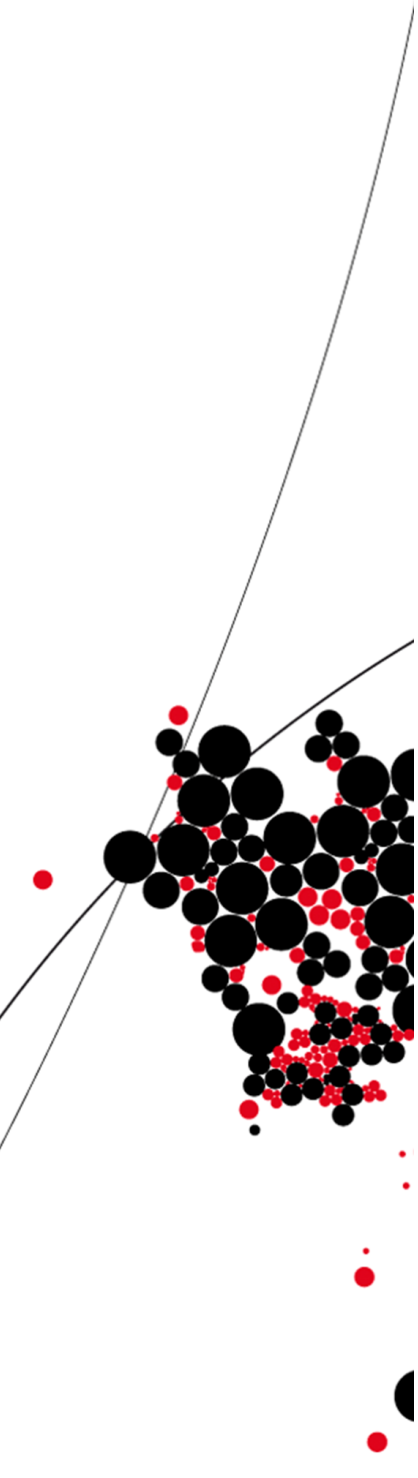
UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering,
Mathematics & Computer Science

# HW-SW co-Design of an On-Chip IJTAG Dependability Processor

**Mochammad Fadhli Zakiy**

**M.Sc. Thesis**
**4 August 2016**



**Supervisors:**
Prof. Dr. Ir. G. J. M. Smit
Dr. Ir. H. G. Kerkhoff
A. M. Y. Ibrahim M.Sc.
Ir. J. Scholten

Computer Architecture and Embedded System Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

# Abstract

Continuous technological advancement enables the growing complexity of System-on-Chip (SoC), so that testing and debugging become harder. Consequently, instrumentation devices need to be embedded into SoCs. Such instrument devices are referred to as embedded instruments, which are intellectual property (IP) blocks that can be accessed externally to test and debug an SoC from inside (on-chip).

Technological advancement also makes SoCs less dependable due to a higher probability of malfunctioning transistors after deployment. Hence, some embedded instruments can be re-used for dependability purposes after deployment such as fault detectors, temperature sensors, voltage sensors, etc. These embedded instruments are accessed externally for testing and debugging, but it can also be accessed internally for dependability purposes. These internal access are employed by an embedded device that executes a dependability application to maintain the dependability of SoC.

Complex SoCs require more embedded instruments. Previously, the increasing number of embedded instruments raised an accessing problem, because it was done in ad-hoc manner. Then in 2014, IEEE 1687 Internal Joint Test Access Group (IJTAG) standard introduced a methodology for accessing embedded instruments in a flexible and standardized way. The standard specified accessing embedded instruments using procedures written in Procedural Description Language (PDL).

IJTAG eases internal access into embedded instruments by using PDL access procedures in a dependability application. This approach makes the complexity of a dependability application grows with the increasing number of PDLs and what kind of application it runs. Hence, an on-chip processor is required to execute a dependability application, thereby the growing complexity of the dependability application does not alter the hardware design that executes it.

This thesis proposes hardware and software co-design of an on-chip IJTAG dependability processor. An on-chip IJTAG dependability processor is an on-chip processor for executing a dependability application as well as accessing embedded instruments on the IJTAG network. The hardware design is based on a single cycle 32 bits Microprocessor without Interlocked Pipeline Stages (MIPS) design that offers a simple and open source processor. Since the dependability application is

written in PDL and is executed in MIPS processor, the software design starts with building a PDL cross-compiler for MIPS. This cross compiler for PDL is developed using ANother Tool for Language Recognition (ANTLR) tool. Finally to verify the on-chip IJTAG dependability processor along with the PDL cross compiler, it is tested to execute benchmark tests and a real dependability application test.

# Acknowledgements

**In the Name of Allah, the Beneficent, the Merciful**

First praise to Allah, the Almighty, which allows this thesis finished within His Greatness. Second, my sincere gratitude to my supervisor Dr. Ir. Hans G. Kerkhoff, for giving me an opportunity to work on this challenging project under his supervision. Third, I would like to thank my daily supervisor Ahmed Ibrahim, for his guidance and constructive discussions. Next, I would also like to thank all of my colleges in the CAES group for the amount of time together.

Last but not least, I would like to thank Indonesian Embassy to the Royal Kingdom of Netherlands, Indonesian Students Association in the Netherlands, Indonesian Students Association in Enschede, University of Twente Muslim Association, Indonesian Moslem Association in Enschede and Islamitische Vereniging Enschede en Omgeving for coloring my past two years in the Netherlands. Special thank you goes to Indonesia Endowment Fund for Education for giving me a chance to study abroad, I will always be in your debt. Another special thank you goes to BASECAMPERS, my little family on this foreign land, you guys are rock. See you guys on the top of the world.

In particular, I would like to thank my family and friends, who supported me during my ups and downs. To my father, Nazief and my mom, Rita, you are the most wonderful parents in the world. And the one who always supports me from far far away, I wish this thesis will become one piece of a puzzle map that leads to you.

# Contents

# List of Figures

# List of Tables

# List of acronyms

| | |
|---|---|
| **ADDI** | Add Immediate |
| **ALU** | Arithmatic Logic Unit |
| **ASIP** | Application-Specific Instruction set Processor |
| **AST** | Abstract Syntax Tree |
| **BIST** | Built-In Self-Test |
| **FPGA** | Field Programable Gate Array |
| **FPU** | Floating Point Unit |
| **H-Array** | Hierarchy Array |
| **ICL** | Instrument Connectivity Language |
| **IJTAG** | Internal Joint Test Access Group |
| **IP** | Intellectual Property |
| **LUI** | Load Upper Immediate |
| **LWC** | Load Word Co-Processor |
| **MFC** | Move From Co-Processor |
| **MTC** | Move To Co-Processor |
| **PDL** | Procedural Description Language |
| $RD$ | Register Destination |
| **RISC** | Reduced Instruction Set Computer |
| $RS$ | Register Source |
| $RT$ | Register Target |

**SoC** System-on-Chip

**SWC** Store Word Co-Processor

**TAP** Test Access Port

**TDR** Test Data Register

# Chapter 1

# Introduction

The concept of dependability was coined by Jean Claude Laprie in 1980s as a trustworthiness of a computer system such that reliance can justifiably be placed on it. Trustworthiness is usually misunderstood by people with usefulness. A system does not have to be trusted to be useful, even a faulty system may produce correct results. But a faulty system has a higher chance of failure which leads to a loss of use.

Technological advancement enables millions of transistors to be implemented into a System-on-Chip (SoC). Then people start to integrate more complex processors, bigger memories and more buses that grows the complexity of SoCs. The growing complexity of SoCs affects on testing and debugging that becomes harder. Consequently, instrumentation devices need to be embedded into SoCs which is known as embedded instruments. Embedded instruments are Intellectual Property (IP) blocks that can be accessed externally to test and debug an SoC from inside (on-chip) such as Built-In Self-Test (BIST) engine, complex I/O characterization and calibration, embedded timing instrumentation, etc.

Technological advancement also raises a dependability issue. The increasing number of transistors means a higher probability of malfunctioning transistors after deployment. Hence, some embedded instruments can be re-used to monitor the malfunctioning transistors and its environment after deployment such as fault detectors, temperature sensors, voltage sensors, etc. These embedded instruments are accessed externally for testing and debugging, but it can also be accessed internally for dependability purposes. These internal access are employed by an embedded device that executes a dependability application to maintain the dependability of an SoC.

The growing complexity of SoCs requires more embedded instruments. Previously, the increasing number of embedded instruments raised an accessing problem, because accessing embedded instruments was done in ad-hoc manner. This problem triggered the emergence of IEEE 1687 Internal Joint Test Access Group (IJTAG) standard that was ratified in 2014. IEEE 1687 IJTAG standard introduced a

methodology for accessing embedded instruments in a flexible and standardized way. Nowadays, embedded instrument vendors are encouraged to present an embedded instrument as an IJTAG wrapped IP block and procedures to access the embedded instrument using the standard. Those procedures are written in Procedural Description Language (PDL) that was also introduced along with the standard.

## 1.1 Motivation

Using IJTAG eases internal access into embedded instruments. Because PDL access procedures, which is originally intended for testing and debugging, can be used in a dependability application to access embedded instruments on the IJTAG network. For example : IJTAG wrapped temperature sensors, fault detectors and voltage sensors are accessed internally just by executing its respective PDL access procedures. Afterwards, the information from embedded instruments are processed and a preventive action can be done if necessary. This will enable the execution of life-time dependability procedures using embedded instruments. This approach requires the dependability application to be written in PDL too. Yet the complexity of a dependability application grows with the increasing number of PDLs and what kind of application it runs. Hence an on-chip processor, whose sole purpose to execute a dependability application, is required. Thereby the growing complexity of the dependability application does not alter the hardware design that executes it.

