Universität Stuttgart

Institute of Computer Engineering and Computer Architecture
Prof. Dr. rer. nat. habil. Hans-Joachim Wunderlich
Pfaffenwaldring 47, 70569 Stuttgart

Master Thesis Nr. 3439

Online Self-Test Wrapper for Runtime-Reconfigurable Systems

Jiling WANG

MSC THESIS

in partial fulfillment of the requirements
for the degree of **Master of Science**

Supervisors :	Dipl.-Inf. Michael KOCHTE
	Dipl.-Inf. Michael IMHOF
	Dipl.-Inform. Claus BRAUN
Examiner :	Prof. Dr. rer. nat. habil. Hans-Joachim WUNDERLICH
Start Date :	December 03, 2012
Submission Date :	June 04, 2013
CR Classification :	B.5.2, B.6.1, B.8.1, B.8.2, C.4
Study Program :	M.Sc. Information Technology (Embedded Systems)

Abstract

Reconfigurable Systems-on-a-Chip (SoC) architectures consist of microprocessors and Field Programmable Gate Arrays (FPGAs). In order to implement runtime reconfigurable systems, these SoC devices combine the ease of programmability and the flexibility that FPGAs provide. One representative of these is the new Xilinx Zynq-7000 Extensible Processing Platform (EPP), which integrates a dual-core ARM Cortex-A9 based Processing System (PS) and Programmable Logic (PL) in a single device. After power on, the PS is booted and the PL can subsequently be configured and reconfigured by the PS.

Recent FPGA technologies incorporate the dynamic Partial Reconfiguration (PR) feature. PR allows new functionality to be programmed online into specific regions of the FPGA while the performance and functionality of the remaining logic is preserved. This on-the-fly reconfiguration characteristic enables designers to time-multiplex portions of hardware dynamically, load functions into the FPGA on an as-needed basis. The configuration access port on the FPGA can be used to load the configuration data from memory to the reconfigurable block, which enables the user to reconfigure the FPGA online and test runtime systems.

Manufactured in the advanced 28 nm technologies, the modern generations of FPGAs are increasingly prone to latent defects and aging-related failure mechanisms. To detect faults contained in the reconfigurable gate arrays, dedicated on and off-line test methods can be employed to test the device in the field. Adaptive systems require that the fault is detected and localized, so that the faulty logic unit will not be used in future reconfiguration steps.

This thesis presents the development and evaluation of a self-test wrapper for the reconfigurable parts in such hybrid SoCs. It comprises the implementation of Test Configurations (TCs) of reconfigurable components as well as the generation and application of appropriate test stimuli and response analysis. The self-test wrapper is successfully implemented and is fully compatible with the AMBA protocols.

The TC implementation is based on an existing Java framework for Xilinx Virtex-5 FPGA, and extended to the Zynq-7000 EPP family. These TCs are successfully redesigned to have a full logic coverage of FPGA structures. Furthermore, the array-based testing method is adopted and the tests can be applied to any part of the reconfigurable fabric.

A complete software project has been developed and built to allow the reconfiguration process to be triggered by the ARM microprocessor. Functional test of the reconfigurable architecture, online self-test execution and retrieval of results are under the control of the embedded processor. Implementation results and analysis demonstrate that TCs are successfully synthesized and can be dynamically reconfigured into the area under test, and subsequent tests can be performed accordingly.

Contents

List of Figures

List of Tables

List of Abbreviations

AMBA	Advanced Microcontroller Bus Architecture
AXI	Advanced eXtensible Interface
BIF	Boot Image Format
BIST	Built-In Self Test
BSB	Base System Builder
BSP	Board Support Package
CFM	Cell Fault Model
CIP	Create or Import Custom Peripheral
CLB	Configuration Logic Block
DevC	Device Configuration
DMA	Direct Memory Access
DRC	Design Rule Check
EDK	Embedded Development Kit
ELF	Executable and Linkable Format
EPP	Extensible Processing Platform
FF	Flip-Flop
FPGA	Field Programmable Gate Array
FSBL	First Stage Bootloader
HWICAP	Hardware Internal Configuration Access Port
ICAP	Internal Configuration Access Port
IPIC_IF	Intellectual Property Interconnect Interface
LUT	Lookup Table
MPD	Microprocessor Peripheral Definition
NCD	Netlist Circuit Description
OCM	On-Chip Memory

ORA	Output Response Analyzer
PCAP	Processor Configuration Access Port
PIP	Programmable Interconnect Point
PL	Programmable Logic
PR	Partial Reconfiguration
PRET	Pre-configuration Test
PS	Processing System
PSM	Programmable Switch Matrix
RA	Reconfigurable Architecture
RM	Reconfigurable Module
RP	Reconfigurable Partition
RTL	Register Tranfer Level
SAF	Stuck-At Fault
SDK	Software Development Kit
SoC	Systems-on-a-Chip
SR	Shift Register
TC	Test Configuration
TF	Transition Faults
TPG	Test Pattern Generator
XDL	Xilinx Design Language
XMD	Xilinx Microprocessor Debugger
XPS	Xilinx Platform Studio

Chapter 1

Introduction

Contents

1.1 Motivation

Reconfigurable architectures are often considered to be able to offer cost saving by time-multiplexing hardware resources that are contained within them. The reconfiguration includes the alteration of the complete system or part of it. However, there is always a trade-off among performance, overhead and efficiency. The most popular reconfigurable architecture is the Field Programmable Gate Array (FPGA), with the dynamic Partial Reconfiguration (PR) enabled in the newer series [6].

Contemporary FPGAs are more prone to aging effects, latent defects and transient effects. The runtime system has to ensure a reliable reconfiguration process, and achieve dynamic adaptability to defective parts in the fabric. To detect the defective fabric in the field, requires part of the logic area to be reprogrammed for testing, while the remaining part of the logic is kept functioning as usual. Once the faulty area is detected, the logic functions can be swapped out from the defective region to a partition where the functionality is tested and known to be good.

With the availability of PR, it is feasible to allow specific regions of the FPGA to be evaluated while the remaining function of the device is preserved. PR performs hardware reconfiguration by dynamically modifying parts of logic blocks in an FPGA, and allows unmodified logic blocks to continue to operate as before [7]. Hence, the defective area in the FPGA can be avoided, a reliable runtime reconfiguration system can be accomplished.

FPGA test includes the Pre-configuration Test (PRET) of unprogrammed logic cells and functional post reconfiguration test. The reconfiguration process can be handled by either an external controller or an embedded processor [3]. This work focuses mainly on online PRET controlled by an on-chip processor core. An embedded processor contained in a runtime system can access the FPGA reconfigurable space by downloading configuration data via the configuration access port [8], and is capable of controlling the execution of Built-In Self Test (BIST) online. This enables the user to implement the reconfigurable architecture by modifying the circuit structure and changing the functionality through software control. Test Configurations (TCs) can be scheduled on a regular basis, and loaded into partitions which are currently not in use. Tests are performed accordingly.

The Xilinx Zynq-7000 SoC leverages the strengths of an ARM processor core and a Xilinx FPGA, it combines ARM cores with 28 nm programmable logic on a single chip, and delivers a reconfigurable embedded processing platform [9]. This core-centric architecture offers processing capability, the ease of programmability and the flexibility of an FPGA [10]. Furthermore, Xilinx embedded solution offers not only a tool to synthesize, place and route the design, but also enables the management of reconfigurable modules; it provides an easy way to floor plan, and acts as the overall project manager when instantiating the embedded systems.

1.2 Reliability Challenges

Aggressive low cost requirements and spatial constrains continuously drive transistor scaling, which results in higher on-chip integration and faster operating frequency. Consequently, FPGAs have enhanced system performance. To stay within certain power budgets, the latest 28 nm FPGA platforms include low power process innovations and stacked silicon interconnects.

However, due to the complexity of the manufacturing process, process variations and latent defects are introduced in these high-end FPGAs. On top of that, environment variations and different life spans of the silicon change the behavior of the device as well [11]. Normal production test and burn-in test are no longer able to capture these transient and aging effects. Performance degradation and erroneous behavior lead to reliability threats, and even critical safety concerns.

To improve the reliability of FPGAs and build a fault-tolerant platform, online testing becomes necessary for these reconfigurable systems.

1.3 Online Test Strategies

The aforementioned reliability issues raise the test requirements for reconfigurable architectures. Prior to reconfiguring with new functions, the FPGA fabric needs first to be validated with self-test. Based on the previous work implemented for the Xilinx Virtex-5 FPGA family [3], this thesis extends the structural online test method [4] to the Xilinx 7 Series programmable logic family. The PRET process can be triggered through either a hardware state machine or software control from an embedded processor. In addition, this thesis develops a self-test wrapper in a hybrid SoC platform, the reconfiguration process as well as the self-test are triggered by the microprocessor.

The reconfigurable partition (container) and test benches are integrated into the self-test wrapper, the wrapper is then connected to a Advanced Microcontroller

Bus Architecture (AMBA) Advanced eXtensible Interface (AXI) bus interconnect. The full implementation extends the flexibility of the PR by allowing the software running in the microprocessor to control the loading of TCs. Figure 1.1 shows the target reconfigurable SoC with the developed self-test wrapper.

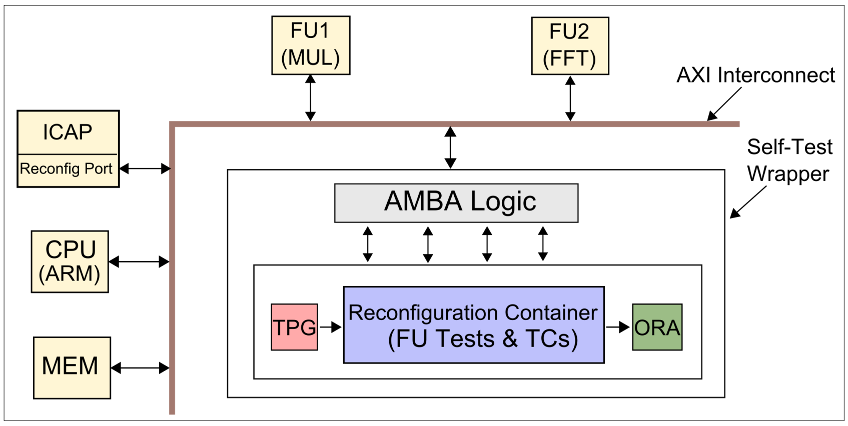


Figure 1.1: Online Self-test Wrapper

Similar approaches have been adopted to develop the TCs for 7 Series FPGA, built on top of the RapidSmith java framework [12]; where code is modified and reused to certain extend, to suit the Configuration Logic Block (CLB) architecture of the 7 Series. Chapter 2 describes the detailed structural differences between Virtex-5 and 7 Series CLBs.

After the corresponding reconfiguration container is configured as a test array with a TC and is ready to be tested, the Test Pattern Generator (TPG) generates input stimuli to the container and the Output Response Analyzer (ORA) analyzes the outputs from the container. The programmable part of the device consists of a PR module and the static logic; after it is fully configured with a complete bitstream, the partial bitstreams can be downloaded to modify the reconfigurable portion, and change the TC design.

1.4 Thesis Goal and Outline

The main purpose of this thesis is to develop and evaluate a self-test wrapper for such a on-line reconfigurable system. The purpose of the wrapper is to encapsulate a block of reconfigurable logic and implement an interface to connect the self-test hardware via the AMBA AXI4-Lite bus interconnect to the ARM microprocessor. The generation of test stimuli and analysis of responses can be performed under the software control, as well as functional evaluation. The tool chains used are introduced and the implementation details are presented.

After a brief introduction, this thesis first introduces FPGA-based reconfigurable systems and partial reconfiguration. It goes on to compare the basic architectural differences between Virtex-5 and 7 Series CLBs. Then in Chapter 3, based on the comparison results, it describes the test requirements and implementation of TCs for the targeted hardware. Followed by design of TPG and ORA for each TC using hardware description language. Chapter 4 gives the detailed information about designing the self-test wrapper, including an overview of the processor architecture, the AXI interface protocol and requirements.

The remainder of the report is organized as follows. Chapter 5 presents the tool chain for the TC generation, the embedded development tool suite and the steps for partial reconfiguration of a processor peripheral. Chapter 6 demonstrates the design of a software project based on the software development platform, which enables the integration of hardware and software components. Software-controlled TCs reconfiguration and tests execution are also explained. It is followed by the validation and implementation results in Chapter 7. This thesis is concluded in Chapter 8 with a summary and possible further tasks.

Chapter 2

FPGA-based Reconfigurable Systems

Contents

2.1 Introduction to FPGAs

Field Programmable Gate Arrays (FPGAs) are silicon devices that are based upon an array of Configurable Logic Blocks (CLBs) connected together through programmable switch interconnects. An FPGA has much more logic resources compared with Programmable Logic Device (PLD) and its programmable nature distinguishes it from Application Specific Integrated Circuit (ASIC); it outperforms an ASIC in the way of dynamical reconfiguring tasks from one to the other at runtime.

A brief overview of the FPGA architecture gives a basic understanding to the testing of reconfigurable systems. The fundamental FPGA building blocks include CLBs, block RAMs, DSP slices and IO logic resources. CLBs communicate with each other through the programmable switch matrices (PSMs) and interconnect wires. Input/Output Blocks (IOBs) are used to connect the FPGA to the outside world [3]. Figure 2.1 represents the basic FPGA block structure. Due to reprogrammable capabilities, SRAM-based FPGAs are the dominant type. The following subsections describe the basic components in an FPGA.

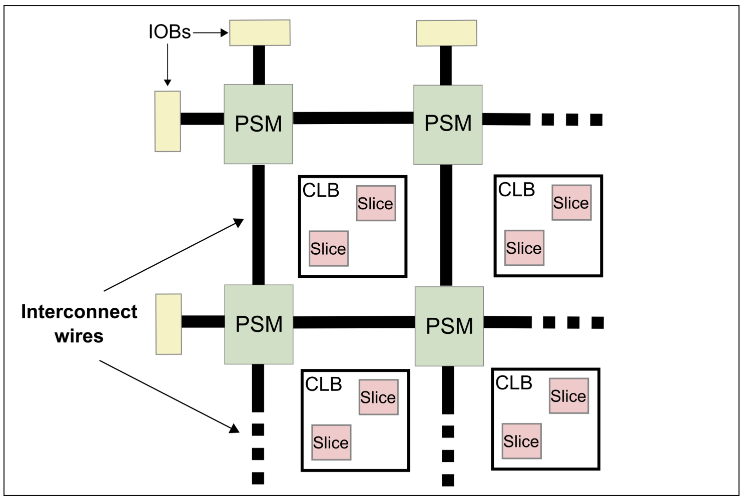


Figure 2.1: FPGA basic building Blocks

2.1.1 Configurable Logic Block

The CLB is the basic logic unit and the foundation of an FPGAs; it is essential for re-programmable digital logic design. A CLB consists of Lookup Tables (LUTs) with 4 to 6 inputs, selection circuitry (Multiplexers, XOR) and sequential elements such as Flip-Flops (FFs). The LUTs act as function generators, and can be flexibly configured as combinatorial logic, shift registers or RAM. Multiplexers are used to specify the signal connections inside the CLB; signals from the logic input portion of a slice are routed through to the sequential elements. The sequential elements can be configured as either a FF or a latch to implement edge or level sensitive designs.

A LUT realizes combinatorial logic functions through SRAM configuration, the SRAM cells store the truth table values based on the logic operation specified. A multiplexer inside the LUT selects the appropriate truth table value for the LUT output depending on the number and combination of LUT inputs. For example, a multiplexer of a 6-input LUT can select from 32 configurable SRAM cells, the SRAM cells are connected to the data inputs of the multiplexer while the 6 inputs act as the functional selection inputs. In this way, the logic function is implemented. The configuration remains stable when a circuit is online and no reconfiguration is performed [3].

The above three subcomponents are combined to form an entire CLB, providing the logic capability of FPGA.

2.1.2 Switch Matrix and Interconnect

Interconnect routing is another essential factor for FPGAs. CLBs interconnect to each other and to IOBs using programmable switch boxes. The flexibility of switch matrices allows any point in the circuitry to be able to be connected to another point. Interconnect wires route the signals between CLBs by horizontal and vertical lines crossing over the device, while clock and global signals are routed by global routing. The design tool takes care of the interconnect routing tasks, user interaction is not required.

Routing is determined by configuration, and carried out by interconnect wires and programmable switches, which route the signals into their correct path. The interconnects between the CLB pins are programmable switches, they are grouped together to form the PSM. The PSM makes connections between the various pins attached to it, as such it connects CLB pins to interconnects. The connections that exist within the same CLB (intra-tile connections) are programmable, they are called Programmable Interconnect Points (PIPs). PIPs establish the possible connection between two local wires by using programmable switches [3].

The programmable switches and interconnect wires are organized in a way to realize unique functions specified.

2.1.3 IOBs, Memory and Clock Management

Contemporary FPGAs support many different Input/Output standards, they provide compatible interface for the system. IO Blocks are grouped into banks, each bank is able to support different I/O standards independently.

Most FPGAs contain embedded blocks of RAM memory, which are connected to form a large on-chip memory. For instance, the processing system contained in the Xilinx Zynq family provides instruction and data L1 cache, shared L2 cache and on-chip RAM memory, which can be included in the design to support low-latency memory access.

The advanced FPGAs also offer digital clock management, this feature provides precise clocks with less deviation from a reference clock and filtering, allows complete clock management.

2.2 Reconfigurable Architectures

Reconfigurable Architectures (RAs) are the devices with programmable logic blocks and programmable interconnects between them. RAs perform computational tasks with logic blocks instead of instruction sets, which avoids the overhead for loading/decoding of instructions and dependence on the sequential execution nature of the application. The programmability and efficiency make RAs a promising solution to bridge the gap between application specific and general purpose architectures for reconfigurable computing [13].

Fine-grained and coarse-grained architectures are the two basic categories of

RAs. RAs can be implemented using FPGAs; the logic elements and interconnects of FPGA are operating at bit-level, they are able to realize applications with arbitrary word-lengths. Therefore, FPGAs are considered as fine-grained reconfigurable architectures. In contrast, coarse grained reconfigurable architectures provide wide-width configurability; increase the granularity (the size of elements can be explicitly reconfigured) of functional blocks and interconnect structures, create less overhead [14].

This section discusses the differences between these two RAs and their major advantages and disadvantages, then comes to the conclusion that tightly coupled hardware architecture with processor and reconfigurable hardware integrated into the same chip gives the most advantage in particular applications.

2.2.1 Coarse-grained Reconfigurable Architectures

When implementing word level operations, coarse-grained RAs use multiple-bit wide datapaths instead of bit-level operations. The wide datapath allows efficient implementation of complex operators; avoids the large routing overhead introduced by bit-level processing units, which need to compose complex operators. Coarse-grained RAs consist of higher granularity of computational elements, typically, the volume of configuration data needed is several orders of magnitude lower than that in an fine-grained FPGA. Therefore, the reconfiguration time is reduced. In addition, the interconnects between processing elements have a wider bit-width, which implies a higher area usage for a single line. Hence, fewer number of lines are required, this results in less area usage for global routing. These devices provide efficient performance, high area utilization and reduced power consumption [15].

2.2.2 Fine-grained Reconfigurable Architectures

There are several disadvantages for fine granularity to perform computational tasks. Due to bit-level operations, operators for wide data sources have to be processed by several processing units, leading to large routing overhead and low silicon area efficiency. Also, the switched routing wires generate high power dissipation. In addition, the high volume of configuration data needed for the large number of processing units and interconnects requires a long configuration time. On top of this, FPGAs are programmed using a high-level language, the operation at bit-level does not match the functions specified in the high level language, therefore the synthesis process takes a long time [15].

These fine-grained reconfigurable hardware architectures consist of processing elements, which communicate with each other via the programmable interconnects. Theoretically, any computation can be implemented on these fine-grained devices, since they are operating at a bit-level. However, they require more reconfiguration data and longer reconfiguration time compared to coarse-grained RAs, which leads to inefficient usage of the reconfigurable hardware. Furthermore, using pure reconfigurable hardware architectures, the reconfigurable hardware usually is not large enough to load the entire program, reconfiguration may be needed for runtime execution of the application. In this case, the code that is rarely used will also be mapped onto the reconfigurable hardware, which leads to inefficient usage of the reconfigurable hardware and slowdown the program execution [13].

Coupling a general-purpose processor with reconfigurable hardware on the same chip, can result in optimal flexibility and efficiency. Code regions that are frequently used to be mapped onto the reconfigurable hardware, can have faster program execution. Some reconfigurable architectures like RISPP (Rotating Instruction Set Processing Platform) [13], introduces the concept of special instructions (SI) and instruction rotation. A single SI can have multiple implementation instances and the runtime system decides which instance should be loaded onto reconfigurable hardware. At design time, multiple implementations are composed, and at runtime, the RISPP system controls the infrastructure of the reconfigurable fabric and realizes runtime adaptive execution [13].

It concludes that integrating a fine-grained reconfigurable fabric with a processor on the same chip, and utilizing reconfigurable hardware for mapping special instructions, can achieve runtime adaptability, better performance and speed up.

2.3 Partial Reconfiguration

An FPGA has the flexibility of being configured and reconfigured after manufacturing, which allows the user to change the functions of the device without refabricating it. The Partial Reconfiguration (PR) feature goes one step further, it allows us to modify a subset of the resources on an operating FPGA by loading the reconfiguration bit file. After downloading the full bit file to configure the FPGA, the design of an operating FPGA can be dynamically modified by loading the partial bitstream in the reconfigurable logic blocks, while the application running on the remaining logic is not interrupted [8]. The overall concept of PR action is depicted in Figure 2.2.

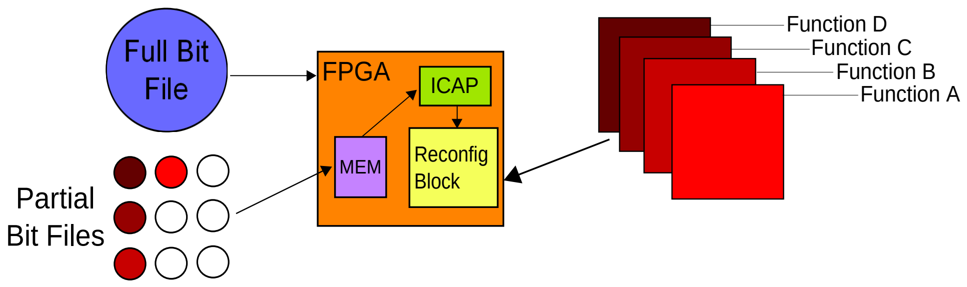


Figure 2.2: Partial Reconfiguration

To date, devices with hardware support for PR have existed for quite a long time. Now with the software and tool support, PR is widely adopted for reconfigurable systems. PR enables specific regions of the FPGA to be tested without compromising the device operation, it brings flexibility, cost and power reduction to the overall system.

2.3.1 Partial Reconfiguration of a Processor Peripheral

In this thesis, the reconfiguration process as well as the self-test are supposed to be triggered by the embedded processor. In order to perform partial reconfiguration in a system with a microprocessor during runtime, each partial configuration file is first converted into a partial bitstream, and then stored in the memory. The runtime reconfigurable system fetches the partial bitstream out of the memory and sends it through the Internal Configuration Access Port (ICAP) or Processor Configuration Access Port (PCAP) interface into the reconfigurable partitions. After the PR flow loads the partial bits to the reconfigurable region, the content of that region in the FPGA is modified. The AXI Hardware Internal Configuration Access Port (HWICAP) core for the AXI Interface enables an embedded processor to read and write the FPGA configuration space via the ICAP, it supports data reading and partial bitstream loading through AXI4-Lite Interface [16]. This enables the user to write software programs to modify the circuit structure and functionality during runtime, so PR can be conducted for the reconfigurable system under software control.

2.3.2 ICAP versus PCAP

This thesis is based on an evaluation board of the Zynq-7000 device. PCAP is the recommended reconfiguration mechanism for Zynq-7000 designs [7]. At power on, the processor selects PCAP as the configuration interface to the Programmable Logic (PL) and configuration is performed by the processor through PCAP to the PL. After configuration, the processor may switch the configuration interface to ICAP (internal to PL) by writing the control bit through an instruction executed on the processor, the control is then passed over to ICAP. PCAP configuration belongs to the processor, while ICAP belongs to the PL [7]. The processor may take over the control of PCAP back at any time. ICAP does not have security management. PCAP and ICAP are mutually exclusive, and cannot be used concurrently.

2.4 Target Hardware

The targeted hardware platform of this work is the Xilinx Zynq-7000 SoC. The Zynq family consists of a dual-core ARM Cortex-A9 processor based processing system and PL built on 28 nm process technology. The PL of Zynq device on ZedBoard is an Artix-7 based FPGA, which belongs to 7 Series FPGA. To test the programmable part of this SoC device, we need to know the structure of 7 Series FPGA. DSP blocks, block RAM and IOB are not taken into consideration, they are out of the scope of this thesis. The main logic components in the FPGA are the CLBs, this work will focus only on CLB tests.

In the previous work for Virtex-5 (refer to [3]), the test configurations were developed based on the RapidSmith Java framework. For porting the existing CLB test designs from Virtex-5 to the programmable resources on the Zynq-7000 device, first let us take a close look at both CLBs.

2.4.1 Common Features between Virtex-5 and 7 Series CLBs

There are many similar aspects between Virtex-5 and 7 Series CLBs. Each CLB contains a pair of logic slices, the two slices in the same CLB have no direct connections to each other. Slices are organized as columns and form independent carry chains [1] [2]. See figure 2.3 below for the arrangement of CLBs and slices for both Virtex-5 and 7 Series.

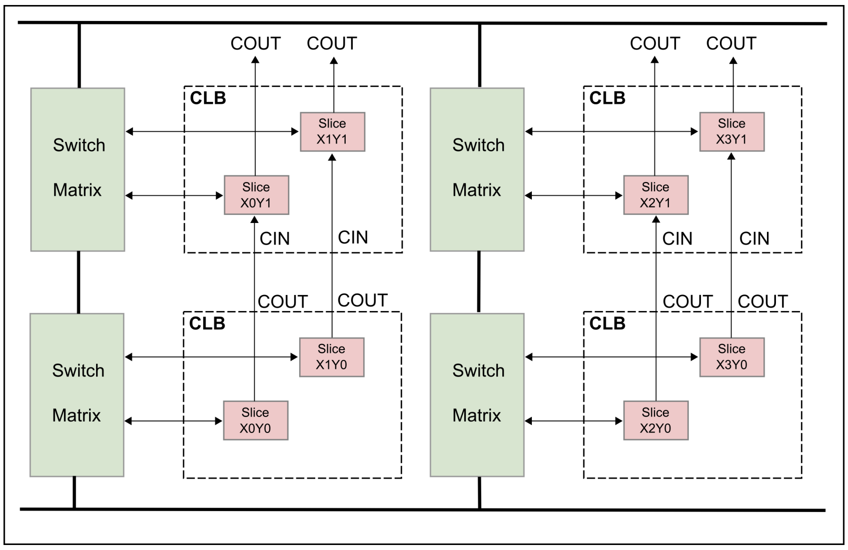


Figure 2.3: Arrangement of CLBs and Slices

Slices have unique names in the form of XnYn: the X number denotes the column position of the slice, whereas the Y number denotes the row position, the Y number remains the same within a CLB. Starting from the bottom left of the die, the number counts up in sequence from X0Y0. Thus, the position of the slice is identified. Figure 2.3 illustrates how the slices are defined for four CLBs with start point in the bottom-left corner of the die.

