UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
COMPUTER ENGINEERING

CONRADO PILOTTO

# A Fault Injection Platform Based on Dynamic Partial Reconfiguration

Graduation Project

Prof.Dr. Fernanda Lima Kastensmidt
Advisor

Porto Alegre, Jun 2012

**CIP – CATALOGING-IN-PUBLICATION**

Pilotto, Conrado

A Fault Injection Platform Based on Dynamic Partial Reconfiguration / Conrado Pilotto. – Porto Alegre: PPGC da UFRGS, 2012.

84 f.: il.

Final Report (Master) – Universidade Federal do Rio Grande do Sul. Computer Engineering, Porto Alegre, BR–RS, 2012. Advisor: Fernanda Lima Kastensmidt.

1. Fault Injection. 2. Dynamic Partial Reconfiguration. 3. FPGAs. I. Lima Kastensmidt, Fernanda. II. Título.

# CONTENTS

# LIST OF ABBREVIATIONS AND ACRONYMS

| | |
|---|---|
| ACE | Advanced Configuration Environment |
| API | Application Programming Interface |
| ASIC | Application Specific Integrated Circuit |
| BRAM | Block RAM |
| CCLK | Configuration Clock |
| CE | Clock Enable |
| CLB | Configurable Logic Block |
| CLK | Clock |
| COTS | Commercial Off-The-Shelf |
| CRC | Cyclic Redundancy Check |
| CUT | Circuit Under Test |
| DAQ | Data Acquisition |
| DCM | Digital Clock Manager |
| DDR | Double Data Rate |
| DMA | Direct Memory Access |
| DPR | Dynamic Partial Reconfiguration |
| DUT | Device Under Test |
| FF | Flip-Flop |
| FIFO | First In, First Out |
| FPGA | Field Programmable Gate Array |
| FSM | Finite State Machine |
| FTP | File Transfer Protocol |
| HDL | Hardware Description Language |
| HID | Human Interface Device |
| I/O | Input/Output |
| ICAP | Internal Controller Access Port |

| | |
|---|---|
| ID | Identification |
| IEEE | Institute of Electrical and Electronics Engineers |
| IOB | I/O Block |
| IP | Intellectual Property |
| ISP | In System Programming |
| JTAG | Joint Test Access |
| LFSR | Linear Feedback Shift Register |
| LUT | Look-Up Table |
| MSB | Most Significant Bit |
| N/A | Not Applicable |
| PC | Personal Computer |
| PCI | Peripheral Component Interconnect |
| PHY | PHYsical layer |
| PLD | Programmable Logic Device |
| PROM | Programmable Read-Only Memory |
| RAM | Random Access Memory |
| ROM | Read-Only Memory |
| RTL | Register-Transfer Level |
| SR | Set/Reset |
| SRAM | Static Random-Access Memory |
| TMR | Triple Modular Redundancy |
| UART | Universal Asynchronous Receiver/Transmitter |
| USB | Universal Serial Bus |
| VHDL | Very High Speed Integrated Circuit Hardware Description Language |
| XHWIF | Xilinx HardWare InterFace |

# LIST OF FIGURES

# LIST OF TABLES

# ABSTRACT

The use of SRAM-based Field Programmable Gate Arrays (FPGA) in aerospace applications is becoming more popular as the silicon industry delivers new products capable of hosting entire digital systems inside a single device. The capability to perform remote reconfiguration provides the possibility to extend the duration of the missions by updating the obsolete modules as required. Unfortunately, SRAM based FPGAs are extremely sensitive to faults induced by radiation particles present in high-altitude environments. Several techniques have been proposed to mitigate the effect of those particles, but in order to successfully guarantee the correctness of a project, engineers must validate the robustness of the design when in the presence of faults. One of the possible validation methods is called fault injection.

Fault injection platforms developed in the past have significant drawbacks related to intrusiveness, complexity and costs. This work focuses on defining a fault injection platform based on high-speed, low cost and non-intrusive aspects, associating the hardware prototyping performance with the partial reconfigurability of the Xilinx Virtex FPGAs.

Dynamic Partial Reconfiguration is explored as a way of inserting bit flips inside the device configuration memory, thus inducing the same effect of a radiation particle hitting the silicon substrate. Experimental results validate the proposed work using commercial available prototyping boards.

**Keywords:** Fault Injection, Dynamic Partial Reconfiguration, FPGAs.

**Uma Plataforma para Injeção de Falhas baseada em Reconfiguração Dinâmica Parcial**

# RESUMO

O uso de FPGAs em projetos aeroespaciais tem se tornado uma prática freqüente, em parte pelo crescente avanço tecnológico que possibilita a indústria de semicondutores a integrar um número cada vez maior de funcionalidades em um único dispositivo, mas principalmente por sua capacidade de reprogramação. A possibilidade de efetuar reprogramação remota de dispositivos em órbita permite que o tempo das missões seja estendido através da atualização de módulos obsoletos. Contudo, FPGAs baseados em células de memória SRAM são extremamente sensíveis à falhas induzidas por partículas de radiação presentes no espaço. Várias técnicas foram desenvolvidas para atenuar o efeito destas partículas, mas para garantir o funcionamento correto do projeto, os engenheiros devem validar a robustez do circuito sob a presença de falhas. Uma das formas de validação é chamada de injeção de falhas.

Plataformas de injeção de falhas desenvolvidas no passado têm inconvenientes relacionados a intrusividade, complexidade e custo. Este trabalho foca na definição de uma plataforma não intrusiva, de alta velocidade e baixo custo, que associa o alto desempenho da prototipação em hardware com a capacidade de reconfiguração dinâmica dos FPGAs da família Xilinx Virtex.

A técnica de reconfiguração dinâmica parcial é usada para inverter o valor lógico das células de memória dos dispositivos FPGAS, a fim de reproduzir o mesmo efeito causado pela radiação. Resultados experimentais validam o trabalho proposto usando placas de desenvolvimento disponíveis no mercado.

**Palavras-chave:** Injeção de Falhas, Reconfiguração Dinâmica Parcial, FPGAs.

# 1 INTRODUCTION

Since the last decade, the silicon industry has presented numerous advances in the fabrication processes of the semiconductor components. These technological improvements have lead to a higher packing device density, allowing manufactures to integrate more functionality in their products. One of the applications that has been directly favored by this is the SRAM-based Field Programmable Gate Arrays (FPGAs). Since then, the use of this kind of FPGAs in embedded applications became an uprising trend, not only for they low cost and high flexible architecture, but mainly for they reprogrammability, which represents reduced time-to-market and fast turnaround time. This characteristic allows design changes and upgrades depending on variations off the mission requirements, thus providing the possibility to test developed applications on remote systems with minimal or no re-engineering cost.

However, the same technology that made feasible all this progress also brought some new challenges to system designers. The transistor scaling, reduced voltage operation and higher frequencies utilization have significantly reduced the reliability of integrated circuits by making them more susceptible to faults induced by sources of noise and crosstalk (A. Johnston 2000). Other than that, the radiation present in the space environment, and recently neutrons present in the high-altitude atmosphere, constitute an important source of errors (E. Normand 1996). When an energized particle strikes the silicon, it produces an ionization process that generates a transient current pulse that can either be misinterpreted as a valid signal by the circuit, or change the value of a memory cell (D. Alexandrescu et al 2002). Along with these, faults concerning nanometer technology FPGAs can also originate from electromigration processes, from time-dependent dielectric breakdown and from manufacturing precision variability. As a consequence, raises the necessity of fault-tolerance techniques as a way to mitigate those faults for high dependable systems under operation of hostile environments, such as space and avionics (J. Ritter 1990).

In general, there are two ways to implement fault-tolerance in SRAM-based FPGAs (F. Kastensmidt et al 2004). The first option is to design a new FPGA architecture with elements that are capable of handling these faults. This approach, however, involves high costs and demands a great design effort, which varies accordingly to the time spent, number of personnel required and foundry technology employed. The second possibility is to add redundant logic in the high-level description of the circuit, mapped to the currently FPGA architecture. This way, off-the-shelf devices may be used and custom strategies may be implemented to fulfill the requirements of the project, such as the overhead in area, performance and power dissipation. Currently, due to commercial constraints, the last is the industry standard approach.

## 1.1 Motivation

Independently of the adopted fault-tolerant strategy, it is fundamental to define a proper mechanism to assert the dependability degree of a particular design in the presence of faults. Fault injection is a well known and consolidated technique used to achieve this (M. Hsueh et al 1997 ). It consists on reproducing the same effect a fault would produce, but in a controlled environment in which is possible to evaluate its consequences and measure its impact on the systems robustness.

There are two main paradigms to approach fault injection (P. Folkesson et al 1998): hardware and software based. On the hardware side, faults can be generated by memory corruption, heavy ion injection, power supply disturbances (J. Karlsson et al 1991) and laser exposure (V. Pouget et al 2008). All these techniques, tough, make used of particularly complex processes and expensive equipment. On the software side, this is achieved early in the design process by injecting faults into the high level simulation models of the circuit. Considering different levels of abstraction (M. Shokrolah-Shirazi et al 2008) describe various techniques to perform such experiments. However, as mentioned in (R. Leveugle 1999), the main drawback related to this last approach is the huge amount of time required to run the simulation when large sets of faults need to be analyzed.

As a mean to overcome time limitations imposed by traditional simulation, it has been proposed to take advantage of hardware prototyping to perform fault injection campaigns, using FPGAs as fault emulators. (C. Lopez-Ongil et al 2007) showed that partial reconfiguration is a effective approach to fault emulation in FPGA-based platforms. By transferring the simulation process of combinational circuits to actual hardware, a speed-up of two orders of magnitude (A. Parreira et al 2004) has been indicated over software simulation , while a three-order magnitude gain has been reported (P. Kenterlis et all 2006 ) for sequential circuits.

Based on that, this work focuses on associating the hardware prototyping speed and the partial reconfigurability of modern FPGAs to propose a fault injection platform based on high speed, low cost and non-intrusive aspects to be used in future experiments conducted by the Microelectronics Group at the Federal University of Rio Grande do Sul.

## 1.2 Work Plan

The proposed platform is based on FPGA-based fault emulation by means of dynamic partial reconfiguration (DPR). DPR is a feature present in Xilinx Virtex architecture, which provides the flexibility to reconfigure portions of the circuit without having to reset or interrupt the entire system. This way, fault injection can be accomplished by reconfiguring only bits at a time, thus emulating the effect of bit flip in the device configuration memory. As faults can be injected in the FPGA at any time during circuit operation, it is possible to evaluate not only the effect of single faults, but also the effect of fault accumulation over time. Since DPR is an intrinsic capability of Virtex FPGAS, there is no need for any kind of modification in the original circuit description, neither for complex and expensive equipment. As a consequence, the overall cost required to achieve fault injection is much lower if compared to other techniques.

The project has two main goals to achieve: design i) a system capable of performing the dynamic partial reconfiguration on a ii) second designed system capable of being dynamic partially reconfigured. These will be accomplished by means of two separated, but interconnected prototyping boards, each one with its FPGA device.

There are many challenges involved when designing a complex project such as this. From specification until the final prototype, a series of implementation steps must be performed, along with validation and simulation. The following is a list of the major project phases:

1. Understand the Virtex Family FPGA architecture and its configuration interfaces.
2. Understand the dynamic partial reconfiguration process, tool-chain and design flow.
3. Develop a configuration core responsible for performing the dynamic partial reconfiguration of a device under test.
4. Design an embedded system comprised of a microprocessor, local memory and I/O interfaces in order to host the configuration core and act as a configuration master.
5. Code low level drivers to control the reconfiguration core from the embedded microprocessor.
6. Design a dynamic partial reconfigurable system to serve as circuit under test.
7. Integrate the whole system, connecting the configuration master and device under test, each residing on a different COTS board.
8. Validate the proposed platform architecture by means of experimental tests.

## 1.3 Outline

In order to present the concepts of this work in a organized and gradual way, this document is organized in eight chapters. Chapter 2 introduces the concept of FPGAs and presents the basic internal architecture of the Virtex family devices. Moreover, it presents an explanation of the main logic resources used for implementing sequential and combinatorial circuits, along with the different modules available to extend functionality and increase performance.

In chapter 3 a detailed explanation of the configuration process is presented. All the relevant aspects of the configuration mechanism are explained, such as interfaces, modes and file formats. Also, in this chapter, the principles of dynamic partial reconfiguration are presented, together with the available methodologies used to achieve a partial reconfigurable design.

Chapter 4 contains an overview of the related work, and a comparison of the current academic methods used to perform fault injection using FPGA reconfiguration.

Chapter 5 introduces the proposed fault injection platform, presents information regarding its architecture and explains the design of each module. After, chapter 6 details the implementation process of the platform prototype. In chapter 7, a test plan composed of a simple circuit hardened with a traditional fault tolerant technique is proposed as a way to evaluate the platform's functionality.

Chapter 8 contains the experimental data obtained during platform validation, together with a discussion of the results. Chapter 9 presents the overall conclusions and proposes future works that can be conducted in order to extend and improve the functionality of the platform.

# 2 THE VIRTEX-4 FPGA

## 2.1 Introduction

The concept of FPGAs was first conceived back in the early 80's with the intent to design a new programmable logic device (PLD) focused on flexibility. At that time, the other PLDs used a more rigid interconnection scheme, making then only suitable for glue-logic in most of the cases. Even if the first FPGAs were relatively small, as the silicon industry started to develop new techniques for high density devices, the upcoming FPGA families were capable of implementing entire digital systems. Thus, the use of FPGAs for rapid prototyping become quickly a world-class trend. Today, the industry is dominated by two main vendors, Xilinx Inc. and Altera Corp., although there are a number of other companies offering more niche-specific products (J. Edwards 2006).

All FPGA devices have a similar basic logical structure, formed by an array of logic blocks and routing channels interconnected by a switch matrix, as show in figure 2.1. Inside the routing channels is a set of wire segments, whose size vary accordingly to the technology. By configuring the switch matrix, these wires are connected together to form signal paths between logic blocks. The actual content of a logic block vary from vendor to vendor, but all of them are based on a look-up table (LUT) and a memory element for sequential logic. LUTs are programmable tables capable of implementing combinatorial logic and are usually comprised of four inputs, although state-of-the art products already use wider ones. This way, by grouping several logic blocks together using the switch matrix, it is possible to define any combinatorial and sequential circuit, customizing the device to behave as intended.

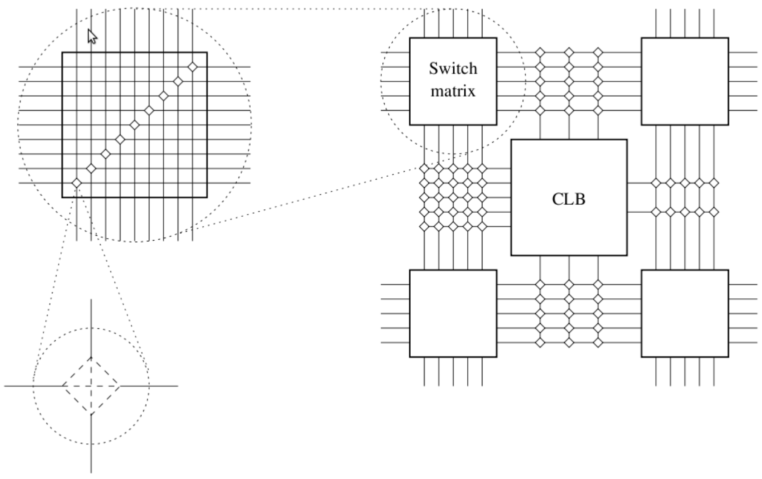


Figure 2.1: Generic FPGA Structure

Customizing an FPGA involves, therefore, configuring the logic blocks and the appropriated interconnection between then with a series of commands and application-specific data. This process is done through a configuration file, called bitstream, that contains all information the device needs to be properly programmed. Inside the FPGA, this information is stored using one of two different approaches: non-volatile or volatile. The first one uses anti-fuse or flash memory technology to retain the bitstream information, even when the device is unpowered. The second makes use of SRAM cells. While this means that the customization is lost once the FPGA has been turned off, this also implies in the intrinsic capability of reconfiguring it an unlimited number of times.

This work will focus on the Xilinx Virtex-4 family. This is a well established device in the market, and was chose due to its low price, high performance and DPR capability, though any device of the Virtex family may be used for the purpose.

## 2.2 Configurable Logic Blocks

In order to build combinatorial and sequential logic designs, the Virtex-4 architecture is organized in array structure composed of configurable logic blocks (CLB). Each CLB element is linked to the switch matrix in order to access the general routing matrix, and is comprised of 4 similar slices, as shown in Figure 2.2.

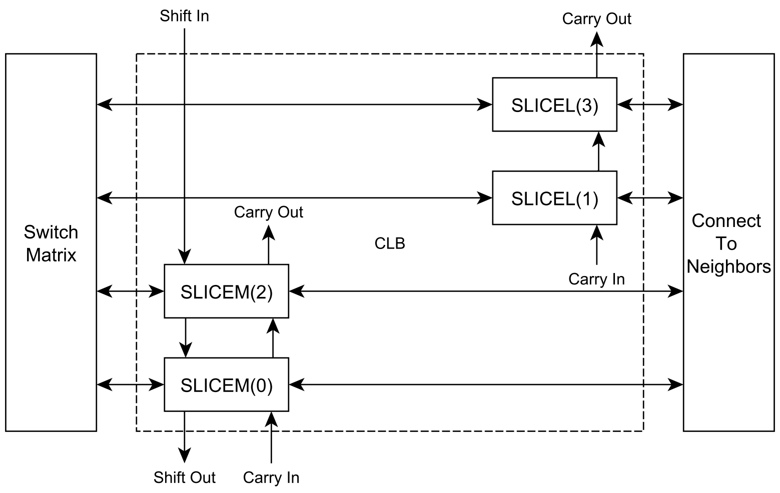


Figure 2.2: Virtex-4 CLB Element

These slices are grouped together in pairs, and each one of these pairs is located in a column. When the slice is in the left column, it is said to be SLICEM. Similarly, slices in the right column are said to be SLICEL. Each pair in a given column has an independent carry chain, and those in the left columns also have a shared shift chain (Xilinx DS112). Table 2.1 summarizes the available logic resources in each CLB.

| Slices | LUTs | Flip-Flops | Function Multiplexers | Arithmetic Carry Chains | Distributed RAM(1) | Shift Registers(1) |
|---|---|---|---|---|---|---|
| 4 | 8 | 8 | 8 | 2 | 64 bits | 64 bits |

Notes: (1) SLICEM only

Table 2.1: CLB Logic Resources

## 2.3 Slice Architecture

As mentioned before, slices pairs of type SLICEM and SLICEL have both common and unique resources. Each slice, regardless of its type, includes two four-input function generators, carry logic, arithmetic logic gates, function multiplexers and two storage elements. These elements are used by both slice types to provide logic and arithmetic functions, as well as ROM functionality. Other than this, slices of type SLICEM also support two additional functions: storing data using distributed RAM and shifting data using 16-bit shift registers.

A diagram of the SLICEM is shown in Figure 2.3 and represents the set of all elements and connections found in a slice. SLICEL is shown in Figure 2.4.

#### *2.3.0.1 Look-Up Table*

Virtex-4 function generators are implemented as 4-input LUTs. Four independent inputs are provided to each one of the two function generators in a slice. These function generators are capable, each, of implementing any arbitrarily defined boolean function of four variables. Therefore, the propagation delay is independent of the function implemented. Signals originating from the function generators can exit the slice, feed internal arithmetic logic gates, feed the input carry-logic, feed the D input of the storage element or go to the function multiplexers. These multiplexers are used to combine up to eight LUTs to provide any function of five, six, seven, or eight inputs in a single CLB. Wide multiplexers can combine LUTs within the same, or from different CLBs, making logic functions with even more input variables.

#### *2.3.0.2 Storage Elements*

The storage elements in a Virtex-4 FPGA slice can be configured in two different ways: either as edge-triggered D-type flip-flop, or as a level-sensitive latch. The D input of each flip-flop can be driven by the output of a LUT, or directly by the slice input, bypassing the logic function of the function generators. All control signals (CLK, CE and SR) are common to both storage elements in the same slice. The initial state of each storage element, following device configuration, is defined by an individual initialization attribute. For each slice, the set and reset control signals can be synchronous or asynchronous.

#### *2.3.0.3 Distributed RAM*

Multiple LUTs residing in SLICEMs can be combined together to store larger amounts of data. This is possible since each function generators in a SLICEM can be arranged to implemented a 16 x 1-bit synchronous RAM element called Distributed RAM. The possible configurations of distributed RAM elements inside a CLB are:

- Single-Port 16 x 4-bit RAM
- Single-Port 32 x 2-bit RAM
- Single-Port 64 x 1-bit RAM
- Dual-Port 16 x 2-bit RAM

Distributed RAM modules have, by default, synchronous write and asynchronous read signals. However, synchronous read operations can be achieved by connecting the RAM output to the input of the storage element in the same slice. No additional control is necessary since both share the same clock line.

##### *2.3.0.4 Read Only Memory*

Each function generator in SLICEM or SLICEL can implement a 16 x 1-bit ROM, with contents being loaded at device initialization. These elements can be cascaded in order to implement wider and/or deeper memories. Table 2.2 shows the number of LUTs occupied by each possible configuration:

| ROM | Number of LUTs |
|---|---|
| 16 x 1 | 1 |
| 32 x 1 | 2 |
| 64 x 1 | 4 |
| 128 x 1 | 8 |
| 256 x 1 | 16 (2 CLBs) |

Table 2.2: ROM Configuration

##### *2.3.0.5 Shift Registers*

The function generator in a SLICEM can also be configured as a 16-bit shift register. This way, each LUT can delay serial data from one up to 16 clock cycles. Moreover, the SHIFTIN and SHIFTOUT lines can be used to connect other LUTs and form even deeper shift registers. The four LUTs in the SLICEM of a single CLB can be cascaded to produce delays up to 64 clock cycles. It is also possible to combine shift registers across different CLBs.

The configurable 16-bit shift register write operation is synchronous with a CLK input and an optional CE, but the read operation is asynchronous by default. A storage element can, however, be used to provide a synchronous read.

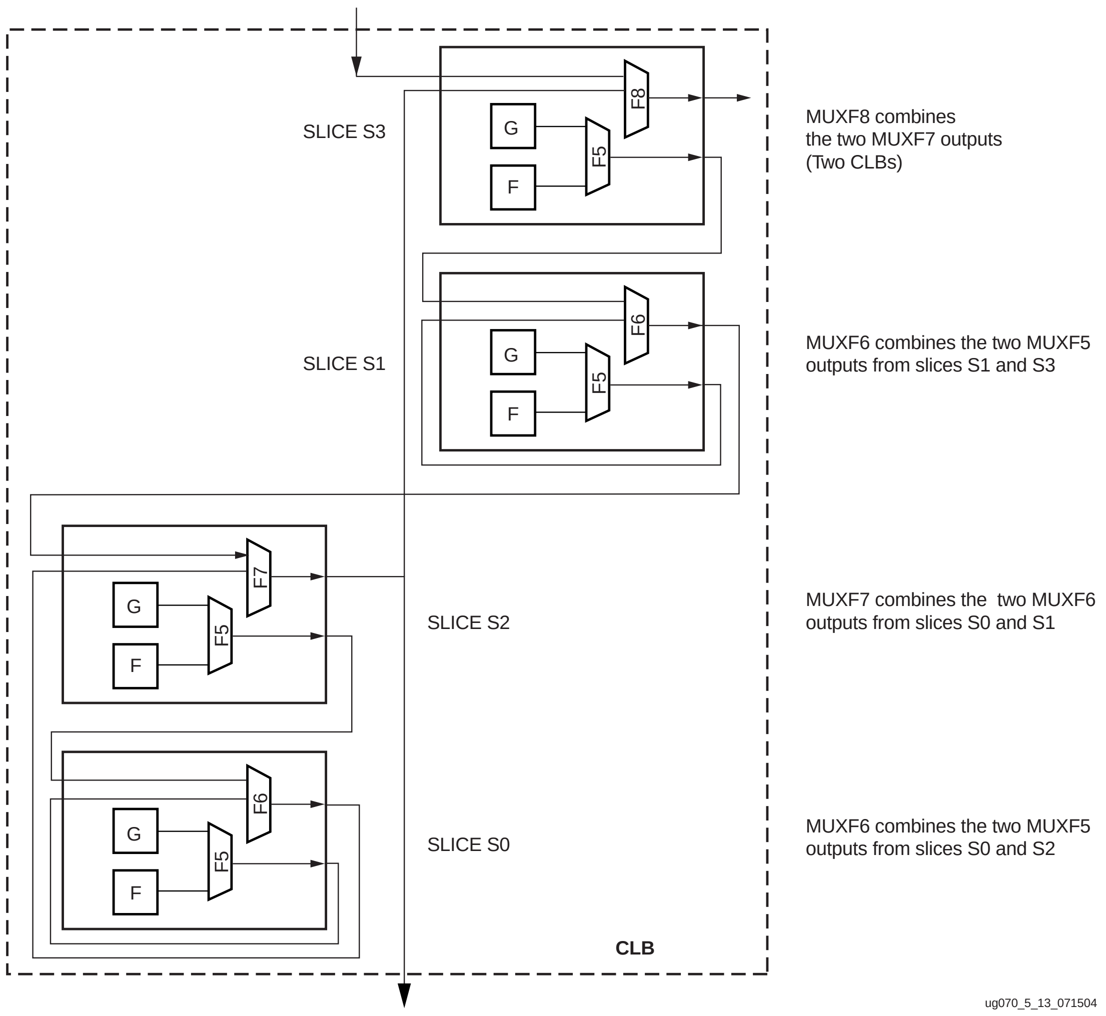
Applications developers can benefit from this feature in order to balance the timing of data pipelines, or to implement synchronous FIFO designs.