In general, a processor design needs to consider both hardware and software parts. The hardware of the on-chip processor is a machine that supports to execute a dependability application as well as accessing embedded instruments on the IJTAG network. On the other side the software of the on-chip processor is a machine code of a dependability application that provides what kind of operations that the hardware should do. This thesis describes the design of an on-chip processor from hardware and software perspectives as a co-design for executing a dependability application using IJTAG network, which is specified using PDL.

## 1.2 Problem Statement

The challenge of hardware and software co-design is that the solution can be built unequally. It is possible to have a simple software in a cutting edge hardware that consumes area or a complex software in a simple hardware that takes a lot of time. Certainly, further analysis from hardware and software perspectives are necessary to determine the design requirements.

Dependability application as the software side runs on the processor hardware.

Since the dependability application is written in PDL, it entails the hardware to be able to execute PDL syntaxes. Nowadays, there are many processor options that can be extended to do such operations. Therefore, further investigation is required to study what kind of processor is suitable for executing dependability application as well as accessing embedded instruments on the IJTAG network.

Executing PDL on-chip requires PDL to be compiled into a machine code, which requires a cross compiler for PDL. The compiled machine code will be executed in the on-chip processor as the dependability application. Since the needs of a machine code for PDL is obvious, a cross compiler for PDL is considered to be an important part in this thesis.

Summing up the problems, this thesis is conducted to achieve the following objectives :

1. Analyze the solution for an on-chip processor to execute a dependability application from hardware and software perspectives.

2. Determine and extend a processor design to be able to execute a dependability application written in PDL.

3. Design a cross compiler for translating a PDL code into a machine code for the selected processor.

4. Test the on-chip processor along with the cross compiler for PDL to perform a real dependability application.

## 1.3 Contributions

There are two major contributions in this thesis. First, this thesis contributes the design of an on-chip IJTAG dependability processor. An on-chip IJTAG dependability processor is an on-chip processor that executes a dependability application as well as accessing embedded instruments on the IJTAG network. The software side is a machine code of a dependability application written in PDL.

Second, this thesis contributes a prototype of a PDL cross compiler which has not been explored before. PDL cross compiler compiles PDL syntaxes into a machine code. A compiler generally needs years of development to be able to target many machines and ensures the absence of bugs and errors. This approach might give an insight for making a compiler or a cross compiler for programming languages that has not been explored yet.

What this thesis does NOT contribute to is new ideas in the field of dependability. This thesis does not discuss how to increase the dependability of a circuit.

This thesis focus on easing further development of a dependability application by hardware-software co-design of an on-chip IJTAG dependability processor.

## 1.4 Outline

The first chapter introduces the topic, problem statements and contributions of this thesis.

Chapter 2 describes the related works of this thesis. All related works on IJTAG, hardware and software for building an on-chip IJTAG dependability processor and PDL cross compiler are explained briefly in this chapter.

Chapter 3 analyzes the solution for an on-chip processor to execute a dependability application. This analysis has two point of views : hardware and software perspectives. The result of this chapter is design requirements for hardware and software.

Chapter 4 explains the works related to hardware design of an on-chip IJTAG dependability processor. It implements the hardware design based from the hardware requirements in chapter 3. Then it is concluded with a discussion and the hardware design of an on-chip IJTAG dependability processor.

Chapter 5 describes the works related to PDL cross compiler. It implements the cross compiler design based from the software requirements in chapter 3. Afterwards it is followed with a section for how to use the PDL cross compiler. Subsequently it is concluded with a discussion and the PDL cross compiler design.

Chapter 6 discusses the experimental results for executing a dependability application. It begins with verifying the on-chip IJTAG dependability processor and the PDL cross compiler with benchmark testing. Then it is tested for performing a dependability application. Finally, it is closed with discussion and analysis of the results.

The final chapter concludes this thesis and suggests the future works.

# Chapter 2

# Related Works

An on-chip IJTAG dependability processor is an on-chip processor for executing a dependability application as well as accessing embedded instruments on the IJTAG network. Since a machine code of a dependability application, which is written in PDL, is required, the software design starts with building a cross compiler for PDL. This chapter explains related works that are required to build an on-chip IJTAG dependability processor.

## 2.1 IJTAG

IEEE 1687 IJTAG standard was ratified in 2014. It introduced a methodology for accessing embedded instruments via the IEEE 1149.1 Test Access Port (TAP) [7]. This standard emerged as the solution for widespread development of embedded instruments which had its own access method. IEEE 1687 IJTAG described an instrument-centric approach that allows procedural access to a Test Data Register (TDR) accessible via TAP. The methodology included a network interface (ICL) and a description language (PDL).

Using IJTAG offers a reconfigurable scan network (figure 2.1), which becomes one of the advantages of IJTAG. Accessing a specific instrument on the IJTAG network, which has been specified in ICL, is instantiated within PDL commands. Then dedicated scan vectors are generated for accessing the specific instrument through a process known as retargeting. The following section will explain ICL, PDL and retargeting.

### 2.1.1 ICL

IEEE 1687 defines the purpose of Instrument Connectivity Language (ICL) as a facility to describe the elements that comprise of embedded instrument access network as well as their logical connections to each other. IEEE 1687 uses ICL as

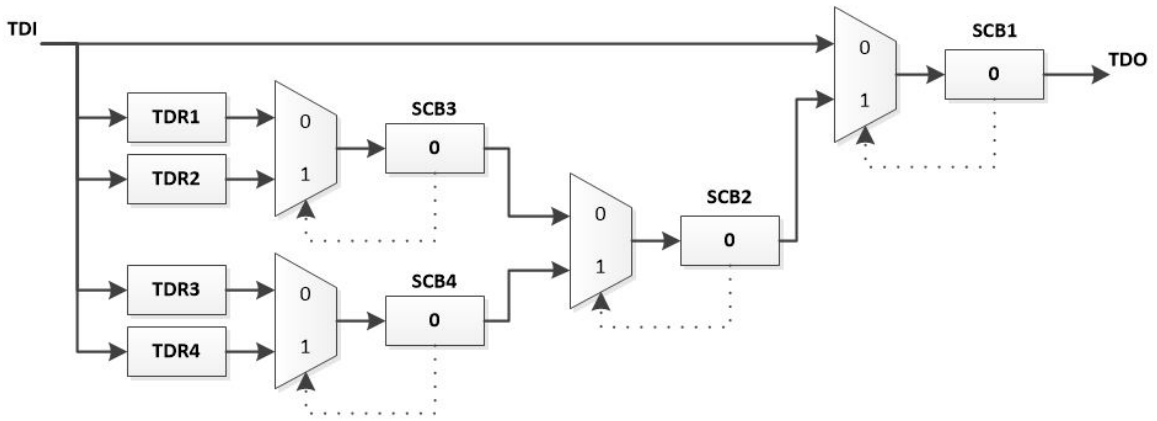



**Figure 2.1:** Example of a reconfigurable scan network

the representation of IJTAG network. ICL calls its fundamental entity as $module$. A device may consist of several $module$s that have hierarchical order with the root module is referred to as $top - level\ module$. Each connections between $module$s may be constructed from primitive building blocks such as multiplexers or storage elements. Even more black box can also be used so that instrument vendors can hide its connections as long as it allows retargeting tool to navigate the network to control and observe any instrument on the network [7].

### 2.1.2 PDL

PDL is used as an amenity to provide a means to define procedures for accessing instruments (embedded instruments). IJTAG standard uses two level of PDL, PDL level 0 and level 1. PDL level 0 is limited for IJTAG related operations where PDL level 1 extends Tcl scripting language that covers what programming languages can do (figure 2.2).

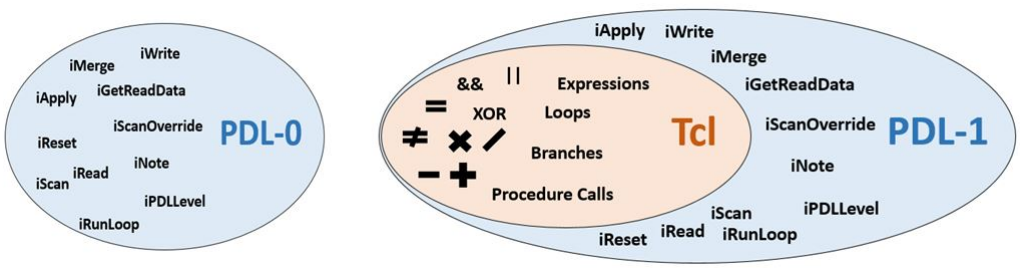



**Figure 2.2:** Comparison of PDL level 0 and 1

Tcl can be used to define mathematical and logical operations, along with expressions, statements, procedure calls, branches, etc. Tcl scripting language has many common syntaxes with C programming language. In terms of mathematical

operations, it behaves similar with C programming language but with different syntaxes (figure 2.3).