A CLB can have two types of logic slices: SLICEL and SLICEM. Each CLB contains either two SLICELs or one SLICEL and one SLICEM. Every slice contains LUTs, multiplexers, carry chain logic and storage elements. These elements provide logic, arithmetic, and memory functions. In addition to LUT functionality, SLICEM LUTs can also be configured as storage elements using distributed RAM and shift registers. SLICEL does not support these additional functions. Therefore, SLICEM represents a superset among all slices [1] [2].

Each slice has 4 basic LUTs, each of which can implement an arbitrarily defined 6-input or 5-input Boolean function. When implemented as a 6-input function, the

LUT has only one output. When implemented as dual 5-input LUTs, these two functions share common inputs, and both outputs can be used. In this case, the most significant input bit is driven high by the software. There is no difference in propagation delay through a LUT for a 6-input or 5-input LUT. Refer to figure 2.4 and figure 2.5. Signals that exit from the outputs of a LUT, feed into a multiplexer and finally reach a storage element, which can be configured as either edge-triggered D-type FFs or level-sensitive latches. When configured as a latch, the latch is transparent when the clock signal is low [1]. The carry chain runs upward and has a height of four bits per slice. For each bit, there is a carry multiplexer and a dedicated XOR gate which runs through the middle of the slice to perform fast carry computations. The carry path and multiplexers can also be used with LUTs to implement more logic functions [1] [2].

There are several configuration options for the four sequential storage elements, they can be configured as either FFs or or latches, initialized to "0" or "1", and the reset (SR) can be set as either active-high or active-low. The set and reset function for the slice has several options, such as no set/reset, synchronous set/reset or asynchronous set/reset, the work in this thesis always selects asynchronous reset.

In addition to the LUTs, slices contain three additional multiplexers to combine with the four LUTS to provide seven- or eight-input functions in a slice. These multiplexers will not be tested in the scope of this work, this can be considered for future work. Functions with more than eight-input can also be implemented using multiple slices, in this case, there are no direct fixed connections between slices.

A LUT in both SLICEM and SLICEL can implement a 64 x 1-bit Read Only Memory (ROM), three configuration options are available which depend upon the number of LUTs used. ROM contents are loaded during device configuration. Refer to table 2.1 for details. The number of LUTs needed and associated multiplexers for the implementation are also shown. A LUT in SLICEMs can also be configured as distributed RAM elements and shift registers.

ROM	Multiplexer	Number of LUTs
64 x 1	4:1	1
128 x1	8:1	2
256 x 1	16:1	4

Table 2.1: ROM and Multiplexer versus Number of LUTs

2.4.2 Virtex-5 CLB Architecture

In Virtex-5 CLB, each slice consists of 4 LUTs and 4 FFs. Table 2.2 shows the logic resources within one Virtex-5 CLB.

Slices	LUTs	Flip-Flops	Arithmetic and Carry Chains	Distributed RAM (SLICEM only)	Shift Registers (SLICEM only)
2	8	8	2	256 bits	128 bits

Table 2.2: Logic resources in one Virtex-5 CLB

Figure 2.4 shows a SLICEM of Virtex-5. It consists of a circuit repeated four times. This circuit consists of a LUT connected to multiplexers and finally a sequential element.

Every other CLB column contains a SLICEM. In addition, the two CLB columns to the left of the DSP48E columns both contain a SLICEL and a SLICEM.

Note that the control signal reverse (REV) is not present in a 7 Series slice (function equals to REV=0 in Virtex-5).

2.4.3 7 Series CLB Architecture

Given the above mentioned Virtex-5 CLB architecture, the main difference in the 7 series CLB is that each slice contains 4 LUTs but 8 storage elements (one additional storage element per LUT). Figure 2.5 shows a 7 Series SLICEM and the differences to Virtex-5 SLICEM. Only four of the eight storage elements can be configured as either edge-triggered FFs or level-sensitive latches. The other four additional storage elements can only be configured as FFs, and the D input can be driven by the output of 5-input LUT (O5) or the BYPASS slice inputs via AX, BX, CX, or DX input. When the original four of the eight storage elements in a slice are configured as latches, the remaining four storage elements in that slice must remain unused.

The multiplexers and arithmetic carry logic are the same as Virtex-5 CLB. LUTs can be used for random logic implementation or distributed memory, and can be configured as either one 6-input LUT (64-bit ROMs) with one output, or as two 5-input LUTs (32-bit ROMs) with separate outputs but common addresses or logic inputs. When memory LUTs are configured as 64x1 or 32x2 bit RAM or shift register, the data inputs are 'I' in 7 Series SLICEM instead of 'X' in Virtex-5 SLICEM. Each LUT output can optionally be registered in a flip-flop. See table 2.3 below for reference.

Slices	LUTs	Flip-Flops	Arithmetic and Carry Chains	Distributed RAM (SLICEM only)	Shift Registers (SLICEM only)
2	8	16	2	256 bits	128 bits

Table 2.3: Logic resources in one 7 Series CLB

Based on the above CLB structural comparison results, the test approach and methodology for testing Virtex-5 CLBs can be leveraged for configuring and testing of 7 Series CLB components.

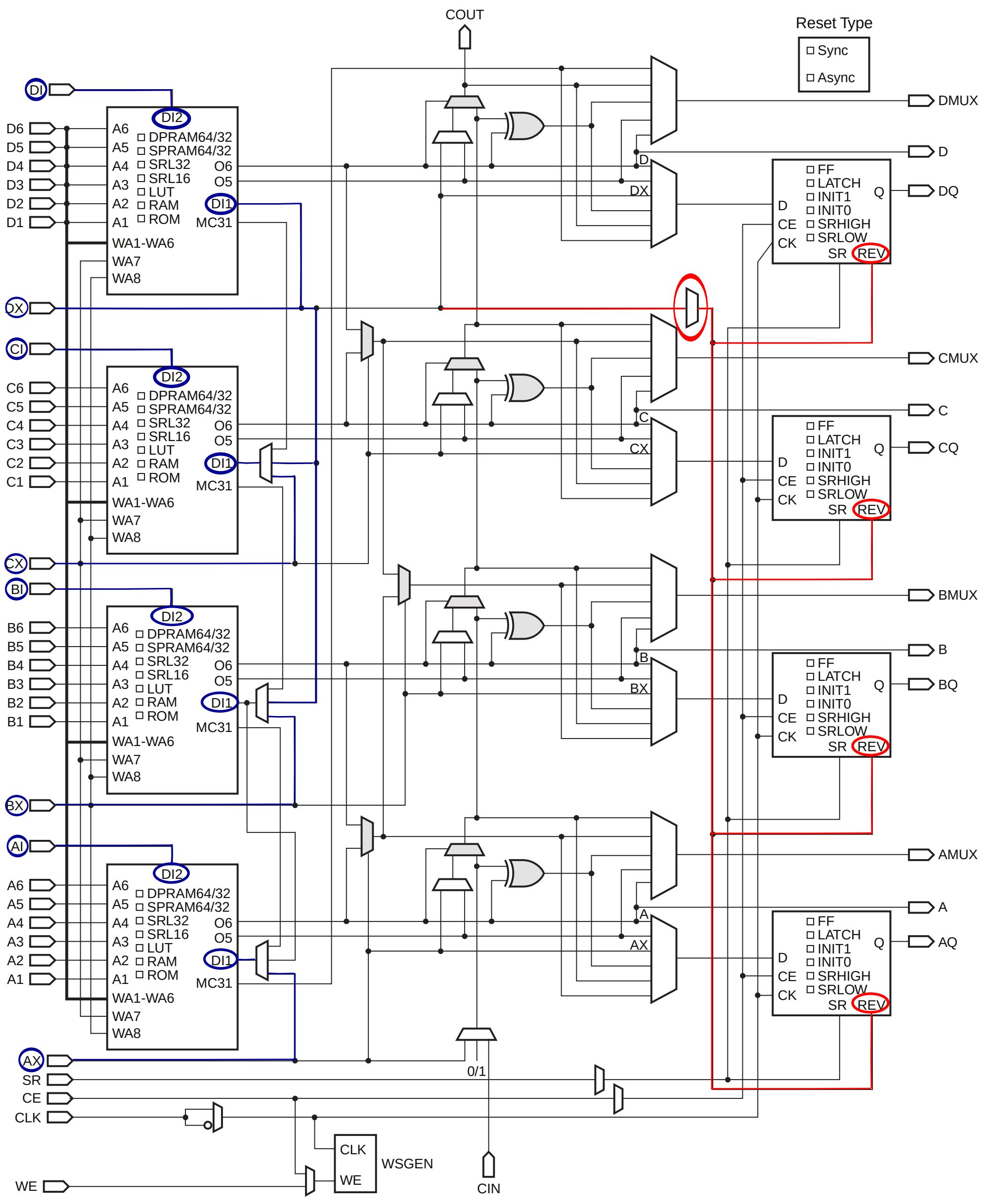
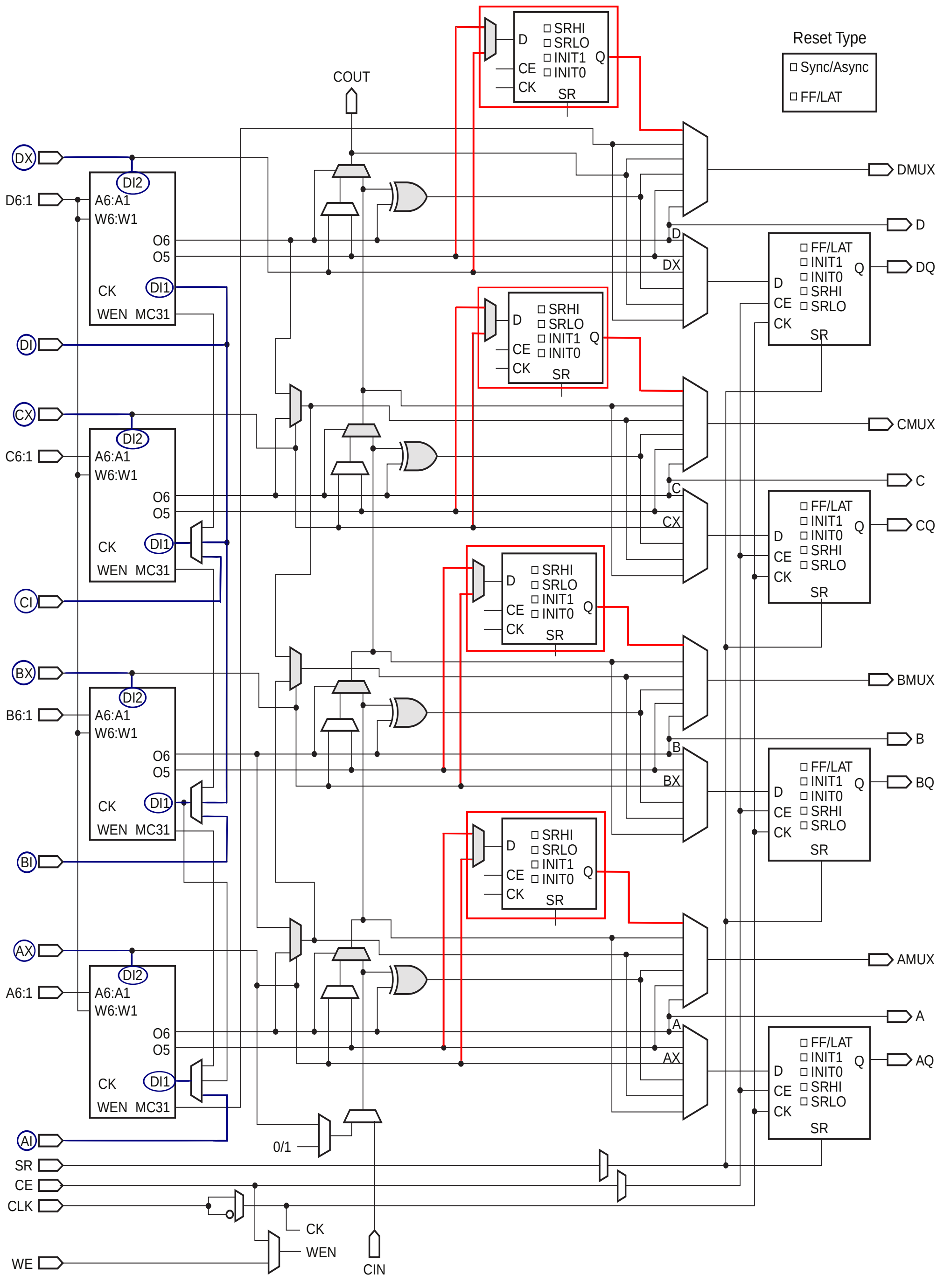


Figure 2.4: Diagram of Virtex-5 SLICEM (from [1])



* Red Mark: extra in 7 Series. Blue Mark: difference between Virtex-5 and 7 Series CLB.

Figure 2.5: Diagram of 7 Series SLICEM (from [2])

Chapter 3

Test of Reconfigurable Hardware

Contents

This chapter discusses FPGA testing. After presenting fault models and FPGA test methodologies, CLB subcomponent tests are introduced. Then based on the previous work on Virtex-5 FPGA, the chapter describes the design of Test Configurations (TCs) for the target hardware of this thesis, as well as the Test Pattern Generation (TPG) and the Output Response Analysis (ORA).

3.1 Fault Models and Principles of Test

Modern FPGAs are more prone to latent faults and aging defects. For performing online test of FPGAs in the field, external equipment is not available. It is necessary to perform on-chip self testing, i.e., Built-In Self Test (BIST). By applying test patterns at the inputs and comparing output responses with the expected values, one can tell if the test is a pass or fail [3].

For testing FPGAs, it is required that the structural knowledge of the circuit to be tested. As mentioned in the last chapter, CLB testing is the main focus of this work. To evaluate each subcomponent of CLB, several TCs need to be established to ensure the testability of the circuit, corresponding test stimuli must be applied and outputs need to be analyzed. FPGAs are reprogrammable, so the circuit to be tested can be reconfigured by multiple TCs. Each TC targets a subset of components. It is also required to test the interconnects, however, many of the interconnects are already tested during CLB testing [17]. In addition, due to complexity, the number of TCs required for interconnect testing is much higher. The number of TCs determines the test speed, because the configuration time is several orders of magnitude higher than that of test application [4]. To minimize the test execution time, this work focuses only on CLB structural testing, interconnect testing is not described here.

3.1.1 Fault Models

To detect CLB structural defects, the Stuck-At Fault (SAF) model is the most commonly employed for fault derivation [18]. However, CLBs consist of logic gates, memory elements, multiplexers and storage elements, the implementation details of most CLB subcomponents are unknown. To derive a more accurate list of fault models, additional functional faults have to be accounted for a specific set of components. For instance, to detect defects in the RAM and combinational faults in the Lookup Table (LUT), respective fault models are used.

This subsection describes the fault models used for each CLB subcomponent. Figure 3.1 provides an overview of complete CLB fault models for easy understanding.

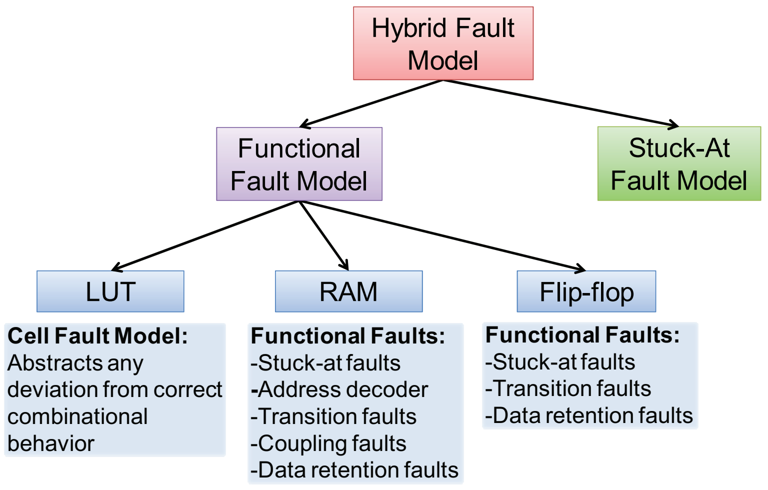

Hybrid Fault Model
Functional Fault Model
Stuck-At Fault Model
LUT
RAM
Flip-flop
Cell Fault Model: Abstracts any deviation from correct combinational behavior
Functional Faults: -Stuck-at faults -Address decoder -Transition faults -Coupling faults -Data retention faults
Functional Faults: -Stuck-at faults -Transition faults -Data retention faults

Figure 3.1: Complete list of CLB Fault Models (from [3])

This thesis is under the presumption of the single SAF model. In this fault model, one of the signal lines in the circuit under test is stuck at a fixed logic value, either "0" or "1", no matter what combination of inputs is applied. Therefore, a circuit with total of n signal lines, the maximum possible number of single stuck-at faults is 2n.

3.1.1.1 LUT in Functional Mode

Under the assumption that the internal structure of the unit under test (black box) is unknown, to test LUTs in combinational function mode, the Cell Fault Model

(CFM) [19] is applied. By thoroughly applying any of the input combinations to the black box and comparing the results received from outputs with the expected values, any mismatch is considered as a cell fault. The CFM is a more extensive functional fault model, compared to the SAF [19], for it models any fault that deviates a cell from the correct behavior. All single and multiple stuck-at faults that may occur are covered for the cell under test.

The drawback of CFM is that all of the input combinations must be exercised and the total number of cell faults can be huge for cells with a large number of input and output lines.

3.1.1.2 LUT in RAM Mode

When a LUT is configured as RAM, the functional faults associated are similar to classic memory testing faults, which can be categorized as the following fault models:

1. Address decoder Faults (AFs): The faults in the address decoder cause an incorrect address access.
2. Stuck-At Faults (SAFs): A memory cell is stuck at either a “0” or “1” value.
3. Transition Faults (TFs): A cell is incapable of switching from “0” to “1” or from “1” to “0” promptly.
4. Coupling Faults (CFs): Memory cells undertake a wrong value due to the switching activity in neighbouring cells.
5. Data Retention Faults (DRFs): A memory cell is unable to retain its data value after a certain period.

In this work, to reduce the hardware overhead, only reduced fault sets are taken into consideration. These are AFs, SAFs and TFs. Besides, these three faults predominate all defects in RAM modules [3]. An existing memory test algorithm is used in this work to test RAM faults.

3.1.1.3 LUT in Shift Register Mode and Sequential Elements

When a LUT is in Shift Register (SR) mode, the connection between the SRAM cells in a LUT is similar to Flip-Flops (FFs) connected in series. When testing interconnection of the LUT SRAM cells connected in series, the functional faults detected are the same as the faults occurred during testing FFs. For all sequential elements in a slice, such as the storage elements configured as FFs or latches and LUTs in SR mode, the SAFs (stuck at “0” or stuck at “1”) and transition faults (slow to rise or slow to fall) are the four dominant faults [3].

3.1.1.4 Structural Stuck-At Fault Model

The faults in the aforementioned three modes belong to functional faults. For the remaining subcomponents in a CLB, such as the multiplexers, XOR cells and interconnect wires in a CLB, structural faults are targeted, SAFs are used to model the structural faults [3].

3.1.2 Principles of CLB Test

As presented in the last chapter, CLB components can be divided into three main subsets, which can be tested separately. These are LUTs, multiplexers and the sequential elements. Different test approaches are applied for each of the CLB subcomponent. The next section explains the underlying test methodology in detail.

It is important to mention the term of C-testability in the context of CLB testing. An array of logic circuits that can be tested using a fixed number of TCs, irrelevant of array size [3], is called C-testable. The structure of FPGA is fairly homogeneous, CLBs are arranged in a regular array architecture, repeatedly throughout the FPGA. In order to perform an array-based CLB test, a container is set up. The logic elements are configured into an iterative C-testable array inside the container. To have a full-coverage test, the container requires an appropriate number of TCs, each of them targeting specific CLB subcomponents.

When a TC is configured into a dedicated test container, the corresponding TPG for that TC is applied, and ORA is captured at the output accordingly. Figure 3.2 shows the CLBs configured as a C-testable array in a container with external TPG and ORA, it is based on the similar picture taken from [4].

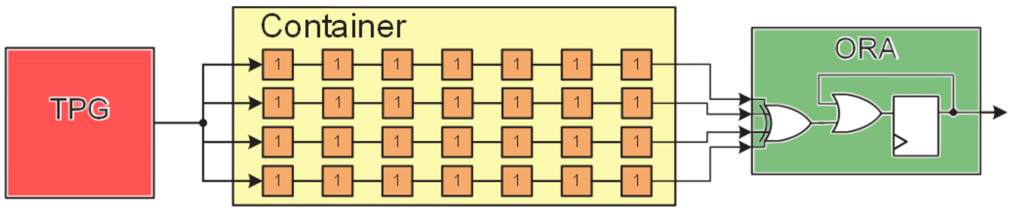


Figure 3.2: Container in a C-testable array with external TPG and ORA

A C-testable CLB array can be fully tested by applying exhaustive test patterns at the input of the first cell in an array, and obtaining the output responses at the last cell of the array, no matter how long the array is. However, to avoid long timing critical paths when the array becomes deep, the CLB subcomponents are pipelined. This is achieved by including the sequential element the sequential element of CLB into the path and connecting the CLBs in an interleaved way [4]. The structure for an interleaved CLB array is shown in figure 3.3.

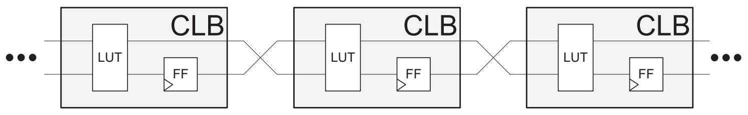


Figure 3.3: CLB in a fully interleaved array (from [4])

3.1.3 Test Flow for FPGAs

The first step is to define the location and size of the container by the two coordinates. The CLBs to be tested are connected in a C-testable array. The appropriate number of TCs needs to be defined and implemented to have a full coverage of the CLB test. Finally the partial bitstreams containing the TCs set-up for the container are generated. For test execution, the container is subsequently configured with all TCs following the Partial Reconfiguration (PR) process, with each TC enabling test of a subset of complete logic elements.

Each TC has its own specific TPG, whereas ORA design is the same among all different TCs, it is implemented with XOR gates providing the required comparison [3]. The TPG applies the suitable test patterns to the CLB array, its responses are captured and evaluated by the ORA. The final test result is then obtained from the output of the ORA. The steps of partially reconfiguring the container followed by the test execution are repeated throughout all TCs to ensure the full coverage of CLB faults.

3.2 Design of Test Configurations

In previous work [3], the CLB structural self-test was evaluated on a Virtex-5 evaluation board. It includes 9 test configurations to perform a full structural test of the Virtex-5 CLB. TCs were developed on RapidSmith java framework, the techniques covered the following CLB subcomponents:

- Lookup Table - Function mode
- Lookup Table - Shift Register mode
- Lookup Table - RAM mode
- Multiplexer
- Fast Carry Chain
- Flip-Flop or Latch

Based on Virtex-5 TCs design, this work extends the development of TCs for 7 Series CLBs. For the evaluation of the concepts in real hardware, an evaluation and development board based on the Xilinx Zynq-7000 SoC - ZedBoard is used. The Artix-7 CLBs are the main logic elements inside the programmable logic part of the Zynq device on that board. In addition, the TPGs and ORAs are taken out from java implementation, which were included in Virtex-5 TCs design [3], and are designed separately together with the static logic. This eliminates the BIST hardware overhead from the test architecture.

3.2.1 CLB Subcomponent Tests

In this subsection, the test for each of the subcomponents of CLB is presented.

3.2.1.1 LUT - Function mode

LUTs are the main functional elements in CLBs. When a LUT is configured as a combinational function mode, to cover all single or multiple internal faults, all cell faults are targeted. An exhaustive set of input patterns is applied as the test patterns. A LUT is configured with either an XOR or XNOR configuration to ensure C-testability.

3.2.1.2 LUT - Shift Register Mode

The FFs are configured into a long shift register, both SAFs and TFs are targeted. The scan chain test pattern "01100" is applied, because it contains the transitions from "0" to "1" and from "1" to "0".

The LUTs in SR mode are connected into multiple scan chains. For response analysis, the outputs of the scan chains are compared with each other. To reduce the number of required TCs, the FF in the CLB is included into the scan chain. By placing FFs in-between SRs, these FFs are tested simultaneously with the SR test. Hence, two CLB subcomponents are tested in a single TC.

3.2.1.3 LUT - RAM mode

Test patterns are generated based on the MATS++ algorithm to ensure coverage of all SAFs, AFs and TFs. The response analysis is performed by mutually comparing the outputs of these RAM blocks and aggregating the results into the global ORA.

3.2.1.4 Multiplexer

The MUX is tested by applying all possible configurations to exercise all combinations of the selected inputs. SAFs are tested by applying both '0' and '1' stimuli for data inputs. Multiplexer testing is often included in other tests since they are used for internal routing of subcomponents in a slice.

3.2.1.5 Fast Carry Chain

The carry chain consists of multiplexers and XOR cells, for which SAFs are targeted. To test carry chain elements effectively, they must be connected in pipelined C-testable arrays. Two TCs are required for a full coverage of carry chain test. In one of the TCs, the multiplexers are transparent and the carry chains are configured into XOR arrays. In the other TC, the carry-out pin has a dedicated interconnect to the carry-in pin of the neighboring slice. The chain elements are connected in a long carry chain and "0" and "1" values are propagated through to test for SAFs. A FF at the end of each column is used to pipeline the test.

3.2.1.6 Latches and Flip-Flops

As described in LUT in SR mode section, testing of the edge sensitive FFs is performed together with the test of the SRs. If sequential elements are configured as level sensitive latches, a separate test is required for the proper latch function. For testing the targeted SAFs and TFs, two non-overlapping clocks are required as input to the scan chain. The same test pattern "01100" is used for both FF and latch testing.

3.2.2 Extension for 7 Series TCs

The number of required TCs can be different for different FPGA families, because the CLB architecture might differ from one to the other. However, after analyzing the architectural comparison results between Virtex-5 and 7 Series CLBs from chapter two, it turns out that CLB test for the target 7 Series hardware can be achieved with the same number of configurations. The test for the four additional FFs in a slice can be integrated into existing TCs. The following subsections explain in detail the changes made to the Virtex-5 TCs for testing 7 Series CLBs.

