


Figure 28: B02 and C17 Mapping and Placement

Figure 28 and Figure 29 shows the resource placement and mapping of the 4 test bench circuits. As shown in the figures, the B02 and C17 have similar small resource utilization and the

B02 and C1908 have similar middle resource utilization. This size variations and similarity actually also reflect in their translation timing evaluations, which shows that resource utilization can be a direct metric for the translation timing estimations.

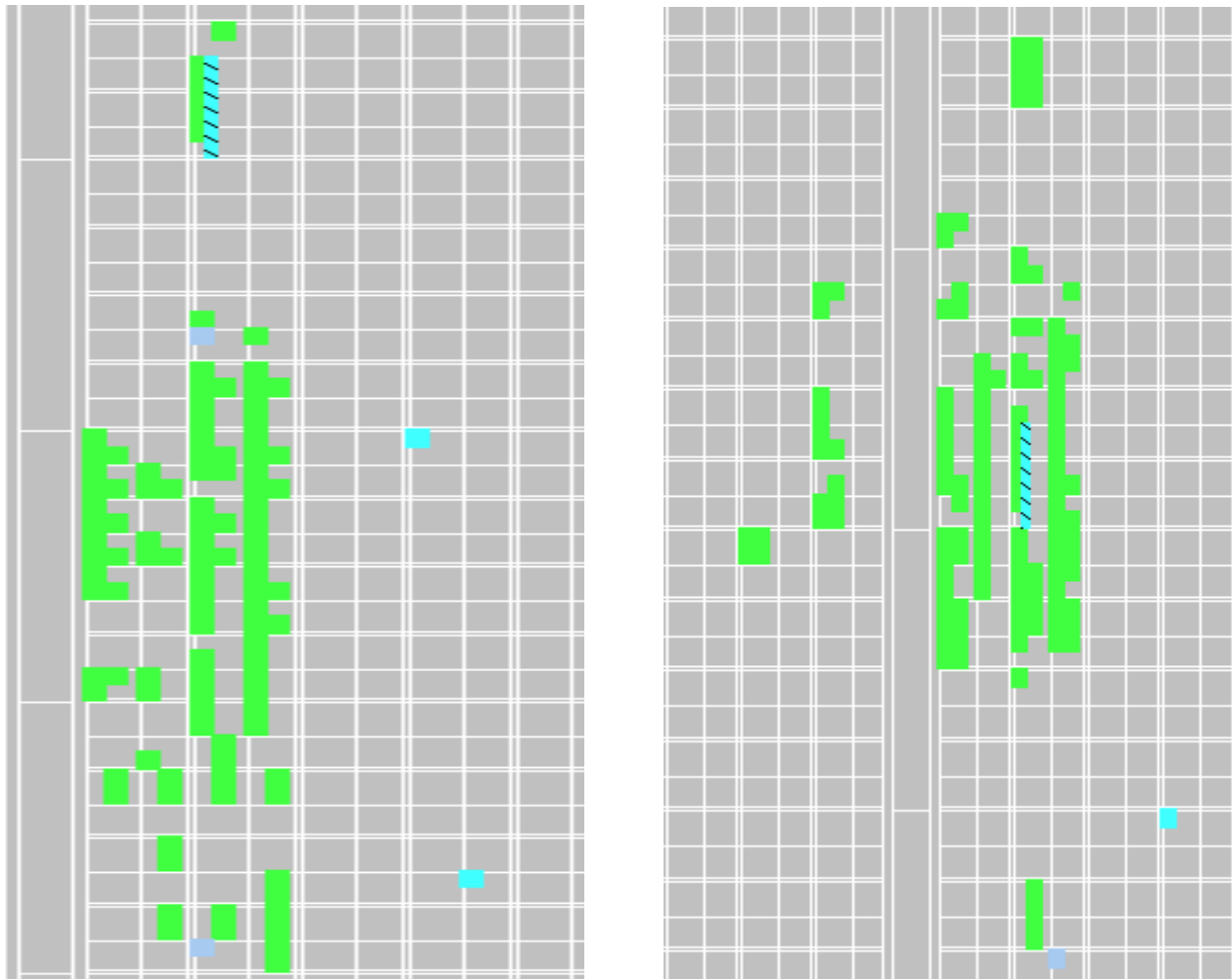


Figure 29: B03 and C1908 Mapping and Placement

Table 9 lists the detailed timing results. The first row of the table lists results for the full MRRA prototype used as a baseline for comparison. Among the 5 benchmark circuits, C17 and C1908 which are combinational designs were described at the gate level. B02, B03, and MD5

were developed at Register Transfer Level instead. In Table 9, the Original Equivalent Gate column lists the number of gate-equivalents reported when these benchmarks were instantiated directly using the Xilinx design tools. However to incorporate these circuits into the MRRA framework, a standard IPIF, has to be added above the standard logic to maintain the correct data communication to the OPB bus. This IPIF logic increases the size of the partial reconfigurable modules, which can be observed from the corresponding Occupied Slices column in Table 9.

The partial bitstream file size adequately shows the result of these slice utilization differences, which are demonstrated in the fourth column. The last column lists the translation time for partial reconfiguration module implementations of the benchmarks. For these module implementations, the time is partially dependent on their size, although not linearly related. Although a significant improvement in the total time taken by the process has been achieved when compared to the full configuration bitstream generation, they still require tens of seconds. The partial reconfiguration modules have also been evaluated by integrating both ISE 6.3 and the latest version ISE 9.1 within MRRA. The timing results for these two versions of the ISE are shown in the last two columns of Table 9. These figures were obtained on a Windows XP environment with a 2.0 GHZ Pentium CPU and 512 MB RAM. Clearly, the ISE 9.1 version runs much slower than the ISE 6.3. Upgrading the development hardware will definitely improve the performance significantly, however, the translation time will remain greater than ten seconds. Therefore, it is not highly recommended to call these scripts at runtime unless it is essential for relocating the modules. Alternatively, the scripts can be pipelined with other running tasks efficiently as described below.

Table 9: Translation Engine Evaluation

Test Circuit	Oringal Equivalent Gate	Occupied Slices	Bit file Size (Byte)	Genera-tion Time (V6.2)	Generation Time (V9.1)
MRRA	N/A	1472	548 K	4m 31s	N/A
C17	6	8	66 K	67s	101s
C1908	603	41	89 K	69s	109s
B02	28	11	66 K	66s	107s
B03	160	45	75 K	70s	163s
MD5	2496	168	120K	71s	111s

5.3.3 Timing Analysis

For each reconfiguration task with a conventional serial flow, the total latency L_i is determined as:

$$L_i = T_{TAT}(i) + T_{DRT}(i)$$

where T_{TAT} is the Task Arrival Time and T_{DRT} is the Dynamic Reconfiguration Time.

Within the MRRA framework, pipelining can be easily adopted to overlap the execution time of different layers and components for possible speed-up if multiple tasks are running on the platform. Clearly, since this is a CPU/FPGA hybrid design, at least a two-level pipeline can be involved to overlap the Task Arrival Time and the Dynamic Reconfiguration Time. Therefore the latency now can be calculated as:

$$L_i = \max(T_{TAT}(i), T_{DRT}(i))$$

The Task Arrival Time is comprised of two parts:

$$T_{TAT}(i) = T_E(i-1) + T_A(i) + T_G(i)$$

where $T_E(i-1)$ is the previous task evaluation or running time for routing data process, $T_A(i)$ is the time delay for the top layer application to generate the new reconfiguration task request, and $T_G(i)$ is the new hardware module generation time.

As listed in Table 8, one cycle of data communication is completed at the sub-second level, which means a full task evaluation $T_E(i-1)$ may consume a few seconds or more, depending on the quantity of the routine data and the complexity of the data processing algorithm running on the host PC. The term $T_A(i)$ simply depends on the complexity of the top-level user logic. Using a typical example of autonomous reconfiguration such as evolvable hardware, with population-based Genetic Algorithm strategy such as [2], [29], the program will produce each new generation of circuits leading to a new reconfiguration task request at the sub-second level. As for $T_G(i)$, based on Table 8 and Table 9, this parameter is determined by which part of translation engine is used. If a precompiled stored module or only bit manipulation is used, the time required to generate or load the modules will be in micro to milliseconds level. However, when the Xilinx tool scripts are involved, the time scale will increase to the level of tens of seconds. Therefore, the Task Arrive Time $T_{TAT}(i)$ can vary from the best case scenario of sub-second level to the level of tens of seconds in the worst case.

On the other hand, for the Dynamic Reconfiguration Time $TDRT(i)$, using different interfaces leads to different delays. For instance, the SelectMAP interface is fast enough even for the full device configuration transfer to remain at a sub-second level. For the JTAG interface, tens of seconds is unavoidable. As for ICAP, the time is given by:

$$T_{DRT}(i) = (T_W + T_R) \times F, \quad (5)$$

where T_W and T_R are the frame write and read time respectively, and F is the total number of frames that need to be modified, which makes $T_{DRT}(i)$ for ICAP vary from milliseconds to possibly seconds.

Based on the above analysis, both $T_{TAT}(i)$ may equal $T_{DRT}(i)$ at the sub-second level or at the tens of seconds level. $T_{TAT}(i)$ is actually dominated by the factor $T_G(i)$, which leads to the order of sub-seconds or tens of seconds depending on the size of the reconfiguration module and the MRRA strategies selected for bitstream generation. Using the Xilinx scripts shows significant improvement compared to generating the full bit file, yet this remains a bottleneck for large runtime task reconfiguration. $T_{DRT}(i)$ is simply related to the reconfiguration interface that is chosen. Although JTAG shows great resource utilization reduction and design flexibility, the timing bottleneck that it creates is also significantly large compared to SelectMAP or ICAP interfaces. Thus, design strategies need to be carefully established to deal with these delays to obtain the best runtime efficiency and reconfiguration flexibility.

5.4 Chapter Summary

This chapter is organized into four sections. In Chapter 5.1, the resource utilization and power consumption were analyzed using two Hashing algorithms as a top-level application case study. The experiments on the MD5/SHA-1 hash algorithms compared the two metrics for the different design flow. Results show significant improvement when using the MRRA partial reconfiguration control flow, especially the MRRA Frame-based flow.

Chapter 5.2 demonstrates the detailed FPGA resource utilization for each MRRA platform with different reconfiguration interface and testing methodology. Prototypes for the

different reconfiguration interfaces based the MRRA concept were developed on two standalone COTS hardware platforms: Avnet Virtex II Pro development board and a Pentium PC with Xilinx ISE and EDK FPGA CAD software suites. Adoption of several standard industry hardware interfaces, IPIF, PCI, etc., along with the design and refinement of appropriate communication and synchronization protocols provides a powerful and useful abstraction technique. Resource utilization estimations have been carried out on these prototype platforms. Overall, when the SelectMAP or ICAP interface with external SRAM is used to establish reconfiguration and testing channels, sophisticated hardware logic is involved and excessive pins usage is incurred, which consumes a factor of 5 to 18 times more logic resources in the fabric than the JTAG-based prototype. These costs can limit the size and area placement flexibility of the reconfigurable modules. As a result, less physical rectangle areas will be available for the partial reconfiguration modules. Hence, this implies reduced flexibility is for the control logics. In this case, larger capacity FPGAs with extra external pins, are highly recommended. Furthermore, additional effort is also required for pin assignments and connections with special bus macros thereby resulting in an increase in design complexity.

