uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Arkan Khalaf

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.A.Sc. (Electrical Engineering)

GRADE / DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

A Self-Reconfigurable Platform for Built-in-self-test Applications

TITRE DE LA THÈSE / TITLE OF THESIS

V. Groza

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

T. Kwasniewski

A. Nayak

Gary W. Slater

Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

A SELF-RECONFIGURABLE PLATFORM FOR BUILT-IN-SELF-TEST APPLICATIONS

by

Arkan Khalaf

A thesis submitted to the faculty of Graduate Studies and Postdoctoral Studies in partial fulfillment of the requirements for the degree of Master of Applied Science, Electrical Engineering

2006

Ottawa-Carleton Institute for Electrical and Computer Engineering School of Information Technology and Engineering
University of Ottawa
Ottawa, Ontario, Canada

Your file Votre référence
ISBN: 978-0-494-49223-9
Our file Notre référence
ISBN: 978-0-494-49223-9

Abstract

This thesis introduces a novel architecture of a run-time reconfigurable microsystem on chip (SoC). This system consists of a logic block that can be reconfigured at run time, and an embedded multi-microprocessor system that connects to this logic block and can reconfigure it at run time using special resources of Field Programmable Gate Arrays (FPGA). A design flow for run-time reconfigurable logic circuits has been developed and is presented in the context of the implementation of the SoC on a FPGA.

This reconfigurable architecture is validated by an application that implements the novel idea of verifying algorithms for testing digital circuits by using run-time reconfigurable techniques, in order to minimize circuit area, as well as test generation and application time. The idea revolves around the dynamic partial reconfiguration of circuits under test, in order to inject stuck-at faults at different locations of the circuit, to verify for and uncover logic structural faults.

The thesis presents the design and implementation of a self-reconfigurable platform, where faults are injected at run-time to the circuit under test. It analyzes the ways of injecting faults and the run-time reconfiguration overhead associated with it, while the rest of the circuit is present on the reconfigurable architecture, in order to validate run-time reconfigurable built-in-self-test techniques, as compared to the more traditional methods.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank Dr. Voicu Groza for his ongoing support, and patience throughout the research period, as well as for Dr. Rami Abielmona, whom his support and dedication to the ERACE project can not be forgotten.

My special thanks to Xilinx Incorporated and Canadian Microelectronics Corporation (CMC) for their hardware and software donations.

I would like to thank also my colleagues at the ERACE project for their support and encouragement. The same is for Dr. R. Habbash, from Ottawa University, and my friends for their support and encouragement.

Lastly, I would like to thank my wife and two kids for their support and patience. I dedicate this thesis to them.

Chapter 1: Introduction

The emergence of reconfigurable computing (RC), as was first defined at UCLA [1], presents a new paradigm for system design. Reconfigurable computing systems combine programmable hardware with programmable processors to benefit from the best of hardware and software. In one of its many forms, it provides the combination of the microprocessor handling of all real-time deadlines and a reconfigurable co-processor allowing for the different configuration in and out of the reconfigurable hardware as they are needed during program execution, performing what is called as run-time reconfiguration (RTR) of the hardware [2]. The hardware mentioned is Field Programmable Gate Arrays or FPGA as widely known. The introduction of bigger FPGAs with faster reconfiguration times and partial reconfiguration support led to new and diversified applications for reconfigurable computing.

The increasing complexity of integrated circuits (IC) has increased the probability of fault occurrences in digital systems. Testing and design for testability became more important, and thus verification and testing are being the major components of design and manufacturing costs of a new product.

Fault-injection techniques [4] are one of the techniques used to validate the dependability of a system. Introducing stuck-at-0 and stuck-at-1 faults will inform us of detectable and undetectable, or hidden, faults. The former is a measure of how well the circuit responds to the injection of faults, but the latter is a measure of how reliable the circuit is, since a hidden fault can give the impression that the circuit is correctly functioning, while in reality, it is not. These fault injection techniques for testing digital systems have shown to be very efficient in studying circuit behaviour when faults occur.

Fault injection methods usually applies once the system or circuit is available, however, several authors recently showed the benefits of fault-injection testing earlier in the digital design flow,

mainly at the design entry level (i.e. directly to the HDL models), where simulations are used to evaluate the impact of the faults on the circuit behaviour [3]. The latter technique is very useful, however, the main drawback related to the use of simulation is that the amount of simulation time required to process complex circuits, increases dramatically with the number of injected faults [4].

With the availability of bigger and faster FPGAs, the next logical step was to speed up fault simulation by using them as a test prototype medium (emulation). This process usually requires the user to fully synthesize, place and route and map the circuit onto a particular FPGA device, which add to the overhead time for using such method.

1.1 Motivations and Original Contributions

This work is part of the ongoing Embedded Research Architecture for Co-design Environment (ERACE) project by the Group for Embedded MicroSystems (GEMS) research lab at the University of Ottawa. The research team developed a system-on-chip (SoC) that combines a microprocessor to handle real-time deadlines and a reconfigurable coprocessor that allows for the parallel execution of multiple hardware functional units. Given the long duration of the software verification of testing algorithms, and based on this SoC, I have implemented a self-reconfigurable system dedicated to the validation of testing techniques in the area of built-in-self-testing (BIST) of digital cores. I also devised a design flow of run-time self-reconfigurable systems for a class of FPGAs. This design flow was validated during the implementation of the architecture of the original ERACE system, and of the current testing dedicated self-reconfigurable system.

The thesis introduces several competing realizations of an algorithm used to catch both detectable and hidden faults within a Circuit Under Test (CUT). In this thesis, I present the sequential run-time reconfiguration (RTR) realization of the fault-model strategy, while utilizing the RTR technique to insert stuck-at faults at different wires of the circuit under test (CUT), and record the circuit's behaviour accordingly [5]. This work is an advanced extension to the previous work done by the same group that used a development framework for Xilinx FPGAs based on

the Java language, called the JBits SDK [6]. The work now extends to build an actual generic self reconfigurable platform that is used for BIST applications.

1.2 Outline of the Thesis

The thesis introductory section summarizes the motivation and the original contributions of the author. Chapter 2 gives a background introduction on run-time reconfigurable systems, partial reconfigurable devices and run-time reconfiguration techniques. There are presented partial reconfiguration principles and techniques along with a review of the available Xilinx flows. Chapter 3 describes the hardware and software development platforms that were used during implementation phase of this thesis. Chapter 4 presents the ERACE architecture that was conceived as a generic reconfigurable computing platform. Details on how to build the self reconfigurable platform that is to be used for our BIST applications are provided. Chapter 5 presents the Built-in-Self-Testing (BIST) methodology and describes the steps needed for a circuit under test (CUT) to be tested, including faults injections. Evaluation of BIST for one example logic circuits (C17) is presented as well. The final chapter (Chapter 6) presents conclusions and suggested future work.

Chapter 2: Reconfigurable Systems

2.1 Introduction

The paradigm of reconfigurable computing differs substantially from traditional approaches to digital logic design, as FPGAs define the core for this paradigm. FPGA reconfiguration can be divided in two major classes: Compile-Time Reconfiguration and Run-Time Reconfiguration. The two differ mostly in the matter a design is used; whether in a static way already fixed at the time of compilation or dynamically that is changed at run-time.

Run-time reconfiguration is usually performed by a software system that decides when to reprogram the FPGA and with what. The simplest kind of run-time software selects a precompiled circuit and transmits the programming data directly to the FPGA. A typical application would be to have a collection of pre-routed and placed circuits that reside in the main memory of the host system, ready for download onto the FPGA [35]. That was similar to the compile time reconfiguration in that both configure the whole FPGA.

As the FPGA were getting bigger, and with the ability to reconfigure any part(s) of the FPGA without affecting the rest of the FPGA, a new era of reconfigurable computing emerged. The system now has to manage, in a runtime environment, which hardware part to execute and when, and also to manage the reconfiguration of the reconfigurable parts within the FPGA. With the introduction of embedded processors cores, this runtime environment can be implemented on the FPGA itself, and thus reduce the reconfiguration overhead and frees the main processor for other applications.

In the following section, an overview of some relevant classification options can be used to characterize a component according to its specific properties or architecture. Next, different reconfiguration techniques that can be used to build a reconfigurable platform are clarified.

2.2 Reconfiguration Principles & Classifications

Any reconfigurable computing, in its basic definition, involves some form of customizing how the hardware uses a number of physical control points (joints). These control points can then be changed periodically in order to execute different applications using the same hardware [39]. Using this definition, not all the logic devices/technologies can be used in reconfigurable computing. A fixed hardware, such as ASICs, can be optimized for performance but is hardwired and its functionality can not be modified. Same for configurable devices such as OTP (one-time programming) fuse and anti-fuse devices (PLDs, SimplePLDs), where programming/erasing might take several hours.

2.2.1 Reconfiguration Technology

Most of the recent advances in reconfigurable computing are for the most part derived from the technologies developed for FPGAs. Most current FPGAs and reconfigurable devices are SRAM-programmable, meaning that SRAM cells are connected to the configuration points in the FPGA, and programming the SRAM bits configures the FPGA. Thus, these chips can be reprogrammed by the user as easy as standard static RAMs. As with SRAMs, the SRAM based FPGAs need to be reprogrammed every time they are powered up. Figure 1 presents and visualizes the SRAM configuration.

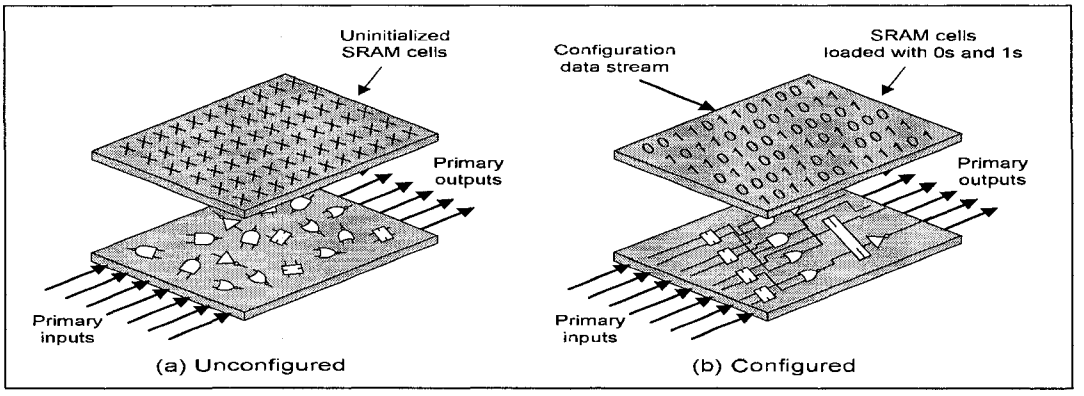


Figure 1 FPGA SRAM representation before and after being configured.

2.2.1.1 SRAM Cells

The SRAM cell is the basic element in the FPGA configuration and programming, and is shown in Figure 2 (left). To store data, 'sel' is set to logic 1 and data flow from left to right. After data stabilizes around the two NOT gates, 'sel' is set to logic 0, and data remains running as long as the cell is powered up. Figure 2 (right) shows an interconnect point (aka programmable routing switch) that is used to connect two wires inside the FPGA.

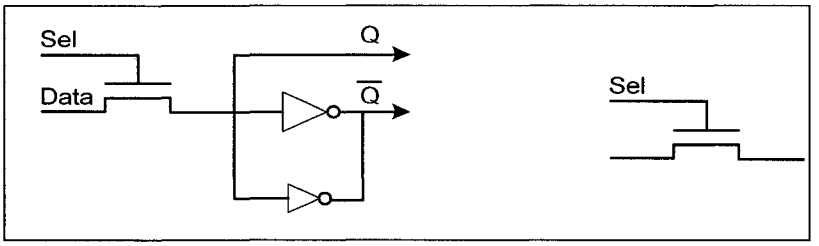


Figure 2 A programmable bit for an SRAM based FPGA (left) and a programmable interconnect (right)

2.2.1.2 FPGA Fabric

The combination of both the logical and routing resources makes what we call an FPGA fabric. FPGA wiring with programmable interconnect is slower than typical wiring in a custom chip for two reasons: the pass transistor and wire lengths.

Figure 3 shows a typical FPGA fabric. By programming the memory cells differently we get different logic functionality and different connections. FPGA wires are generally longer than would be necessary for a custom chip (ASIC). In a custom layout, a wire can be made just as long as necessary. In contrast, FPGA wires must be designed to connect a variety of logic elements and other FPGA resources based on the particular FPGA architecture. A logical connection between any two logical cells might require programming several interconnecting points.

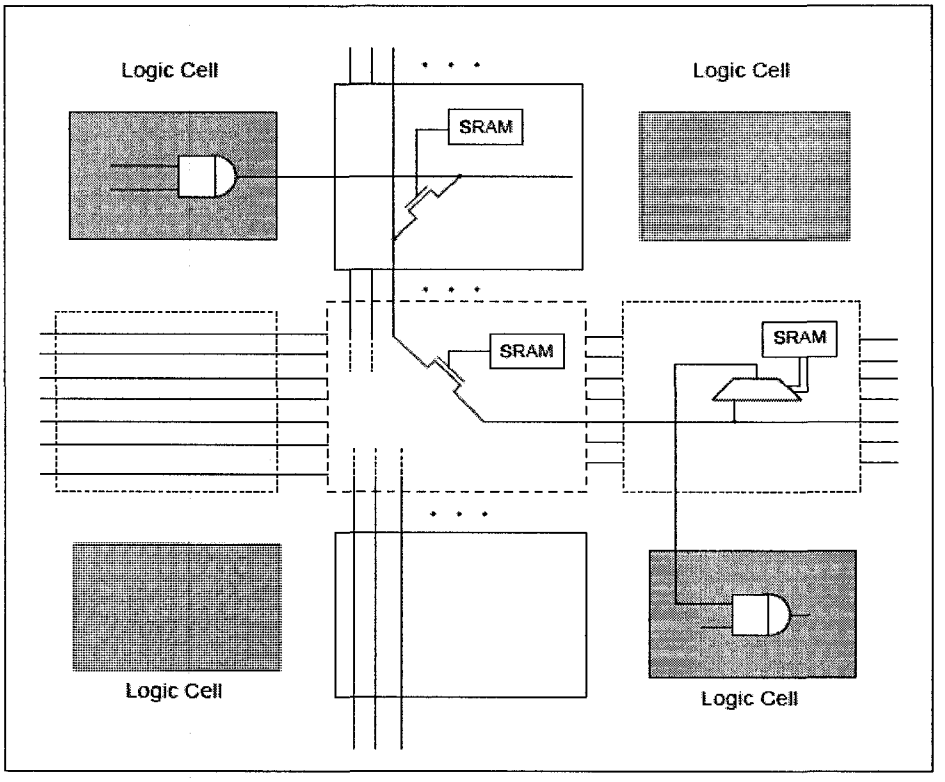


Figure 3 A typical SRAM based FPGA Fabric

2.2.2 *Configuration Memory Style*

There are three main configuration memory styles that are available to change the configuration data of the reprogrammable device (FPGA), as shown in Figure 4 [39]:

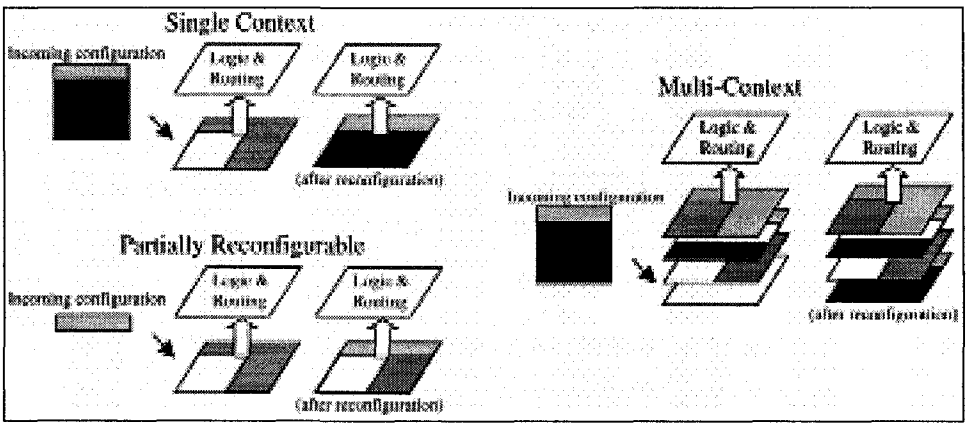


Figure 4 The different configuration memory style: Single context, multi-context and partially reconfigurable.

2.2.2.1 Single Context

A **single context** FPGA has only one configuration plane. A configuration plane is an SRAM memory plane in which the configuration data is programmed. The Logic & Routing plane is located on top of the configuration plane.

Single context devices are programmed using a serial stream of configuration information and only sequential access is supported. This means that when any change to a configuration is desired, a complete reprogramming of the entire chip is needed. An obvious disadvantage of this technique is that it will introduce a high reconfiguration overhead, especially when only minor changes to the present configuration are needed. On the other hand, an advantage is that the reconfiguration hardware can be held very simple and this makes the device cheap.

Most classic FPGAs are single context devices, e.g. the Altera Flex10K, Xilinx 4000 series, Lucent Orca series.

2.2.2.2 *Multi-Context*

A multi-context device can be considered as a multiplexed set of single context devices. Multiple memory bits are available for each programming bit location. Therefore this collection of memory bits can be thought of as multiple planes of configuration information [43].

One plane of configuration information can be active at a given moment, but the device can quickly switch between different planes. This switching can occur very fast, in the order of nanoseconds. The number of configuration planes one device has at its disposal is determined by the manufacturer and can differ from component to component.

Xilinx filed a patent on the multi-context programmable device in 1995 and presented the work as time-multiplexed FPGA. The patented device has architecture similar to the Xilinx XC4000E with multiple configuration planes [40]. The first commercial product that used this technique was the CS2000 Reconfigurable Communication Processor (RCP) series developed by Chameleon Systems Inc. It has a reconfigurable fabric with two configurable planes; one for executing while the other configures the next part of the application [41].

2.2.2.3 *Partially Reconfigurable*

In a partially reconfigurable FPGA, the underlying programming bit layer operates like a RAM device. Using addresses to specify the target location of the configuration data allows for selective reconfiguration of the array, while the undisturbed portions of the array may continue execution. Since only a part of the array is reconfigured at a given point in time, the entire array does not require reprogramming and valuable execution time is spared. Using this selective reconfiguration can greatly reduce the amount of configuration data that must be transferred to the FPGA at a time.

Commercially available products that support partial reconfiguration are: Virtex, Virtex-2 and Virtex-2 Pro from Xilinx, FPSLIC from Atmel, XPP64-A1 from PACT [42].

2.3 Architectures of Reconfigurable Computing Systems

As discussed before, the main characteristics of Reconfigurable Computing (RC) is the presence of hardware that can be reconfigured to implement specific functionality more suitable for specially tailored hardware than on a simple processor. RC systems join up microprocessors and programmable hardware, which is often called Reconfigurable Processing Unit (RPU), in order to take advantage of the combined strength of hardware and software [44]. Different RC architectures [39] exist based on the type of interconnection between the RPU and the host system as shown in Figure 5 . The following sections give a brief description of each.

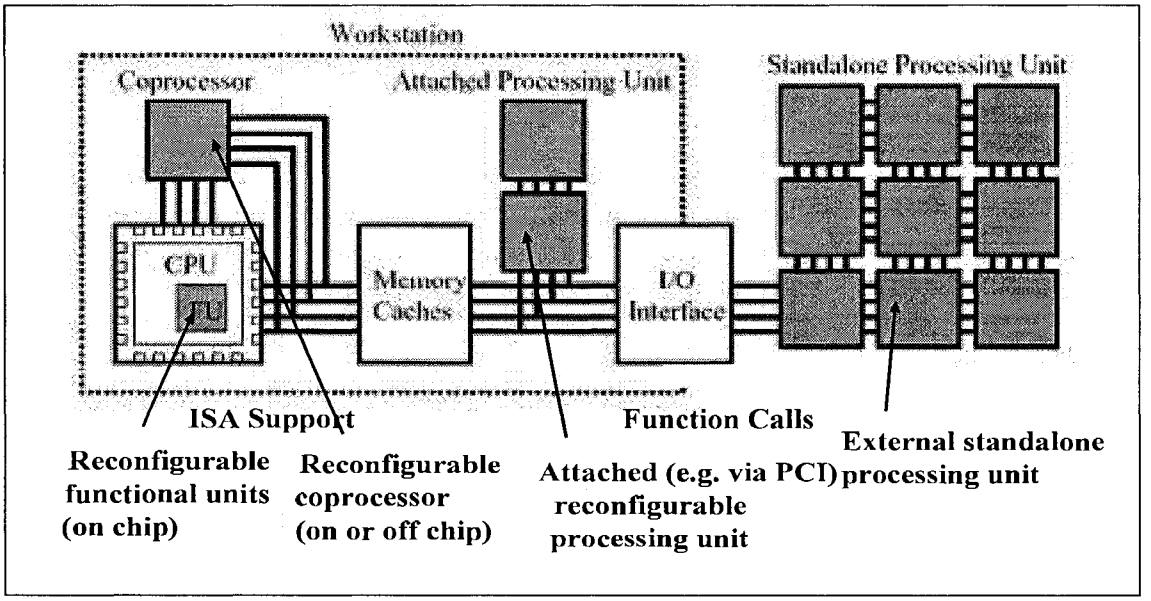


Figure 5 Different Host-RPU interconnections

2.3.1 External Stand-alone Processing Unit

The RECON system is a good example for such architecture [22] where the RPU is an external stand-alone. It consists of a SUN SparcStation host and a reconfigurable coprocessor board (the board exploits a XC4010 FPGA as RPU). The board is connected to the host processor through the PCI bus.

2.3.2 *Attached Processing Unit*

In this architecture, the RPU is attached to the host processor local (memory) bus instead of the slower I/O bus. A typical implementation of this architecture is the TKDM hardware environment [45], where the FPGA module (RPU) uses the DIMM (dual inline memory module) bus for high-bandwidth and low-latency communication with the host processor. The system's firmware is integrated with the Linux host operating system and offers functions for data communication and FPGA reconfiguration.

2.3.3 *Reconfigurable Functional Unit*

Most of the previous architectures separate the reconfigurable logic from their host processor, and thus suffer from communication bottlenecks. This architecture integrates the reconfigurable logic into the host processor itself. Several works have been done on integrating processors and reconfigurable logic (see the list in [46]), however, most of those systems use, in general, standard FPGA architecture that has not been designed to effectively support the need of integrated FPGA-processor systems.

The Chimaera System architecture [46], as shown in Figure 6, uses an architecture for its reconfigurable logic that is inspired by the Triptych FPGA [47] and the Altera Flex 8000 series. The reconfigurable Array logic gets its inputs directly from the host processor's register file, or a shadow register file which duplicates a subset of the values in the host's register file. The Chimaera system treats the reconfigurable logic as a cache for the Reconfigurable Functional Unit (RFU) instructions. Those instructions that have been recently executed, or that can be predicted to be needed soon, are kept in the reconfigurable logic.