##### *2.3.0.6 Multiplexers*

Each Virtex-4 FPGA slice has one MUXF5 and one MUXFX multiplexer, where MUXFX stands for MUXF6, MUXF7, or MUXF8, depending on the slice position. Each CLB element has two MUXF6, one MUXF7 and one MUXF8 multiplexer, as shown in Figure 2.5. These multiplexers are used to provide functions of five, six, seven, or eight inputs, respectively, for each CLB.

Other than that, Virtex-4 FPGA function generators and associated multiplexers can be combined to implement fully combinatorial, one-level logic wide input multiplexers. The following configurations are possible:

- 4:1 multiplexer in one slice
- 8:1 multiplexer in two slices
- 16:1 multiplexer in one CLB element (4 slices)
- 32:1 multiplexer in two CLB elements (8 slices - 2 adjacent CLBs)



SLICE S3
SLICE S1
SLICE S2
SLICE S0
CLB
G
F
F5
F6
F7
F8
MUXF8 combines the two MUXF7 outputs (Two CLBs)
MUXF6 combines the two MUXF5 outputs from slices S1 and S3
MUXF7 combines the two MUXF6 outputs from slices S0 and S1
MUXF6 combines the two MUXF5 outputs from slices S0 and S2
ug070_5_13_071504

Figure 2.5: MUXF5 and MUXFX Multiplexers

### *2.3.0.7 Fast Lookahead Carry Logic*

The Virtex-4 FPGA CLB has two separate carry chains, as shown in Figure 2.2. The use of this dedicated carry logic provides fast addition and subtraction operations. The dedicated carry path can also be used to cascade LUTs and implement wider logic functions.

### *2.3.0.8 Arithmetic Logic*

In order to improve the efficiency of arithmetic operations, one extra XOR gate is included in each slice. This mechanism allows a 2-bit full adder to be implemented within the slice boundary. On top of that, a dedicated AND (FAND or GAND) gate, to improve the efficiency of multiplier implementation, is also included. Both elements can be seen in Figure 2.3.

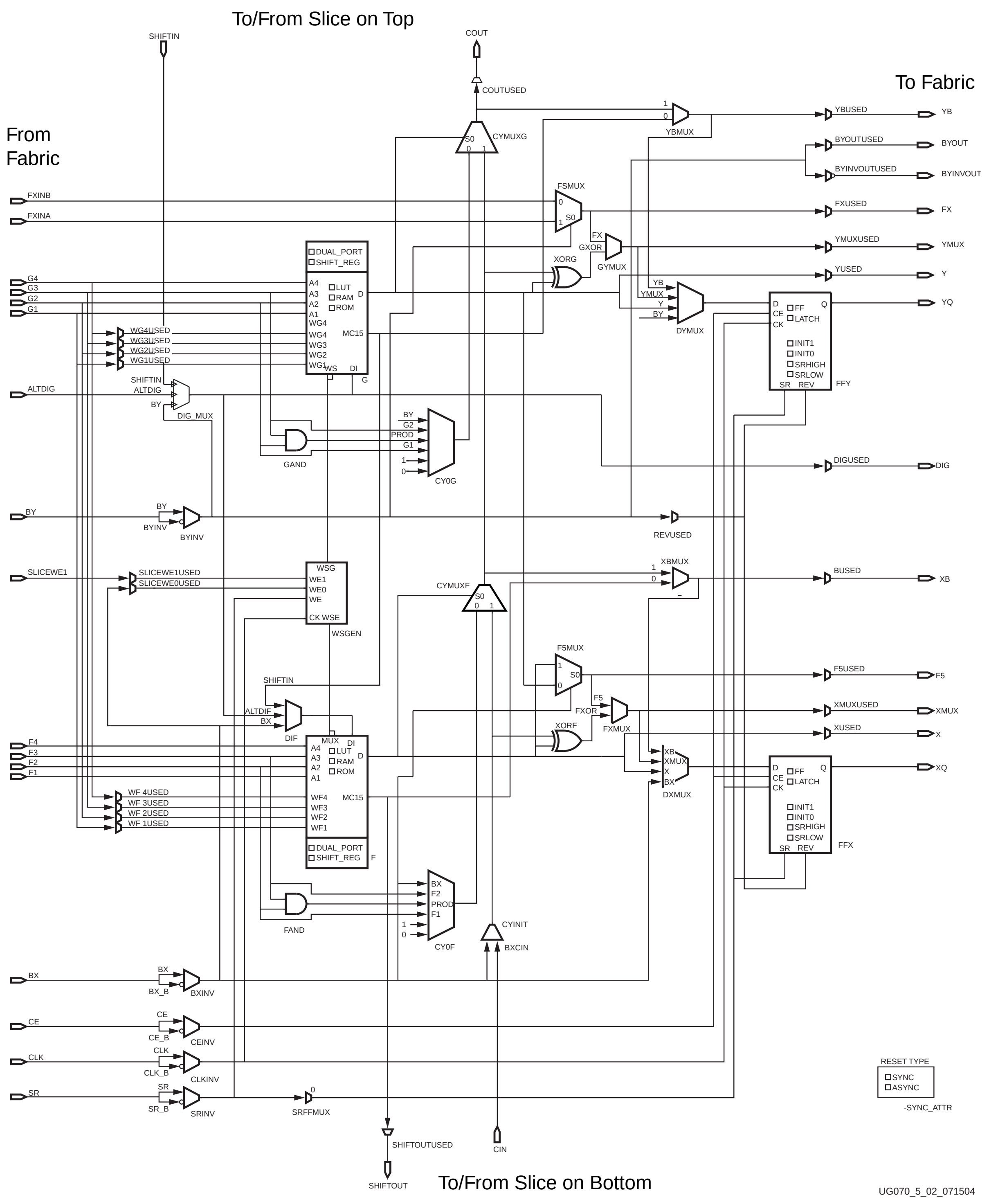


Figure 2.3: Diagram of SLICEM

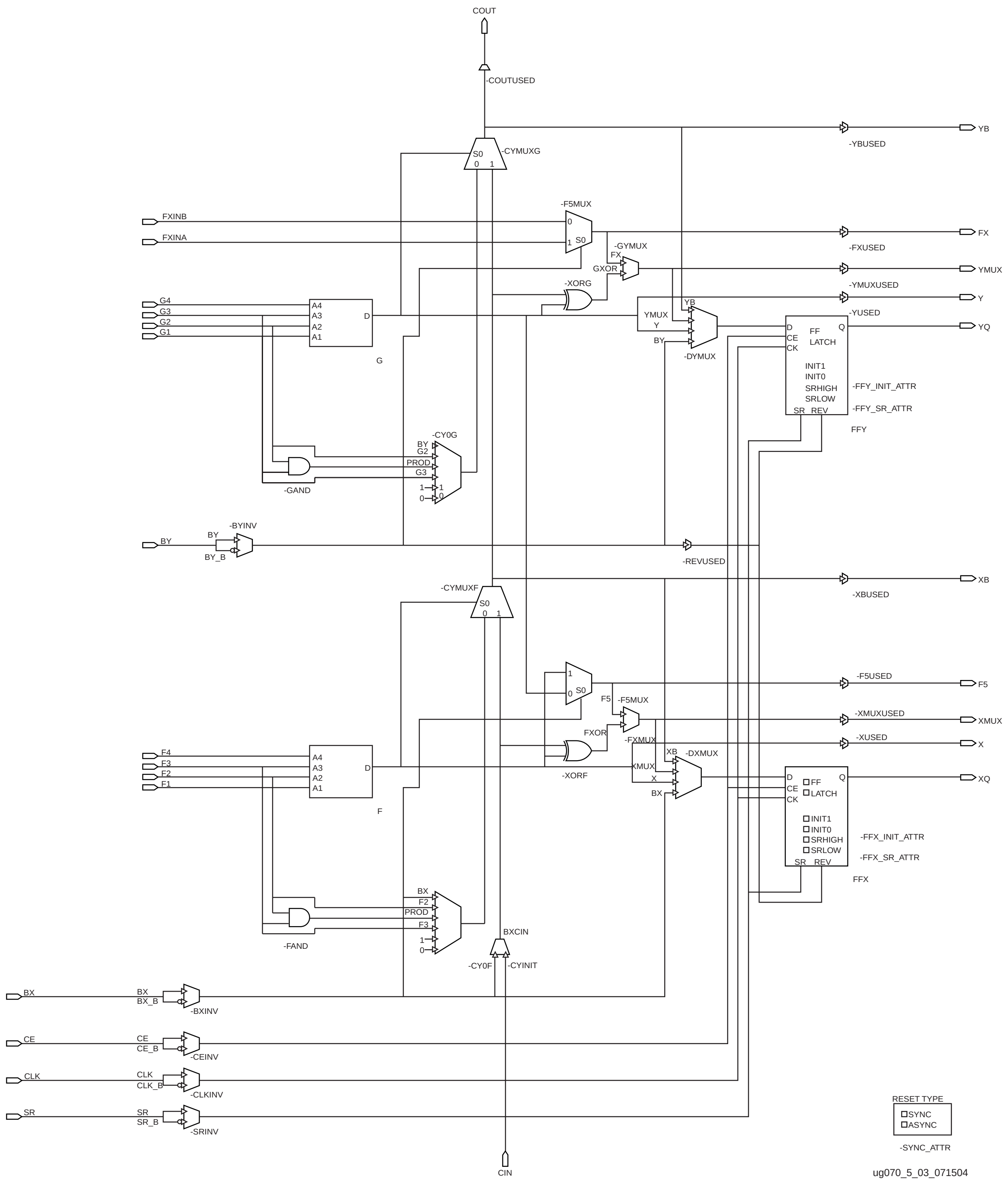


Figure 2.4: Diagram of SLICEL

# 3 CONFIGURATION OVERVIEW

All user programmable aspects inside a Virtex-4 FPGA are stored in SRAM memory cells, which due to their volatile characteristics, must be configured every time the device is powered-up. These memory cells are known as the configuration memory, and they define LUT equations, signal routing, voltage standards and all other aspects necessary to customize the device in order for it to behave as expected. To program this memory space, instructions, for the configuration control logic, and data, for the configuration memory, are loaded from the bitstream through one of the configuration interfaces.

This chapter explores the interfaces, file formats and device connections used during the Virtex-4 configuration process.

## 3.1 Configuration Interfaces

Configuration interface is a logical interface consisting of a subset of the device pins used for loading the configuration data into the configuration memory (Xilinx UG070). Each configuration interface is associated with one or more configuration modes, and each configuration modes is targeted to specific application requirements. These modes are selectable via external device pins, called Mode input pins. Table 3.1 shows the supported modes and summarizes their characteristics.

Regardless of the mode pin settings, the JTAG/Boundary-Scan configuration interface is always available, since it can be used also for debug purposes. However, when selected explicitly, it disables all other interfaces to prevent pin conflicts.

| Configuration | Configuration | Mode Pins | | | CCLK | Data |
|---|---|---|---|---|---|---|
| Interface | Mode | M2 | M1 | M0 | Direction | Width |
| Serial | Master Serial | 0 | 0 | 0 | Out | 1 |
| Serial | Slave Serial | 1 | 1 | 1 | In | 1 |
| Parallel | Master Select-Map | 0 | 1 | 1 | Out | 8 |
| Parallel | Slave Select-Map8 | 1 | 1 | 0 | In | 8 |
| Parallel | Slave Select-Map32 | 0 | 0 | 1 | In | 32 |
| JTAG | Boundary-Scan | 1 | 0 | 1 | N/A(1) | 1 |

Notes: (1) JTAG mode uses the JTAG TCK pin instead of the CCLK.

Table 3.1: Configuration Modes And Interfaces

For serial and parallel interfaces, the configuration clock (CCLK) direction depends on the configuration mode selected. Thus, the terms Master and Slave represent the following convention:

- If the Master Mode is selected, the configuration clock is an output pin driven by an internal Virtex-4 oscillator.
- If the Slave Mode is selected, the configuration clock is an input pin driven by external configuration controllers.

While some of the configuration pins are dedicated to the interface and retain their functionality even after the configuration is completed, others are general-purpose I/O pins and can be reused by user-logic after device initialization. This behavior can be controlled by design constrains set during the implementation phase.

### 3.1.1 Serial Interface

Programming trough the serial interface can be done using master and slave modes. The Slave Serial configuration mode allows for FPGAs to be configured from other logic devices (e.g. microprocessors) or in daisy-chain fashion, while the Master Serial mode makes possible to configure the FPGA from an external serial PROM. Data is loaded at one bit per CCLK cycle in both modes, and the MSB of each data byte is always written to the DIN pin first. Table 3.2 shows the signals involved in Serial Modes.

| Signal | Direction | Description |
|---|---|---|
| CCLK | Input/Output | Configuration clock source. |
| PROGRAM_B | Input/Output | Active-Low asynchronous full-chip reset. |
| INIT_B | Input/Output | Input to delay configuration, output to set CRC error. |
| DONE | Input/Output | Input to delay device start-up. |
| M[2:0] | Input | Mode Pins – determine configuration mode. |
| D_IN | Input | Serial configuration data input. |
| DOUT_BUSY | Output | Data output for downstream daisy-chained devices. |

Table 3.2: Serial Interface Signals Description

### 3.1.2 Parallel Interface

The SelectMAP configuration interface provides an 8-bit or 32-bit bidirectional data bus interface to the Virtex-4 configuration control logic that can be used for both configuration and readback. Readback is an operation that allows the user to read the configuration data back from the device, in case it must be checked for integrity purposes.

This configuration mode is used when speed is an important factor. It's target applications are the same as serial modes, but due to its handshake signals, it involves a slightly more complex protocol. Table 3.3 shows the signals involved in Parallel Interface.

*ICAP*

The Internal Configuration Access Port (ICAP) provides a configuration interface from within the FPGA fabric allowing users to modify the device functionality at run-time. The ICAP is a subset of the SelectMAP interface, implementing some of its signals and a separated data bus for read and write operations. Following the example of SelectMAP, the ICAP can also be configured to use both 8-bits or 32-bits data bus widths. Table 3.4 shows the ICAP interface signals.

There are two ICAP modules in the Virtex-4 devices sharing the same underlying logic: TOP and BOTTOM. The only difference between them is their physical location

| Signal | Direction | Description |
|---|---|---|
| CCLK | Input/Output | Configuration clock source. |
| PROGRAM_B | Input/Output | Active-Low asynchronous full-chip reset. |
| INIT_B | Input/Output | Input to delay configuration, output to set CRC error. |
| DONE | Input/Output | Input to delay device start-up. |
| M[2:0] | Input | Mode Pins – determine configuration mode. |
| Data | Input/Output | Configuration data input and readback data output. |
| CS_B | Input | Active-Low chip select to enable the data bus. |
| RDWR_B | Input | Determines the direction of the SelectMAP data bus. |
| DOUT_BUSY | Output | Handshaking to indicate successful data transfer. |

Table 3.3: Parallel Interface Signals Description

on the chip. Since the same resources are used for both instances, the two interfaces can never be active at the same time. If both sites are required to be present in the user design, the TOP site must be activated before switching to the BOTTOM site.

| Signal | Direction | Description |
|---|---|---|
| CLK | Input/Output | Configuration clock source. |
| WRITE | Input | Determines the direction of the SelectMAP data bus. |
| CE | Input | Active-Low chip select to enable the data bus. |
| BUSY | Output | Handshaking to indicate successful data transfer. |
| I[31:0] | Input | Configuration data input. |
| O[31:0] | Output | Readback data output. |

Table 3.4: ICAP Signals Description

### 3.1.3 JTAG Interface

Boundary-Scan mode is based on an industry standard serial programming interface(IEEE 1149.1). The IEEE 1149.1 Test Access Port and Boundary-Scan Architecture is commonly referred to as JTAG. JTAG is an acronym that stands for Joint Test Action Group, in reference to the the technical committee responsible for its development. This interface provides the capability of testing individual components and its interconnections by sending standard-defined instructions and application-data trough the I/O pins of the interface. Other than testing, JTAG provides the possibility for a device to have its custom set of instructions, such as configure and verify. This flexibility and standardization makes possible to program FPGAs, PLDs, and PROMs through the same pins. For this reason, Boundary-Scan has became the most popular mode of configuration. Table 3.5 shows the signals involved in JTAG interface.

| Signal | Name | Description |
|---|---|---|
| TCK | Input | Test Data Clock |
| TDI | Input | Test Data IN |
| TDO | Output | Test Data OUT |
| TMS | Input | Test Mode Select |

Table 3.5: JTAG Interface Signals Description

## 3.2 Configuration Process

The configuration process for the Virtex-4 device is independent of the configuration interface used. This means that even though each interface has different modes of operation, the device configuration follows the same basic procedure regardless of the way configuration data is being loaded into the device.

The overall process is divided in eight phases, as shown by figure 3.1. Following is a brief explanation of each phase.

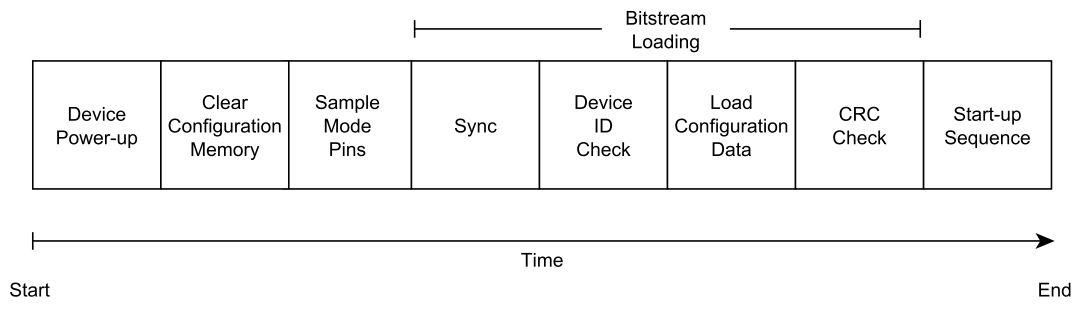


Figure 3.1: Configuration Flow

*Device Power-up*

This is the first stage of the device initialization, and it occurs when power is first applied to the FPGA. The device remains in this stage until power requirements across all voltage banks are met. During power-up all internal state machines are forced to be reset, and PROGRAM and INIT pins are both driven low by the device.

*Clear Configuration Memory*

During the second stage, while PROGRAM is still being driven low, the configuration memory is cleared sequentially. All I/O pins are placed in a high impedance state, except for the dedicated configuration interface and JTAG pins. After the configuration memory is cleared, PROGRAM goes high.

*Sample Mode Pins*

At the third stage, after INIT_B transitions to high, the device samples the mode pins. If the mode pins indicate that a Master Serial or Master SelectMap interface is in use, the FPGA starts to drive the CCLK. At this point, the device also begins to sample the configuration data input pins.

At this time, device configuration can be delayed by externally holding low, either INIT_B (to avoid sampling mode pins), or PROG (to keep clearing the configuration memory).

*Synchronization*

Before the configuration process can proceed, a special 32-bit synchronization word (0xAA995566) must be sent to the device. The purpose of the fourth stage is to align the start of the configuration data together with the internal configuration logic. Any data sent prior to the synchronization is ignored by the device.

*Check Device ID*

In order to prevent potential damages to the device caused by incompatible bitstreams, after device synchronization, a special device ID check is issued to assure the correctness of the configuration data. At the fifth stage, the device ID present in the bitstream is compared to the ID register in the internal configuration logic. Upon a detected mismatch, an ID Error signal is generated and the configuration process is aborted.

*Load Configuration Data Frames*

Sixth stage begins after the internal configuration logic is synchronized, and after the bitstream is validated by means of the ID check. At this point the configuration data is ready to be loaded through one of the configuration interfaces. The bitstream content is then feed to the device sequentially, following the interface protocol.

*Cyclic Redundancy Check*

As the configuration data is loaded, a cyclic redundancy check (CRC) value is calculated from the received data packets. After this process is finished, the bitstream can issue a Check CRC instruction to the device, along with the expected value. If the calculated value does not match the expected, the device drives INIT_B low and aborts configuration. If this happens, the device must be resynchronized and the configuration process has to be restarted.

*Start-up*

After the device receives all the configuration data, the start-up sequence begins. This sequence is controlled by an eight phase sequential state machine responsible for handling the last events in the configuration process, including when the global internal reset signals are toggled and when the DONE pin goes high, indicating the end of the process.

## 3.3 Operational Modes

When the configuration process finishes, and the DONE pin is driven high, the device transitions from configuration to user mode. From this point forward it operates according to the design that was programmed until power-off is applied, or until it is brought back to configuration mode. This can happens if power is cycled, PROGRAM_B is driven low, or the JPROGRAM instruction is sent to the JTAG interface. Every time the device enters configuration mode its internal configuration memory is cleared, and all prior programmed data is lost. Figure 3.1 shows a regular operation cycle.

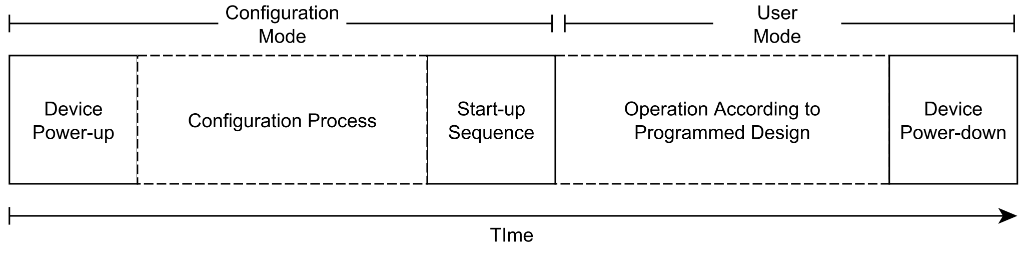


Figure 3.2: Regular FPGA Operation Cycle

## 3.4 Dynamic Partial Reconfiguration

The term Dynamic Partial Reconfiguration describes the process of reprogramming a device while still in user mode. This advanced feature extends the inherent flexibility of SDRAM FPGAs by allowing only specific regions of the device to be reconfigured while the remainder of the design is still operational. This capability allows the designers to reduce the overall cost associated with board space, to update the design remotely and even to reduce power consumption. Other than this, the possibility to dynamically change the hardware functionality of only a small portion of the device offers real-time flexibility to choose the most adequate algorithms and protocols to handle the tasks at hand (Xilinx WP374).

Regarding implementation flows, there are two main styles of partial reconfiguration: Partition-based (Xilinx UG702) and Difference-based (Xilinx XAPP290). Partition-based dynamic partial reconfiguration is used when the reconfigurable portions of the design are complete self-contained partitions. The reconfigurable regions on the device can then be used to time-multiplex modules similarly to the way a microprocessor switches tasks. On the contrary, difference-based DPR is targeted to small reconfigurations, such as LUT equations, user-memory values and I/O standards. This way, on-the-fly updates of hardware-based algorithm parameters can be performed.

Despite of the reconfiguration method used, implementing a reconfigurable design is very much alike to implementing multiple regular designs that share common resources. After all designs have been implemented, and an initial configuration is selected, partial bitstreams can be derived. Partial bitstreams are configuration files that contain only the data associated with the difference between two regular bitstreams. Switching the configuration of a design from one implementation to the other is very fast, as the partial bitstream sizes tend to be significantly smaller. Thus, the reconfiguration process can be performed in fractions of the time required for a full configuration cycle.

Designs modifications can be performed at two different levels. The first one, at the front-end; and the second, at the back-end. For front-end changes, the design must be modified in the HDL or schematic file. This process requires the design to be re-synthesized and re-implemented in order to generate a new placed and routed netlist. For back-end changes, the original placed and routed netlist is directly modified. This process, however, requires a much more in-depth knowledge of the device internal architecture and toolchain. While there are a vast amount of possible changes to be made in the back-end, it is only recommended to modify I/O standards, BRAM contents and LUT functions. Other modifications, such as routing, may induce resource contention during the reconfiguration process, which can produce a system failure.

In comparison to the back-end, front-end changes are usually much more intuitive, since they take place at a more familiar and comfortable level. On the other hand, optimizations in the synthesis, mapping and routing process can lead to changes in the design that were not initially considered. Although functionally speaking the same result can be achieved regardless of the technique employed, there is a trade-off between abstraction level versus control over the affected resources. Usually, back-end changes are associated with difference-based reconfiguration, while front-end changes are associated with partition-based.

## 3.5 Managing Dynamic Partial Reconfiguration

Partial bitstreams already contain all the information required by the FPGA internal configuration logic. Thus, managing dynamic partial reconfiguration does not require any knowledge of the affected resources, nor any pre-processing of the configuration file before it is loaded in the device. Generally speaking, managing dynamic partial reconfiguration means retrieving the partial bitstream from a external repository and transferring it to the device via one of the configuration interfaces.