Chapter 5.3 evaluates the timing performance of the MRRA system, including the basic reconfiguration and testing time and the Translation Engine overhead. A number of benchmark and hashing algorithm case studies demonstrate the range timing variations of autonomous and dynamic reconfiguration operations. In contrast of the area arrangement flexibility, the SelectMAP and ICAP interface shows much higher speed when transferring the reconfiguration and testing data than JTAG. The result also illustrates that when physical re-routing by Xilinx tool engine is involved, the speed of this translation process becomes the bottleneck of the system performance, even though it has already shown significant improvement comparing to

the baseline timing performance. Experiments also prove that the integrated design flow is able to retain and demonstrate upward compatibility with vendor toolsets.

CHAPTER 6: RECONFIGURATION OPTIMIZATION TECHNIQUES

The timing analysis in Chapter 5 shows that the reconfiguration time latency can be as high as tens of seconds, which is clearly a major bottleneck for a practical autonomous partial reconfiguration process. One of the major reasons of this large delay is due to the Xilinx routing translation engine for the physical re-routing request from the higher level. In order to alleviate this bottleneck and provide a faster reconfiguration control paradigm, extra optimization techniques and analysis are developed in this Chapter.

6.1 Area and Bitstream Optimization

All reconfiguration interfaces have certain speed limit. In previous chapter, JTAG based MRRA has shown the biggest flexibility for the partial reconfiguration design. However, on the other hand, similar to basic serial port, JTAG interface use only one pin for input and one for output, which provides a maximum 400Kb/sec data throughput, not considering other non-trivial logic control delay when doing partial reconfiguration. This is a much more limited data transferring speed compared to other parallel interfaces, such as ICAP. On the other hand, ICAP interface is normally used in a System-on-Chip architecture, which provides only limited storage space for partial reconfiguration files. Therefore, reducing the reconfiguration overhead, including both the reconfiguration time and the reconfiguration data storage space, are important concerns in this research area. Foremost, these two reconfiguration cost are directly related to

the size of the reconfiguration bitstream file. Minimize the bitstream file size can potentially improve the performance significantly.

For the Xilinx Virtex II/-Pro family, there are several configuration column types, including Global Clock (GCLK), Input Output Block (IOB), Input Output Interconnect (IOI), Configuration Logic Block (CLB), Block RAM (BRAM), and BlockRAM Interconnect (BMINT). Each type has a given number of frames, as described in Figure 30, where each configuration frame has a unique 32-bit address that is composed of a Block Address (BA), a Major Address (MJA), a Minor Address (MNA), and a byte number [70]. The major address identifies a specific column within a block, and the minor address identifies a specific frame within a column.

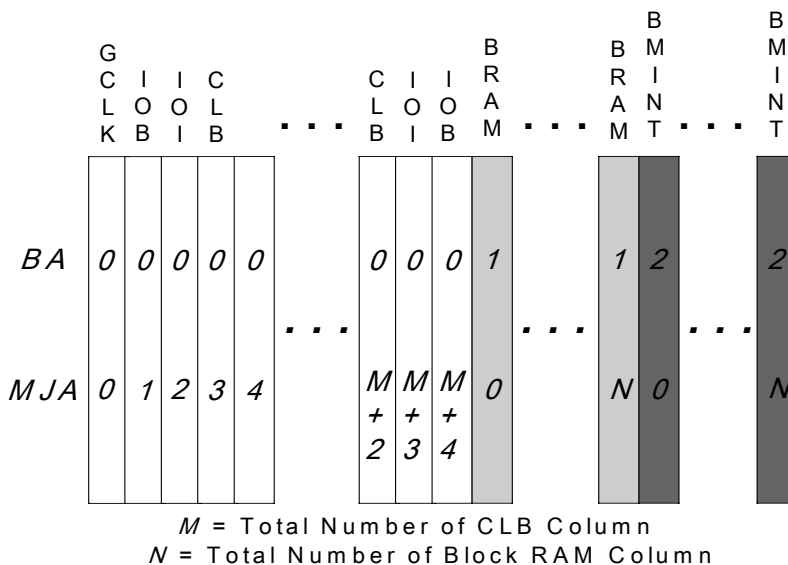


Figure 30: Column Level Configuration Memory Map

Among all these types of columns, the CLB columns control the configurable logic blocks, routing, and most interconnect resources. IOBs on the top and bottom edges of the

device are also programmed by CLB configuration columns. The number of CLB configuration columns matches the number of physical CLB columns in the device.

For each CLB column, there are two columns of slices. To denote the configuration of these slices, 22 frames are utilized within the bitstream for a complete reconfiguration file. Each frame has a fixed size of 424 bytes. By comparing the bit files for a series of test circuits, it is possible to determine that the logic for each CLB column, which is stored in the two LUTs of each slice, actually only occupies two of the 22 frames. In particular, the contents for the first slice column LUTs, i.e. with an even slice column number starting from '0', can be found in the second frame, while those for the second slice column, i.e. with an odd slice column number starting from '1', are in the third frame. IOB usage at the top and bottom edges of this CLB column are located in the first frame. The remainder of the frames are all used to describe the routing resources usage of the CLB column.

For unused CLB frames, a compression technique is used in the partial reconfiguration bitstream file. Instead of writing 106 instances of the word value of '0', which is a full frame length, the Multiple Frame Write Register (MFWR) is employed. This involves setting the corresponding frame address to the FAR first, and then writing two padding words to the MFWR (normally '0'). Using this padding technique, the full-unused frame can be set with a total cost of just ten bytes in the bit file. Therefore, for each unused frame, the number of saved bytes is 414, yielding 97% area savings per frame.

More generally, since configuration frames are arranged vertically, designs that span the fewest possible configuration frames achieve greater compression. To estimate the compression achieved, let the number of unused frames be denoted by U on a system that uses B bits per

frame. An estimate of the number of saved configuration bits, S , under a fixed region F per frame is given by:

$$S \approx U \times (B - F).$$

Here $B \gg F$ so S is nearly the product of U and B . Therefore, this 2-dimensional area management strategy inside modules can achieve high compression rates to minimize the partial reconfiguration data file size, which may be crucial for embedded applications using dynamic reconfiguration. Embedded SOCs often have limited storage capacities and real-time transfer timing requirements, and therefore can benefit from this bitstream compression strategy.

As suggested in the previous Chapter, inside each module, the 2D area management strategy can be incorporated into the Design-Time Flow to minimize the partial reconfiguration file size. This additional area management strategy needs to be carried out after the synthesis process of the design is complete and before the translation, mapping, placing and routing steps. Since this strategy deals with the real physical resource arrangement, the logic elements are identified at a very fine granularity, such as Slices, LUTs and D-flip flops, etc., which the Translation Layer can then directly translate and map. The steps involved in this procedure include:

1. **Region Allocation:** Assign an area for the partial reconfiguration module, which is large enough to accommodate all the external input and output signals at either the top or the bottom edge of the designated area. With an FPGA model such as the Virtex II Pro VP7 or higher, an area with 40 pins or higher along the edge can be easily partitioned, which normally will be able to satisfy an 8-bit or even 16-bit module design.

2. **Pin Assignment:** Choose either the top or the bottom edge and place all the external signals adjacent to each other if possible. When the assigned area contains the left or

the right edge of the device, these edges may be chosen as well. Place the remainder of the pins on the other side of the edge if any unoccupied pins are available. This step tries to eliminate, or at least minimize, any unnecessary signals that will span the full height of the device, which clearly will occupy more routing resources in different frames.

3. **Column Alignment:** Attempt to place all the logic elements into a single slice column consecutively or with only a short slice row gap, near the edge where the external pins were placed. One and only one frame will be used to describe all the LUT logic contents of a full column of slices, regardless of the number of LUTs of the slice column actually used, as long as it is not zero. Thus this step will minimize the number of frames used to describe the design logic as well as most of the interconnection resources.

4. **Choke-Point Elimination:** If there are any logic elements with a fan out greater than 4, place the destination elements around its side, including top and bottom of the same slice column as well as the adjacent slice column side-by-side. This will typically reduce the routing resource usage even more than simply by mandatory placing of all logic elements inside a single slice column.

5. **Repeat:** If there are any elements left to be processed after finishing one column, repeat steps 3 and 4. Place the rest of the logic elements into the adjacent slice column with the same principles until all or at least elements along major logic paths are completed. With an FPGA model as Virtex II Pro VP7 or higher, each slice column contains 160 or more 16-bit LUTs and the same amount of D flip-flops, which normally will be able to contain a small to middle size module design in one or two columns.

To summarize, the procedure places the logic elements into the least possible number of slice columns. The logic sequence of the elements may also need to be considered when placing along the path to achieve the highest possible optimization.

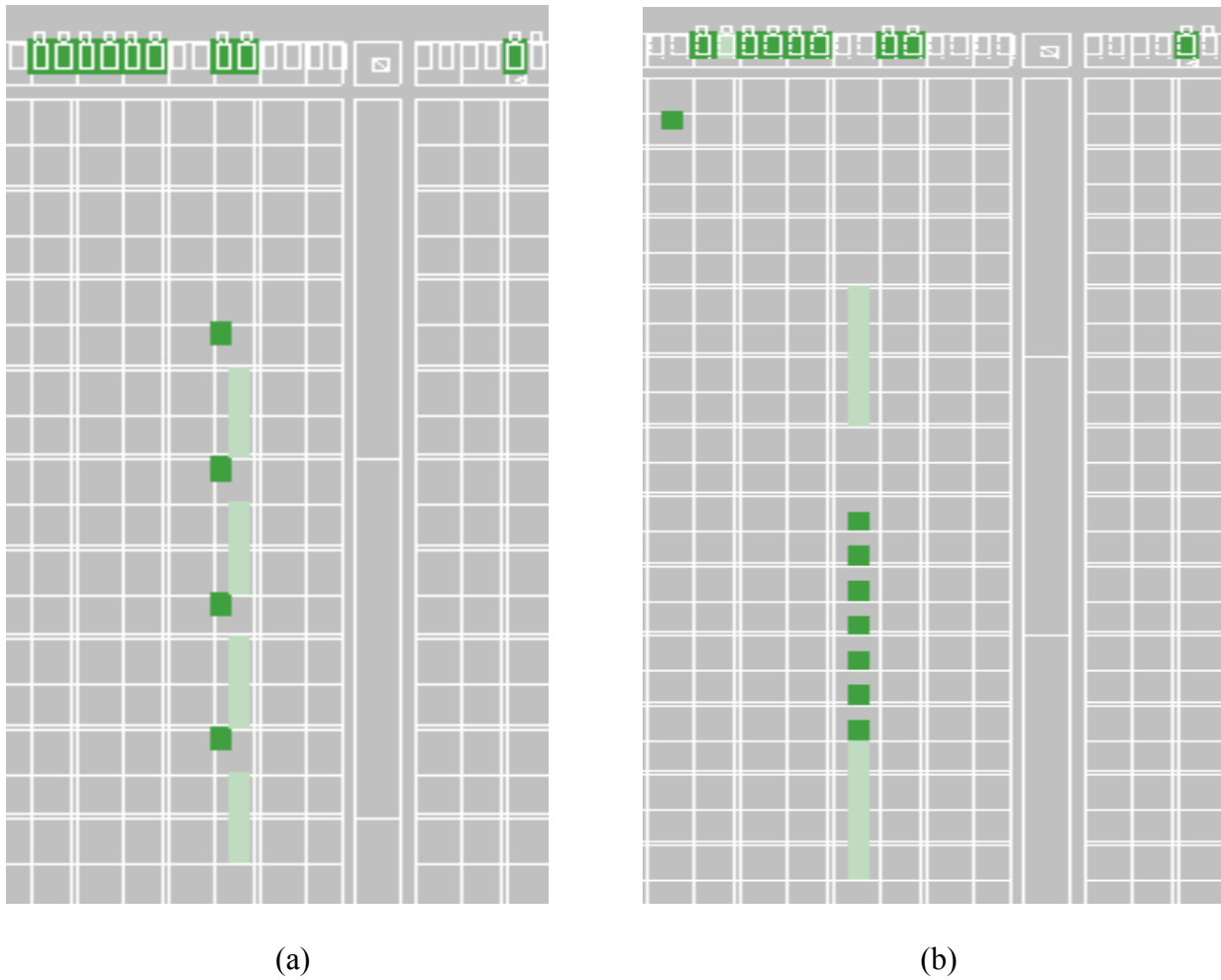


Figure 31: Optimized Design Layout for Case 1 and 2

To evaluate the area optimization strategy, several case studies have been carried out. Since MD5 and SHA-1 have the same dataflow structure, MD5 results are presented to demonstrate a larger design but SHA-1 is similar. Other case studies include four representative small cases, which illustrate all of the steps and scenarios mentioned above, and one middle-sized case study.