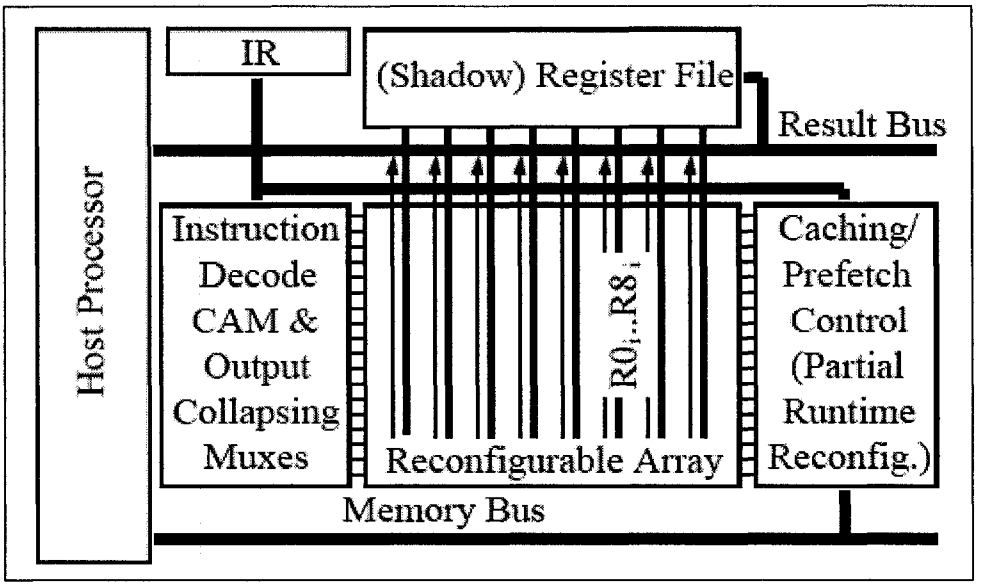


Figure 6 The overall Chimaera Architecture

The system uses partial run-time reconfiguration techniques to manage the reconfigurable logic that is symmetric so that a given instruction can be placed into the FRU wherever there is available logic.

2.3.4 Co-processor

To have an even better coupling with the host processor, the GARP [37] reconfigurable architecture defines the reconfigurable logic (called here reconfigurable array) as a slave computational unit located on the same die as the processor itself, as shown in Figure 7. This reconfigurable array can be tailored to perform certain computations that take too long to execute on the processor itself, and thus can act as a coprocessor to the main application CPU.

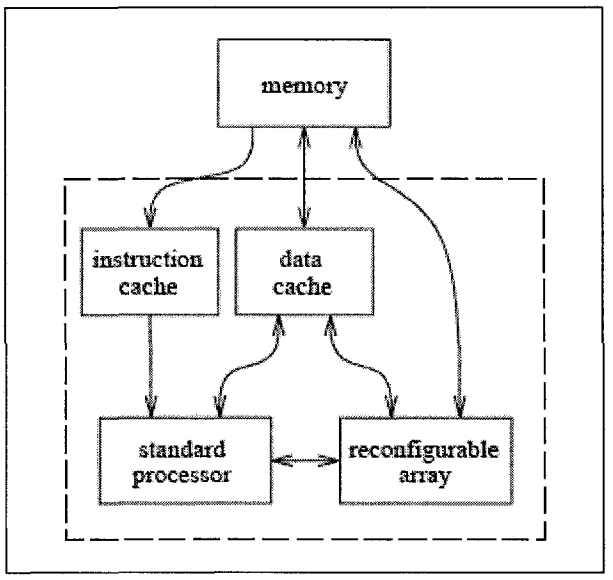


Figure 7 Basic Garp block diagram

Garp makes external storage accessible to the reconfigurable array by giving the array access to the standard memory hierarchy of the main processor, which also provides memory consistency between the two. Memory accesses can be initiated in the array without direct processor intervention. In such applications, the main processor initializes the configurable hardware and either directly sends the input data, or provides it with pointers to the required data. The reconfigurable array processes its tasks in parallel with the main processor, and returns the results after completion.

The Garp reconfigurable arrays is composed of entities called blocks which corresponds roughly to the CLBs of the Xilinx 4000 series.

2.3.5 Processor Embedded in Reconfigurable Fabric (ERACE)

The ERACE's system architecture makes use of the advance progress in partial FPGA self reconfiguration and the increasing FPGA size by having the host processor and the reconfigurable logic share the same FPGA fabric. Chapter 4 contains a detailed description of the ERACE architecture.

2.4 Partial Reconfigurable Devices: Virtex-II FPGAs

Partial Reconfiguration in its simplest form allows for the reconfiguration of only a part of a device while the rest of the device is running. One of the greatest advantages to using SRAM based FPGAs is the ability to modify the configured circuit at anytime. In the vast majority of past applications, however, this capability was used exclusively in the design phase of those projects. Much of the reason for this can be attributed to the lack of both hardware and software support for partial reconfiguration [10]. The introduction of the Xilinx XC6200 family of devices [11] changed this, as it featured an open architecture with all the circuit configuration data exposed to the users. Xilinx, however, chose to stop the development work on its XC6200 line of partially reconfigurable FPGAs and offer most of its feature in its next generation, known as the Virtex [12].

The Virtex-II family [18] is a platform FPGA developed for high performance from low-density to high-density designs that went into full production in mid 2001. It was the first Xilinx family that contained the Internal Configuration Access Component Port (ICAP), and thus makes it ideal for run-time reconfiguration (RTR) applications.

As shown in Figure 8, the programmable device is comprised of input/output blocks (IOBs) and configurable logic blocks (CLBs). Each CLB has internal fast interconnect lines and connects to a switch matrix to access general routing resources. CLB resources include four slices and two 3-state buffers.

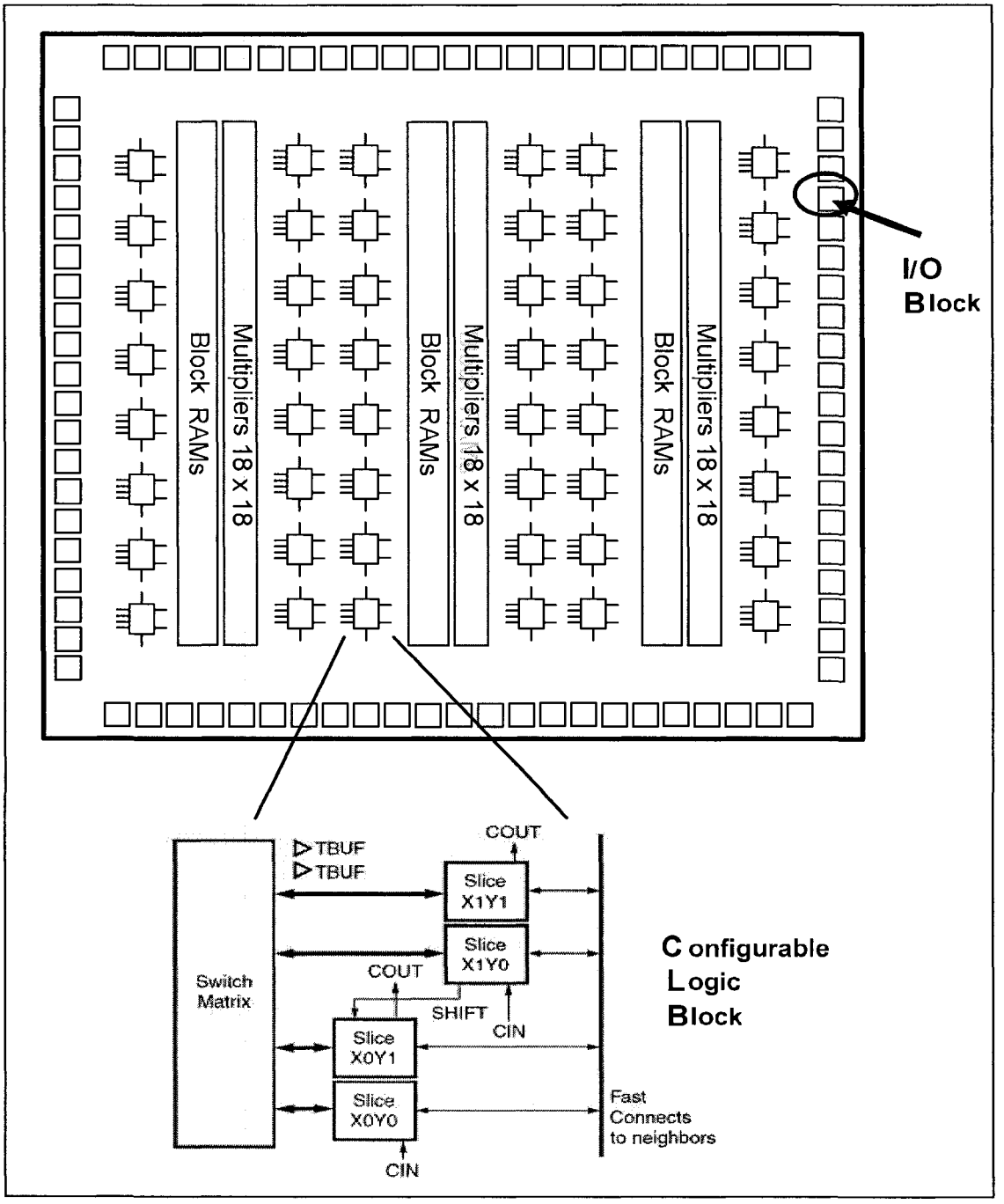


Figure 8 Virtex-II Architecture

The four slices are identical and each contains:

- Two function generators (F & G)
- Two storage elements
- Arithmetic logic gates
- Two 2-input logic multiplexers (MUXF5 and MUXFX).

As shown in Figure 9 [18], the two function generators (F & G) can be configured as 4-input look-up tables (LUTs), as 16-bit shift registers, or 16-bit distributed RAM memory. Only LUTs mode configuration is allowed during RTR, as reading or writing to columns which contain function generators configured as shift registers, and/or RAM modes, can cause corruption of data in those elements [28].

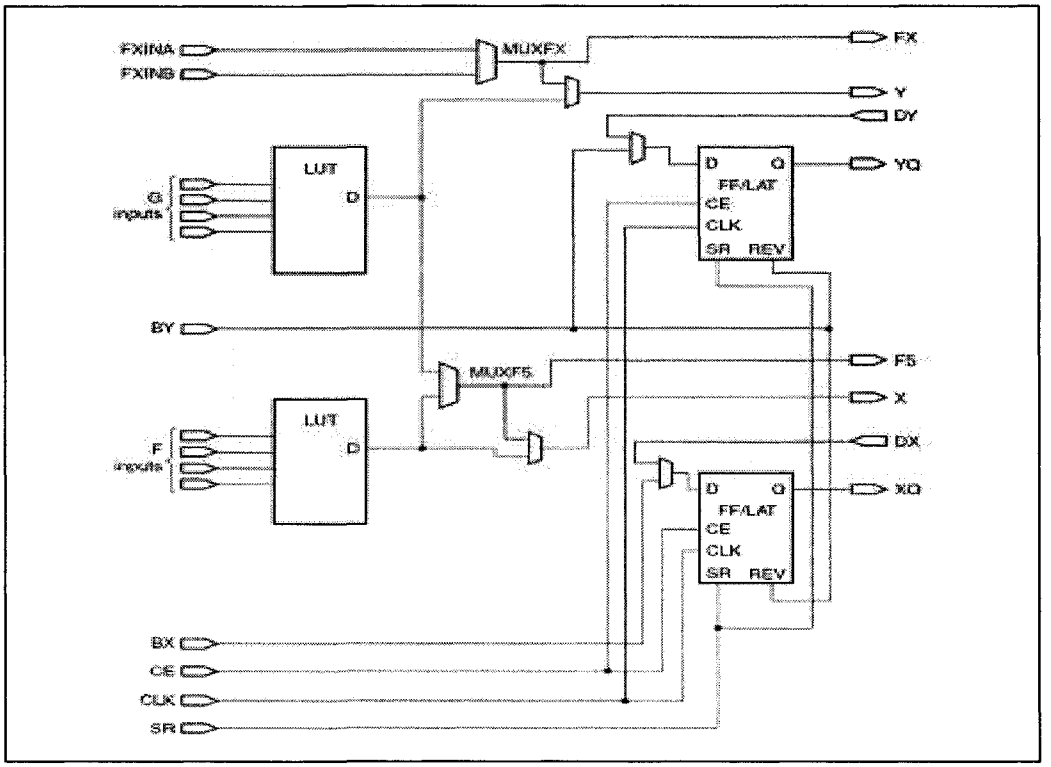


Figure 9 Virtex-II Slice Diagram

Look-up tables (LUT) are the primary elements for implementing combinational logic in Virtex-II FPGAs. Each LUT can implement any arbitrarily defined Boolean function of four inputs or less. Five-Input functions can be implemented using two LUTs, within the same slice, by using the MUXF5 primitive. Using multiplexers MUXF6, MUXF7 and MUXF8 allows for combining 4 and 16 LUTs which yield to implement winder-input functions. MUXFX implements MUXF6, MUXF7 or MUXF8 according to the position of the slice in the CLB and as shown in Figure 10 [18].

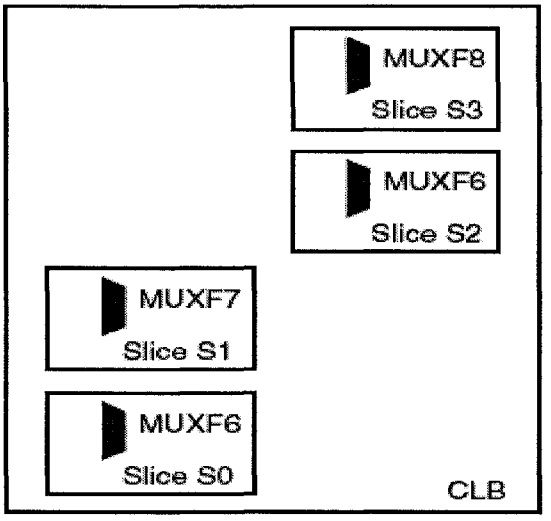


Figure 10. MUXFX implementation within each slice of a CLB

2.5 Partial Reconfiguration Flows

Xilinx specifies more than one method for partially reconfiguring an FPGA. In all the methods, the static part will resemble the intact logic, while the active part is the piece of logic being reconfigured. In this section, we will explain the different ways to achieve this on Xilinx Virtex-II architecture FPGAs.

2.5.1 *Module Based Partial Reconfiguration*

This method follows the Xilinx Modular Design methodology guidelines [14]. For module-based partial reconfiguration, additional guidelines are required in addition to the Xilinx Modular Design methodology guidelines [15] [16]:

- The reconfigurable module height is always the full height of the device.
- The minimum reconfigurable module width is four slices (one CLB).
- All logic resources located within the width of the active module are considered part of the reconfigurable module's bitstream frame. This includes, tri-state buffers (TBUFs), block RAMs (BRAMs), multipliers, input/output blocks (IOBs), and all routing resources.
- A reconfigurable module's boundary cannot be changed. The position and region occupied by any single reconfigurable module is always fixed.
- Bus Macros are required between the active and static modules of the design (when communication is needed between active and static modules). No signals going to or from an active module will load or source any element in another module without first passing through a bus macro
- There should be no unconstrained top-level logic.

Figure 11 [14] shows a typical design with more than one reconfigurable module (Partial Reconfigurable logic) that interfaces with the fixed logic through Bus Macros.

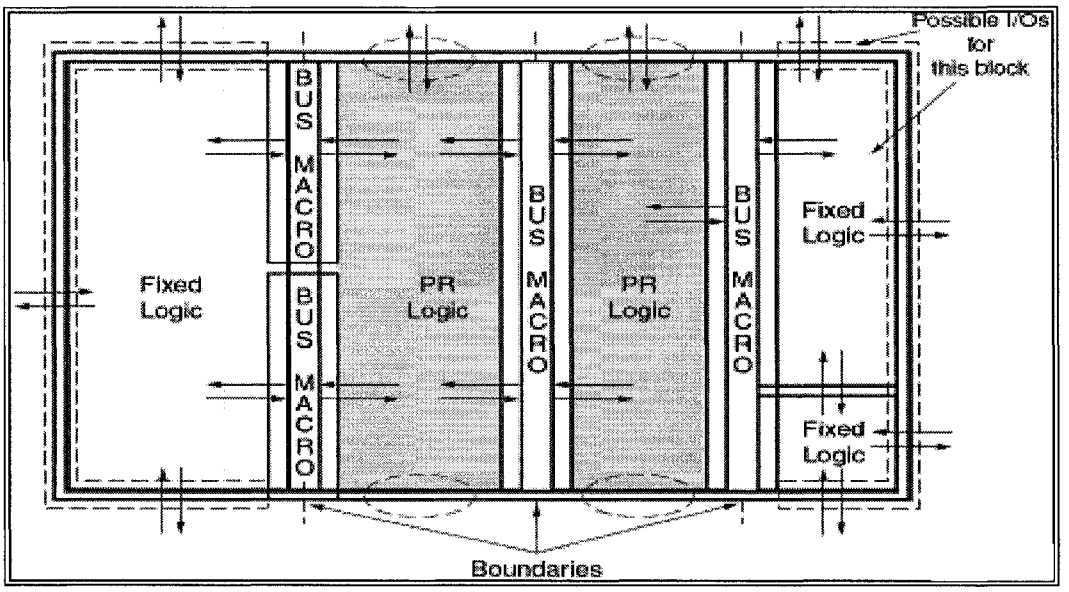


Figure 11 FPGA Design with Reconfigurable Modules

2.5.1.1 Bus Macros

Bus Macros (BMs) are predefined physical routing bridges that provide a means of locking the routing between the static and the active modules, and thus making the active modules pin compatible with the static one. As a result, all connections between the active and the static modules must pass through a bus macro, with the exception of the clock signal (global signals, GND and VCC are handled automatically by the tools). Bus Macros are usually provided as pre-routed hard micros (.nmc files) that will not change from compilation to compilation. They are implemented using either 3-state buffers or slices. We chose the slices bus macros as they give higher concentration of communication bits per CLB.

There are several types of slices bus macros that are available:

- Signal direction (left_to_right and right_to_left)
- Physical width of the bus macro (wide – 4CLBs, or narrow – 2 CLBs wide)
- Whether signals passing through the bus macro are registered or not (synchronous or asynchronous)

All bus macros, regardless of direction or physical width, provide eight bits of data bandwidth in either direction (left-to-right or right-to-left) depending on the type of the bus macro, as shown in Figure 12.

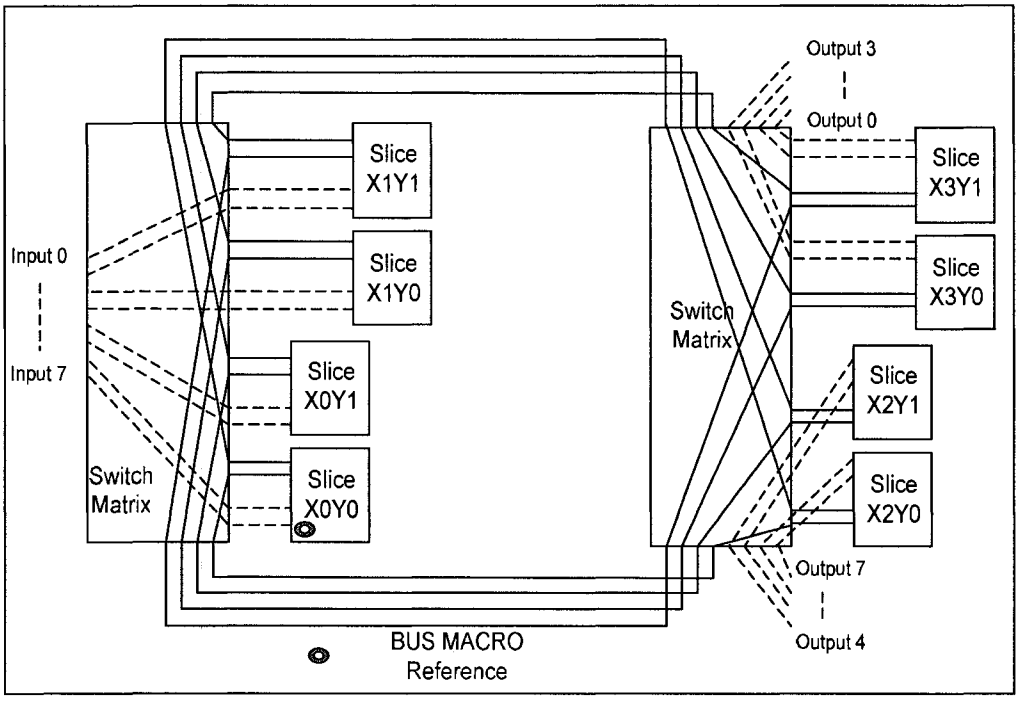


Figure 12 Bus Macro Detailed View (Left to Right)

The bus macros are placed on the active-static boundaries, with four slices on the active side, and the other four on the static side. The bus macros must be locked in exactly the same position for all compilations. More bus macros can be instantiated in the design, as needed.

2.5.1.2 Partial Bitstream Generation

A partial bitstream is generated for each active module of the FPGA. Multiple bitstreams can be generated for every partially reconfigurable module variation. Partial bitstreams are indistinguishable from full bitstreams and are generated using the same Xilinx's BitGen. The only difference is that the *–g ActiveReconfig:Yes* option is required for partial reconfiguration, meaning

that the device remains in full operation while the new partial bitstream is downloaded. Failing to utilize this command will assert the global set reset (GSR) during configuration, resetting the entire design.

After initial device configuration with a full bitstream, partial bitstreams can be loaded into the FPGA to swap the active module(s). Partial bitstreams contain all configuration commands and data necessary for configuring only the columns that are specified as part of the active module area. The task of loading a partial bitstream into an FPGA does not require knowledge of the physical location of the active module because configuration frame addressing information is included in the partial bitstream.

2.5.2 Difference Based Partial Reconfiguration

This approach is based on the ability of BITGEN Xilinx utility to produce a partial bitstream that only programs the difference between the original design and the new modified one. The resulting partial bitstream can be much smaller than the original depending on the design changes which can be as simple as a different LUT programming, or requires changes made on the front –end (HDL). Most of the simple design changes can be made easily by directly editing the routed NCD file in Xilinx FPGA editor.

2.5.3 Self-Reconfiguration

Self-reconfiguration extends the concept of partial reconfiguration. It assumes that specific logic on the static part is used to control the reconfiguration of the active part(s). One form of self-reconfiguration is where an FPGA can reconfigure itself under the control of an embedded microprocessor. To achieve this, both an internal reconfiguration interface and the necessary SW API to drive this interface are required.