Loading a partial bitstream can be done in the same way as a regular one, although not all configuration modes are available. Specifically, only Slave SelectMAP, Slave Serial, JTAG and ICAP can be used for this purpose. Moreover, if Slave SelectMAP needs to be used, special constraints are required during the design implementation phase in order to prevent non-dedicated interface pins to be released after initial device configuration. Another difference is that the partial bitstream contains, essentially, only configuration data, so it does not have the commands to trigger the device start-up sequence. This means, among other things, that after all the information in a partial bitstream file has been transferred to the FPGA, the DONE pin is not asserted. In this case, the configuration controller must monitor the data being sent in order to know when the operation has finished.

Usually, INIT and PROG pins are driven low to start a full device configuration. However, before uploading a partial configuration file this must not be done, as it would clear all the FPGA internal memory. Any indication required by the active design before the partial reconfiguration starts, such as toggling enable signals and disabling clock regions, must be performed by its own logic. Generally, as soon as file transfer is completed, the reconfigured region can be released for active use.

Although a wide variety of methods can be used to implement the configuration controller, the two techniques in Figure 3.3 are most commonly used.

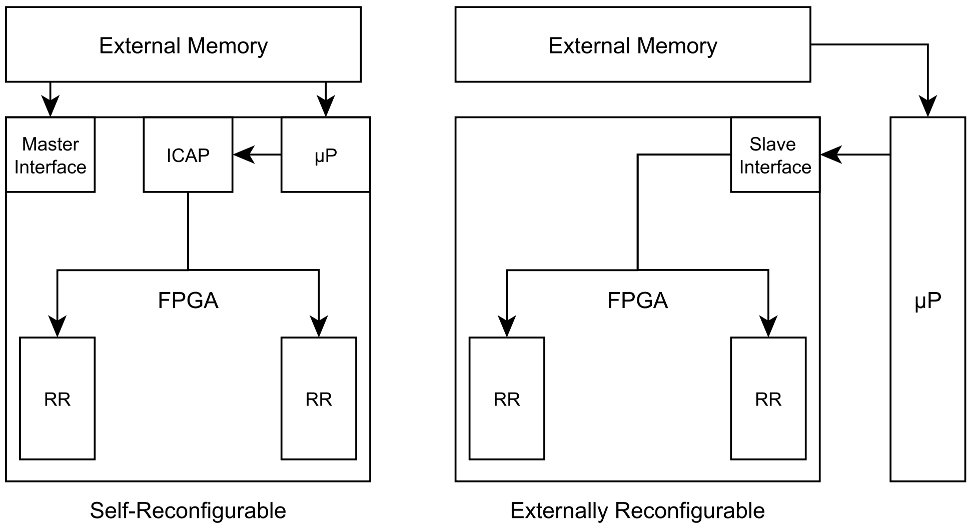


Figure 3.3: Methods of Delivering a Partial Bit File

The first technique, shown at the left of Figure 3.3, uses a self-reconfigurable approach. The FPGA-embedded microprocessor is responsible for loading the partial bitstream files from an external repository and sending them to the ICAP interface. This solution benefits itself from the internal configuration port to implement both the reconfiguration controller and reconfigurable design in the same device. This leads to a compact and cost efficient design. On the other hand, due to the static requirements of the configuration controller, fewer resources are available to implement the reconfigurable module. Moreover, initial device configuration must be accomplished by other means, such as by using a master mode interface connected to a non-volatile memory.

On the right side of Figure 3.3 the second technique is presented. This solution is very similar to the first, but instead of implementing the configuration controller internally to the device, everything is moved off-chip. Here, an external microprocessor is connected to an outside configuration port. The advantage of this approach lies on the fact that the whole device can be used to implement the reconfigurable design. Other than this, the same controller that performs the reconfiguration can be used for the initial bitstream uploading. The drawback of this solution is the associated cost of one extra system module.

## 3.6 Configuration Data Files

In order to provide system designers with more flexibility, Xilinx tools can generate different formats of bitstreams. Each one is formatted in a different way, so system constraints must be analyzed to determine which format is more appropriated. Table 3.6 shows the different formats of the configuration files.

| File Extension | Description |
|---|---|
| .bit | Binary file containing header information that should not be downloaded to the FPGA. |
| .rbt | ASCII file containing a text header and ASCII 1s and 0s. |
| .bin | A binary file containing no header information. |
| .mcs, .exo, .tek | ASCII PROM formats containing address as well as checksum information. |
| .hex | ASCII PROM format only containing data. |

Table 3.6: Xilinx Configuration File Formats

# 4 RELATED WORK

Several fault injection platforms based on dynamic partial reconfiguration have been proposed in the past. This section presents an overview of the usually employed methods and some of the latest publications on the subject.

## 4.1 General Board Fault Injection Platforms

General board platforms are generic fault injection platforms that can be implemented on any commercial FPGA development board, provided that it contains the same resources as the one in which the platform was initially developed in. That is not particularly difficult, since most vendors tend to offer similar solutions with equivalent embedded memory size, communication peripherals and configuration options. In other words, this platforms do not require a unique proprietary board design manufactured only for the fault injection system.

### 4.1.1 Reconfiguration Trough XHWIF Using JBITS

The JBits software (S. Guccione et al 1999) is a set of Java classes which provide an Application Programming Interface to access the Xilinx FPGA bitstream. The interface operates on either bitstreams generated by Xilinx design tools, or on bitstreams retrieved from the actual devices by readback operations. This allows all configurable resources like Look-up tables, routing and the flip-flops in the FPGA to be individually configured and modified under software control in real time operations.

The work proposed by (P. Kenterlis et all 2006 ) is based on a mixed hardware/software platform for fault injection in both sequential and combinational circuits using the JBits API. The platform is composed by a workstation and a single FPGA board. The FPGA board used was a XESS XSV800 equipped with a Virtex device (XCV800). To connect the board with the workstation, and special USB controller was developed in an Atmel AVR chip. This controller provides a USB connection to the host and a modified 8bit bidirectional port connecting to the FPGA's logic XHWIF port, a interface for programming Xilinx FPGAs from Java code. The FPGA device is responsible for hosting the XHWIF port, the circuit under test, the test pattern generator, possible embedded memory and a comparator. Implementations details on the placement and implementation flow were not provided.

The software running on a workstation is primarily responsible for analyzing the devices bitstream and retrieve a list of LUTs input usage. In synthesized circuits, it is possible that some of the LUT inputs are not used for logic implementation, and faults affecting those inputs would not manifest themselves in the LUT's output. Although in a real scenario faults can affect those inputs, in the paper the authors choose to conduct their experiment only on used LUT bits. After this, a list of the fault location is compiled, and bitstream modification can be performed in order to flip bits in the CUT's frame space. All software routines make use of the JBITS API to access bitstream information.

Fault injection is performed by dynamically partial reconfiguring the device. Results are retrieved after each fault injection to perform fault classification based on location and effect. Experimental results presented by the authors show that the platform obtained a speed gain of 221.14x over estimated simulation-based fault injection time. Although this theoretical speed-up has been achieved, the mechanism of fault classification may constitute a bottleneck, since after each fault injection data must be transmitted back to the workstation.

### 4.1.2 Reconfiguration Trough ICAP

The fault-injection platform developed by (L. Sterpone et al 2007 ) is composed of two modules: a host computer and a general FPGA board equipped with a Virtex-II Pro device. To connect the FPGA device and the host computer, a serial communication cable over a RS-232 line is used.

The host computer is used for configuring the Virtex-II Pro and for generating the fault location list. During the execution of the fault injection experiment, it's only purpose is to provide a user interface to the fault-injection experiments and to collect the results in terms of fault effect classification.

The architecture of the proposed fault-injection system is completely implemented on the FPGA device. Four components are mapped on FPGA and interconnected by an On-chip Peripheral Bus (OPB). They are:

- Timing Unit: Drives the CUT clock and reset. The clock of the CUT has the same frequency of the FPGA device.

- Circuit Under Test (CUT): The circuit under test. Both it's input and output pins are connected to the OPB Bus, while the reset and clock signals are connected to the Timing Unit.

- ICAP: The reconfiguration interface.

- PowerPC microprocessor: Hardcore microprocessor. Executes the partial reconfiguration for fault injection and reports the fault injection results to the host computer trough the serial line.

The fault impact analysis is performed in three steps. First, in the pre-running phase, the test patterns that will be applied to the CUT are loaded within the PowerPC memory. If the CUT has its own memory (i.e. a microprocessor), it is initialized as well. Secondly, the CUT is executed in order to record the correct output vectors. At the end of this execution, the total number of clock cycles need by the CUT and the correct outputs are stored within a memory block connected to the PowerPC trough the PLB bus. Finally, the fault is injected in the CUT.

The fault injection process is performed by reading back the configuration memory frames corresponding to the fault location, flipping a random value and writing back the modified frame in the ICAP interface. Then, after the correct amount of clock cycles, the output of modified CUT is compared to the initial run corresponding to the correct outputs. At last, the PowerPC updates a fault classification list with the results obtained by the fault-injection and classifies each injected fault according to it's effect. Faults can be classified as silent, if the output produced by the DUT are equal to the golden outputs; wrong answer, if a mismatch was detected; or time-out in the case the DUT IP core is blocked. Experimental data presented by the authors show that the average reconfiguration time and fault analysis is 6ms/fault.

Similar work was proposed by (L. Kafka 2008) using a Microblaze softcore as the microprocessor unit. Other than the fault injection platform, the author also presented improvements on the original ICAP device drivers in order to speed-up the reconfiguration process. Results shown indicate a speed gain of eight times over the Xilinx available solution.

Although the ICAP provides a high-speed interface for device reconfiguration, it's use implies in a restricted placement for the circuit under test. Since the ICAP physical resources are in the bottom right corner of the Virtex-II Pro Devices, this column space cannot be used for hosting the reconfigurable modules (i.e, the CUT). Since in each side border of this device there is an IOB Column, the use of ICAP prevents the CUT to be placed in configuration range of one of this columns, and prevents one entire border of I/O pins of being used for inter-chip communication. Other than that, since the same device is used to implement the entire fault injection platform, the overall left area for CUT implementation is significantly reduced.

### 4.1.3 Reconfiguration Trough Boundary Scan

The work proposed by (N. Battezzati et all 2008) is fault injection platform formed by a software application, running on a PC; and a general FPGA board, hosting the Device Under Test. The hardware is not designed to support the test, as many platforms suggest; instead, the same device to be used during the mission is used to perform the fault injection. This characteristic provides the possibility to evaluate faults affecting the exactly same circuit routing to be used during mission time.

The software platform, in order to perform the injection campaign, uses a list of faults, a set of input vectors and the information describing the boundary scan architecture that provides access to the DUT. The communication is handled through the JTAG protocol, either by the PC's parallel port or the USB interface.

The fault injection process is based on dynamic partial reconfiguration using the difference based flow. Faults are injected reconfiguring the device with partial bitstreams that differ from the original by only one bit. Partial bitstreams are generated on the fly based on the original bitstream and a list of fault locations. After a fault is injected, the software platform uploads the entire list of input vectors through the JTAG interface and reads back the resulting state of the circuit for comparison with expected values. If any difference is found, the fault is labeled as detected and the application is stopped, in order to save time. A report file is updated with the details about the fault location and received outputs. Otherwise, if no differences are found, the fault is classified as not detected. The device is configured again to remove the bit flip introduced by the last fault injection, and the procedure restarts for another fault location until the list is over.

This platform has some important characteristics that differs itself from the others. First, due to the use of boundary scan architecture, the implementation does not depend on the device family, model or package. Secondly, it is completely non intrusive, and does not require any modification in the circuit to be tested. Finally, by being a purely software solution, it does not require additional hardware to perform the fault injection.

However, the low cost and the non intrusiveness come at a cost. The boundary scan approach implies in a serial configuration interface that leads to reduced performance during reconfiguration. Other that that, the great amount of data exchanged between the PC and the DUT create a bottleneck for speed-up. For this reason, long run times are expected for this platform.

The experiments used a general board with a Virtex 4(XC2VP4) device for the DUT and a Xilinx Parallel Cable III at 200 kHz to connect it to the PC. Accordingly to the tests performed, in the best case scenario, the fault injection time was 268ms/fault, while the worst case produced a 862ms/fault time. The variation is produced by the number of I/O pins used in the DUT, which represent more shifts through the JTAG registers. While in the best case there were only 17 I/O pins used, in the worst there were 99 I/O pins.

### 4.1.4 Bitstream Corruption for Configuration Control Mechanism Evaluation

So far, the works proposed in the literature were only interested in faults affecting the circuit implemented in the FPGA device, but none was concerned with faults affecting the device's internal mechanisms. The work presented by (M. Alderighi et al 2003) introduces a different approach for evaluating the effect of faults in Xilinx Virtex FPGAs. The work focuses on the structure that performs the device configuration, called configuration control mechanism, and targets designs that performs reconfigurations for fault-mitigation or adaptive routines as part of a normal operation cycle. The more often a design reconfigures itself, the more likely a fault affecting the configuration mechanism will manifest itself.The configuration control block is comprised of SRAM memory cells and is prone to faults the same way as the configuration memory.

Based on that, the paper presents a tool for injection faults in the configuration control mechanism of Virtex devices. A specific device (XQV100) is used, but the methodology can be generally applied to other devices of the Virtex and Virtex II families. Fault Injection is performed by modifying the bitstream while it is loaded into the device. The instructions for the configuration control mechanism are corrupted and fed to the device. Results show that faults affecting the configuration control block can provoke complete device failure caused by erroneous bitstream loading. Although there was no fault scenario in which the device was able to complete the configuration process, error signals were always set by the device. Details regarding implementation processes were not provided.

## 4.2 Custom Board Fault Injection Platforms

Custom board platforms are dedicated fault injection platforms designed specially for one system architecture. They comprise software applications running on high-end workstations and custom-made FPGA boards. Multiple FPGA devices are used in order to isolate the configuration controller from the circuit under test.

The main examples of such systems found in the literature are the FLIPPER and the FT-UNSHADES-C Platforms.

### 4.2.1 The FLIPPER Platform

The FLIPPER Platform (M. Alderighi et al 2007) is a tool for fault injection targeting the FPGA's internal memory. It's development was funded by the European Space Agency with the purpose of evaluating the single-bit and multiple-bit upset effects in Xilinx SRAM-based FPGAs. FLIPPER is used to evaluate fault sensitivity by collecting a probability distribution of the number of randomly injected faults necessary to cause a functional error. The platform is comprised of three main parts: a control board, a device under test and a workstation computer.

The control board manages the overall fault injection procedure by means of a Virtex-II Pro (XC2VP20) device. This board also contains 128Mbytes of SDRAM and 16Mbytes of Flash memory, and communicates to the workstation via a USB 2.0 port controlled by a dedicated microcontroller. Furthermore, it also has two 240 pin connectors plus one 60 pin connector for the test data and control communication with the Device Under Test.

The Device Under Test board contains a Virtex II (XQR2V6000) device for fault injection tests. The entire device is intended for the DUT implementation, so no modifications are required. Even so, designs must constrain to the device's pin-out, so proper communication can be established with the control board. The DUT board is connected to the control board trough a piggy-back style connector, and share with it up to 416 signals. Additionally, a temperature sensor is included on the board to monitor the device's temperature during the experiments.

The workstation is responsible for hosting a software application that was specifically developed for the FLIPPER system. Trough this application it is possible to set up the experiment options such as I) the target of injection, II) the test mode (internal memory bit may be randomly addressed and faults may accumulate until a functional failure occurs, or the bits may be addressed sequentially, one at a time), III) the fault type (either single bit flip or multiple bit flip), IV) the DUT clock rate, and V) the address range of memory bits that are involved in the current experiment. Other than that, it is possible to import input and output vectors from a ModelSim simulation at every clock edge. These vectors are used as reference values and test stimuli during the fault injection process, and are 150bit and 120bit wide respectively. This data is stored on the control board memory prior to the fault injection session.

Fault injection is achieved using frame modification and dynamic partial reconfiguration from the control board over the DUT board trough an unspecified configuration interface. The DUT is partially reconfigured to inject single or multiple bit flips, and then exercised with the whole set of test vectors to verify the functional influence of such a fault on the device. Outputs from the DUT are compared with golden values stores in the control board's memory. When a mismatch is detected, a fault packet including all the information relevant to the system behavior is sent to the workstation. Once the test is started it proceeds until the stop condition, configurable by software, is met.

For the fault injection campaign, the DUT device is initially configured and, after checking the configuration signals (INIT, BUSY and DONE), the whole set of stimuli is applied for verifying the experimental set up and the correct design behavior. FLIPPER then injects a fault by dynamic partial reconfiguration into a random configuration memory location and applies the stimuli. This procedure iterates, accumulating bit flips in the configuration memory, until one or more output signals deviate from the reference ones. The fault is logged to the workstation.

Experiments using the FLIPPER platform took a execution time of 17 hours over two XTMR design variants and the plain version of a particular design. During this time there were injected 3,3 million faults, corresponding to a injection time of 18ms/fault. No operation frequency was provided for comparison.

### 4.2.2 The FT-UNSHADES-C Platform

The FT-UNSHADES-C (J. Tombs et al 2004) (Fault Tolerant - UNiversity of Sevilla HArdware DEbugging System) platform is a mixed hardware/software platform that focuses in producing a functional design test over the circuit's Flip-Flops. The main objective is to perform a study about the circuit robustness targeting latter ASIC fabrication. The system generates a fault dictionary, where every pair {fault,instant} is classified to obtain the following information: sensitive FF, time of fault injection, outputs modified and time of output discrepancy. According to the FT-UNSHADES-C, faults can be classified as damage, if it produces abnormal behavior on the CUT outputs; and latent, if no discrepancy can be found. The platform is comprised of three main parts: a control device (C-FPGA), a core emulation device (S-FPGA) and a workstation computer.

The S-FPGA device is a Virtex FPGA (XC2V8000) used to hold two versions of the CUT. The first is dedicated to produce the right outputs (GOLD), while the second is used for fault emulation (FAULTY). Fault injection is achieved using dynamic partial reconfiguration over the Faulty module FFs. By definition, when a partial reconfigurable module is reconfigured, it's internal storage elements are not modified, so in order to modify bits in the a CLB Flip-Flop, an additional scheme is required. One important issue inside this scheme is time. Time is controlled in terms of clock cycles applied to both, faulty and gold circuits, which is represented by a counter. In the same way, time is the way to address input vectors stored in the SRAM memory banks. When the fault injection time is achieved or a fault is detected, the circuit has to be frozen in order to perform the necessary internal manipulations in the configuration memory of the S-FPGA. In other words, clock has to be carefully stopped at a precise cycle and continued when the accesses are completed to assure the clock counter integrity. To handle this task, the two circuits are wrapped in what is called a test shell.

A test shell is a set of hardware resources used to control the fault injection procedure and fault classification. It is comprised of three blocks: the time counter, the clock handler and the vector addressing.

- The Time Counter block maintains a log of the relative clock cycles that drive the Golden and Faulty circuits.

- The Clock Handler is responsible for stopping the circuits before the fault injection, relaunching the circuits after the fault was injected, producing the necessary signals to indicate fault detection by comparing both circuits outputs, and, finally, handling the debug signals for single-stepping analysis.

- The Vector Addressing is responsible for generating an address derived from the Time Counter that points to the corresponding input vector stored in the on-board memories.

The test shell uses about 300 system gates, and doesn't introduce any delay penalty over the system behavior, since it only operates over the golden and faulty circuits clock.

To perform the fault injection process and control the aspects of the test shell, a control device is used. The C-FPGA is a Xilinx Spartan II-50 device connected to the S-FPGA through a SelectMap link, and to the workstation computer through a parallel or USB port. The C-FPGA is also responsible for gathering the data from the S-FPGA and report them back to the workstation.

All parameters regarding the fault injection process are defined by the software running in the workstation. The application was developed to generate the test vectors and download them to the C-FPGA memory along with a list of the FFs where faults need to be injected. Also, the application provide analysis tools to elaborate single-stepping analysis and fault classification based on the results gathered by the C-FPGA.

Experimental results show that a single fault injection takes approximately 41,6ms at a frequency of 50MHz. For designs with a small number of flip-flops, the FT-UNSHADES-C Platform is not valid. Accordingly to the authors, only designs with more that 500 FFs and 200K+ test vectors can beneficiate from this platform.

# 5 PROPOSED WORK

The proposed fault injection platform aims at low cost and modularity, which allows it to be implemented on most of the commercially available FPGA development kits. This feature opens the possibility for future design updates without the necessity of re-engineering and manufacturing custom boards. Other than this, the proposed architecture is focused on non intrusive techniques. Thus, no reallocation of the circuit under test is required during the fault injection campaign. This way, tests can be conducted on the original system implementation, and the fault impact analysis can be performed over the same device that will be deployed. In order to assure compatibility with most target boards, faults are injected using an industry standard interface.

## 5.1 Proposed Architecture

The proposed platform is divided in four different modules: desktop PC, configuration controller, data acquisition module and device under test. The top-level architecture is shown in figure 5.1.

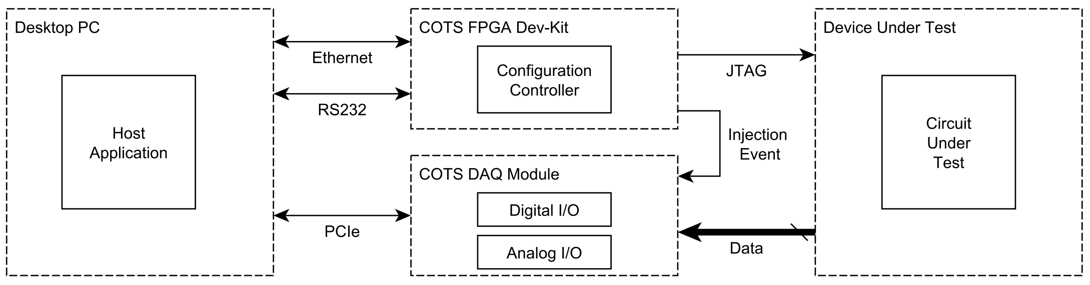


Figure 5.1: Top-Level Architecture for Proposed Platform

### 5.1.1 Desktop PC

The desktop computer is responsible for hosting the applications used to operate the fault injection platform. The whole set is composed of four different software items.

The first application, called fault manager, is responsible for analyzing the netlist of the circuit under test and determining to which resources of the DUT it is mapped to. After this, the fault manager creates a set of partial bitstreams that reconfigure only the resources occupied by the CUT. These partial bitstreams are created using the difference-based DPR flow and contain, each, modifications targeting only one CLB at a time. Thus, each partial bitstream can be considered a different fault to be injected. It is important, though, that for each generated configuration file, another one is created with the intention of reverting the changes caused by the first. On the contrary, after n injections, the circuit under test would have n different faults accumulated. Although sometimes it can be interesting to evaluate the effect of multiple faults accumulating over time, that's not always the case. Other than the bitstreams, the fault manager also creates a scheduler for the configuration manager. This binary file determines the order in which the faults will be injected, and for how much time each fault will be active.

In order to evaluate the effect of the injected faults, another application, called data processing interface, is required. This application is responsible for acquiring all information transmitted from the data acquisition module and comparing it in real-time to pre-compiled results obtained from simulation. Faults and their impacts on the system can be correlated using the configuration scheduler generated by the fault manager.

The other two applications do not require any development effort, since they are utilities commonly found under free/open source licenses. However, they are still important for the platform management. One of them is a FTP server, and the other, a serial port terminal. The FTP server provides the configuration controller all the files required during the fault injection process. The serial terminal, on the other hand, receives and transmits data from/to the configuration controller. This data is used to monitor the status of the current fault injection campaign and control the overall process, allowing it to be started, paused, resumed and aborted.

### 5.1.2 Data Acquisition Module

The data acquisition module is the platform component responsible for acquiring all relevant output signals from the device under test and transmitting them to the desktop computer. Usually, only the computation result is analyzed, but it is also possible to monitor other variables such as voltage levels, current consumption and core temperature. There are several commercially available solutions that provide variable number of input pins, sampling rates, measurement resolutions and bus connections. According to the monitoring requirements of the fault injection study, different devices can be employed.

For example, suppose the platforms requires 8-bit resolution DAQ sampling at 4 MS/s on 32 different channels. The required bus bandwidth between the desktop computer and the DAQ would be of:

$$32 \times \frac{1byte}{Sample} \times \frac{1MSamples}{second} = \frac{32MB}{second}$$

So, in order to support this data rate, a bus of at least 128MB should be used. Table 5.1 shows the maximum theoretical data streaming rates of common data acquisition buses. Values are according to the following specifications: PCI, PCI Express 1.0, USB 2.0, Gigabit Ethernet and Wi-Fi 802.11g.

| Bus | Streaming Rates |
|---|---|
| PCI | 132 MB/s (shared) |
| PCI Express | 250 MB/s (per lane) |
| USB | 60 MB/s |
| Ethernet | 125 MB/s (shared) |
| Wi-Fi | 6.75 MB/s (per 802.11g channel) |

Table 5.1: Streaming Rates of Common Data Acquisition Buses

It is important to note that the actual bus bandwidth is always lower than the theoretical limit. Observed values vary according to the number of devices sharing the same bus and host system performance. For this reason, in order to provide the maximum data rate between the DAQ and the desktop computer, one or more PCI-Express data acquisition cards are suggested.