Each design was implemented as partial reconfiguration modules as listed in Table 10. Each of the four small cases has its own distinct features including parallel and cascaded LUT arrangements, dedicated physical resource usage and large fan out elements. The first design is a simple quad 4-input 16-bit LUTs design with a random combinational logic functions specified in the truth table. The second design is a 9-bit shifter with cascaded logic. The third design is a 4-bit×4-bit multiplier with a block multiplier used during synthesis. And the last is again a 4-bit×4-bit multiplier, but with LUT logic only. To increase the accuracy of the comparison, all 4 modules have been defined using the same number of external signals. All these signals have been managed to be placed at the top edge of the partial reconfiguration region.

Figure 31 shows the optimized logic element arrangement of first 2 small designs using MRRA. For the elementary 4 LUT element design in Case #1, since all LUTs are in the parallel logic path with direct input from external signals and connected to the output through flip flops, putting them in a single column close to the external pins is a straightforward solution. The resource arrangement is shown in left Figure 31(a). Case #2 for the shifter is shown in Figure 31(b), since all logic elements are logically serially cascaded, from input to output, the simple single column solution is again the best choice.

However, for Case #3 the 4-bit×4-bit multiplier uses the dedicated hardware block multiplier resource, which is circled in red in Figure 32(b). The original placement arranged by the tool automatically is shown in Figure 32(a) for comparison. As shown in the figure, the position of the slice column in this case needs to be balanced to minimize the routing between the path of the block multiplier and the LUTs, and the path of the LUTs and the external pins, which leads to an unchanged maximum delay value instead of an improvement after the

optimization. The extra cost of routing also explains the decreased savings in bitstream length compared to the shifter or the LUT-based multiplier design, as listed in Table 10.

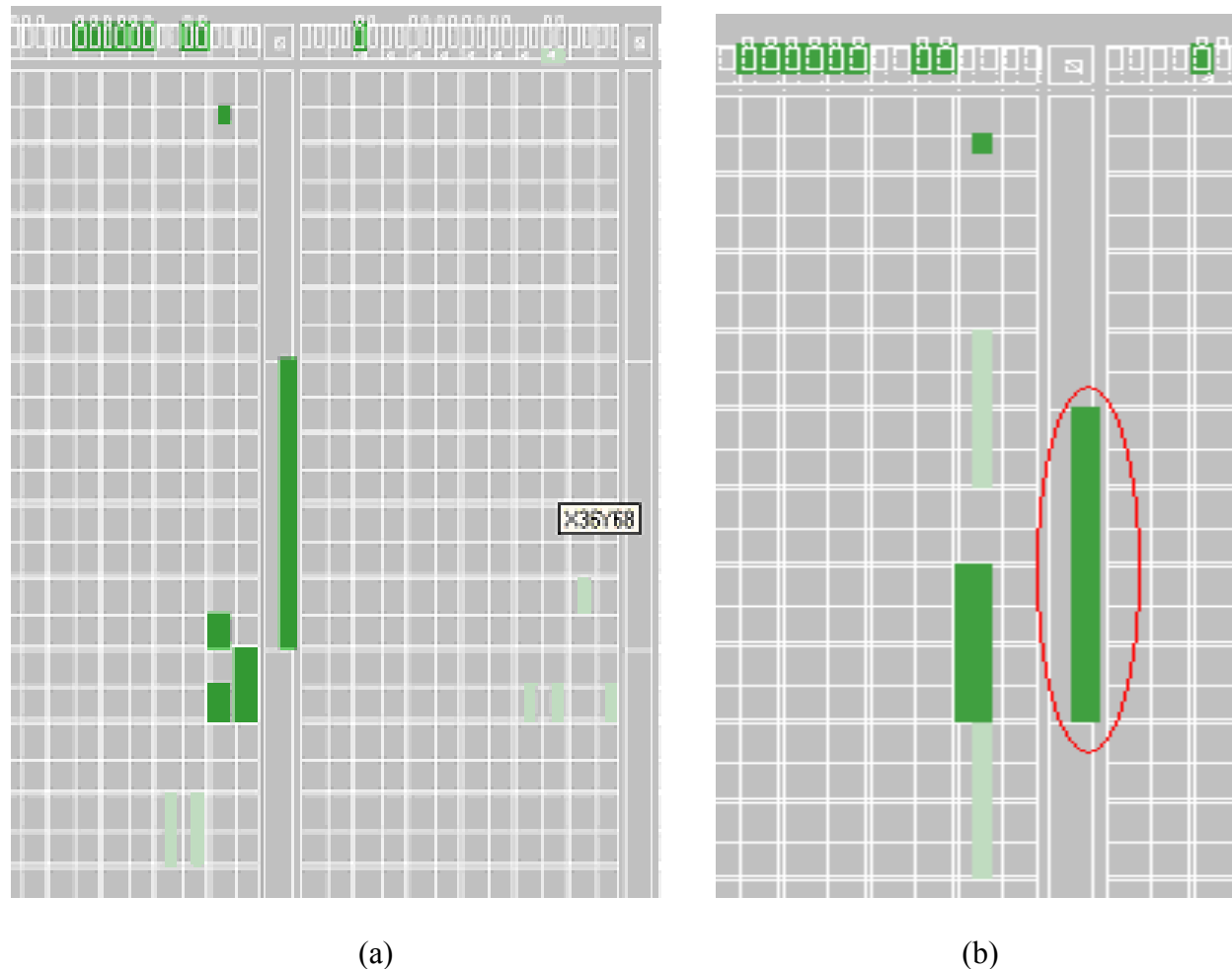


Figure 32: Case 3 Before and After Optimization

For Case #4, the 4-bit×4-bit LUT-based multiplier, the high fan-out situation mentioned above needs to be dealt with. The carry chains, marked in brown, red, and blue in Figure 33(b), have multiple connections to the LUT logic elements in the dark green blocks. Therefore, these carry chains are arranged around the LUT logic blocks instead of in the simple one column style to achieve the best resource area optimization. Figure 33(a) show the tool’s original arrangement and Figure 33(b) shows the result after the optimization.

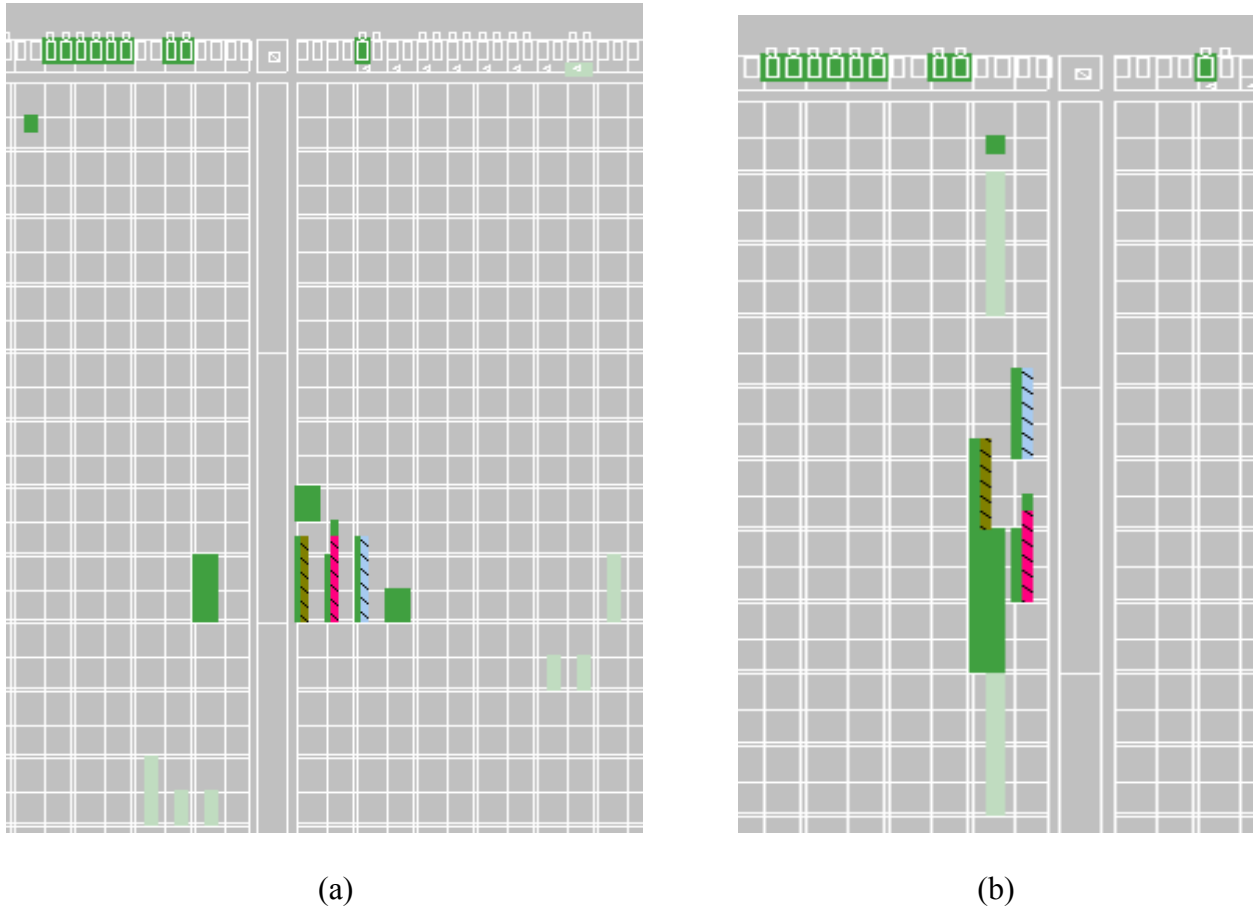


Figure 33: Case 4 Before and After Optimization

The comparative optimization results for these case studies using MRRA are listed in Table 10. The logic resource usage of each of design is also summarized in the table. Partial reconfiguration for designs that comprise as few as four LUTs can achieve 14% area reduction saving. The more complicated case study, involving the 4-bit×4-bit LUT-based multiplier, demonstrates almost 30% reduction using the presented strategy. While the four small case studies illustrate the concept, larger and more involved designs using partial reconfiguration design can achieve higher degrees of bitstream savings. Results also show that in most cases, the maximum propagation delay has been decreased slightly.