The additional 4 storage elements can only be configured as edge-triggered D-type FFs. The D input can be driven by the O5 output of the LUT or the BYPASS slice inputs via AX, BX, CX, or DX input. When the original four storage elements are configured as latches, they can not be used [2]. Besides the four additional storage elements, the different input ports for SR and RAM testing of SLICEM must also be considered

After carefully evaluating all the TCs for the possibility to add the tests for the additional 4 FFs, the final decision is to integrate the tests into the TCs for xor_test and xnor_test. There is no increase in the number of TCs, so the impact on test time is minimized.

The xor_test and xnor_test for the Virtex-5 are somewhat redundant. The 6-bit counters is driving the LUT, the only difference between xor_test and xnor_test is the inverted MSB bit of the input. In the modification for the 7 Series, xor_test has only one output from 5-input LUT configured as an XOR function, which drives the D input of the additional 4 FFs to test the D input path. Xnor_test tests full 6-input XNOR function as in the Virtex-5, with additional input to X input to test the BYPASS slice input to the 4 additional FFs. For xor_test, 6-input LUT XOR function is covered by 6-input xnor_test.

7 Series CLBs have an irregular numbering scheme compared to Virtex-5 CLBs and affect SLICEM Test. Especially for ram_test, the code for creating the TC has to be changed substantially, to address the numbering differences between CLBLM of the Virtex-5 and CLBLM_L/CLBLM_R of the 7 Series. Figure 3.4 and figure 3.5 illustrate the different numbering scheme of these two CLBLMs.

The TCs are developed using java programming, generated with RapidSmith framework. Chapter 5 will give an inside view of the RapidSmith tool flow for TC generation.

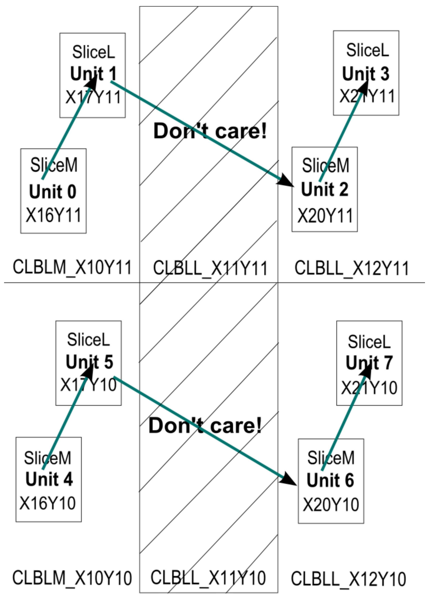


Figure 3.4: Slice arrangement of Virtex-5 CLBLM

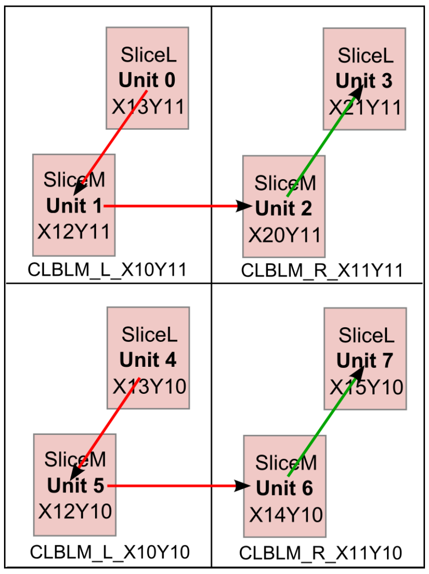


Figure 3.5: Slice arrangement of 7 Series CLBLM_L and CLBLM_R

3.2.3 Design of TPG and ORA

The TPGs generate input stimuli to the reconfigurable modules. The ORAs are used to analyze results by mutually comparing corresponding outputs of similarly configured logic under test. In previous Virtex-5 related work, the TPG and ORA for each TC were designed using java programming, this work moves the testbench design for respective TC out of Java implementation to eliminate PRET overhead. As such, TPGs and ORAs are designed using HDL (Hardware Description Language) instead. Java codes for Virtex-5 are analyzed and interpreted to create VHDL designs for TPGs and ORAs, to realize the equivalent functions. TPGs output test done signal, ORAs give error flag if there is any fault detected.

The ORAs are carried out by mutually comparing the output responses from identically configured C-testable arrays, the result of the comparison is then captured in a storage element (flip-flop) for tracing back. Since ORA is common for all TCs, the implementation is briefly described here. As illustrated in Figure 3.6 below, the outputs are compared with each other using XOR gates, when an error is detected, the last XOR gate outputs a '1' when the test is done [3].

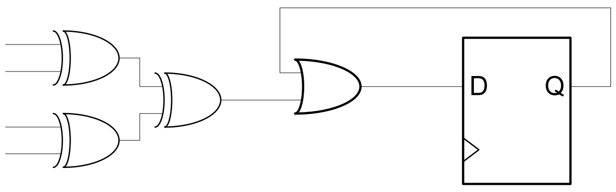


Figure 3.6: ORA for all TCs

The TPGs for all 9 TCs are outlined as follows.

3.2.3.1 TPG for xor_test and xnor_test

To test all transition faults, the LUTs are configured as XOR or XNOR. An exhaustive set of test patterns is applied for this combinational logic. The TPG to 5-input or 6-input LUT is basically implemented by a 5-bit counter or 6-bit counter respectively. The counter increases from 0 to the full value to switch each bit. The BYPASS slice inputs 'X' is implemented using toggle flip-flop to provide the transitions for single bit from "0" to "1" and from "1" to "0".

3.2.3.2 TPG for carry_test_cout_and_ff

The TPG input to both 'A4' and 'X' is from the same source, it is implemented by a toggle flip-flop as well to switch single bit.

3.2.3.3 TPG for sr_test

Both the clock enable input and the stimuli signal input are implemented using toggle flip-flops, with 180 degree phase difference between them.

3.2.3.4 TPG for ram_test

For testing memories, March tests are used [18]. Same as sr_test, ram_test is targeting on SLICEM. On the other hand, RAM testing is more complicated. To test the LUT as a 64-bit RAM, the MATS++ algorithm is used to ensure full test coverage. A 6-bit counter is used as an address input and a test pattern is applied to the data input. Table 3.1 describes the detail MATS++ algorithm.

```
MATS++ algorithm:

{↑(w0); ↑(r0,w1); ↓(r1,w0,r0)}

For i=0 to n-1
Write 0 in cell $C_i$
For i=0 to n-1
Read cell $C_i$ and check its content (0 expected)
Write 1 in cell $C_i$
For i=n-1 to 0
Read cell $C_i$ and check its content (1 expected)
Write 0 in cell $C_i$
Read cell $C_i$ and check its content (0 expected)
```

Table 3.1: MATS++ Algorithm

3.2.3.5 TPG for latch_test_cy and latch_test_o5

Toggle flip-flop is used as stimuli for ‘X’ input. These two TCs require two non-overlapping clock signals, depending on the quantity of total slices, respective clock is provided to TPG and ORA. If the total number of slices is modulo 4, then different clock sources are applied to TPG and ORA; otherwise, TPG and ORA use the same clock source.

3.2.3.6 TPG for carry_test_sum_ff and carry_test_sum_mux

Each slice contains a four-stage fast carry chain. It consists of static multiplexers and XOR cells. For both carry_test_sum_ff and carry_test_sum_mux test configurations, the same input stimulus are applied. Figure 3.7 shows how test patterns connected to ‘A’ or ‘X’ are generated, with both initial value equals to “0”.

3.2.4 Test Hardware Design

In the previous subsection we have created the TPG and ORA for each TC, here we link them up to form the test hardware to test the container. The container_interface module is the wrapper for the partially reconfigurable module, i.e., the container. Multiplexers are used for select input stimuli and output response for respective TC, they belong to the static logic. There are variations in the number of inputs and outputs for different TCs, however the container for the TC must have a common test interface. The defined container_interface consists of follows: one system clock input, two non-overlapping clock inputs (clk_0 and clk_1), one reset input, two 32-bit inputs from TPG or functional test data input, and two 32-bit outputs to the ORA or functional test result output. Refer to figure 3.8 for details.

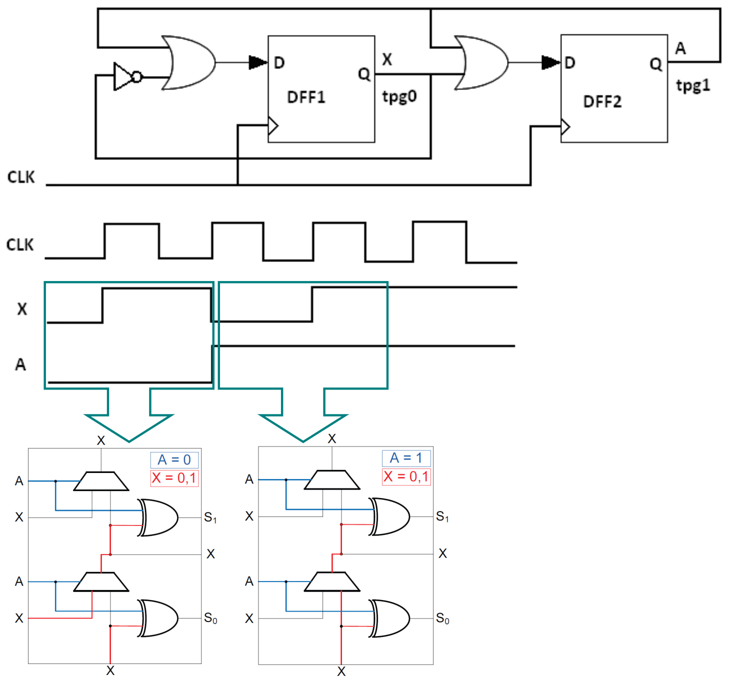


Figure 3.7: TPG for CarrySumFF and CarrySumMux

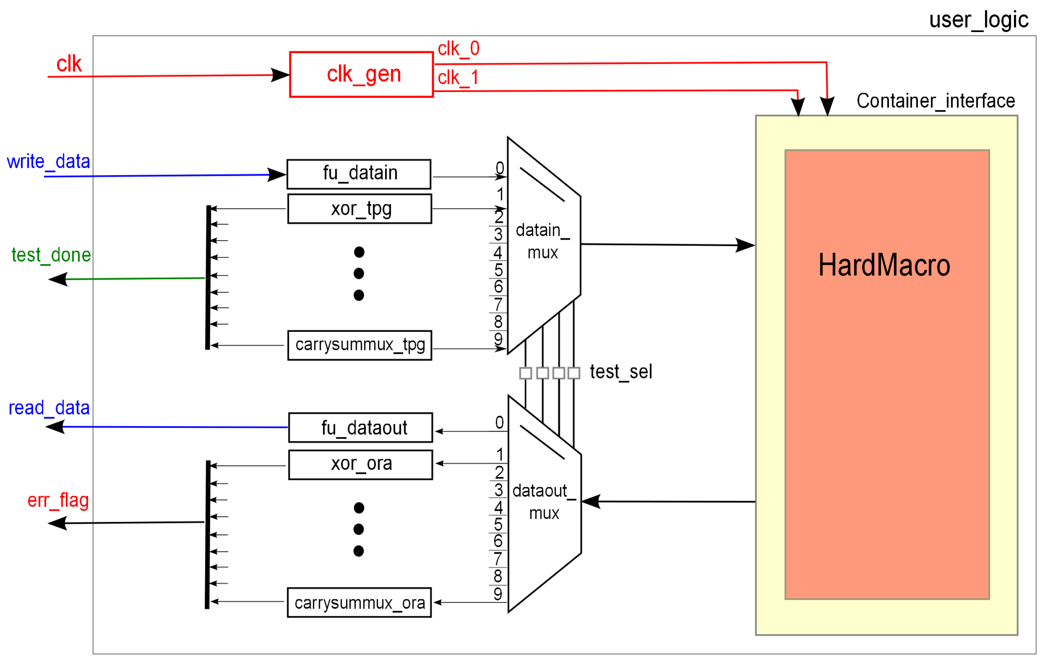


Figure 3.8: Overview of User_logic structure

The container communicates with TPG and ORA via this I/O interface. Each TC uses only lower 8 bits of one of the two 32-bit inputs to get input signals from the TPG and lower 8 bits of one of the two 32-bit outputs to output to the ORA. The container interface is therefore a superset of the signals needed by the TPGs and ORAs.

The two non-overlapping clocks for the latch tests are generated from the system clock. The clock generator is used to create the two clocks with a 25% duty cycle and 180 degree phase shift, it is assigned as a sub-module under top-level user_logic module. Figure 3.9 shows the two non-overlapping clocks generated from system input clock.

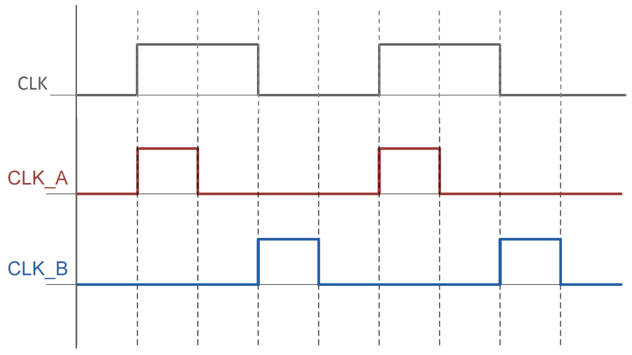


Figure 3.9: Two non-overlapping Clocks based on System clock

The port names and designators have been consolidated in Java codes to match with the port names of container_interface module. Table 3.2 shows the one-to-one matching between the port names of the TCs and the container_interface module.

clk	=> clk
clk_0	=> clk_0
clk_1	=> clk_1
rst	=> rst
en	=> ain(7)
in_tpg	=> ain(6)
in_tpg5	=> ain(5)
in_tpg4	=> ain(4)
in_tpg3	=> ain(3)
in_tpg2	=> ain(2)
in_tpg1	=> ain(1)
in_tpg0	=> ain(0)
out_ora0	=> result_lsb(0)
out_ora1	=> result_lsb(1)
out_ora2	=> result_lsb(2)
out_ora3	=> result_lsb(3)
out_oraMUX0	=> result_lsb(4)
out_oraMUX1	=> result_lsb(5)
out_oraMUX2	=> result_lsb(6)
out_oraMUX3	=> result_lsb(7)

Table 3.2: Port Map

Some errors encountered during implementation, such as “Clk port has illegal connections, this port is connected to an input buffer and other components”, “Clock buffers are lined up in series” and “Input pad net is driving non-buffer primitives”. Solutions to these kinds of problems are as follows:

1. Adding global clock buffer to the top level module, eliminate I/O buffers insertion when generating the sub-module netlist.

2. If the IBUF/OBUFs are instantiated in the sub-module, need to manually remove the input clock buffering instantiation from the sub-module and instantiate it in the top level.

The user_logic top module contains four sub-modules: clock generator, datain_mux, container_interface and dataout_mux. The datain_mux is used to select the input stimuli to the container whereas the dataout_mux is to choose the output analyzer module into which the output from container goes based on the test selection.

The sources for datain_mux are the TPG for each TC described in the last subsection. The ORA are the same for all TCs, except that TCs have variable numbers of outputs, as a consequence, the dataout_mux has to cater for the differences.

In addition to TC tests, there is an option for the functional test, which uses two 32-bit inputs and two 32-bit outputs of the container interface. The functional test tests the container when it is configured as either an inverter or an adder. The overall architecture of the user_logic module is shown in figure 3.8.

Chapter 4

Self-test Wrapper for Hybrid Reconfigurable SoC

Contents

This chapter presents the design of a self-test wrapper for the reconfigurable test hardware, elaborates the wrapper design in a hybrid SoC. The wrapper is to be connected to a standard bus interconnect, so the reconfiguration process and self-test can be triggered by the microprocessor.

4.1 Overview

To design a self-test wrapper in a hybrid reconfigurable SoC, we need to consider the type of application processor unit, the bus system that the wrapper is connected to, and the functions for the wrapper to accomplish.

4.1.1 General Descriptions

The wrapper is for the self-test hardware, which contains a Built-In Self Test (BIST) enabled reconfigurable block, and local TPG and ORA; it includes a bus protocol compatible logic, so it can be connected to a microprocessor via a standard bus interface. For our targeted hardware, we can choose either the soft-core MicroBlaze processor or one of dual Cortex-A9 hard-core processors. The bus interconnect system can be the traditional Processor Local Bus (PLB) v46 or the Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface (AXI), it will determine the later base system build. The PLB v46 is supported for the MicroBlaze processor only, while the AXI is supported for both the Cortex-A9 and MicroBlaze processors.

Through the self-test wrapper, the embedded processor can access to the reconfigurable block instantiated in the fabric, perform functional online tests to validate the bus connection and the implementation of partial reconfiguration.

TC tests can be performed offline. The configuration access port inside the FPGA allows the TCs to be fetched from memory and loaded to the the reconfigurable block, which makes it possible for fabric internal reconfiguration. The TPG generates input stimuli to the reconfigurable block, the ORA mutually compares the corresponding outputs from that block under test. The application of appropriate TPG and retrieval of ORA result can be controlled by a hardware state machine.

On the other hand, the embedded processor based online self-test is also possible, with the availability of respective TPG and ORA. This approach stores the partial configuration data in the embedded processor's program memory. The embedded processor controls and executes the self-test sequence, including reconfiguration of the resources under test with subsequent TCs and retrieval of self-test results.

4.1.2 Implementation Details

The self-test wrapper is implemented on the Xilinx Zynq-7000 Extensible Processing Platform (EPP), it combines the Processing System (PS) with the tightly integrated Programmable Logic (PL) in the same chip. The PS consists the dual ARM Cortex-A9 processors and multiple peripherals. The PL is a Artix7-based FPGA. In this thesis, we choose one of the ARM cores to be the processor and the AMBA AXI to be the bus system.

For the wrapper design, first we define and create registers to add into the user_logic module. The two most important registers are Control register and Status register. We have designed test hardware in chapter 3, together with the reconfigurable module and registers, they are integrated into the user_logic. The wrapper contains the user_logic and AMBA interface logic for being connected to the AXI interconnect, it is operating in two modes:

- Functional mode
- TC test mode

When the wrapper is operating in the functional mode, the reconfigurable block is configured with a functional TC. The processor sends two 32-bit input data through the AXI interconnect to the reconfigurable container inside the user_logic, and the two 32-bit results are read back by the processor. The purpose of functional test is to check the AXI interface connections and reconfigurability of the self-test wrapper, figure 4.1 depicts the functional test.

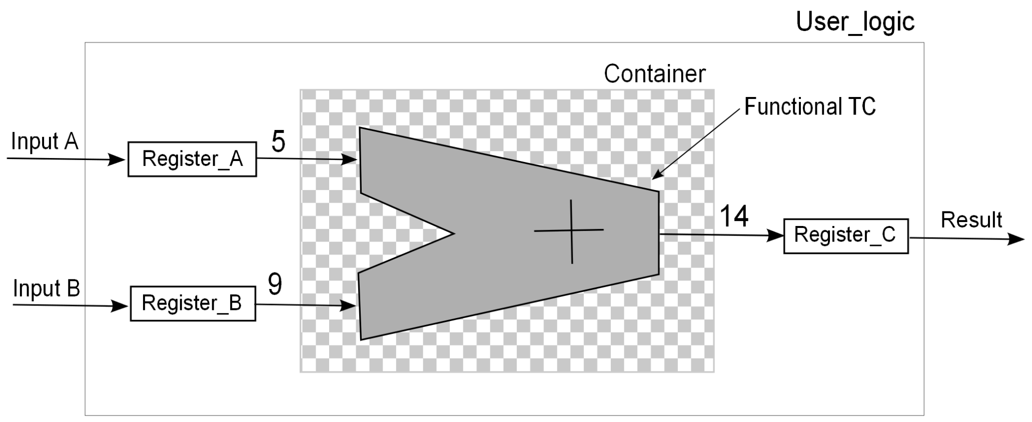


Figure 4.1: Functional Test

When the wrapper is operating in TC test mode, the reconfigurable block is configured with a specific TC, and the respective TPG and ORA are applied. When the test is done, the output from ORA is fetched by the processor through the Status register.

The embedded processor performs reconfiguration of TCs via Processor Configuration Access Port (PCAP) interface, controls test executions and retrieval of test results. Thereby, the self-test can be performed online for a fault-tolerant reconfigurable system, when fault detection is desired.

4.1.3 Wrapper Design Approaches

The subject of wrapper designs can be found in several past research literature, where the various approaches are reviewed and compared. In order to choose the most suitable methodology for our particular application, the AMBA AXI specification and several papers related to the AMBA compatible wrapper are explored.

The design of a wrapper connecting a special core to the AMBA bus is presented in [20]. By studying and comparing the two different bus protocols, the mapping relationship between them is established. Based on the outcome of analysis, the internal functional structure of the wrapper is described, which includes the ABMA initiator and target wrapper. The initiator wrapper consists of request and response machine, FIFO and AMBA master engine. The target wrapper is composed of various control logic, data paths, buffers, and various registers for dynamic configuration. This paper provides the fundamental steps in designing a wrapper.

AMBA compatible wrappers are proposed in [21], which allow the AMBA compatible IPs with different clock frequencies to be synchronized with a common clock. These wrappers use combinational circuits to avoid the latency, and are designed to minimize bus speed slow down, for instance, to insert latches or a bus arbiter. This kind of wrapper models a bus synchronized wrapper.

Article [22] describes the design methodology for a WISHBONE to AMBA interface wrapper. It illustrates the detailed interconnection signals of master and slave wrapper interface, reveals the timing analysis for data, control and address signals. And in [23], model data transaction on SoC bus with AXI4 protocol, depict the signals in a AXI4 module and read/write channels. State machine structure is explained in [24]. Last but not least, the first embedded soft core processor based FPGA BIST approach is presented in paper [25]. Some other articles address bus signal comparison, control logic and automatic generation of a wrapper as well.

All of the above provide good guidelines for designing a AMBA wrapper.

4.2 Self-test Wrapper Architecture

Designing a self-test wrapper in an embedded system could be a very complicated process, it requires more effort than a logic only FPGA design. One needs to analyze AXI bus specification, gets the hardware and software portions of an embedded design to work together. However, Zynq-7000 SoC presents a new paradigm in embedded design, combines the PS, which is built around Cortex processors, with the integrated PL. Moreover, Xilinx tool chain offers a integrated design environment with sets of development tools for the embedded system design, simplifies the design process. This work expedites the complicated design process by adding user required logic functions into the mixed wrapper design project.

In order to conduct reconfiguration during runtime, each TC packed as a partial bitstream is stored in the memory, the runtime system of the reconfigurable architecture fetches the partial bit file out of the memory and sends it over to the configuration access port. The corresponding container is then configured as a test array and ready for being tested. Figure 4.2 represents the general architecture of self-test wrapper in an embedded processor system with a partially reconfigurable container.

ARM embedded processors, AMBA bus system with the available AMBA compatible IP cores is today's most popular SoC architecture. In our design case, we have the following particulars:

- CPU	:	ARM Cortex-A9 processor
- BUS	:	AMBA AXI
- Wrapper	:	Self-test

4.3 Design of Self-test Wrapper

This section provides the detailed self-test wrapper design in a SoC system and the implementation methodology.

Take the advantage of the processor-centric platform, we create a specific wrapper design in the Xilinx Integrated Software Environment (ISE). The embedded version of ISE can be used for the Zynq-7000 EPP architectural development; for software development, a Zynq-based Software Development Kit (SDK) project needs to be built. With the available AMBA compatible IP cores, the design can achieve maximum design reuse and save development time [9]. ISE design suite 14 introduces

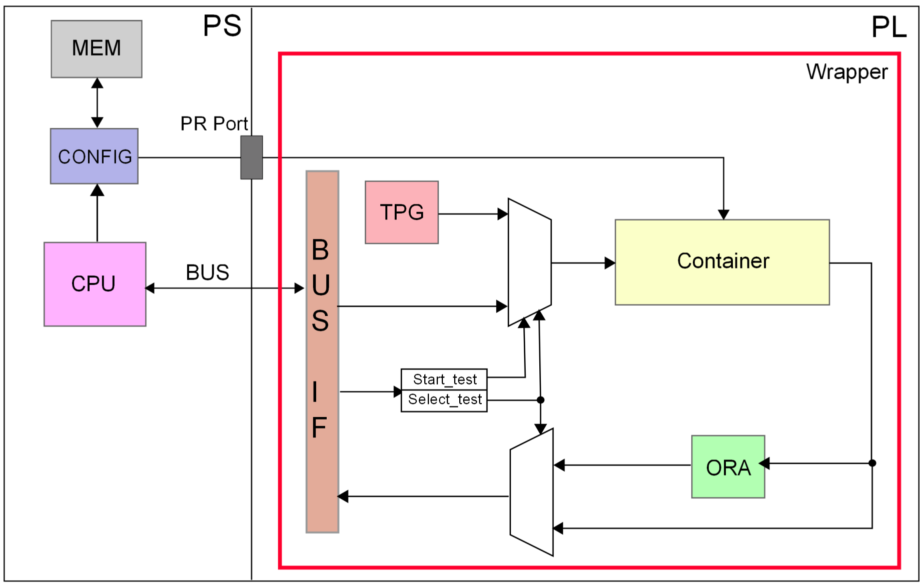


Figure 4.2: Architecture of Self-test Wrapper

Partial Reconfiguration (PR) support for the Zynq-7000 device, the standard PR flow generates the partial bitstreams, and the configuration access port on device is used to load the partial bitstream. PlanAhead is used as the tool to synthesize and place and route the design, it enables the management of reconfigurable modules and provides an easy way to floorplan.

4.3.1 ARM

ARM stands for Advanced RISC Machine. The ARM architecture describes a RISC-based processor family. It was first developed in the 1980s, now it becomes the main stream of 32-bit instruction set architecture. ARM-based processors and SoCs are used in the market for smartphones, digital televisions, and mobile computers, etc.

When designing a computer, the RISC-based ARM processors require significantly fewer logic elements, compared with traditional processors. The advantages of ARM processors are lower cost, less heat dissipation and lower power consumption, which makes them suitable for use in light, portable, battery-powered devices such as smartphones and tablet computers. The reduced complexity allows the developers to design and build a low-power SoC for an embedded system, which comprises memory, interfaces, videos and more.

ARM provides 16-bit and 32-bit embedded RISC microprocessor solutions, current widely adopted ARM cores use 32-bit instructions with 32-bit address space. Recently, ARM has included 64-bit architecture versions.

4.3.2 AMBA AXI Interconnect

The AMBA protocol is a widely-used open bus standard proposed by ARM, it specifies on-chip interconnect. 10 years ago, ARM announced the release of the AXI standard, a protocol that is part of the AMBA specifications. The goal is to meet the requirements of high performance and complex SoC designs. The AXI Protocol represents the third major generation of AMBA protocols which includes APB, AHB and AXI. The AXI specification describes an interface and communications protocol, it is a point-to-point interconnect architecture, not a bus. Broad IPs are available for AXI interconnect. AMBA 4 standard introduces new AXI4 protocol with three variances:

- Enhanced Interface Performance - AXI4-Full
- Streaming - AXI4-Stream
- Lightweight - AXI4-Lite

In this work, the AXI4-Lite version of protocol is chosen, because it is simple and sufficient for the design purpose. In order to connect the self-test Wrapper to AXI interconnect as a slave, first we have to know the signal transactions on the AXI4-Lite interface.