### 5.1.3 Configuration Controller

The configuration controller is responsible for managing the dynamic partial reconfiguration of the device under test. It is composed, essentially, by an application running on a FPGA embedded microprocessor.

In order to host the configuration controller, a FPGA board must meet a few requirements. It should have, at least: an Ethernet PHY transceiver and a RS232 UART port, to communicate with the desktop computer; flash memory, to store the FPGA configuration file and the embedded application; DDR memory, to store all partial bitstreams and host the application; and available I/O pins, to connect the DUT; The block diagram for the minimal configuration controller board configuration is shown in figure 5.2.

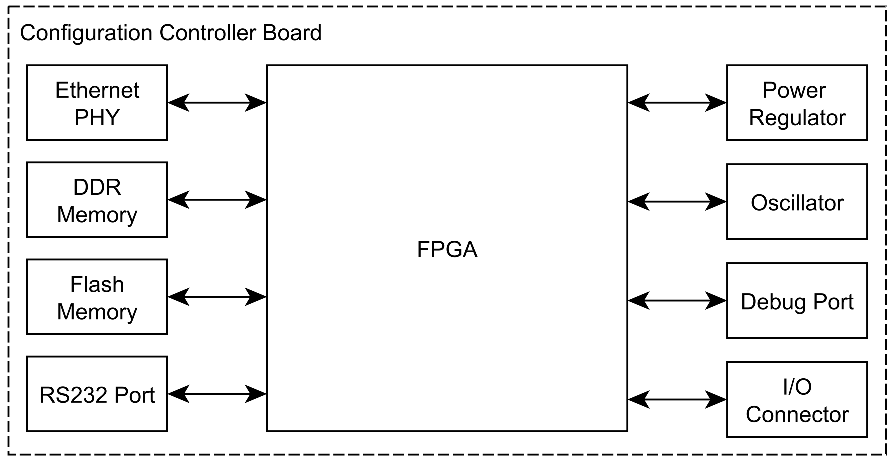


Figure 5.2: Minimal Configuration of Configuration Controller Board

Upon power-up, the FPGA reads from the flash memory it's associated bitstream using a master mode configuration interface. This configuration file contains an small bootstrap allocated to the microprocessor local memory. After device start-up, it begins to load the embedded application from the user section of the flash memory into RAM.

Once loaded, the configuration manager application attempts to connect to the desktop computer through the Ethernet interface. If the connection is established, it requests from the FTP server all files required during the active fault injection campaign. This includes the full bitstream to be initially configured at the DUT, the configuration scheduler and the set of partial bitstreams generated by the fault manager. All data received is allocated in RAM for latter use.

After receiving all requested files, the configuration manager application configures the DUT with it's initial bitstream and sleeps until start command is received through the serial port. This stand-by period can be used to perform a basic sanity check on the DUT system board in order to verify cable connections and power lines. When commanded, the configuration manager starts to upload the partial bitstreams to the DUT, following the configuration scheduler definitions until the cycle has been completed.

In order to synchronize each fault injection with the data sampled by the data acquisition module, a event signal is asserted by the configuration controller each time a partial bitstream is uploaded. This signal is captured by the DAQ in the same clock as the output signals from the DUT, so the data processing interface can correlate each fault and it's impact.

The configuration manager is connected with the DUT by a JTAG bus, as shown in figure 5.3. The use of a industry standard interface assures compatibility with most system boards, eliminating the need for custom layouts to expose less used configuration interfaces. Moreover, it allows the configuration manager to reconfigure multiple devices connected to the boundary-scan using the same connection.

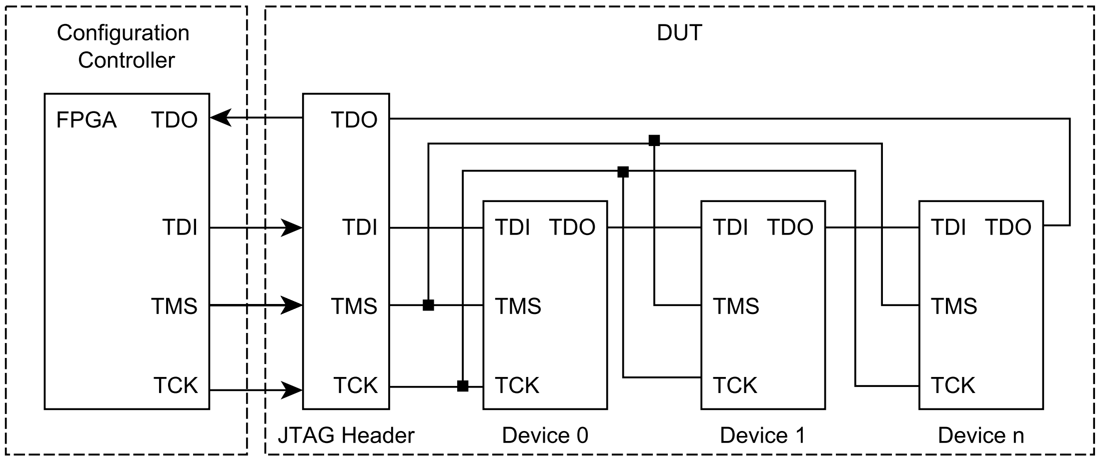


Figure 5.3: Connection Between Configuration Manager and DUT

Figure 5.4 shows the configuration controller FPGA block diagram.

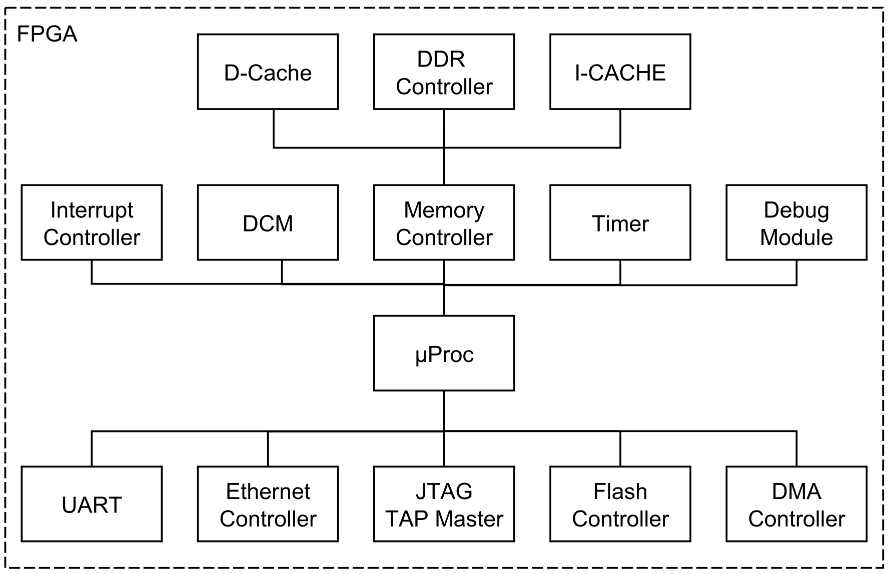


Figure 5.4: Configuration Controller Block Diagram

### 5.1.4 Device Under Test

One key aspect of the proposed fault injection platform is its non intrusiveness aspect. This is accomplished by three different premises:

- No modifications are required in the CUT.
- No extra logic is added around the CUT.
- Device under test hosting the CUT is reconfigured using an industry standard interface.

Since no extra logic is added around the CUT, even large designs occupying the entire FPGA logic are assured to be tested in their original target devices. Moreover, since no modifications are required in the CUT, the fault coverage analysis can be conducted on the same FPGA resources mapped to the original design. Finally, by using industry standard interface for reconfiguration, faults can be injected on the same board used in the deployed system.

The benefit of using the same board of the deployed system for performing the fault injection campaign is reflected on several aspects of the design validation process. First aspect is the ease of operation. Once no additional modification is required at any level of the system design, the platform is plug-n-play. The second aspect concerns the hardware used for testing. Since the actual system board is used for fault injection, stimuli can be performed through the same communication interfaces used during the device mission. This feature enables the analysis of faults targeting the device I/O pins. This is particularly good to understand faults affecting the electric characteristics of the different interfaces connected to the DUT. Third aspect is related to results reliability. Evaluating fault mitigation techniques over the original system design provides results prone to less error than the ones obtained with extra instrumentation logic.

# 6 IMPLEMENTATION

In order to evaluate the feasibility of the proposed fault injection platform, a prototype of the configuration controller was developed. The implementation was based on the architecture presented in chapter 5 and comprises both the FPGA design and the embedded application. The resulting work is a fully operational module capable of performing fault injections on any DUT connected over the JTAG bus. The following sections describe the development process and the implementation details.

## 6.1 Development Board

The development board used for implementing the configuration controller of the proposed fault injection platform should fulfill the requirements presented in figure 5.2. Based on these constraints, and the available options at the microelectronics group in UFRGS, the chosen board was the Digilent Atlys.

The Atlys circuit board is a complete, ready-to-use digital circuit development platform based on a Xilinx Spartan-6 LX45 FPGA, speed grade -3 (502-178). The large FPGA device and variety of on-board peripherals make this an ideal host for a wide range of digital systems. Not only that, but this development platform is fully compatible with all Xilinx CAD tools, including ISE and EDK.

*Features*

Although there is a great amount of features on the board, the ones relevant to this work are:

- Xilinx Spartan-6 LX45 FPGA
- 128 MByte DDR2 with 16-bit wide data
- USB-UART
- 10/100/1000 Ethernet PHY
- 116M Byte x4 SPI Flash for configuration and data storage
- 100MHz CMOS oscillator
- JTAG debug port
- 48 I/O’s routed to expansion connectors

*Board Configuration*

The Atlys board can be configured through three different interfaces: JTAG, Serial and SPI.

An on-board jumper, JP11, selects between JTAG/Serial and SPI. If JP11 is not placed, the FPGA will automatically load a bitstream from the flash memory at power-up. If JP11 is placed, the FPGA will remain in configuration mode until a bitstream is received through the JTAG or the Serial interface. The JTAG interface can be operated either from a desktop PC USB connection, or direct from it's board header; the serial interface can be accessed from a memory drive attached to the USB HID port of the board.

Another on-board jumper, JP12, selects the voltage level of the FPGA bank 2 pins (either on 3.3V or 2.5V). If JP12 is not placed, the pull-ups for CCLK, DONE, PROGRAM_B and INIT_B are not provided. If this happens, the FPGA is held in the reset state and cannot be programmed.

Figure 6.1 shows the available configuration interfaces for the Digilent Atlys board.

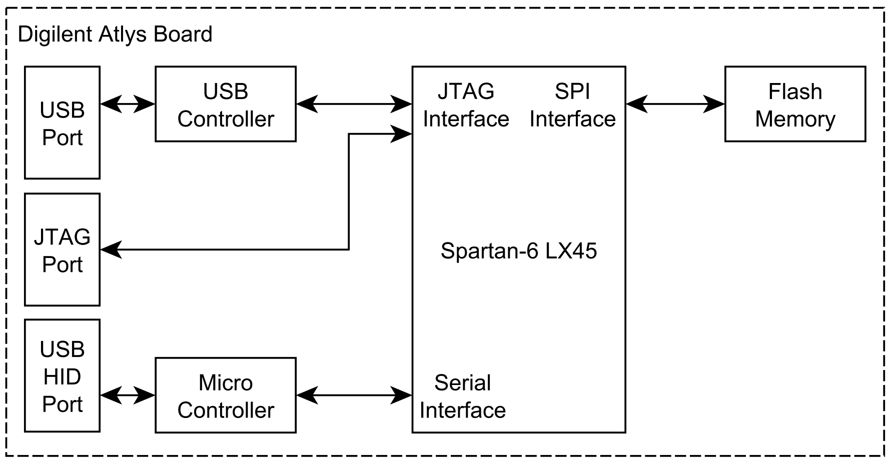


Figure 6.1: Atly Configuration Interfaces

## 6.2 Toolchain

Xilinx offers several tools to assist in the embedded system design process. These tools are collectively called the Integrated Software Environment (ISE) Design Suite. ISE is composed of the following modules:

- Integrated Software Environment components
- PlanAhead design analysis software
- ChipScope Pro on-chip debugging software
- Embedded Development Kit

### 6.2.1 The Embedded Development Kit

The Embedded Development Kit (EDK) is a suite of tools and IPs used to design a complete embedded microprocessor system targeting Xilinx FPGA devices. It was conceived to assist in all the development phases of the embedded design process. EDK depends on ISE components to synthesize the microprocessor hardware design, to map it to a specific FPGA device, and to generate and download the resulting bitstream. EDK, however, does not require the use of the ChipScope Pro and PlanAhead softwares. These act only as supporting tools to help simplify the design flow and verification process.

In EDK, the Development process is split in two separate and independent flows: the hardware and the software platform. The hardware platform typically consists of one or more microprocessors, device controllers and memory blocks, interconnected via microprocessor buses. On top of that, it also contains port connections to communicate with off-chip modules. The software platform consists of a collection of device drivers and embedded applications that run over the hardware platform. Figure 6.2 provides an overview of the EDK structure and development flow.

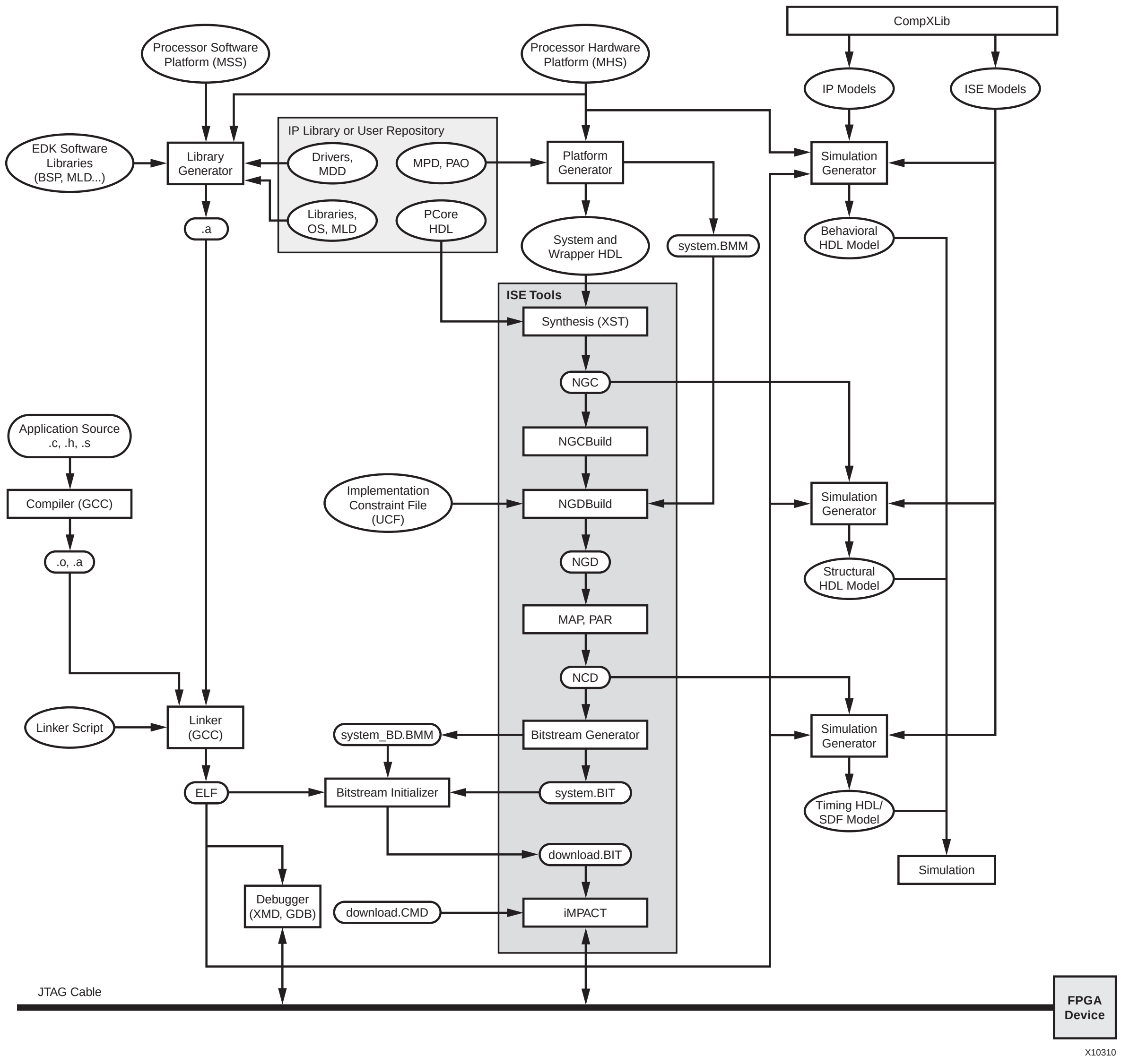


Figure 6.2: Embedded Development Kit (EDK) Tools Architecture

#### *6.2.1.1 Xilinx Platform Studio*

The Xilinx Platform Studio (XPS) is the development environment used for designing the hardware portion of the embedded processor system. It can be run either in batch mode, or using the GUI. Among the features provided by XPS, the main ones are:

- Ability to add processor and peripheral cores, edit core parameters and assign bus connections to generate a Microprocessor Hardware Specification (MHS) file.
- Ability to manage the tool flow dependency.
- Ability to generate and view a system block diagram and/or design report.
- Ability to export hardware specification files for SDK use.

#### *6.2.1.2 Software Development Kit*

The Software Development Kit (SDK) is an integrated development environment, complementary to XPS, that is used for C/C++ embedded software application development and verification. It is built on the Eclipse open-source framework. The SDK has the following built-in features:

- Independent installation from ISE and XPS.
- Integrated environment for debugging and profiling of embedded targets.
- Built-in interface to generate linker scripts for software applications.
- Support for single and multi-processor systems.
- Automatic make file generation.

## 6.3 Hardware Design

In order to create the EDK base-level project for the configuration controller, the first thing that was required was to download the Base System Builder (BSB) support files for the Atlys development board (DSD-0000332). The BSB is a wizard in the Xilinx Platform Studio that provides a graphical interface to create a new embedded system project based on the available board resources.

Once the support files were installed in the development environment, the hardware design proceeded based on the block diagram presented in figure 5.4.

### 6.3.1 Microprocessor

The XPS provides an automated flow to integrate the MicroBlaze processor in the user design. The MicroBlaze is a soft-core reduced instruction set computer (RISC) embedded processor optimized for implementation in Xilinx FPGAs. It has over 70 user-configurable options, providing enough flexibility to support virtually any system, regardless if it requires a very small footprint microcontroller, or a high performance compute-intensive platform (Xilinx UG081).

For the configuration controller implementation, the default configurations were used. This architecture provides the best compromise between area and performance.

### 6.3.2 Bus Connections

The MicroBlaze processor is based on the Harvard architecture, with separate bus interface units for data and instruction accesses. It supports four different memory interfaces: Local Memory Bus (LMB), the AMBA AXI4 interface (AXI4), the IBM Processor Local Bus (PLB) and the Xilinx CacheLink (XCL). The LMB provides single-cycle access to on-chip dual-port block RAM. The AXI4 and PLB interfaces provide a connection to both on-chip and off-chip peripherals and memory. The CacheLink interface is intended for use with specialized external memory controllers.

Although both AXI4 and PLB interfaces can be used to connect peripherals and memory, the PLB interface is being discontinued by Xilinx in future FPGA families. For this reason, it is not recommended for designs that might be ported to newer devices. Based on this note, only AXI4 compliant cores were used during the implementation of the configuration controller.

AXI is part of ARM AMBA, an open standard specification for the connection and management of different cores in an embedded platform. AMBA 4 is the last release of this family, and it includes the specification for the AXI4 protocol. There are three types of AXI4 interfaces:

- AXI4-Lite, for simple, low-throughput memory-mapped communications.
- AXI4, for high-performance memory-mapped requirements.
- AXI4-Stream—for high-speed streaming data.

#### *6.3.2.1 AXI4*

The AXI4 specification describe an interface between a single master and a single slave, connected together using a structure called an Interconnect Block. The communication between master and slave can occur simultaneously, since different channels for read/write operations and address/data transactions are defined. Typically, data transfer sizes can vary according to the operation, but are always limited by the maximum burst size of 256 data beats.

#### *6.3.2.2 AXI4-Lite*

AXI4-Lite is very similar to AXI4 in various aspects, but it has some limitations; the most significant is not supporting burst mode. Memory access is performed 1 data transfer per transaction.

#### *6.3.2.3 AXI4-Stream*

The AXI4-Stream interface defines a single channel for streaming data, modeled after the write data channel of the AXI4. IPs using this interface are optimized for performance and target applications that are data end-points.

### 6.3.3 ACE Player

In order to drive the device under test reconfiguration from within the configuration controller, the bitstreams should be decoded into JTAG operations, and these operations, into JTAG bus signals. Often, this is a vendor-specific process with proprietary algorithms, so this task can become quite complex.

#### *6.3.3.1 SVF Format*

In an effort to standardize this process in a compact and portable fashion, the industry has developed a standard file format called Serial Vector Format (SVF). This human-readable ASCII file is used to describe the JTAG chain operations required to shift the configuration data into the device chain. Xilinx provides software that can directly generate SVF files for in-system programming solutions, such as this platform. Information regarding this process and the tool used to generate this files are described in details in the application note (Xilinx XAPP503).

#### *6.3.3.2 ACE Format*

Although SVF is a powerful language for managing configuration through the JTAG interface, due to its large size it is not recommended for embedded systems with memory and performance constraints. For this reason, Xilinx has developed the Advanced Configuration Environment (ACE) format. This format is generated from complex algorithms that translate the SVF file into a binary structure already containing the required JTAG bus signaling. This optimization greatly reduces the processing and memory requirements of the embedded system, while still preserving compatibility with the industry standards for the SVF format and JTAG interface. The complete definition of this format is presented in the application note (Xilinx XAPP424).

#### *6.3.3.3 HDL Player*

Also in (Xilinx XAPP424), Xilinx offers a VHDL IP to parse the ACE file structure. This IP can be integrated in any design in order to provide a synchronous interface for a JTAG in-system programming (ISP) solution. The user design, however, is still responsible for driving all control signals of this core. The HDL Player interface is shown on figure 6.3.

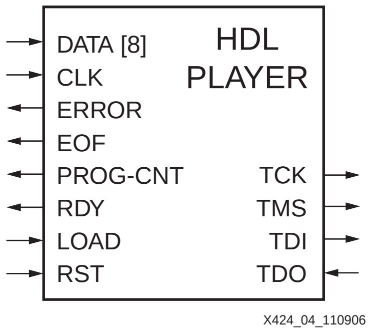


Figure 6.3: HDL Player Interface

#### *6.3.3.4 AXI ACE Player*

In order to integrate the HDL Player provided by Xilinx in the configuration controller platform, an AXI4 compliant core was developed. This core provides the necessary signals in order to properly drive the HDL Player signals, at the same time that it provides an interface to allow its management from the embedded processor. Other than that, it also contains data buffers used to avoiding player idle states, maximizing the output data rate. Figure 6.4 shows the block diagram for the developed AXI ACE Player core.

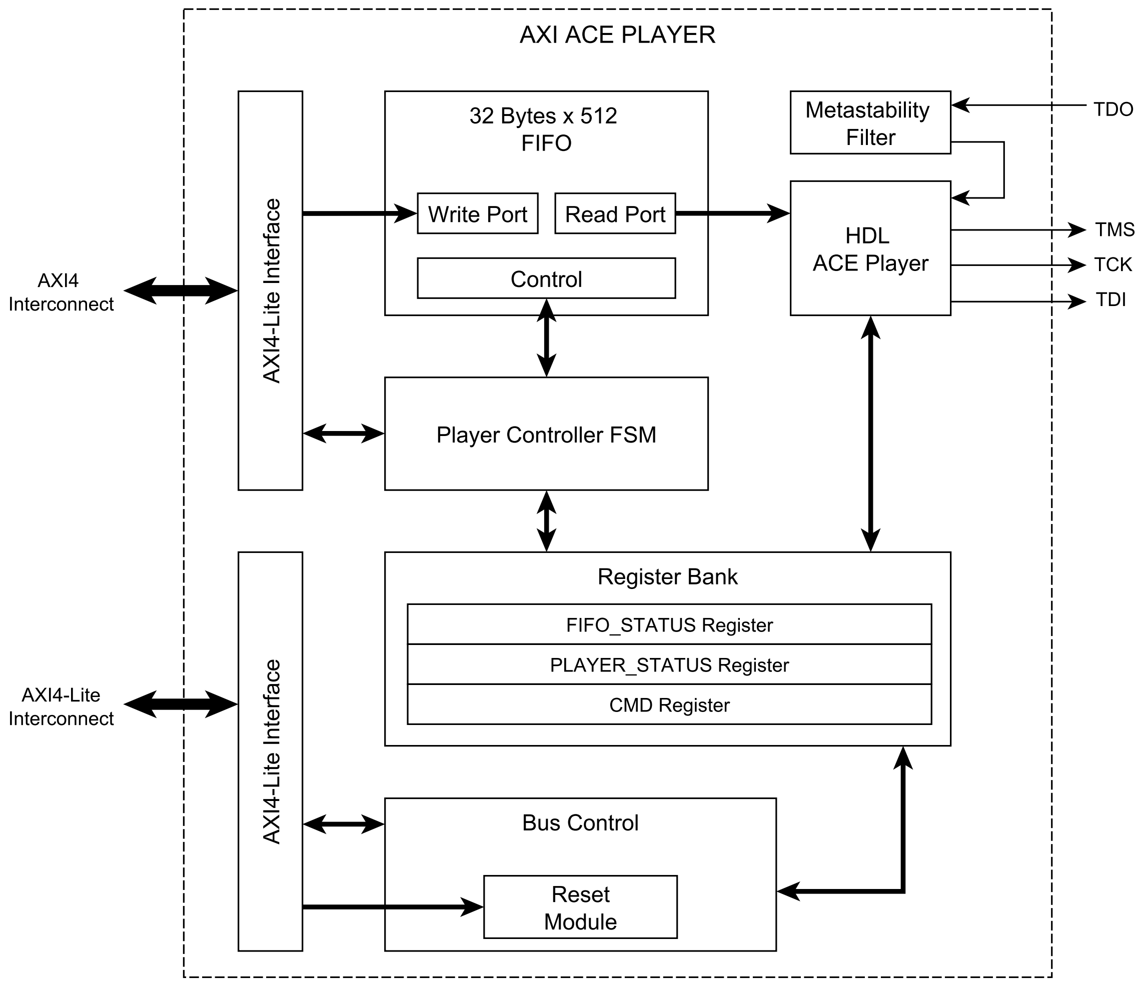


Figure 6.4: AXI ACE Player Core

As it can be seen from the block diagram, the AXI ACE Player has two AXI interfaces: one AXI4 and one AXI4-Lite. This distinguishment comes from the limitation imposed by normal AXI4-Lite connections, in which only 1 data word can be transferred in each transaction. This way, by separating data and control interfaces, ACE file data can be transferred to the player using a high-performance channel, while control sequences can be operated in a familiar register-based fashion.