2.5.3.1 Xilinx Internal Configuration Access Port (ICAP)

Virtex-II architecture features the ICAP interface that provides a configuration interface to the FPGA fabric. The ICAP interface, located in the lower-right corner of the FPGA, is similar to the external SelectMAP interface, but it is accessible from general interconnect, and thus

available to the internal logic resources, rather than the device pins. To use ICAP, the ICAP_VIRTEX2 primitive must be instantiated in the user design. Figure 13 [28] shows the ICAP_VIRTEX2 primitive.

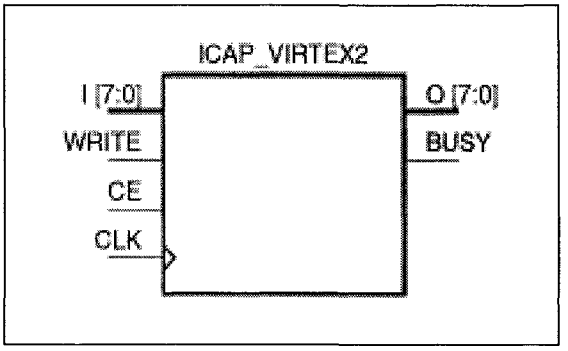


Figure 13 ICAP_VIRTEX2 Primitive

The control pins to the ICAP are as follows:

/CE - ***Port enable for write/read with programmable input inversion active low.***

/WRITE - ***Allow writing/reading to/from configuration port (write=1, read=0) with programmable input inversion; it is active low.***

The input pins are:

/CCLK - ***Configuration Clock. Port enable for write/read.***

/DI[7:0] - ***Data Input Bus..***

The output pins are:

/DO[7:0] - ***Data Output Bus..***

/BUSY - Same as SelectMAP.

The Hardware ICAP (HWICAP) core for the On-Chip Peripheral Bus (OPB) is build on the ICAP module, and enables the embedded microprocessor (MicroBlaze in our case) to access the FPGA configuration memory at run-time, at the clock rate of the OPB bus [28].

The HWICAP uses one block of dual-port RAM (16k bits), enough to cache and hold a single frame of the biggest Virtex-II device (XC2VP125 has a frame length of 11k bits). The provision for the cache memory, within the HWICAP core, is necessary to reduce system bus communication during FPGA configuration accesses using the ICAP interface. Figure 14 shows how the ICAP module is wrapped inside the HWICAP.

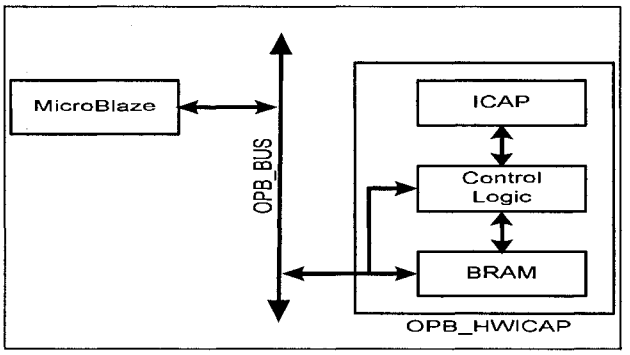


Figure 14 HWICAP Block Diagram

A more detailed block diagram of the HWICAP is shown in Figure 15 [28].

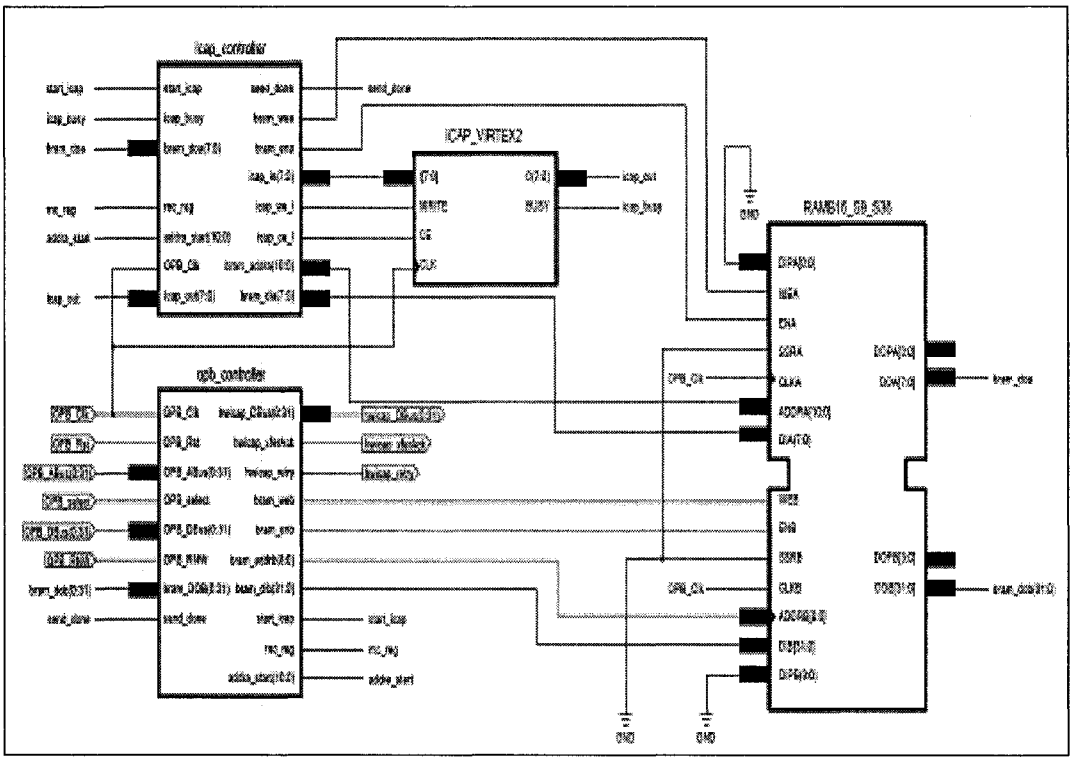


Figure 15 HWICAP Detailed Block Diagram

2.5.3.2 Software Application Programming Interface (API)

Xilinx provided the software API definitions of the HWICAP module. These function calls are necessary to have read/access the internal configuration resources of the FPGA. These function calls are listed in Table 1.

Functions	Description
XHwICAP_Initialize()	Initialize a XHwIcap instance.
XHwICAP_DeviceWrite()	Writes bytes from the storage buffer and puts it in the device (ICAP).
XHwICAP_DeviceRead()	Reads bytes from the device (ICAP) and puts it in the configuration storage buffer.
XHwICAP_StorageBufferWrite()	Stores data in the configuration storage buffer at the specified address.
XHwICAP_StorageBufferRead()	Read data from the specified address in the configuration storage buffer.
XHwICAP_DeviceWriteFrame()	Writes one frame from the configuration storage buffer and puts it in the device (ICAP).
XHwICAP_DeviceReadFrame()	Reads one frame from the device and puts it in the configuration storage buffer.
XHwICAP_CommandDesync()	Sends a DESYNC command to the ICAP port.
XHwICAP_SetClbBits()	Sets bits contained in a Center tile specified by the CLB row and col coordinates.
XHwICAP_GetClbBits()	Gets bits contained in a Center tile specified by the CLB row and col coordinates.

Table 1. HWICAP API

On-Chip reconfiguration is accomplished using a read-modify-write mechanism. As an example, to modify the contents of an LUT, the following steps are needed:

1. Calculate the target configuration frame.
2. Find the LUT bits in the target frame.
3. Read the target configuration frame from the device and put it in the storage buffer.
4. Modify the LUT bits in the storage buffer using XHwICAP_StorageBufferWrite().
5. Reconfigure the device with new LUT bits using XHwICAP_DeviceWriteFrame()

The steps above seems straightforward, however, what is missing is the relationship between the LUT bits and the frame. Xilinx consider this proprietary information that will not share (Xilinx support case # 627098). Instead, Xilinx added two extra functions (XHwICAP_SetClbBits() & XHwICAP_GetClbBits()) that allows all the above in one function call.

The source code for the XHwIcap_SetClbBits and XHwIcap_GetClbBits functions are not included in the code provided. These functions are delivered as ".o" files: "xhwicap_get_clb_bits_mb.o" and "xhwicap_set_clb_bits_mb.o". Xilinx's Libgen tool uses those ".o" files for the target processor [31].

2.6 Virtex-II FPGA Configuration Methods

SRAM based FPGAs, like the Virtex-II family, are configured, fully or partially, by loading application specific data into memory cells known collectively as the configuration memory These memory cells define the LUT equations, signal routing, IOB voltage standards and all other aspects of user design. The configuration memory is not cleared when reconfiguring the device through the ability to perform partial reconfiguration. However, this memory is volatile and must be configured at power-up.

Configuration is carried out using a subset of the device pins, some of which are dedicated, while others can be used as general-purpose input and output after the configuration is complete [18]. Depending on the system design, several configuration methods are selectable via the dedicated mode pins (M2, M1 and M0). Other dedicated pins are:

- CCLK - the configuration clock pin.
- DONE – configuration status pin.
- TDI, TDO, TMS, TCK – boundary-scan pins.
- PROG_B – configuration reset pin.

The configuration memory is distributed throughout the device and organized as a shift register. Data are serially loaded, that is why the configuration time is long and depends on the device size. Multiplying the number of shift registers can reduce this time and allows for parallel loading of the configuration bit stream. Several configuration modes are supported on Xilinx Virtex-II FPGAs as shown in Table 2.

Configuration Mode	M2	M1	M0	CCLK Direction	Data Width	Daisy Chain
Master Serial	0	0	0	Out	1	Yes
Slave Serial	1	1	1	In	1	Yes
Master SelectMAP	0	1	1	Out	8	No
Slave SelectMAP	1	1	0	In	8	No
Boundary Scan (JTAG)	1	0	1	N/A	1	No

Table 2 Virtex-II Configuration modes and pins settings

Programming is carried out either using Flash in-system programmable configuration PROMS (with JTAG interface), or externally, using special dedicated cables that connects the host PC to the JTAG connection of targeted system, such as the Parallel Cable IV and Platform USB cables. As listed in Table 2 some configuration methods allows for daisy chaining of more than one FPGA.

2.7 Virtex-II Configuration Memory Details

To program the configuration memory, instructions for the configuration control logic and data for the configuration memory are provided in the form of a bitstream. The bitstream is delivered to the device via the JTAG, SelectMAP, Serial or ICAP configuration interface.

For the Virtex family, Xilinx application note 151 [21] describes in a detailed way the configuration bitstream, specifying the position of LUT contents on the bitstream. However, for the Virtex-II family this information is not available and just a limited bitstream description can be found in Xilinx's documentation [18]. One reference suggests such a relation by going through the bitstream files [17].

2.7.1 Configuration Memory: Columns and Frames

Virtex-II configuration memory is arranged in vertical frames that are one bit wide and stretch from the top edge of the device to the bottom. These frames are the smallest addressable segments of the Virtex-II configuration memory space; therefore, all operations must act on whole configuration frames. Configuration memory frames do not directly map to any single piece of hardware, rather, they configure a narrow vertical slice of many physical resources [18].

The length of a Virtex-II frame is not fixed and depends on the size of the device, as shown in Table 3 [18]. However, the number of frames per column type is constant for all devices.

Device	Frames	Frame Length (32-bit Words)	Configuration Bits	Default Bitstream Size
XC2V40	404	26	336,128	338,976
XC2V80	404	46	594,688	598,816
XC2V250	752	66	1,588,224	1,593,632
XC2V500	928	86	2,553,856	2,560,544
XC2V1000	1104	106	3,744,768	4,082,592
XC2V1500	1280	126	5,160,960	5,170,208
XC2V2000	1456	146	6,802,432	6,812,960
XC2V3000	1804	166	9,582,848	10,494,368
XC2V4000	2156	206	14,212,352	15,659,936
XC2V6000	2508	246	19,742,976	21,849,504
XC2V8000	2860	286	26,174,720	26,194,208

Table 3 Virtex-II Frame Count, Frame Length, and Bitstream Size.

Table 4 [18]shows the number of columns per device and the number of frames per column. Of most concerns to us here is the number of CLB configuration columns as it programs the configurable logic blocks (including LUTs). The number of CLB configuration columns matches the number of physical columns in the device.

Column Type: →	IOB		IOI		CLB		BRAM		BRAM Interconnect		GCLK	
Device: ↓	Columns per Device:	Frames per Column:	Columns per Device:	Frames per Column:	Columns per Device:	Frames per Column:	Columns per Device:	Frames per Column:	Columns per Device:	Frames per Column:	Columns per Device:	Frames per Column:
XC2V40	2	4	2	22	8	22	2	64	2	22	1	4
XC2V80	2	4	2	22	8	22	2	64	2	22	1	4
XC2V250	2	4	2	22	16	22	4	64	4	22	1	4
XC2V500	2	4	2	22	24	22	4	64	4	22	1	4
XC2V1000	2	4	2	22	32	22	4	64	4	22	1	4
XC2V1500	2	4	2	22	40	22	4	64	4	22	1	4
XC2V2000	2	4	2	22	48	22	4	64	4	22	1	4
XC2V3000	2	4	2	22	56	22	6	64	6	22	1	4
XC2V4000	2	4	2	22	72	22	6	64	6	22	1	4
XC2V6000	2	4	2	22	88	22	6	64	6	22	1	4
XC2V8000	2	4	2	22	104	22	6	64	6	22	1	4

Table 4 Virtex-II Column Types and Frame Counts.

For the FPGA device used in the Multimedia card (XC2V2000), there are a total of 1456 frames. Each frame size is thus 4672 bits.

Xilinx, with the release of their latest Virtex-4, changed the way its configuration memory is structured by fixing the frame size for all the devices [26]. As frames are the smallest addressable segments when configuring the FPGA, this will fix the time required for configuring a frame across all the devices. Table 5 [26] shows some of Virtex-4 devices configuration frames counts and sizes.

Xilinx continued to use the same structure for the configuration memory in its latest Virtex-5 FPGA devices [38]. One extra note is the use of 6-input LUTs in these new devices.

Device	Non-Configuration Frames	Configuration Frames	Device Frames	Frame Length (words)	Configuration Array Size (words)	Configuration Overhead (words)
XC4VLX15	140	3,600	3,740	41	147,600	1312
XC4VLX25	234	6,022	6,256	41	246,902	1312
XC4VLX40	376	9,548	9,924	41	391,468	1312
XC4VLX60	536	13,868	14,404	41	568,588	1312
XC4VLX80	710	18,290	19,000	41	749,890	1312
XC4VLX100	948	24,184	25,132	41	991,544	1312
XC4VLX160	1,260	31,840	33,100	41	1,305,440	1312
XC4VLX200	1,572	38,536	40,108	41	1,579,976	1312

Table 5 Virtex-4 Frame Count, Frame Length Bitstream size.

Chapter 3: S/W and H/W Development Platforms

This chapter explains the Software and Hardware platforms used in the implementation.

3.1 Design Environment Details

The following work was performed utilizing v7.1 of both the EDK and ISE tools. At the time of the research, a new set of tools were available (v8), however, there were several service packs releases and we chose to stay with the more stable version. The prototyping platform was the Xilinx Multimedia board, which carries XC2V2000-FF896-4 Virtex-II FPGA. The design station was the IBM IntelliStation Z Pro, which processes a 3.6 GHz Intel Xeon processor with 2MB L2 cache and 2 GB of system memory.

3.1.1 ISE

ISE is Xilinx's Integrated Software Environment (ISE) FPGA tools. It is a complete graphical environment that can handle the various design stages from entry going through all the implementation phases to create the bitstream programming file. However, this tool doesn't support yet modular programming, thus, all the implementation stages were done using Xilinx's stand alone tools (NGDBUILD, MAP, PAR, BITGEN) run by a batch script. The release version we used is v7.1.04i.

3.1.2 EDK

The Embedded Development Kit (EDK) bundle is an integrated software solution for designing embedded processing systems. This pre-configured kit includes the Platform Studio tool suite as well as all the documentation and IP required for designing Xilinx Platform FPGAs with embedded MicroBlaze processors. All the peripherals we used came from the IP library provided by Xilinx, except the SnoopP core which was custom built [19]. Version V7.1.02i was used with the ISE tool set.

3.2 MicroBlaze Core

The MicroBlaze soft processor core is a 32-bit Reduced Instruction Set Computer (RISC) optimized for implementation in Xilinx FPGAs. Figure 16 shows a block diagram of the MicroBlaze core.

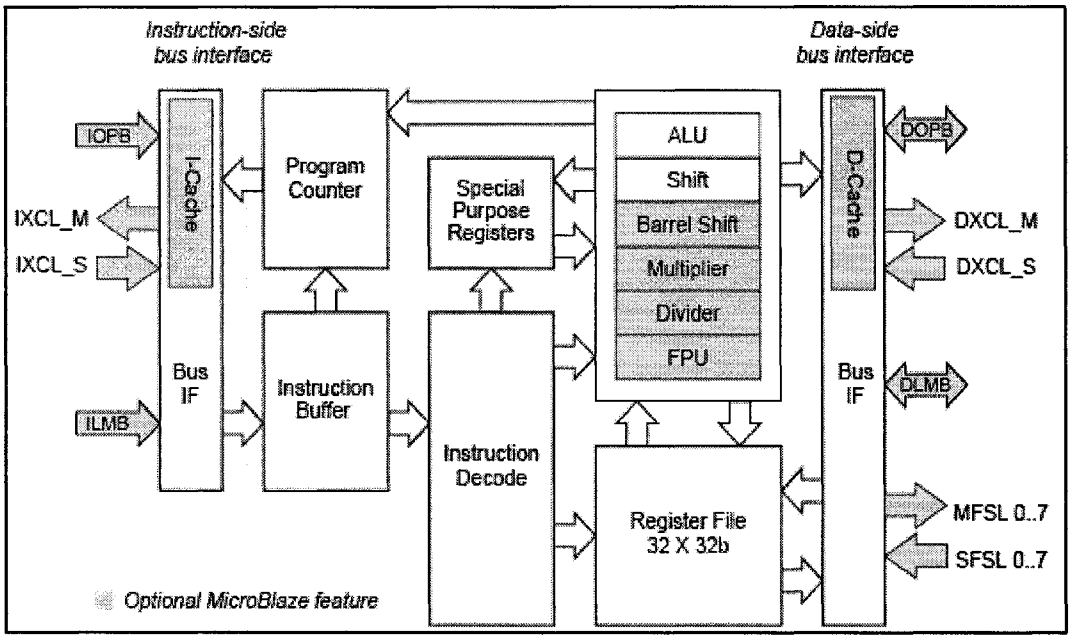


Figure 16. MicroBlaze Core Block Diagram.

The MicroBlaze soft processor is central to the MicroBlaze embedded system. It supports several bus interfaces. For a complete detailed description, please see [36].

3.2.1 On-Chip Peripheral Bus (OPB) interface

The On-Chip Peripheral Bus (OPB) interface is a 32-bit fully synchronous interface, based on the IBM's CoreConnect architecture, designed for easy connection of on-chip peripheral devices. Commonly used on-chip peripheral devices come with the Xilinx Embedded Design Kit (EDK) and can be used directly or can be custom built using a special wizard without the need to have a detailed understanding of the bus protocols.

3.2.2 Local Memory Bus (LMB) interface

The Local Memory Bus (LMB) interface is an asynchronous bus used primarily to access on-chip block RAM. It uses a minimum number of control signals and a simple protocol to make ensure that local block RAM are accessed in a single clock cycle.

3.2.3 Debug Interface

The debug interface on MicroBlaze is designed to work with the Xilinx Microprocessor Debug Module (MDM) IP core. The MDM enables JTAG based debugging of one or more MicroBlaze processors using the Xilinx Microprocessor Debugger (XMD) environment. This environment is used also to download new software programs without the need to reconfigure the FPGA.

3.2.4 Fast Simplex Link (FSL) Interface

The MicroBlaze soft processor can be configured with up to eight Fast Simplex Links (FSL) interfaces, each consisting of one input and one output port. The FSL channels are dedicated uni-directional point-to-point FIFO-based data streaming interfaces, ideal for fast communication between any two design elements. In our system, FSL were used as means of communications between the two processors.

3.3 µC/OS-II Real Time Operating System (RTOS)

µC/OS-II is a highly portable, scalable, preemptive RTOS that was designed specifically for embedded system applications and ported for the Xilinx MicroBlaze soft-core processor. The following sections describe some of the main features that make this RTOS a perfect match for our application.

3.3.1 Scalable

µC/OS-II was designed to be scaled based on the services required in the application. In the MicroBlaze processor, µC/OS-II can be scaled to fit into FPGA block RAMs, as shown in Table 6 [13].

	Code (ROM)	Data (RAM)
Minimum Size	5.5 Kb	1.25 Kb (excluding task stacks)
Maximum Size	24 Kb	9 Kb (excluding task stacks)

Table 6. Memory Foot-print for μC/OS-II on the MicroBlaze Processor

3.3.2 Preemptive

μC/OS-II is fully preemptive real-time kernel. This means that μC/OS-II always attempt to run the highest-priority task that is ready to run, while suspending the current task.

3.3.3 Multitasking

μC/OS-II can manage up to 235 tasks. Each task has a unique priority assigned to it, which means that μC/OS-II does not facilitate round-robin scheduling. There are thus 256 priority levels.

3.3.4 Deterministic

Execution time and services is deterministic. This means that we can always know how much time μC/OS-II will take to execute a function or a service.

3.3.5 Statistics Task

μC/OS-II contains a task that provides run-time statistics. This task executes every second and computes the percentage of CPU (MicroBlaze processor in our case) usage. In other words, it will report how much of the time the CPU time is used by the application.

3.4 The Xilinx Multimedia Development board

The Virtex-II Multimedia Development Board [23] is designed to be used as a compact platform for developing multimedia applications, with video input and output interfaces, with serial and Ethernet interfaces. It uses a Virtex-II XC2V2000-FF896 as the user applications FPGA. The board can be used as a platform for general application that includes the Xilinx MicroBlaze 32-bit soft processor. It has several push buttons, DIP switches and two Light Emitting Diodes (LEDs) for user interaction with the system. Figure 17 [23]shows the Xilinx Multimedia Development Board components.