ARM defines the functionality and signal requirements of AXI4-Lite components. The key aspects of AXI4-Lite operation [5]) are as follows:
- All transactions are with burst length of 1, data interleaving is not supported.
- All transactions are of the same width as the data bus.
- All data accesses use a fixed data bus width, either 32-bits or 64-bits, with 32-bit mostly used.
- All accesses are non-modifiable, non-bufferable, no exclusive accesses.

Table 4.1 shows the required signals on an AXI4-Lite interface:
(* The AXI4-Lite interface does not fully support RRESP, BRESP)

Global	Write address channel	Write data channel	Write response channel	Read address channel	Read data channel
ACLK	AWVALID	WVALID	BVALID	ARVALID	RVALID
ARESETn	AWREADY	WREADY	BREADY	ARREADY	RREADY
–	AWADDR	WDATA	BRESP	ARADDR	RDATA
–	AWPROT	WSTRB	–	ARPROT	RRESP

Table 4.1: AXI4-Lite interface signals (from [5])

The AXI4-Lite protocol supports WSTRB (write strobes). This means registers can be implemented with different sizes, and memory structures that require 8-bit and 16-bit accesses are also supported.

The master interfaces and interconnect components must provide correct write strobes, the slave component can choose whether to use the write strobes or not. The permissible options are [5]:
- Fully use of the write strobes
- Ignore the write strobes and treat all write accesses with the full data bus width
- Detect write strobe that is not supported and responses with an error
- Slaves in the memory map support only a limited write strobe option.

AXI4-Lite requires that all transactions are in sequential order, and every access has its own fixed ID value. Optionally, AXI4-Lite also supports multiple outstanding transactions, but a slave can restrict this by using proper handshake signals. In addition, an AXI4-Lite slave can optionally support AXI ID signals, so that it can be connected to a AXI4-Full interface without modification [5].

4.3.3 Xilinx AXI4-Lite Interconnect

In this work, the design is based on an evaluation board from Xilinx. It is imperative to have a look at specific Xilinx AXI interface. The Xilinx AXI interfaces conform to the AMBA AXI4 version, include the subset of AXI4-Lite control register interface. AXI4-Stream transfers are not supported. The AXI4-Lite interconnect is intended for memory-mapped transfers only, has no burst transfer, with traditional 32-bits width for both interface data and address [26]. Figure 4.3 and figure 4.4 exhibit simple read and write communication from AXI4-Lite master to AXI4-Lite slave, and handshake signals between them.

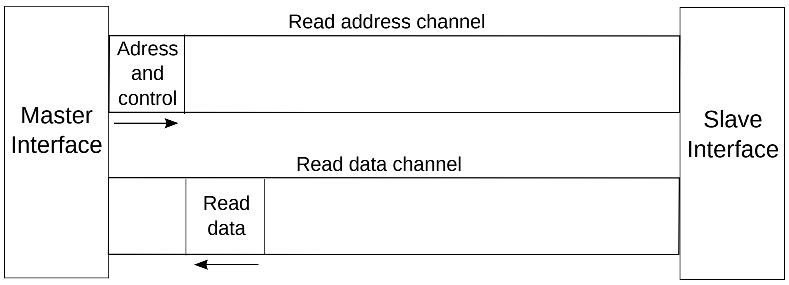


Figure 4.3: AXI4-Lite Read

The AXI interconnect core connects the memory-mapped master devices to several memory-mapped slave devices. The Xilinx Platform Studio (XPS) tool flow provides access to features of AXI Interconnect core for the embedded designs. In Zynq SoC architecture, the PS provides hardware access to memories and communication to its peripherals. This allows the PS to be the master device and can operate on its own without powering up or configuring the PL. The Self-test Wrapper belongs to the PL, it is connected to AXI interconnect as a slave device through the AXI4-Lite slave interface. In addition, the slave interfaces include a direct link to on-chip memory and AXI Hardware Internal Configuration Access Port (HWICAP) as well.

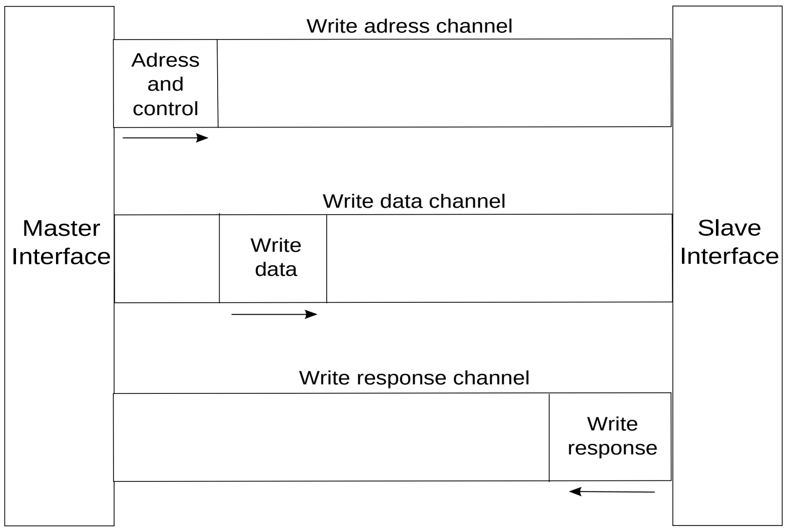


Figure 4.4: AXI4-Lite Write

4.3.4 Definition of Registers

The first thing to start the embedded wrapper design is to determine the number of registers required. After carefully reviewing the interface requirements and TCs needed to be loaded to the partial reconfiguration partition, we decide to have six 32-bit software accessible registers to be included in the user_logic. The addresses of the internal registers inside the user_logic are offset from the base address C_BASEADDR, start with 0x66E00000. At reset, all registers hold the value of 0. The self-test wrapper internal register set is described in table 4.2 below:

C_BASEADDR + Address	Register Name	Access	Default Value	Description
0x00	Write Register 1	Read/Write	0x0	Data write register 1
0x04	Write Register 2	Read/Write	0x0	Data write register 2
0x08	Result_LSB	Read	0x0	Output data register 1
0x0C	Result_MSB	Read	0x0	Output data register 2
0x10	Control Register	Read/Write	0x0	Control register
0x14	Status Register	Read	0x0	Status register

Table 4.2: Internal Registers

Write Register 1

The Write Register 1 (WR1) is a 32-bit register, the bit definitions are shown in table 4.3 with the offset from C_BASEADDR value and accessibility of this register.

Bit	Mnemonic	Access	Code Equivalent
31 – 0	WR1	R/W	slv_reg0

Table 4.3: Write Register 1 (C_BASEADDR + 0x00)

Write Register 2

The Write Register 2 (WR2) is a 32-bit register, the bit definitions are shown in table 4.4 with the offset from C_BASEADDR value and accessibility of this register.

Bit	Mnemonic	Access	Code Equivalent
31 – 0	WR2	R/W	slv_reg1

Table 4.4: Write Register 2 (C_BASEADDR + 0x04)

Result_LSB Register

The Result_LSB Register (RL) is a 32-bit register, the bit definitions are shown in table 4.5 with the offset from C_BASEADDR value and accessibility of this register.

Bit	Mnemonic	Access	Code Equivalent
31 - 0	RL	R	slv_reg2

Table 4.5: Result_LSB Register (C_BASEADDR + 0x08)

Result_MSB Register

The Result_MSB Register (RM) is a 32-bit register, the bit definitions are shown in table 4.6 with the offset from C_BASEADDR value and accessibility of this register.

Bit	Mnemonic	Access	Code Equivalent
31 - 0	RM	R	slv_reg3

Table 4.6: Result_MSB Register (C_BASEADDR + 0x0C)

Control Register

The Control Register (CR) is a 32-bit register, the bit definitions are shown in table 4.7 with the offset from C_BASEADDR value and accessibility of this register.

Bit	Mnemonic	Access	Description
31:5	Reserved	N/A	Reserved bits
4:1	MUX_ctrl	R/W	Mux control for test selection
0	Start	R/W	Start bit to enable test

Table 4.7: Control Register (C_BASEADDR + 0x10) = slv_reg4

Status Register

The Status Register (SR) is a 32-bit register, the bit definitions are shown in table 4.8 with the offset from C_BASEADDR value and accessibility of this register.

Bit	Mnemonic	Access	Description
31:2	Reserved	N/A	Reserved bits
1	Flag	R	Error flag bit, ‘1’ = error encountered
0	Done	R	Bit to indicate test done, ‘1’ = done

Table 4.8: Status Register (C_BASEADDR + 0x14) = slv_reg5

4.3.5 Wrapper Design Process

In order to create a base system with an embedded processor, first we have to create a new RTL project in PlanAhead, add processing sub-system to it. Having ZedBoard as the evaluation board, the ZedBoard definition file needs to be downloaded; it is an XML (EXtensible Markup Language) file, contains the information needed to configure the ARM sub-system.

The Base System Builder (BSB) wizard within XPS is utilized to create the base system. Since the ARM processing sub-system uses the AXI standard to communicate to its peripherals, we choose AXI based design and the default peripherals to create the design, and select ZedBoard as the base configuration board. A new XPS project is then created for the PS sub-system which holds the name as what we provide to it. Afterwards, the ARM processor is added to the design by importing the ZedBoard definition XML file, the new configuration is then loaded in XPS. The PS sub-system is eventually created. A graphical representation of the PS within the Zynq device is illustrated in figure 4.5, this is the screenshot taken from Xilinx XPS.

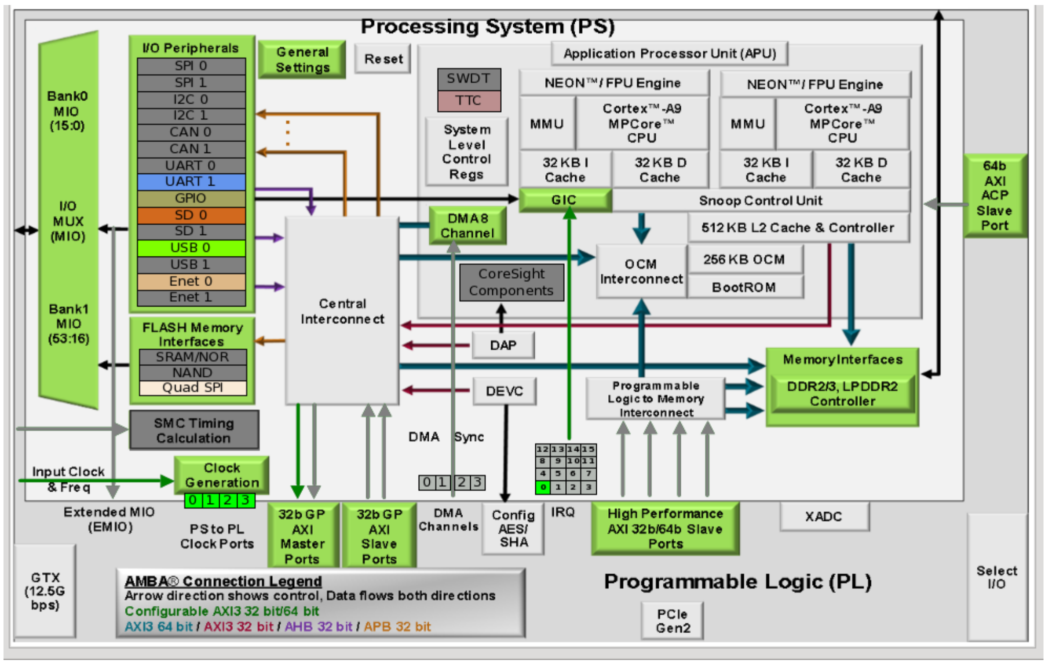


Figure 4.5: Zynq System Assembly View

Running the Design Rule Check (DRC) allows us to make sure there is no errors up to this moment. When exit from XPS, we will go automatically back to PlanAhead.

The Create Top HDL option in PlanAhead lets us generate the RTL netlist for the PS sub-system. The tool creates a VHDL file for the processor module, this is the wrapper to instantiate the XMP (Xilinx Microprocessor Project) file that describes the PS. Up to now, we have the base system with only the processor sub-system instantiated, there is nothing from the PL portion of the Zynq device included in the code. At this step, the Generate Bitstream option in PlanAhead can be used to create the bit file for describing the PS only sub-system.

To create the self-test wrapper as an AXI4-Lite compliant slave peripheral of the PS sub-system, we use the Create or Import Custom Peripheral (CIP) wizard in XPS. In this work, we have decided to include 6 registers in the user_logic to cater for the two functional tests (adder and inverter) and TC tests. We set the AXI4-Lite as the interface for the ARM core to talk to the wrapper peripheral. Three architectural modules are created through that, they are combined together to form the top-level self-test wrapper design. These are:

1) AXI4-Lite IPIF module: Connect the wrapper peripheral to AXI interconnect
2) Soft_Reset module: Reset the peripherals via software control
3) User_logic module: Allow software to access the registers within the peripheral

In order to accomplish the design, the files created in the XPS have to be modified accordingly to suit the user's required functionality. The two particular HDL templates have to be modified, one is user_logic.vhd, the other is selftest_wrapper.vhd. They are located at VHDL directory of selftest_wrapper folder. The user_logic connects to the self-test wrapper peripheral through the AXI4-Lite slave interface module. The user_logic module is equivalent to the 'Custom Functional Block', while the selftest_wrapper module is equivalent to the 'AXI4-Lite slave'. The self-test wrapper connects to AXI4-Lite interconnect with Intellectual Property Interface (IPIF), which provides a point-to-point two directional connection. The user_logic interfaces with the self-test wrapper by the Intellectual Property Interconnect Interface (IPIC_IF) module, which comprises logic to acknowledge the write and read transactions initiated by the selftest_wrapper module. The self-test wrapper block diagram is shown in figure 4.6, it is based on a diagram in [16].

The user_logic module is the primary focus in this work. In the further partial reconfiguration design flow, it will be modified for our specific application. For right now, we do not change it. The reset input to the reconfigurable part is a software reset, generated by a soft_reset block inside the self-test wrapper peripheral. The software reset is necessary, since it is used to reset the reconfigurable logic after reconfiguration. When there is a need to connect a port to the other parts on board, such as buttons and LEDs, the parameters in the selftest_wrapper module have to be modified as well. Besides, the MPD (Microprocessor Peripheral Definition) file must be updated to reflect these changes.

The AXI4-Lite slave interface communicates with the user_logic through the bus protocol ports listed in table 4.9.

The ARM processor acts as an AXI master, it sends command to the its AXI slave peripheral by writing and reading values to the defined registers inside the user_logic module. The self-test wrapper is the the AXI slave. As an example, the processor writes to write_registers to provide the input data for the functional test, and read back from result_registers to get the answers. In order to test and debug design, software application programs need to be developed using SDK. Below lists

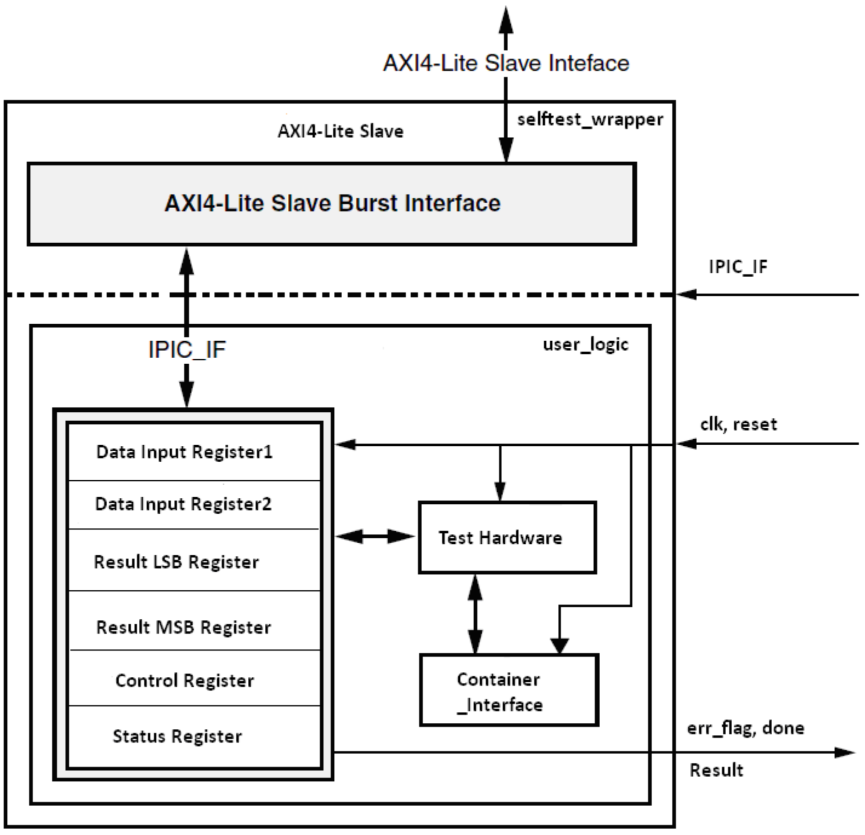


Figure 4.6: The Self-test Wrapper Block Diagram

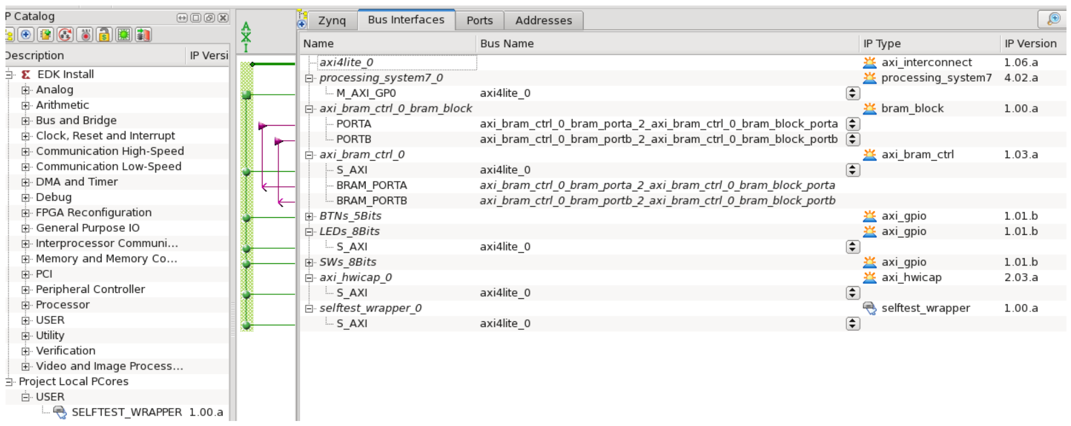
Bus protocol ports	Description
Bus2IP_Clk	AXI4_Lite to user_logic clock
Bus2IP_Resetn	AXI4_Lite to user_logic reset
Bus2IP_Addr	AXI4_Lite to user_logic address bus
Bus2IP_CS	AXI4_Lite to user_logic chip select
Bus2IP_RNW	AXI4_Lite to user_logic read/not write
Bus2IP_Data	AXI4_Lite to user_logic data bus
Bus2IP_BE	AXI4_Lite to user_logic byte enables
Bus2IP_RdCE	AXI4_Lite to user_logic read chip enable
Bus2IP_WrCE	AXI4_Lite to user_logic write chip enable
IP2Bus_Data	User_logic to AXI4_Lite data bus
IP2Bus_RdAck	User_logic to AXI4_Lite read transfer acknowledgement
IP2Bus_WrAck	User_logic to AXI4_Lite write transfer acknowledgement
IP2Bus_Error	User_logic to AXI4_Lite error response

Table 4.9: User_logic bus protocol ports

the relevant information abstracted from the CIP tool, which are useful for the rest of design:

top name : selftest_wrapper
top entity : selftest_wrapper.vhd
user logic : user_logic.vhd
type : AXI4-LITE slave
Address block:
user_logic slv : C_BASEADDR + 0x00000000 - C_BASEADDR + 0x000000FF
soft_reset : C_BASEADDR + 0x00000100 - C_BASEADDR + 0x000001FF
Driver source : /sources_1/edk/proc_module/drivers/selftest_wrapper_v1_00_a/src
header : selftest_wrapper.h
source : selftest_wrapper.c
selftest : selftest_wrapper_selftest.c

After the self-test wrapper peripheral is created and connect to the AXI4-Lite interconnect, we add AXI HWICAP peripheral and connect it via the AXI4-Lite bus as well. The HWICAP core provides the interface to transfer bitstreams through the Internal Configuration Access Port (ICAP) for PR. Figure 4.7 shows the peripherals and bus connections in the base system, it is the screenshot taken from Xilinx XPS.



Description	IP Versi
EDK Install	
Analog	
Arithmetic	
Bus and Bridge	
Clock, Reset and Interrupt	
Communication High-Speed	
Communication Low-Speed	
DMA and Timer	
Debug	
FPGA Reconfiguration	
General Purpose IO	
Interprocessor Communi...	
Memory and Memory Co...	
PCI	
Peripheral Controller	
Processor	
USER	
Utility	
Verification	
Video and Image Process...	
Project Local PCores	
USER	
SELFTEST_WRAPPER	1.00.a

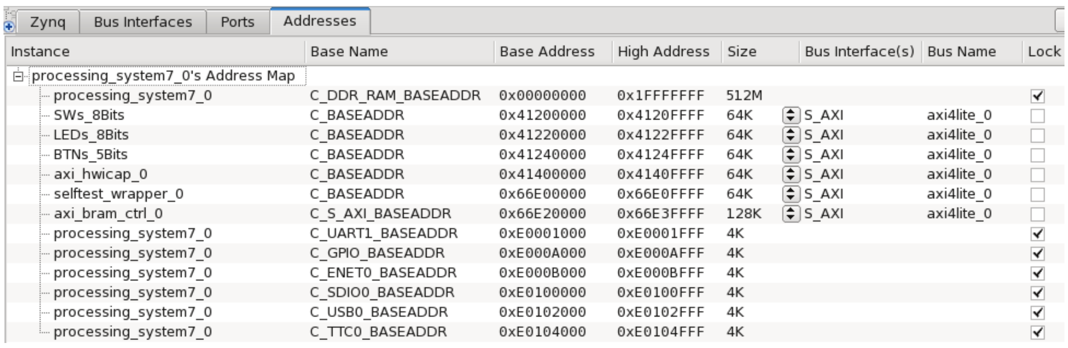
Name	Bus Name	IP Type	IP Version
axi4lite_0		axi_interconnect	1.06.a
processing_system7_0		processing_system7	4.02.a
M_AXI_GP0	axi4lite_0		
axi_bram_ctrl_0_bram_block		bram_block	1.00.a
PORTA	axi_bram_ctrl_0_bram_porta_2_axi_bram_ctrl_0_bram_block_porta		
PORTB	axi_bram_ctrl_0_bram_portb_2_axi_bram_ctrl_0_bram_block_portb		
axi_bram_ctrl_0		axi_bram_ctrl	1.03.a
S_AXI	axi4lite_0		
BRAM_PORTA	axi_bram_ctrl_0_bram_porta_2_axi_bram_ctrl_0_bram_block_porta		
BRAM_PORTB	axi_bram_ctrl_0_bram_portb_2_axi_bram_ctrl_0_bram_block_portb		
BTNs_5Bits		axi_gpio	1.01.b
LEDs_8Bits		axi_gpio	1.01.b
S_AXI	axi4lite_0		
SWs_8Bits		axi_gpio	1.01.b
axi_hwicap_0		axi_hwicap	2.03.a
S_AXI	axi4lite_0		
selftest_wrapper_0		selftest_wrapper	1.00.a
S_AXI	axi4lite_0		

Figure 4.7: Bus Connections

When the selected peripherals are added to the system and interface connections are established, the addresses are assigned automatically. Figure 4.8 shows the address map of the instances, it is the screenshot taken from Xilinx XPS. One can see from it that the self-test wrapper has a C_BASEADDR of 0x66E00000.

The system clock is the PL fabric clock FCLK_CLK0, the default value is 100 MHz, it can be changed. The PL is running at this single clock speed. The connected clock, ports and their direction are shown in figure 4.9, it is the screenshot taken from Xilinx XPS.

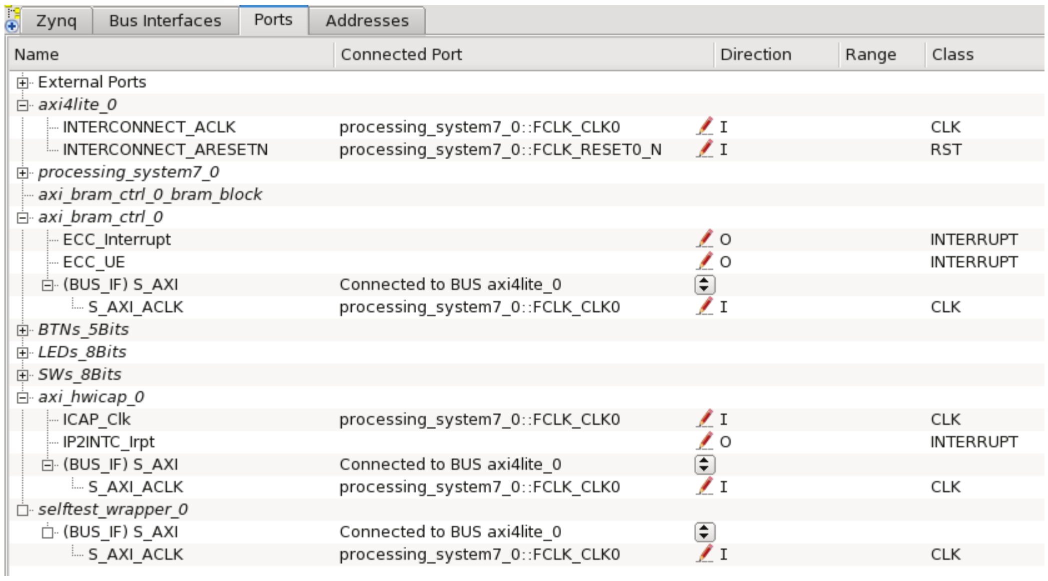
We perform another DRC in XPS to check that the system design is correct at this point. If the console window shows design is done with 'no error', denotes that



Zynq | Bus Interfaces | Ports | Addresses

Instance	Base Name	Base Address	High Address	Size	Bus Interface(s)	Bus Name	Lock
processing_system7_0's Address Map							
processing_system7_0	C_DDR_RAM_BASEADDR	0x00000000	0x1FFFFFFF	512M			✓
SWs_8Bits	C_BASEADDR	0x41200000	0x4120FFFF	64K	S_AXI	axi4lite_0	☐
LEDs_8Bits	C_BASEADDR	0x41220000	0x4122FFFF	64K	S_AXI	axi4lite_0	☐
BTNs_5Bits	C_BASEADDR	0x41240000	0x4124FFFF	64K	S_AXI	axi4lite_0	☐
axi_hwicap_0	C_BASEADDR	0x41400000	0x4140FFFF	64K	S_AXI	axi4lite_0	☐
selftest_wrapper_0	C_BASEADDR	0x66E00000	0x66E0FFFF	64K	S_AXI	axi4lite_0	☐
axi_bram_ctrl_0	C_S_AXI_BASEADDR	0x66E20000	0x66E3FFFF	128K	S_AXI	axi4lite_0	☐
processing_system7_0	C_UART1_BASEADDR	0xE0001000	0xE0001FFF	4K			✓
processing_system7_0	C_GPIO_BASEADDR	0xE000A000	0xE000AFFF	4K			✓
processing_system7_0	C_ENET0_BASEADDR	0xE000B000	0xE000BFFF	4K			✓
processing_system7_0	C_SDIO0_BASEADDR	0xE0100000	0xE0100FFF	4K			✓
processing_system7_0	C_USB0_BASEADDR	0xE0102000	0xE0102FFF	4K			✓
processing_system7_0	C_TTC0_BASEADDR	0xE0104000	0xE0104FFF	4K			✓

Figure 4.8: Address Map



Zynq | Bus Interfaces | Ports | Addresses

Name	Connected Port	Direction	Range	Class
External Ports				
axi4lite_0				
INTERCONNECT_ACLK	processing_system7_0::FCLK_CLK0	I		CLK
INTERCONNECT_ARESETN	processing_system7_0::FCLK_RESET0_N	I		RST
processing_system7_0				
axi_bram_ctrl_0_bram_block				
axi_bram_ctrl_0				
ECC_Interrupt		O		INTERRUPT
ECC_UE		O		INTERRUPT
(BUS_IF) S_AXI	Connected to BUS axi4lite_0			
S_AXI_ACLK	processing_system7_0::FCLK_CLK0	I		CLK
BTNs_5Bits				
LEDs_8Bits				
SWs_8Bits				
axi_hwicap_0				
ICAP_Clk	processing_system7_0::FCLK_CLK0	I		CLK
IP2INTC_Irpt		O		INTERRUPT
(BUS_IF) S_AXI	Connected to BUS axi4lite_0			
S_AXI_ACLK	processing_system7_0::FCLK_CLK0	I		CLK
selftest_wrapper_0				
(BUS_IF) S_AXI	Connected to BUS axi4lite_0			
S_AXI_ACLK	processing_system7_0::FCLK_CLK0	I		CLK

Figure 4.9: Ports and Clock Connection

the design is correct. The Generate Bitstream option allows us to create the bit file for reflecting the PS sub-system and its peripherals, including the self-test wrapper.