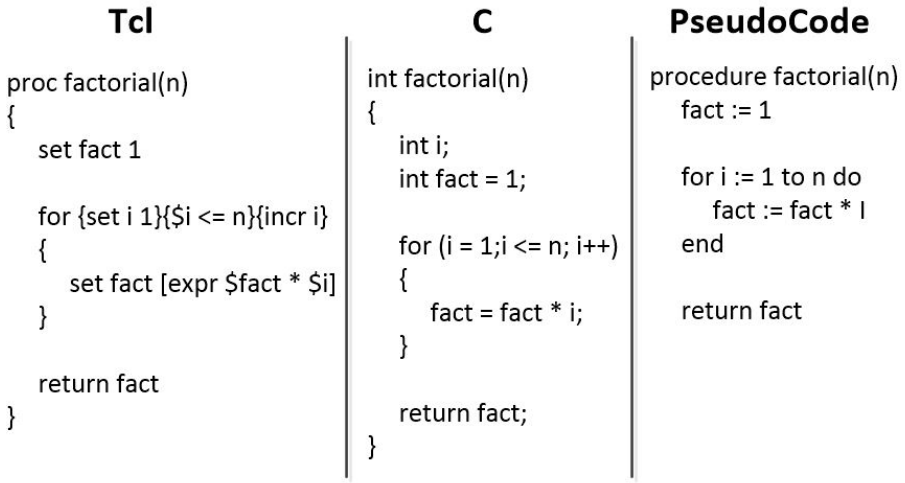



**Figure 2.3:** Comparison of Tcl, C and pseudocode syntaxes

PDL is designed to handle IJTAG related operations. A PDL code starts with an iPDLLevel command to define the PDL level. Then it is followed by iProcsForModule command to define which $module$ in ICL that will execute the procedure. With PDL level 1, the user can specify their own needs in the PDL file using Tcl syntax. For example, listing 2.1 shows an example of a PDL level 1 script for measuring average temperature from two temperature sensors. Commands with 'i-' prefix are IJTAG related commands and the rest are Tcl syntaxes. Requesting a temperature is done by accessing the particular temperature sensor. In this example, it is done by writing 0x1F1F into the temperature sensor within an iWrite command and read the temperature later within an iRead command. However, those iWrite and iRead commands are not executed individually, but in a group. This group of commands consists of iWrite, iRead or iScan commands from previous iApply command into the next iApply command. Each group are executed concurrently. So that in this example there are two concurrent groups, the first one is for writing and the second is for reading. The amount of time for executing a concurrent group is non-deterministic, it depends on the number of instruments, the commands within the concurrent group and the retargeted pattern (which will be explained later in section 2.1.3). PDL also supports execution of waiting state using iRunLoop command. In this example it was used to wait for the temperature sensors to finish, before it is available to be read later. Next the temperatures are assigned into a variable $acc$ using iGetReadData. Then the calculation for measuring average temperature can be done.

**Listing 2.1:** Example of a PDL level 1 script

```
iPDLLevel 1 -version STD_1687_2014;
iProcsForModule Integrator
iProc measureTemp{}
{
    iWrite tempSensor0 0x1F1F //request temperature0
    iWrite tempSensor1 0x1F1F //request temperature1
    iApply

    iRunLoop 2000 -sck        //wait the sensors

    iRead tempSensor0         //fetch the temperature0
    iRead tempSensor1         //fetch the temperature1
    iApply


    set acc [expr iGetReadData tempSensor0]
    set acc [expr acc + iGetReadData tempSensor1]
    set acc [expr acc/2]
}
```

### 2.1.3 Retargeting

Previously, embedded instruments were connected in serial. For example an attempt to access *TDR2* needs to provide a scan vector from *TDR1-TDR4* (figure 2.4).

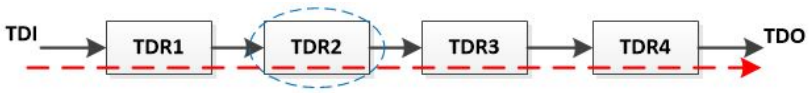



**Figure 2.4:** A scan network before IJTAG

After IJTAG, the *TDR*s can be connected in a multiplexed fashion with logical connections between them (figure 2.5). These connections are documented in ICL, while accessing the TDR is instantiated within PDL commands. In this manner, accessing a TDR does not need to go through all TDRs, but it needs dedicated scan vectors. For example, an attempt to access *TDR2* starts from a PDL commands :

iWrite TDR2 0x101

This command means an access request for writing 0x101 to *TDR2*. On the network level those commands are translated to set ScanMux Control Bit-1 (*SCB1*) to 1 that has initial value 0. This will open the scan network to access *TDR1*, *SCB2* and

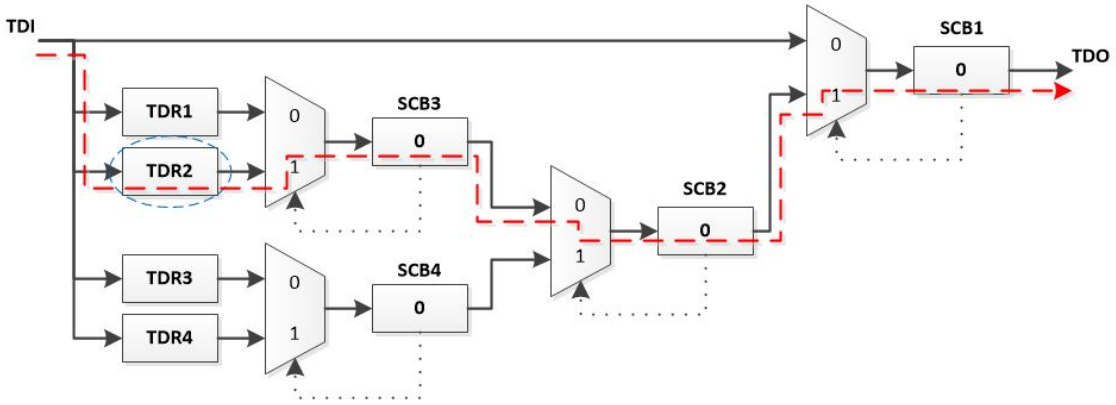



**Figure 2.5:** A scan network after IJTAG

*SCB3* only, because the values of *SCB2* and *SCB3* are initially 0. Then, in the next cycle the values of *SCB1*, *SCB2* and *SCB3* respectively need to be set to 1, 0 and 1. Finally, *TDR2* is accessible. This process of translating an instrument-level pattern (in this example is an iWrite command) into scan vectors is called retargeting.

## 2.2 Retargeting Engine

Retargeting engine is a hardware accelerator for on-chip retargeting. This retargeting engine is proposed by [9]. Retargeting engine is developed under CAES-TDT department in the University of Twente as a part of a dependability manager.

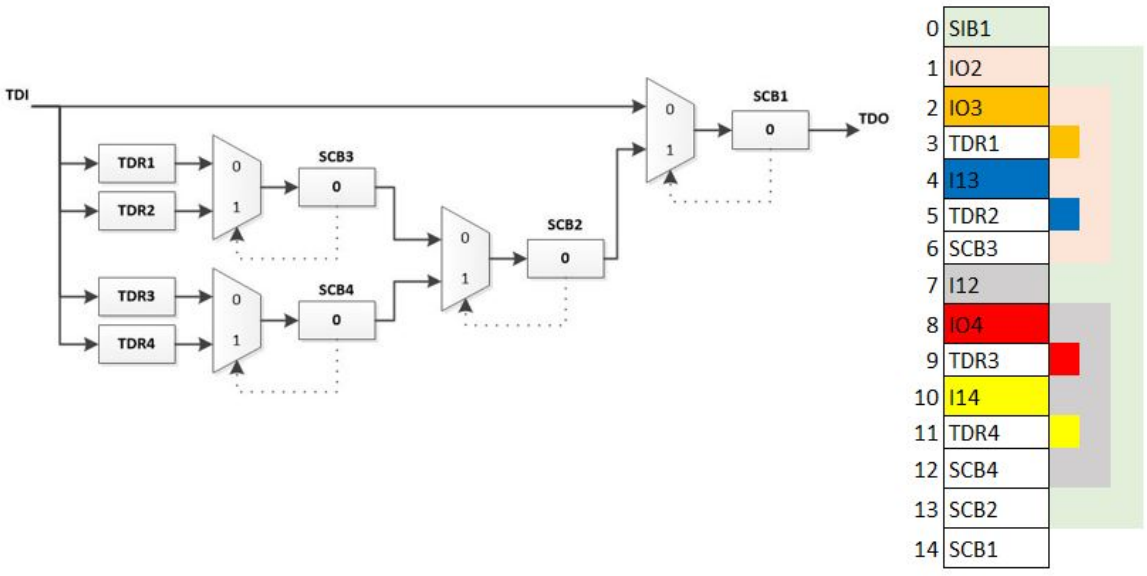



**Figure 2.6:** Example of H-Array representation for a reconfigurable scan network

### 2.2.1 Hierarchy Array

Retargeting requires processing on the network model (ICL) for generating specific scan vectors. Thus the retargeting engine also proposed an on-chip version of ICL which is referred to as Hierarchy Array (H-Array). For example, figure 2.6 shows the H-Array representation of a reconfigurable scan network. The color on the right side of the table represents the network that is included if the same color is activated. For example, *SIB1* (green) will include the whole scan network when it is activated. Meanwhile *IO3* will only include *TDR1*, etc.

### 2.2.2 Retargeting Engine Interface

Retargeting engine interacts with the IJTAG scan network, therefore it has IJTAG ports interface : *SI*, *RST*, *Sel*, *CE*, *SE*, *UE*, *TCK* and *SO* (figure 2.7). Other than the IJTAG related ports, it has seven other ports. Those seven ports comprise of :
1. **RegValue** : provide the value for a register (instrument);
2. **RegID** : provide the value of register ID corresponding to the register index in the H-Array;
3. **Concurrent** : define the concurrent group of commands;
4. **Read/Write** : define the read or write of an access request (1 for read, 0 for write);
5. **ReadID** : return the read ID of a register corresponding to the register index in the H-Array;
6. **ReadValue** : return the read value of a register (instrument);
7. **Error** : Return an alert if an error occurs;


**Figure 2.7:** The Interface of Retargeting Engine

### 2.2.3 How Retargeting Engine Works

Retargeting engine works by first receiving access requests for one or more registers (instruments). Each access request needs to provide the operations (write or read) on *Read/Write* port, register value (used for write value) on *RegValue* port and register ID (correspond to H-Array) on *RegID* port. Retargeting engine executes a group of access requests concurrently. A group of access requests is formed while the *Concurrent* port stays HIGH. So several access requests that are requested while the *Concurrent* port HIGH are considered to be in the same group. After a group of access requests is formed and *Concurrent* port goes LOW, the retargeting engine starts to generate scan vectors for the access requests. Finally, the retargeting engine returns *ReadID* and *ReadValue* if there are read access requests unordered, because the retargeting engine produces effective scan vectors that may access embedded instruments not in order.