In order to verify our area optimization strategy further, one middle-sized module, a Single Error Correction Double Error Detection (SECDED) algorithm and a larger-sized module of the MD5 algorithm, are also implemented with the same area management strategy as the smaller cases using a similar pin arrangement. A total of 74 and 160 slices were used to implement the respective algorithms. In both cases, the partial reconfiguration module occupies 2 or more columns of slices. Due to the large number of resources involved, only slice on the critical path are constrained during the optimization process. The results from the implementation of these modules are listed in the last two rows of Table III. As suggested before, increased bitstream savings of 33% and 30% are achieved because these are comparatively larger modules. Overall, with this area management strategy, about a one-fourth size reduction or higher can be achieved for partial reconfiguration modules. On the other hand, the larger the module is, the more complicated and time consuming the process of specifying resource usage becomes.

Table 10: Area and Bitstream Optimization

Module name	# of LUT	# of FF	# of block Multiplier	# of Slices	Original File Size (Byte)	Original Max. Delay (ns)	Optimized File Size (byte)	Optimized Max. Delay (ns)	Area Saving
4 LUTs	4	16	0	12	64K	1.371	55K	1.347	14%
Shifter	1	24	0	13	87K	1.377	63K	1.367	28%
Block Multiplier	8	25	1	17	88K	1.346	66K	1.346	25%
LUT Multiplier	22	22	0	22	96K	1.367	68K	1.346	29%
SECDED	93	41	0	74	89K	1.355	60K	1.355	33%
MD5	292	128	0	168	120K	1.380	84K	1.322	30%

6.2 Application Analysis

6.2.1 Data Structure Mapping

In order to allow the top-level application process the circuit at logic level and use mapping engine in the translation layer, a detailed circuit data structure has been defined in Chapter 3. It is crucial that the designed partial reconfiguration module can be linked to these data structures correctly. To meet this requirement, currently a few restrictions needs to be made to the circuit design when using the frame based module flow:

- In partial reconfiguration module, all the combinational logic elements are located inside LUT slices only, with determined logic interconnection at design,
- All LUTs are instantiated as 4-input/1-output logic elements, regardless the real logic input requirements,
- All the LUTs' physical locations are determined and recorded at design time, which can be modified at run-time,
- LUT based special elements, such as shifter, RAM or ROM, are not recommended at the current version. If Instantiated, they should maintain their mode, with only their contents to be modified,
- Flip flop can be used, but should not be changed at runtime,
- Circuit with feed back logic is not recommended, and
- Special on-chip resources, such as block RAM, carry chain, or block multiplier, are not recommended.

With above restrictions in effect, the testing circuit is designed and coded in RTL form or even at the LUT granularity level. Behavior-level coding is not acceptable because after synthesis, behavior-level code will be optimized by the tool into an indeterminate number of LUTs with random labels, which cannot be controlled at runtime specifically. With such style of design, the structure of the whole module is fixed at design time without letting the synthesis tools modify or optimize them without notification. Since the circuit is described at RTL or LUT level, after synthesis, all the logic elements are clearly labeled, which will enable the detailed area constraints for each logic element inside the module to be defined and placed into `.ucf` file. The area group constraints, which partition the FPGA resources into column-based rectangle and attach to each module respectively, should be decided at the design time beforehand. Next, each instance of LUT resources inside the partitions needs to be described at specific row and column positions through the `.ucf` file. Eventually, these modules will merge into the top design as components with some communication overhead, such as GNAT or IPIF. After merging, if the scripts fail to run the placement and routing with all the area constraints successfully, the constraints will need to be adjusted until the bistream skeleton is successfully generated. Eventually, the detailed design information can be extract from the `.ucf` file and the bistream file and fill into the data structure designed in Chapter 3.

Table 11 lists the related APIs developed for the high level applications. In order to get the area location functions to work, at the end of each area constraints in `.ucf` file, no space or tab should be put before the semi column, i.e. the semi column directly follows the end of the area constraints text. This will make sure the API can parse and process them correctly. The set of functions only change the modification area of the data structure. The changes only become

effective after the UpdateDesign function is called. In the function of UpdateDesign and DownloadDesign, script is called to carry out part of the tasks by Xilinx toolsets at background.

Table 11: Intermediate Translation APIs

API name	Input Parameter	Operation
ShowDesign	SLUTInfo *pLUTInfo	Show the details of specific individual circuit
GetRow	SLUTInfo *pLUTInfo	Return slice row number of specific LUT
GetColumn	SLUTInfo *pLUTInfo	Return slice column number of specific LUT
GetLUTPosition	SLUTInfo *pLUTInfo	Return LUT position inside the slice
GetTable	SLUTInfo *pLUTInfo	Return truth table of specific LUT
SetPosition	SLUTInfo *pLUTInfo, unsigned short row, unsigned short column, char GorFLUT	The modification fields of the structure are changed
SetTable	SLUTInfo *pLUTInfo, unsigned short NewTable	Only the modification fields of the structure are changed
UpdateDesign	SLUTInfo *pLUTInfo[], char* bitstream, char* ucf	The status fields of the structure are updated, flags are cleared and the new bitstream is generated

6.2.2 Flexible Routing

Recently, a lot of recent of theoretical research work has been carried out on FPGA fault tolerance and area management with partial reconfiguration ability. Most of these algorithms have assumed that the partial reconfiguration process can be implemented at very fine granularity tile level, including routing lines. However, ever since Xilinx had dropped the support on the software of Jbits, this is not the actual case any longer. For current standard partial

reconfiguration flow presented by Xilinx, the granularity is actually at module level, which requires each module to be arranged at slice column level with a four-slice boundary requirement. Special bus macro is also required to communicate between modules. Besides the restricted flexibility due to the coarse granularity, this module based partial reconfiguration flow can only be controlled at a very high level at design time, mostly depending on the Xilinx toolsets to interpret the placement and routing process, which actually can lead to complications when implementing, especially when the partial configuration module size is a little bigger or requires extensive routing resources.

Intuitively, direct manipulation on bitstream should be beneficial, since it can not only provide precise control on both logic function and physical placement but also avoid the lengthy time delay caused by Xilinx toolsets. The biggest challenge for this idea is the understanding of the physical bitstream structure, especially the routing information. Although the logic contents of the LUT in FPGA is still possible to be read out by applying the equations introduced in Chapter 5 through ICAP APIs, the complexity of the routing information makes it almost impossible to be interpreted and manipulated directly in the form of bitstream. Therefore only if a shortcut to deal with routing issue can be found, the direct bitstream manipulation methodology can be really implemented at very fine granularity tile level.

Inside the standard FPGA device, different routing lines are connected by dedicated switch boxes. The bitstream information of these switch boxes are entangled with other complicated routing connection information and can hardly separate out directly. On the other hand, each logic unit of the FPGA, i.e. the LUT, is actually a simple 4 input and 1 output SRAM unit. By changing the content stored inside the SRAM, the logic equations of each LUTs are modified. These LUTs can be actually considered as 4 to 1 multiplexers. The logic content inside

the LUT will decide which one of the 4 input signals can pass through. Therefore, by modifying the content of the LUT with the equations introduced in Chapter 4.5 at runtime, different signals can be dynamically passed to the load signals. In another words, these LUTs become runtime dynamic switch boxes.

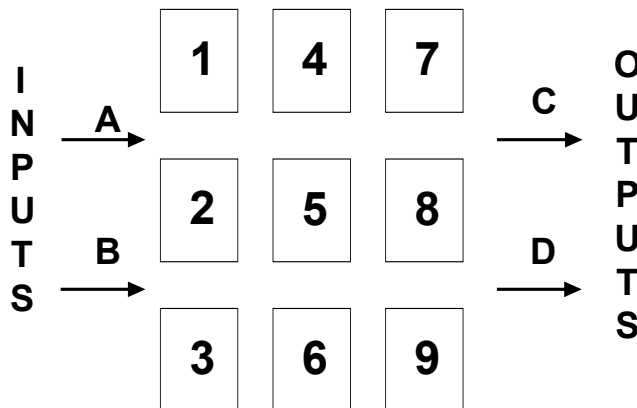


Figure 34: Flexible Routing Example

Figure 34 shows an example of the Flexible Routing functionality structure. This is a 3 x 3 array of logic tiles with 2 required inputs and 2 required outputs. The tiles are labeled from 1 to 9, which can be used to represent the logic interconnection between the logic elements in high-level algorithms. In other words, this numbering system may be used to define a design's netlist. For each tile, it can have 2, 3 or 4 inputs with a corresponding truth table of 4, 8 or 16 bits.

To simplify the design inter-connectivity complexity, the scheme can be restricted as all individual tile inputs could only be connected to cell outputs, which were of a lower number within this scheme. This is sometimes important because it eliminates any possibility of feedback connections, which would give raise to non-combinational time dependent behavior. Figure 35 shows an example of the detailed scheme of the inter-connectivity nets for tile 9. For this tile

there are two inputs. Each connects to an identical interconnect block. The details of the interconnect block I has also been demonstrated in Figure 35.

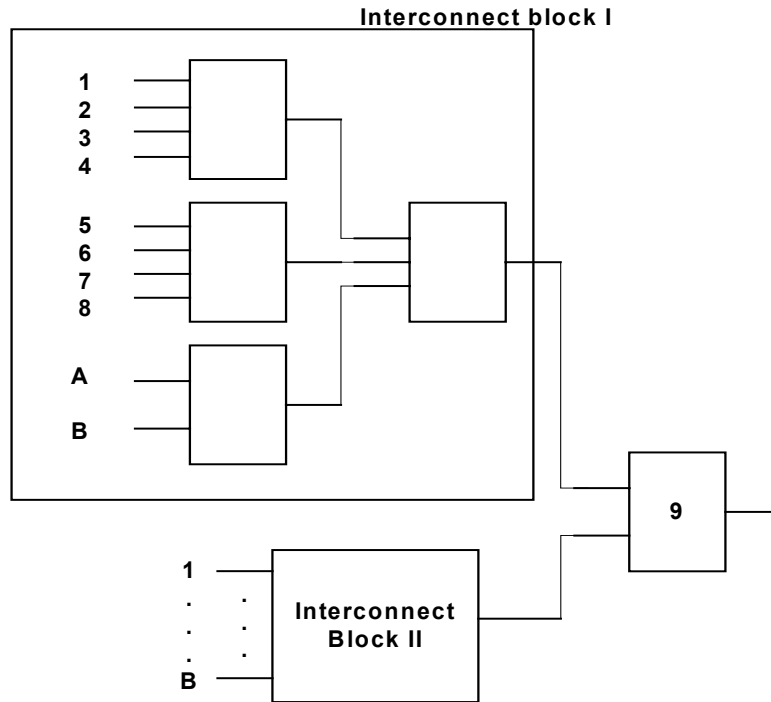


Figure 35: Inter-connectivity Nets for Flexible Routing

As shown in the Figure, the interconnect block has actually been fed with all possible inputs including the previous 8 logic tiles and the 2 external inputs A and B. Since there are 10 inputs, 3 extra tiles have been used to receive these inputs. One more tile is also been placed to consolidate the 3 interconnect tile outputs to a single input to for tile 9. Clearly, with such an interconnect block, the extra tiles are acting as a multiplexer or a switch box. Each time by simply changing the logic function of the tiles in the interconnect block, the input of tile 9 can be changed arbitrarily. In another words, with such a interconnect block insert between logic tiles, the logic interconnects between them can be adjusted by simple true table modification inside the interconnect tiles instead of changing real physical routing resources.

More generally, for each fan-in point, assume there are M instances of LUT tiles used for interconnections with a pyramid structure as shown in Figure 35. Each tile has possible K inputs, where $K = 2, 3, 4$. The maximum allowed flexible routing input using the LUT switch box is given by:

$$I_i = M \times (K - 1) + 1$$

Therefore, the number of maximum input I_i is linearly increased with the tile number M .