4.3.6 Wrapper Communication

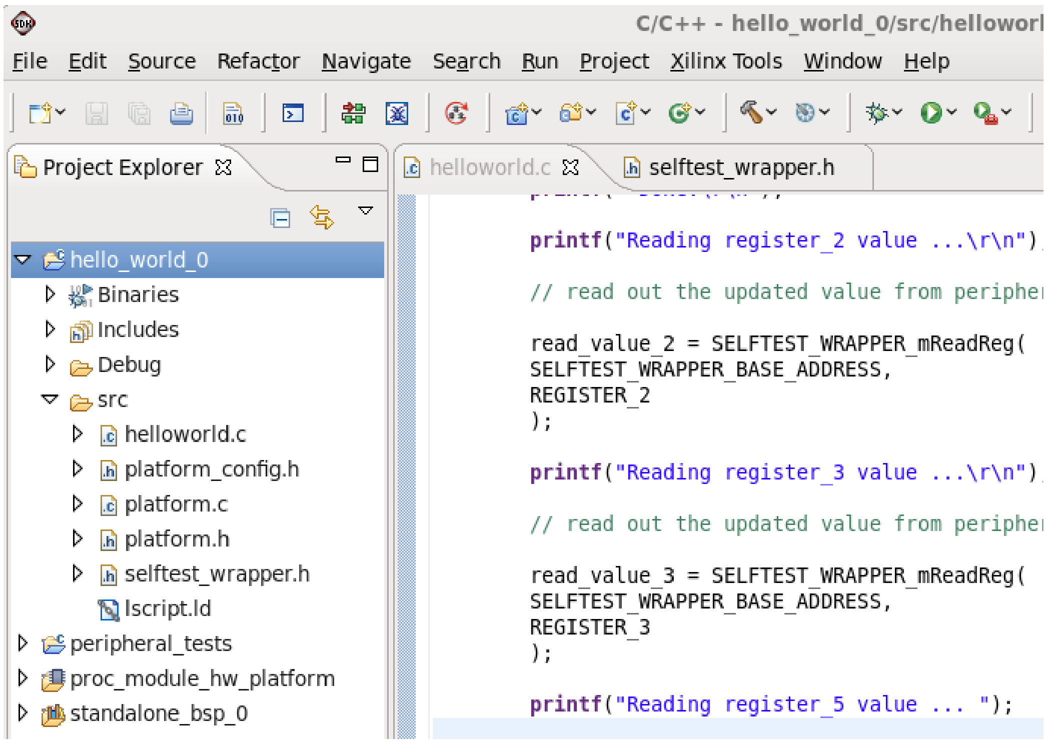
Having created the custom peripheral of the PS, i.e., the self-test wrapper, naturally, we want to talk to our wrapper peripheral. For doing that, we need to launch a software project.

At the last step of creating the base system with ARM processor in PlanAhead, the bitstream for the PS hardware platform is generated and is ready to be exported to SDK. We launch SDK with the exported hardware information. SDK window is loaded with the embedded processor PlanAhead bit file output.

We need to create a new standalone Board Support Package (BSP) to be used by the tools to interface to our hardware. We select one of the ARM cores for the code to run on, and the libraries we need that are included in the SDK tools. We pick the default Hello_World template to create a new C project, targeting the created BSP.

The selftest_wrapper.h located in the PS drivers folder, that is the source directory of self-test wrapper, is needed for the software program. We import selftest_wrapper.h to the source folder of the SDK project. This is a simple driver that handles the communication to the self-test wrapper hardware peripheral through registers.

Now a new C program is created and it can be modified with the included driver to suit for our special application, such as executing the registers reading and writing functions within it. Figure 4.10 shows the SDK project explorer, it is the screenshot taken from Xilinx SDK. Chapter 6 will describe the detailed software development flow.



```
printf("Reading register_2 value ...\r\n")

// read out the updated value from periphe

read_value_2 = SELFTEST_WRAPPER_mReadReg(
SELFTEST_WRAPPER_BASE_ADDRESS,
REGISTER_2
);

printf("Reading register_3 value ...\r\n")

// read out the updated value from periphe

read_value_3 = SELFTEST_WRAPPER_mReadReg(
SELFTEST_WRAPPER_BASE_ADDRESS,
REGISTER_3
);

printf("Reading register_5 value ... ");
```

Figure 4.10: SDK Project Explorer

Chapter 5

Tool Flow

Contents

This chapter describes the flows for the tools used in this thesis. After the Test Configurations (TCs) are created in RapidSmith framework, they are imported to the Xilinx development environment. Several Xilinx development tools are utilized in this work, the embedded edition of the Integrated Software Environment (ISE) design suite comprises [27]:

- ISE tools
- PlanAhead design software
- Embedded Development Kit (EDK)

In ISE, we design the static logic and instantiate the container_interface module as a black box. The EDK is a set of tools used for a embedded processor system development, a broad range of Intellectual Properties (IPs) are available [27]. In EDK, we configure the self-test wrapper as the peripheral for the Processing System (PS). Back to PlanAhead, in a RTL project, we generate the bitstream for the PS hardware platform, and export the hardware information to Software Development Kit (SDK) and launch a software project. Then, we follow the Partial Reconfiguration (PR) design flow in PlanAhead to generate full and partial bitstreams to dynamically reconfigure the area under test. Finally we test both hardware and software design in a evaluation board. Figure 5.1 depicts the overall flow chart for the entire design.

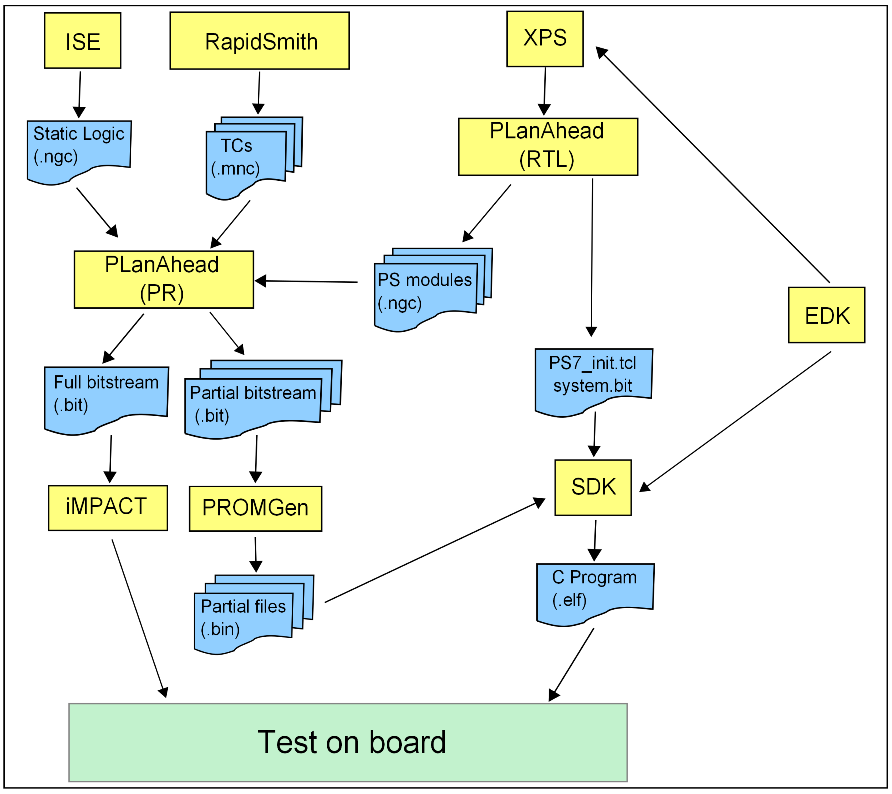


Figure 5.1: Overall design Flow Chart

5.1 RapidSmith for TC Generation

The implementation of a structural test for CLBs requires low level access to the FPGA circuitry and generates TCs. A suitable implementation platform is needed to achieve this objective. Xilinx provides the Xilinx Design Language (XDL) which is in a human readable format. It describes the configuration and placement information of each building block (primitive), as well as the interconnections between these instances [12]. The XDL is equivalent to the more widely used Netlist Circuit Description (NCD) format, both represent a Xilinx design in a development status from un-placed and un-routed to fully placed and routed [12]. Xilinx also provides an executable script to convert XDL to the NCD format, and vice versa [28]. The provided information in a XDL file is sufficient to have low level access to the internal structure of the FPGA and to configure the CLB subcomponents in a proper way for performing tests [3].

In addition, the RapidSmith design tool provides a framework for low level FPGA circuit manipulation performed in XDL. In RapidSmith, each device structure is specified in a unique XDLRC file, which contains the subcomponents, input and output ports as well as the interconnection between them. The XDLRC file is a human readable file as well, it has huge file size, provides extreme detail information on the structure of a specific Xilinx FPGA. RapidSmith can parse and extract device information from it, compact it and generate much smaller files out of it. These compact files represent the basic elements form a particular FPGA and can be loaded and executed much more quickly. Since there is no need to perform synthesis and mapping steps, RapidSmith brings the advantages of fast design implementation and the optimized design flow [28].

5.1.1 XDL Design Flow

The first step to implement tests for the target FPGA is to get the specific XDLRC description, parse it and create from it a more compatible data formation that can be used in the design flow. This had already been done in the RapidSmith tools [12]. All of the test configurations are manipulated in XDL format and are built on the RapidSmith framework.

Both NCD and XDL files can represent a design in phase between MAP and PAR. Figure 5.2 based on [28] shows where a NCD file can be directly converted into XDL format and vice versa in the FPGA design flow. It also illustrates that RapidSmith tools use XDL as the interface to perform a rapid design implementation at different points in the design flow.

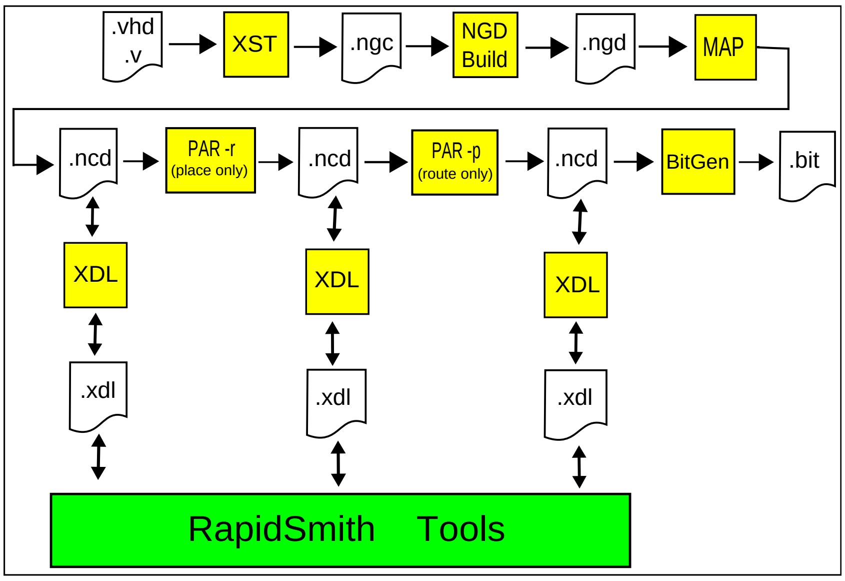


Figure 5.2: Where XDL fits in the FPGA design flow

For testing the internal logic cells of the FPGA, a set of TCs has to be first developed based on the test templates provided in RapidSmith framework. The outputs are XDL files that will be used for the generation of test configurations [3]. Manipulation of these XDL files are performed accordingly using Java programming, the eventual XDL representations depict the pre-placed and partially routed hard macros. Hard macros are conventionally defined as pre-placed and routed modules, due to the homogeneous structure of the FPGA, they can be placed virtually anywhere on the device. The hard macro methodology allows rapid design implementation.

The NCD format can not be used for hard macros; the NMC format is used instead, which is the hard macro counterpart format to NCD. In order to instantiate the hard macro in a VHDL entity with a wrapper, the XDL files have to be converted into NMC format. The same executable script is used to convert files from XDL to NMC, except that the output file name is specified with an NMC extension. The wrapper with the enclosed hard macro is then provided as an input source for the Xilinx tool, together with the static logic design, they are integrated into the design flow. The placement and routing information are contained in the XDL description specified for each unit. Finally, the full and partial bitstreams are generated for the implemented TCs, and are ready to be used for the run-time reconfigurable system.

5.1.2 TC Implementation for Target Device

The Programmable Logic (PL) of the Zynq device on the evaluation board is an Artix-7 FPGA. This device family is currently not supported by RapidSmith. RapidSmith tools depend on Xilinx XDLRC files for describing the devices. The XDLRC file for the Artix-7 has to be imported to the framework. In addition, there is no RapidSmith test templates available for testing Artix-7 CLBs in the RapidSmith library. Furthermore, there is neither documentation for XDL description nor official support from Xilinx. Lastly, some of the new features for 7 Series CLBs are not clearly documented. All of these obstacles have to be resolved to generate TCs with correct XDL syntax for Artix-7 CLBs.

To test the target subcomponents of the CLB, a proper VHDL file has to be created to instantiate the particular CLB primitive. A special Xilinx script is used to convert the generated NCD file into the XDL representation. Based on that, the correct XDL syntax information for CLB primitives is obtained, and is used for the TC implementation. FPGA editor, a graphical tool that can be used for viewing the configuration and the nets connection of interested logic elements, also provides useful data for the slice. For specific port names and naming convention of a particular configuration options, special HDL instantiations are required to get the correct syntax for configuring the slice. Xilinx documentation [29] helps to accomplish the low level configuration. Appendix A shows code segments of a Artix-7 slice low level configuration written in XDL. XDL files for all TCs have been generated and are successfully converted to hard macros in NMC format.

FPGA editor can be used to view the NMC representation of hard macro based TCs, figure 5.3 shows a snapshot of a TC taken from the FPGA Editor. Since the hard macros are black boxes instantiated in the VHDL wrapper (container_interface), there is neither real synthesis nor mapping or packing performed. The final synthesis, place and route are happened in the Xilinx PlanAhead design flow.

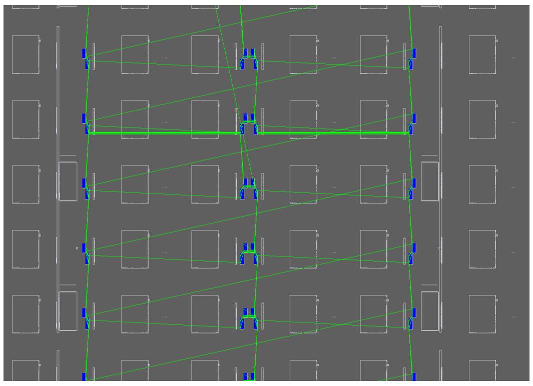


Figure 5.3: Close view of Hard Macro from FPGA editor

5.2 Xilinx Embedded Design Flow

The Xilinx EDK supports the integrated design of both hardware and software components, so that they function as a whole. To create a Zynq SoC design, the following tools are needed: ISE Design Suite, PlanAhead, Xilinx Platform Studio (XPS), SDK and iMPACT for programming.

5.2.1 EDK tool

EDK is a set of tools suited for the development of in most cases embedded processor-centric SoC designs. By comparison, the EDK approach outperforms the traditional implementation by being more efficient. If processors are deployed, naturally the task of building an embedded system can be split into two fundamental base jobs. First, the configurable hardware has to be specified in terms of embedded processors, peripherals and interconnects among its components. This hardware-centric tool component is called XPS. Its main alleviation to the designer is the graphical systems editor which ultimately generates Register Transfer Level (RTL) code from the graphical description of the SoC. The detailed configurations of the hardware components take place in XPS as well. In addition, XPS provides all the required information to set up a software project under the SDK tool. SDK is based on the open source standard Eclipse framework [27], it provides all the state of the art software development tools via plug-ins (compiler, debugger, source code revision control, etc.) to create, verify and debug embedded software. On top of this core functionality for hardware and software development, EDK also provides a tool-set for simulation of the embedded system on both behavioral and gate level. A lot of lower level FPGA tasks can also be done in EDK, like generating specific bit files for non volatile configuration devices. Moreover, hardware and software development can be done concurrently. Hence, the EDK tool suite simplifies and streamlines the embedded design, and accelerates the development speed [30].

In addition to floor planing, the PlanAhead tool plays as a rule of overall project manager, it is capable of integrating all tools. The entire design implementation flow is centrally launched by PlanAhead, after creating a RTL PlanAhead project with the embedded processor, XPS is launched to create a new sub system, specific IPs are available for hardware development. After that, the hardware design is exported to SDK to create a software project. Refer to figure 5.1 for the detail.

5.2.2 Creation of a Hardware System with Embedded Processor

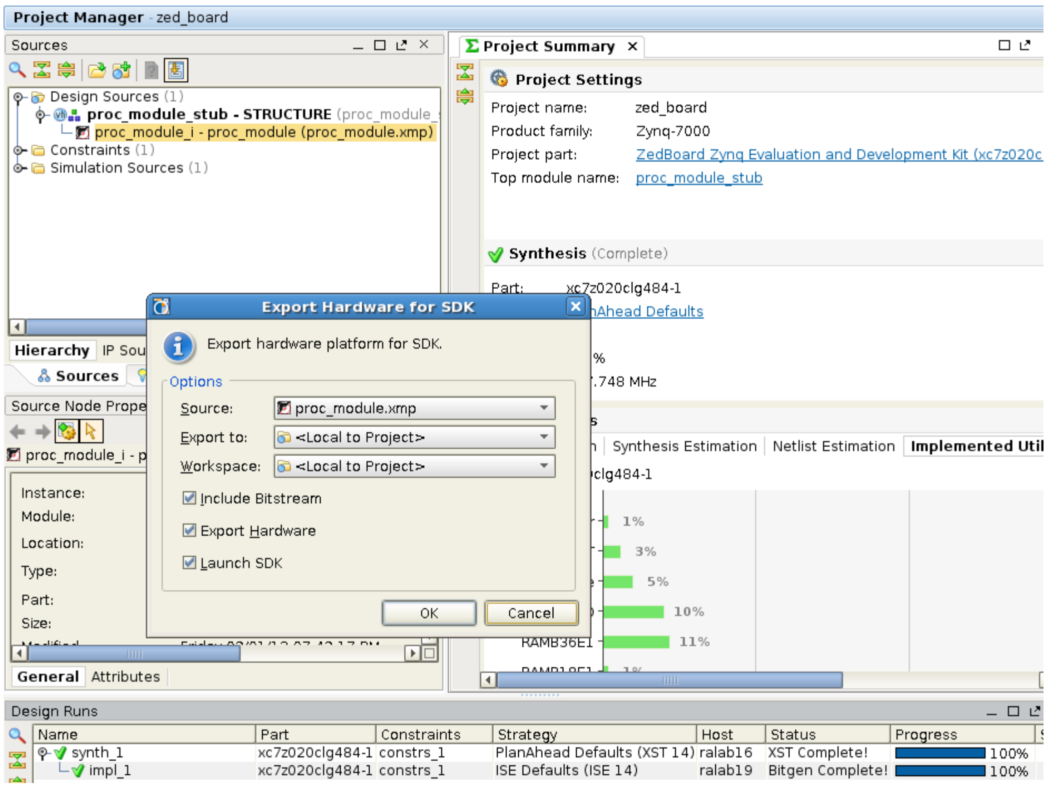
As described in the previous chapter, the PL part of the overall design is combined with the PS part within a XPS project. The new RTL project is first created in the PlanAhead, from where the XPS design environment launches. The next step is to import the correct peripheral configuration file describing the target board. The PS sub-system is then created based on this information. It is highly recommended to perform DRC at the various steps of the design flow to isolate errors early. Closing the XPS environment will bring PlanAhead back. In PlanAhead, we create a VHDL wrapper for instantiating the PS sub-system, the final bitstream for the PS sub-system is generated there as well. The bit file can be used with the iMPACT tool to configure target hardware directly, or exported to a SDK project to reflect the hardware platform.

The custom peripheral implemented in the PL of the Zynq device has to be added to the embedded system. The CIP (Create or Import Custom Peripheral) wizard in XPS is utilized for this task. It is possible to create a new peripheral or import an existing one by adding the available code to the design. The connection to the PS is achieved with the AXI bus interconnect. The proper AXI interface between the ARM Core and the custom peripheral has to be chosen, and the registers required for the peripheral have to be set. XPS generates automatically a system address map for the registers of all the peripheral components connected to the AXI interface. In addition, the associated device driver templates can be created providing an interface for the user software to the embedded system.

5.2.3 Software Project Creation

Having defined the processor sub-system, we can now create a software project that is running on the PS. In order to do so, SDK, the environment of software development, needs to be launched. For SDK to be aware of the hardware system and to generate the correct software, we need to export hardware configuration in PlanAhead to SDK. When launching SDK from PlanAhead, check all three options to include the bitstream, export the hardware, and launch SDK, as shown in figure 5.4. Once the hardware specification is changed for the project in PlanAhead, SDK project has to be re-targeted and re-built to reflect the new specification.

Upon opening the SDK project, the hardware specification is imported to SDK. The next step is to create a board support package (BSP) for the hardware. This BSP is hardware specific, it contains the peripheral drivers within the system as well as numerous C header files, such as platform.h, which defines basic functions for the implementation and platform initialization, xparameter.h defines memory



Project Manager - zed_board

Sources

- Design Sources (1)
  - proc_module_stub - STRUCTURE (proc_module
    - proc_module_i - proc_module (proc_module.xmp)
- Constraints (1)
- Simulation Sources (1)

Project Summary

Project Settings

Project name:	zed_board
Product family:	Zynq-7000
Project part:	ZedBoard Zynq Evaluation and Development Kit (xc7z020c
Top module name:	proc_module_stub

Synthesis (Complete)

Part: xc7z020clg484-1

Export Hardware for SDK

Export hardware platform for SDK.

Options

Source:	proc_module.xmp
Export to:	<Local to Project>
Workspace:	<Local to Project>

[✓] Include Bitstream

[✓] Export Hardware

[✓] Launch SDK

Design Runs

Name	Part	Constraints	Strategy	Host	Status	Progress
synth_1	xc7z020clg484-1	constrs_1	PlanAhead Defaults (XST 14)	ralab16	XST Complete!	100%
impl_1	xc7z020clg484-1	constrs_1	ISE Defaults (ISE 14)	ralab19	Bitgen Complete!	100%

Figure 5.4: Launch SDK from PlanAhead

map of the system and hardware configuration parameters, etc.

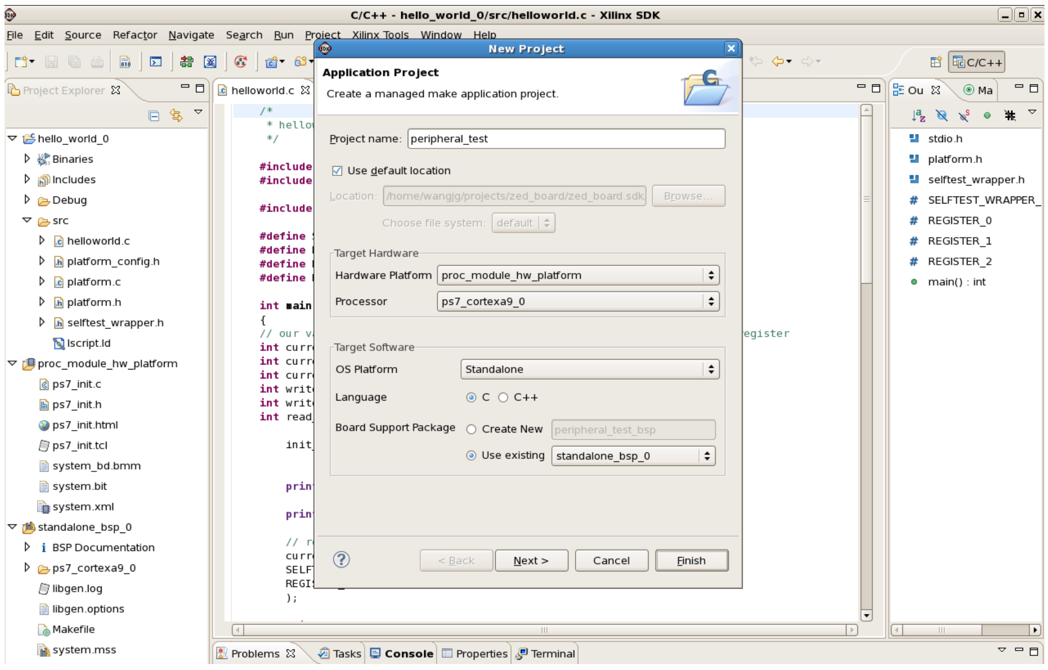
We create a new software application project by using the existing BSP for the target hardware. Figure 5.5 shows the screenshot for creating a new SDK project.