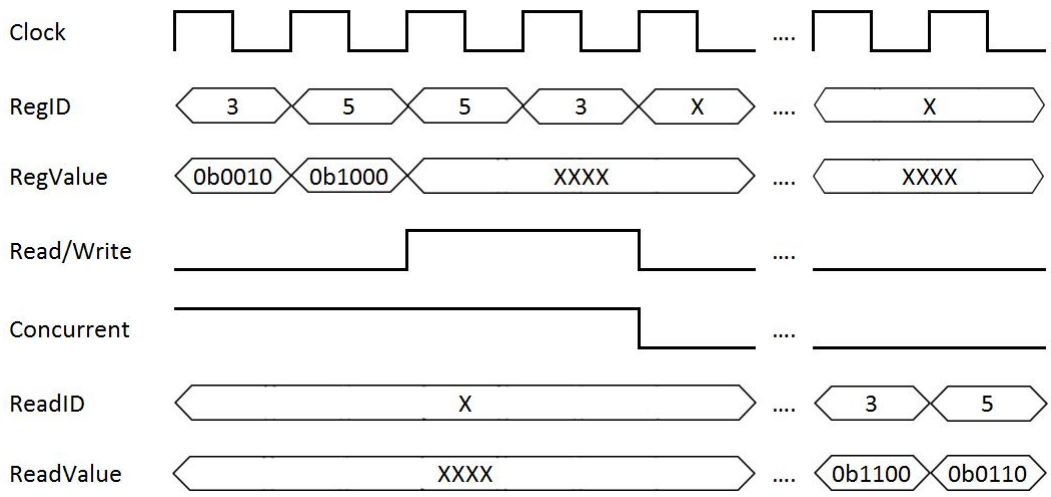



**Figure 2.8:** Example of a group of concurrent access requests

For example, figure 2.8 shows the waveform of a group of access requests for retargeting engine. This example uses the same H-Array that has shown before in figure 2.6. These access requests comprise of writing 0b0010 to *TDR1*, writing 0b1000 to *TDR2*, read from *TDR2* and read from *TDR1*, while the *Concurrent* port stays HIGH. When *Concurrent* port goes to LOW, retargeting engine interprets it as the end of a group of access requests and starts to generate scan vectors for these access requests. Since the returning of read access requests are unordered, it is possible to get the value of *TDR1* first followed by *TDR2* although the orders are the other way around.

## 2.3 Compiler

Compiler is a computer program that reads a program written in one language (source language) and translates it into another language (target language) [1] (figure 2.9). The target language of a compiler is generally a machine executable language (machine code). The first complete compiler was implemented by John Backus who compiled FORTRAN into IBM 704 computer in 1957.


**Figure 2.9:** Compiler [1]

### 2.3.1 Compiler Phases

As a translator from source language into another language, a compiler works in several phases. Alfred Aho in the infamous *dragon book* [1], defined six phases of a compiler. It consists of :

1. **Lexical Analyzer** : Reads the characters in the source program and returns stream of tokens;
2. **Syntax Analyzer** : Imposes hierarchical structure on the token stream;
3. **Semantic Analyzer** : Ensures the declarations and statements are semantically correct;
4. **Intermediate Code Generator** : Generates intermediate representations of the source program (optional);
5. **Code Optimizer** : Improves the source-represented code in order to produce faster machine code (optional);
6. **Code Generator** : Generates target code.

For example, a program that counts a position from *initial* position and *rate* is shown in figure 2.10. *Lexical Analyzer* changes the *position*, *initial* and *rate* into $id_1$, $id_2$ and $id_3$ respectively. Then the *Syntax Analyzer* produces the correct parse tree and the *Semantic Analyzer* fills the parse tree with correct types. Next the *Intermediate Code Generator* generates intermediate representations of the parse tree. Furthermore, the *Code Optimizer* improves the intermediate representations and finally the *Code Generator* generates the target code.

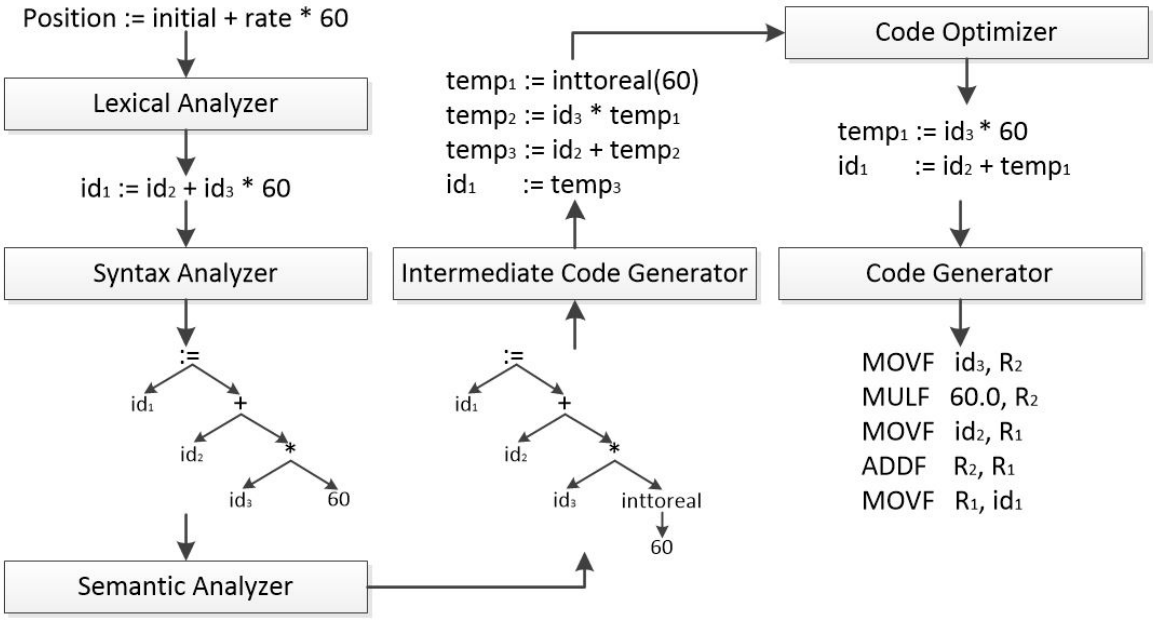



**Figure 2.10:** Example of phases in compiling a program [1]

## 2.4 Cross Compiler

Cross compiler is a compiler that can generate a machine code for another platform other than the platform where the compiler is running [10]. This approach is used to compile a machine code for a platform that is not capable to run a compiler for itself. The history of cross compiler dated back in 1979 when it was impossible to compile ALGOL 68 to Z80 CPU due to insufficient memory. Then ALGOL 68 code was compiled in other platform to generate ZCODE for Z80 CPU.

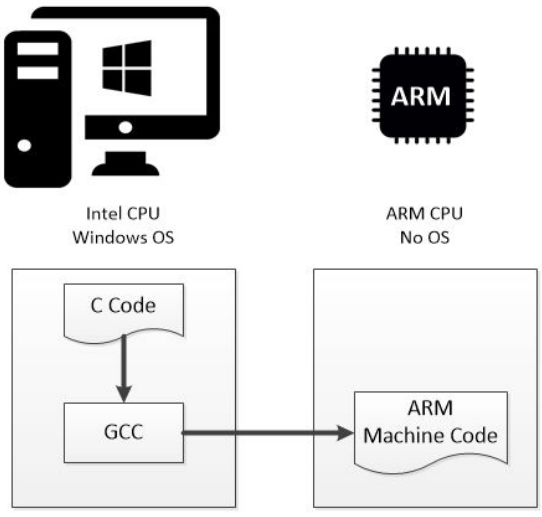



**Figure 2.11:** Example of cross compiler

Nowadays, cross compiler is used to compile a machine code for embedded processors that has no operating system or a platform with limited systems like mobile phone. For example, An Intel CPU with Windows operating system compiles C source code into ARM machine code for ARM CPU that has no operating system (figure 2.11). The compiler result is not executable for the host machine (Intel CPU) but it is executable for ARM CPU.

# Chapter 3

# Analysis of HW-SW Co-Design

An on-chip IJTAG dependability processor is a dedicated processor for executing a dependability application. While executing a dependability application, the processor needs to access embedded instruments on the IJTAG network. This makes the on-chip IJTAG dependability processor falls into Application-Specific Instruction set Processor (ASIP) category. Hence, ASIP design methodology [2] is required as a framework to design an on-chip IJTAG dependability processor from hardware and software sides.

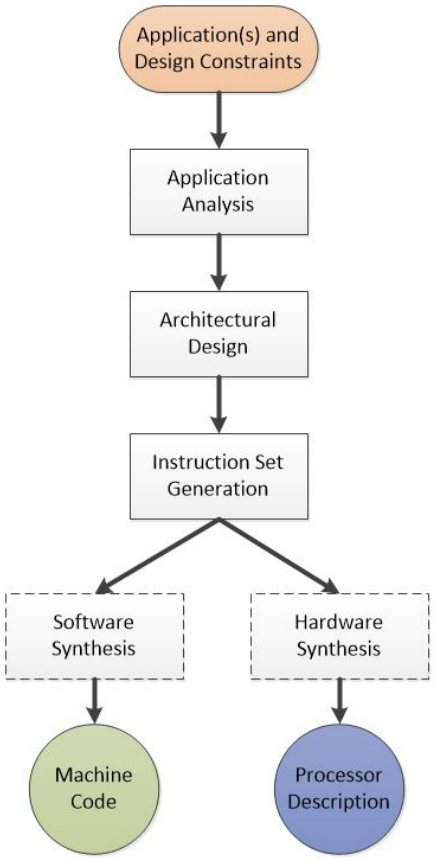



**Figure 3.1:** ASIP Design Methodology [2]

The input of ASIP design methodology is an application and the design constraints. The application is a dependability application written in PDL. Section 2.1.2 explained that PDL consists of PDL level 0 and 1, hence PDL level 1 extends Tcl scripting language that enables what programming languages can do such as mathematical and logical operations along with branches, loops, statements, etc. This thesis will implements all PDL operations in PDL level 1 to be able to execute what programming languages can do. Within PDL there are several commands that instantiate retargeting. Thus, there is a previous work that proposed an on-chip retargeting which is referred to as retargeting engine [9]. Hence the design requirements of an on-chip IJTAG dependability processor comprise of : enable to execute PDL and re-use the retargeting engine as a co-processor. As for design constraints, since the on-chip IJTAG dependability processor is a dependability system of an SoC, it must be very reliable. Such condition can be achieved by having a simple processor as possible that has lower probability of malfunctioning transistors.