Assume there are N pieces of tiles for functionality. Each tile has possible K inputs, where $K = 2, 3$ or 4 . If the constraints discussed in above example applies, the number of possible input I_i of each tile T_i , should be:

$$I_i = T_i - 1 + I_{input}$$

where I_{input} is the total number of external input signals; $T_i = 1$ to $N-1$.

Therefore the number of stages S_i required in the interconnect block will be

$$S_i = \lceil \log_4 I_i \rceil$$

since the max input number of each tile should be 4.

For each stage, the number of tiles used for interconnect N_s is given by:

$$N_s = \left\lceil \frac{I_i}{4^s} \right\rceil$$

where $s = 1$ to S_i . Therefore the total number of tiles N_i used in interconnect block for tile T_i is:

$$N_i = K \times \sum_{s=1}^{S_i} N_s$$

And the total number of tiles N_{total} for interconnect is:

$$N_{total} = \sum_{i=1}^N N_i$$

Based on the equations, clearly with increase in the number of block of the flexible routing scheme, the extra LUT penalty also increases significantly at polynomial rate if a full scheme of flexible routing is attempted. Hence, it is not practical nor necessary to connect all the LUT block inside the design module. Instead, locating the critical path or fan-in points and coupling multiple groups of LUTs units will be a more reasonable strategy and result in much more efficient flexible interconnections with a small resource penalty.

6.3 GA Execution MRRA

As mention in the previous section, GA changes during evolution must adhere to the Xilinx-defined format of the bitstream. Since not all bitstream information is decipherable and can be manipulated, adaptations need to be made to the general GA operators. To Undesirable conditions that may damage the FPGA, such as mutation, which might inadvertently tie two logic outputs tied together, must also need to be prevented. After using the design strategy described in Chapter 6.2, the data structure, which contains logic and physical ordering information plus the configuration I/O information is generated.

To perform the GA, first a group of circuit individuals are generated. These individuals should use the same amount of physical resources. Extra amount of LUT resources may need to be added into the design to provide logic and routing margins for evolving flexibility. The individuals are functionally identical from the outside black box view, which means the same input will generate same outputs through all individuals. Yet these individuals are also physically

different from the inside clear box view, which means each circuit individuals shows its own unique physical resource arrangement and the functions instantiate inside each LUT which also exhibits variations between individuals. This will provide enough diversity for GA to evolve and yet prevent to generate significant deviations through evolving process.

The mutation operator of the GA is modified in order to fit the FPGA architecture, which varied with a traditionally defined mutation operator. In addition to the inverse binary bit approach at the LUT function level, certain input interconnections of LUTs may also be mutated using the flexible routing design strategy described in Chapter 6.2. The mutation will rearrange the input interconnection to each input pin of LUTs in order to search the potential unused resources for an occluded fault impacted resource. In this way, the functionalities of LUTs are undistorted and explored in the search space. The new interconnection may use some inherent redundancy resource existing in the original design. This operator will provide some opportunity for fault correction strategy for either input stuck-at fault or LUT content stuck-at fault.

For the crossover operator of the GA, since the logic orders of each LUT are fixed, the possible search space of the initial design is limited. The crossover point cannot be randomly picked between the two configurations. Therefore two configurations need to be aligned, and a crossover site is picked uniformly at random along the boundary of the LUTs. This crossover point defines an exchange section that is used to switch the circuit information between the two parents through LUT-by-LUT exchange operations. This results in both configuration having duplicate elements and similar replacement mutation reoccurs to avoid such correct functionality behavior.

6.3.1 Performance Evaluation

To test the designed GA workflow, a 4-bit x 4-bit adder is incorporated into the MRRA platform. This provides a tractable circuit for the GA to evolve that exhibits characteristics for large arithmetic circuits including a variable amount of redundancy and combinational logic behavior. The GA parameters used throughout the experiments are shown in Table 12. Total of 8 LUTs were used in the design experiments, this number was increased to 13 LUTs in the repair experiment to add some redundancy margin for the GA to evolve within. The second column shows the tested parameters by running extrinsic evolution of the GA. The third column illustrates the optimal values that are found through the experiment. For example, population sizes between 5 and 20 were evaluated and best results were achieved using population size of 10.

There are three types of experiments that have been performed:

Unseeded Design: In this experiment the GA evolved the 4-bit x 4-bit adder with a randomly-seeded initial population. The purpose of this experiment is to demonstrate the capability to intrinsically evolve 100% functional circuits starting from random functional bitstream.

Seeded Design: In this experiment, the GA evolved the 4-bit x 4-bit adder from an initial population of partially functional individuals in addition to completely random ones. The partially functional seeds were originally fully functional designs, which were tampered by deliberately exposing them to mutation operator. This arrangement emulates a fault-scenario in real life avionics or space applications in which the configuration bitstream is partially affected by Single Event Upset (SEU) due to cosmic radiation.

Table 12: GA Parameters [52]

Parameter	Range Evaluated	Value Selected
Number of LUTs for design	8	8
Number of LUTs for repair	8-13	13
Population Size	5-20	10
Mutation Rate	5%-90%	50%
Crossover Rate	30%-90%	60%
Tournament Size	1-8	6
Elitism Size	1-2	1

Repair: A single stuck-at one and stuck-at zero were adopted as a case study to show the capability of the platform to repair the faulty circuit. Since an actual fault cannot be readily or precisely introduced into the device, the circuit is stimulated to behave as if the fault actually exists.

6.3.2 Results Analysis

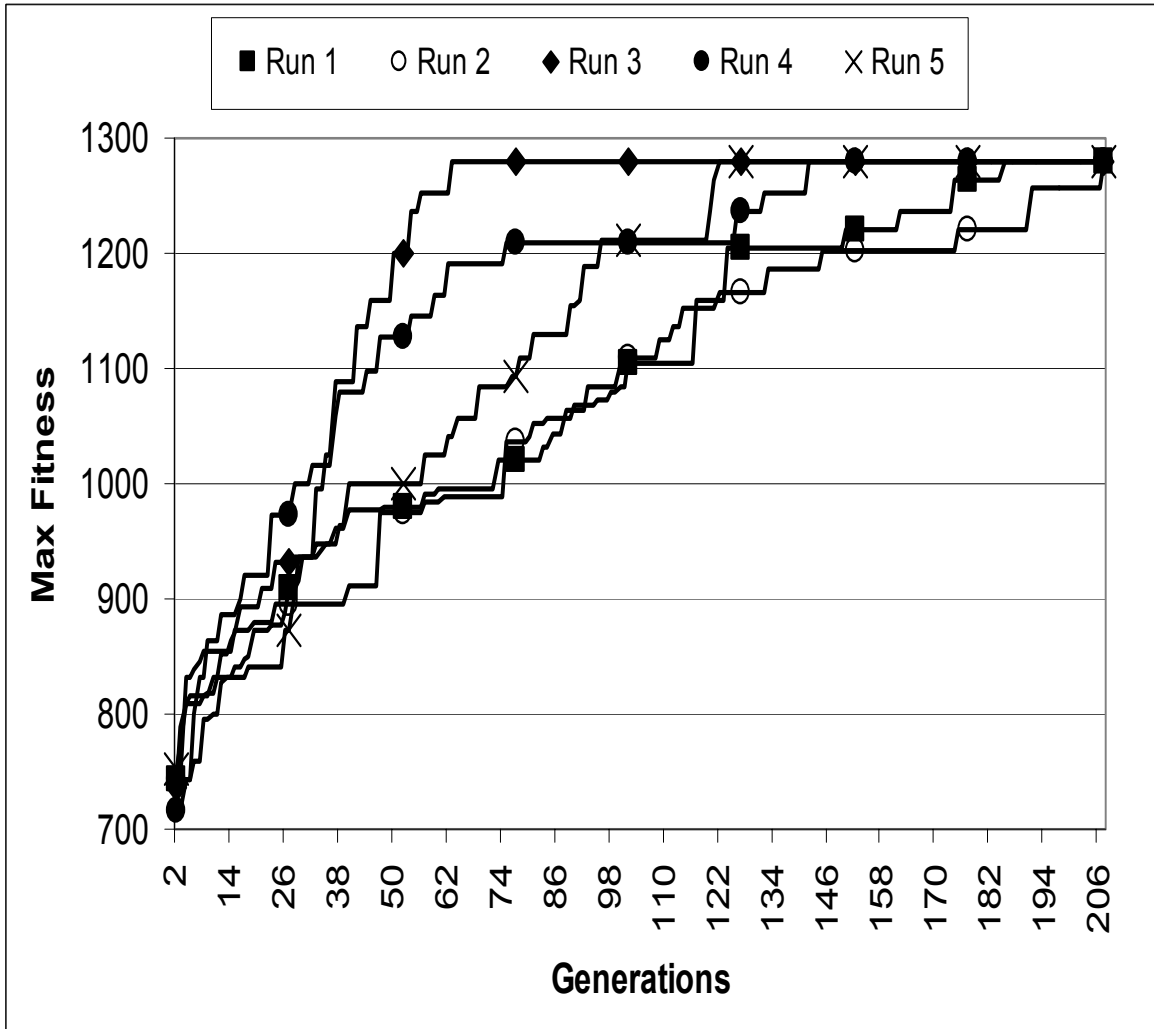


Figure 36: Unseeded Design GA Runs [52]

Five intrinsic evolutions were achieved for each of the unseeded, seeded, and repair experiments. The GA parameters listed in Table 12 were used. To quantify the capability of the platform, maximum fitness, which is the numerical measure of the fitness for the best individual of the final generation of the run, is used.

Figure 36 shows five runs that demonstrate the capability of the platform to evolve to fully working 4-bit x 4-bit Adder designs starting from scratch. The maximum fitness starts as low as 716 out-of-1280, and rapidly increases during the first few generations. Figure 37 shows five runs where a fully working 4-Bit x 4-Bit Adder was designed from a partially working seed. Five different seeds were used in the five runs.