There are two choices to implement the project, either based on a bare-metal system, which is the one without operating system, or with the Linux OS. Once the SDK project is created, we can write code for the intended application. Compile the program and download the ELF (Executable and Linkable Format) file to the board to test the application. Refer to figure 5.1 as well.

5.3 Partial Reconfiguration Flow

PlanAhead tool is used to control the PR flow, it defines a Reconfigurable Partition (RP) in the reconfigurable region, creates multiple Reconfigurable Modules (RMs) and guides the PR implementation flow. At the end, full and partial bit files are generated.

The PR flow requires the input sources to PlanAhead to be netlist files in NGC format. Whereas TCs are pre-mapped and pre-placed hard macros in NMC format converted from XDL, they describe the low-level configuration of the CLB primitives. Special handling is required for TCs before they can be integrated into PlanAhead for further processing [4]. TCs are regarded as reconfigurable modules


C/C++ - hello_world_0/src/helloworld.c - Xilinx SDK
File Edit Source Refactor Navigate Search Run Project Xilinx Tools Window Help
Project Explorer
helloworld.c
Problems
Tasks
Console
Properties
Terminal

New Project

Application Project

Create a managed make application project.

Project name: peripheral_test

[✓] Use default location

Location: /home/wangjg/projects/zed_board/zed_board.sdk

Choose file system: default

Target Hardware

Hardware Platform	proc_module_hw_platform
Processor	ps7_cortexa9_0

Target Software

OS Platform	Standalone
Language	[✓ C] C++
Board Support Package	Create New peripheral_test_bsp
	[✓ Use existing] standalone_bsp_0

< Back | Next > | Cancel | Finish

Figure 5.5: Create a new Project in SDK

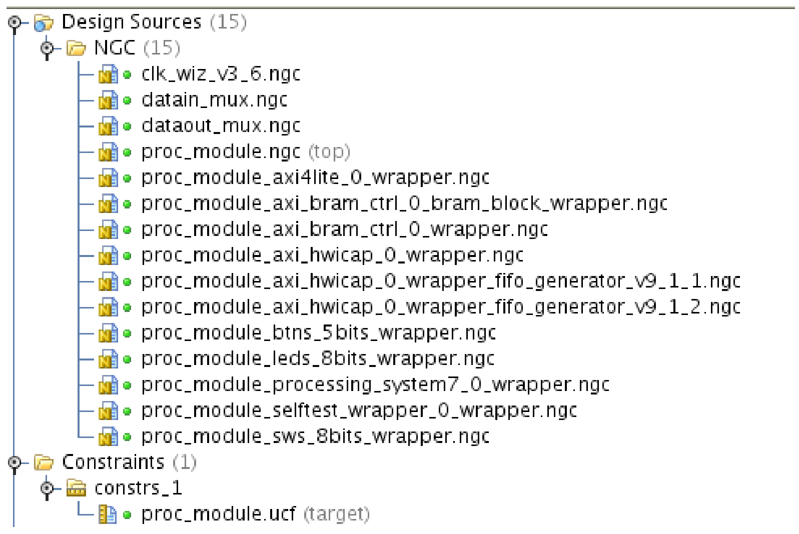
in PR flow, they need to be wrapped into a VHDL top-level entity, so that they have a common interface to other modules in the static logic, and can be instantiated with the same wrapper interface. Even though each TC has specific connections to its TPG and ORA, these discrepancies are masked by the wrapper, so that all the TCs have the common interface to the static logic of the design [4].

SDK is used to create a software application program that enables the PR. PL of the Zynq device can share memory with PS, which allows fine-grained interaction between the processor and user_logic. In the tool flow, each TC is instantiated in a separate design, and is then integrated into RP as a RM in the design. Finally a partial bitstream is generated. This section demonstrates the steps to create a PR design with a embedded processor [8].

5.3.1 Creation of PlanAhead Partial Reconfiguration Project

This time, We create a new PR project in PlanAhead, select post-synthesis project as the project type, and enable the PR option. In the RTL PlanAhead project, we have created the base system targeting ZedBoard evaluation platform. We now import all the netlist files generated from the base system build to the new PR project, and specify the top-level design. Optionally, we can create an user constrain file. Figure 5.6 shows the NGC files required by the PR project.

New modules have been added as sub-modules to the user_logic design to achieve the functions required for TCs and functional tests. The user_logic contains a reconfigurable component called ‘container_interface’, this is the VHDL wrapper



```
Design Sources (15)
  NGC (15)
    clk_wiz_v3_6.ngc
    datain_mux.ngc
    dataout_mux.ngc
    proc_module.ngc (top)
    proc_module_axi4lite_0_wrapper.ngc
    proc_module_axi_bram_ctrl_0_bram_block_wrapper.ngc
    proc_module_axi_bram_ctrl_0_wrapper.ngc
    proc_module_axi_hwicap_0_wrapper.ngc
    proc_module_axi_hwicap_0_wrapper_fifo_generator_v9_1_1.ngc
    proc_module_axi_hwicap_0_wrapper_fifo_generator_v9_1_2.ngc
    proc_module_btns_5bits_wrapper.ngc
    proc_module_leds_8bits_wrapper.ngc
    proc_module_processing_system7_0_wrapper.ngc
    proc_module_selftest_wrapper_0_wrapper.ngc
    proc_module_sws_8bits_wrapper.ngc
Constraints (1)
  constrs_1
    proc_module.ucf (target)
```

Figure 5.6: All NGC files for new PR Project

entity for all the TCs. The port map declaration of this component defines the common interface of all TCs. It is also the reconfigurable partition inside the PL.

5.3.2 Definition of Reconfiguration Partition and Region

To define a RP in PlanAhead, we set the physical size of the reconfiguration partition to include the required resources. Some overhead should be accounted for routing resources when setting the partition size. It should be bigger than the hard macro size. To obtain the best place and route results, align the RP vertically to clock region boundaries, draw a box that includes the physical location of the hard macro and contains the region from Slice X12Y0 to Slice X19Y37, as demonstrated in figure 5.7. The resulting RP physical region (Pblock area) will be stored in the primary UCF file as AREA_GROUP RANGE constraints.

Then we add the 9 TCs plus two functional TCs RMs to the RP, and run the specific DRC by selecting only Partail Reconfig option, de-select all the rest. At this step, a fatal violation was encountered:

```
Partition Reconfigurable Module ‘adder’ contains BUFGP symbol
‘clk\_ BUFGP’ that cannot be reconfigured. Please redefine your
Reconfigurable Module to remove the illegal logic.
```

Solution to this problem is: Redesign the functional test modules, avoid BUFGP during synthesize. Refer to figure 5.8. Warnings about the reconfigurable instances that do not affect the configuration process normally can be ignored.

While loading the TCs as RMs, they are treated as black boxes as there is no netlist associated with them.

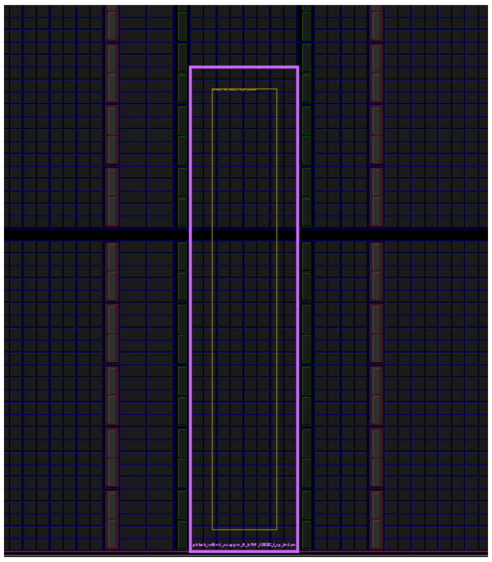


Figure 5.7: Closer view of Pblock area

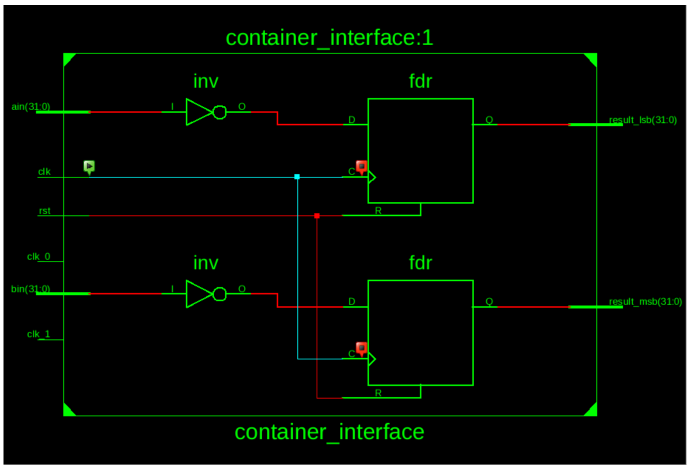


Figure 5.8: Without BUFGP

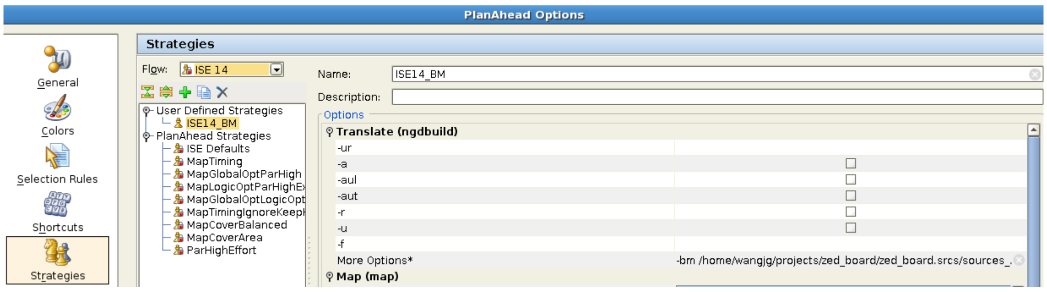
5.3.3 Creation of Configuration

When the synthesized design is opened, we need to add more option for translate (ngdbuild). The -bm option is used to point to the BMM (Block Ram Memory Map) file for the new strategy, as illustrated in figure 5.9.

When creating and implementing the inverter as the first configuration, 160 translate errors were encountered. They were caused by latches, which were synthesized on the output signals of the combinational logic, due to an incomplete assignment.

Solution: Create a default assignment to every variable and signal in the process before any of the normal functional code in the process.

After successfully implementing the first configuration, we promote it, so that the



Strategies

Flow: ISE 14

Name: ISE14_BM

Description:

Translate (ngdbuild)

-ur	
-a	☐
-aul	☐
-aut	☐
-r	☐
-u	☐
-f	
More Options*	-bm /home/wangjg/projects/zed_board/zed_board.srcs/sources_.

Map (map)

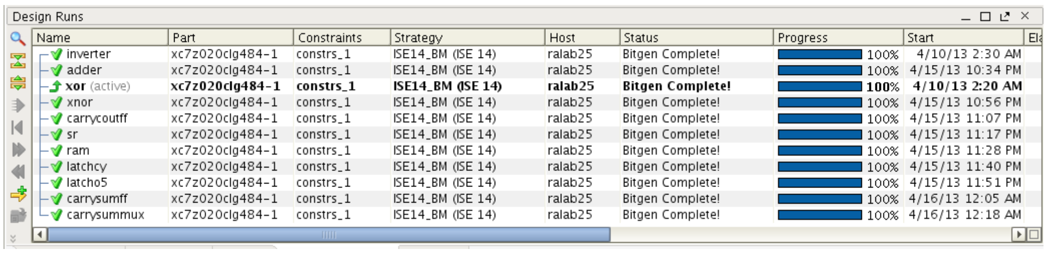
Figure 5.9: Add -bm option for Translate (ngdbuild)

implementation result of the static logic can be reused for the subsequent configurations. To create additional configurations and implement them, we select Create Runs option from Flow menu, and change the name of configuration from default to the name of added configuration. Clicking on More and repeating the steps allow us to add more runs. In the Partition Action field, by extending the Module Variant, we can select the proper variant respectively.

The PR verification utility is used to validate the implemented configurations. We employ PR verification utility to all of the implemented configurations. The utility runs and reports no errors found.

5.3.4 Bit Files Generation

As the last step, we generate bit files for the entire project. The Generate Bitstream option runs the bitstream generation process and generates full and partial bitstreams. All the bitstreams are generated successfully, figure 5.10 show the results.



Design Runs

Name	Part	Constraints	Strategy	Host	Status	Progress	Start	El
inverter	xc7z020clg484-1	constrs_1	ISE14_BM (ISE 14)	ralab25	Bitgen Complete!	100%	4/10/13 2:30 AM	
adder	xc7z020clg484-1	constrs_1	ISE14_BM (ISE 14)	ralab25	Bitgen Complete!	100%	4/15/13 10:34 PM	
xor (active)	**xc7z020clg484-1**	**constrs_1**	**ISE14_BM (ISE 14)**	**ralab25**	**Bitgen Complete!**	**100%**	**4/10/13 2:20 AM**	
xnor	xc7z020clg484-1	constrs_1	ISE14_BM (ISE 14)	ralab25	Bitgen Complete!	100%	4/15/13 10:56 PM	
carrycoutff	xc7z020clg484-1	constrs_1	ISE14_BM (ISE 14)	ralab25	Bitgen Complete!	100%	4/15/13 11:07 PM	
sr	xc7z020clg484-1	constrs_1	ISE14_BM (ISE 14)	ralab25	Bitgen Complete!	100%	4/15/13 11:17 PM	
ram	xc7z020clg484-1	constrs_1	ISE14_BM (ISE 14)	ralab25	Bitgen Complete!	100%	4/15/13 11:28 PM	
latchcy	xc7z020clg484-1	constrs_1	ISE14_BM (ISE 14)	ralab25	Bitgen Complete!	100%	4/15/13 11:40 PM	
latcho5	xc7z020clg484-1	constrs_1	ISE14_BM (ISE 14)	ralab25	Bitgen Complete!	100%	4/15/13 11:51 PM	
carrysumff	xc7z020clg484-1	constrs_1	ISE14_BM (ISE 14)	ralab25	Bitgen Complete!	100%	4/16/13 12:05 AM	
carrysummux	xc7z020clg484-1	constrs_1	ISE14_BM (ISE 14)	ralab25	Bitgen Complete!	100%	4/16/13 12:18 AM	

Figure 5.10: Bitstram generation successfully completed

The bit files are located in the respective folder for each individual run. Archive the full bit file for the first configuration and partial bit files for the rest of configurations for further processing.

In the Zynq system, the PS is the master and boots following a normal boot process. It loads the application from non-volatile memory and executes in place. The PL of the Zynq loads both partial and full configuration via the Processor Configuration Access Port (PCAP). As an alternative, the iMPACT tool allows the user to program the bit file into the system via a JTAG chain.

Chapter 6

Software Development

Contents

This chapter introduces the use of software-controlled Partial Reconfiguration (PR) to dynamically control the runtime system reconfiguration process. The Processing System (PS) is a processor and software-centric paradigm and boots itself immediately after power on. The tightly integrated PS and Programmable Logic (PL) structure of Zynq allows the peripherals to be dynamically enabled or disabled through software control. The embedded processor carries out PR by sequentially loading the TCs to the Reconfigurable Partition (RP), performs PRET during runtime and displays the test results on the console screen.

6.1 Functional Test of Self-test Wrapper

To validate the correct behaviour of the self-test wrapper, the RP in PL is first configured with a functional TC, which is either an inverter or an adder. A bare-metal application is then developed to write to and read back the registers in the user_logic module via the AXI4-Lite bus interconnect. Bare-metal refers to a software application without an operating system (OS), typically this type of system does not require numerous features provided by an OS. Using OS consumes certain amount of processor throughput and also creates overhead to the system comparing with simple bare-metal systems without OS. The design of this work does not necessarily include Linux OS, for the sake of simplicity, bare-metal is chosen as the embedded solution.

6.1.1 Create a Bare-metal Application

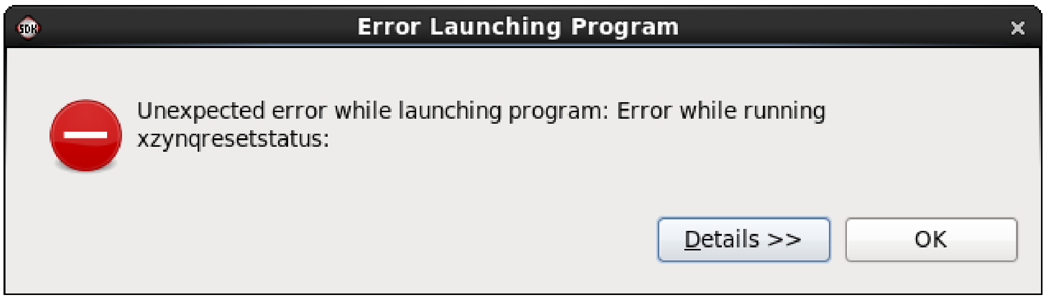
Based on the Eclipse framework, the SDK uses a C/C++ development environment to create software applications for the targeted ARM processor. It is the supplementary to the XPS [27]. In XPS, the self-test wrapper peripheral of PS was created and connected to the AXI4-Lite bus interface utilizing the CIP option. The XPS also provides PS hardware platform information, including configuration settings, the address map of peripheral registers, and the platform initialization bit file. These information are exported to SDK when SDK is launched.

In the EDK flow stated in Chapter 5, we have created a C bare-metal project to interface with the hardware. For targeting the existing evaluation board, the default standalone Board Support Package (BSP) is selected. For the C program, any code written in the main should be between init_platform and cleanup_platform functional calls. These two functions handle all of the register initialization and cleanup. After the code is written, the linker script is created to determine the location to place the executable file, can be either the internal memory of one of the processor cores or the DDR memory. If the code can be successfully compiled, the project is built. The next stage is to set up the run configurations to test the application on the board.

6.1.2 Test on Board

The jumpers on the ZedBoard need to be configured in the right position. Once they are correctly setup, a terminal window is used to monitor the test results. The logic implemented in the PL portion of the Zynq device represents the specific hardware properties, needs first to be programmed into the board. We launch iMPACT programming tool from PlanAhead, select the full bit file of one TC, and download it to the board. We can also do it with SDK, but it hangs up sometimes. When the PL is properly configured, it is ready to be tested. In addition, the ELF file needs to be created before loading it to the board via the JTAG chain. We select C/C++ ELF as option to run configurations and execute the test.

During the test execution, an unexpected error was encountered while launching program, refer to figure 6.1.



Unexpected error while launching program: Error while running xzynqresetstatus:

Figure 6.1: Unexpected error while Launching program

The SDK error log shows java.lang.RuntimeException:

```
at com.xilinx.sdk.targetmanager.internal.TM.downloadELF(Unknown Source)
at com.xilinx.sdk.debug.core.internal.AppRunner.run(Unknown Source)
```

The solution is to totally delete the standalone_bsp imported from PlanAhead in SDK, including even the contents in the harddisk. A new BSP is recreated to be loaded within the hardware platform. The test is successfully executed with the results shown on the screen.

6.2 Device Configuration

In this work, only the AXI-PCAP bridge of the Device Configuration (DevC) interface is used for non-secure PL configuration. Figure 6.2 illustrates the flow for device configuration after boot and PR, it is based on the similar picture in [31].

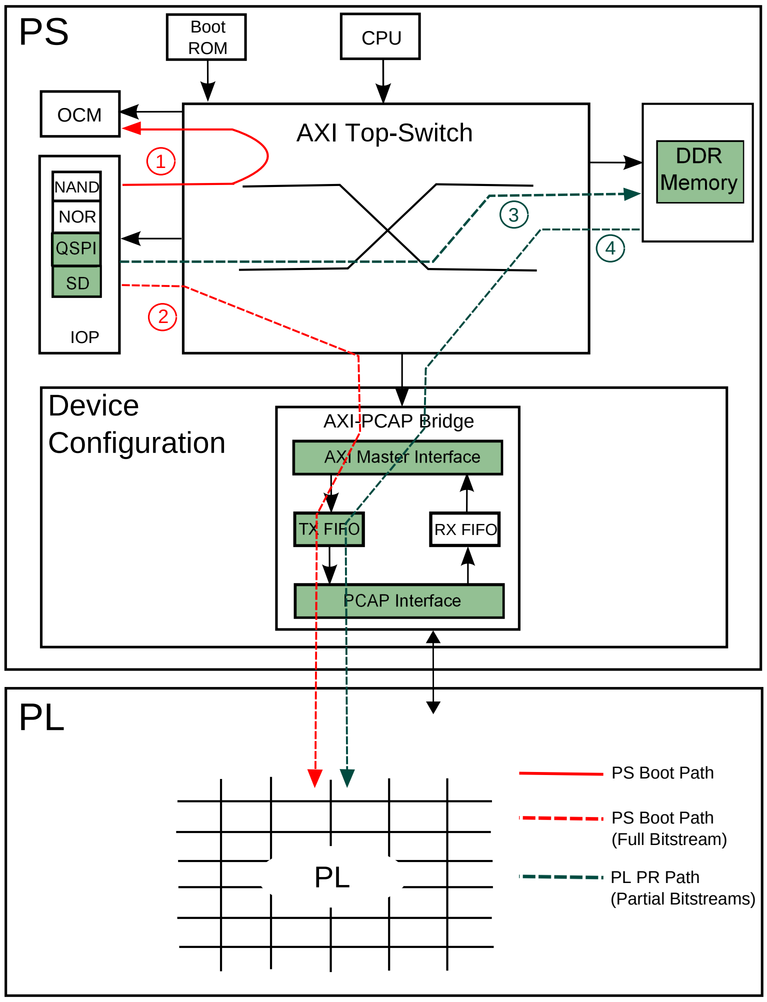
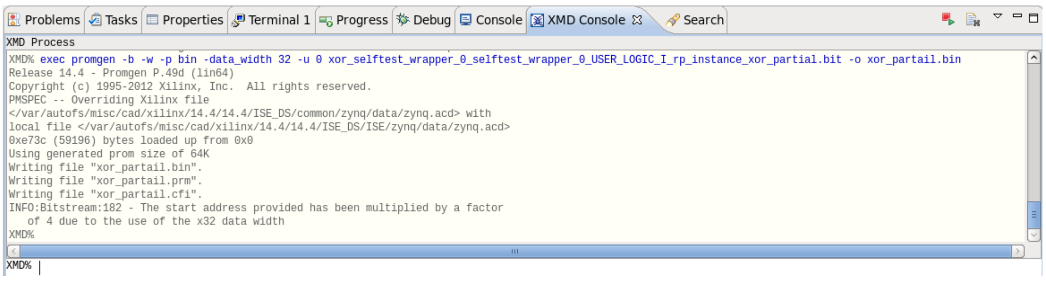


Figure 6.2: Device Configuration and Partial Reconfiguration Flow

The software-controlled partial reconfiguration is carried out through DevC and Processor Configuration Access Port (PCAP) interface [31]. The AXI-PCAP bridge converts 32-bit AXI data to the format compatible to the PCAP protocol. A DMA engine handles the data movement between TX/RX FIFO and DDR memory. The device configuration function needs to be programmed, and be called by the embedded processor. The operating frequency of PCAP interface is 100 MHz, so 32-bit PCAP can have 400 MB/s download throughput for configuring PL in non-secure mode [31]. The DevC driver first clears up DMA and PCAP done interrupt bits to start a new transfer; then configures the PCAP mode and initiates the DMA engine for transferring the bitstream from DDR into PL; finally it polls for DMA and PCAP done bits to trigger an interrupt.

6.2.1 Generate Bin File

The partial bitstreams for test configurations generated in PlanAhead cannot be directly used to configure the PL through PCAP transfer. When Zynq is booted, the online test system is first configured with the full bitstream of the first TC through the boot path. In order to perform tests for different TCs, the partial bit files for the subsequent TCs must be first converted into binary format using PROMGen tool at TCL console, and then they can be used by PCAP to transfer to the PL reconfigurable region. Figure 6.3 shows the screenshot of the command used for converting bit files to the bin files.



```
XMD% exec promgen -b -w -p bin -data_width 32 -u 0 xor_selftest_wrapper_0_selftest_wrapper_0_USER_LOGIC_I_rp_instance_xor_partial.bit -o xor_partail.bin
Release 14.4 - Promgen P.49d (lin64)
Copyright (c) 1995-2012 Xilinx, Inc.  All rights reserved.
PMSPEC -- Overriding Xilinx file
</var/autofs/misc/cad/xilinx/14.4/14.4/ISE_DS/common/zynq/data/zynq.acd> with
local file </var/autofs/misc/cad/xilinx/14.4/14.4/ISE_DS/ISE/zynq/data/zynq.acd>
0xe73c (59196) bytes loaded up from 0x0
Using generated prom size of 64K
Writing file "xor_partail.bin".
Writing file "xor_partail.prm".
Writing file "xor_partail.cfi".
INFO:Bitstream:182 - The start address provided has been multiplied by a factor
   of 4 due to the use of the x32 data width
XMD%
XMD% |
```

Figure 6.3: Convert Partial bitstreams to Binary format

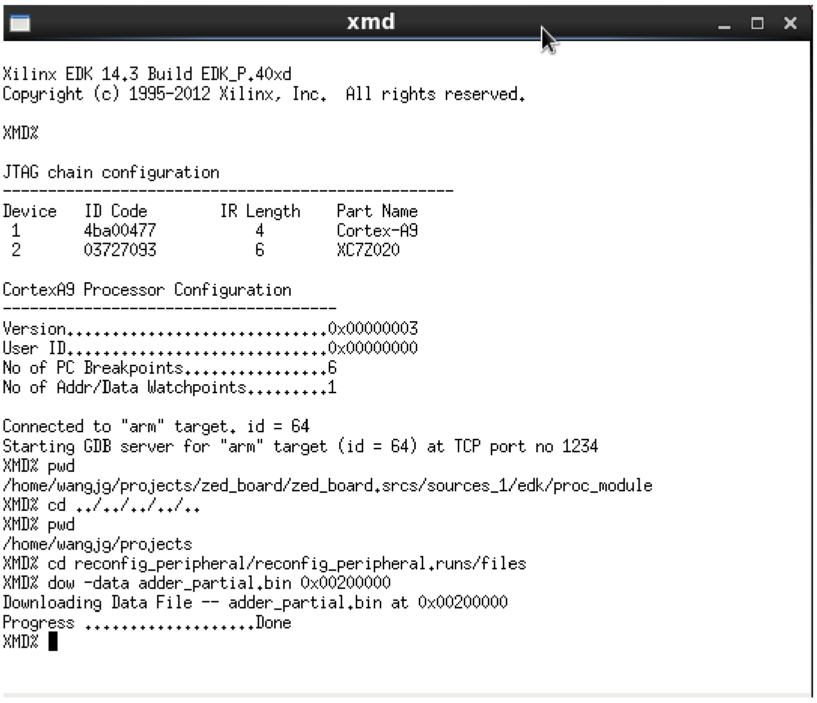
Note that the length of binary file is in bytes, the driver expects the data transfer in double words (32-bit), that means bin file size is four times that of the parameter of bitstream size for PCAP transfer. PROMGen also reports the size of the generated partial bin files, which is required by DevC driver when transmitting the reconfiguration data through the AXI-PCAP bridge to the PL.