ASIP design methodology comprises of 4 steps that cover :

1. **Application Analysis** : Analyze what kind of application that the processor can do.
2. **Architectural Design** : Explore possible architectures using step 1 as the given design constraints.
3. **Instruction Set Generation** : Generates instruction sets for an on-chip IJTAG dependability processor.
4. **Software and Hardware Synthesis** : Machine code generator and processor design.

This chapter explains the first 3 steps, meanwhile the hardware synthesis and software synthesis will be discussed in the next chapter as hardware and software implementations.

## 3.1 Application Analysis

In this thesis PDL becomes an important part because it is used to write the dependability application. PDL level 1 as an extension of Tcl sciprting language can be used to define mathematical and logical operations, along with expressions, statements, procedure calls, branches, etc. Hence in order to execute PDL, at least the hardware and software parts of an on-chip IJTAG dependability processor needs to be able to provide those operations.

PDL also has 24 commands that is defined in the IJTAG standard (table 3.1). However this thesis only implements iPDLLevel, iReset, iRead, iWrite, iApply, iRunLoop and iGetReadData commands (table 3.2). Because these 7 commands are

**Table 3.1:** PDL commands [7]

| Command | Purpose |
|---|---|
| iPDLLevel | Identify PDL level |
| iPrefix | Specify hierarchical prefix |
| iReset | Reset the network |
| iWrite | Queue data to be written |
| iRead | Queue data to be read |
| iScan | Queue data to be scanned |
| iOverrideScanInterface | Indicate the capture, update and broadcast behavior to be imposed on a list of scan interfaces |
| iApply | Execute queued operations |
| iClock | Specify a system clock which is required to be running |
| iClockOverride | Override definition of system clock when it is generated on-chip |
| iRunLoop | Issue a number of clocks |
| iProc | Wrapper for a PDL |
| iCall | invoke a PDL procedure |
| iProcsForModule | Identify the module in the ICL with which subsequent iProcs are associated |
| iUseProcNameSpace | Use namespace for subsequent iCalls |
| iNote | Send text to runtime |
| iMerge | Allow merging of iCalls |
| iTake | Disallow other merge threads from modifying a model resource |
| iRelease | Re-allow other merge threads to modify a model resource |
| iState | Document the current state of the network |
| iGetReadData | Return the value from most recently applied iRead operation |
| iGetMiscompares | Return the XOR of the value from most recently applied iRead operation |
| iGetStatus | Return the decimal number of iApply miscompares that have occured since the last time iGetStatus was issued |
| iSetFail | Return the message string to the controlling program to indicate an unexpected condition |

the fundamental commands that enables to use IJTAG network. iPDLLevel command translates the PDL code depends on its level. iReset command resets the IJTAG network. iRead and iWrite commands queue the data to be executed and these queues are only executed when iApply command is given. iRunLoop command issues a number of clocks, it is usually used for waiting instruments to finish its process. Finally iGetReadData command is used to fetch the data that has been read by an iRead commands.

From those 7 commands iRunLoop, iGetReadData and iPDLLevel commands are only able to be executed in the software side. In the implementation of iRunLoop, it can instantiate No Operation (NOP) instructions to make a waiting state until the required time is fulfilled (explaied later in section 5.6). iGetReadData command only enables the data that has just been fetched for further processing, this can be implemented by moving the data from the retargeting engine into main processor. Finally iPDLLevel does not need any instructions, this can be done in compiler level to check whether the PDL level is correct. Nonetheless iReset, iWrite, iRead and iApply require serious software implementations, but these commands also require to be generated as instructions in the hardware side. Because these commands interact with the retargeting engine directly.

**Table 3.2:** Implementation of PDL commands

| Command | Implemented in |
|---|---|
| iPDLLevel | SW |
| iReset | SW-HW |
| iWrite | SW-HW |
| iRead | SW-HW |
| iApply | SW-HW |
| iRunLoop | SW |
| iGetReadData | SW |

Meanwhile, the other 17 PDL commands are not going to be implemented in this thesis with specific reasons. iPrefix, iUseProcNameSpace, iProcsForModule, and iProc commands are not necessary because the retargeting engine substitutes the ICL with H-Array, this also affects on the implementation of iCall command since iProc command is not implemented. iOverrideScanInterface and iState commands are optional because overriding and documentation are not fundamental. iTake, iMerge and iRelease commands enable threading for parallel processing, these commands are optional for improving the performance later. iClock and iClockOverride commands are not necessary to be implemented, because this thesis assumes to only have one clock source, the system clock. iScan is not implemented because it behaves similar to iWrite and iRead but in more detail, thus iScan can be substitued with iWrite and iRead commands. Finally, iNote, iGetStatus, iGetMiscompares and iSetFail commands behave as notification system for the user which are not applicable since an on-chip IJTAG dependability is an embedded processor.

Thereby, the requirements to execute PDL comprise of :

**1.** Being able to perform common programming language ability. Such as mathematical and logical operations, expressions, statements, procedure calls and branches;

**2.** Being able to perform iReset, iRead, iWrite and iApply commands for retargeting engine co-processor.
**3.** Being able to perform iRunLoop, iGetReadData and iPDLLevel commands in the software side.

## 3.2 Architectural Design

Architectural design explores possible architecture based on the requirements of : executing PDL and re-use retargeting engine as a co-processor. The requirements to execute PDL has been explained on section 3.1. There are many processor types that meet such requirements such as Intel, ARM, Power PC, LEON and MIPS processors. Thus most of it falls under proprietary rights that requires a license to produce, use and/or synthesize. Only the early generation of MIPS and LEON that is available.

The only constraint of an on-chip IJTAG dependability processor is the hardware design should be as simple as possible in terms of area. Compared to early generation of MIPS, LEON processor requires vast area which does not meet the constraint. Since the early generation of MIPS was developed in 1985, until now researchers around the world has explored several variants from the early generation of MIPS, such as Mini MIPS [11], Fault tolerant MIPS [12] and Single cycle 32 bits MIPS [5]. From these three options Single Cycle 32 bits MIPS offers the simplest architecture from the remaining options. Hence the design of an on-chip IJTAG dependability processor is based on the architecture of Single cycle 32 bits MIPS.

Single cycle 32 bits MIPS covers what general embedded processor can do. It can handle mathematical and logical operations, jumps, branches, load and store. Statements are done by assigning a value to a register, while loops and procedure calls are handled by performing jumps into a specific address. Single cycle 32 bits MIPS is also able to integrate retargeting engine as a co-processor by following MIPS specification [4]. This section will explains the analysis of how single cycle 32 bits MIPS can meet the design requirements without violating the constraint.

### 3.2.1 Fixed Point Representation

Mathematical operations in computer system sometimes deal with real number operations. Usually, it is handled by a Floating Point Unit (FPU). Yet the initial design of single cycle 32 bits MIPS [5] does not include co-processor 0 (trap and exception handler) and 1 (FPU). OpenCores.org provides an open source FPU core [13] that can be integrated into MIPS processor. However the synthesis result in 0.18 $\mu$

technology shows that FPU is 70 % larger than the whole single cycle 32 bits MIPS processor (Table 3.3). It is clear that integrating FPU is not an option for the sake of simplicity. Hence there is another option for a processor without FPU to calculate real numbers by changing the number representation into fixed point in the software side.

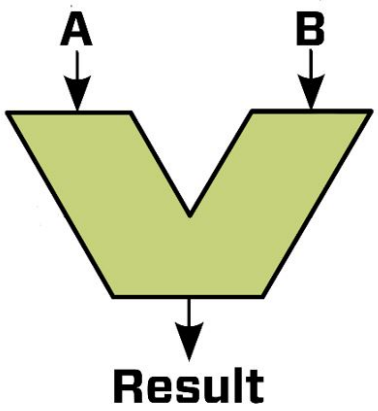
**Table 3.3:** Area comparison Single Cycle 32 bits MIPS & FPU

| Hardware | Area $\mu^2$ |
|---|---|
| Single Cycle 32 bits MIPS | 193551.73 |
| FPU [13] | 331635.98 |

### 3.2.2 Software Emulated Operations

Arithmatic Logic Unit (ALU) is a digital circuit that can perform mathematical and logical operations. It is the main building block of a processor. The ALU of modern processors can perform complex mathematical and logical operations. In this case, it can process complex operations extremely fast. However it consumes enormous areas than a simple ALU that can only do add and shift operations.

**Table 3.4:** Single cycle 32 bits MIPS ALU support [5]




| Name | Operation |
|---|---|
| ADD | Result = A + B |
| AND | Result = A & B |
| LUI | Result[31..16] = B[15..0] |
| OR | Result = A \| B |
| SLL | Result = A << B |
| SLT | if(A<B); Result = 1; else Result = 0 |
| SRL | Result = A >> B |
| SUB | Result = A - B |
| XOR | Result = A XOR B |

The ALU of single cycle 32 bits MIPS supports several mathematical and logical operations [5] (Table 3.4). It supports ADD, AND, LUI, OR, SLL, SLT, SRL, SUB and XOR operations, but it does not support for multiplication (MULT) and division (DIV) operations. On the other hand multiplication and division are fundamental operations and are required in several dependability applications along with square root and power operations. There are two approaches to solve this problem. The first approach is to implement those required operations in the ALU of single cycle

32 bits MIPS. And the second approach is to emulate those required operations in the software side.