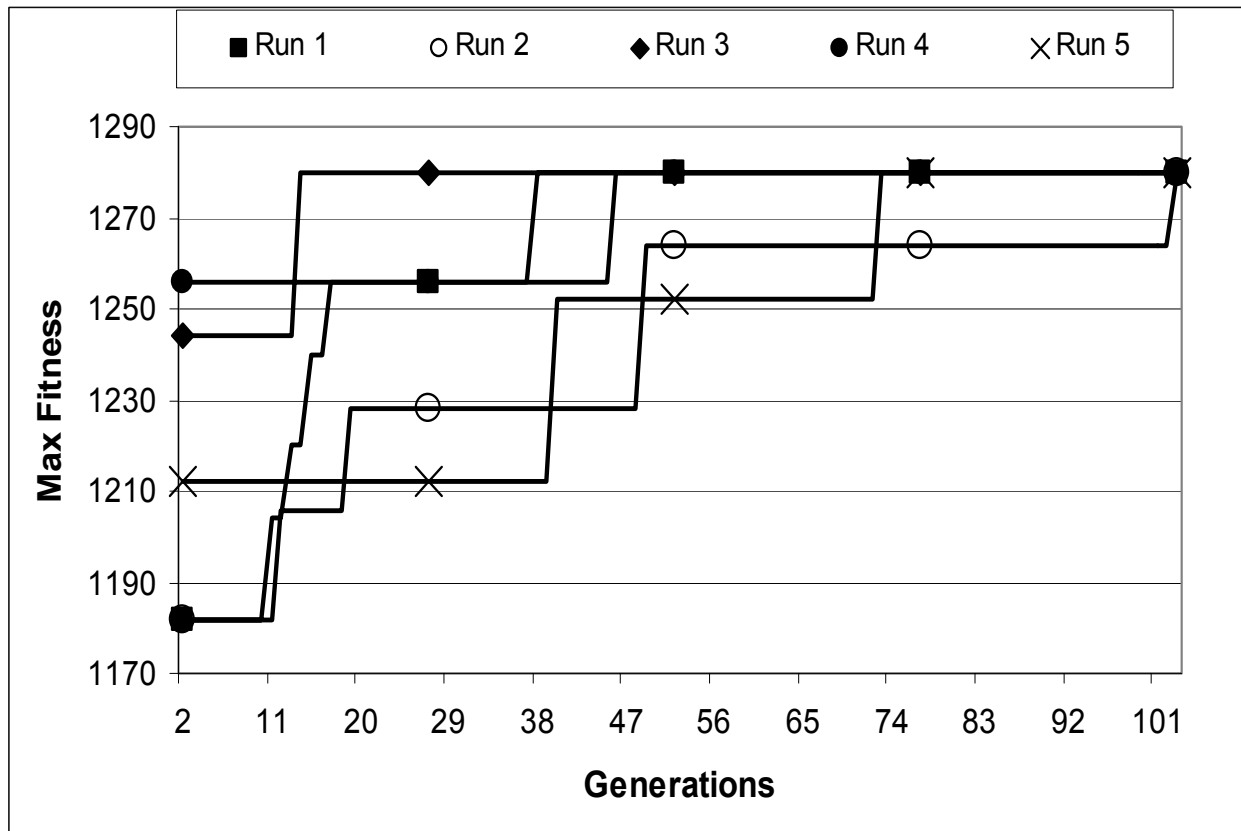


Figure 37: Seeded Design GA Runs [52]

Figure 38 shows five runs in which the platform was used to repair the broken 4-Bit x 4-Bit Adder. A stuck-at zero fault was randomly injected in the first input pin of the third LUT of the original design. The fault was introduced using the technique mentioned in section 4.1. This

fault reduces the circuit's fitness to 1152 out-of-1280. The fastest converging run was Run 4, which reached to full fitness after 94 generations.

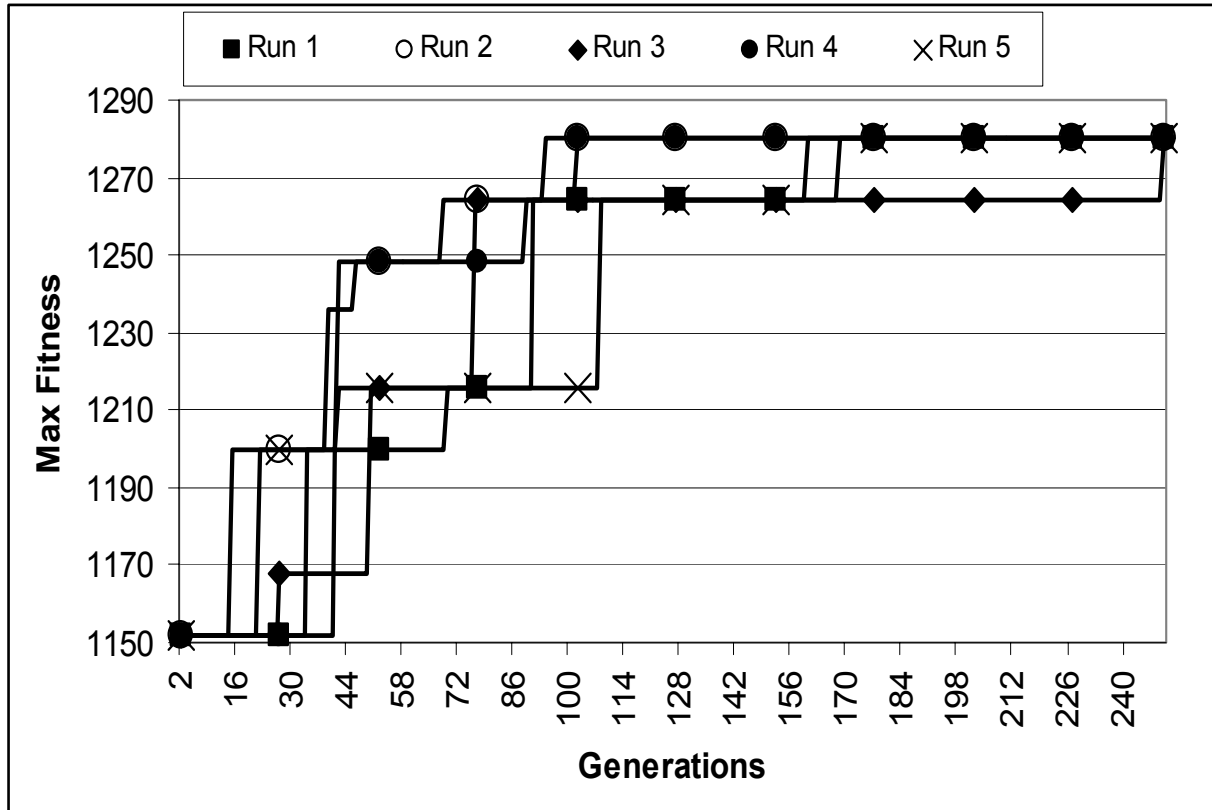


Figure 38: Repair GA Runs [52]

6.4 Chapter Summary

This chapter is organized into four sections. In Chapter 6.1, a bitstream optimization technique is introduced. Instead of relying on the design tools' random placement, most of the logic resources are predetermined at specific physical positions based on several principles. The proposed methodology is evaluated on the Virtex II Pro platform. Six diversified circuits, including 4 small case studies, one mid-sized case study and one application-sized evaluation,

were studied using the proposed techniques. The optimized bitstream sizes of these circuits are compared to the original tool-generated cases. The result shows file sizes can be reduced up to 30% on a variety of designs compared to non-area managed configurations. The experiments also imply that even higher rates of reduction can be achieved on larger designs. This has positive implications for both bit stream storage in SOC environment and correspondingly faster reconfiguration time due to reduced file transfer time.

Chapter 6.2 demonstrated the advanced design strategy to link the hardware and the software control with routing flexibility. A detailed design strategy based on the current Logic Layer data structure were presented. A LUT-based switch box structure was also introduced for runtime dynamic routing. The extra resource utilization using this structure was quantitatively evaluated. Analysis shows that a full flexible routing structure will result in high resource penalty. Instead, locating the critical path or fan-in points and coupling multiple groups of LUTs units will provide a more reasonable strategy.

In the Final section of Chapter 6, a fault tolerance GA application is introduced. High-level GA performance is evaluated by using the above techniques. The fitness evaluations of 3 different groups of GA experiment are carried out and evaluated to show the MRRA overall capability for autonomous operation. Experiments show that MRRA successfully enables the general fault repair logic with the real hardware circuit and attains suitable original algorithm specifications.

CHAPTER 7: CONCLUSION

The basic idea of this dissertation relies on a technology named partial reconfiguration, which has been introduced by several vendors as a hardware capability in very recent years. A lot of attention from academic arena has been drawn towards this new technology as workable commercial paths remain to be identified. NASA, as an interested user of such technology, has attempted to perform autonomous fault repair towards a faulty hardware system in an environment where human intervention is impractical, such as in deep space missions. Significant research work has been made from different aspects to address such basic capability. The aspects include special hardware communication infrastructure, full hardware platform development, hardware speed optimization and software control optimization. In this dissertation, we proposed, developed, and evaluated a comprehensive architecture covering all the current research area aspects, evaluated the available techniques, and created the missing pieces to bridge theory and technology gaps to provide a demonstrated hardware and software solutions between the detailed hardware system and the abstract high-level application.

7.1 Summary

The Multi-Layer Runtime Reconfiguration Architecture utilizes a three-layer paradigm including the Reconfiguration Layer, Translation Layer and Logic Layer to support autonomous partial reconfiguration. The Reconfiguration Layer is mostly a hardware layer, which provides the basic reconfiguration interface and multiple testing and communication hardware control modules. A set of low-level communication and testing drivers are also developed and attached to these hardware modules. The Translation Layer stands between the bottom Reconfiguration

Layer and the top Logic Layer. It shields the details of the hardware from the general control logic by presenting a translation engine with a set of standard APIs. This translation engine can interpret the independent control logic, including both the logic functions and the physical routings, into corresponding hardware dependent digital circuit implementations. A general data structure containing the hardware logic function information as well as physical allocation information has been defined in the Logic Layer. This works well because high-level algorithms only need to focus on these general data structure and perform their normal routines without concerns for hardware compatibility. The final modification result will be stored in the defined general data structure and passed to the Translation Layer for further interpretation. This three-tier framework enables task-level modularity, framework routine encapsulation, and API standardization. This brings the feasibility of hardware control and modifications to the autonomous software algorithms.

A corresponding control flow is also proposed in details for both design time and runtime scenario. At design time, the adopted Module Based Flow is used to generate the full design. The full hardware system is first partitioned into modules and designed from the top view to the bottom. Meanwhile, *One-Dimensional Area Management* is performed on the full physical FPGA device by partitioning it into 1-dimensional column-shaped rectangles, in which all the fixed and reconfigurable modules will be arranged based on the size of each module and extra specified area constraints from design specifications. After these arrangements all of these top views and modules are implemented and verified individually. The size of each module can be further optimized by additional *Two-Dimensional Area Allocation* placements inside each module. The Optimized partial reconfiguration bitstream for the specific modules are also generated at this

step. Finally, all the individual modules are created by *Final Assembly* based on the top-level view and are ready to be downloaded to the FPGA device as Configuration Data bitstreams.

After the initial bitstream is downloaded, based on the user logic control, the precompiled partial bitstream can be stored at non-volatile memory and monitored by the algorithms in the Logic Layer and updated directly to the device for dynamic reconfiguration when it is necessary. This control flow may require larger amount of storage memory to be carried out. On the other hand, new modification requests can be generated by using the Frame Based Flow at run-time instead of using predefined and precompiled modules. Although the boundary of each module is fixed, the physical logic resources inside each module can be re-allocated at runtime. Logic function modification request for each LUT inside of the modules can also be generated based on the user requirement as well. Both requests from the Logic Layer will eventually wait for the translation engine from lower layer to interpret to the corresponding configuration and reconfiguration by the Reconfiguration Layer. Detailed calculation of equations and their interpretation have been presented with examples in this dissertation.

In order to achieve the maximum performance and system design flexibility for MRRA, three different mainstream reconfiguration interfaces have been investigated, including the SelectMAP, JTAG, and ICAP. Different reconfiguration and verification methodology are formed based on the different characteristics of these interfaces. Several varieties of loosely-coupled or SOC reconfiguration and testing system have been also developed based on the strategy, providing standard APIs for the upper layer to use. These systems illustrate a continuum of paradigms for runtime partial reconfiguration interfaces and control. The demands for runtime partial reconfiguration capability in embedded SOC applications are achieved by using the on-chip CPU core and Block memory, providing multiple bitstream generation choices, including

direct bitstream manipulation for logic functions and hybrid one-dimensional and two-dimensional physical area re-allocation control.