6.2.2 XMD Programming

There are several ways to store the bin file into the DDR memory. One way is to open the XMD from XPS, select Launch XMD from Debug menu, go to the directory where the partial binary files located. Program the .bin file by typing:

```
dow -data filename_partial.bin 0x01000000
```

01000000 is the hex format of start address in the DDR memory. The address can be anywhere that is within the address range of DDR. Figure 6.4 shows the screen shot of XMD command.



```
Xilinx EDK 14.3 Build EDK_P.40xd
Copyright (c) 1995-2012 Xilinx, Inc.  All rights reserved.

XMD%

JTAG chain configuration
--------------------------------------------------
Device   ID Code        IR Length    Part Name
 1       4ba00477          4         Cortex-A9
 2       03727093          6         XC7Z020

CortexA9 Processor Configuration
-------------------------------------
Version.............................0x00000003
User ID.............................0x00000000
No of PC Breakpoints................6
No of Addr/Data Watchpoints.........1

Connected to "arm" target. id = 64
Starting GDB server for "arm" target (id = 64) at TCP port no 1234
XMD% pwd
/home/wangjg/projects/zed_board/zed_board.srcs/sources_1/edk/proc_module
XMD% cd ../../../../..
XMD% pwd
/home/wangjg/projects
XMD% cd reconfig_peripheral/reconfig_peripheral.runs/files
XMD% dow -data adder_partial.bin 0x00200000
Downloading Data File -- adder_partial.bin at 0x00200000
Progress ...................Done
XMD% █
```

Figure 6.4: Load the Bin File into DDR Memory

The alternative way is to open up the XMD through the SDK. We can also place the bin file at SDK when set up the device initialization, specify the bin files to be added and provide the address locations. The bin file will be automatically loaded into the DDR memory during the initialization phase.

6.2.3 PL Reconfiguration Driver

The Zynq EPP extends the capability of the PR, after the PL is configured with the full bitstream, part of the PL can be reconfigured with ICAP or PCAP. PCAP allows the software running on the PS to partially reconfigure the PL [31]. For preparing to configure the fabric over the PCAP interface, the partial bitstreams containing TC information are first loaded into specified DDR memory locations, this gives the maximum configuration throughput and shortens the configuration time. Together with the imported hardware platform data and the created standalone BSP, a DevC driver program is developed in SDK using bare-metal application, it enables software-controlled partial bitstream to be transferred from DDR to the PL via PCAP. The software control flow for PR through DevC/PCAP is as follows [32]:

1. Initialize the DevC interface driver. For the first time AXI (Advanced eXtensible Interface) access to the DevC interface block, it is necessary to write to

the Unlock register with the value of 0x757BDF0D. At power on, the boot ROM will unlock the DevC interface by writing the same value to this Unlock Register. This step has to be done after reset, the subsequent AXI accesses can be carried on after this. The Unlock register is used to protect the DevC interface configuration registers from corrupted ROM code.

2. Set PCAP_PR (Control Register bit [27]) and PCAP_MODE (Control Register bit [26]) to High. Because power-on reset default value of both bits are equal to 1, this step is optional.

3. Clear the previous configuration from the PL, this step is also optional. For doing that, global reset is provided to PL by setting PCFG_PROG_B (Control Register bit [30]) to High and then Low. Check if PCFG_INIT (Status Register bit[4]) is equal to 0, which indicates the PL is busy doing housecleaning and is not ready to receive PCAP data. After that, clear PCFG_DONE_INT by writing a 1 to Interrupt Status Register bit [2], to indicate that PL configuration is cleared and not in user mode.

4. Check if PCFG_INIT (Status Register-bit[4]) is set to High by the PL, which denotes the PL housecleaning is done, the previous configuration is cleared and PL is ready to receive next PCAP data.

5. Clear D_P_DONE_INT by writing a 1 to Interrupt Status Register bit [12], to clear DMA (Direct Memory Access) and PCAP transfers done bit, declare the start of a new transfer.

6. To initiate a DevC DMA transfer, provide the source address as the location of new partial bitstream in the DDR memory and source length to be the total number of 32-bit words in the new bitstream. Set the destination address to be 0xFFFF_FFFF and destination Length to be the same as the source length, the total number of 32-bit words in the new bitstream.

7. In this bare-metal case, polling is used for DevC interface Interrupt Status Register bit [13] (DMA_DONE_INT) and bit [12] (D_P_DONE_INT), to determine if the DMA and PCAP transfer are completed.

8. Check if PCFG_DONE_INT (Interrupt Status Register bit [2]) is set to low by the PL, which means the PL is programmed.

To initialize the DMA transfer, the source address in the DDR memory where each partial bitstream is stored and the size of the bitstream need to be passed over to the DevC driver. A functional call is used to transfer the partial bitstream, polling is used to indicate that DMA and PCAP transfer are done. The functional call returns when the transfer is finished [31].

As an additional feature for an overall DMA transfer, to differentiate the last transfer, we can set the two LSBs of the source and destination address to 2'b01. The DMA controller will take this information to provide the DMA_DONE interrupt. For the last DMA transfer, only when both the AXI and PCAP transfers are done, the DMA_DONE interrupt is triggered. For all other DMA transfers, the DMA_DONE interrupt is triggered after the AXI transfers are done, nevertheless, the PCAP transfers might still be carrying on. This allows overlapped AXI and PCAP transfers, except for the last DMA transfer [32].

6.3 Automated Tests Selection and Execution

To allow the configuration of TCs, generation of test stimuli and analysis of responses to be controlled by software, a SDK project is developed, which combines the driver program with the control logic accessing the user_logic. So the various test configurations can be scheduled to be downloaded to the reconfigurable partition of PL; TPG and ORA are applied accordingly; tests are performed automatically; results are printed on screen. It delivers the capability of PCAP control and provides the flexibility of test selection with minimum user interaction.

Once the start-up initialization phase is finished, the software application displays the options to be selected on screen, and waits for the user's input through the command line. Once an option is selected, the designated TC is configured into RP region of the PL through DevC/PCAP. The TPG and ORA for each TC are external to the RP, they are applied after the TC is reconfigured, this is done by writing a specific value to the control register. After the test is executed, the output result is written to the status register or output registers (for functional tests), and finally gets printed on screen.

The dynamic reconfiguration of the logic function in the PL and execution of the selected test are fully automated through software control. Figure 6.5 shows the software control flow of this bare-metal application. The yellow boxes are related to the partial reconfiguration flow for transferring a partial bitstream across the PCAP interface.

The Self-test Wrapper has an AXI4-Lite interface that is used by the PS to configure TCs and control the execution of the tests. Before starting a transfer, the interrupt handler has to be disabled, otherwise the PR process could generate a fake interrupt to the ISR (Interrupt Status Register). Such an interrupt triggers false access to RP of the PL which has not been fully configured yet, and in turn causes the system hang up. In order to prevent the above stated scenario, software control is used to decouple the interface. For ensuring the TC is successfully reconfigured to the RP, a PL reset is asserted before the reconfiguration process to disable the AXI interconnect [31]. After the reconfiguration is finished, the FPGA reset signal is de-asserted, the AXI connection is thereafter enabled, testing is ready to be performed with the configured TC.

6.3.1 Standalone Boot Image Creation

If required, a standalone boot image can be created. After power-on, the ARM processor will first boot from the BootROM which determines the external memory interface; then based on the boot mode, it reads the PS boot image from the external boot device (SD card). When BootROM shuts down, PS hands over the control to the First Stage Bootloader (FSBL). The FSBL image is loaded from external non-volatile device into the On-Chip Memory (OCM) after boot. FSBL can fetch the configuration data from flash memory, automatically load the first full bitstream to configure the PL via PCAP, load the user application code into DDR memory and

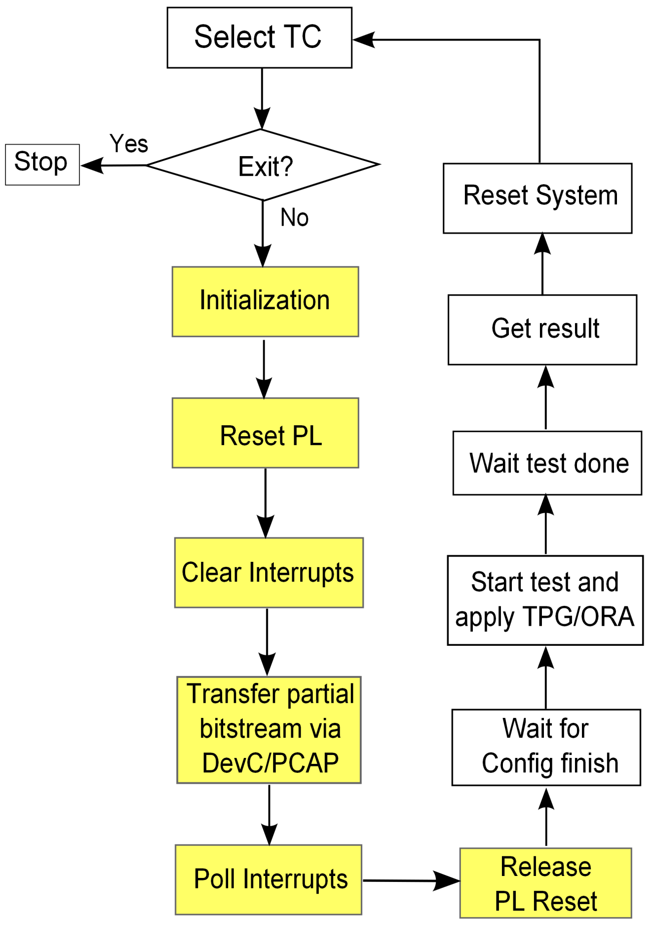


Figure 6.5: Software Control Flow

execute. SDK includes integrated applications for boot image creation, the FSBL project and the flash programmer [33].

A program called Bootgen can be utilized to create a boot image file. It builds the required boot header; provides partition tables; and processes the input data files. The input files include the ELF file for the software project, the first full configuration bitstream and the partial binary files. Bootgen consists of additional features like allocating destination memory location or providing alignment required by each partition. The input source to this utility is a configuration file with BIF (Boot Image Format) extension, it comprises BootROM header, FSBL image and partition images. The header is required by the BootROM loader to load the FSBL image, the partition images are handled by the FSBL [33]. The output BIN file contains the boot header and the data partitions derived from the input BIF file [33]. Below is an example of using the Bootgen command line to create the boot image:

```
bootgen -image Design.bif -o i DesignImage.bin
```

To create the boot image for the FSBL application at SDK, select Create Boot Image option and follow the procedures below:

1. Take the default BIF file
2. Locate the FSBL elf file
3. Use the full hardware bitstream to program the PL
4. Specify BOOT.BIN as output file to create boot image

Once the boot image is created, it can be used for programming the flash and running the application.

6.3.2 Booting From the SD Card

Booting the Zynq SoC device in non-secure mode can also use flash memories. After power-on reset, the boot mode strapping pins on ZedBoard determine the external memory interface. Since the procedure for booting from QSPI flash is more complicated, we can select to boot the system from the Secure Digital (SD) card. The procedure to prepare the SD Card as boot flash is as below:

1. Insert the SD card in a SD card slot on PC, backup the SD card contents, delete all files from the SD card.
2. Copy the created BOOT.BIN file from the PC to /boot_image folder on the SD card.
3. Turn ZedBoard off. Set the configuration mode jumpers for SD card MODE pins as described below:

```
MODE3 (JP10) is shunted to 3.3V
MODE2 (JP9) is shunted to 3.3V
All other MODE pins (JP7, JP8 and JP11) are shunted to GND
```

4. Remove the SD card from the PC and insert it into the SD card slot on the ZedBoard.
5. Power on board, a green LED will illuminate immediately to indicate ‘power good’ and a blue LED will light up to signal ‘Done’ after the processor has been initialized and the PL is configured.

The booting process from SD card will take only few seconds.

Chapter 7

Evaluation

Contents

This chapter contains the validation steps and obtained results. The evaluation of the implemented self-test wrapper includes carrying out the TC configuration and test execution. In addition, the implementation of software-controlled reconfiguration and test execution is also discussed.

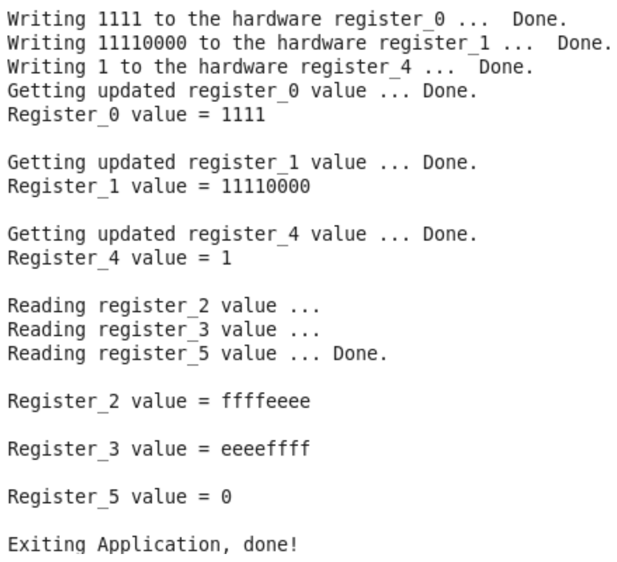
7.1 Validation

7.1.1 Functional Test of Self-test Wrapper

Having written the code for functional tests and successfully compiled it, we can now download the application to the board to test the operation of self-test wrapper with partial reconfiguration. To do this, first we need to create a linker script file that defines the location of the executable in the system memory. Since this application program is small, we put it within the on-chip memory. While it is often the best to place the program in the DDR memory, so we can determine if this interface is configured correctly. We use Generate Linker Script option to create a linker script, and run the application. The figure 7.1 shows the result of the functional test Inverter. The figure 7.2 shows the result of the functional test Adder.

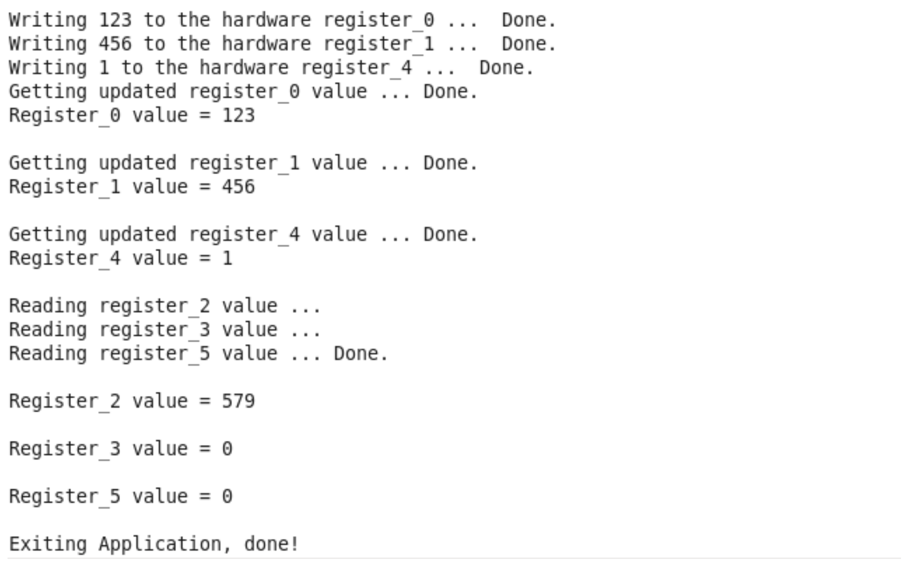
7.1.2 Configuration Time

The configuration time scales approximately linearly with the bitstream size. It grows with the number of reconfigurable cells with small variances towards the location and the configurations of the cells. The PCAP interface is operated at



```
Writing 1111 to the hardware register_0 ...  Done.
Writing 11110000 to the hardware register_1 ...  Done.
Writing 1 to the hardware register_4 ...  Done.
Getting updated register_0 value ... Done.
Register_0 value = 1111

Getting updated register_1 value ... Done.
Register_1 value = 11110000

Getting updated register_4 value ... Done.
Register_4 value = 1

Reading register_2 value ...
Reading register_3 value ...
Reading register_5 value ... Done.

Register_2 value = ffffeeee

Register_3 value = eeeeffff

Register_5 value = 0

Exiting Application, done!
```

Figure 7.1: Result of the Functional test Inverter



```
Writing 123 to the hardware register_0 ...  Done.
Writing 456 to the hardware register_1 ...  Done.
Writing 1 to the hardware register_4 ...  Done.
Getting updated register_0 value ... Done.
Register_0 value = 123

Getting updated register_1 value ... Done.
Register_1 value = 456

Getting updated register_4 value ... Done.
Register_4 value = 1

Reading register_2 value ...
Reading register_3 value ...
Reading register_5 value ... Done.

Register_2 value = 579

Register_3 value = 0

Register_5 value = 0

Exiting Application, done!
```

Figure 7.2: Result of the Functional test Adder

100MHz, for a 32-bit wide transfer, it can achieve 400MB/s throughput for non-secure PL configuration in a standalone application [31]. Each partial bitstream has the size of 59196 bytes, in theory, it takes about 150 us to configure through the PCAP interface. The actual configuration time is measured between the start and the end of the DevC DMA and PCAP transfer driver function call, which includes interrupt handling, the time measured by software is around 276 us.

7.1.3 TC Tests

When implementing partial reconfigurable design in PlanAhead, the first configuration is fully implemented and the static logic is promoted to be used for the rest of configurations. We choose the XOR TC to be the first configuration, and the test is running at 100 MHz PL clock, which is the default value provided by the PS clock generator. The PlanAhead tool automatically ensures that the resources for each TC configuration are contained within the defined RP region and do not cross over the static portion of the design. After successfully implementing the XOR configuration, we promoted its static implementation results to be reused by the rest configurations. Finally, the full and partial bitstreams are successfully generated for all TCs, and are ready to be used for partially reconfiguring the test container.

When the container is configured sequentially by individual TC, and it is tested by applying respective TPG and ORA for each TC. By default, when the test is done, should be no error flag turned on, since the fabric is error-free. However, during software-controlled test executions, we found some tests were failing.

One observation is that PL clock frequency and board temperature affect the test results. Therefore, we adjusted the clock speed for PL at XPS, and generated 20 MHz FCLK0 from PS clock generator to be used as PL clock, reset the output products from PlanAhead after having changed the clock frequency. We set the period constrain and rebuild the system, and regenerate full and partial bitstreams for all TCs. Each time the changes made to the the hardware have to be exported to SDK. For SR and RAM test, retry is required if the first time test fails.

7.2 Wrapper synthesis Result

7.2.1 Resource usage and CLB count

The resources utilization for the whole design including each TC implementation is listed in table 7.1, the resources utilization for static portion are the same for all TCs, there are slight differences in Registers and LUTs counts for different TCs.

		Xor_Latch_ CarryCoutFF		**Xnor**		**SR**		**RAM**		**CarrySum**	
Resource	**Available**	**Util**	**%Util**	**Util**	**%Util**	**Util**	**%Util**	**Util**	**%Util**	**Util**	**%Util**
Register	106400	2449	2%	3249	3%	1649	1%	1749	1%	2049	1%
LUT	53200	2708	5%	2708	5%	2708	5%	2408	4%	2708	5%
Slice	13300	1186	8%	1186	8%	1186	8%	1186	8%	1186	8%
IO	200	13	6%	13	6%	13	6%	13	6%	13	6%
RAMB36E1	280	32	11%	32	11%	32	11%	32	11%	32	11%
RAMB18E1	280	2	1%	2	1%	2	1%	2	1%	2	1%
ICAP	2	1	50%	1	50%	1	50%	1	50%	1	50%
BUFG	32	5	15%	5	15%	5	15%	5	15%	5	15%
BUFGCTRL	32	1	3%	1	3%	1	3%	1	3%	1	3%

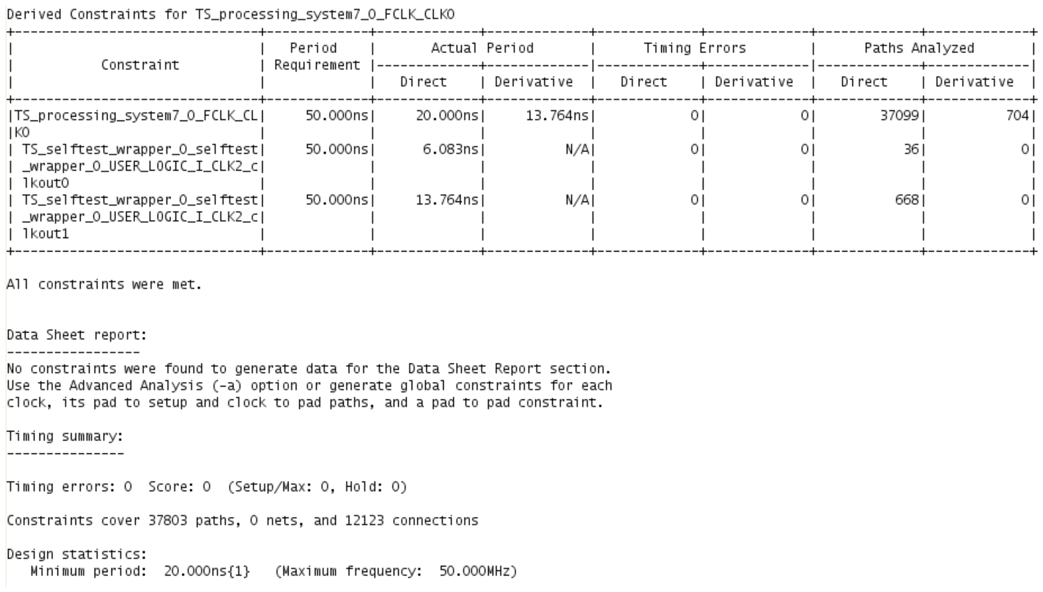
Table 7.1: Resources utilization for all TCs

7.2.2 Timing Result

Timing constraint:

```
TS_processing_system7_0_FCLK_CLK0 = PERIOD
TIMEGRP "processing_system7_0_FCLK_CLK0" 50 ns HIGH 50\%;
```

The implemented timing results and the derived constraint report for clocks is shown in figure 7.3.

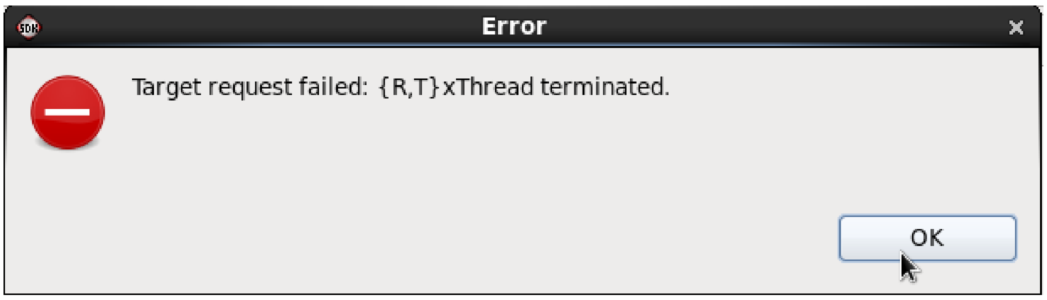


```
Derived Constraints for TS_processing_system7_0_FCLK_CLK0
+-------------------------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+
|                               |   Period    |       Actual Period       |      Timing Errors        |      Paths Analyzed       |
|           Constraint          | Requirement |-------------+-------------|-------------+-------------|-------------+-------------|
|                               |             |   Direct    | Derivative  |   Direct    | Derivative  |   Direct    | Derivative  |
+-------------------------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+
|TS_processing_system7_0_FCLK_CL|    50.000ns|    20.000ns|    13.764ns|            0|            0|        37099|          704|
|K0                             |             |             |             |             |             |             |             |
| TS_selftest_wrapper_0_selftest|    50.000ns|     6.083ns|          N/A|            0|            0|           36|            0|
| _wrapper_0_USER_LOGIC_I_CLK2_c|             |             |             |             |             |             |             |
| lkout0                        |             |             |             |             |             |             |             |
| TS_selftest_wrapper_0_selftest|    50.000ns|    13.764ns|          N/A|            0|            0|          668|            0|
| _wrapper_0_USER_LOGIC_I_CLK2_c|             |             |             |             |             |             |             |
| lkout1                        |             |             |             |             |             |             |             |
+-------------------------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+

All constraints were met.


Data Sheet report:
-----------------
No constraints were found to generate data for the Data Sheet Report section.
Use the Advanced Analysis (-a) option or generate global constraints for each
clock, its pad to setup and clock to pad paths, and a pad to pad constraint.

Timing summary:
---------------

Timing errors: 0  Score: 0  (Setup/Max: 0, Hold: 0)

Constraints cover 37803 paths, 0 nets, and 12123 connections

Design statistics:
   Minimum period:  20.000ns{1}   (Maximum frequency:  50.000MHz)
```

Figure 7.3: Derived Constraints Report for TS_PS7_FCLK0

7.3 Evaluation of Standalone Software Application

During the execution of TC tests, we encountered test flow hangs, most of time happened at second TC test when the program tries to access the PL peripheral via the AXI port. The error message is showed in figure 7.4.

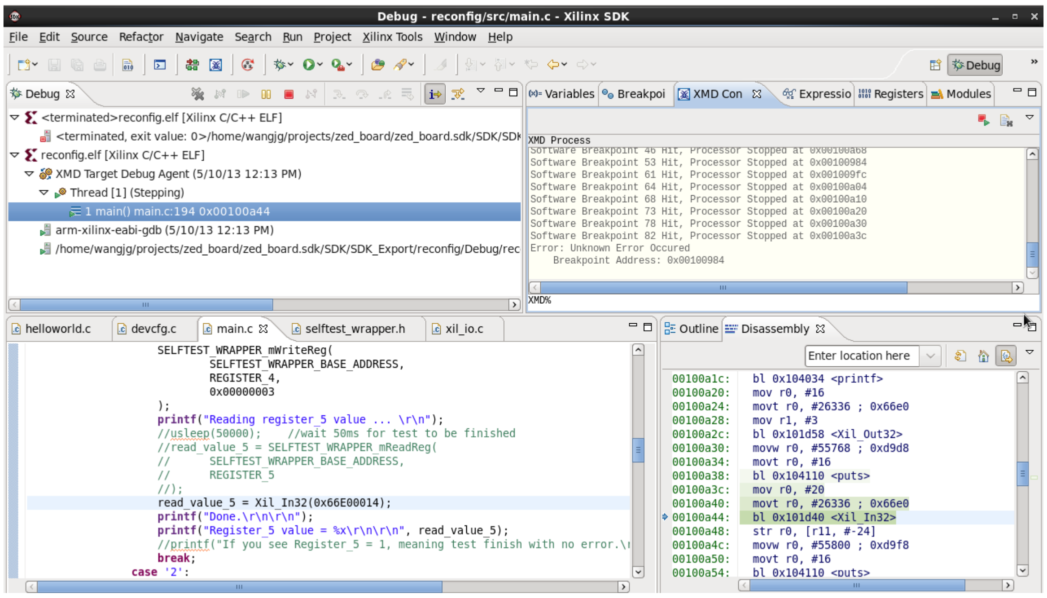


Error

Target request failed: {R,T}xThread terminated.