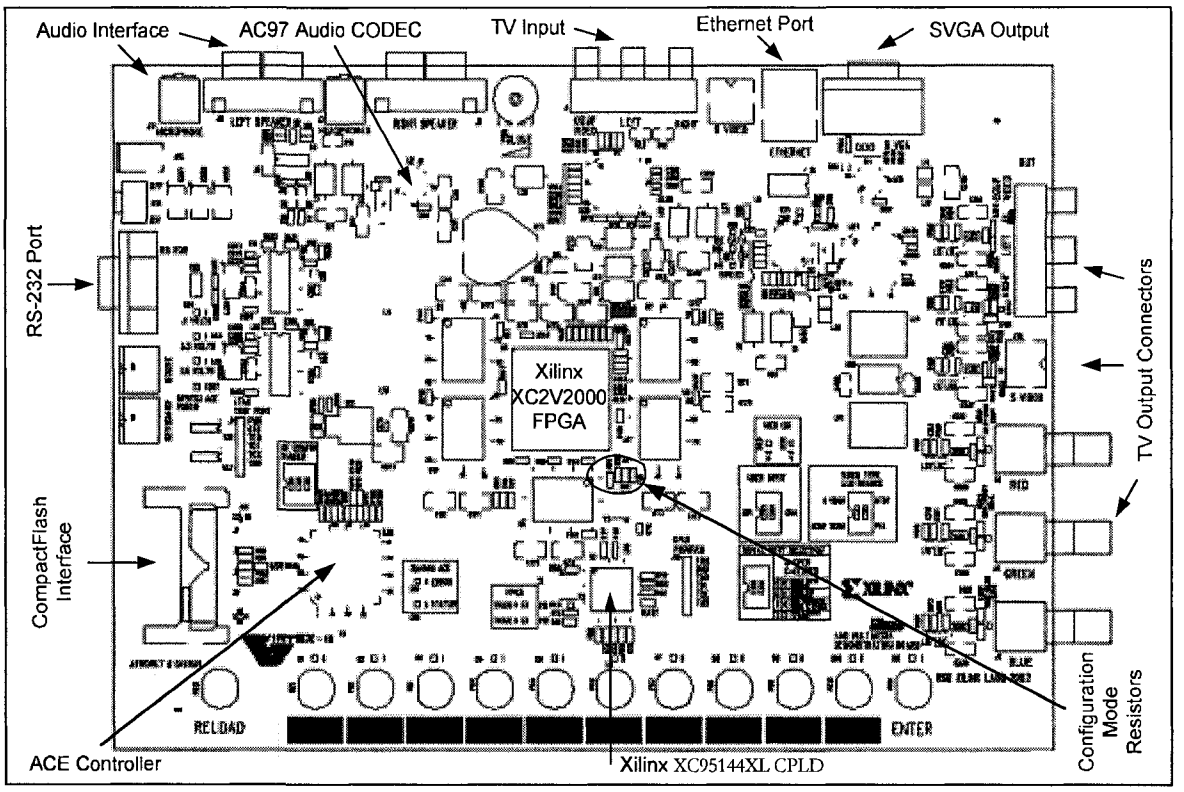


Figure 17. Xilinx Multimedia Development Board

Figure 18 shows a block diagram for the major blocks on the multimedia board. The following sections will describe each of the main board functions that were used in our application.

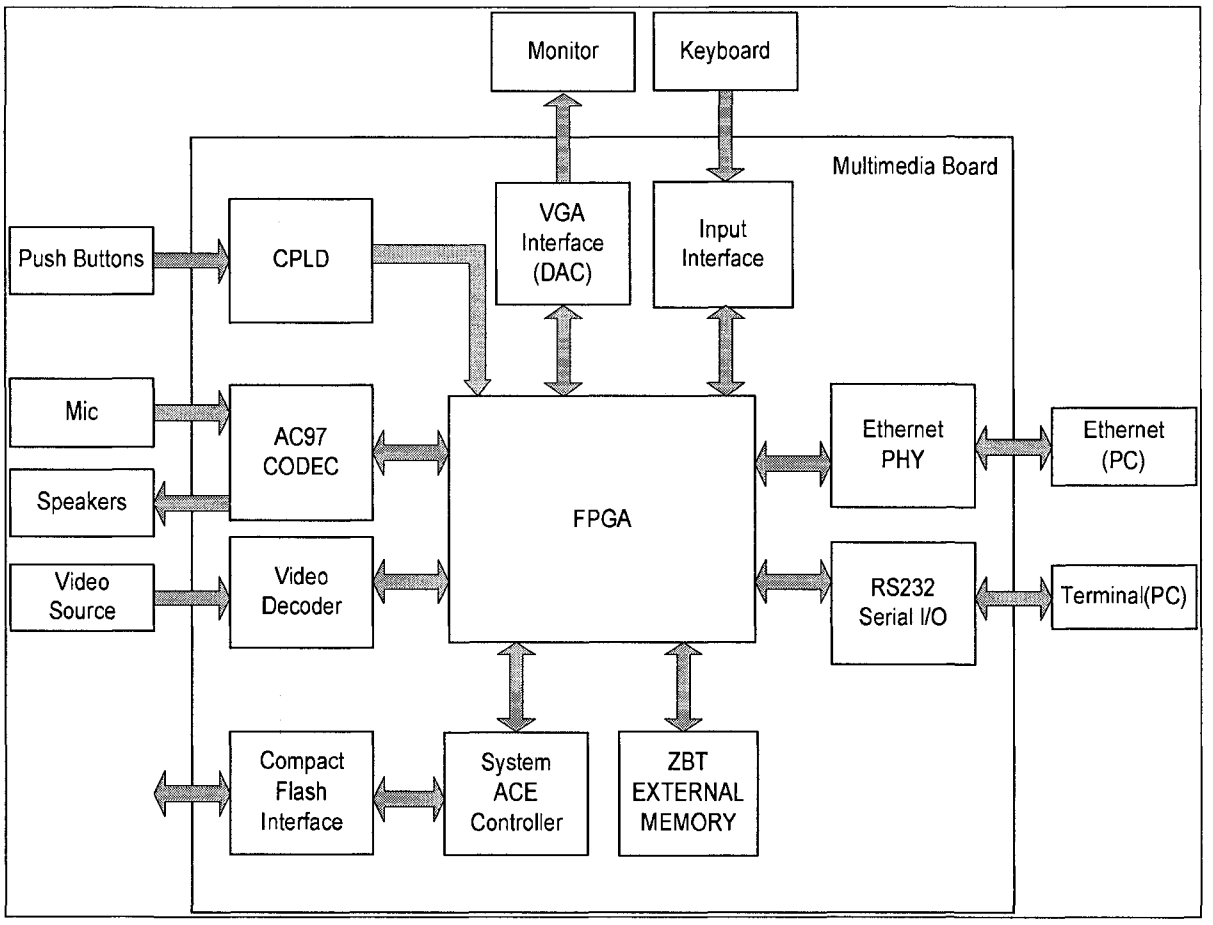


Figure 18. Major Blocks of the Xilinx Multimedia Board

3.4.1 Multimedia Board Virtex-II FPGA Configuration

Virtex-II Configuration is carried out using a subset of the device pins, some of which are dedicated, while others can be used as general-purpose input and output after the configuration is complete [18]. Several configuration methods are available and can be selected via the dedicated mode pins (M2, M1 and M0). Several configuration modes are supported on Xilinx Virtex-II FPGAs as shown in Table 7.

Configuration Mode	M2	M1	M0	CCLK Direction	Data Width	Daisy Chain
Master Serial	0	0	0	Out	1	Yes
Slave Serial	1	1	1	In	1	Yes
Master SelectMAP	0	1	1	Out	8	No
Slave SelectMAP	1	1	0	In	8	No
Boundary Scan (JTAG)	1	0	1	N/A	1	No

Table 7.Virtex-II Configuration modes and pins settings

Programming is carried out either using Flash in-system programmable configuration PROMS (with JTAG interface) or externally using special dedicated cables that connects the host pc to the JTAG connection of targeted system such as the Parallel Cable IV and Platform USB cables. As noted in Table 7, some configuration methods allows for daisy chaining of more than one FPGA when configuring.

3.4.2 System ACE

System Advanced Configuration Environment (System ACE) is a configuration mode that is supported by Xilinx and is used on the multimedia board. The System ACE solution is a chipset, consisting of a controller device (ACE controller), and a CompactFlash storage device.

Figure 19 shows the ACE controller block diagram. The ACE controller provides an intelligent interface between the JTAG port of the FPGA and three configuration sources: CompactFlash, Microprocessor (MPU) and Test JTAG [33].

The ACE Controller converts the configuration data stored on the Compact Flash into IEEE1149.1 Boundary Scan (JTAG) serial data. On the Multimedia board, the ACE Controller allows the user to select from one of eight possible configuration loads (bitstreams) each time the board is powered up or the reload push button is pressed.

The FPGA can be configured directly with one of the download cables, by connecting it to the test JTAG port header.

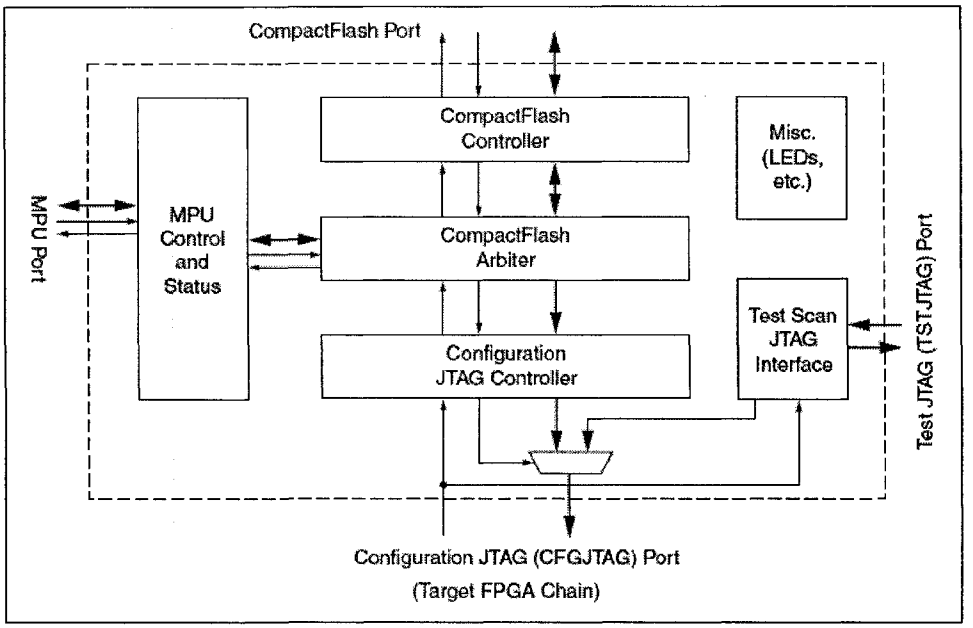


Figure 19 ACE Controller Block Diagram

The MPU interface is composed of a set of registers that provide a means for communicating with the CompactFlash control logic, configuration control logic, and other resources in the ACE controller. It can also be used to control the configuration flow: start configuration, determine the source of configuration, control the bitstream version, etc.

In the multimedia board, the MPU interface is connected directly to the Virtex-II FPGA. The OPB System ACE interface controller (OPB_SYSACE) core is an OPB interface core that connects this interface to the OPB bus, and thus allows the MicroBlaze microprocessor, on the Virtex-II FPGA, to access the ACE controller internal registers and its different ports.

3.4.3 Multimedia Board FPGA Configuration Mode Pins

The FPGA configuration Mode Pins, on the Multimedia Board, were permanently set to Boundary Scan/JTAG Mode (M2:M0 = 101), as Boundary Scan (JTAG) is the method used for configuring the on board Virtex-II FPGA. However, the Virtex-II FPGA internal ICAP port is disabled when the configuration pins are set to this mode, and enabled in all other configuration

modes [28]. Also, setting the configuration modes to anything other than Boundary Scan still allows configuring the FPGA using the JTAG port, as this method overrides other means of configuration. For this reason, a minor modification was done on the board, as shown in Figure 20, to change the configuration mode settings to anything other than M2:M0=101, and thus enabling the ICAP port while maintaining the ability to program the Virtex-II FPGA using the JTAG port.

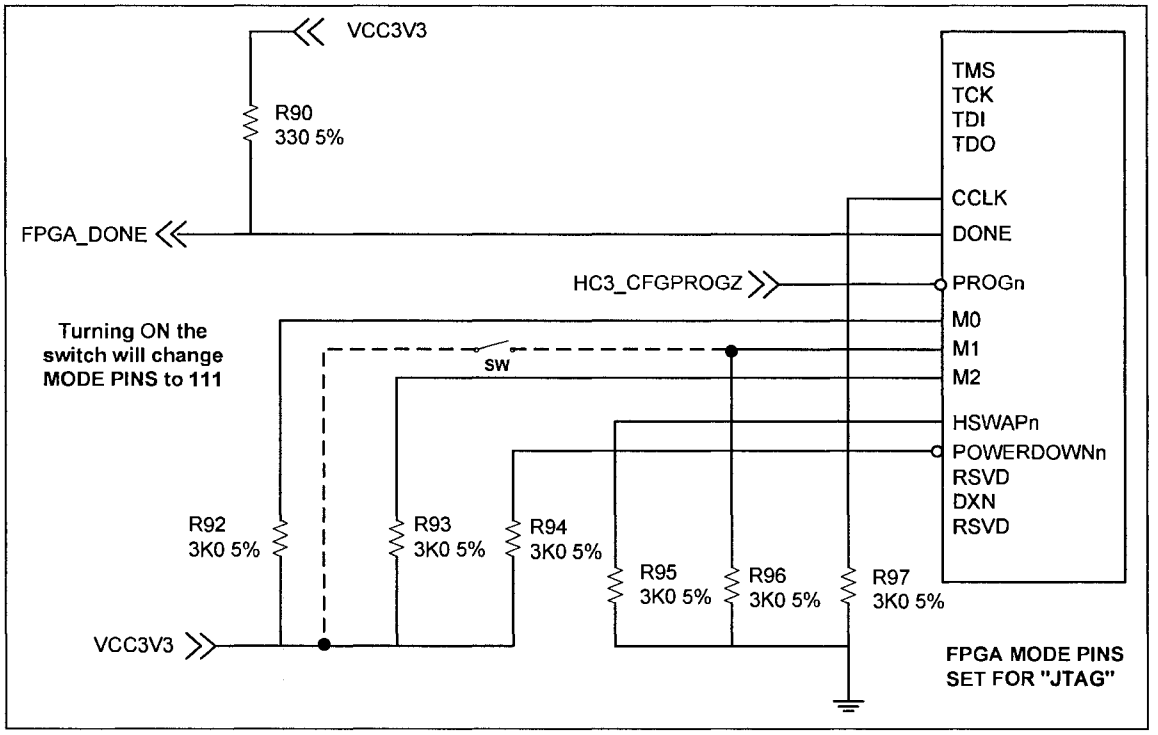


Figure 20 Multimedia Virtex-II Configuration Mode.

3.4.4 Clock Generation

The internal operation of the FPGA is based on clocks derived from the 27 MHz system clock provided by a CPLD on board (XC95144XL). This CPLD will also generate the proper reset signals for the system to initialize properly.

3.4.5 Ethernet Port

An onboard network connection supporting 10/100 Ethernet is provided on board. The physical interface is created using a LevelOne LXT972 3.3 Volt PHY. The LXT972 is an IEEE-compliant Fast Ethernet transceiver that directly supports both 100BASE-TX and 10Base-T applications.

3.4.6 RS-232 Port

This serial port is directly connected to the FPGA and it is used to print the MicroBlaze processor messages on screen.

Chapter 4: ERACE Self Reconfigurable System-on-Chip

This chapter details the hardware setup and steps required to build a self reconfigurable system-on-chip. It also explains the setup and configuration for the used embedded system.

4.1 Introduction

Partial reconfiguration implies that the active array maybe partially reconfigured, while ensuring that the correct operation of the active circuits is not changed. Self reconfiguration extends this concept by assuming that specific logic circuits on the FPGA are used to control the reconfiguration of other parts of the FPGA.[31],

Partial reconfiguration can be achieved by applying some of the methods explained in chapter 3. In those methods, special guidance (modular flow) has to be followed at compile time to be able to partially reconfigure the system at run time. With self reconfiguration, this can be done, on the Virtex-II FPGAs, at run time, using the internal configuration access port (ICAP). The self reconfigurable platform, defined in this chapter, combines both the modular-flow based partial reconfiguration, and self-reconfiguration. The self reconfiguration is controlled by a dedicated reconfiguration processor.

With the modular flow, a predefined area is allocated for the active part. This is the area where the circuit under test (CUT) will be synthesized, placed-and-routed and then configured. A new CUT can be applied by reconfiguring the active module, while keeping the embedded processor(s) logic area intact, thus separating development work for both. New CUTs can be reconfigured, at run-time, externally by using Xilinx configuration tools, or locally by the reconfiguring processor using the ICAP interface. Since the area allocated for the active module (CUT) occupies a fraction of the overall FPGA, its reconfiguration time is considerably less than reconfiguring the whole FPGA.

A multi-processor system implementation was developed where one processor (Processor R) is responsible for the reconfiguration tasks, and the other one (Processor A) is for the applications. The multi-processor system is generated and implemented using the Xilinx EDK tools, while the modular flow is part of the Xilinx ISE tools set. The following paragraphs explain the steps necessary to combine both flows, and to implement the self reconfigurable platform.

4.2 ERACE'S High Level System Architecture

The presented system is built for generic reconfigurable applications. It consists of two high-level components, namely the application flow (AF) and the reconfiguration flow (RF) as presented in Figure 21. The application flow is mostly executed as a software program, with some of its parts being executed in hardware blocks (HB) that are dynamically inserted and removed under the reconfigurable flow to form the reconfigurable processing unit (RPU)[8]. The HW and SW sequences of the application flow are extracted from the source code by just-in-time (JIT) compiler which generates both the AE and the RF and implements a mechanism that ensures synchronization between the two. This JIT compiler is discussed in detail in [9].

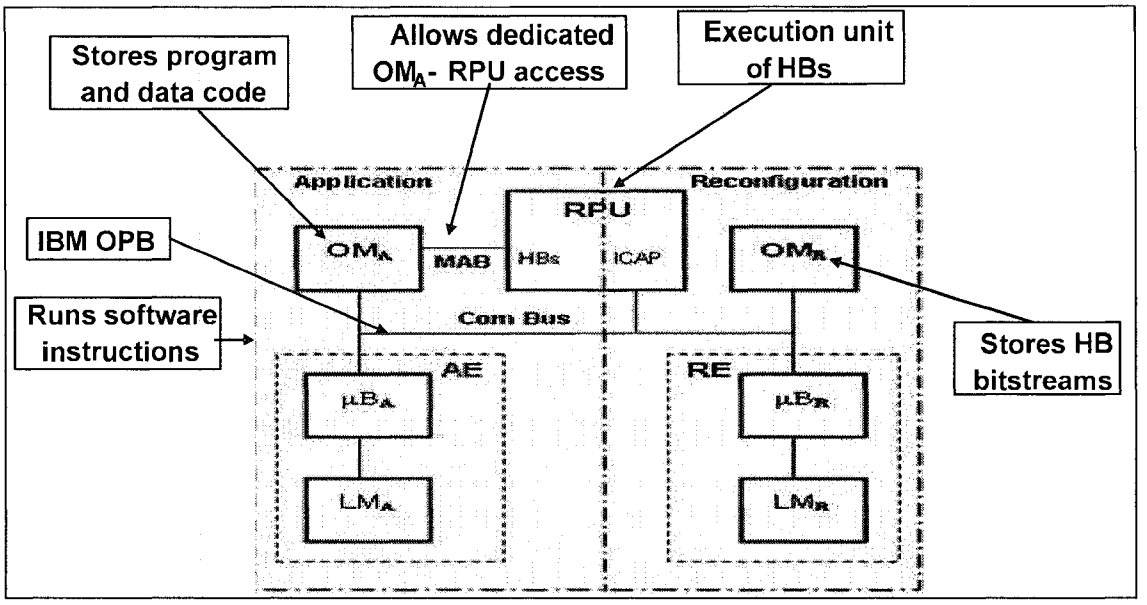


Figure 21 High Level System Architecture

The system architecture includes the following components:

AE (μB_A + LM_A): The Application Element is a soft core microprocessor (MicroBlaze μB_A) with the Local Memory (LM_A). This is the execution unit for the main system applications.

RE (μB_R + LM_R): The Reconfiguration Element consists of a second soft core processor (μB_R) with the Local Memory (LM_R). This element will be the execution unit for the run-time reconfiguration of the RPU.

RPU: The Reconfigurable Processor Unit is the execution unit for the tasks which are to be mapped directly in hardware. The RPU is dynamically reconfigurable with different hardware blocks.

OM: The On-chip Memory. The OM_A stores the user data and the instructions for the application flow, whereas the OM_R stores the bitstreams for the different tasks to be carried out on the RPU as well as the instructions for the reconfiguration flow.

ComBus: The Communications Bus is used between the components. For our architecture, we employ the IBM On-board Peripheral Bus (OPB).

MAB: The Memory Access Bus allows dedicated access to the application data in OMA by the RPU.

The operations of the application flow (AF) and reconfiguration flow are shown in **Error! Reference source not found.**.

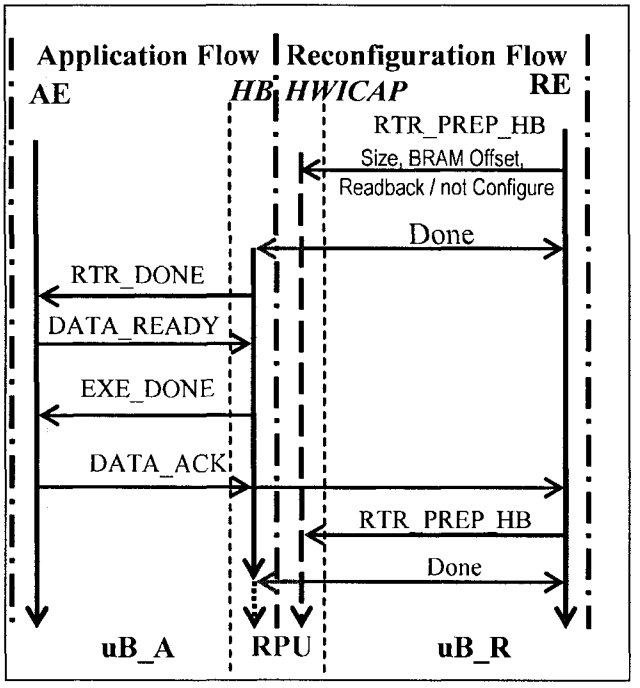


Figure 22 Application and Reconfiguration Flows of the System

The Internal Configuration Access Port (ICAP) [13] is the FPGA interface that allows for the dynamic partial reconfiguration of the RPU with the appropriate HBs under the control of the reconfiguration flow through the HWICAP controller. While the application flow (AF) runs on the AE, the reconfiguration flow (RF), which runs on the RE, sends the signal RTR PREP HB to the ICAP controller (HWICAP) to start the loading of the first HB bitstream (of length "Size", from the address "BRAM Offset" in OMR) onto the RPU. Once that HB is ready within the RPU, the HWICAP sends back a "Done" signal to the RE and sets the flag RTR DONE to make the application flow aware that it is ready for use. Once the application flow has prepared the data that the HB needs for computation, it makes the HB aware of this by asserting the DATA READY flag. The application flow continues to run as long as it does not need the results computed by the HB. The latter asserts EXE DONE when it completes its task and has prepared the results to be read by the application flow. When the application flow needs these results, it checks the EXE DONE flag, and blocks if it is not yet set. The application flow gets the results and then asserts DATA ACK to acknowledge to the HB that it has received the valid data. Once this HB is no longer needed it can be replaced with another one. Also, another HB can be loaded at a different location of RPU as soon as the current uploading process is completed [7].