Prototypes for the different reconfiguration interfaces based the MRRA concept were developed on two standalone COTS hardware platforms: Avnet Virtex II Pro development board and a Pentium PC with Xilinx ISE and EDK FPGA CAD software suites. Adoption of several standard industry hardware interfaces, IPIF, PCI, etc., along with the design and refinement of appropriate communication and synchronization protocols provides a powerful and useful abstraction technique. Resource utilization estimations have been carried out on these prototype platforms. Overall, when the SelectMAP or ICAP interface with external SRAM is used to establish reconfiguration and testing channels, sophisticated hardware logic is involved and excessive pin usage is incurred, which consumes a factor of 5 to 18 times more logic resources in the fabric than the JTAG-based prototype. These costs can limit the size and area placement flexibility of the reconfigurable modules. As a result, less physical rectangle areas will be available for the partial reconfiguration modules. Hence, less flexibility is available for the control logic. In this case, larger capacity FPGAs with extra external pins, are highly recommended. Furthermore, additional effort is also required for pin assignments and connections with special bus macros thereby resulting in a corresponding increase in design complexity.

MD5/SHA-1 hash and other circuits have been implemented as reconfigurable modules to evaluate the performance of the hardware and the logic control flow. The experiments on the MD5/SHA-1 hash algorithms compared the resource utilization and power consumption for the different design flows. Results show significant improvement when using the proposed partial reconfiguration control flow, especially the Frame Based flow. A number of benchmark and

hashing algorithm case studies demonstrates the range timing variations of autonomous and dynamic reconfiguration operations. In contrast to area arrangement flexibility, the SelectMAP and ICAP interface shows much higher speed when transferring the reconfiguration and testing data than JTAG. The result also illustrates that when physical re-routing by Xilinx tool engine is involved, the speed of this translation process becomes the bottleneck of autonomous system performance, even though it has already shown significant improvement compared to the baseline timing performance. Experiments also demonstrate that the integrated design flow is able to retain and achieve upward compatibility with vendor toolsets.

Advanced optimization techniques, including bitstream optimization, high-level module design strategy and LUT base switch box strategies are presented. Additional experiments have been carried out on a diversified group of circuits using the bitstream optimization. The result shows up to 30% of area saving can be achieved after applying such techniques. In the Final section of Chapter 6, a fault tolerance GA application is discussed. Experiments show that MRRA successfully connects the general fault repair logic with the real hardware circuit and reaches the original algorithm specifications.

7.2 Future Work

The results in this dissertation can surely be a point of departure for further research. Currently, when combining with high-level applications with partial reconfiguration modules, only combinational logic has been evaluated. The next step will be attempting adding sequential elements with possible feedback paths into the design. Special on-chip resources, such as Block RAMs, Multipliers, etc., may also be added in. This will require extra control state machines and

storage elements at a fixed region to save the states before modification and make necessary restorations after the reconfigurations. Corresponding control algorithms also need to be developed at either Translation Layer or Logic Layer as an independent module.

Recently, a new FPGA family, such as Xilinx Virtex 4, has officially been released to the public. The new FPGA family provides similar reconfiguration interface yet smaller reconfiguration area granularity at certain 2-dimensional rectangle areas. Therefore another possible branch for the continuation of this work is to port the MRRA to the new hardware platform. The current one runs on Virtex II and Pro family, which limits the module level area management to be 1-dimensional. Because of the intrinsic layered characteristic of the design, the system should be able to port to the new hardware in a very short period of time. On the new Virtex 4 FPGA family, a two-dimensional area management can be truly proposed at the hardware level, instead of simple theoretical discussions with hardware ability assumptions. This will open an even further area of resource defragmentation and allocation at in a practical form.

In this dissertation, Genetic Algorithms are the only major top-level application that has been explored. It is important to continue the path for further evaluations with the GA behaviors on larger and more complicated case studies. As pointed out in Chapter 6, redundancy is a requirement for the success of the evolutionary process. This dissertation has explored the possibility of using the redundancy as routing switch box to add diversity to the design. How much extra flexibility needs to provide, where to insert these redundancies, and how to pinpoint the key elements have not been fully analyzed at algorithm level yet. To extend these questions into a formal graph-driven study will be a key follow up for the GA part of research.

As discussed in Chapter 2, a lot of theoretical research has been done at design level to reach a faster partial reconfiguration paradigm. Yet previously less had been done at the actual

hardware level. With the current MRRA achievement, it becomes possible to evaluate these design-time strategies and provide measurable benefit in meeting the challenges in routing and area management.

LIST OF REFERENCES

- [1] A.K. Raghavan, and P. Sutton, "JPG - A Partial Bitstream Generation Tool to Support Partial Reconfiguration in Virtex FPGAs," in *Proceedings of International Parallel and Distributed Processing Symposium, (IPDPS'02)*, Fort Lauderdale, Florida, USA, April 15-19, 2002.
- [2] A. Thompson, "An Evolved Circuit, Intrinsic in Silicon, Entwined with Physics," *Evolvable Systems: From Biology to Hardware*, Vol. 1259, 1997, pp. 390-405.
- [3] A. Upegui, "Evolving Hardware by Dynamically Reconfiguring Xilinx FPGAs," in *Proceedings of the ICES 2005 on Evolvable Hardware*, Sitges, Spain, September 12-14, 2005.
- [4] A. Upegui, C. A. Peña-Reyes, and E. Sanchez, "An FPGA Platform for Online Topology Exploration of Spiking Neural Networks," *Microprocessors and Microsystems*, Vol. 29, Issue 5, 1 June 2005, pp. 211-223.
- [5] B. Blodget, P. James-Roxby, E. Keller, S. McMillan, and P. Sundararajan, "A Self-Reconfiguring Platform," in *Proceedings of Field-Programmable Logic and Applications 2003*, Lisbon, Portugal, September 1-3, 2003.
- [6] B. Blodget, S. McMillan, and P. Lysaght, "A Lightweight Approach for Embedded Reconfiguration of FPGAs," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, Munich, Germany, March 03-07, 2003.
- [7] B. Jeong, S. Yoo, S. Lee, and K. Choi, "Hardware-software Cosynthesis for Run-time Incrementally Reconfigurable FPGAs," in *Proceedings of Design Automation Conference Asia and South Pacific 2000 (ASP-DAC 2000)*, Yokohama, Japan, January 25-28, 2000.
- [8] C. Bobda, M. Majer, D. Koch, A. Ahmadinia, and J. Teich, "A Dynamic NoC Approach for Communication in Reconfigurable Devices," in *Proceedings of International Conference on Field-Programmable Logic and Applications (FPL'04)*, Antwerp, Belgium, August 30 - September 01, 2004.
- [9] C. Bobda, M. Majer, D. Koch, A. Ahmadinia, A. Linarth and J. Teich, "The Erlangen Slot Machine: Increasing Flexibility in FPGA-Based Reconfigurable Platforms," in *Proceedings of IEEE 2005 Conference on Field-Programmable Technology (FPT'05)*, Singapore, December 11-14, 2005.
- [10] C. Haubelt, S. Otto, C. Grabbe, and J. Teich, "A System-Level Approach to Hardware Reconfigurable Systems," in *Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC'05)*, Shanghai, China, January 18-21, 2005

- [11] D. Mesquita, F. Moraes, J. Palma, L. Moller, and N. Calazans, "Remote and Partial Reconfiguration of FPGAs: Tools and Trends," in *Proceedings of Parallel and Distributed Processing Symposium 2003*, Nice, France, April 22-26, 2003.
- [12] E. Horta, J. Lockwood, D. Taylor, and D. Parlour, "Dynamic Hardware Plugins in an FPGA with Partial Run-time Reconfiguration," in *Proceedings of Design Automation Conference (DAC'02)*, New Orleans, LA, USA, June 10-14, 2002.
- [13] G. Tufte, and P. C. Haddow, "Biologically-Inspired: A Rule-Based Self-Reconfiguration of a Virtex Chip," in *Proceedings of Computational Science - ICCS 2004*, Boston, MA, USA, May 16-21, 2004.
- [14] G. J. M. Smit et al., "Dynamic reconfiguration in Mobile System," in *Proceedings of the 12th International Conference on Field Programmable Logic and Applications (FPL'02)*, Montpellier, France, September 2-4, 2002.
- [15] G. Wigley, and D. Kearney, "The Development of an Operating System for Reconfiguration Computing," in *Proceedings of Design Automation and Test in Europe (DATE'03)*, Munich, Germany, March 03-07, 2003.
- [16] H. Kalte, G. Lee, M. Pormann, and U. Ruckert, "REPLICA: A Bitstream Manipulation Filter for Module Relocation in Partial Reconfigurable Systems," in *Proceedings of 19th IEEE International Proceedings of Parallel and Distributed Processing Symposium*, Denver, Colorado, USA, April 04-08, 2005.
- [17] H. Tan, "A Technical Report on Virtex II/ Pro Bitstream Format," available at: <http://cal.ucf.edu>.
- [18] H. Tan, and R. F. DeMara, "A Device-Controlled Dynamic Configuration Framework Supporting Heterogeneous Resource Management," in *Proceedings of Engineering of Reconfigurable System and Algorithm (ERSA'05)*, Las Vegas, USA, June 27-30, 2005.
- [19] H. Tan, and R. F. DeMara, "A Physical Resource Management Approach to Minimizing FPGA Partial Reconfiguration Overhead," in *Proceedings of the IEEE International Conference on Reconfigurable Computing and FPGAs (Reconfig'06)*, San Luis Potosi, Mexico, September 20 – 22, 2006.
- [20] H. Tan, R. F. DeMara, A. J. Thakkar, A. Ejnoui and J. D. Sattler, "Complexity and Performance Evaluation of Two Partial Reconfiguration Interfaces on FPGAs: a Case Study," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'06)*, Las Vegas, Nevada, U.S.A, 2006.
- [21] H. Tan, and R. F. DeMara, "A Multi-layer Framework Supporting Autonomous Runtime Partial Reconfiguration," accepted by IEEE Transaction of VLSI on 17 July 2007.

- [22] H. Walder; C. Steiger; and M. Platzner, "Fast Online Task Placement on FPGAs: Free Space Partitioning and 2D-hashing," in *Proceedings of Parallel and Distributed Processing Symposium 2003*, Nice, France, April 22-26 2003.
- [23] H. Walder, and M. Platzner, "Reconfigurable Hardware Operating Systems: From Design Concepts to Realization," in *Proceedings of Engineering of Reconfigurable System and Algorithm (ERSA '03)*, Las Vegas, Nevada, USA, June 23-26, 2003.
- [24] H. Walder, and M. Platzner, "A Runtime Environment for Reconfigurable Hardware Operating Systems," in *Proceedings of the 14th Field Programmable Logic and Applications (FPL '04)*, Leuven, Belgium, August 30-September 1, 2004.
- [25] I. Kennedy, "Fast Reconfiguration Through Difference Compression," in *Proceedings of Field-Programmable Custom Computing Machines 2003 (FCCM '03)*, Napa, California, USA, April 9-11 2003.
- [26] I. Robertson, J. Irvine, P. Lysaght and D. Robinson, "Improved Functional Simulation of Dynamically Reconfigurable Logic," in *Proceedings of the 12th International Conference on Field Programmable Logic and Applications (FPL '02)*, Montpellier, France, September 2-4, 2002.
- [27] J. D. Lohn, G. Larchev, and R. F. DeMara, "A Genetic Representation for Evolutionary Fault Recovery in Virtex FPGAs," in *Proceedings of 5th International Conference on Evolvable Systems*, Trondheim, Norway, March 2003.
- [28] J. Emmert; C. Stroud; B. Skaggs; and M. Abramovici, "Dynamic Fault Tolerance in FPGAs via Partial Reconfiguration," in *Proceedings of Field-Programmable Custom Computing Machines 2000*, Napa Valley, CA, USA, April 17-19 2000.
- [29] J. F. Miller, P. Thomson, and T. Fogarty, "Designing Electronic Circuits using Evolutionary Algorithms. Arithmetic Circuits: A Case Study," in *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science*, D. Quagliarella, J. Periaux, C. Poloni and G. Winter, Eds. UK-Wiley, 1997, pp. 105-131.
- [30] J. Lohn, J. Crawford, A. Globus, G. Hornby, W. Kraus, G. Larchev, A. Pryor, and D. Srivastava, "Evolvable Systems for Space Applications," accepted for oral presentation at the International Conference on Space Mission Challenges for Information Technology (SMC-IT), Pasadena, CA, USA, July 13 – 16, 2003.
- [31] J.O. Cadens, G.M. Megson, and T.P. Plaks, "Quantitative Evaluation of Three Reconfiguration Strategies on FPGAs: A Case Study," in *Proceedings of High Performance Computing in the Asia-Pacific Region 2000*, Beijing, China, May 14-17 2000.