Multiplication and division are complex operations. Multiplication hardware comprises of adders and shift registers. Divider hardware is even more complex than multiplication hardware, because it needs to handle exclusive cases such as division by zero and remainders. The main reason to use single cycle 32 bits MIPS processor is that it offers the simplest processor for the hardware of dependability system. Therefore, the second option is preferable, multiplication and division operations will be emulated in the software side. This solution also applies for other complex operations that are required such as square root and power.

## 3.3 Instruction Set Generation

Instruction set connects the hardware and the software sides. The software side compiles a PDL file into a set of instructions and the hardware side executes this set of instructions. This thesis uses MIPS instruction set from [3] [4]. However there are PDL requirements to generate iReset, iRead, iWrite and iApply commands into instructions that has been discussed in section 3.1 (table 3.2), therefore the MIPS instruction set needs to be extended. Before that, MIPS instruction set must be investigated first.

Majority of MIPS instructions fall into three categories: R-Type, I-Type and J-Type [3]. Register-Type (R-Type) instructions are used when all the data values are located in registers. Immediate-Type (I-Type) instructions are used when the instructions must operate with an immediate value. Finally Jump-Type (J-Type) instructions are used to perform a jump to an address.

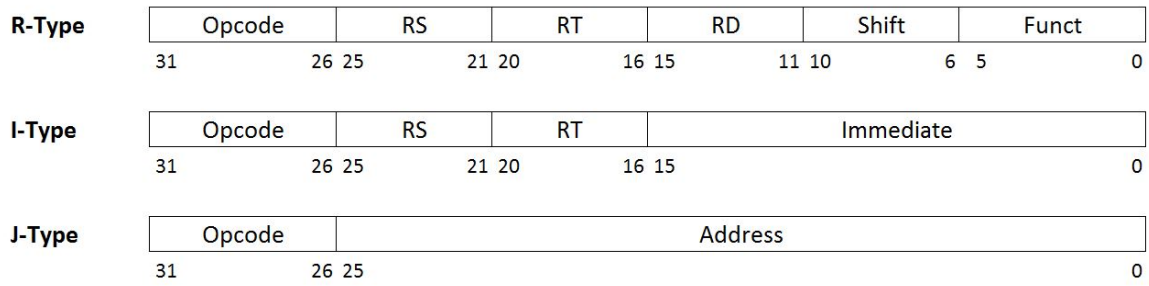



**Figure 3.2:** MIPS instruction formats [3]

Each instruction type has its own instruction format (figure 3.2). All instructions have an $opcode$ part on the first 6 bits of its most significant bits, where the rest may differ for each type. With Register Destination ($RD$), Register Source ($RS$) and Register Target ($RT$), R-Type instructions can access 2 registers in register file

simultaneously for its operations. The R-Type instructions also have a $shift$ part for shifting amount and a $funct$ part for defining the function for its operations. On the other side, I-Type allocates its least significant 16 bits to an immediate value. Meanwhile, J-Type allocates its least significant 26 bits outside of $opcode$ for a jump address.

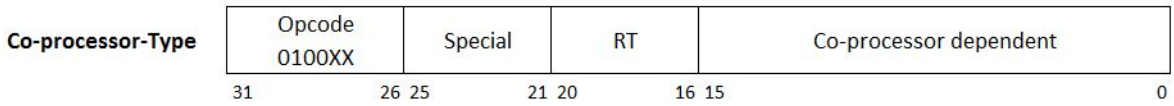



**Figure 3.3:** MIPS Co-Processor Type Instruction Formats [4]

Outside of R, I and J-types, MIPS processors also support another type that is co-processor instructions. These instructions enable MIPS main processor to command its co-processors and exchange data between them [4]. The $opcode$ of co-processor type is '0100XX' where the last two bits refer to a specific co-processor (figure 3.3). The format of co-processor type instructions depend on its co-processor. For example, figure 3.4 depicts how Move From Co-Processor (MFC) instructions for co-processor 0, 1 and 2 have different formats. MFC0 moves a data to register $RT$ in the main processor from register $RD$ in co-processor 0 with specific $sel$. MFC1 also moves a data to register $RT$ in the main processor from register $FS$ in co-processor 1, this data movement is used between MIPS processor with FPU co-processor. Nonetheless, MFC2 moves a data to register $RT$ in the main processor, but the $implementation$ part in the co-processor side depends on the co-processor designer to implement it.

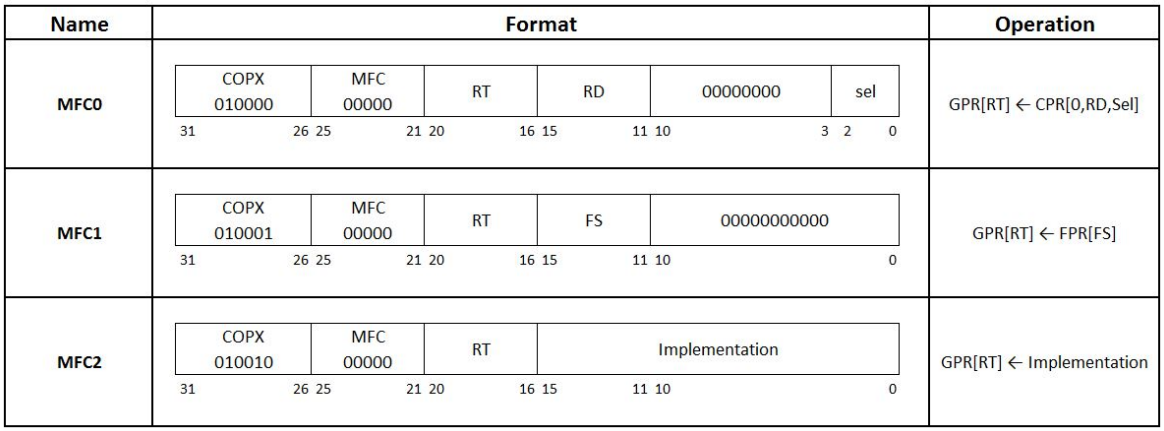



**Figure 3.4:** MFCX instruction formats [4]

Integrating retargeting engine as a MIPS co-processor requires instruction set design. First, retargeting engine co-processor is assigned into co-processor 2, so that the hardware design of an on-chip IJTAG dependability processor does not alter

general MIPS architectures that employ co-Processor 0 and 1. Since retargeting engine enables write and read into an the IJTAG network, it needs to be able to move data to and from the co-processor. From co-processor point of view there are two sources/destinations for data movements which are main processor and memory (figure 3.5). Move a data from co-processor to MIPS processor can be handled with MFC instruction and move a data to co-processor from MIPS processor can be handled with Move To Co-Processor (MTC) instruction. For data movements between memory and co-processor, Store Word Co-Processor (SWC) and Load Word Co-Processor (LWC) instructions can be used to store and load the data respectively. In the co-processor side, a register file will be added to hold the data and to ease data movements.

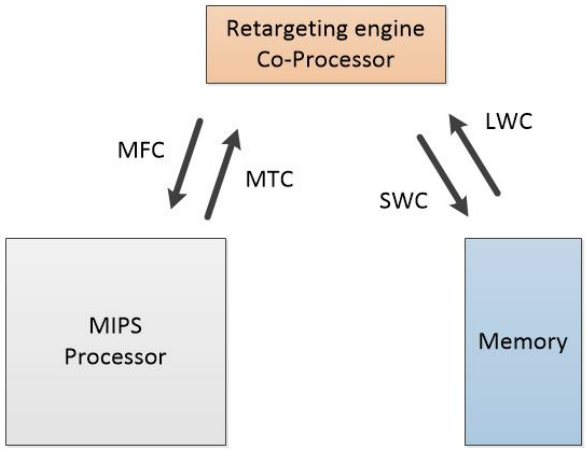



**Figure 3.5:** Co-Processor Data Movement

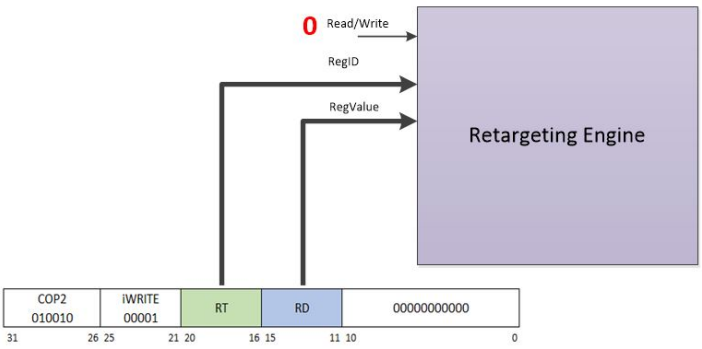



**Figure 3.6:** Mapping iWRITE instruction to retargeting engine

To fulfil PDL requirements the hardware needs to generate iWrite, iRead, iApply and iReset PDL commands into MIPS instructions. As explained in section 2.2, retargeting engine requires two data (*RegID* and *RegValue*) to process an iWrite command. In the hardware implementation, those two data requires two registers

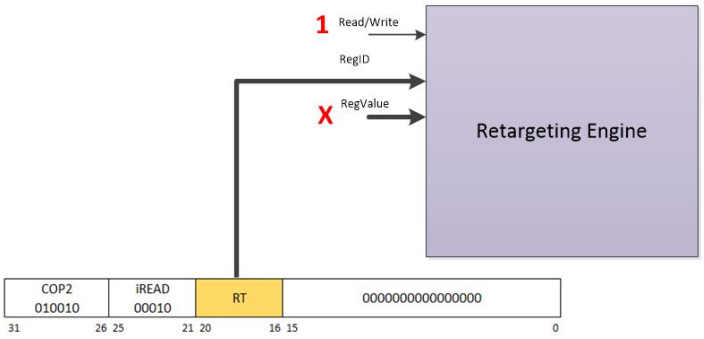



**Figure 3.7:** Mapping iREAD instruction to retargeting engine

which can be placed in $RT$ and $RD$ because $RS$ has been reserved for $special$ part (figure 3.3). In the instruction format, iWRITE instruction takes the usual place of $RT$ and $RD$ for representing *RegID* and *RegValue* (figure 3.6). Meanwhile iRead command only requires one data (*RegID*), so the iREAD instruction can take usual place of $RT$ for representing *RegID* (figure 3.7). Finally iApply and iReset commands need no data, so iAPPLY and iRESET instructions can be implemented by $opcode$ and $special$ only. Although generating iAPPLY instruction is just simply add a new instruction, but the real work is in the software side that will be explained in section 5.6. Table 3.5 shows the instructions along with its formats and operations for retargeting engine co-processor.