The implementation of this architecture employed several elements that improved the system's speed while observing the platform's technological constraints. Figure 23 presents the detailed block diagram of the system. The implementation phase brought a number of modifications and enhancements, with the most important ones being (i) the incorporation of Micrium's μC/OS-II as the real time operating system (RTOS) for both μB_A and μB_R (the kernel and the main application code for each processor fit into the processor's local memory), (ii) the addition of OPB slave peripherals, (iii) the addition of the RPU and its associated bitstream buffer, OM_R, and, (iv) the insertion of four unidirectional FSLs forming the two intercommunication paths between the RPU and the AE and RE.

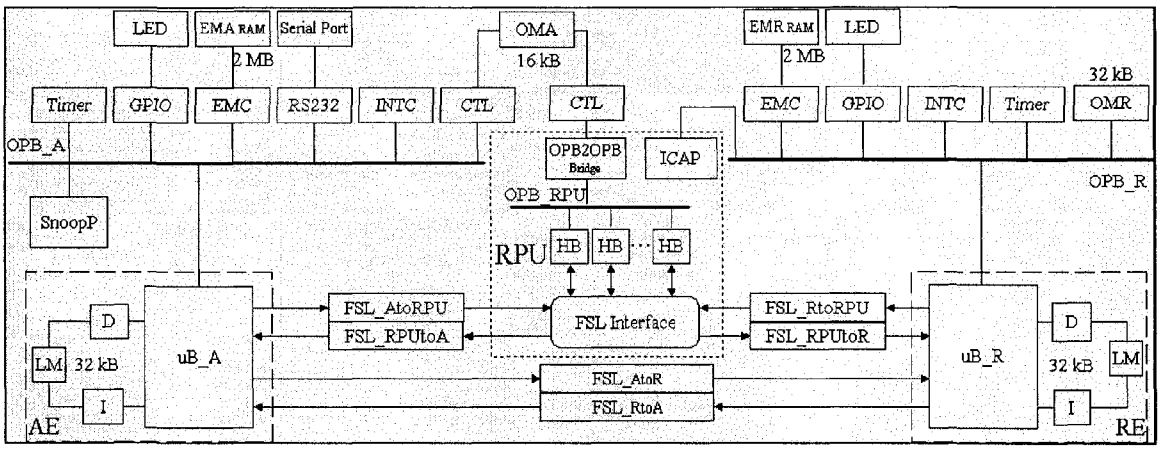


Figure 23. ERACE detailed System Architecture

4.3 FPGA Top Level Design

The top level design contains instantiations of three basic modules: static (the processor(s) module), active (the reconfigurable block which contains the CUT logic), and the bus macro(s) as shown in Figure 24. EDK flow is used to implement and build the system files for the static module, and different active modules can be generated (from RTL) using the ISE flow. The bus macros are provided as hard macros and instantiated on the top level RTL file. Blocks are assembled to generate the final netlist using the modular flow.

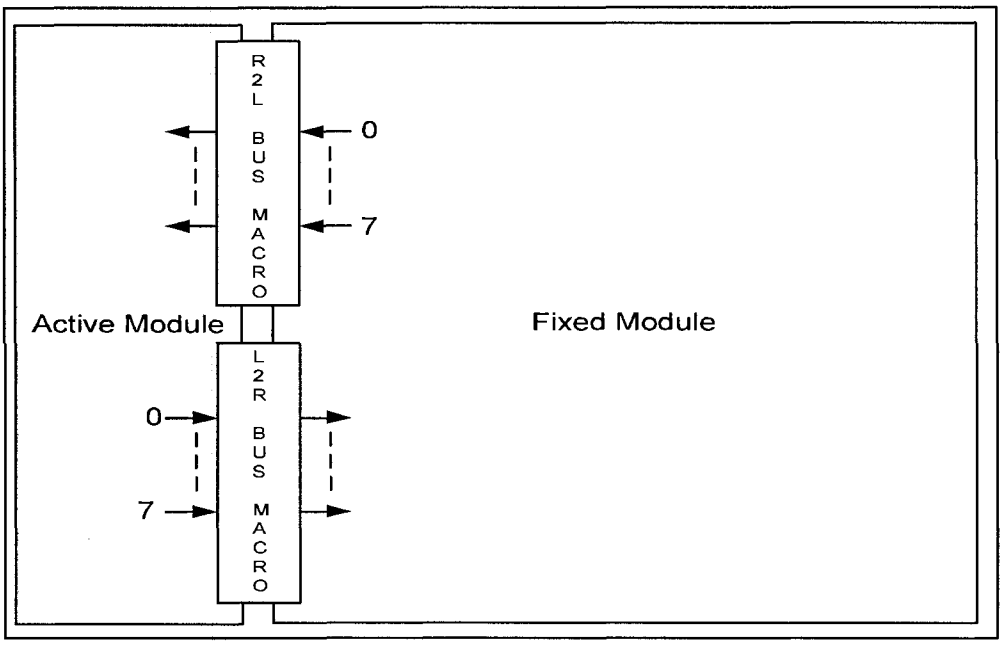


Figure 24. Top level Design Modular View

4.4 Platform Generation (EDK flow)

The Xilinx Embedded Development Kit (EDK) [32] tool was used for creating the embedded processor design subsystem. As opposed to a fully static circuit, the design hierarchy (project options menu) is set now as a sub-module rather than the top level as in the case when the EDK tool is only (normally) used. By setting this option, the EDK tool will only be used to generate the netlist, and not the bitstream file. It will also generate the following files:

/hdl/

/system_stub.vhd — ***HDL subsystem with Xilinx I/O primitives inserted (top level RTL)***

/system.vhd — ***HDL subsystem without Xilinx I/O primitives inserted.***

/implementation/

/system_stub.bmm — ***BMM file with top-level subsystem instance in path.***

/system.bmm — ***BMM file without the top-level subsystem instance in path.***

/peripherals.ngc ***XST-generated peripheral files.***

System_stub.vhd contains embedded processors instantiation (system) and will be used as our top RTL file after adding the active block and bus macros instantiations. The Block Memory Map file (system_stub.bmm) will be attached to the top level RTL file as it contains the complete hierarchical path. No user constraints file (.ucf) is needed for the EDK modular flow. This process needs to be run again if any changes were needed in any of the embedded processors system.

Note that the system_stub.vhd file, generated by the EDK tool, has the same name for the system module and its instantiation. A small script was written to change all the instantiation names of the system module to "system_i". This includes the BMM files too.

The EDK tool is used to add, debug and compile the software project to be used. An executable and linkable format (*ELF*) file is generated that is ready for running on the processor, and is used to update the final bitstream file generated by the Xilinx Integrated Software Environment (ISE) tool [18].

4.5 Modular Design Implementation (ISE flow)

This is where all the modules of the design will be defined, generated and finally assembled following the Modular Design Flow described earlier.

4.5.1 Initial Budgeting

This phase involves creating the floorplan and constraints for the overall design. It involves defining the size and location of the both modules (column boundaries and as explained earlier). At the end of this phase, created the necessary constraints file (.ucf), to be used in the modular flow. We started initial budgeting assuming the following requirements:

- No memory requirements for the active module. All memory on board is required for the two embedded processors.

- Any IO pins required for the embedded processors should reside within the active module (e.g. RS-232, clocks, resets etc).
- The ICAP_VIRTEX2 primitive is located in the lower right of the device. It should be part of the static module.
- The active block has enough area to hold the biggest CUT we are going to test. In our case CUT C432.
- The static module should be large enough to hold the two embedded systems.
- Enough bus-macros to hold all top input and output pins for CUT C432.

The Xilinx XC2V2000 FPGA has 48 CLB columns and 56 rows. A CLB column is the minimum we can assign for the active module. One CLB column has 56 CLBs, each with 4 slices, which gives us a logic space of 224 slices (448 LUTs). By comparison, the CUT C432 synthesizes to 41 slices. We found one CLB column for the active module enough for our application. We chose the left most column of the device (Figure 24). This will leave the rest of the device for the static module including the entire block RAMs on the device. Incidentally, one LED is located in the active module, while the other is on the static one. Below is the constraints used to floor plan the area of each module:

AREA_GROUP "AG_system" MODE=RECONFIG;

AREA_GROUP "AG_hw_block" MODE=RECONFIG;

INST "system_i" AREA_GROUP = "AG_system";

AREA_GROUP "AG_system" RANGE = SLICE_X2Y111:SLICE_X95Y0;

AREA_GROUP "AG_system" RANGE = TBUF_X2Y111:TBUF_X84Y0;

AREA_GROUP "AG_system" RANGE = RAMB16_X0Y13:RAMB16_X3Y0;

AREA_GROUP "AG_system" RANGE = MULT18X18_X0Y13:MULT18X18_X3Y0;

INST "dynamic" AREA_GROUP = "AG_hw_block";

AREA_GROUP "AG_hw_block" RANGE = SLICE_X0Y111:SLICE_X1Y0;

AREA_GROUP "AG_hw_block" RANGE = TBUF_X0Y111:TBUF_X0Y0 ;

CUT C432 has 36 inputs and 7 output pins. Each bus macro can hold 8-bits of communications, so we instantiated 5 *right_to_left* and 2 of *left_to_right* bus macros:

INST "busStatToDyn" LOC = "SLICE_X0Y52";

INST "busDynToStat" LOC = "SLICE_X0Y50";

INST "busStatToDyn2" LOC = "SLICE_X0Y48";

INST "busDynToStat2" LOC = "SLICE_X0Y46";

INST "busStatToDyn3" LOC = "SLICE_X0Y54";

INST "busStatToDyn4" LOC = "SLICE_X0Y58";

INST "busStatToDyn5" LOC = "SLICE_X0Y60";

More constraints were added to define IO pins and the global clock period of the design. The following constraints were added to recover from a crash during the final assembly phase:

INST " system_i/microblaze_a/microblaze_a/Data_Flow_I/PC_Module_I/PC_Bit_I*"

USE_RLOC=FALSE;

INST " system_i/microblaze_r/microblaze_r/Data_Flow_I/PC_Module_I/PC_Bit_I*"

USE_RLOC=FALSE;

4.5.2 Active Module

The active module should be implemented with the same constraints file. One or more active modules can be implemented, each with its own subdirectory. The top RTL of each active module should be like this:

entity hw_block is

```
port( InBus : in std_logic_vector(0 to 39);
      Led1_out: out std_logic;
      OutBus : out std_logic_vector(0 to 6);
      IDBus : out std_logic_vector(0 to 8)
    );
end hw_block;
```

The InBus is connected to the *right_to_left* bus macros, while the OutBus and the IDBus are connected to the *left_to_right* bus macros.

4.5.3 Final Assembly

The final assembly phase is the process of combining the embedded processor static module and the active module back into a complete FPGA design. The placement and routing achieved during the active implementation phase of the two modules will be preserved, thereby, maintaining the performance of each module.

Figure 25 shows the final layout for the assembled design as captured by the FPGA editor design tool.

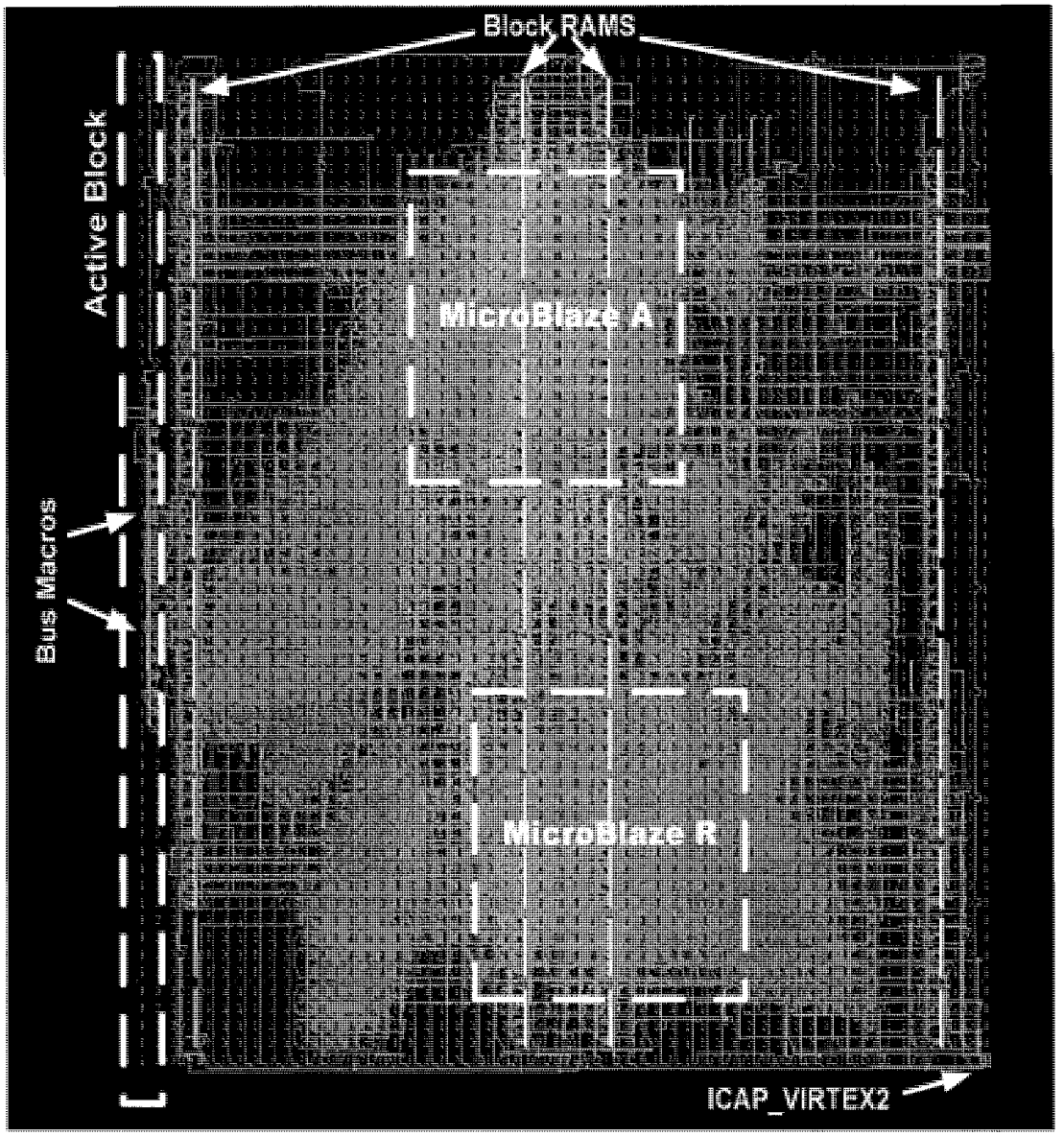


Figure 25. Final design layout

At the very minimum, at least one final Assembly phase must be run. The Partial Reconfiguration flow requires that the initial bitstream loaded into the FPGA device be a complete design. This is required so that all global, non-reconfigurable logic is placed and locked down, and the only reconfigurable portions of the design will change during Partial Reconfiguration.

4.5.4 Creating & downloading Module-Based Partial Reconfiguration Bitstreams

Initial configuration of the FPGA requires the generation of a bitstream for the complete and assembled design. This can be done using:

bitgen -m -w system_stub_routed.ncd system_stub_routed.bit

Multiple bitstreams can be generated for every partially reconfigurable module (active). This allows us to create a new bitstream for different CUT's independently using:

bitgen -w -m -g ActiveReconfig:Yes -d system_stub1.ncd system_stub_partial.bit

The new generated bitstream is only a subset of the full bitstream and this is reflected in the size. The size will depend on the area of the active module. Again, when downloading any new active module bitstream, the full bitstream must be already being programmed into the device.

The downloading of the new bitstream(s) can be done externally, through either the download cable or the CompactFlash, or internally through processor R using the ICAP interface (HWICAP core).

4.6 Embedded System Configuration

The embedded system used in this platform is based on the previous work done for the ERASE project. Some customization was required to reflect the size of the FPGA part used on multimedia board, and some of the available interfaces. Figure 26 shows the full system block diagram.

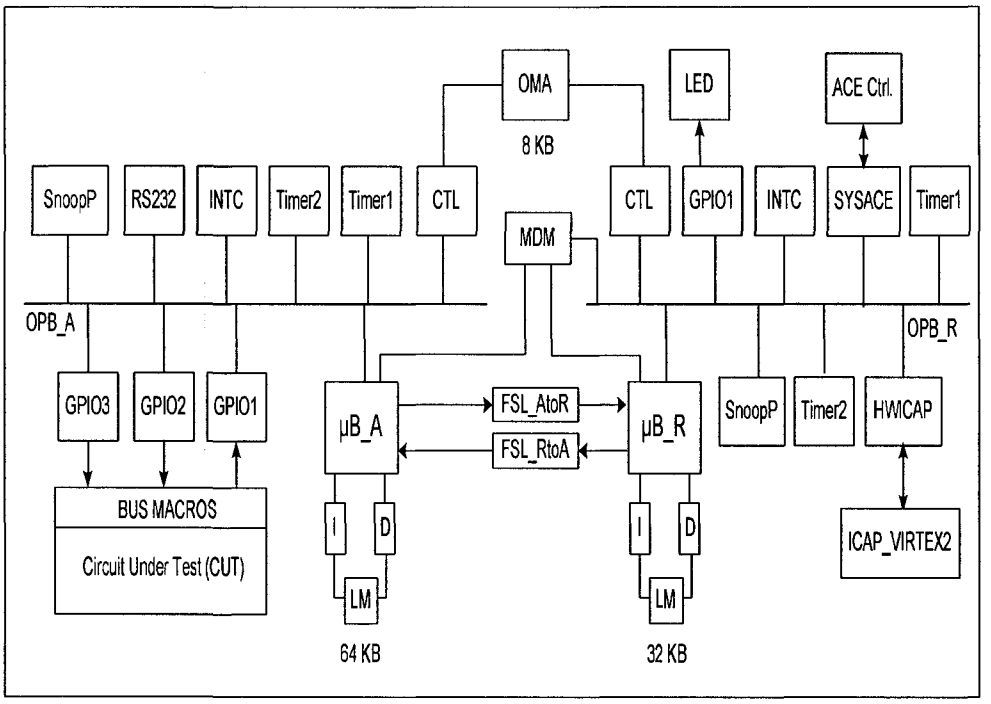


Figure 26. Embedded System Architecture

All the reconfiguration tasks were done specifically by MicroBlaze R tasks through the HWICAP, while MicroBlaze A runs all the necessary CUT analysis and tests. Communication between the two MicroBlaze processors is done through two FSLs.

4.6.1 MicroBlaze processor

Two MicroBlaze processors are instantiated. MicroBlaze R handles all reconfiguration tasks through the HWICAP, while MicroBlaze A runs all the necessary CUT analysis and tests. Both MicroBlaze processors were customized and set for µC/OS-II from the EDK tool set.

4.6.2 Peripherals

The following peripherals were instantiated with each MicroBlaze and as indicated. All the peripherals were provided by Xilinx as part of the EDK tool set except the SnoopP [19]:

- LMB BRAM Interface Controller (A & R): Data Local Memory Bus Controller.
- LMB BRAM Interface Controller (A & R): Instruction Local Memory Bus Controller.
- Block RAM (A & R): Handle the Block RAM.
- OPB UART Lite (A only): Allows communicate through the serial port.
- OPB GPIO (A & R): Four of the General Purpose I/O cores are instantiated. One for MicroBlaze R (output: connected to board led), and 3 for MicroBlaze A (2 Outputs, 1 input all connected to bus macros).
- OPB Timer 1 (A & R): This timer is needed for generating the necessary clock tick required by μC/OS-II.
- OPB Timer 2 (A & R): This timer is used for the μC/OS-II statistics task.
- OPB Interrupt Controller (A & R): Handles all interrupts (timer1, timer2 and RS-232).
- DCM module (A & R): Digital Clock Manager for the main system clock on the FPGA.
- Microprocessor Debug Module (A & R): This module enables JTAG based software debugging and downloads. It is connected internally to the two MicroBlaze's debug ports; however, it is instantiated on the MicroBlaze R OPB bus to allow for JTAG based UART through this bus.
- OPB HWICAP (R only): Allows MicroBlaze R to Read/Write the configuration memory through the ICAP interface.
- Fast Simplex Link 1: This FSL is instantiated for MicroBlaze A to MicroBlaze R communications.
- Fast Simplex Link 2: This FSL is instantiated for MicroBlaze R to MicroBlaze A communications.
- OPB BRAM OMA (A and R): This is a shared block memory between the two MicroBlaze processors. It is used to store shared data between the two MicroBlaze processors.
- SnoopP (A & R): SnoopP is a custom IP module that was developed as part of a PhD degree at University of Toronto [19]. It is used to accurately measure SW processes execution time.

- OPB_SYSACE (R only): Allow MicroBlaze R to control configuration loads saved on the CompactFlash card.
- MDM (Microprocessor Debug Module: A&R): The MDM module is connected to MicroBlaze R & A, through JTAG, to allow for software debugging which includes S/W & H/W break points, external processor control and read and write to memory.

4.6.3 Memory

The XC2V2000 device contains a total of 56 18K bit RAM blocks (organized as 1 kilo-word of 16+2 bits) distributed over four memory columns. This will give a total block RAM memory of 1008K bits (56 kwords). 64K bytes (32 kwords, i.e., 32 RAM Blocks) are allocated for MicroBlaze A, and 32K bytes (16 RAM Blocks) for MicroBlaze R. Since the HWICAP core requires one Block RAM, this leaves 8K bytes for the shared memory (OMA). In total we are using 53 Block RAMs out of the 56 ones available on the device. Table 8 and Table 9 list the memory map for MicroBlaze A and R respectively.