Figure 7.4: Error encountered during Software execution

SDK includes Xilinx Microprocessor Debugger (XMD) to debug the software project. We launch SDK debug by selecting Debug As function and run debug on hardware. Step through, we find that it hangs within a call to 'Xil_In32', it is a register read function call. Try to call 'Xil_In16' and 'Xil_In8', get the same error result. When we force the program to terminate, typically we not only have to power cycle the board, but also have to close SDK completely and open again to start another execution. Without the 'Xil_In32' call, the program runs smoothly. Figure 7.5 is the screen shot during software debug, it shows the program stops at 0x00100a44 when it executes the Xil_In32 command.



```
<terminated>reconfig.elf [Xilinx C/C++ ELF]
    <terminated, exit value: 0>/home/wangjg/projects/zed_board/zed_board.sdk/SDK/SDK
reconfig.elf [Xilinx C/C++ ELF]
  XMD Target Debug Agent (5/10/13 12:13 PM)
    Thread [1] (Stepping)
      1 main() main.c:194 0x00100a44
    arm-xilinx-eabi-gdb (5/10/13 12:13 PM)
    /home/wangjg/projects/zed_board/zed_board.sdk/SDK/SDK_Export/reconfig/Debug/rec
```

```
XMD Process
Software Breakpoint 46 Hit, Processor Stopped at 0x00100a68
Software Breakpoint 53 Hit, Processor Stopped at 0x00100984
Software Breakpoint 61 Hit, Processor Stopped at 0x001009fc
Software Breakpoint 64 Hit, Processor Stopped at 0x00100a04
Software Breakpoint 68 Hit, Processor Stopped at 0x00100a10
Software Breakpoint 73 Hit, Processor Stopped at 0x00100a20
Software Breakpoint 78 Hit, Processor Stopped at 0x00100a30
Software Breakpoint 82 Hit, Processor Stopped at 0x00100a3c
Error: Unknown Error Occured
    Breakpoint Address: 0x00100984
XMD%
```

```
            SELFTEST_WRAPPER_mWriteReg(
                    SELFTEST_WRAPPER_BASE_ADDRESS,
                    REGISTER_4,
                    0x00000003
            );
            printf("Reading register_5 value ... \r\n");
            //usleep(50000);    //wait 50ms for test to be finished
            //read_value_5 = SELFTEST_WRAPPER_mReadReg(
            //      SELFTEST_WRAPPER_BASE_ADDRESS,
            //      REGISTER_5
            //);
            read_value_5 = Xil_In32(0x66E00014);
            printf("Done.\r\n\r\n");
            printf("Register_5 value = %x\r\n\r\n", read_value_5);
            //printf("If you see Register_5 = 1, meaning test finish with no error.\r
            break;
        case '2':
```

```
00100a1c:   bl 0x104034 <printf>
00100a20:   mov r0, #16
00100a24:   movt r0, #26336 ; 0x66e0
00100a28:   mov r1, #3
00100a2c:   bl 0x101d58 <Xil_Out32>
00100a30:   movw r0, #55768 ; 0xd9d8
00100a34:   movt r0, #16
00100a38:   bl 0x104110 <puts>
00100a3c:   mov r0, #20
00100a40:   movt r0, #26336 ; 0x66e0
00100a44:   bl 0x101d40 <Xil_In32>
00100a48:   str r0, [r11, #-24]
00100a4c:   movw r0, #55800 ; 0xd9f8
00100a50:   movt r0, #16
00100a54:   bl 0x104110 <puts>
```

Figure 7.5: Unknown error occurred during Debug

Finding: Starting from EDK Design Suite 14.2, the TCL (Tool Command Language) file for PS initialization does not enable the level shifters. Also, in version 2 silicon, the boot ROM does not bring the PL out of reset state. The user has to enable the level shifter and to bring the PL out of reset after the bitstream is downloaded in the XMD flow.

Solution to this problem is to enable the level shifters for a non-PS instantiated bitstream and enable the AXI interface after the download of the bitstream. Certain steps in the driver flow have to be disabled to have a smooth transfer.

All the online self-tests can now be executed without hanging.

Chapter 8

Conclusion and Outlook

Contents

8.1 Summary

This work presents the design and evaluation of a self-test wrapper in a runtime reconfigurable architecture. The purpose of the wrapper is to encapsulate a block of reconfigurable logic, provide a well-defined interface, and support self-test. The self-test wrapper allows to test the reconfigurable logic by TCs. The wrapper is controlled by an embedded processor.

The BIST enabled reconfigurable architectures ensures the reliability of the runtime reconfigurable systems. Once a fault is detected in the area under test, an alternative solution can be employed to salvage the operation from the failing part, thereby guaranteeing the correct behavior of the system. One important aspect to the BIST is the embedded processor based BIST approach. The software running on the embedded processor can efficiently control the reconfiguration process and test execution.

The Xilinx Zynq SoC provides a processing system and programmable logic part. The embedded processor inside the PS performs the partial reconfiguration of the reconfigurable region inside the PL with the prebuilt TCs, each TC is associated with a particular partial bitstream. The processor takes over the control of tests, application of input stimuli, and retrieval of the ORA contents through the self-test wrapper. The entire design is based on tools from the Xilinx tool chain, for the development of test configurations, the static logic part and software, as well as the overall system integration.

The CLBs in the reconfigurable region are thoroughly tested by the selectable test sessions online. The impact of tests to the overall system operation is minimal. The reconfiguration process together with the testing of the reconfigurable architecture is fully automated under software control, and requires minimal user interaction. Since the TPG and ORA are moved out from the reconfigurable area, there is no hardware overhead incurred. The execution of the complete test suite takes only a small amount of time.

The self-test wrapper has been successfully implemented, the functionality of the self-test wrapper is validated by both functional tests and TC tests. All test configurations have been designed to achieve full logic fault coverage of the Artix-7 CLBs. Since the download of partial bitstreams requires much shorter time than full bitstreams, partial reconfiguration reduces test time by reconfiguring only the resources under test, once the full configuration has been downloaded into the fabric. In addition, the use of the PCAP interface for reconfiguration provides a considerable speed-up over the serial Boundary Scan interface (JTAG). JTAG mode can also be used to configure the PL, however, it is here mainly used for development and debug. The maximum achievable operating frequency is 50 MHz.

The online self-test wrapper is implemented on the Zynq-7000 EPP with ARM Cortex-A9 processor cores. However, the CLB structural test method and EDK approach are applicable to different reconfigurable architecture that contains an embedded processor. Utilizing the core-centric approach greatly maximizes the design reuse, and reduces the amount of time required for the development.

8.2 Further Tasks and Outlook

As stated in chapter 6, the Zynq device can boot from external flash memory, for example, from an SD card. To extend the system in that way, first the ELF file is to be generated from First Stage Bootloader; then it is combined with the full bitstream from the first configuration, and ELF file from standalone software application to create the boot image. Finally, the standalone SD-card image can be built to be loaded to the SD card, which should include the boot image and all partial bitstreams.

Another extension is to develop a Linux software application for partial reconfiguration. In order to achieve that, one can use the Linux device configuration driver, built on top of a virtual file system provided by Linux, export devices and drivers information from kernel to the user space. To develop the Linux driver to transfer the partial bitstreams over the PCAP interface, a device configuration driver is needed to write to the device configuration space; it initiates the DMA transaction, and waits for an interrupt signaling the completion of the transfer. Partial reconfiguration can also be initiated from a shell.

Appendix A

Artix-7 XDL Code

Listings A.1 illustrates a XDL code example for Hard Macro xnor_test as test configuration, it indicates the input and output ports, the reference point unit 17 and the configuration string specifies the detail slice programming for unit19. It is a SLICEL, placed in CLB “CLBLM_R_X11Y0” and slice name is “SLICE_X15Y0”.

```
design "__XILINX_NMC_MACRO" xc7z020-1clg484;

module "xnor_test" "unit17" , cfg "_SYSTEM_MACRO::FALSE ";
  port "clk" "unit0" "CLK";
  port "in_tpg" "unit0" "AX";
  port "in_tpg5" "unit0" "A6";
  port "in_tpg1" "unit0" "A2";
  port "in_tpg2" "unit0" "A3";
  port "in_tpg3" "unit0" "A4";
  port "in_tpg4" "unit0" "A5";
  port "in_tpg0" "unit0" "A1";
  port "rst" "unit0" "SR";
  port "out_ora0" "unit11" "DQ";
  port "out_ora3" "unit23" "DMUX";
  port "out_ora1" "unit23" "DQ";
  port "out_ora2" "unit11" "DMUX";
inst "unit19" "SLICEL",placed CLBLM_R_X11Y0 SLICE_X15Y0 ,
  cfg " AOUTMUX::A5Q AFFINIT::INIT0 CLKINV::CLK DFF::#FF A5FFMUX::
IN_B BFFSR::SRLOW DFFINIT::INIT0 B6LUT:XORB_unit19:#LUT:O6=
(A1@(A2@(A3@(A4@(A5@~A6))))) AFFSR::SRLOW BOUTMUX::B5Q D5FFINIT::INIT0
D6LUT:XORD_unit19:#LUT:O6=(A1@(A2@(A3@(A4@(A5@~A6))))) C5FFSR::SRLOW
BFFINIT::INIT0 BFFMUX::O6 BFF::#FF D5FFSR::SRLOW B5FFSR::SRLOW
CFFINIT::INIT0 CFF::#FF DFFSR::SRLOW A6LUT:XORA_unit19:#LUT:O6=
(A1@(A2@(A3@(A4@(A5@~A6))))) AFFMUX::O6 SYNC_ATTR::ASYNC DOUTMUX::D5Q
A5FFSR::SRLOW DFFMUX::O6 CFFSR::SRLOW COUTMUX::C5Q A5FFINIT::INIT0
SRUSEDMUX::IN C6LUT:XORC_unit19:#LUT:O6=(A1@(A2@(A3@(A4@(A5@~A6)))))
D5FFMUX::IN_B B5FFINIT::INIT0 C5FFMUX::IN_B CFFMUX::O6 AFF::#FF
B5FFMUX::IN_B C5FFINIT::INIT0 ";
  net "rst_IBUF" ,
    inpin "unit19" SR ,    ;
  net "clk_BUFGP" ,
    inpin "unit19" CLK ,   ;
  net "stim<1>" ,
    inpin "unit19" A2 ,    ;
endmodule "xnor_test" ;
```

Figure A.1: XDL code example for HM xnor_test

Listing A.2 reveals the detail slices configuration for Hard Macro as ram_test.

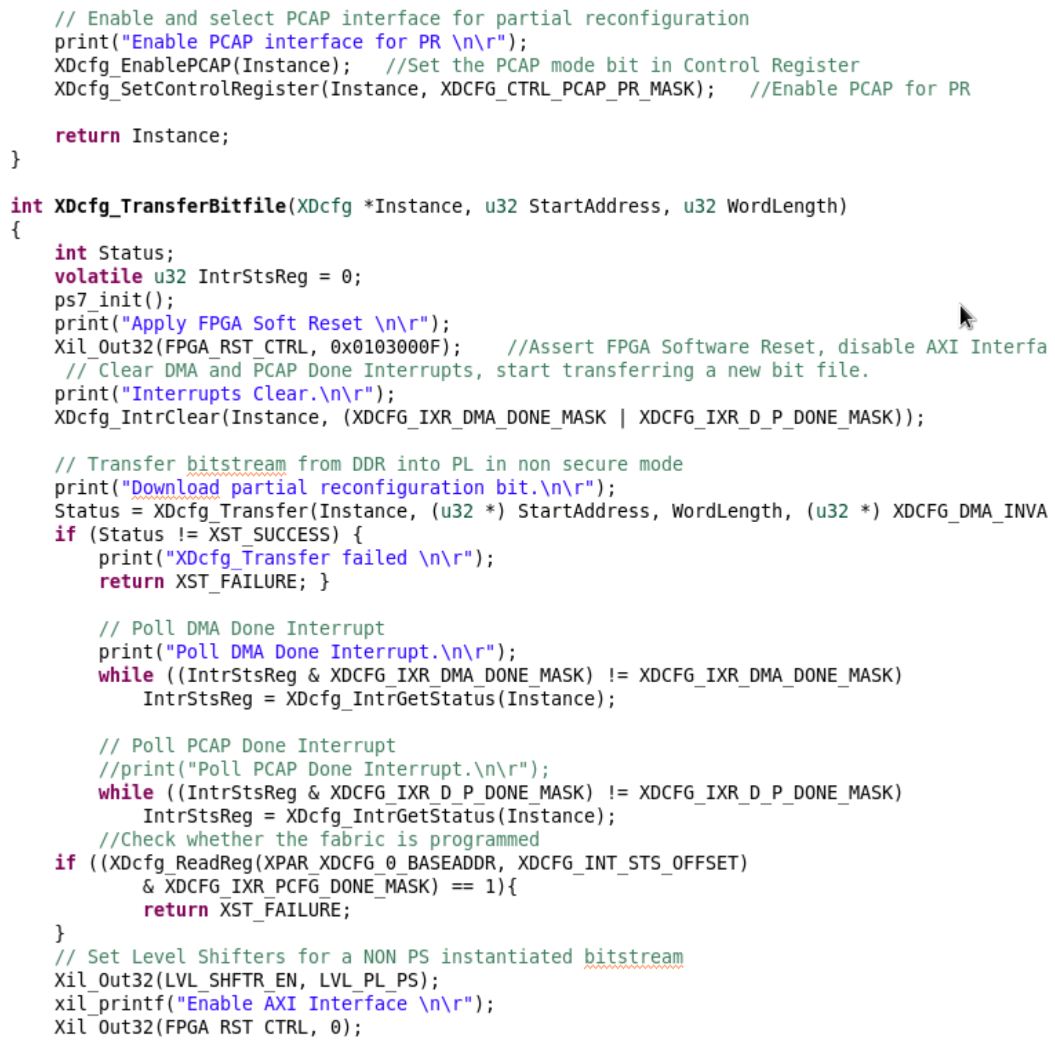
```
inst "unit19" "SLICEL",placed CLBLM_R_X11Y0 SLICE_X15Y0  ,
  cfg " A6LUT:ora19:#LUT:O6=A1+(A2@(A3@(A4@A5))) SRUSEDMUX::IN
AFFMUX::O6 SYNC_ATTR::ASYNC AFFINIT::INIT0 CLKINV::CLK
CEUSEDMUX::IN AFF:oraff19:#FF AFFSR::SRLOW "
    ;
inst "unit18" "SLICEM",placed CLBLM_R_X11Y0 SLICE_X14Y0  ,
  cfg " DUSED::0 CUSED::0 A6LUT:RAMA_unit18:#RAM:
O6=0x0000000000000000 A6RAMMODE::SPRAM64 SYNC_ATTR::ASYNC
CLKINV::CLK BUSED::0 B6LUT:RAMB_unit18:#RAM:O6=0x0000000000000000
B6RAMMODE::SPRAM64 D6LUT:RAMD_unit18:#RAM:O6=0x0000000000000000
D6RAMMODE::SPRAM64 ADI1MUX::AI C6LUT:RAMC_unit18:#RAM:
O6=0x0000000000000000 AUSED::0 WEMUX::CE C6RAMMODE::SPRAM64
BDI1MUX::BI CDI1MUX::CI "
    ;

endmodule "ram_test" ;
```

Figure A.2: XDL code example for HM ram_test

Appendix B

Device Configuration Driver Code

Figure B.1 discloses the C code segment for using PCAP to configure the PL in zynq device, to transfer the partial bitstream from DDR to PL via PCAP interface.



```
    // Enable and select PCAP interface for partial reconfiguration
    print("Enable PCAP interface for PR \n\r");
    XDcfg_EnablePCAP(Instance);   //Set the PCAP mode bit in Control Register
    XDcfg_SetControlRegister(Instance, XDCFG_CTRL_PCAP_PR_MASK);   //Enable PCAP for PR

    return Instance;
}

int XDcfg_TransferBitfile(XDcfg *Instance, u32 StartAddress, u32 WordLength)
{
    int Status;
    volatile u32 IntrStsReg = 0;
    ps7_init();
    print("Apply FPGA Soft Reset \n\r");
    Xil_Out32(FPGA_RST_CTRL, 0x0103000F);    //Assert FPGA Software Reset, disable AXI Interfa
     // Clear DMA and PCAP Done Interrupts, start transferring a new bit file.
    print("Interrupts Clear.\n\r");
    XDcfg_IntrClear(Instance, (XDCFG_IXR_DMA_DONE_MASK | XDCFG_IXR_D_P_DONE_MASK));

    // Transfer bitstream from DDR into PL in non secure mode
    print("Download partial reconfiguration bit.\n\r");
    Status = XDcfg_Transfer(Instance, (u32 *) StartAddress, WordLength, (u32 *) XDCFG_DMA_INVA
    if (Status != XST_SUCCESS) {
        print("XDcfg_Transfer failed \n\r");
        return XST_FAILURE; }

        // Poll DMA Done Interrupt
        print("Poll DMA Done Interrupt.\n\r");
        while ((IntrStsReg & XDCFG_IXR_DMA_DONE_MASK) != XDCFG_IXR_DMA_DONE_MASK)
            IntrStsReg = XDcfg_IntrGetStatus(Instance);

        // Poll PCAP Done Interrupt
        //print("Poll PCAP Done Interrupt.\n\r");
        while ((IntrStsReg & XDCFG_IXR_D_P_DONE_MASK) != XDCFG_IXR_D_P_DONE_MASK)
            IntrStsReg = XDcfg_IntrGetStatus(Instance);
        //Check whether the fabric is programmed
    if ((XDcfg_ReadReg(XPAR_XDCFG_0_BASEADDR, XDCFG_INT_STS_OFFSET)
            & XDCFG_IXR_PCFG_DONE_MASK) == 1){
            return XST_FAILURE;
    }
    // Set Level Shifters for a NON PS instantiated bitstream
    Xil_Out32(LVL_SHFTR_EN, LVL_PL_PS);
    xil_printf("Enable AXI Interface \n\r");
    Xil_Out32(FPGA_RST_CTRL, 0);
```

Figure B.1: DevC/PCAP driver code

Bibliography

[1] *"Virtex-5 FPGA User Guide (UG190)"*, v5.4 ed., Xilinx, Inc., March 2012. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug190.pdf (Cited on pages ix, 11, 12 and 14.)

[2] *"7 Series FPGAs Configurable Logic Block User Guide (UG474)"*, v1.4 ed., Xilinx, Inc., November 2012. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf (Cited on pages ix, 11, 12, 15 and 23.)

[3] M. Abdelfattah, "Evaluation of advanced techniques for structural FPGA self-test," Master's thesis, Universität Stuttgart, 2011. [Online]. Available: http://elib.uni-stuttgart.de/opus/volltexte/2012/7416 (Cited on pages ix, 1, 2, 5, 6, 7, 10, 17, 18, 19, 20, 21, 25, 48 and 49.)

[4] M. Abdelfattah, L. Bauer, C. Braun, M. Imhof, M. Kochte, H. Zhang, J. Henkel, and H. Wunderlich, "Transparent structural online test for reconfigurable systems," in *18th IEEE International On-Line Testing Symposium (IOLTS)*, 2012, pp. 37–42. (Cited on pages ix, 2, 18, 20, 53 and 54.)

[5] *"AMBA AXI and ACE Protocol Specification"*, ARM Ltd Specification IHI0022E, Rev. E, October 2011. (Cited on pages xi, 36 and 37.)

[6] A. Popp, Y. Le Moullec, and P. Koch, "Fast feasibility estimation of reconfigurable architectures," in *4th IEEE Conference on Industrial Electronics and Applications (ICIEA)*, 2009, pp. 117–122. (Cited on page 1.)

[7] *"Partial Reconfiguration User Guide (UG702)"*, v14.3 ed., Xilinx, Inc., October 2012. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_3/ug702.pdf (Cited on pages 1 and 10.)

[8] *"Partial Reconfiguration of a Processor Peripheral Tutorial (UG744)"*, v14.1 ed., Xilinx, Inc., April 2012. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/PlanAhead_Tutorial_Reconfigurable_Processor.pdf (Cited on pages 1, 9 and 54.)

[9] *"Zynq ZedBoard Concepts, Tools, and Techniques"*, v14.1 ed., Xilinx, Inc., August 2012. [Online]. Available: http://www.xilinx.com/university/zynq/documentation/ZedBoard_CTT_v14.1_120808.pdf (Cited on pages 2 and 34.)

[10] *"Zynq-7000 All Programmable SoC Overview (DS190)"*, v1.3 ed., Xilinx, Inc., March 2012. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf (Cited on page 2.)

[11] J. B. Bernstein, M. Gurfinkel, X. Li, J. Walters, Y. Shapira, and M. Talmor, "Electronic circuit reliability modeling," *Microelectronics Reliability*, vol. 46, no. 12, pp. 1957–1979, 2006. (Cited on page 2.)

[12] C. Lavin, M. Padilla, P. Lundrigan, B. Nelson, and B. Hutchings, "Rapid prototyping tools for FPGA designs: Rapidsmith," in *International Conference on Field-Programmable Technology (FPT)*, 2010, pp. 353–356. (Cited on pages 3, 48 and 49.)

[13] N. N. Hasan, "Fine-grained reconfigurable architectures," Tech. Rep., Universität Stuttgart, June 2012. (Cited on pages 7 and 9.)

[14] H. Svensson, "Reconfigurable architectures for embedded systems," Ph.D. dissertation, Lund University, 2008. (Cited on page 8.)

[15] U. Nageldinger, *"Coarse-grained reconfigurable architecture design space exploration"*. Book, Universität Kaiserslautern, 2001. (Cited on page 8.)

[16] *"LogiCORE IP AXI HWICAP (DS817)"*, v2.03.a ed., Xilinx, Inc., July 2012. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/axi_hwicap/v2_03_a/ds817_axi_hwicap.pdf (Cited on pages 10 and 41.)

[17] C. Metra, G. Mojoli, S. Pastore, D. Salvi, and G. Sechi, "Novel technique for testing FPGAs," in *Design, Automation and Test in Europe*, 1998, pp. 89–94. (Cited on page 17.)

[18] M. Renovell, J. M. Portal, J. Figueras, and Y. Zorian, "SRAM-based FPGA's: testing the LUT/RAM modules," in *Proceedings, International Test Conference*, 1998, pp. 1102–1111. (Cited on pages 18 and 25.)

[19] M. Psarakis, D. Gizopoulos, and A. Paschalis, "Test generation and fault simulation for cell fault model using stuck-at fault model based test tools," *Journal of Electronic Testing (JETTA)*, vol. 13, no. 3, pp. 315–319, 1998. (Cited on page 19.)

[20] Y. Ming-yan, Z. Qing-li, W. Jin-xiang, Y. Yi-zheng, and L. Feng-chang, "The design of AMBA AHB/VCI wrapper," in *Proceedings. 5th International Conference on ASIC*, October 2003, pp. 438–442. (Cited on page 33.)

[21] N.-J. Kim and H.-J. Lee, "Design of AMBA wrappers for multiple-clock operations," in *2004 International Conference on Communications, Circuits and Systems (ICCCAS)*, 2004, pp. 1438–1442. (Cited on page 33.)

[22] C. Fan, C. Lan, and Y. Bo, "Designing WISHBONE to AMBA wrapper," in *IEEE 8th International Conference on ASIC (ASICON)*, 2009, pp. 403–406. (Cited on page 33.)

[23] S. Math, R. Manjula, S. Manvi, and P. Kaunds, "Data transactions on system-on-chip bus using AXI4 protocol," in *2011 International Conference on Recent Advancements in Electrical, Electronics and Control Engineering (ICONRAEeCE)*, 2011, pp. 423–427. (Cited on page 34.)

[24] M. Bertola and G. Bois, "A methodology for the design of AHB bus master wrappers," in *Proceedings, Euromicro Symposium on Digital System Design*, 2003, pp. 90–95. (Cited on page 34.)

[25] B. Dutton and C. Stroud, "Soft core embedded processor based built-in self-test of fpgas," in *24th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, DFT '09.*, 2009, pp. 29–37. (Cited on page 34.)

[26] *"LogiCORE IP AXI Interconnect (DS768)"*, v1.05.a ed., Xilinx, Inc., January 2012. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v1_05_a/ds768_axi_interconnect.pdf (Cited on page 37.)

[27] *"EDK Concepts, Tools, and Techniques (UG683)"*, v14.3 ed., Xilinx, Inc., October 2012. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_3/edk_ctt.pdf (Cited on pages 47, 51 and 60.)

[28] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, B. Hutchings, and M. Wirthlin, "RAPIDSMITH," CHREC, Department of Electrical and Computer Engineering, Brigham Young University, Provo, UT 84602, Technical Report and Documentation, September 2012. [Online]. Available: http://rapidsmith.svn.sourceforge.net/viewvc/rapidsmith/trunk/doc/TechReportAndDocumentation.pdf (Cited on pages 48 and 49.)

[29] *"Xilinx 7 Series FPGA and Zynq Libraries Guide for HDL Designs (UG768)"*, v14.3 ed., Xilinx, Inc., October 2012. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_3/7series_hdl.pdf (Cited on page 50.)

[30] *"Zynq-7000 All Programmable SoC: Concepts, Tools, and Techniques (UG873)"*, v14.3 ed., Xilinx, Inc., October 2012. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_3/ug873-zynq-ctt.pdf (Cited on page 51.)

[31] *"Partial Reconfiguration of a Hardware Accelerator on Zynq-7000 All ProgrammableSoC Devices"*, v1.0 ed., Xilinx, Inc., Partial Reconfiguration of a Hardware Accelerator on Zynq-7000 All ProgrammableSoC Devices, January 2013. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp1159-partial-reconfig-hw-accelerator-zynq-7000.pdf (Cited on pages 61, 62, 63, 64, 65 and 70.)

[32] *"Zynq-7000 AP Technical Reference Manual (UG585)"*, v1.5 ed., Xilinx, Inc., March 2013. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf (Cited on pages 63 and 64.)

[33] *"Zynq-7000 All Programmable SoC Software Developers Guide (UG821)"*, v4.0 ed., Xilinx, Inc., March 2013. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug821-zynq-7000-swdev.pdf (Cited on page 66.)

Declaration - Erklärung

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Stuttgart, 04 June 2013, Jiling Wang

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäßaus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Stuttgart, 04 June 2013, Jiling Wang