**Table 3.5:** Retargeting engine co-processor instructions

| Name | Format | Operation |
|---|---|---|
| **MFC** | COP2 (31–26) 010010 \| MFC (25–21) 00000 \| RT (20–16) \| RD (15–11) \| 00000000000 (10–0) | RT ← COP2[RD] |
| **MTC** | COP2 (31–26) 010010 \| MTC (25–21) 00100 \| RT (20–16) \| RD (15–11) \| 00000000000 (10–0) | COP2[RD] ← RT |
| **LWC** | COP2 (31–26) 010010 \| LWC (25–21) 01010 \| RT (20–16) \| RD (15–11) \| Immediate (10–0) | COP2[RT] ← MEM[RD + Immediate] |
| **SWC** | COP2 (31–26) 010010 \| SWC (25–21) 01011 \| RT (20–16) \| RD (15–11) \| Immediate (10–0) | MEM[RD + Immediate] ← COP2[RT] |
| **iWrite** | COP2 (31–26) 010010 \| iWRITE (25–21) 00001 \| RT (20–16) \| RD (15–11) \| 00000000000 (10–0) | RegID ← COP2[RT]<br>RegValue ← COP2[RD]<br>IWRITE (RegID, RegValue) |
| **iRead** | COP2 (31–26) 010010 \| iREAD (25–21) 00010 \| RT (20–16) \| 0000000000000000 (15–0) | RegID ← COP2[RT]<br>IREAD (RegID) |
| **iReset** | COP2 (31–26) 010010 \| iRESET (25–21) 00011 \| 00000000000000000000 (20–0) | IRESET |
| **iApply** | COP2 (31–26) 010010 \| iAPPLY (25–21) 00101 \| 00000000000000000000 (20–0) | IAPPLY |

## 3.4 Discussion

This chapter describes the analysis of hardware and software co-design of an on-chip IJTAG dependability processor. It uses ASIP design methodology [2] that comprises of 4 steps : Application Analysis, Architectural Design, Instruction Set Generation and Software-Hardware Synthesis. This chapter only covers the first 3 steps and leaves the last step for the implementation chapters later.

The input of ASIP design methodology is application and design constraints. In this thesis the application is a dependability application that is written in PDL. PDL has two levels 0 and 1. PDL level 1 is developed as an extension of Tcl that can be used to define mathematical and logical operations, expressions, statements, procedure calls and branches. This thesis treats all PDL commands as PDL level 1. Within PDL there are several commands that instantiate retargeting. Thus, there is a previous work that proposed an on-chip retargeting referred to as retargeting engine [9]. Hence the design requirements of an on-chip IJTAG dependability processor are : executing PDL and re-use the retargeting engine as a co-processor. There is only one design constraint which is the hardware design must be very reliable, such condition can be achieved by having a simple processor as possible that has lower probability of malfunctioning transistors.

On the Application Analysis step, the requirements for PDL is explained. Since dependability application is written in PDL, it inherits what PDL can provide. PDL level 1 as an extension of Tcl can be used to define mathematical and logical operations, expressions, statements, procedure calls and branches. Moreover PDL has 24 PDL commands (not part of Tcl) that are IJTAG related. This thesis only implements 7 commands which are fundamental to use IJTAG : iWrite, iRead, iApply, iReset, iRunLoop, iPDLLevel and iGetReadData commands. Hence only iWrite, iRead, iApply and iReset commands that will be generated into instructions because it will interact directly with the retargeting engine.

On the Architectural Design step, single cycle 32 bits MIPS processor is chosen as the base of an on-chip IJTAG dependability processor, because single cycle 32 bits MIPS processor offers the simplest and open source processor architecture. To meet the design constraints which are having a simple hardware design, co-processor 0 and 1 are not included. Hence, real number operations will be compensated in the software side. Complex arithmetic operations such as division, multiplication, power and square root are not implemented in the ALU, but it will be emulated in the software side as well to keep the hardware design as simple as possible.

Instruction Set Generation step determines instruction set for an on-chip IJTAG dependability processor. This thesis uses MIPS instruction set [3] [4] for handling

mathematical, logical, jumps, branches and what common processor can do. The requirement for re-using retargeting engine co-processor also requires instruction set design. First the retargeting engine is placed in co-processor 2 so that it does not alter general MIPS architectures. Then it is followed with generating iWrite, iRead, iApply and iReset commands as MIPS co-processor type instructions for IJ-TAG related operations. Finally other co-processor instructions are added for data movement from and to co-processor such as MFC, MTC, SWC and LWC instructions.

From those ASIP design methodology, hardware and software design requirements can be concluded. Hardware design requirements comprise of :
1. Implement retargeting engine as MIPS co-processor
2. Implement the co-Processor type instructions for retargeting engine co-processor.

Software design requirements comprise of :
1. Enable to translate PDL syntaxes into MIPS machine code
2. Map PDL commands into co-processor type instructions
3. Use fixed point representation
4. Compensate complex hardware operations by emulating multiplication, division, square root and power operations in the software side.

These design requirements will be implemented separately in hardware and software.

# Chapter 4

# IJTAG Dependability Processor

As explained in chapter 3, the hardware side of an on-chip IJTAG dependability processor design based on a single cycle 32 bits MIPS [5]. The analysis on that chapter produces hardware requirements to extend the design of a single cycle 32 bits MIPS to :
1. Implement retargeting engine as MIPS co-processor
2. Implement the co-Processor type instructions for retargeting engine co-processor.

This chapter explains how the initial design of single cycle 32 bits MIPS is extended to achieve those previous hardware requirements. It starts with brief explanation about a single cycle 32 bits MIPS. Then it is followed by hardware design and is closed with a discussion. The result of this chapter is a hardware design of an on-chip IJTAG dependability processor.

## 4.1 Single Cycle 32-bits MIPS

MIPS stands for Microprocessor without Interlocked Pipeline Stages. The research of MIPS was started in 1981, led by John L. Hennessy. It yielded the first MIPS processor in 1985. Nowadays, the first generation of MIPS becomes the most studied processor that can be accessed easily in [3]. This ignites researchers around the world to investigate and produce many variants of first generation MIPS, such as Mini MIPS [11], Fault tolerant MIPS [12] and Single cycle 32 bits MIPS [5].

### 4.1.1 MIPS Stages

MIPS is a Reduced Instruction Set Computer (RISC) that suits for general processor. There are 5 operations that each instruction holds which are : Instruction Fetch (IF); Instruction Decode (ID); Execute (EX); Memory Access (MEM); and Write

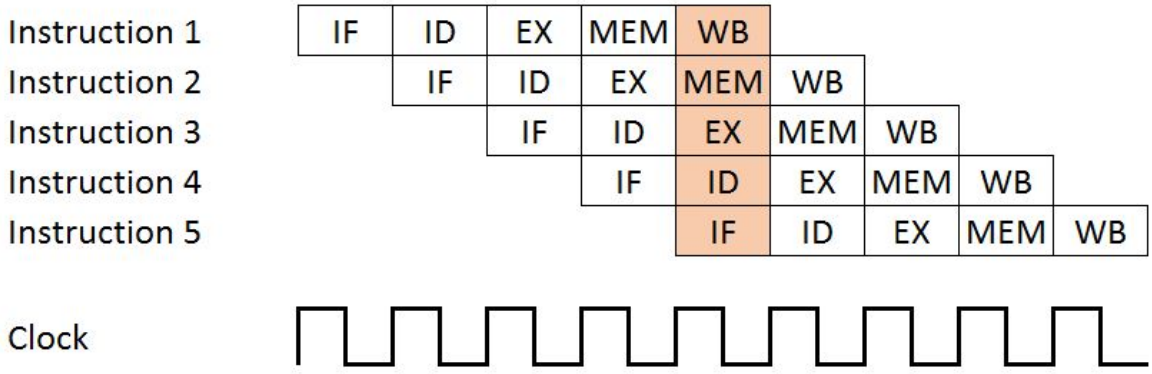



**Figure 4.1:** MIPS stages [3]

Back(WB). Those stages are done in pipeline manner (figure 4.1). On the orange highlighted clock cycle, MIPS processor executes WB for instruction 1, MEM for instruction 2, EX for instruction 3, ID for instruction 4 and IF for instruction 5.

Single ccle 32 bits MIPS processor is a 32 bits RISC processor that preceded modern MIPS architectures. Single cycle 32 bits MIPS executes all five stages (IF, ID, EX, MEM and WB) in a single clock cycle (figure 4.2). In return, this architecture can not work in high frequency clock.