Module	Base Address	High Address	Size	Min Size
dlmb_ctrl_A	0x00000000	0x0000FFFF	64 KB	0x800
ilmb_ctrl_A	0x00000000	0x0000FFFF	64 KB	0x800
RS232	0x00030100	0x000301FF	256	0x100
opb_gio_A	0x00030200	0x000303FF	512	0x200
opb_timer_R	0x00030400	0x000304FF	256	0x100
snoopy_R	0x00030600	0x000306FF	256	0x100
opb_gio2_A	0x00030800	0x000309FF	512	0x200
opb_gio3_A	0x00030A00	0x00030BFF	512	0x200
opb_int_A	0x00030F00	0x00030F3F	64	0x20
opb_bram_cntlr_OMA	0x00050000	0x00051FFF	8 KB	0x800

Table 8. Memory Map for MicroBlaze Processor A

Module	Base Address	High Address	Size	Min Size
dlmb_ctrl_R	0x00000000	0x00007fff	32 KB	0x800
ilmb_ctrl_R	0x00000000	0x00007fff	32 KB	0x800
opb_gio_R	0x00030200	0x000303FF	512	0x200
opb_timer_R	0x00030400	0x000304FF	256	0x100
snoopy_R	0x00030600	0x000306FF	256	0x100
opb_mdm	0x00030700	0x000307FF	256	0x100
opb_intc_R	0x00030F00	0x00030F3F	64	0x20
opb_bram_cntlr_OMA	0x00050000	0x00051FFF	8 KB	0x800
opb_hwicap_R	0x00060000	0x00061FFF	8 KB	0x800

Table 9. Memory Map for MicroBlaze Processor R

Chapter 5: Self-reconfigurable B.I.S.T. Application

5.1 Introduction

Built-in self-test (BIST), in its generic definition, is a set of structured-test techniques for combinational and sequential logic, memories, multipliers, and other embedded logic blocks. In each case the principle is to generate test vectors, apply them to the circuit under test (CUT) or device under test (DUT), and then check the response. The aim is to make sure that the circuit is functioning and that we are able to catch fault when it happens. Catching such faults requires test vectors that are able to catch all kinds of faults when it is happening.

This chapter presents the steps required to validate digital circuit testing on a benchmark circuit to validate and quantify the run-time reconfiguration computing paradigm.

5.2 Structural Testing: A Sample Target Application

One target application for our reconfigurable ERACE architecture is the field of structural testing to develop BIST algorithms. Structural testing, as opposed to functional testing, is based on ensuring that the circuit under test is free from physical faults (most likely developed during manufacturing). The following sections give a brief introduction to structural testing, and some of the techniques used in testing BIST algorithms.

5.2.1 Built-In-Self-Test

BIST is a set of structured-test techniques for combinational and sequential logic, memories, multipliers and other embedded logic blocks. In each case, the principle, as shown in Figure 27, is to generate test vectors, apply them to the circuit under test, and then check the response. The test vector generation and checking are built on the same integrated circuit as the circuit under test. The provision for on-chip testing reduces reliance on external test equipment and thus reduces cost.

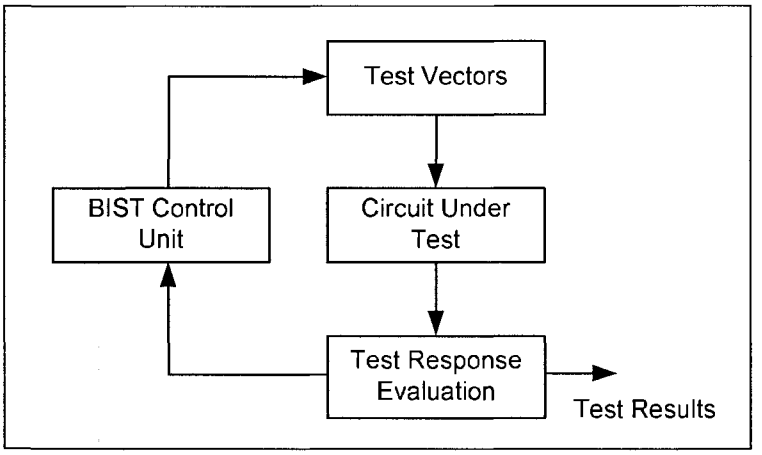


Figure 27 BIST Principle

5.2.2 Fault Modeling

The principle of fault modeling is to reduce the number of effects to be tested by considering how defects are represented. For example, a physical defect can be represented as a logical fault. This fault maybe static (shorts, breaks); dynamic (components out of specification, timing failure) or intermittent (environmental factors) [48].

5.2.2.1 Single-stuck Fault Model

Static-faults are usually modeled by the stuck fault model. Many physical defects can be modeled as a circuit node being either stuck at 1, or stuck at 0. Other fault models include stuck open and stuck short faults.

The single-stuck fault model (SSFM) assumes that a fault directly affects only one node and that the node is stuck at either 0 or 1. These assumptions make test pattern generation easier, but the validity of the model is questionable. Multiple faults do occur and multiples faults can theoretically mask each other. On the other hand, the model appears to be valid most of the time, hence, almost all test pattern generation relies on this model.

5.2.3 *Fault-oriented Test Pattern Generation*

The task of test pattern generation is that of determining a set of inputs to indicate unambiguously whether the internal node is faulty. For an n-input combinational circuit, the number of possible input combinations is 2^n. Thus, we could apply all 2^n inputs (exhaustive functional test), but in general it is better to find the minimum necessary number of input patterns. In order to generate a minimum number of test patterns, a fault-oriented test generation strategy is adopted, as follows [48]:

Prepare a fault list (e.g all nodes stuck-at-0 or stuck-at-1)

Repeat:

- Write a test;

- Check fault cover (one test might cover more than one fault)

- Delete covered faults from list

Until fault cover target is reached

This is also called deterministic testing, and although this method promises the detection of all detectable faults, the duration of the search process and the length of the resulting test pattern maybe excessive.

In a completely different approach a fixed number of test patterns are generated without feedback from the fault model. The sequence of test patterns is called *pseudo-random*, because it has some important properties of a random sequence, while being totally predictable and repeatable. The coverage of the test sequence is checked by fault simulation (e.g. the fault list described above).

In short, test pattern generation maybe random or optimized. Any one test may cover more than one fault, with often faults are indistinguishable.

5.2.4 Fault Simulation

Fault simulation consists basically of simulating a circuit in the presence of faults. Comparing the fault simulation results with those of the fault-free simulation of the same circuit simulated with the same applied test, we can determine the faults detected by that test.

5.2.5 Test Pattern Generator Circuits

The methods for test pattern generation are highly correlated with the types of test pattern discussed above. In the following some common approaches are discussed:

1. ROM (stored pattern test): Since no restrictions on the sequence of patterns apply, this method can provide excellent fault coverage and is used in combination with the conventional deterministic algorithm. However, sequence length is directly proportional to ROM size, which is not so much a problem for external testers, but usually results in a prohibitive area overhead for built-in test.

2. Processor (test pattern calculation): If the CUT includes a processor and memory, a test program can be employed to calculate an appropriate sequence of test patterns.

3. Counter (exhaustive test): This simple way of test pattern generation is used for exhaustive and pseudo-exhaustive testing.

4. Pseudo-random generator (random test): These methods of recursive generation of test patterns provide reasonable coverage with low hardware-overhead. For this reason random testing is most often found in built-in test designs. Hardware circuits suitable as random generators are register chains like linear feedback shift registers (LFSRs, cellular automata or built-in logic block observers (BILBOs). An LFSR modified such that it cycles through all possible states can be used as cheaper replacement of a counter for exhaustive testing, if the sequence of pattern is irrelevant (combinatorial testing). As mentioned above, pseudo-random pattern generators may be designed

under the constraint that their output sequence includes a set of deterministically generated test vectors.

5.2.6 Software Realization

The software realization of the fault model, where the CUT and set of aliasing-free compressors[7] are simulated in software in order to extract all the faults of that particular CUT. An aliasing-free compressor is one that does not compromise the CUT's fault-detecting capabilities. Thus, all faults that can be detected at the output of the CUT can also be detected at the outputs of the CUT and the compressor combined. The compressors are used to reduce the number of system outputs, and thus allow for a smaller storage area. This strategy is a common method used to prototype fault models, without paying too much attention to real-time interactions or results.

5.2.7 Sequential Compile Time Reconfiguration (CTR) Realization

In this strategy, the CUT and compressors are synthesized and mapped to the target device, with fault-injection multiplexers (FIMs) (described later in this section) built into the CUT.

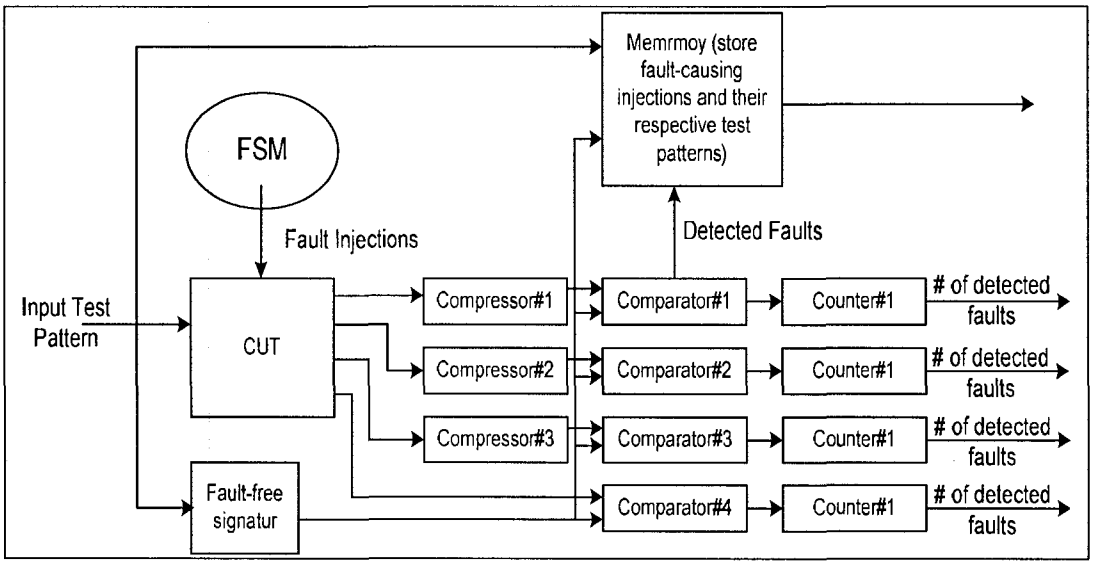


Figure 28 Basic Test Strategy Architecture

This method allows us to sequentially perform test verification on the circuit by applying the test patterns and recording the circuit's behaviour for each applied test pattern and injected fault. This strategy is equivalent to the other ones used to test circuits after the synthesis and mapping steps.

5.2.7.1 Fault-Injection Multiplexers

The hardware fault injection technique is imperative in order to iteratively inject faults to every mutually-exclusive wire, and to test both the stuck-at-0 and stuck-at-1 faults. As we can see from Figure 29, every mutually exclusive wire now has a FIM introduced within it, which allows us to either run the wire as is (00/11), or inject stuck-at-0 (10) or stuck-at-1 faults (01).

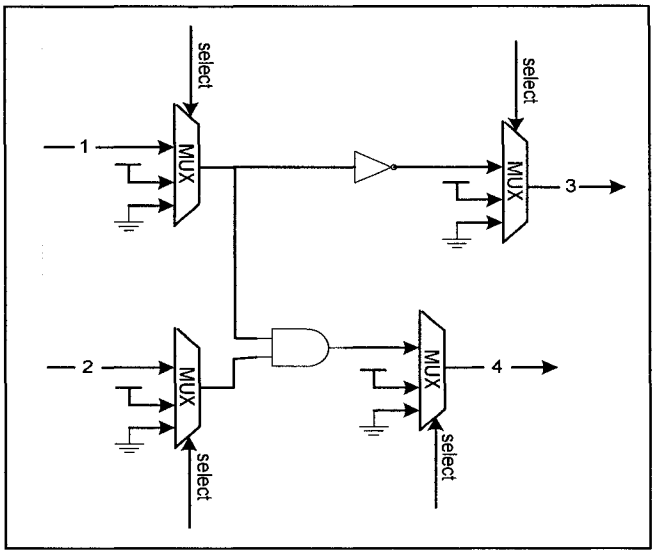


Figure 29. Fault-Injection Multiplexers Scheme for an AND gate and an inverter

5.2.8 Partially-Parallel CTR Realization

This strategy is a partially-Parallel CTR realization of the fault-model, where n * m * 2 CUTs are synthesized and mapped to the target device, with one particular fault-injection applied to each CUT. Hence, the test patterns are applied to all the CUTs simultaneously, providing us with the circuit's behaviour for each applied test pattern only. Note that in a fully-parallel strategy, n * m * l * 2 CUTs are synthesized and mapped, where, n is the number of wires in the CUT; m is the number of test patterns used; l is the number of available test patterns; and 2 is the number of stuck-at faults to inject.

The partially-parallel strategy is similar to the previous one; however, there is no need for the fault injection multiplexers as faults are already synthesized in the circuit.

If implementing the fully-parallel strategy, only one clock cycle is required to generate the circuit's behaviours for all applications of test patterns and injected faults. However, the resources that this method would need exceed any available ones, and thus this method was not adopted.

5.2.9 Sequential Run-Time Reconfiguration Realization

In this strategy, the circuit under test is synthesized and mapped to the device without fault insertion multiplexers, and without synthesized particular fault-injections. Instead, we utilize run-time reconfiguration techniques to insert stuck-at faults, and record the circuit behaviour accordingly. This is the strategy being used in this thesis.

5.3 Benchmark Circuits

Benchmark circuits have been used for years to allow objective comparison of methods and tools. In the domain of testing integrated circuits, the ISCAS'85 and ISCAS'89 are probably the best-known and most-used sets of benchmarks. The ISCAS'85 benchmarks were first presented by Brglez and Fujiwara [29]. They consist of 10 combinational digital circuits in gate level netlist representation. As the full gate-level implementation of these benchmark circuits is provided, they have been used for a wide variety of research problems, including automatic test pattern generation, fault simulation, and test data compression. The homogenous character of the combined ISCAS'85 and ISCAS'89 benchmarks makes them very suited for comparing methods or tools for a list of circuits [30]. In this thesis we used ISCAS'85 as our main source of the benchmark circuits.

5.4 Compile Time Fault Injection

The definition used here for compile time fault injection is to have several versions of the CUT for each fault to be injected. Each version of the circuit is synthesized and placed and routed for that particular fault. Several downloading of the CUT has to be done during the test period and depending on the fault requested to be tested.

5.5 Run-Time Fault Injection

Contrary to the compile time fault injection method, only one version of the CUT is synthesized and placed and routed. Any fault that needed to be simulated will be injected by reconfiguring

some of the circuit components during run-time without affecting the operation of the other logic devices on the FPGA.

5.5.1 Run-Time Fault Injection Techniques

Faults injection using LUTs is one of the main features of using run-time reconfiguration for fault injections, as different faults can be emulated, for the same CUT, without the need to recompile or download the CUT.

5.5.1.1 LUT Input Stuck-at Fault Injection

Faults can be injected at the inputs of the LUT as shown in Figure 30. In this figure, the truth table of a 4-input LUT is shown in (a). If a stuck-at-1 fault is to be injected on the F3 input of the LUT, then the output value of F4F3F2F1 = 0100 must be written to the output of value of F4F3F2F1 = 0000, the value for F4F3F2F1 = 0101 to F4F3F2F1 = 0001, etc. Figure 30(b) shows how the LUT has to be modified to resemble such fault. The modified outputs are in bold.

F4	F3	F2	F1	Output
0	0	0	0	Y0
0	0	0	1	Y1
0	0	1	0	Y2
0	0	1	1	Y3
0	1	0	0	Y4
0	1	0	1	Y5
0	1	1	0	Y6
0	1	1	1	Y7
1	0	0	0	Y8
1	0	0	1	Y9
1	0	1	0	Y10
1	0	1	1	Y11
1	1	0	0	Y12
1	1	0	1	Y13
1	1	1	0	Y14
1	1	1	1	Y15

(a)

F4	F3	F2	F1	Output
0	0	0	0	**Y4**
0	0	0	1	**Y5**
0	0	1	0	**Y6**
0	0	1	1	**Y7**
0	1	0	0	Y4
0	1	0	1	Y5
0	1	1	0	Y6
0	1	1	1	Y7
1	0	0	0	**Y12**
1	0	0	1	**Y13**
1	0	1	0	**Y14**
1	0	1	1	**Y15**
1	1	0	0	Y12
1	1	0	1	Y13
1	1	1	0	Y14
1	1	1	1	Y15

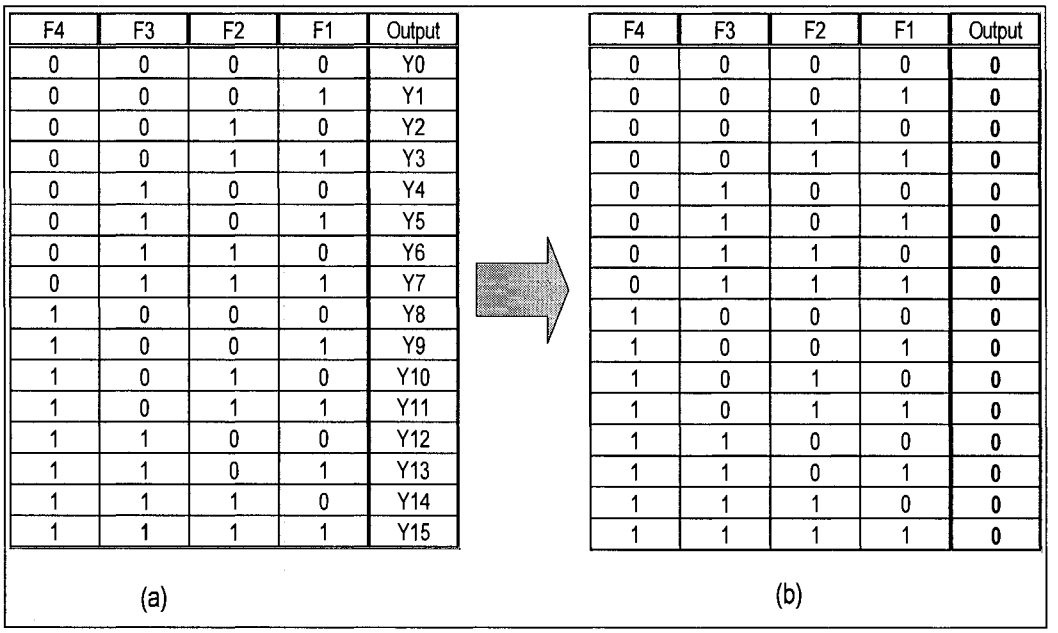
(b)

Figure 30 Injection of a stuck-at-1 fault on the F3 input of an LUT. (a) Original LUT truth table (b) modified.

The same can be applied for stuck-at-0 faults (except now output value of F4F3F2F1 = 0000 must be written to the output of value of F4F3F2F1 = 0100, etc.), and to other inputs. Note that the output column is the actual reconfiguration values of LUT.

5.5.1.2 LUT Output Stuck-at Fault Injection

To inject a stuck-at-1 or stuck-at-0 fault at the output of an LUT, then all the 16 bit positions of the LUT reconfiguration vector must be programmed to 1 or 0 respectively. Figure 31 shows the LUT representation before and after inserting the output fault.



F4	F3	F2	F1	Output
0	0	0	0	Y0
0	0	0	1	Y1
0	0	1	0	Y2
0	0	1	1	Y3
0	1	0	0	Y4
0	1	0	1	Y5
0	1	1	0	Y6
0	1	1	1	Y7
1	0	0	0	Y8
1	0	0	1	Y9
1	0	1	0	Y10
1	0	1	1	Y11
1	1	0	0	Y12
1	1	0	1	Y13
1	1	1	0	Y14
1	1	1	1	Y15

(a)

F4	F3	F2	F1	Output
0	0	0	0	**0**
0	0	0	1	**0**
0	0	1	0	**0**
0	0	1	1	**0**
0	1	0	0	**0**
0	1	0	1	**0**
0	1	1	0	**0**
0	1	1	1	**0**
1	0	0	0	**0**
1	0	0	1	**0**
1	0	1	0	**0**
1	0	1	1	**0**
1	1	0	0	**0**
1	1	0	1	**0**
1	1	1	0	**0**
1	1	1	1	**0**

(b)

Figure 31. Injection of a stuck-at-0 fault on output of an LUT. (a) Original LUT truth table (b) modified.

5.5.1.3 LUT Contents Stuck-at Fault Injection

This fault injection method is functionally driven instead of line driven as the previous two methods. The number of simulated faults in this method is identical to the possible combinations of LUT active inputs, i.e. a fault can be injected by inverting only the LUT position

that corresponds to one input combination. Total coverage of faults in this methods corresponds to the exhaustive LUT functionality test [25].

5.5.2 LUT Implementations

Xilinx Virtex-II FPGAs support only 4-input/1-output LUT primitives. Those 4-input LUTs can be used to implement functions of 0,1,2,3 and 4 inputs respectively. Multiple LUTs can be cascaded together to form one function.

LUTs are represented by 16-bit vectors. However, not all these bits are used when implementing functions with less than 4 inputs. For example, a 3-input function can be implemented by one LUT. However, only 8 bits are used to represent the function within the 16-bit LUT vector.

It is important to identify the number of inputs relevant for each LUT used in order to include the corresponding faults in the fault list. Some researchers [25] suggested the use of the bit file generated by Xilinx tools to extract such information. This can be easily done by comparing the 8-bit vector values when the relevant input is either 0 or 1. If both vectors are identical then the relative input is not active. However, and as explained in chapter 3, such information is no longer available for the Virtex-II family; however, the information can be extracted by reading back each LUT configuration vector during run-time using the ICAP interface.

Our approach is to try to extract such information in advance by controlling the way the HDL code is synthesized and LUTs are instantiated.

5.6 ISCAS-85 Benchmark Circuit C17

The C17 circuit is the smallest circuit of the ISCAS'85 benchmark circuits. It consists of 5 inputs, 2 outputs and contains 6 gates. Figure 32 shows the circuit. The CUT HDL code is wrapped by being instantiated in a top HDL module that contains the standard interface to of the active module and as was defined in 4.5.2.

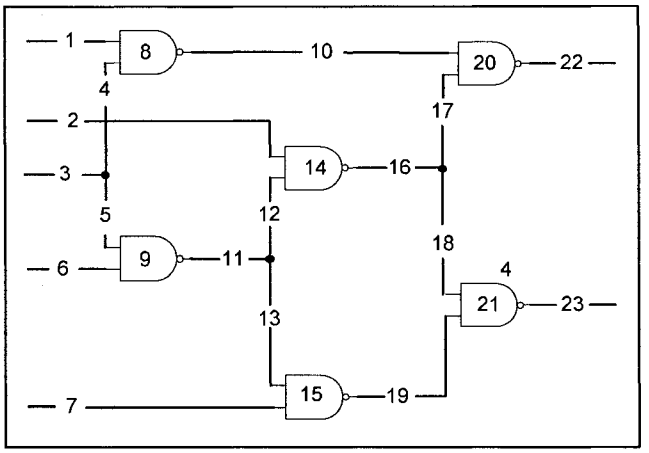


Figure 32.C17 Circuit Under Test (CUT)

Figure 33 shows the synthesis results for C17 CUT. The VHDL code is shown in Figure 34 .