As a mean to maximize the data transfer rate between the configuration manager and the DUT, a FIFO was placed together with the HDL player. This aims to eliminate possible idle states of the player caused by the latency of processor bus transactions. This FIFO is a synchronous dual-port 32-byte wide x 512 in depth memory that is capable of storing twice the data that can be transmitted in a single AXI-4 data beat. Thus, while the player is consuming data from the FIFO, the processor can already start a new transfer cycle. In other words, the FIFO serves as a data buffer to avoid the player starvation.

In order to manage the FIFO and the player, a finite state machine (FSM) was included in the design. This FSM is responsible for taking data out from the FIFO and sending it to the HDL player. To accomplish this, it monitors both entities signals to determine if the player is requesting data, and if it is possible to take data from the buffer. It is designed in a way to avoid any illegal states, such as FIFO underrrun and player overrun.

All player and FIFO statuses can be monitored from the register bank provided from the AXI4-Lite interface. This registers contain information about the FIFO vacancy and the player current cycle. From this interface, the microprocessor application can determine if there is space in the buffer to start a new data transfer, or if the player was halted from an error during DUT configuration. A full detailed description of the register bank content can be seen in figure 6.5 and tables 6.1, 6.2 and 6.3;

Additionally, to eliminate problems related to metastable signals received from different clock domains, a metastability filter was added to the input JTAG TDO signal. This filter was constructed by cascading two D-type flip-flops.

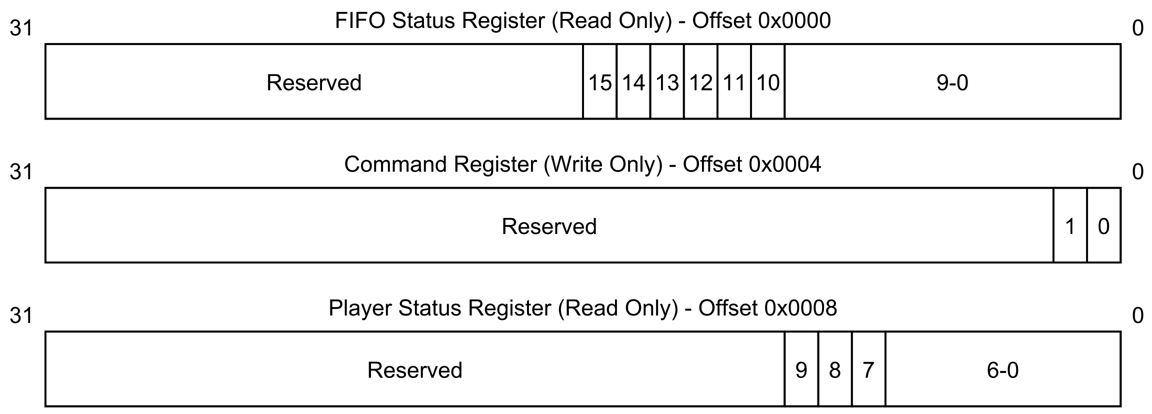

31
FIFO Status Register (Read Only) - Offset 0x0000
0
Reserved
15
14
13
12
11
10
9-0
31
Command Register (Write Only) - Offset 0x0004
0
Reserved
1
0
31
Player Status Register (Read Only) - Offset 0x0008
0
Reserved
9
8
7
6-0

Figure 6.5: AXI ACE Player Core Registers

| Bits | Field Name | Description |
|---|---|---|
| 0-9 | DATA_CNT | Number of bytes available at the FIFO. |
| 10 | EMPTY | FIFO is empty. |
| 11 | FULL | FIFO is full. |
| 12 | OVERF | FIFO was overrun. This is a sticky bit. |
| 13 | UNDERF | FIFO was underrun. This is a sticky bit. |
| 14 | VLD | Data at FIFO output is ready and valid. |
| 15 | HALF | FIFO has only half of its capacity, or less. |
| 16-31 | Reserved | This bits are reserved and should not be used. |

Table 6.1: FIFO Status Register Description

| Bits | Field Name | Description |
|---|---|---|
| 0 | RESET | Active low global core Reset. |
| 1 | TRIGGER | Sets the trigger for the fault event signal. The following condition must be valid for the fault event signal to be set: TRIGGER and (not EOF) and (not ERROR). |
| 2-31 | Reserved | This bits are reserved and should not be used. |

Table 6.2: Command Register Description

| Bits | Field Name | Description |
|---|---|---|
| 0-6 | PPC | Current ACE Player FSM state. |
| 7 | EOF | ACE Player finished to transfer file. This is a sticky bit. |
| 8 | ERROR | ACE Player error indication. This is a sticky bit. |
| 9 | READY | ACE Player ready to receive data. |
| 10-31 | Reserved | This bits are reserved and should not be used. |

Table 6.3: Player Status Register Description

### 6.3.4 IP Core Library

Aside from the MicroBlaze embedded processor and the AXI ACE Player core, several other cores from the EDK IP Library were instantiated to provide the necessary microprocessor buses, local memory controllers, RAM blocks and off-chip peripheral connections. Table 6.4 summarizes the cores utilized in the configuration controller FPGA design.

| Instance | IP Core | Version |
|---|---|---|
| proc_sys_reset_0 | proc_sys_reset | 3.00.a |
| microblaze_0_intc | axi_intc | 1.01.a |
| microblaze_0_ilmb | lmb_v10 | 2.00.b |
| microblaze_0_i_bram_ctrl | lmb_bram_if_cntlr | 3.00.b |
| microblaze_0_dlmb | lmb_v10 | 2.00.b |
| microblaze_0_d_bram_ctrl | lmb_bram_if_cntlr | 3.00.b |
| microblaze_0_bram_block | bram_block | 1.00.a |
| microblaze_0 | microblaze | 8.20.b |
| debug_module | mdm | 2.00.b |
| clock_generator_0 | clock_generator | 4.03.a |
| axi4lite_0 | axi_interconnect | 1.05.a |
| axi4_0 | axi_interconnect | 1.05.a |
| RS232_Uart_1 | axi_uartlite | 1.02.a |
| MCB_DDR2 | axi_s6_ddrx | 1.05.a |
| Ethernet_Lite | axi_ethernetlite | 1.01.b |
| Digilent_QuadSPI_Cntrl | d_qspi_axi | 1.00.a |
| axi_timer_0 | axi_timer | 1.03.a |
| axi_cdma_0 | axi_cdma | 3.02.a |
| axi_ace_player_0 | axi_ace_player | 2.00.a |

Table 6.4: IP Cores Used In The Configuration Controller

*axi_intc*

The AXI INTC core provides an AXI4-Lite interface-based interrupt controller of up to 32 signals to be connected to the single MicroBlaze interrupt port.

*lmb_v10*

The LMB V10 module is used as the LMB interconnect for the MicroBlaze processor. It provides the processor with a LMB interface.

*lmb_bram_if_cntlr*

The LMB BRAM Interface Controller is the interface between the LMB module and the bram_block peripheral.

*bram_block*

The BRAM Block is a configurable dual-port memory module used to provide MicroBlaze a local memory to host a embedded application, and also to cache data and instructions from the DDR memory.

*mdm*

The MicroBlaze Debug Module (MDM) provides a JTAG-based debugging interface compliant with the AXI4-Lite protocol.

*clock_generator*

The Clock Generator core is used to generate the required clock tree for the entire design.

*axi_interconnect*

The AXI Interconnect core is used to connect one or more AXI memory-mapped master devices to one or more memory-mapped slave devices.

*axi_uartlite*

The Universal Asynchronous Receiver Transmitter (UART) Lite provides a AXI4-Lite compliant interface for asynchronous serial data transfer, with configurable baud rate, number of data bits and parity information.

*axi_s6_ddrx*

The S6 DDRx core provides an AXI4 compliant interface for high-performance connections to DDR3 and DDR2 SDRAMs.

*axi_ethernetlite*

The AXI Ethernet Lite core is used to provide a AXI4-Lite compliant Ethernet PHY controller to the platform.

*d_qspi_axi*

The Digilent Quad SPI controller provides access to the on-board SPI flash memory through the AXI4-Lite interface.

*axi_timer*

The AXI Timer/Counter is a 32/64-bit timer module that attaches to the AXI4-Lite interface in order to provide the platform with timer functions. This is used to overcome the MicroBlaze lack of internal time registers.

*axi_cdma*

The AXI CDMA provides high-speed direct memory access (DMA) transactions between a memory-mapped source and destination addresses mapped in the AXI4 interconnection.

### 6.3.5 Memory Layout

In order to communicate with memory-mapped peripherals on the processor buses, the MicroBlaze uses a 32-bit word to address any memory position in the range between 0x00000000 and 0xFFFFFFFF. Table 6.5 shows the memory layout used in the configuration controller hardware platform specification.

| Instance | Base Address | Size | Bus Interface |
|---|---|---|---|
| microblaze_0_d_bram_ctrl | 0x00000000 | 64KB | microblaze_0_dlmb |
| microblaze_0_i_bram_ctrl | 0x00000000 | 64KB | microblaze_0_ilmb |
| RS232_Uart_1 | 0x40600000 | 64KB | axi4lite_0 |
| Ethernet_Lite | 0x40E00000 | 64KB | axi4lite_0 |
| microblaze_0_intc | 0x41200000 | 64KB | axi4lite_0 |
| debug_module | 0x41400000 | 64KB | axi4lite_0 |
| axi_timer_0 | 0x41C00000 | 64KB | axi4lite_0 |
| axi_ace_player_0 | 0x7D000000 | 64KB | axi4lite_0 |
| axi_cdma_0 | 0x7E200000 | 64KB | axi4lite_0 |
| Digilent_QuadSPI_Cntlr | 0x7E400000 | 64KB | axi4lite_0 |
| axi_ace_player_0 | 0xC4000000 | 64KB | axi4_0 |
| MCB_DDR2 | 0xC8000000 | 128MB | axi4_0 |

Table 6.5: Memory Map for Processor Microblaze_0

### 6.3.6 Design Summary

Table 6.6 summarizes the synthesis results for each IP core in the hardware platform.

Table 6.7 provides an simplified overview of the logic utilization. From this table it can be noticed that the configuration controller design requires approximately only half of the device slices, and only one third of the available IOBs. This demonstrates that even smaller FPGAs could be used to implement this platform, or that the remaining logic of the device could be used to extend the platform's functionality.

## 6.4 Software Design

Once the hardware platform has been defined, its definitions can be exported to the SDK in order to begin the software design. There are three major aspects of the software design process: configuring the BSP, writing the software application and creating a bootloader.

### 6.4.1 BSP

The Board Support Package (BSP) is a generic name that refers to the software components required to support a given operating system, and its programming environment, in a specific hardware design. The BSP provides upper software layers the abstraction

| Instance | Logic Utilization | | |
|---|---|---|---|
| | Flip-Flops | LUTs | BRAMs |
| axi_ace_player_0_wrapper | 304 | 525 | - |
| clock_generator_0_wrapper | - | 1 | - |
| axi4_0_wrapper | 1625 | 1720 | - |
| axi_cdma_0_wrapper | 1507 | 1446 | 5 |
| axi_timer_0_wrapper | 217 | 312 | - |
| digilent_quadspi_cntlr_wrapper | 432 | 574 | 1 |
| ethernet_lite_wrapper | 607 | 726 | 2 |
| mcb_ddr2_wrapper | 823 | 1309 | - |
| rs232_uart_1_wrapper | 90 | 123 | - |
| axi4lite_0_wrapper | 180 | 392 | - |
| debug_module_wrapper | 131 | 142 | - |
| microblaze_0_wrapper | 2202 | 2403 | 3 |
| microblaze_0_bram_block_wrapper | - | - | 32 |
| microblaze_0_d_bram_ctrl_wrapper | 2 | 6 | - |
| microblaze_0_dlmb_wrapper | 1 | - | - |
| microblaze_0_i_bram_ctrl_wrapper | 2 | 6 | - |
| microblaze_0_ilmb_wrapper | 1 | - | - |
| microblaze_0_intc_wrapper | 48 | 74 | - |
| proc_sys_reset_0_wrapper | 69 | 55 | - |
| **total** | **8241** | **9814** | **43** |

Table 6.6: Configuration Controller Synthesis Summary

| Logic Utilization | Used | Available | Utilization |
|---|---|---|---|
| Number of Slice Registers | 7,119 | 54,576 | 13% |
| Number of Slice LUTs | 8,125 | 27,288 | 29% |
| Number of occupied Slices | 3,368 | 6,822 | 49% |
| Number of bonded IOBs | 79 | 218 | 36% |

Table 6.7: Configuration Controller Global Utilization Summary

level necessary to mask the low-level device drivers used to perform direct hardware accesses.

Since FPGA-based architectures are intrinsically reconfigurable, fixed board support packages cannot be provided. Therefore, a custom board support package must be generated for each different hardware design that is created.

The Xilinx SDK provides the option to generate a BSP based on the architecture definition imported from XPS. Xilinx already provides device drivers for all IP cores in its library, so user only needs to implement the ones related to proprietary cores. On top of that, two different kernel modules are offered: Standalone and Xilkernel.

Standalone is a simple, low-level software layer. It provides direct access to basic processor features such a cache management, interrupt control and exception handling. Also, it provides basic features of a hosted environment, such as standard input/output and code profiling. On the other hands, Xilkernel is a lightweight kernel based on POSIX services. It includes libraries to support threads, synchronization mechanisms, message passing and scheduling policies.

For the development of the configuration controller prototype the standalone kernel was chosen primarily because the services of the xilkernel were not going to be used.

#### *6.4.1.1 ACE Player Driver*

In order to complement the standalone kernel with functions to manage the ACE Player IP, a custom driver was developed and integrated with the BSP. This driver provides the necessary API for the embedded application running on the Microblaze processor to start and monitor the reconfiguration process of the DUT. Three functions are provided, as shown in listing 6.1, 6.2 and 6.3.

Listing 6.1: ACE Player Driver API

```
NAME
ace_player_program () - Programs DUT with an ACE file

SYNOPSIS
t_status ace_player_program
(
   Xuint32 baseAddress, /* Base Address of ACE Player */
   Xuint32* pAceFile,   /* Pointer to ACE file */
   Xuint32 length       /* Length of file in bytes */
);

DESCRIPTION
This function is used to transfer an ACE file through the
   ACE Player core residing in <baseAddress>.
The ACE file pointed by <pAceFile> must be 64-bit aligned.

RETURNS
OK if file was successfully transferred, otherwise ERROR.

ERRCODE
INVALID_FILE_ERR - Invalid file format.
TDO_CHECK_ERR - Mismatch between expected and actual JTAG
   TDO response.
```

The first function, ace_player_program(), is the main function of the driver. It is called every time the application wants to transmit a new bitstream to the DUT. Its flowchart is shown on figure 6.6 and 6.7.

In case a problem happens during the transmission, and error code is set on the driver. To recover this error code, the application can call ace_player_get_error_code(). If the purpose is only to log the code, this should be sufficient, but in case an error message needs to follow, ace_player_get_error_str() can be also called. The flowchart for these functions are showed in figures 6.8 and 6.9, respectively.

### 6.4.2 Application

The main application running on the MicroBlaze processor is the responsible for coordinating the DUT reconfiguration through the ACE player. Although in the proposed platform description this application contains features such as downloading the partial

Listing 6.2: ACE Player Driver API (Continued)

```
NAME
t_status ace_player_get_error_code() − Gets the ACE Player
   Error Code

SYNOPSIS
t_status ace_player_get_error_code
(
   Xuint32 baseAddress, /* Base Address of ACE Player */
   Xuint32* errCode     /* Returned error code */
);

DESCRIPTION
This function is used to retrieve the ERRCODE set by
   ace_player_program() in case of problems during the ACE
   file programming.

RETURNS
OK if error code was successfully retrieved, otherwise
   ERROR.
```

Listing 6.3: ACE Player Driver API (Continued)

```
NAME
ace_player_get_error_str() − Gets a string representation
   of ERRCODE

SYNOPSIS
char* ace_player_get_error_str
(
   Xuint32* errCode     /* Error code to translate */
);

DESCRIPTION
This function is used to get a string representation of
   ERRCODE.
RETURNS
String representation of the ERRCODE.
```

bitstreams from a remote FTP server and managing time using hardware timers, the prototype contains only the basic set of functionalities required to perform a fault injection test campaign.

Since the bitstreams cannot be retrieved through Ethernet, they have to be embedded directly on the application binary file to emulate a remote transmission. To accomplish this, the ACE files were converted to MicroBlaze ELF32 objects, and linked together with the application executable. Appendix A.1 shows how to operate the GNU objcopy utility in order to produce these objects. Appendix A.2 and A.3 shows how to operate the GNU objdump utility in order to retrieve the entry point for the data section in each object, so they can be referenced in the application configuration scheduler.

In order to communicate directly with the hardware platform, this application makes use of the ACE Player driver, and other low-level primitives of the standalone kernel. Figure 6.10 shows the application flowchart.

### 6.4.3 Bootstrap

The Xilinx SDK provides a simple SREC bootloader example that can be used for loading software images from a non volatile memory. This is an industry-standard format developed by Motorola, used for transmitting binary files to target systems, and composed by a single ASCII hexadecimal text file.

The provided example, however, does not provide any support for SPI flash memories, as the one found in the Atlys board. In order to overcome this problem, the QUAD SPI flash driver developed by Digilent (DSD-0000332) was integrated to the example source code. At the end, a fully functional bootloader for the configuration controller board was obtained.

This bootloader assumes that the target SREC image is fit for the MicroBlaze architecture, and that it does not overlap any bootloader sections in memory. This way, the application and the bootloader had to assigned to separate physical memories in the hardware. While the first resides in the DDR, the second runs from the embedded block RAM. Tables 6.8 and 6.9 shows the linker attributes of each software item.

| Section | Size | Allocated |
|---|---|---|
| .text | 0x00004502 | microblaze_0_i_bram_ctrl_microblaze_0_d_bram_ctrl |
| .data | 0x00000138 | microblaze_0_i_bram_ctrl_microblaze_0_d_bram_ctrl |
| .bss | 0x000005DE | microblaze_0_i_bram_ctrl_microblaze_0_d_bram_ctrl |
| .stack | 0x00000400 | microblaze_0_i_bram_ctrl_microblaze_0_d_bram_ctrl |
| .heap | 0x00000000 | microblaze_0_i_bram_ctrl_microblaze_0_d_bram_ctrl |
| **Total** | **0x00004C18 bytes** | |

Table 6.8: Bootstrap Linker Attributes

## 6.5 Design Integration

Once the hardware and software designs were completed, the system integration could be performed. The bitstream generated from the hardware design was merged with the bootstrap image to form a new bitstream. This new configuration file was generated using the Data2MEM tool from within the SDK environment. This tool performs data translation between multiple blocks of BRAMs to create a contiguous logical address space for the embedded application image. This way, the microprocessor can jump to

| Section | Size | Allocated |
|---|---|---|
| .text | 0x00005AAA | MCB_DDR2_S0_AXI_BASEADDR |
| .data | 0x00000574 | MCB_DDR2_S0_AXI_BASEADDR |
| .bss | 0x01F001EE | MCB_DDR2_S0_AXI_BASEADDR |
| .stack | 0x00100000 | MCB_DDR2_S0_AXI_BASEADDR |
| .heap | 0x01E00000 | MCB_DDR2_S0_AXI_BASEADDR |
| **Total** | **0x01F0620C bytes** | |

Table 6.9: Application Linker Attributes

the start of the executable code and start to run soon as the FPGA is configured. Then, in order to generate a PROM file and program the on-board SPI flash, application note (Xilinx XAPP951) was used as reference.

After, the application SREC file was created from the compiled ELF binary. To accomplish this, the GNU objcpy utility was manually invoked, as shown in appendix A.4. This SREC file was also programmed in the flash, so the bootloader could read it.

## 6.6 Platform Operation

At system initialization, after FPGA power-up, it is configured automatically from the SPI Flash containing the bitstream that was programmed previously. Once the device enters user-mode, the microprocessor jumps to the reset vector and starts to execute the bootloader. The bootloader copies the application from flash to DDR memory, and at the end, jumps to its entry point. The application, then, runs until returning from the main() function.