- [32] J. Rose, A. El Gamal, A. Sangiovanni-Vincentelli, "Architecture of Field-Programmable Gate Arrays," in *Proceedings of the IEEE*, Volume: 81, Issue: 7, July 1993, pp. 1013 – 1029.
- [33] J. Williams, and N. Bergmann, "Embedded Linux as a Platform for Dynamically Self-Reconfiguring Systems-On-Chip," in *Proceedings of Engineering of Reconfigurable Systems and Algorithms (ERSA 2004)*, Las Vegas, Nevada, USA, 21-24 June, 2004.
- [34] J-Y. Mingolet, V. Noller, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Infrastructure for Design and Management of Relocatable Task in a Heterogeneous Reconfigurable System-on-Chip," in *Proceeding of Design Automation and Test in Europe (DATE'03)*, Munich, Germany, March 3-7 2003.
- [35] K. Bazargan; R. Kastner; and M. Sarrafzadeh; "Fast Template Placement for Reconfigurable Computing Systems," *IEEE Design & Test of Computers*, Volume: 17, Issue: 1, Jan.-March 2000, pp: 68 – 83.
- [36] K. Compton, L. Zhiyuan, J. Cooley, S. Knol, and S. Hauck, "Configuration Relocation and Defragmentation for Run-time Reconfigurable Computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Volume: 10, Issue: 3, June 2002, pp. 209 – 220.
- [37] K.P. Raghuraman, H. Wang, and S. Tragoudas, "A novel Approach to minimize reconfiguration cost for LUT-based FPGAs," in *Proceedings of the 18th International Conference on VLSI Design (VLSID'05)*, Kolkata, India, January 3-7, 2005.
- [38] K. Zhang, R. F. DeMara, and C. A. Sharma, "Consensus-based Evaluation for Fault Isolation and On-line Evolutionary Regeneration," in *Proceedings of the International Conference in Evolvable Systems (ICES'05)*, Barcelona, Spain, September 12 - 14, 2005.
- [39] M. Abramovici, J. M. Emmert, and C. E. Stroud, "Roving STARS: An Integrated Approach To On-Line Testing, Diagnosis, And Fault Tolerance For FPGAs In Adaptive Computing Systems," in *Proceedings of NASA/DoD Workshop on Evolvable Hardware 2001* Long Beach, California, USA, July 12 - 14, 2001.
- [40] M. Huebner, T. Becker, and J. Becker: "Real-time LUT-based Network Topologies for dynamic and partial FPGA Self-Reconfiguration," in *Proceedings of SBCCI'04*, Porto de Galinhas, Brazil, September 7-11, 2004.
- [41] M. Huebner, C. Schuck, J. Becker: "Elementary Block Based 2-Dimensional Dynamic and Partial Reconfiguration for Virtex-II FPGAs," in *Proceedings of RAW2006*, Rhodes Island, Greece, April 25-26, 2006.
- [42] Michael Barr, "A Reconfigurable Computing Primer," *Multimedia Systems Design*, pp. 44-47, September 1998.

- [43] M.G. Gericota, G.R. Alves; M.L. Silva; and J.M. Ferreira, "Run-time Management of Logic Resources on Reconfigurable Systems," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, Messe Munich, Germany, March 3-7, 2003.
- [44] M.Y. Wang, C. P. Su, C. T. Huang, and C. W. Wu, "An HMAC Processor with Integrated SHA-1 and MD5 Algorithms," in *Proceedings of the Asia and South Pacific Design Automation Conf. 2004 (ASP-DAC'04)*, Yokohama, Japan, January 27 – 30, 2004.
- [45] N. Bergmann, J. Williams, and P. Waldeck, "Egret: A Flexible Platform for Real-Time Reconfigurable System-on-Chip," in *Proceedings of International Conference on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, Nevada, USA, June 23-26, 2003.
- [46] N. Shirazi; W. Luk, and P.Y.K. Cheung, "Automating Production of Run-time Reconfigurable Designs," in *Proceedings of FPGAs for Custom Computing Machines 1998*, Napa Valley, CA, USA, April 15-17 1998.
- [47] O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt, "Dynamic Scheduling of Tasks on Partially Reconfigurable FPGAs," *Computers and Digital Techniques, IEE Proceedings-*, Volume: 147, Issue: 3, May 2000, pp. 181 – 188.
- [48] P. Sedcole, B. Blodget, J. Anderson, P. Lysaghi, and T. Becker, "Modular Partial Reconfiguration in Virtex FPGAs," in *Proceedings of 2005 International Conference of Field Programmable Logic and Applications*, Tampere, Finland, August 24-26, 2005.
- [49] R. F. DeMara and K. Zhang, "Autonomous FPGA Fault Handling through Competitive Runtime Reconfiguration," in *Proceedings of the NASA/DoD Conference on Evolvable Hardware (EH'05)*, Washington D.C., U.S.A., June 29 – July 1, 2005.
- [50] R. Hartenstein, "A Decade of Reconfigurable Computing: a Visionary Retrospective," in *Proceedings of Design Automation and Test in Europe 2001*, Munich, Germany, March 13-16, 2001.
- [51] R.J. Fong, S.J. Harper, and P.M. Athanas, "A Versatile Framework for FPGA Field Updates: An Application of Partial Self-Reconfiguration," in *Proceedings of 14th IEEE International Workshop on Rapid Systems Prototyping*, San Diego, CA, USA, June 9-11, 2003.
- [52] R. S. Oreifej, R. N. Al-Haddad, H. Tan, R. F. DeMara, "Layered Approach To Intrinsic Evolvable Hardware Using Direct Bitstream Manipulation Of Virtex II Pro Device," "Best paper of session and nominated best of conference" in *Proceedings of the 17th International Conference on Field Programmable Logic and Applications (FPL'07)*, Amsterdam, Netherlands, 27-29 August 2007, 2007.

- [53] S. A. Guccione, D. Levi, and P. Sundararajan, "JBits: A Java-based Interface for Reconfigurable Computing," in *Proceedings of Second Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD'99)*, Laurel, MD, USA, September 28-30, 1999.
- [54] S. Ghiasi, and M. Sarrafzadeh, "Optimal Reconfiguration Sequence Management [FPGA Runtime Reconfiguration]," in *Proceedings of Design Automation Conference Asia and South Pacific 2003*, Kitakyushu, Japan, January 21-24, 2003.
- [55] S. Ganesan, and R. Vemuri, "An Integrated Temporal Partitioning and Partial Reconfiguration Technique for Design Latency Improvement," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition 2000*, Paris, France, March 27-30, 2000.
- [56] S. Hauck; L. Zhiyuan; and E. Schwabe, "Configuration Compression for the Xilinx XC6200 FPGA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Volume: 18, Issue: 8, pp. 1107 – 1113, August 1999.
- [57] S. Vigander "Evolutionary Fault Repair of Electronics in Space Applications," Dissertation, Norwegian University Sci. Tech., Trondheim, Norway, February 28, 2001.
- [58] T. Kwok, and Y. Kwok, "On the Design of a Self-Reconfigurable SoPC Cryptographic Engine," in *Proceedings of 24th International Conference of the Distributed Computing Systems Workshops*, Hachioji, Tokyo, Japan, March 23-24, 2004.
- [59] U. Kimmo, T. Matti, and O. Jorma, "A Compact MD5 and SHA-1 Co-Implementation Utilizing Algorithm Similarities," in *Proceeding of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA 2005)*, Las Vegas, Nevada, USA, June 27 - 30, 2005.
- [60] W. Mak, and E.F.Y. Young, "Temporal Logic Replication for Dynamically Reconfigurable FPGA Partitioning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Volume: 22, Issue: 7, pp. 952 – 959, July 2003.
- [61] W. Zhang, N. K. Jha, and L. Shang, "NATURE: A Hybrid Nanotube/CMOS Dynamically Reconfigurable Architecture," in *Proceedings of IEEE Design Automation Conference (DAC06)*, San Francisco, CA, USA, July 24 – 28, 2006.
- [62] Y. E. Krasteva et al, "Flexible Core Reallocation for Virtex II Structure," in *Proceedings of Engineering of Reconfigurable System and Algorithm (ERSA 05)*, Las Vegas, Nevada, USA, June 27-30, 2005.
- [63] Y.K. Kang, D.W. Kim, T.W. Kwon, and J.R. Choi, "An Efficient Implementation of Hash Function Processor for IPSEC," in *Proceedings of the 2002 IEEE Asia-Pacific Conf. on ASIC (AP-ASIC 2002)*, Taipei, Taiwan, August 6 – 8, 2002.

- [64] Federal Information Processing Standards, "Secure Hash Standard," FIPS PUB 180-2, August 1, 2002.
- [65] Altera, Inc., "Stratix Device Handbook," v1-3.4, 2006.
- [66] Atmel, Inc., "5K - 50K Gates Coprocessor FPGA with FreeRAM," July 2006.
- [67] Jungo, Inc., "WinDriver PCI/ISA/CardBus v7.01 User's Guide," 2005
- [68] Xilinx, Inc., "PlanAhead Methodology Guide," v8.2, 2006.
- [69] Xilinx, Inc., "Virtex-II Pro Platform FPGA User Guide," v2.4, August 2004.
- [70] Xilinx, Inc., "Virtex-II Pro™ Platform FPGA Documentation," v1.0, Jan. 2002
- [71] Xilinx, Inc., "Virtex-II Platform FPGA User Guide," v1.7, February 2004.
- [72] Xilinx, Inc., "Two Flows for Partial Reconfiguration: Module Based or Difference Based," v1.1, November 2003.
- [73] Xilinx, Inc., "Using a Microprocessor to Configure Xilinx FPGAs via Slave Serial or SelectMAP Mode," v1.4, November 2003.
- [74] Xilinx, Inc., "Parallel Cable IV Connects Faster and Better," *Xcell Journal*, Spring 2002.
- [75] Xilinx, Inc., "ChipScope Pro Software User Guide," v6.3.1, October 2004.
- [76] Xilinx, Inc., "Constraints Guide," v6.3, October 2004.
- [77] Xilinx, Inc., "User Core Templates Reference Guide," v1.2, April 2003
- [78] Xilinx, Inc., "Libraries Guide," v6.3i, Aug, 2004
- [79] Xilinx, Inc., "Development System Reference Guide," v3.5.1, April, 2003
- [80] Xilinx, Inc., "In-System Programming Using an Embedded Microcontroller," v3.1, April, 2003