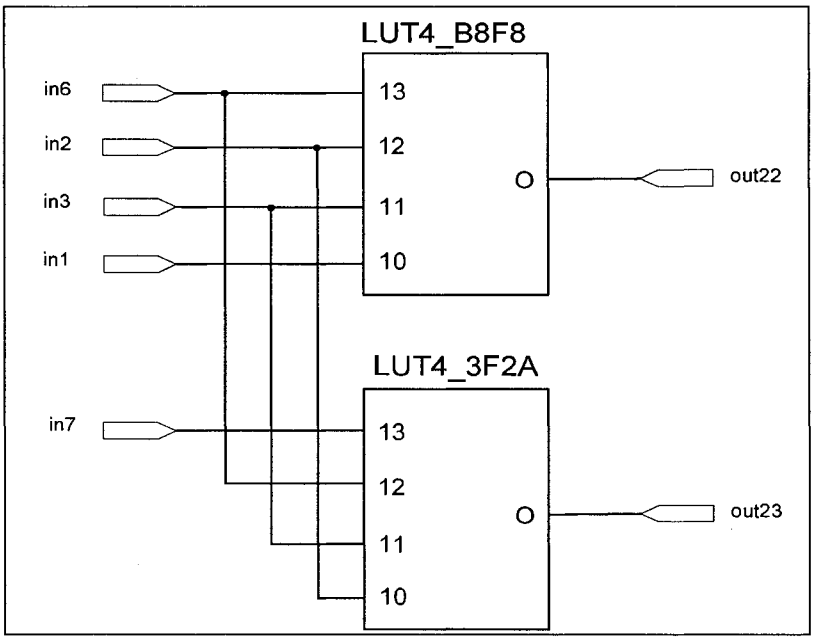


Figure 33. C17 Synthesis technology view

```
entity Circuit17 is
port( in1   : in std_logic;
      in2   : in std_logic;
      in3   : in std_logic;
      in6   : in std_logic;
      in7   : in std_logic;
      out22 : out std_logic;
      out23 : out std_logic
    );
end Circuit17;
architecture Behavioral of Circuit17 is
signal n4,n5,n10,n11,n12,n13,n16,n17,n18,n19 : std_logic;
begin
n4    <= in3;
n5    <= in3;
n10   <= not (in1 and n4);
n11   <= not (n5 and in6);
n12   <= n11;
n13   <= n11;
n16   <= not (in2 and n12);
n17   <= n16;
n18   <= n16;
n19   <= not (in7 and n13);
out22 <= not ((n10) and (n17) );
out23 <= not ((n18) and (n19) );
end Behavioral
```

Figure 34. C17 CUT VHDL Code

As seen from synthesis technology view, two LUTs were inferred, and are initialized with vectors B8F8 and 3F28 respectively.

5.6.1 HDL Code Preparation

In most cases, it will be difficult to predict how the design will be synthesized and how many LUTs the design will end up using. It gets even harder if the faults simulated are the inputs and the outputs of the LUTs, as their port doesn't really relates directly to the original HDL. While working with the C17, a better understanding, and better prediction to how the final

implementation is achieved by re-writing the original code (with no change in functionality), and constraining how the final netlist is going to be. Below some of the guidelines for how to modify the HDL code:

- Rename all the nets by adding a "_x" after the name. "x" represents how many signals on the right of the logical expression.
- Any expression that has 4 signals, on the right, in total (sum of all "x"), is replaced by an LUT.
- Update the total on left to reflect that the four signals are replaced by 1.
- Continue to update the prefixes of all the signals.
- Any output signal is replaced by an LUT.
- Add the necessary constraints (Keep, LUT_MAP etc).

Since the location of any inferred LUTs is bounded by the active area, location area constraints can be added too. A sample of the C17 VHDL code after the modification is shown in Figure 35. This code will infer LUTs as expected. In general, the added LUTs will increase the time it takes to insert faults.

Note also, from that the CUT C17 signals 16,17,18 and 19 are exposed now and faults on them can be simulated by LUT input and output stuck-at fault injection.

```
entity Circuit17 is
port( in1   : in std_logic;
      in2   : in std_logic;
      in3   : in std_logic;
      in6   : in std_logic;
      in7   : in std_logic;
      out22 : out std_logic;
      out23 : out std_logic
   );
end Circuit17;
architecture Behavioral of Circuit17 is
signal n4_1,n5_1,n10_2,n11_2,n12_2,n13_2,n16_3,n17_3,n18_3,n19_3 : std_logic;
attribute LUT_MAP: string;
attribute s: string;
signal LUT1_1 : std_logic;
signal LUT2_1 : std_logic;
signal LUT3_1 : std_logic;
signal LUT4_1 : std_logic;
attribute s of LUT1_1: signal is "yes";
attribute s of LUT2_1: signal is "yes";
attribute s of LUT3_1: signal is "yes";
attribute s of LUT4_1: signal is "yes";
n4_1 <= in3;
n5_1 <= in3;
n10_2 <= not (in1 and n4_1);
n11_2 <= not (n5_1 and in6);
n12_2 <= n11_2;
n13_2 <= n11_2;
n16_3 <= not (in2 and n12_2);
n17_3 <= n16_3;
n18_3 <= n16_3;
n19_3 <= not (in7 and n13_2);
LUT1_1 <= (n17_3);
LUT2_1 <= not ((n10_2) and (LUT1_1) );
out22    <= (LUT2_1);
LUT3_1 <= (n19_3);
LUT4_1 <= not ((n18_3) and (LUT3_1) );
out23    <= (LUT4_1);
end Behavioral;
```

Figure 35. C17 VHDL Code after Modification

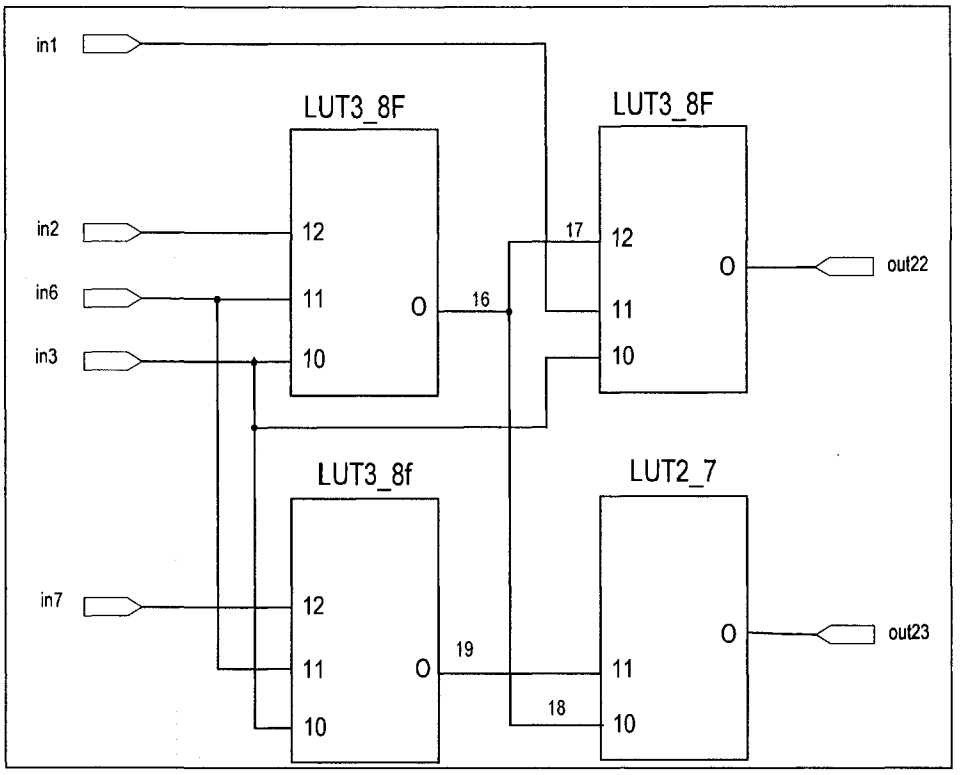


Figure 36. C17 Technology View after Code Modification

5.7 CUT Testing Flow

Testing of the C17 CUT is done by applying test patterns generated by a test pattern generator (or predefined patterns saved in a an array), through the GPIO ports of MicroBlaze processor A, to the C17 circuit and comparing the response, read through another GPIO port, to known correct response. Fault injection is done by MicroBlaze processor R. Figure 37 shows the flow for the test operation, while Figure 38 shows how both the application and the reconfiguration flows communicate.

The test patterns could be generated utilizing one of the following three methods: deterministic test pattern generator, pseudo-random test pattern generator, or exhaustive test pattern generator. The first has a known test-set, while pseudo-random test patterns could be generated

using any pseudo-random techniques. The exhaustive test implies using the whole truth table's inputs as the test-set.

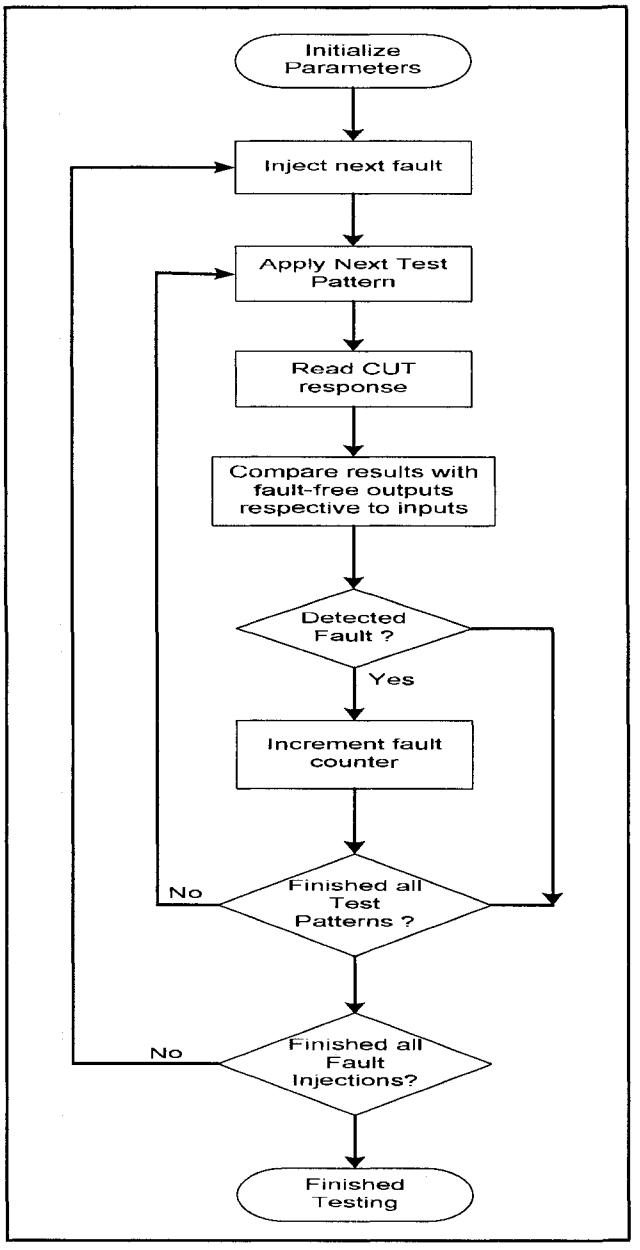


Figure 37. Test testing flow for the CUT

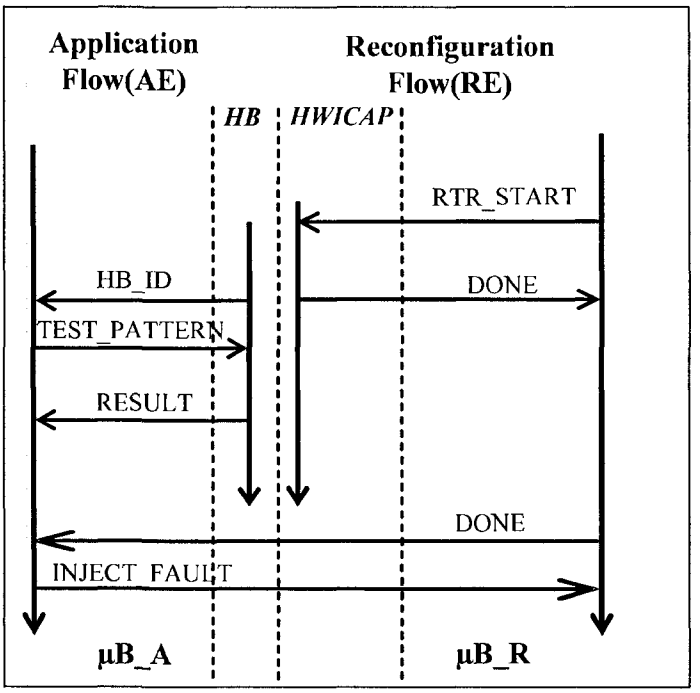


Figure 38 Applications and Reconfiguration flows.

Chapter 6: Discussion and Future Work

6.1 Conclusions and Suggested Future Work

In this thesis, a complete working platform for run-time reconfiguration was built, where a user can modify any logical resource within the FPGA. The application field we chose was the Built In Self Test. However, the same platform can be used for other applications, as the platform comes complete with the embedded processor system.

I conceived an original design flow of run-time self-reconfigurable systems that was applied to several synthesis applications at system and block level for a class of FPGAs. This design flow was validated during the implementation of the architecture of the ERACE system, and of the current testing dedicated self-reconfigurable system.

In the area of BIST, I devised and implemented a framework for testing standardized circuits (CUTs) to verify their functionality through applying certain test patterns and simulating faults within the CUT. In this work, only input/output fault injection methods were implemented. The thesis introduced several efficient realizations of an algorithm used to detect both detectable and hidden faults within CUTs. The thesis presented the sequential run-time reconfiguration (RTR) realization of the fault-model strategy, while utilizing the RTR technique to insert stuck-at faults at different wires of the CUT, and record the circuit's behaviour accordingly. The results of my work proved the feasibility of applying Run-Time-Reconfiguration architectures and techniques in efficiently development and verification of Built In Self Test algorithms.

Much of the work that I would suggest for the future depends on the results of the vendor's synthesis and the place-and-route tools. More work can be done on the LUT vector bits itself, by working to find a relation between the original code and the different faults we want to simulate. As more vendors are having interest in the run-time-reconfiguration field, it makes sense to try to have a vendor-independent flow (for example, trying to control how the design is

implemented from HDL code into LUTs). In the work presented, I explained one way of doing this, but more work is needed to prefect this method and tailor it to the kind of testing we want to perform.

One last thing to note is that all the work presented in my thesis, dealt with combinational logic circuits only. Run-time reconfiguration can be applied to Sequential circuit testing too, as the state of the flip-flop, within any CLB, can be modified along with the LUTs.

6.2 Where do we go from here?

Partial Reconfiguration is a very promising field of reconfigurable computing. However, it is still in its early stages and not progressing as fast as one would think it would. The main reason for this is not the lack of interest from research group, but mainly the lack of support from the FPGA vendors themselves. Xilinx is reluctant to release the full details of its recent devices configuration memories (Xilinx XC6200 was an exception that didn't last long). Not only this, we discovered, through the practical phase of this research, that the tools are far form being reliable enough. There were several SW tools bugs that were discovered and required a work around. Xilinx support for partial reconfiguration in their tools was as is, i.e. the tools have this feature but it is not fully tested (ISE 7.1 tool set). Having said that, things can change rapidly as the support for partial reconfiguration is offered by other FPGA vendors.

6.3 Early Access (EA) Partial Reconfiguration Flow

This is a special program that Xilinx has recently introduced (with restricted access to certain customers only), that will support partial reconfiguration flow and introduce some of the new tools designed specifically to simplify and automate the partial reconfiguration process. Some of the key differences are:

- The requirement for whole column PR regions is removed. The EA PR flow now allows for PR regions of any rectangular size.
- EA PR flow allows signals (routes) in the static module to cross through a partially reconfigurable region without the use of bus macros.

- The 8.1.01 PR flow now supports Virtex-4.

As part of this program, Xilinx recently added the support for partial reconfiguration to the latest version (v8.1) of their PlanAhead software, which is a hierarchical design and analysis software environment. This support included a graphical environment for partial reconfiguration, and a partial reconfiguration flow wizard.

This would greatly simplify the partial reconfiguration flow and makes it easier to work with designs that have Partially Reconfigurable blocks.

Appendix A: Design Scripts and Files

A.1 Introduction

This appendix lists some of the scripts used in setting up the design flow and also lists sample design files. The complete design files will be available in a CD.

A.2 PR Design Script

The following is a collection of the scripts used in each stage of the partial reconfiguration flow stages. The script assumes that the embedded processor system is already implemented using the EDK tools flow (as a module of the ISE flow). The results netlist of the EDK flow will be the static portion of the design. The active portion contains the CUT netlist that is synthesized elsewhere and imported when starting the PR flow.

```
rem The scripts below is used for Partial Reconfiguration flow
cd .. \top_level_initial\
copy .. \.. \synthesis\system_stub\system_stub.ngc . \system_stub.ngc
ngdbuild -modular initial -uc system_stub.ucf system_stub.ngc
cd .. \hw_block\
rem first active block. This script is repeated if more than one PR module.
copy .. \top_level_initial\system_stub.ucf . \system_stub.ucf
copy .. \.. \synthesis\hw_block\hw_block.ngc . \hw_block.ngc
copy .. \.. \.. \implementation\system_stub.bmm
ngdbuild -modular module -active hw_block -uc system_stub.ucf .. \top_level_initial\system_stub.ngc
-bm system_stub.bmm
map -detail system_stub.ngd
par -w system_stub.ncd system_stub1.ncd
rem saving generating .ncd file in the pim directory.
pimcreate -ncd system_stub1.ncd .. \Pim
rem creating the partial bit stream
bitgen -w -m -g ActiveReconfig:Yes -d system_stub1.ncd system_stub_partial.bit
rem Static module. This module is already been generated using EDK tools.
copy .. \top_level_initial\system_stub.ucf . \system_stub.ucf
copy .. \.. \.. \implementation\*.ngc . \
```

```
copy .. \.. \.. \implementation \*.bmm . \
rem Failing to add those command will result in an error. A Xilinx bug.
@ECHO OFF
echo INST "system_i/microblaze_a/microblaze_a/Data_Flow_I/PC_Module_I/PC_Bit_I*"
USE_RLOC=FALSE; >> system_stub.ucf
echo INST "system_i/microblaze_r/microblaze_r/Data_Flow_I/PC_Module_I/PC_Bit_I*"
USE_RLOC=FALSE; >> system_stub.ucf
@ECHO ON
rem script in tcl to change every occurance of system to system_i in system_stub.bmm file
text_replace.exe
ngdbuild -modular module -active system -uc system_stub.ucf .. \top_level_initial\system_stub.ngc -
bm system_i_stub.bmm
map -detail system_stub.ngd
par -w system_stub.ncd system_stub1.ncd
rem saving generating .ncd file in the pim directory.
pimcreate -ncd system_stub1.ncd .. \Pim
cd .. \top_level_final\
rem assmbling the final design
copy .. \.. \.. \implementation \*.bmm . \
copy .. \top_level_initial\system_stub.ucf . \system_stub.ucf
copy .. \top_level_initial\system_stub.ngc . \system_stub.ngc
rem Failing to add those command will result in an error. A Xilinx bug.
@ECHO OFF
echo INST "system_i/microblaze_a/microblaze_a/Data_Flow_I/PC_Module_I/PC_Bit_I*"
USE_RLOC=FALSE; >> system_stub.ucf
echo INST "system_i/microblaze_r/microblaze_r/Data_Flow_I/PC_Module_I/PC_Bit_I*"
USE_RLOC=FALSE; >> system_stub.ucf
@ECHO ON
rem script in tcl to change every occurance of system to system_i in system_stub.bmm file
text_replace.exe
ngdbuild -p xc2v2000-4ff896 -modular assemble -pimpath .. \Pim -use_pim hw_block -use_pim system
system_stub.ngc -bm system_i_stub.bmm
map system_stub.ngd
par -w system_stub.ncd system_stub_routed.ncd
rem generating complete .bit file (active + static)
rem  -d no DRC checking. Use this while debugging
rem  bitgen -d -m -w system_stub_routed.ncd system_stub_routed.bit
bitgen -m –w system_stub_routed.ncd system_stub_routed.bit
rem back annotating the final .bit file with SW loads
data2mem -bm ./modular_design/implementation/top_level_final/system_i_stub_bd -bt
./modular_design/implementation/top_level_final
/system_stub_routed.bit  -bd RTOS_A/executable.elf tag lmb_a opb_bram_oma  -bd
RTOS_R/executable.elf tag lmb_r  -o b
 ./modular_design/implementation/top_level_final/download.bit
copy . \modular_design\implementation\top_level_final\download.bit
. \implementation\download.bit
```

Below is the text_replace.tcl tcl script that is used to modify the system_stub.bmm file, generated by the edk tools, to use it in the PR flow:

```
#!/bin/sh
# \
exec wish "$0" ${1+"$@"}


    if {[file exists "./system_stub.bmm"]} {
       set fileid [open "./system_stub.bmm" r+]
       set fileid_i [open "./system_i_stub.bmm" w+]
    } else {
       puts "Can't find system_stub.bmm or perms are bad"
       exit
  }

seek $fileid 0 start
puts "Reading file..."
while {![eof $fileid]} {
          gets $fileid line;
 set line1 [string map {system system_i} $line];
          puts  $fileid_i $line1;
}

puts "Closing both files..."
close $fileid
close $fileid_i

exit
```

A.3 Design Constraints File

The constraints file (.ucf) is used to define the active and static modules boundaries when implementing the PR flow:

```
AREA_GROUP "AG_system" MODE=RECONFIG;
AREA_GROUP "AG_hw_block" MODE=RECONFIG;

## System level constraints
Net sys_clk PERIOD = 37037 ps;
```

```
Net sys_rst TIG;

NET "RS232_req_to_send" LOC = "B8" ;
NET "RS232_RX" LOC = "C8" ;
NET "RS232_TX" LOC = "C9" ;
NET "sys_clk" LOC = "AH15" ;
NET "sys_rst" LOC = "F14" ;

NET "board_leds<0>" LOC = "B27" ;
NET "board_leds<1>" LOC = "B22" ;

# Start of Constraints extracted by Floorplanner from the Design

INST "system_i" AREA_GROUP = "AG_system";

AREA_GROUP "AG_system" RANGE = SLICE_X2Y111:SLICE_X95Y0;
AREA_GROUP "AG_system" RANGE = TBUF_X2Y111:TBUF_X84Y0;
AREA_GROUP "AG_system" RANGE = RAMB16_X0Y13:RAMB16_X3Y0;
AREA_GROUP "AG_system" RANGE = MULT18X18_X0Y13:MULT18X18_X3Y0;

INST "dynamic" AREA_GROUP = "AG_hw_block";

AREA_GROUP "AG_hw_block" RANGE = SLICE_X0Y111:SLICE_X1Y0;
AREA_GROUP "AG_hw_block" RANGE = TBUF_X0Y111:TBUF_X0Y0;


INST "busStatToDyn" LOC = "SLICE_X0Y52";
INST "busDynToStat" LOC = "SLICE_X0Y50";

INST "busStatToDyn2" LOC = "SLICE_X0Y48";
INST "busDynToStat2" LOC = "SLICE_X0Y46";
```