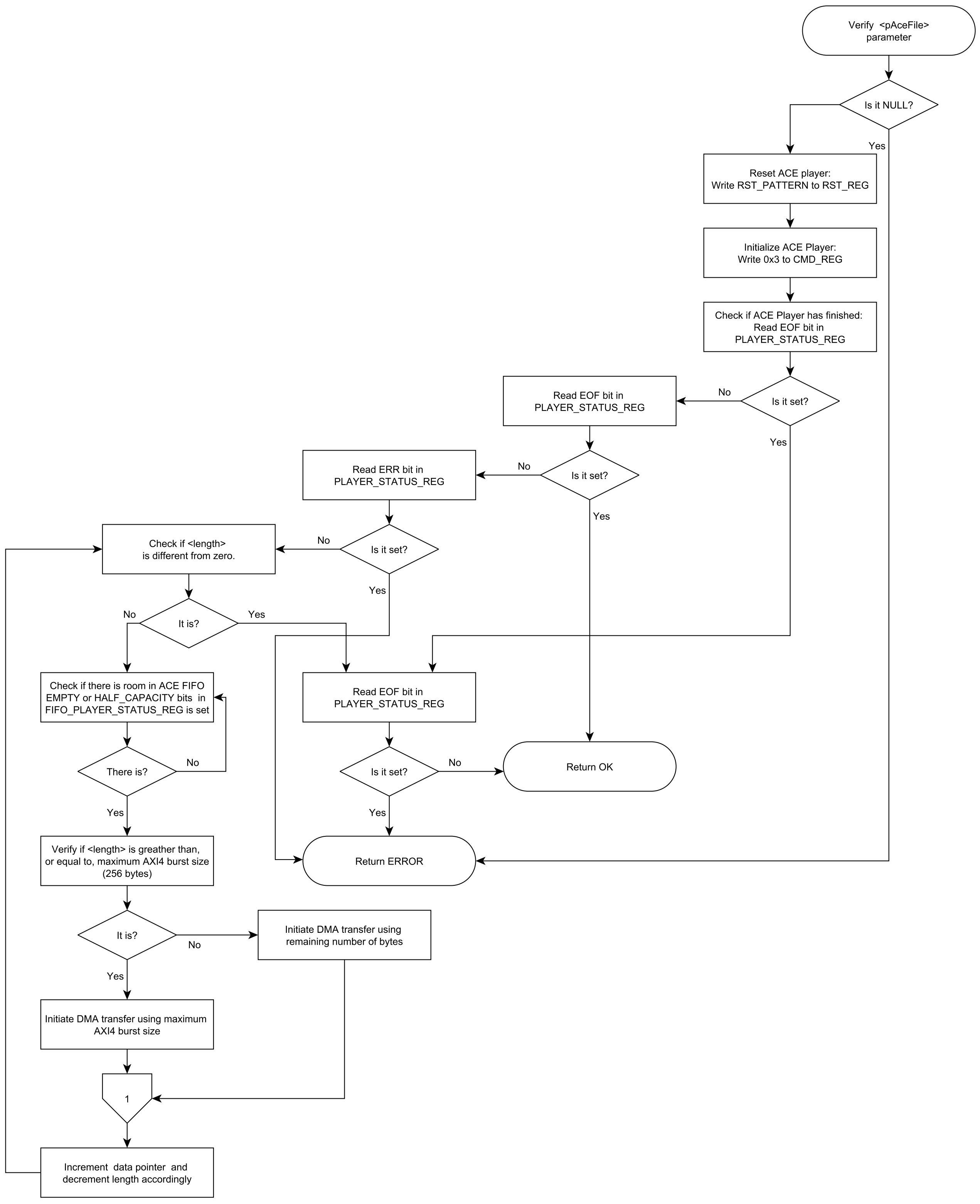

Verify <pAceFile> parameter
Is it NULL?
Yes
Reset ACE player: Write RST_PATTERN to RST_REG
Initialize ACE Player: Write 0x3 to CMD_REG
Check if ACE Player has finished: Read EOF bit in PLAYER_STATUS_REG
Is it set?
No
Yes
Read EOF bit in PLAYER_STATUS_REG
Is it set?
No
Yes
Read ERR bit in PLAYER_STATUS_REG
Is it set?
No
Yes
Check if <length> is different from zero.
It is?
No
Yes
Check if there is room in ACE FIFO EMPTY or HALF_CAPACITY bits in FIFO_PLAYER_STATUS_REG is set
There is?
No
Yes
Read EOF bit in PLAYER_STATUS_REG
Is it set?
No
Yes
Return OK
Return ERROR
Verify if <length> is greather than, or equal to, maximum AXI4 burst size (256 bytes)
It is?
No
Yes
Initiate DMA transfer using remaining number of bytes
Initiate DMA transfer using maximum AXI4 burst size
1
Increment data pointer and decrement length accordingly

Figure 6.6: Flowchart for ace_player_program

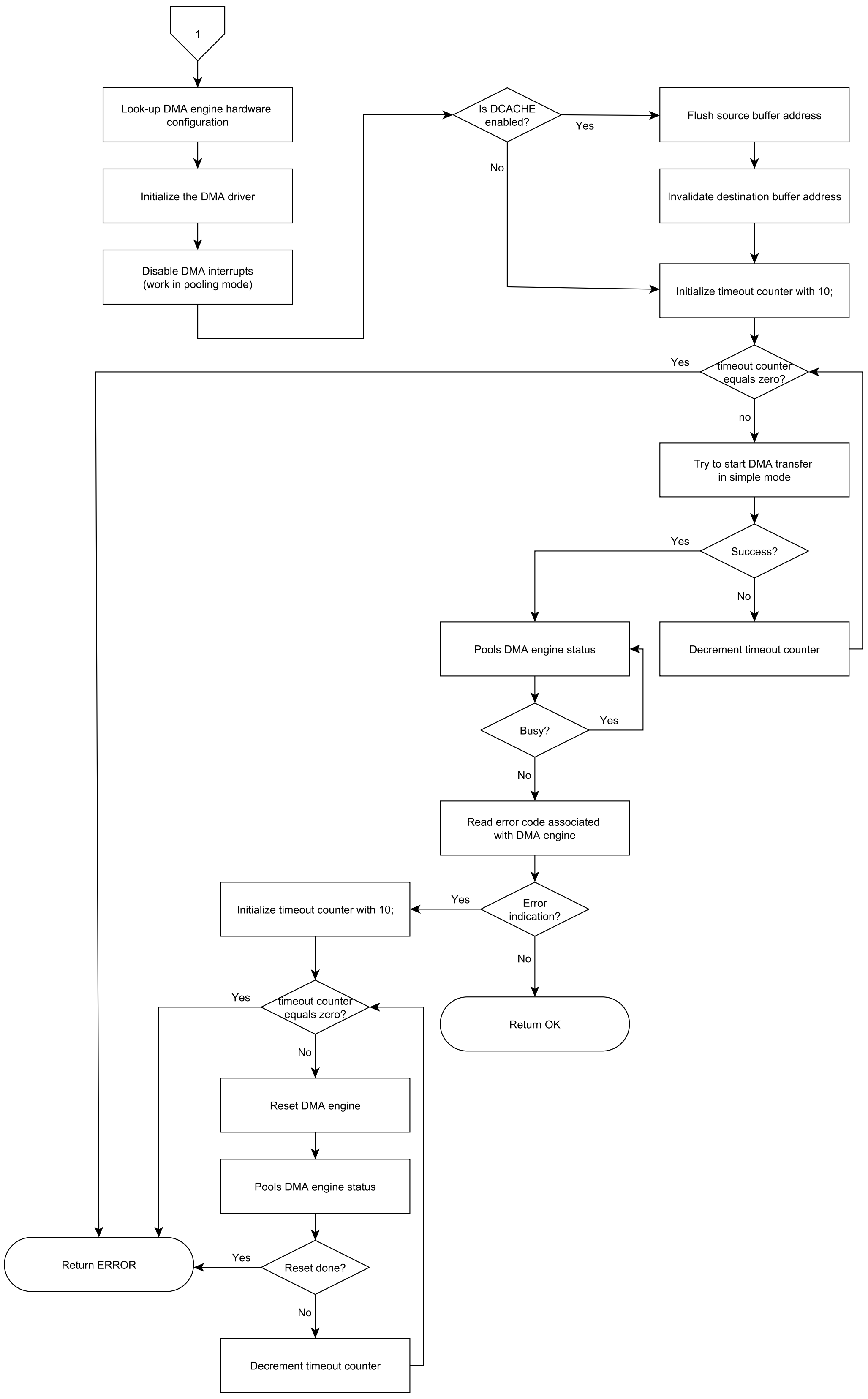

1
Look-up DMA engine hardware configuration
Initialize the DMA driver
Disable DMA interrupts (work in pooling mode)
Is DCACHE enabled?
Yes
No
Flush source buffer address
Invalidate destination buffer address
Initialize timeout counter with 10;
timeout counter equals zero?
Yes
no
Try to start DMA transfer in simple mode
Success?
Yes
No
Pools DMA engine status
Decrement timeout counter
Busy?
Yes
No
Read error code associated with DMA engine
Error indication?
Yes
No
Initialize timeout counter with 10;
Return OK
timeout counter equals zero?
Yes
No
Reset DMA engine
Pools DMA engine status
Reset done?
Yes
No
Return ERROR
Decrement timeout counter

Figure 6.7: Flowchart for ace_player_program (continued)

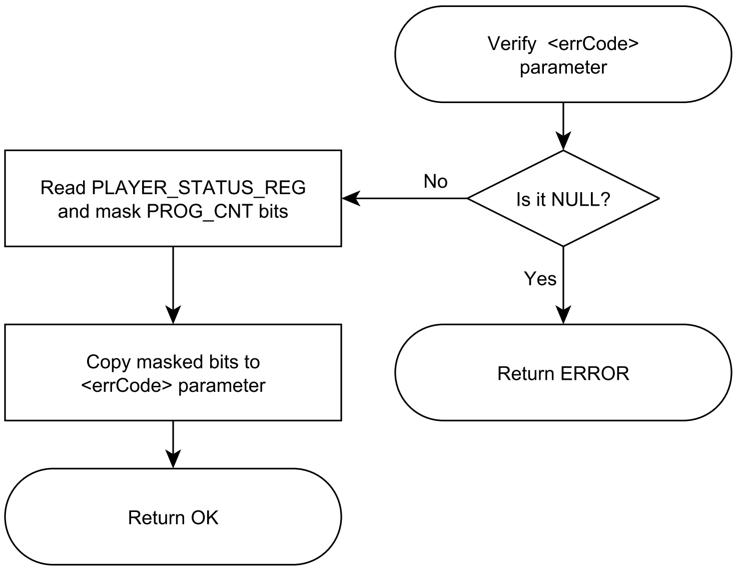

Verify <errCode> parameter
Is it NULL?
No
Read PLAYER_STATUS_REG and mask PROG_CNT bits
Yes
Return ERROR
Copy masked bits to <errCode> parameter
Return OK

Figure 6.8: Flowchart for ace_player_get_error_code

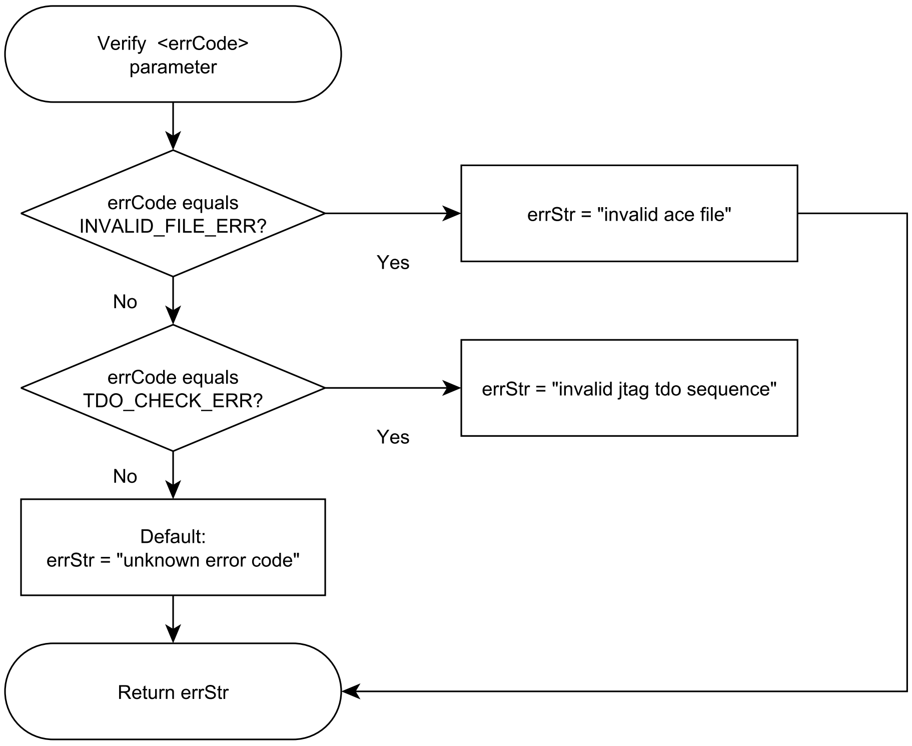

Verify <errCode> parameter
errCode equals INVALID_FILE_ERR?
errStr = "invalid ace file"
Yes
No
errCode equals TDO_CHECK_ERR?
errStr = "invalid jtag tdo sequence"
Yes
No
Default: errStr = "unknown error code"
Return errStr

Figure 6.9: Flowchart for ace_player_get_error_str

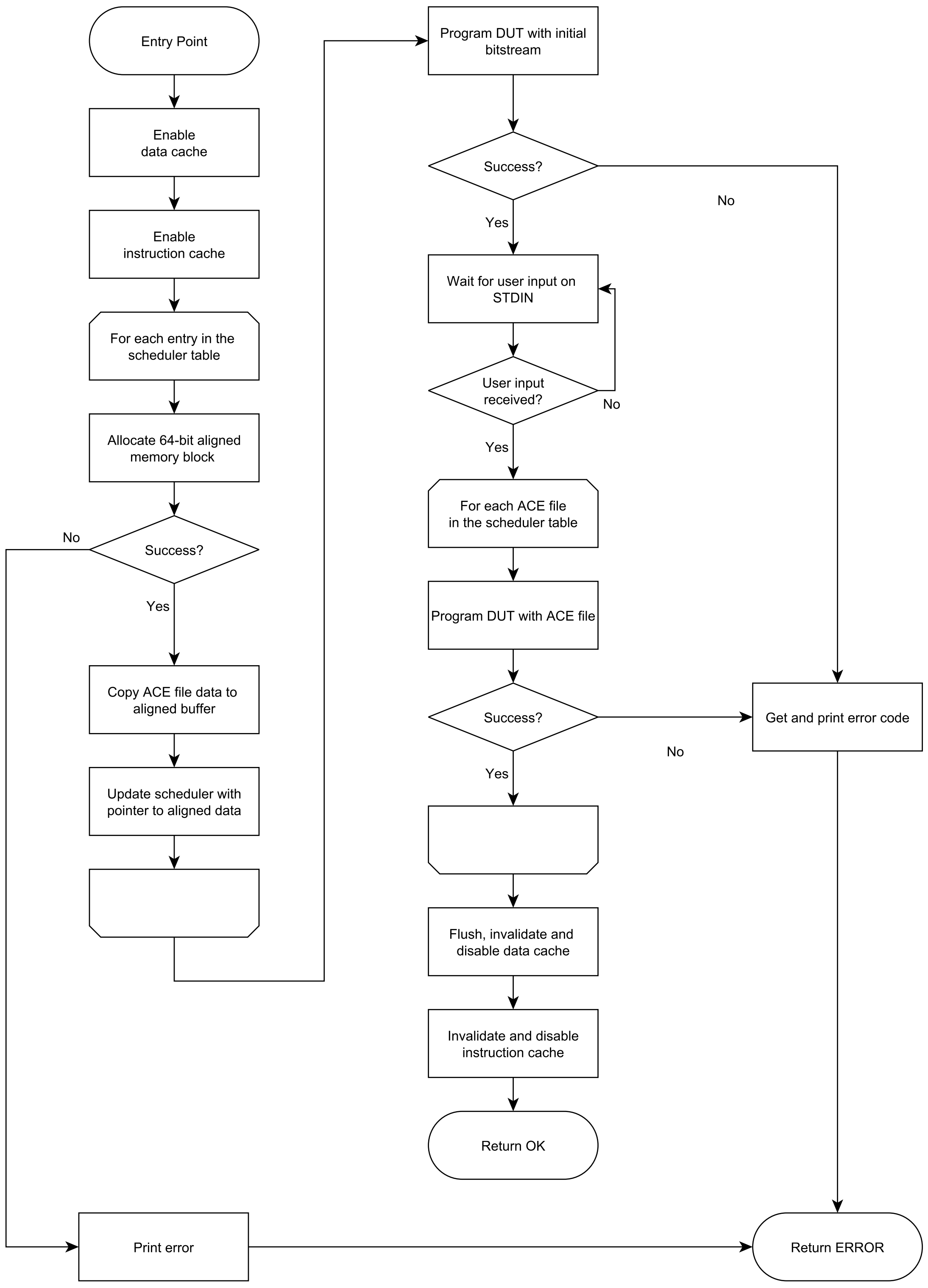

Entry Point
Enable data cache
Enable instruction cache
For each entry in the scheduler table
Allocate 64-bit aligned memory block
Success?
No
Yes
Copy ACE file data to aligned buffer
Update scheduler with pointer to aligned data
Print error
Program DUT with initial bitstream
Success?
Yes
No
Wait for user input on STDIN
User input received?
No
Yes
For each ACE file in the scheduler table
Program DUT with ACE file
Success?
Yes
No
Flush, invalidate and disable data cache
Invalidate and disable instruction cache
Return OK
Get and print error code
Return ERROR

Figure 6.10: Flowchart For Application Running on Microblaze

# 7 TEST PLAN

In order to validate the configuration controller implementation, a simple design was developed to act as circuit under test. This design was kept as simple as possible, so each signal could be easily traced from its VHDL description until the implemented netlist. Furthermore, specific synthesis constraints were used to prevent optimizations that might interfere in the implementation hierarchy. This approach enabled back-end changes to be performed manually, targeting precise CUT elements in order to emulate the desired fault scenarios.

## 7.1 Circuit Under Test

The circuit under test is composed by four different modules:

- Digital Clock Manager (DCM)
- Linear Feedback Shift Register (LFSR)
- Parity Encoder
- Triple Modular Redundancy (TMR) Majority Voter

*Digital Clock Manager*

The digital clock manager is a IP generated from the Xilinx Coregen Library used to regulate the clock tree of the rest of the circuit. It was configured to provide an output clock that is 1/32 the value of the input source. Thus, for a standard 100 MHz board clock, the CUT would operate at 3.125 MHz. This mechanism was used to reduce the bandwidth of the CUT output signals, so they could be more easily monitored.

*Linear Feedback Shift Register*

The purpose of the LFSR is to serve as an on-board data pattern generator. The feedback taps were selected to provide the maximum sequence length using 2-bits. The sequence generated by this configuration is (3-1-2-0). The RTL schematic for the LFSR module is shown on figure 7.2

*Parity Encoder*

The parity encoder is a simple block responsible for calculating the associated parity bit (even) of the 2-bit input data word. In the CUT, three parity encoders are used in a triple modular redundancy scheme. The RTL schematic for the parity encoder module is shown on figure 7.3

*Majority Voter*

The majority voter is a standard NAND-based majority voter used in TMR designs. Its purpose is to vote the output of the three different parity encoders. The RTL schematic for the majority voter module is shown on figure 7.4.

*Top-Level*

The top-level CUT architecture is composed of three parity encoder instances being driven by the data coming from the LFSR. Their outputs are voted by the one majority voter module. The RTL schematic for top-level module is shown on figure 7.5.

*Design Constraints*

During the synthesis process, the Xilix tool inferred that the three instances of the original parity encoder module were identical. For this reason, the tool would remove two instantiations and connect all inputs of the majority voter to the output of the remaining one. In order to prevent this, a register had to be added inside each parity encoder.

*Synthesis Constraints*

In order to prevent the synthesis tool to interfere with the original design entities, and force the final implementation to follow the same design architecture, the following switches had to be modified:

Enable "Keep Hierarchy". This switch specifies whether or not the corresponding design unit should be preserved from being merged with the rest of the design.

Disabled "Resource Sharing". This switch specifies whether or not the corresponding design unit should share arithmetic operator resources.

Disabled "Register Duplication". This switch specifies whether or not the synthesis tool should replicate registers to control the unit fanout.

Disabled "Equivalent Register Removal". This switch specifies whether or not to set flip-flop optimization. Flip-flop optimization includes the removal of equivalent flip-flops and of flip-flops with constant inputs.

Disabled "Slice Packing". This switch specifies whether or not the synthesis tool should force components together in the same slice.

*Simulation*

Figure 7.1 shows the post-route simulation for the implemented CUT. Note that, once a register has been included in the parity encoder, its output is one cycle delayed from the data validity signal.

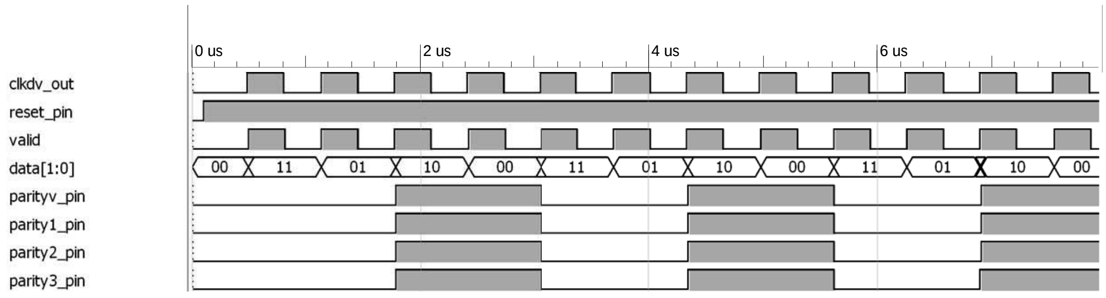


Figure 7.1: Simulation Result for the Circuit Under Test

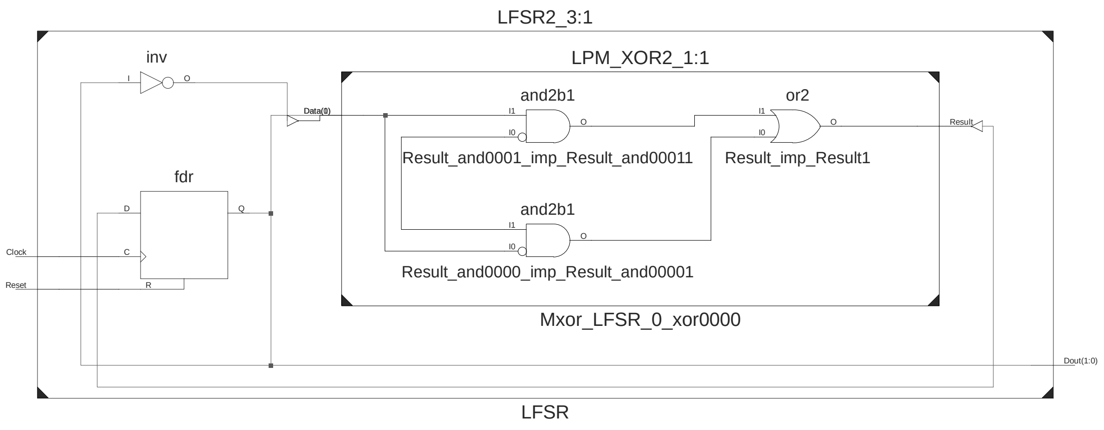


Figure 7.2: RTL Schematic for LFSR

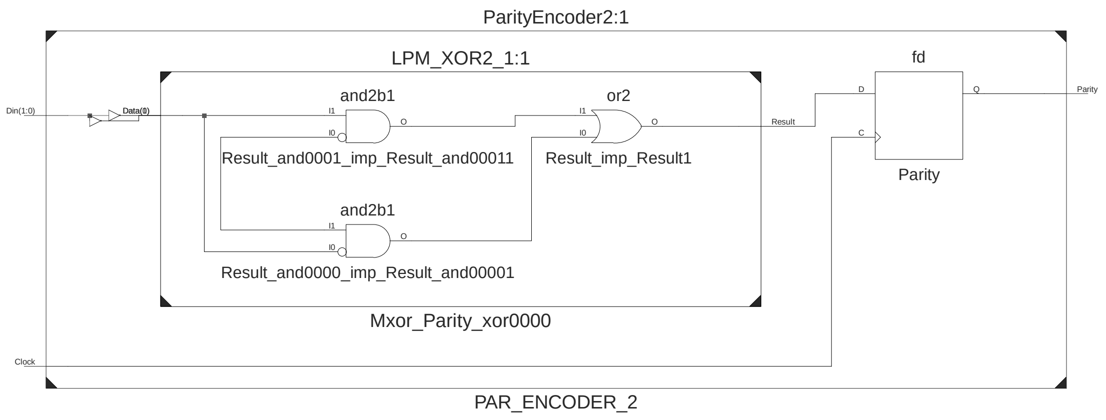


Figure 7.3: RTL Schematic for Parity Encoder

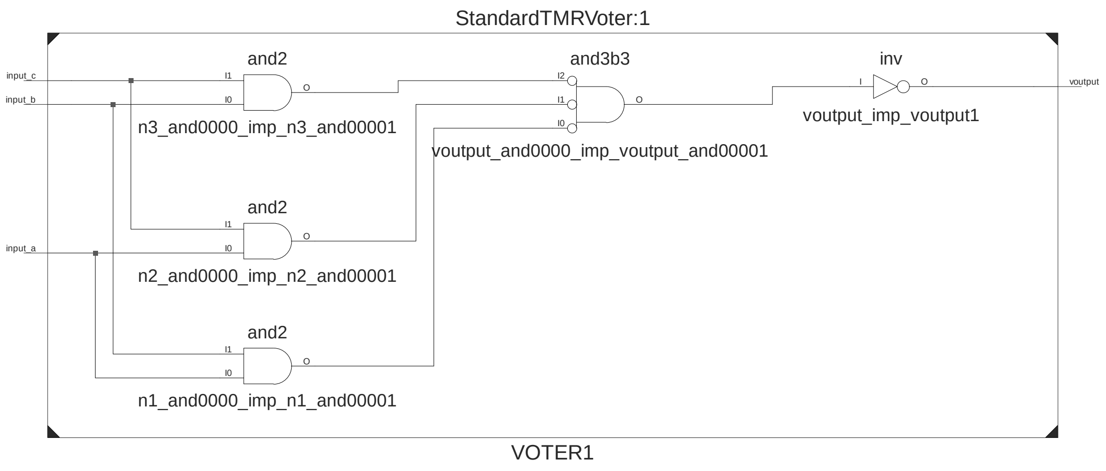

StandardTMRVoter:1
and2
and3b3
inv
input_c
input_b
I1
I0
O
I2
I
voutput
n3_and0000_imp_n3_and00001
voutput_and0000_imp_voutput_and00001
voutput_imp_voutput1
and2
input_a
n2_and0000_imp_n2_and00001
and2
n1_and0000_imp_n1_and00001
VOTER1

Figure 7.4: RTL Schematic for TMR Voter

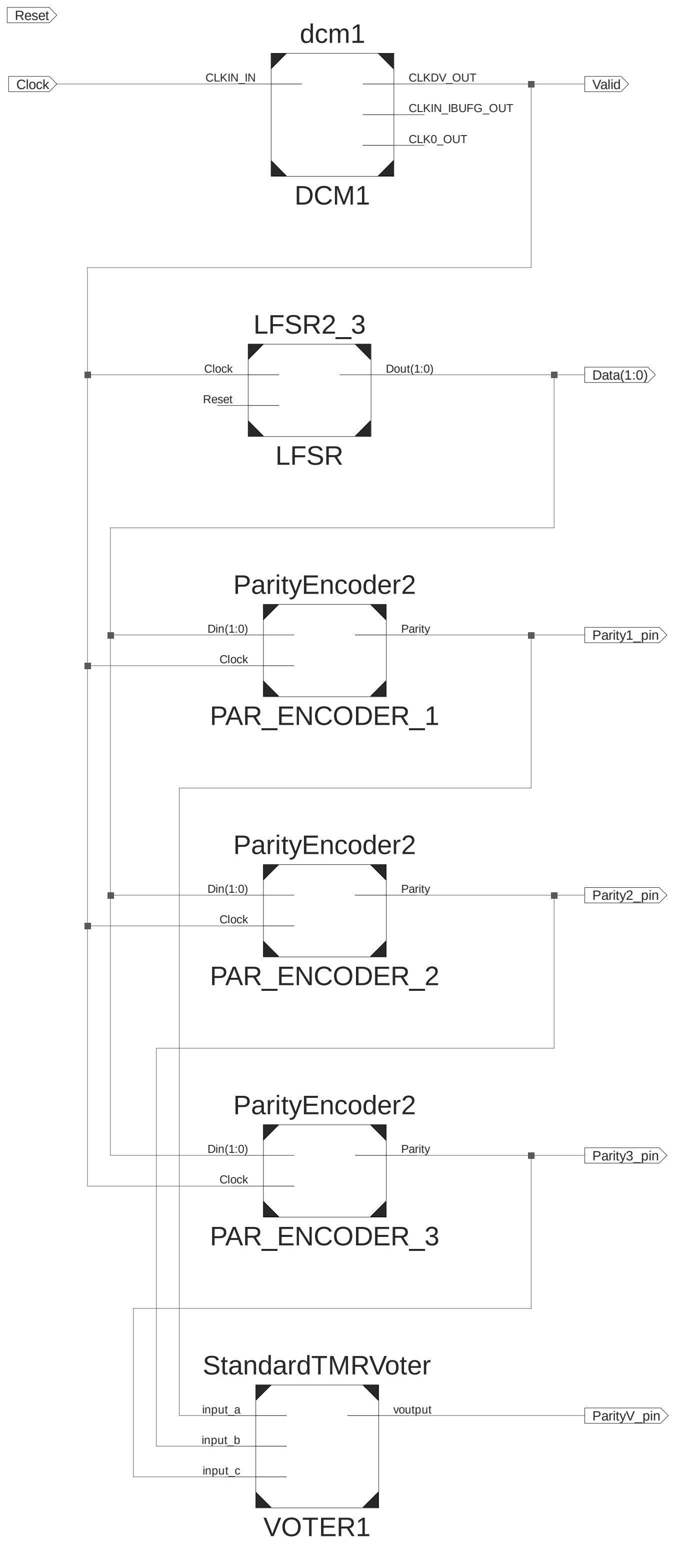


Figure 7.5: RTL Schematic for CUT Top-Level Design

## 7.2 Device Under Test

The circuit under test was synthesized and implemented on a Virtex-4 device (xc4vlx25-10sf363), hosted on a Memec V4LX25LC development board. Table 7.1 shows the IOB properties. Table 7.2 shows the device utilization summary.

| IOB Name | Direction | IO Standard | Drive Strength | Slew Rate | Resistor | LOC |
|---|---|---|---|---|---|---|
| Clock | INPUT | LVCMOS25 | - | - | None | A8 |
| Data<0> | OUTPUT | LVCMOS25 | 12mA | SLOW | None | U17 |
| Data<1> | OUTPUT | LVCMOS25 | 12mA | SLOW | None | V19 |
| Parity1_pin | OUTPUT | LVCMOS25 | 12mA | SLOW | None | U15 |
| Parity2_pin | OUTPUT | LVCMOS25 | 12mA | SLOW | None | R18 |
| Parity3_pin | OUTPUT | LVCMOS25 | 12mA | SLOW | None | H19 |
| ParityV_pin | OUTPUT | LVCMOS25 | 12mA | SLOW | None | T20 |
| Reset | INPUT | LVCMOS25 | - | - | PULLUP | B4 |
| Valid | OUTPUT | LVCMOS25 | 12mA | SLOW | None | T15 |

Table 7.1: DUT IOB Properties

| Logic Utilization | Used | Available | Utilization |
|---|---|---|---|
| Number of Slice Flip Flops | 5 | 21,504 | 1% |
| Number of 4 input LUTs | 6 | 21,504 | 1% |
| Number of occupied Slices | 7 | 10,752 | 1% |
| Number of bonded IOBs | 9 | 240 | 3% |
| Number of GCLKs | 2 | 32 | 6% |
| Number of DCM ADVs | 1 | 8 | 12% |

Table 7.2: DUT Utilization Summary

Figure 7.6 shows the JTAG chain on the V4LX25LC development board, in which the FPGA device can be directly accessed from the JTAG port. Since the configuration data does not have to be shifted through other devices (e.g PROMs), the time required for partial reconfiguration is not affected.

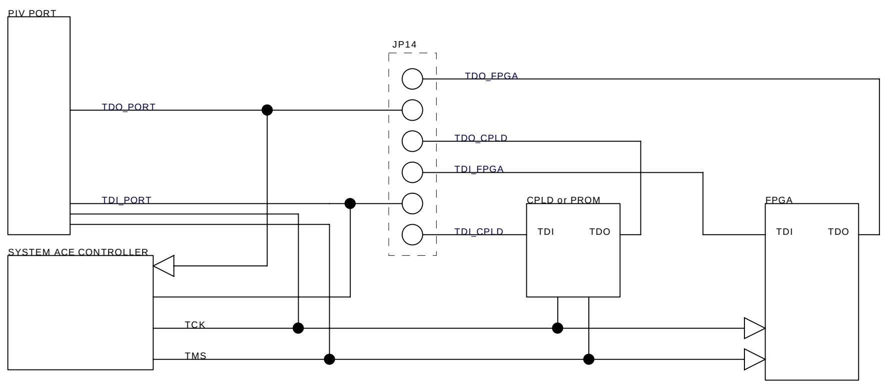


Figure 7.6: DUT JTAG Chain Diagram

## 7.3 Fault Model

In order to validate the proposed platform, and the correct operation of the configuration controller, a fault injection campaign had to be designed for the circuit under test. During this campaign, only faults affecting the function generators inside the parity encoder modules were considered. Even thought this scenario is oversimplified if compared to real world applications, it is enough to demonstrate the correctness of the fault injection process. Other than this, this model allows to prove two important properties of the fault injection mechanism:

1. CUT operation is not disrupted during the fault injection process.
2. Faults targeting one element do not modify the computation of independent elements.

Both these properties can be demonstrated by showing that faults affecting one parity encoder do not disrupt of modify the operation of the other two.

Another advantage of this model is that it can be used to demonstrate the effects of faults accumulating over time, as well as faults being logically masked. Consider the following case: a fault targeting parity encoder 1 is injected. This fault can be either latent, or active; depending on input data. After it, without removing the previous one, another fault targeting encoder 2 is injected. This fault, also, can be either latent or active. If both faults are latent, or only one is active, the output of the voter will be correct. However, if the two are active at the same time, the voted value will be incorrect.

## 7.4 Fault Generation

After the CUT implementation process, the parity encoder of figure 7.3 was mapped to specific FPGA resources, as shown on figure 7.7. Basically, the logic gates present at RTL level on block Mxor_Parity_xor0000 were implemented on a 4-input LUT, while the storage element was mapped to a edge-triggered D-type flip-flop. In this specific case, the schematic reports a lut2 component since only 2-bits are used to drive the associated logic function. If it were not for the synthesis constraints used to prevent resource sharing, this LUT could used to implement another 2-bit function of unrelated logic.

When inspected, the Mxor_Parity_xor0000_Result1 LUT reveals the following logical function:

$$O = ((\neg I0 \times I1) + (I0 \times \neg I1)) \tag{7.1}$$

This function can be represented by the truth table shown on 7.3, which is in accordance with the expected parity encoding function.

| I1 | I0 | O |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Table 7.3: Parity Encoder Truth Table

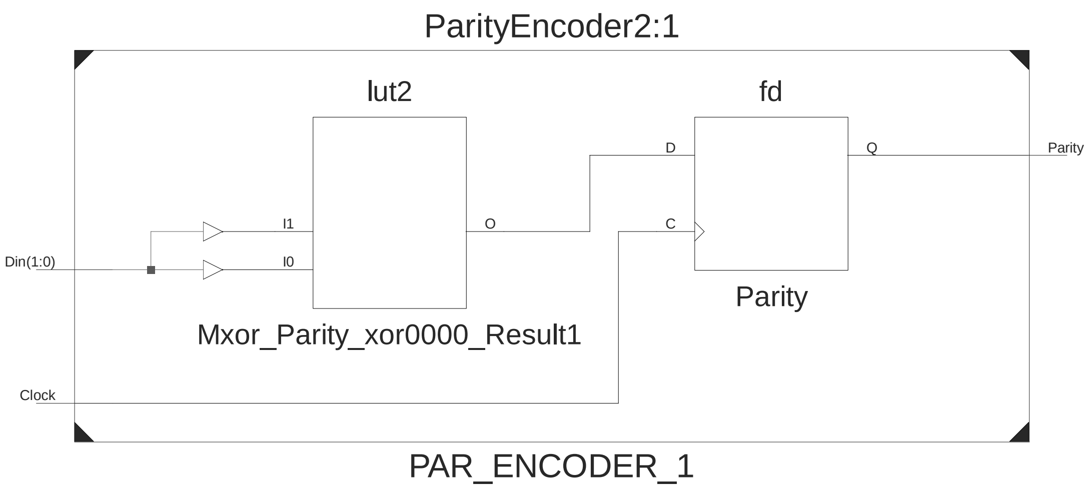


Figure 7.7: Technology Schematic for Parity Encoder

In order to emulate faults affecting the parity encoders, the associated LUTs can be modified to implement different logic functions. For this purpose, four different fault patterns were defined, each one affecting one line of the truth table at a time. The resulting fault patterns, named fp1 to fp4, are shown on table 7.4.

| I1 | I0 | Ofp1 | Ofp2 | Ofp3 | Ofp4 |
|---|---|---|---|---|---|
| 0 | 0 | **1** | 0 | 0 | 0 |
| 0 | 1 | 1 | **0** | 1 | 1 |
| 1 | 0 | 1 | 1 | **0** | 1 |
| 1 | 1 | 0 | 0 | 0 | **1** |

Table 7.4: Parity Encoder Truth Table with Faults

From this table, the associated logic function for each fault pattern can be derived:

$$O_{\mathrm{fp1}} = ((\neg I0 \times \neg I1) + (\neg I0 \times I1) + (I0 \times \neg I1)) \tag{7.2}$$

$$O_{\mathrm{fp2}} = (\neg I0 \times I1) \tag{7.3}$$

$$O_{\mathrm{fp3}} = (I0 \times \neg I1) \tag{7.4}$$

$$O_{\mathrm{fp4}} = ((\neg I0 \times I1) + (I0 \times \neg I1) + (I0 \times I1)) \tag{7.5}$$

### 7.4.1 Back-End Changes

In order to alter the original parity encoder truth table with the defined fault patterns, the CUT netlist had to be directed modified. The original netlist file was replicated 12 times, one for each fault pattern affecting a different parity encoder instance. Each file was manually updated to change the value of the associated resources. To perform this task, the Xilinx FPGA Editor tool was used. This tools allows each FPGA slice to be individually modified according to a series of equations describing the slice customization.

Due to the synthesis constraints, each instance of the parity encoder module was allocated to a different slice, and no other logic was packed together with them. Thus, all instances had the same slice customization equations. These set of equations determine the slice resources utilization and configuration. The following are the equations for the parity encoder module:

$$\mathrm{CLKINV : CLK} \tag{7.6}$$

$$\mathrm{DYMUX : Y} \tag{7.7}$$

$$\mathrm{FFY : \#FF} \tag{7.8}$$

$$\mathrm{FFY\_INIT\_ATTR : INIT0} \tag{7.9}$$

$$\mathrm{FFY\_SR\_ATTR : SRLOW} \tag{7.10}$$

$$\mathrm{G : \#LUT : D = (A1@A3)} \tag{7.11}$$

$$\mathrm{SYNC\_ATTR : ASYNC} \tag{7.12}$$

Analyzing the equation 7.11, two things can be noted: First, the G LUT is being used for the parity encoder logic. This is the upper slice LUT, as shown in 2.4. Second, if compared to the original LUT equation 7.1 obtained from the post-map model, this one is different.

What happened was a variable mapping, as show in relationship 7.13.

$$\mathrm{I0} \rightarrow \mathrm{A1}, \mathrm{I1} \rightarrow \mathrm{A3} \tag{7.13}$$

During the place and route process, different slice inputs were assigned to the LUT, causing the formula discrepancy. However, both equations 7.11 and 7.1 are logically equivalent, as shown in relationship 7.14. Note: the '@' symbol stands for the logical XOR.

$$((\neg \mathrm{I0} \times \mathrm{I1}) + (\mathrm{I0} \times \neg \mathrm{I1})) \equiv \mathrm{A1} \oplus \mathrm{A3} \tag{7.14}$$

Thus, the back-end changes required for the parity encoders are limited to updating the G LUT equation. The full process required to generate the modified netlists is show on figure 7.8.

### 7.4.2 Partial Bitstreams

Partial bitstreams were created from the difference between the original and the modified netlists. In total, 24 files were created. One for each fault pattern, in each parity encoder, to both for set and clear the fault. In order to create the bitstreams, the Xilinx Bitgen tool was used. Appendix A, listing A.7 shows the script used to create each file. Table 7.5 presents the bitstream sizes, both full and partial.

| File | Size (bytes) |
|---|---|
| Full DUT bitstream | 977.579 |
| Partial Bitstream (set) | 1.430 |
| Partial Bitstream (clear) | 897 |

Table 7.5: CUT Bitstreams Size

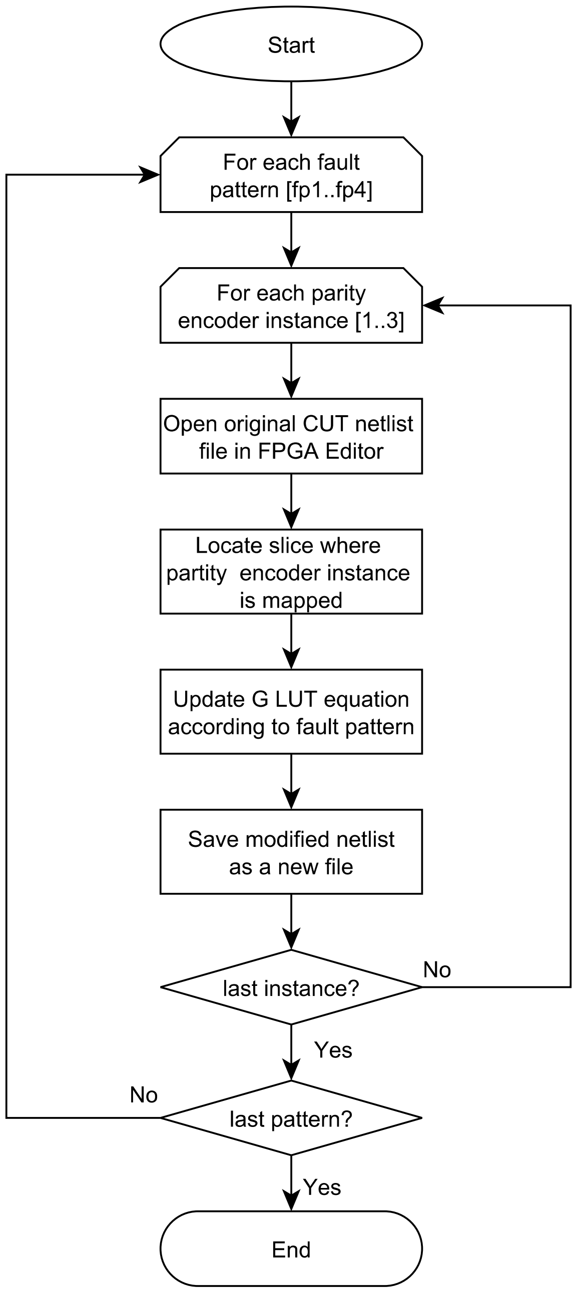

Start
For each fault pattern [fp1..fp4]
For each parity encoder instance [1..3]
Open original CUT netlist file in FPGA Editor
Locate slice where partity encoder instance is mapped
Update G LUT equation according to fault pattern
Save modified netlist as a new file
last instance?
No
Yes
last pattern?
No
Yes
End

Figure 7.8: Back-End Changes Process

### 7.4.3 ACE files

In order for the configuration controller to be able to parse the partial bitstreams, the configuration files had to be converted from the standard .bit format, to the ACE format. This operation requires two steps: one to convert from the .bit file to the .svf file, and another step to convert from the .svf file to the final .ace file.

The first step was performed using the Xilinx Impact software. A small script had to be written for this purpose, as shown on Appendix A, listing A.5. The second step was performed using the svf2ace utility that Xilinx provides for download in (Xilinx XAPP424), as shown in Appendix A, listing A.6.

## 7.5 Test Procedure

The following tests are required to validate the configuration controller prototype:

*General Operation*

The purpose of this test is to evaluate the overall operation of the configuration controller in order to verify that it operates according to the expected behavior. During this test it shall be verified that CUT operation is not disrupted during the fault injection process, and that faults targeting one element do not modify the computation of independent elements.

*Single Fault*

The purpose of this test is to verify that each fault produces the expected impact over the circuit under test. During this test, each one of the four different fault patterns must be injected at any parity encoder. It shall be verified that each fault produces the expected behavioral change in the CUT.

*Fault Accumulation*

The purpose of this test is to verify that multiple faults can be injected, and the effect of fault accumulation can be analyzed. During this test, two faults must be injected, targeting two different parity encoders. Faults should not be cleared in between injections. It shall be verified that each fault produces the expected behavioral change in the CUT.

*Fault Injection Times*

The purpose of this test is to measure the configuration controller performance. All 24 files that were created should be injected sequentially. The times required for each fault injection shall be annotated, as well as the time required by the configuration controller to start a new injection.

# 8 TEST RESULTS

Experimental tests were conducted to evaluate the configuration controller implementation presented in chapter 6. These tests were conducted according to the test plan developed throughout chapter 7, and provide evidence to support the feasibility of the proposed fault injection architecture.

## 8.1 Test Environment

The tests were performed following the proposed fault injection architecture shown in figure 5.1. The configuration controller ACE player port was connected to the DUT JTAG header; and its UART interface, to the host computer via USB. To replace the DAQ module, a 8-bit USB logic analyzer was used to monitor both CUT outputs, and fault event signal. Technical specifications of this product can be found in Appendix B. Figure 8.1 shows the test environment.

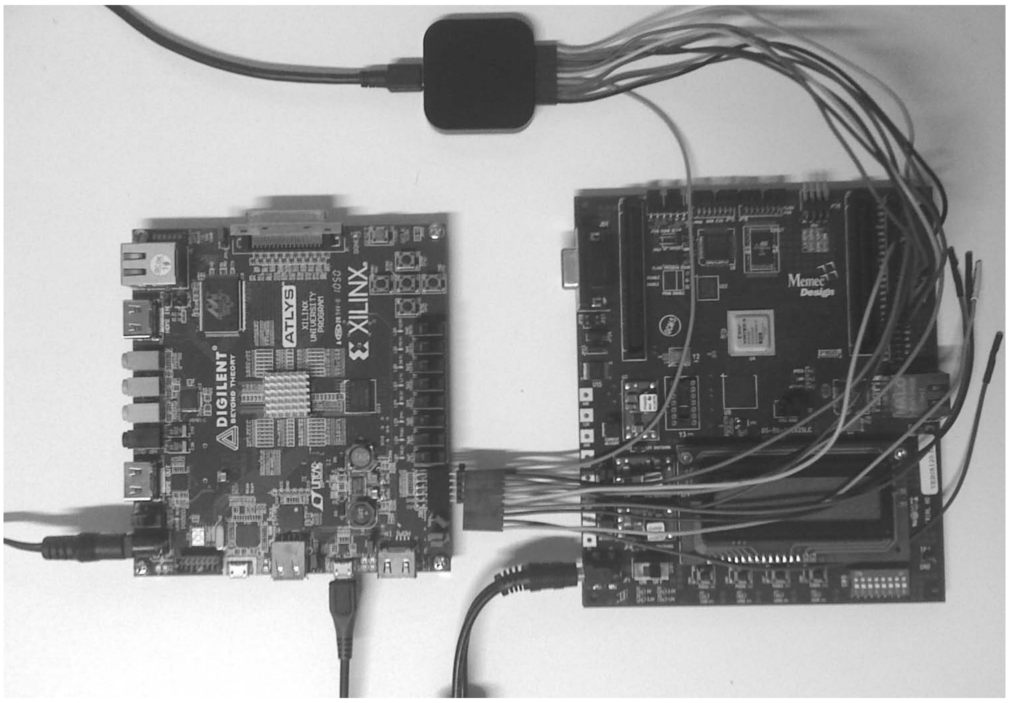


Figure 8.1: Test Environment

## 8.2 General Operation

Figure 8.2 shows the CUT operating under normal conditions. All three parity encoders have the same output, and the voter correctly generates the right value. The fault event signal is low, meaning that no fault is being injected at that time frame.

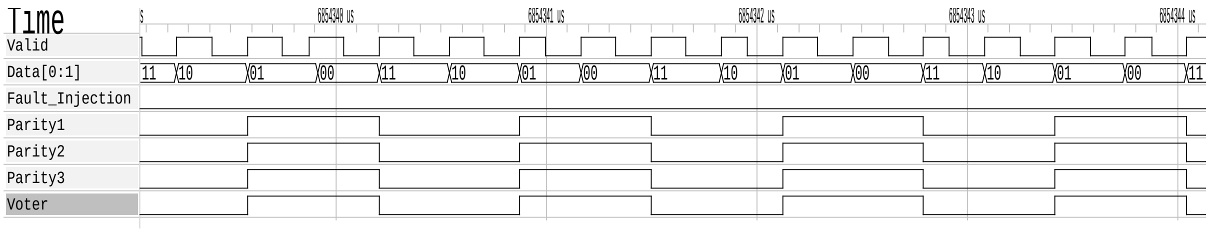


Figure 8.2: CUT Operation Under Normal Conditions

Figure 8.3 shows the CUT operating under the reconfiguration process. The fault event signal is high, meaning that a fault is being injected. Configuration data is still being transfered through the JTAG interface, and the device configuration memory was not yet modified. During this time, all three parity encoders have the same output, and the voter correctly generates the right value.

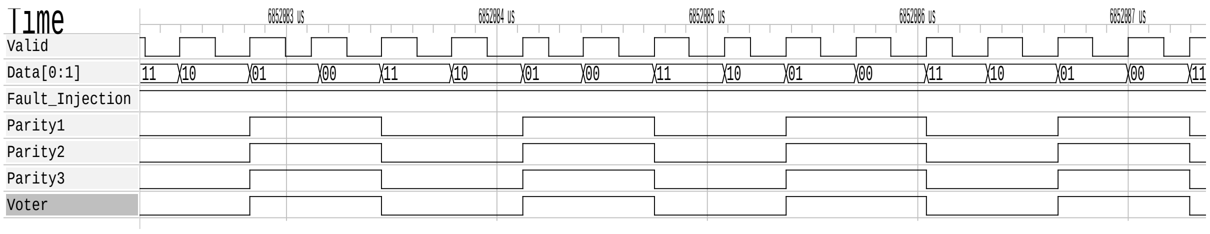


Figure 8.3: CUT Operation Under Partial Reconfiguration

Figure 8.4 shows the first instant in which a fault affecting partity encoder #2 is manifested. At this time, the fault event signal is still high, meaning that the configuration controller is still active. Although JTAG operations are still being sent to the DUT, the device configuration memory has already been modified and updated.

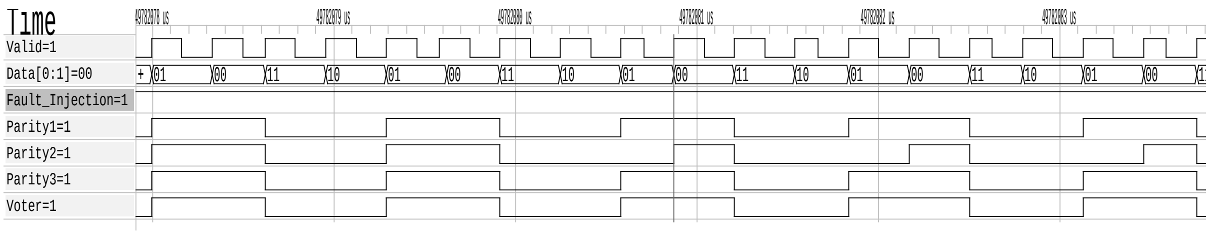


Figure 8.4: CUT Operation During DUT Internal Memory Update

Even while the DUT updates its internal memory, no disruption on the CUT was observed. Also, it was seen that faults affecting one element do not modify the computation of unrelated modules. These results confirm the partial reconfiguration principles, and validate its use for fault injection applications. Furthermore, they validate the correctness of the configuration controller operation.

## 8.3 Single Fault

Figure 8.5 shows the effect of a partial bitstream generated by modifying the original PAR_ENCODER_1 LUT content according to equation 7.2.