A.4 Top Level VHDL File

This is the top level VHDL file where all the components are instantiated. The file is first generated by the edk tools but has to be modified to include the active module and the busmacros.

```
------------------------------------------------------------------------------------------------------
---------------------------------- system_stub.vhd -----------------------------------------
------------------------------------------------------------------------------------------------------
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library UNISIM;
use UNISIM.VCOMPONENTS.ALL;

entity system_stub is
 port (
   RS232_RX : in std_logic;
   RS232_TX : out std_logic;
   RS232_req_to_send : out std_logic;
   sys_clk : in std_logic;
   sys_rst : in std_logic;
   board_leds : out std_logic_vector(0 to 1)
       );
end system_stub;

architecture STRUCTURE of system_stub is

 component system is
  port (
    RS232_RX : in std_logic;
    RS232_TX : out std_logic;
    RS232_req_to_send : out std_logic;
    sys_clk : in std_logic;
    sys_rst : in std_logic;
    opb_gpio_A_GPIO_in : in std_logic_vector(0 to 15);
    opb_gpio3_A_GPIO_d_out : out std_logic_vector(0 to 7);
    opb_gpio2_A_GPIO_d_out : out std_logic_vector(0 to 31);
    opb_gpio_R_GPIO_d_out : out std_logic_vector(0 to 0)
                        );
  end component;

 component hw_block
    port( InBus : in std_logic_vector(0 to 39);
          led1_out : out std_logic;
          OutBus : out std_logic_vector(0 to 7);
          IDBus : out std_logic_vector(0 to 7)
         );
    end component;

component busmacro_xc2v_left2right is
   port (
   input0 : in std_logic;
```

```
    input1 : in std_logic;
    input2 : in std_logic;
    input3 : in std_logic;
    input4 : in std_logic;
    input5 : in std_logic;
    input6 : in std_logic;
    input7 : in std_logic;
    output0 : out std_logic;
    output1 : out std_logic;
    output2 : out std_logic;
    output3 : out std_logic;
    output4 : out std_logic;
    output5 : out std_logic;
    output6 : out std_logic;
    output7 : out std_logic
    );
end component;

component busmacro_xc2v_right2left is
    port (
    input0 : in std_logic;
    input1 : in std_logic;
    input2 : in std_logic;
    input3 : in std_logic;
    input4 : in std_logic;
    input5 : in std_logic;
    input6 : in std_logic;
    input7 : in std_logic;
    output0 : out std_logic;
    output1 : out std_logic;
    output2 : out std_logic;
    output3 : out std_logic;
    output4 : out std_logic;
    output5 : out std_logic;
    output6 : out std_logic;
    output7 : out std_logic
    );
end component;

  component OBUF is
    port (
      I : in std_logic;
      O : out std_logic
    );
```

```
 end component;

 component IBUF is
  port (
    I : in std_logic;
    O : out std_logic
  );
end component;

 component IBUFG is
  port (
    I : in std_logic;
    O : out std_logic
  );
 end component;

  component IOBUF is
  port (
    I : in std_logic;
    IO : inout std_logic;
    O : out std_logic;
    T : in std_logic
  );
 end component;

  -- Internal signals
 signal sys_clk_IBUFG : std_logic;
 signal sys_rst_IBUF : std_logic;
 signal led_out_dyn : std_logic;
 signal result : std_logic_vector(0 to 7);
 signal dataOut1, IDBus, OutBus : std_logic_vector(0 to 7);
 signal hw_gpio_R_GPIO_IO_pin_O : std_logic_vector (0 to 0);
 signal temp, opb_gpio_A_GPIO_in,opb_gpio2_A_GPIO_d_out : std_logic_vector (0 to 31);
 signal InBus : std_logic_vector (0 to 39);

 signal opb_gpio3_A_GPIO_d_out : std_logic_vector (0 to 7);
 signal RS232_RX_IBUF : std_logic;
 signal RS232_TX_OBUF : std_logic;
 signal RS232_req_to_send_OBUF : std_logic;
 signal dip_switch2_buf : std_logic_vector (0 to 0);


begin
 system_i : system
```

```
  port map (
    RS232_RX => RS232_RX_IBUF,
    RS232_TX => RS232_TX_OBUF,
    RS232_req_to_send => RS232_req_to_send_OBUF,
    sys_clk => sys_clk_IBUFG,
    sys_rst => sys_rst_IBUF,
    opb_gpio_A_GPIO_in => temp (0 to 15),
    opb_gpio3_A_GPIO_d_out => opb_gpio3_A_GPIO_d_out (0 to 7),
    opb_gpio2_A_GPIO_d_out => opb_gpio2_A_GPIO_d_out (0 to 31),
    opb_gpio_R_GPIO_d_out => hw_gpio_R_GPIO_IO_pin_O (0 to 0)
            );

-- done to make sure output bus is in lower 8 bits
  temp (0 to 15) <=    opb_gpio_A_GPIO_in(8 to 15) & opb_gpio_A_GPIO_in(0 to 7);

 busDynToStat : busmacro_xc2v_left2right
   port map (
   input0 => IDBus(0),
   input1 => IDBus(1),
   input2 => IDBus(2),
   input3 => IDBus(3),
   input4 => IDBus(4),
   input5 => IDBus(5),
   input6 => IDBus(6),
   input7 => IDBus(7),
   output0 => opb_gpio_A_GPIO_in(8),
   output1 => opb_gpio_A_GPIO_in(9) ,
   output2 => opb_gpio_A_GPIO_in(10),
   output3 => opb_gpio_A_GPIO_in(11),
   output4 => opb_gpio_A_GPIO_in(12),
   output5 =>  opb_gpio_A_GPIO_in(13),
   output6 => opb_gpio_A_GPIO_in(14),
   output7 => opb_gpio_A_GPIO_in(15)
   );

 busDynToStat2 : busmacro_xc2v_left2right
   port map (
   input0 => OutBus(0),
   input1 => OutBus(1),
   input2 => OutBus(2),
   input3 => OutBus(3),
   input4 => OutBus(4),
   input5 => OutBus(5),
   input6 => OutBus(6),
```

```
    input7 => OutBus(7),
    output0 => opb_gpio_A_GPIO_in (0),
    output1 => opb_gpio_A_GPIO_in(1) ,
    output2 => opb_gpio_A_GPIO_in(2),
    output3 => opb_gpio_A_GPIO_in(3),
    output4 => opb_gpio_A_GPIO_in(4),
    output5 => opb_gpio_A_GPIO_in(5),
    output6 => opb_gpio_A_GPIO_in(6),
    output7 => opb_gpio_A_GPIO_in(7)
    );


busStatToDyn : busmacro_xc2v_right2left
    port map (
    input0 => opb_gpio2_A_GPIO_d_out(0) ,
    input1 => opb_gpio2_A_GPIO_d_out(1),
    input2 => opb_gpio2_A_GPIO_d_out(2),
    input3 => opb_gpio2_A_GPIO_d_out(3),
    input4 => opb_gpio2_A_GPIO_d_out(4),
    input5 => opb_gpio2_A_GPIO_d_out(5),
    input6 => opb_gpio2_A_GPIO_d_out(6),
    input7 => opb_gpio2_A_GPIO_d_out(7),
    output0 => InBus(0),
    output1 => InBus(1),
    output2 => InBus(2),
    output3 => InBus(3),
    output4 => InBus(4),
    output5 => InBus(5),
    output6 => InBus(6),
    output7 => InBus(7)
    );

busStatToDyn2 : busmacro_xc2v_right2left
    port map (
    input0 => opb_gpio2_A_GPIO_d_out(8),
    input1 => opb_gpio2_A_GPIO_d_out(9),
    input2 => opb_gpio2_A_GPIO_d_out(10),
    input3 => opb_gpio2_A_GPIO_d_out(11),
    input4 => opb_gpio2_A_GPIO_d_out(12),
    input5 => opb_gpio2_A_GPIO_d_out(13),
    input6 => opb_gpio2_A_GPIO_d_out(14),
    input7 => opb_gpio2_A_GPIO_d_out(15),
    output0 => InBus(8),
    output1 => InBus(9),
```

```
    output2 => InBus(10),
    output3 => InBus(11),
    output4 => InBus(12),
    output5 => InBus(13),
    output6 => InBus(14),
    output7 => InBus(15)
    );

busStatToDyn3 : busmacro_xc2v_right2left
    port map (
    input0 => opb_gpio2_A_GPIO_d_out(16),
    input1 => opb_gpio2_A_GPIO_d_out(17),
    input2 => opb_gpio2_A_GPIO_d_out(18),
    input3 => opb_gpio2_A_GPIO_d_out(19),
    input4 => opb_gpio2_A_GPIO_d_out(20),
    input5 => opb_gpio2_A_GPIO_d_out(21),
    input6 => opb_gpio2_A_GPIO_d_out(22),
    input7 => opb_gpio2_A_GPIO_d_out(23),
    output0 => InBus(16),
    output1 => InBus(17),
    output2 => InBus(18),
    output3 => InBus(19),
    output4 => InBus(20),
    output5 => InBus(21),
    output6 => InBus(22),
    output7 => InBus(23)
    );

busStatToDyn4 : busmacro_xc2v_right2left
    port map (
    input0 => opb_gpio2_A_GPIO_d_out(24),
    input1 => opb_gpio2_A_GPIO_d_out(25),
    input2 => opb_gpio2_A_GPIO_d_out(26),
    input3 => opb_gpio2_A_GPIO_d_out(27),
    input4 => opb_gpio2_A_GPIO_d_out(28),
    input5 => opb_gpio2_A_GPIO_d_out(29),
    input6 => opb_gpio2_A_GPIO_d_out(30),
    input7 => opb_gpio2_A_GPIO_d_out(31),
    output0 => InBus(24),
    output1 => InBus(25),
    output2 => InBus(26),
    output3 => InBus(27),
    output4 => InBus(28),
    output5 => InBus(29),
```

```
  A149output6 => InBus(30),
  output7 => InBus(31)
  );

busStatToDyn5 : busmacro_xc2v_right2left
  port map (
  input0 => opb_gpio3_A_GPIO_d_out(0),
  input1 => opb_gpio3_A_GPIO_d_out(1),
  input2 => opb_gpio3_A_GPIO_d_out(2),
  input3 => opb_gpio3_A_GPIO_d_out(3),
  input4 => opb_gpio3_A_GPIO_d_out(4),
  input5 => opb_gpio3_A_GPIO_d_out(5),
  input6 => opb_gpio3_A_GPIO_d_out(6),
  input7 => opb_gpio3_A_GPIO_d_out(7),
  output0 => InBus(32),
  output1 => InBus(33),
  output2 => InBus(34),
  output3 => InBus(35),
  output4 => InBus(36),
  output5 => InBus(37),
  output6 => InBus(38),
  output7 => InBus(39)
  );

dynamic: hw_block
      port map(InBus    => InBus(0 to 39),
          led1_out => dataOut1(1),
          OutBus   => OutBus (0 to 7),
          IDBus    => IDBus(0 to 7) );


 obuf_130 : OBUF
  port map (
   I => dataOut1(1),
   O => board_leds(0)
  );

 obuf_132 : OBUF
  port map (
   I => hw_gpio_R_GPIO_IO_pin_O (0),
   O => board_leds(1)
  );

 ibuf_0 : IBUF
```

```
 port map (
  I => RS232_RX,
  O => RS232_RX_IBUF
 );

obuf_1 : OBUF
 port map (
  I => RS232_TX_OBUF,
  O => RS232_TX
 );

obuf_2 : OBUF
 port map (
  I => RS232_req_to_send_OBUF,
  O => RS232_req_to_send
 );

ibufg_3 : IBUFG
 port map (
  I => sys_clk,
  O => sys_clk_IBUFG
 );

ibuf_4 : IBUF
 port map (
  I => sys_rst,
  O => sys_rst_IBUF
 );

end architecture STRUCTURE;
```

BIBLIOGRAPHY

[1]. G. Estrin. "Reconfigurable Computer Origins: The UCLA Fixed Plus-Variable (F+V) Structure Computer", IEEE Annals of the History of Computing, pp. 3-9, Oct/Dec 2003.

[2]. Katherine Compton, Scott Hauck, "An Introduction to Reconfigurable Computing". A research funded in part by Defence Advanced Research Projects Agency (DARPA).
http://www.ece.wisc.edu/~kati/Publications/Compton_ReconfigIntro.pdf

[3]. Lörinc Antoni, Régic Leveugle, Béla Fehér, "Using run-time reconfiguration for fault injection applications". IEEE Instrumentation and Measurement Technology Conference, Budapest, Hungary, May 21-23, 2001.

[4]. Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer, "Fault injection techniques and tools". IEEE Computer, vol. 30, no. 4, pp 75-82, 1997.

[5]. Rami Abielmona, Voicu Groza, Arkan Khalaf, "Run-Time Reconfigurable Built-In-Self-Test", Canadian Conference on Electrical and Computer Engineering (CCECE), Ottawa, Canada, May 7-10, 2006.

[6]. M. H. Assaf, R. S. Abielmona, P Abolghasem, S. R. Das, E. M. Petriu, V. Groza, and M. Sahinoglu, "Implementation of embedded cores-based digital devices in JBits java simulation environment", Proceeding of CIT 2004, pp. 315-325, 2004.

[7]. V. Groza, R. Abielmona, M. Elbadri, M. El-Kadri and A. Khalaf, "System On-Chip with a Run-Time Reconfigurable Processor", the 7th International Conference on Technical Informatics, June 8-9, 2006, Timisoara, Romania.

[8]. V. Groza, Nizar Sakr, Mohammed Elbadri, "Run-Time Reconfigurable System-on-Chip", IMTC 2005 – Instrumentation and Measurement Technology Conference, Ottawa, Canada, 17-19 May 2005.

[9]. V. Groza, R. Abielmona, M. El-Kadri, M. Elbadri, N. Sakr, "A Reconfigurable Co-Processor for Adaptive Embedded Systems," WISES 2004 - Second Workshop on Intelligent Solutions in Embedded Systems, ISBN 3-902463-00-7, pp. 105-116, June 25, 2004 - Graz, Austria.

[10]. Scott McMillan and Steven A. Guccione, "Partial Run-Time Reconfiguration Using JRTR". Xilinx Inc., 2100 Logic Drive, San Jose, California 95124. http://www.io.com/~guccione/Papers/JRTR/partial.pdf .

[11]. Xilinx Inc. XC6200 Development System Datasheet, 1997

[12]. Peter Clarke, "Xilinx wraps up work at reconfigurable operation". EE Times, May 22, 1998, http://www.eet.com/news/98/1008news/wraps.html .

[13]. Jean J. Labrosse, Use an RTOS on Your next MicroBlaze Based Product. Embedded Magazine, March 2005, http://www.xilinx.com/publications/magazines/emb_01/xc_pdf/p50-52_emb1-micrium.pdf.

[14]. Xilinx Systems, "Development system Reference Guide", Chapter 4: Modular Design, http://toolbox.xilinx.com/docsan/xilinx7/books/docs/dev/dev.pdf.

[15]. Grégory Mermoud, "A module based Dynamic Partial Reconfiguration", Logic Systems Laboratory, Ecole Polytechnic Fédérale de Lausanne, Nov 1, 2004, http://ic2.epfl.ch/~gmermoud/files/publications/DPRtutorial.pdf.

[16]. University of Kansas, "Creating a Partially Reconfigurable Design with Xilinx EDK (modular design with EDK)", A tutorial. http://wiki.ittc.ku.edu/rtrjvm/EDK_and_MD.

[17]. Andres Upegui and Eduardo Sanchez, "Evolving Hardware by Dynamically Reconfiguring Xilinx FPGAs", Ecole Polytechnique Fédérale de Lausanne – EPFL, http://lslwww.epfl.ch/~upegui/docs/Upegui_ICES05.pdf.

[18]. Xilinx Systems, "Virtex-II Platform FPGA User Guide", Chapter 4: Configuration, UG002 (v2.0) 23 March 2005, http://www.xilinx.com/bvdocs/userguides/ug002.pdf

[19]. Shannon, L, "Leveraging reconfigurability in the design process", International Conference on Field Programmable Logic and Applications, Aug. 24-26, 2005.

[20]. Xilinx Integrated Software Environment (ISE), v7.1i, http://www.xilinx.com/ise/logic_design_prod/foundation.htm.

[21]. Xilinx Systems, "Virtex Series Configuration Architecture User Guide", Application Note XAPP151, V1.7, October 20, 2004, http://www.xilinx.com/bvdocs/appnotes/xapp151.pdf.

[22]. Wo, D.; Forward, K.; "Compiling to the gate level for a reconfigurable co-processor FPGAs for Custom Computing Machines," 1994. Proceedings. IEEE Workshop on 10-13 April 1994 Page(s):147 – 154.

[23]. Xilinx Corp., "MicroBlaze and Multimedia development board user guide", Internet, August 2002, http://www.xilinx.com/bvdocs/userguides/ug020.pdf.

[24]. Peeter Ellervee, Jaan Raik, Valentin Tihhomirov, Kalle Tammemäe, "Evaluating Fault Emulation on FPGA", Dept. of Computer Engineering, Tallinn University of

Technology, Raja 15, 12618 Tallinn, Estonia, http://www.pld.ttu.ee/~jaan/PDF/p100.pdf.

[25]. Abilio Parreira, J. P. Teixira and Marcelino Santos, "A Novel Approach to FPGA-Based Hardware Fault Modeling and Simulation", 2001, IST/INESC-ID, R. Alve Redol, 9, 1000-029 Lisboa, Portugal. http://www.inesc-id.pt/pt/indicadores/Ficheiros/1232.pdf.

[26]. Xilinx Systems, "Virtex-4 Configuration Guide UG071". V1.4, January 24, 2006. http://www.xilinx.com/bvdocs/userguides/ug071.pdf.

[27]. S. R. Das, M. H., Assaf, E. M. Petriu, W. B. Jone and K. Chakrabarty, "A novel approach to designing aliasing-free space compactors based on switching theory formulation", Proc. IEEE Instrum. Meas. Tech. Conf., pp 198-203, 2001.

[28]. Xilinx Systems, "OPB HWICAP", Data Sheet 280, v1.3, March 15, 2004. http://www.xilinx.com/bvdocs/ipcenter/data_sheet/opb_hwicap.pdf.

[29]. Franz Brglez and Hideo Fujiwara. "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Simulator in FORTRAN". In Proceedings International Symposium on Circuits and Systems (ISCAS), pages 695–698, Kyoto, Japan, May 1985.

[30]. Erik J. Marinissen, V. Iyengar, K. Chakrabarty, A set of Benchmarks for Modular Testing of SOCs". IEEE Int. Test Conference, Baltimore, MD, USE – Oct 2002. http://www.hitech-projects.com/itc02socbenchm/papers/itc02paper-corrected.pdf.

[31]. Brandon Blodger, Philip James-Roxby, E. Keller, Scott McMillan and Prasanna Sundarajan, "A self-reconfiguring platform". Field Programmable Logic and Application, 13th International Conference, FPL 2003, Lisbon, Portugal, September 1-3, 2003. http://www.crhc.uiuc.edu/ece497nc/papers/self_reconf.pdf.

[32]. Xilinx Systems, Xilinx Embedded Development Kit (EDK), v7.1i, 2005. http://www.xilinx.com/ise/embedded_design_prod/platform_studio.htm.

[33]. Xilinx Systems, "System ACE Compact Flash Solution", DS080 (v1.5), April 5, 2002, http://www.xilinx.com/bvdocs/publications/ds080.pdf.

[34]. Xilinx Systems, "OPB SYSACE (System ACE) Interface Controller, DS453 December 2, 2005. http://www.xilinx.com/bvdocs/publications/ds453.pdf.

[35]. J. Burns, A. Donlin, J. Hogg, S. Singh, M. De Wit, "A Dynamic Reconfiguration Run-Time System", *fccm*, p. 66, 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM '97), 1997. http://www.dcs.gla.ac.uk/~dew/fccm97.pdf.

[36]. Xilinx Systems, MicroBlaze Processor Reference Guide, UG081, June 1, 2006, http://www.xilinx.com/ise/embedded/mb_ref_guide.pdf.

[37]. Hauser, J. R. And Wawrzynek, J. "Garp: A MIPS processor with a reconfigurable coprocessor," IEEE Symposium on Field-Programmable Custom Computing Machines, 12–21, 1997.

[38]. Xilinx Systems, "Virtex-5 Configuration Guide". V1.1, May 12, 2006. http://www.xilinx.com/bvdocs/userguides/ug191.pdf.

[39]. K. Compton, S. Hauck: 'Reconfigurable Computing: A Survey of Systems and Software'; Northwestern University, May 2002 http://www.ee.washington.edu/faculty/hauck/publications/ConfigCompute.pdf

[40]. S. Trimberger, D. Carberry, A. Johnson and J Wong. A time-multiplexed FPGA. In Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines, April 1997.

[41]. Chameleon Systems, Inc. CS2000 reconfigurable processor. CS2000 Advance product information 2000.

[42]. "XPP64-A1 Reconfigurable Processor – Datasheet", Rev.1.1.PACT XPP Technologies AG.

[43]. G. Van Den Branden, G.e Braeckman, A. Touhafi, "Commercially available Reconfigurable Components", Erasmushogeschool Brussel, Belgium, January 13, 2004.

[44]. João M. P. Cardoso and Mário P. Véstias, "Architectures and Compilers to Support Reconfigurable Computing". Computer Architecture, spring 1999.

[45]. Plessl, C., Platzner, M., "TKDM - a reconfigurable co-processor in a PC's memory slot," IEEE International Conference on Field-Programmable Technology (FPT), 2003. 5-17 Dec. 2003, pp. 252- 259. http://www.tik.ee.ethz.ch/~plessl/publications/fpt03/fpt03.pdf.

[46]. Hauck, S., Fry, T. W., Hosler, M. M., And Kao, "The Chimaera reconfigurable functional unit," IEEE Symposium on Field-Programmable Custom Computing Machines, 87–96, 1997

[47]. S. Hauck, Borriello and C. Ebeling, "TRIPTYCH: An FPGA Architecture with Integrated Logic and Routing", *Advanced Research in VLSI and Parallel Systems: Proceedings of the1992 Brown/MIT Conference*, pp. 26-43, March, 1992.

[48]. Mark Zwolinski, "System Design Using VHDL". Prentice Hall, 2000.

[49]. A. Steininger, "Testing and Built-in Self-Test – A Survey", Journal of systems Architecture. Volume 46, Issue 9, pages 721-747, July, 2000.