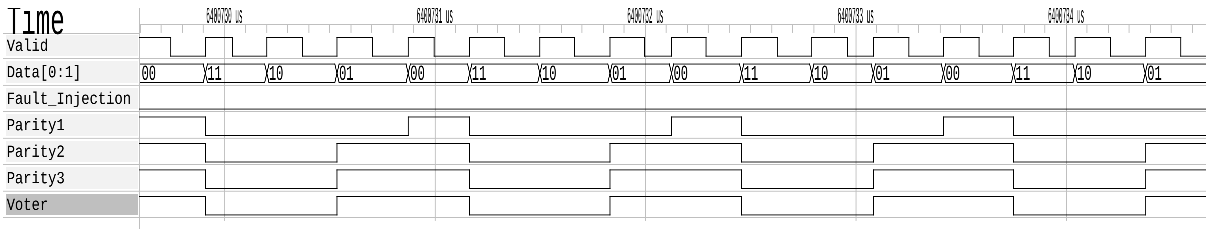


Figure 8.5: Fault pattern #1 affecting Parity Encoder 1

Figures 8.6, 8.7 and 8.8 show, respectively, the implications of PAR_ENCODER_1 LUT modifications due to equations 7.3, 7.4 and 7.5.

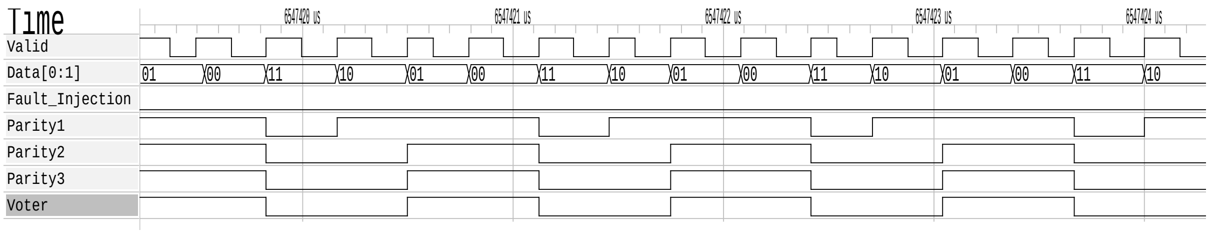


Figure 8.6: Fault pattern #2 affecting Parity Encoder 1

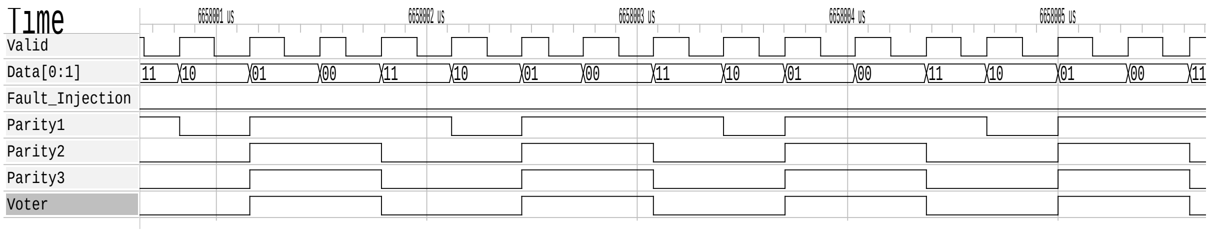


Figure 8.7: Fault pattern #3 affecting Parity Encoder 1

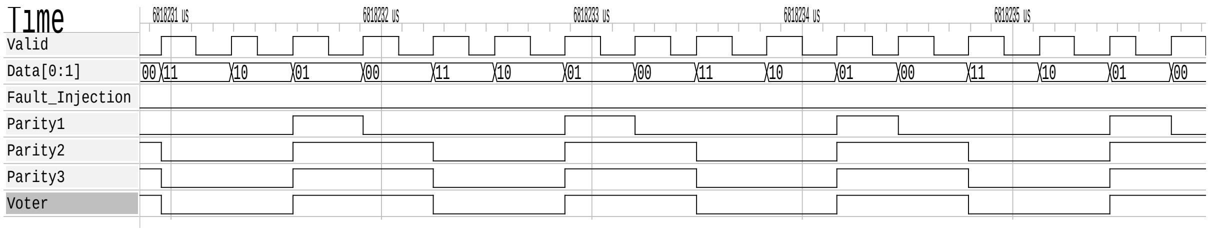


Figure 8.8: Fault pattern #4 affecting Parity Encoder 1

The observed patterns in the PAR_ENCODER_1 are in accordance with the expected results. This fact validates the technique used to perform back-end changes directly on the CUT netlist. More importantly, it validates the proposed approach of generating and injecting faults targeting specific FPGA resources.

## 8.4 Fault Accumulation

Figure 8.9 shows the CUT output signals when operating under the presence of two latent faults. Both PAR_ENCODER_2 and PAR_ENCODER_3 modules have a fault causing their computations to be incorrect, but due to logical masking, one of them always has the right output. Since PAR_ENCODER_1 is unaffected, the majority voter can always rule for the correct value.

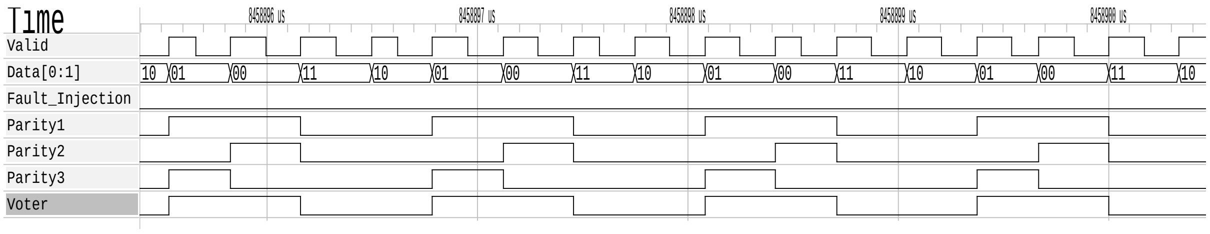


Figure 8.9: CUT Operating Under Two Latent Faults

Figure 8.10 shows the CUT output signals when operating under the presence of two active faults. Both PAR_ENCODER_2 and PAR_ENCODER_3 modules have a fault that causes their computations to be incorrect at the same time frame. In this case there is no logical masking, and the majority voter is forced into ruling for the wrong value.

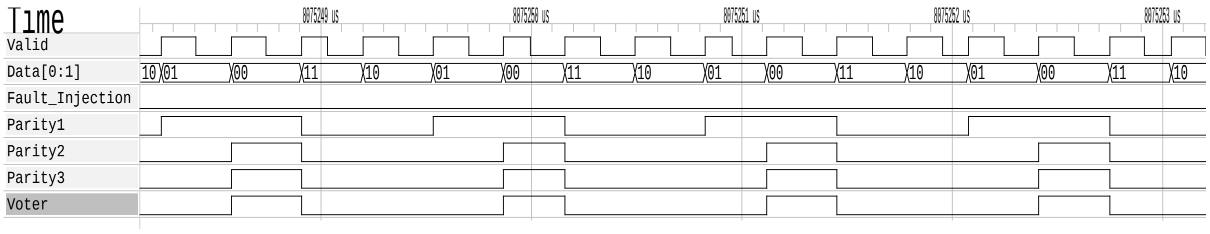


Figure 8.10: CUT Operating Under Two Active Faults

Both this cases illustrate the possibility to inject more than a single fault at any given period of time, in different CUT locations. This validates the platform ability to evaluate multiple faults accumulating over time.

## 8.5 Fault Injection Times

The time required for a fault injection was measured based on the DUT reconfiguration time; i.e., the time required by the configuration controller to transfer a partial bitstream through the JTAG interface. The measurements were performed by calculating the width of the fault event signal over a period of 24 different fault injections. The reported times are shown on table 8.1.

The time required by the configuration controller to start a new fault injection was measured based on the average time between subsequent fault injections. The measurements were performed by calculating the time difference between two adjacent rising edges of the fault event signal over a period of 24 different fault injections. The reported times are shown on table 8.2.

In order to estimate the maximum rate of fault injections per unit of time, the total fault injection time was assumed to be given by equation 8.1.

| Instance | Time (ms) | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Fault Pattern 1 | | Fault Pattern 2 | | Fault Pattern 3 | | Fault Pattern 4 | |
| | Set | Clear | Set | Clear | Set | Clear | Set | Clear |
| ENCODER_1 | 1,3666 | 0,9097 | 1,3667 | 0,9095 | 1,3670 | 0,9097 | 1,3668 | 0,9097 |
| ENCODER_2 | 1,3667 | 0,9099 | 1,3675 | 0,9099 | 1,3671 | 0,9100 | 1,3674 | 0,9097 |
| ENCODER_3 | 1,3667 | 0,9095 | 1,3667 | 0,9096 | 1,3670 | 0,9097 | 1,3668 | 0,9096 |
| **Average** | 1,3667 | 0,9097 | 1,3670 | 0,9097 | 1,3670 | 0,9098 | 1,3670 | 0,9097 |

Table 8.1: Time Required For Fault Injection

| Fault # | Latency (us) | Fault # | Latency (us) | Fault # | Latency (us) |
|---|---|---|---|---|---|
| 01 | 1,5000 | 09 | 1,4583 | 11 | 1,2917 |
| 02 | 1,5417 | 10 | 1,3333 | 12 | 1,3750 |
| 03 | 1,4583 | 11 | 1,4882 | 13 | 1,3333 |
| 04 | 1,3750 | 12 | 1,5000 | 14 | 1,2917 |
| 05 | 1,5000 | 13 | 1,3750 | 15 | 1,4167 |
| 06 | 1,4167 | 14 | 1,2917 | 16 | 1,3333 |
| 07 | 1,5417 | 15 | 1,4583 | 17 | 1,4583 |
| 08 | 1,3333 | 16 | 1,3333 | 18 | 1,4167 |

Table 8.2: Minimum Time Between Consecutive Fault Injections

$$T_{\text{total}} = T_{\text{Injection}} + T_{\text{Recovery}} + (2 \times T_{\text{Latency}}) \tag{8.1}$$

Where T_Injection is the time required for a fault injection; T_Latency is the time required to restore the system from that fault; and T_Between the time between two consecutive fault injections.

This way, equation 8.1 resolves to equation 8.2:

$$T_{\text{total}} \approx 1,3667ms + 0,9097ms + (2 \times 1,4167us) \approx 2,28ms \tag{8.2}$$

Which leads to a rate of injection given by equation 8.3:

$$Rate_{\text{injection}} = \frac{1}{T_{\text{total}}} = \frac{1}{0,00228s} \approx \frac{438faults}{second} \tag{8.3}$$

Of course, this is the theoretical maximum value obtained from the experimental measurements conducted. This rate does not take into consideration the time required for the fault characterization. In other words, the time necessary for the fault to be present on the CUT before being marked as active or latent. This time is directly involved with the size of the test vector set and CUT sampling rate.

# 9 CONCLUSION

Throughout this work, many different aspects concerning fault injection and dynamic partial reconfiguration were explored. First, an overall explanation of the architectural support provided by Xilinx FPGAs was presented, followed by a comprehensive review of the latest publications on the subject. After, a new fault injection platform was proposed in order to overcome issues related to modularity, intrusiveness and cost. This platform focused on a loosely coupled architecture and on a standard industry interface to provide maximum compatibility with different systems, while eliminating the need for further modifications on the circuit under test.

In order to evaluate the feasibility of the proposed solution, a prototype was developed and implemented on a commercial FPGA development board. Although some of the planned functionalities were not completely integrated for this scope, the resulting work contained all the required modules necessary for performing a fully functional fault injection campaign. Furthermore, a complete test case was designed to help validate the implementation and provide numbers to be used as performance indicators.

Experimental results conducted on the prototype showed that the platform responded as expected to all test case stimuli, thus validating the proposed architecture and fault injection methodology.

All goals set in the beginning of this work have been achieved, and for this reason, the author considers this to be a successful project. Its result contributes to the general understanding of fault injection architectures, and help future engineers progress towards more efficient solutions.

Nonetheless, there are several improvements that can be performed in order to extend the functionality of the proposed platform. Future works might focus on the following items that were not covered in this work:

- Implementing the Fault Manager Application, so partial bitstreams can be generated automatically based on a fault model defined by the user.
- Implementing the Data Interface Software, so fault impact analysis can be performed automatically in accordance to results achieved from simulation.
- Implementing the scheduler manager for the configuration controller embedded software, so the fault injection campaign can be customized by a application on the desktop computer.
- Implementing the Ethernet communication library for the configuration controller embedded software, to allow configuration data and scheduler information to be downloaded from a remote FTP server.

# REFERENCES

[A. Johnston 2000] A. H. Johnston. **Scaling and technology issues for soft error rates**. In 4th Annual Research Conf. on Reliability, 2000.

[A. Parreira et al 2004] A. Parreira, J. P. Teixeira, M. B. Santos, **Built-in self-test preparation in FPGAs**, In Proc. Of the 7th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems, pp. 83-90, Apr 2004.

[C. Lopez-Ongil et al 2007] Lopez-Ongil, C.; Entrena, L.; Garcia-Valderas, M.; Portela, M.; Aguirre, M.A.; Tombs, J.; Baena, V.; Munoz, F., **A Unified Environment for Fault Injection at Any Design Level Based on Emulation**, Nuclear Science, IEEE Transactions on , vol.54, no.4, pp.946-950, Aug. 2007

[D. Alexandrescu et al 2002] Alexandrescu, D., Anghel, L., and Nicolaidis, M. 2002. **New Methods for Evaluating the Impact of Single Event Transients in VDSM ICs**. In Proceedings of the 17th IEEE international Symposium on Defect and Fault-Tolerance in VLSI Systems (November 06 - 08, 2002). DFT. IEEE Computer Society, Washington, DC, 99-107.

[E. Normand 1996] Normand, E. **Single event upset at ground level.** IEEE Transactions on Nuclear Science, New York, v.43, n.6, p. 2742–2750, Dec. 1996.

[F. Kastensmidt et al 2004] Kastensmidt, F. L., Neuberger, G., Carro, L., and Reis, R.**Designing and testing fault-tolerant techniques for SRAM-based FPGAs**. In Proceedings of the 1st Conference on Computing Frontiers (Ischia, Italy, April 14 - 16, 2004). CF '04. ACM, New York, NY, 419-432.

[J. Karlsson et al 1991] Karlsson, J., Gunneflo, U., Lidén, P., and Torin, J. **Two Fault Injection Techniques for Test of Fault Handling Mechanisms**. In Proceedings of the IEEE international Test Conference on Test: Faster, Better, Sooner (October 26 - 30, 1991). IEEE Computer Society, Washington, DC, 140-149.

[J. Ritter 1990] Ritter, J.C.**Radiation Effects in Space Systems**, Naval Research Reviews, pp. 25-37, 1990.

[J. Tombs et al 2004] J. N. Tombs, F. Munoz, V. Baena-Lecuyer, A. Torralba, L.G. Franquelo, A. Fernández-Leon, F. Tortosa-Lopez, D. Gutiérrez González, **A Hardware Approach for Seu Immunity Verification Using Xilinx Fpga's**, Proc. 19th Conference on Design of Circuits and Integrated Systems, DCIS 2004. Bordeaux, France. 2004. pp. 479-484.

[L. Kafka 2008] Kafka, L., **Analysis of Applicability of Partial Runtime Reconfiguration in Fault Emulator in Xilinx FPGAs**, Design and Diagnostics of Electronic Circuits and Systems, 2008. DDECS 2008. 11th IEEE Workshop on , vol., no., pp.1-4, 16-18 April 2008

[L. Sterpone et al 2007 ] Sterpone, L.; Violante, M., "**A New Partial Reconfiguration-Based Fault-Injection System to Evaluate SEU Effects in SRAM-Based FPGAs**, Nuclear Science, IEEE Transactions on , vol.54, no.4, pp.965-970, Aug. 2007

[M. Alderighi et al 2003] Alderighi, M., Casini, F., D'Angelo, S., Mancini, M., Marmo, A., Pastore, S., and Sechi, G. R. 2003. **A Tool for Injecting SEU-Like Faults into the Configuration Control Mechanism of Xilinx Virtex FPGAs**. In Proceedings of the 18th IEEE international Symposium on Defect and Fault Tolerance in VLSI Systems (November 03 - 05, 2003). DFT. IEEE Computer Society, Washington, DC, 71.

[M. Alderighi et al 2007] Alderighi, M., Casini, F., D'Angelo, S., Pastore, S., Sechi, G. R., and Weigand, R. 2007. **Evaluation of Single Event Upset Mitigation Schemes for SRAM based FPGAs using the FLIPPER Fault Injection Platform**. In Proceedings of the 22nd IEEE international Symposium on Defect and Fault-Tolerance in VLSI Systems (September 26 - 28, 2007). DFT. IEEE Computer Society, Washington, DC, 105-113.

[M. Hsueh et al 1997 ] Hsueh, M., Tsai, T. K., and Iyer, R. K. 1997. **Fault Injection Techniques and Tools**. Computer 30, 4 (Apr. 1997), 75-82.

[M. Hubner et al 2004] Huebner, M., Becker, T., and Becker, J. 2004. **Real-time LUT-based network topologies for dynamic and partial FPGA self-reconfiguration**. In Proceedings of the 17th Symposium on integrated Circuits and System Design (Pernambuco, Brazil, September 07 - 11, 2004). SBCCI '04. ACM, New York, NY, 28-32.

[M. Shokrolah-Shirazi et al 2008] Shokrolah-Shirazi, M. and Miremadi, S. G. **FPGA-Based Fault Injection into Synthesizable Verilog HDL Models**. In Proceedings of the 2008 Second international Conference on Secure System integration and Reliability Improvement - Volume 00 (July 14 - 17, 2008). SSIRI. IEEE Computer Society, Washington, DC, 143-149.

[N. Battezzati et all 2008] Battezzati, N.; Sterpone, L.; Violante, M., **A new low-cost non intrusive platform for injecting soft errors in SRAM-based FPGAs**, Industrial Electronics, 2008. ISIE 2008. IEEE International Symposium on , vol., no., pp.2282-2287, June 30 2008-July 2 2008

[P. Folkesson et al 1998] P. Folkesson, S. Sevensson, and J. Karlsson, **A Comparsion of Simulation Based and Scan Chain Implemented Fault Injection**, Proc. of the Annual International Symposium on Fault-Tolerant Computing, Jun. 1998, pp. 284-293.

[P. Kenterlis et all 2006 ] Kenterlis, P., Kranitis, N., Paschalis, A., Gizopoulos, D., and Psarakis, M. 2006. **A Low-Cost SEU Fault Emulation Platform for SRAM-Based FPGAs**. In Proceedings of the 12th IEEE international Symposium on on-Line Testing (July 10 - 12, 2006). IOLTS. IEEE Computer Society, Washington, DC, 235-241.

[R. Leveugle 1999] R. Leveugle, **Towards modeling for dependability of complex integrated circuits**, in 5th IEEE International On- Line Testing workshop, July 1999, pp. 194-198

[S. Guccione et al 1999] Guccione, S.,Levi, D.,Sundararajan, P. **JBits: A Java-based interface for reconfigurable computing** 1999. In Proceedings of the Second Annual Military and Aerospace Applications. MAPLD, 1999

[V. Pouget et al 2008] Pouget, V., Douin, A., Foucard, G., Peronnard, P., Lewis, D., Fouillat, P., and Velazco, R. **Dynamic Testing of an SRAM-Based FPGA by Time-Resolved Laser Fault Injection**. In Proceedings of the 2008 14th IEEE international on-Line Testing Symposium - Volume 00 (July 07 - 09, 2008). IOLTS. IEEE Computer Society, Washington, DC, 295-301.

[V. Pouget et al 2008] Heng Tan , Ronald F. Demara , Abdel Ejnioui , Jason D. Sattler **Complexity and Performance Evaluation of Two Partial Reconfiguration Interfaces on FPGAs: a Case Study**. In Reconfigurable Architectures Workshop (RAW), Greek, 2006.

[Xilinx XAPP058] Xilinx, Inc. **XAPP058: Xilinx In-System Programming Using an Embedded Microcontroller**. Xilinx Application Note XAPP058 v4.1 March 6, 2009.

[Xilinx XAPP290] Xilinx, Inc. **XAPP290: Difference-Based Partial Reconfiguration**. Xilinx Application Note XAPP290 v2.0 December 3, 2007.

[Xilinx XAPP424] Xilinx, Inc. **XAPP424: Embedded JTAG ACE Player**. Xilinx Application Note XAPP424 v1.0.2 April 7, 2008.

[Xilinx XAPP503] Xilinx, Inc. **XAPP503: SVF and XSVF File Formats for Xilinx Devices**. Xilinx Application Note XAPP503 v2.1 August 17, 2009.

[Xilinx XAPP951] Xilinx, Inc. **XAPP951: Configuring Xilinx FPGAs with SPI Serial Flash**. Xilinx Application Note XAPP951 v1.3 September 23, 2010.

[Xilinx UG071] Xilinx, Inc. **UG071: Virtex-4 FPGA Configuration User Guide** v1.11 June 9, 2009.

[Xilinx UG070] Xilinx, Inc. **UG070: Xilinx Virtex-4 FPGA User Guide** v2.6, December 1, 2008.

[Xilinx DS112] Xilinx, Inc. **DS112: Virtex-4 Family Overview** v3.1, August 30, 2010.

[Xilinx UG702] Xilinx, Inc. **UG702: Partial Reconfiguration User Guide** v14.1 April 24, 2012.

[Xilinx WP374] D. Dye **WP374: Partial Reconfiguration of Xilinx FPGAs Using ISE Design Suite**. Xilinx White Paper WP374 v1.1 July 6, 2011.

[Xilinx UG081] Xilinx, Inc. **UG081: MicroBlaze Processor Reference Guide**. v14.1 April 24, 2012.

[502-178] Digilent Inc. **Attllys Board Refference Manual**. Digilent Inc. Reference Manual Rev C February 28, 2011.

[J. Edwards 2006] Digilent Inc. **No room for Second Place: Xilinx and Altera slug it out for supremacy in the changing PLD market**. EDN, retrieved May 11, 2012

[DSD-0000332] Digilent Inc. **Atlys board support files for EDK BSB wizard**. retrieved March 16, 2012

# APPENDIX A SCRIPTS

Listing A.1: Script For Generating Microblaze Objects

```
mb-objcopy -I binary -O elf32-microblaze file_name.ace file_name.o
```

Listing A.2: Script For Dumping Microblaze Objects

```
mb-objdump -x file_name.o >> file_name.dmp
```

Listing A.3: Microblaze Object Dump

```

system_part_parityencoder_2_0100_clear.o:     file format elf32-big
system_part_parityencoder_2_0100_clear.o
architecture: UNKNOWN!, flags 0x00000010:
HAS_SYMS
start address 0x00000000

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .data         00000b8c  00000000  00000000  00000034  2**0
                  CONTENTS, ALLOC, LOAD, DATA
SYMBOL TABLE:
00000000 l    d  .data  00000000 .data
00000000 g       .data  00000000 _binary_system_part_parityencoder_2_0100_clear_ace_start
00000b8c g       .data  00000000 _binary_system_part_parityencoder_2_0100_clear_ace_end
00000b8c g       *ABS*  00000000 _binary_system_part_parityencoder_2_0100_clear_ace_size
```

Listing A.4: Script For Generating a SREC from a ELF file

```
mb-objcopy -O srec application.elf application.srec
```

Listing A.5: Script For Generating a .SVF file from a .BIT file

```
impact -batch
setMode -bs
addDevice -p 1 -file file_name.bit
setCable -port svf -file file_name.svf
program -p 1
closeCable
quit
```

Listing A.6: Script For Generating a .ACE file from a .SVF file

```
svf2ace.exe -wtck -i file_name.svf -o file_name.ace
```

Listing A.7: Bitgen Options Used to Create Partial Bitstreams

```
bitgen -w -g ActiveReconfig:Yes -g Persist:yes -r
    original_bitstream.bit modified_netlist.ncd
    modified_netlist_partial_bitstream_fault_set.bit
bitgen -w -g ActiveReconfig:Yes -g Persist:yes
    modified_netlist.ncd
    modified_netlist_full_bitstream_fault_set.bit
bitgen -w -g ActiveReconfig:Yes -g Persist:yes -r
    modified_netlist_full_bitstream.bit original_ntelist.ncd
     modified_netlist_partial_bitstream_fault_clear.bit
```

# APPENDIX B LOGIC ANALYZER SPECIFICATIONS

*Input Voltages and Thresholds*

- Input voltage range: -0.5V to 5.25V
- Input Low Voltage: -0.5V to 0.8V
- Input High Voltage: 2.0V to 5.25V
- Works with 5V, 3.3V, 2.5V, 2.0V systems. May work with 1.8V but not recommended.
- ESD protected per CE requirements
- Over-voltage protection to +/- 15V. Not meant for continuous operation outside -0.5V to 5.25V.
- Input Impedance: 1Mohm || 10pF (typical, approximate)
- Crystal: +/-20ppm, 24MHz
- Error/Accuracy: pulse-width measurement: +/- 42ns (at 24MHz).

*Sample Rate & Depth*

- 24MHz. 16MHz, 12MHz, 8MHz, 4MHz, 2MHz, 1MHz, 500KHz, 250KHz, 200KHz, 100KHz, 50KHz, 25KHz;
- 10B samples. Absolute max depends on data compressibility, available RAM and operating system. 10B samples assumes reasonably high compressibility.

*Connectors*

- 1x8 male IDE .1 in pitch (aperture size: .110 in x 1.840 in; .030in radiused corners)
- USB Mini-B