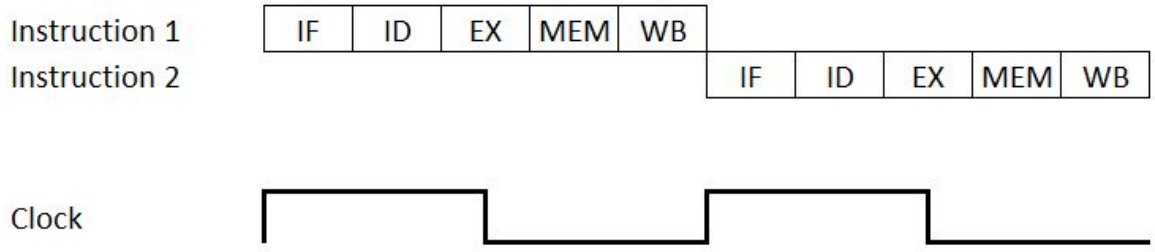



**Figure 4.2:** Single Cycle MIPS stage [5]

### 4.1.2 MIPS Register

MIPS processor has 32 general purpose registers. These registers are placed in a register file. MIPS assembly language employs a convention for use of registers. This convention must be followed by MIPS assembly language programmers in order to avoid unexpected behaviours of module that is written by different people. These 32 general purposes registers has its own usage [8] (table 4.1). Register $\$0$ is hard-wired into 0 and is not allowed for holding data. Register $\$at$ is a temporary register that its use is limited to assembler. Register $\$v0$ and $\$v1$ are used to hold return values from functions. Register $\$a0$ - $\$a3$ serve as arguments to functions. Register $\$t0$ - $\$t9$ are used for temporary storage that is not preserved when MIPS processor

calls subprograms. Register $\$s0$ - $\$s7$ are saved registers that is preserved when MIPS processor calls subprograms. Register $\$k0$ and $\$k1$ are reserved by kernel and are restricted to use. Finally register $\$gp$, $\$sp$, $\$fp$ and $\$ra$ are used for global pointer, stack pointer, frame pointer and return address respectively.

**Table 4.1:** MIPS register [8]

| Register Number | Conventional Name | Usage |
|---|---|---|
| $0 | $zero | Hard-wired to 0 |
| $1 | $at | Assembler temporary |
| $2 - $3 | $v0,$v1 | Return values from functions |
| $4 - $7 | $a0 - $a3 | Arguments to functions |
| $8 - $15 | $t0 - $t7 | Temporary registers, not preserved by subprograms |
| $16 - $23 | $s0 - $s7 | Saved registers, preserved by subprogram |
| $24 - $25 | $t8,$t9 | More temporary data, not preserved by subprograms |
| $26 - $27 | $k0,$k1 | Reserved by kernel. Do not use |
| $28 | $gp | Global pointer |
| $29 | $sp | Stack pointer |
| $30 | $fp | Frame pointer |
| $31 | $ra | Return address |

### 4.1.3 MIPS Co-Processor

MIPS R2000, the first generation of MIPS, is able to embed four co-processors. This specification is still preserved until the latest model of MIPS architectures [4]. The first two co-processors have been reserved in MIPS architectures (figure 4.3). Co-Processor 0 handles traps, exceptions, interrupt service routines and virtual memory. Meanwhile, co-Processor 1 handles floating point operations. This configuration leaves two more slots to put co-processors in it.

MIPS co-processors may have a register file in it. There are no strict naming convention for register file for co-processors. Hence this thesis will use the common register naming that is generally used in MIPS reference [4]. MIPS reference [4] address a register in main processor with the conventional name that is shown in table 4.1. When addressing a register in co-processor, it starts with 'CPR' followed by a bracket that consists of co-processor number and the register number. For example addressing register number 9 in co-processor 3 is written :

CPR [3, $9]

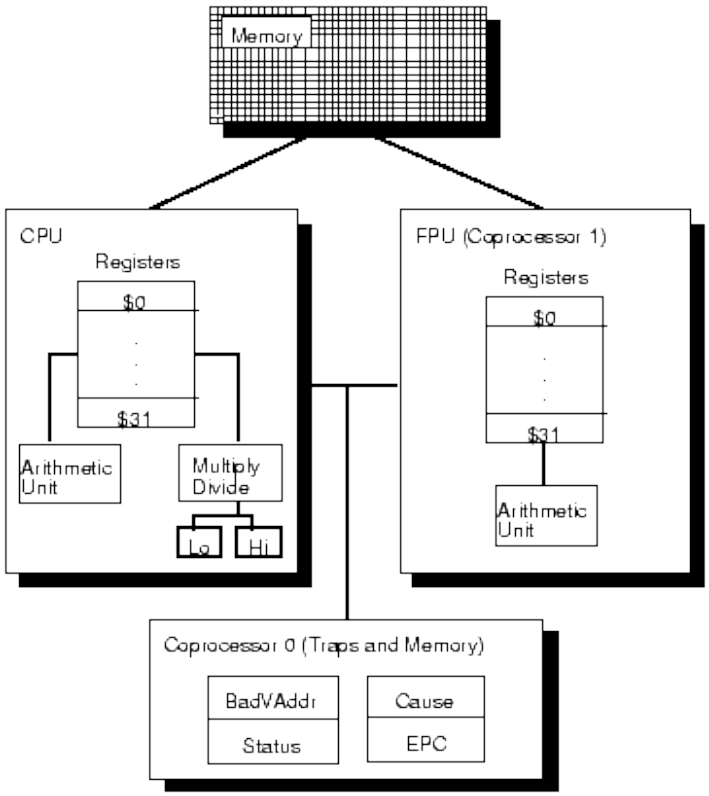



**Figure 4.3:** MIPS R2000 [3]

## 4.2 Extending Single Cycle 32 bits MIPS

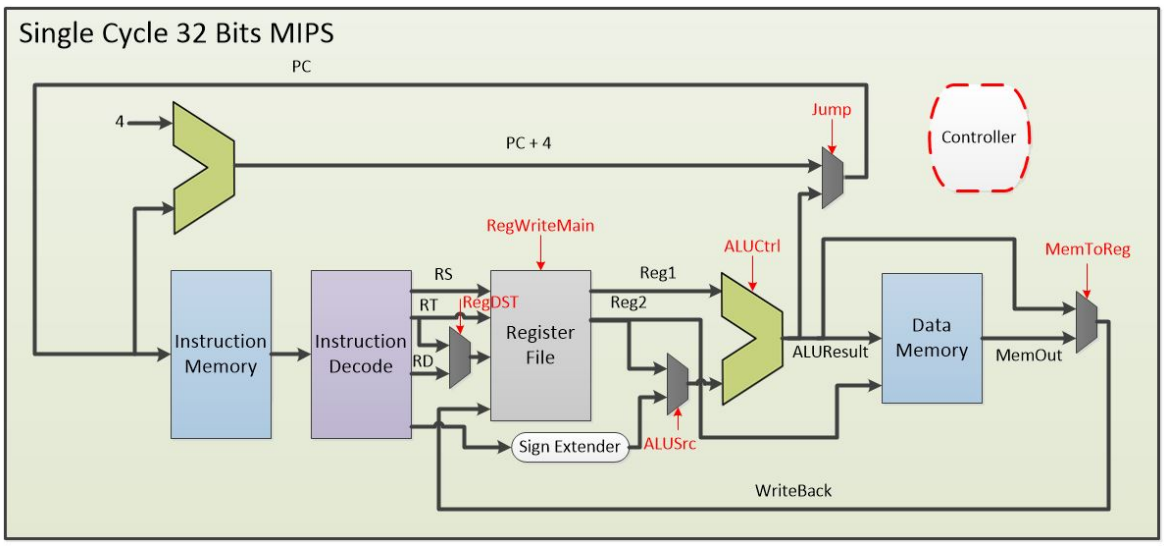



**Figure 4.4:** Single Cycle 32 Bits MIPS [5]

Before integrating retargeting engine as a co-processor, the initial design of single cycle 32 bits MIPS (figure 4.4) needs to be extended. According to section 3.3,

data movements are required from and to co-processor that employ MTC, MFC, SWC and LWC instructions. Also in section 3.3, it is known that $RS$ part is used for $special$, so the data can only be placed in $RT$ and $RD$. Thus the input data to co-processor comes from the output of $Reg2$, because $Reg2$ is connected to $RT$ and $RD$. According to section 3.3, register file will be added into the retargeting engine co-processor, hence it needs a control signal to enable write into register file in the co-processor $RegWriteCOP2$. So when an MTC2 instruction is executed the data will pass the $Reg2$ into the co-processor 2 and $RegWriteCOP2$ signal needs to be active (figure 4.5).

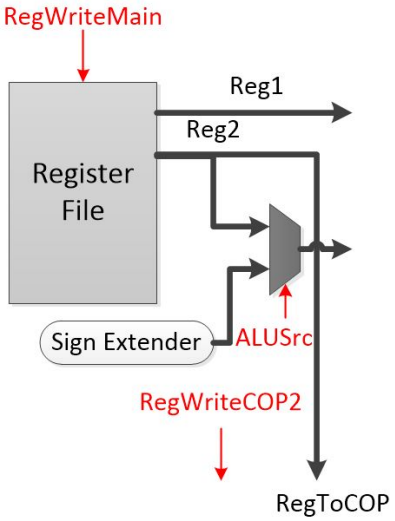



**Figure 4.5:** Extending for MTC

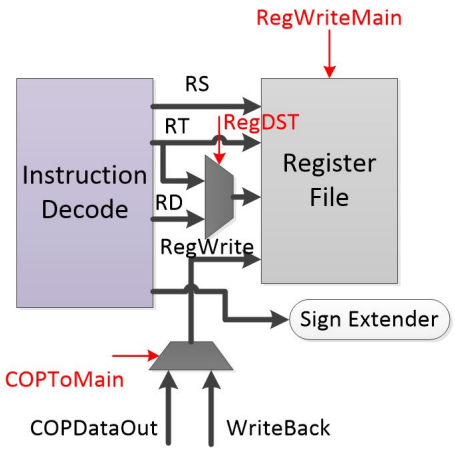



**Figure 4.6:** Extending for MFC

Next when a data is moved from co-processor to main processor, the main processor will place it in the register file. Therefore a multiplexer is required that can

choose between $WriteBack$, the initial path for writing to register file in main processor, and $COPDataOut$, the output of co-processor, with select signal $COPToMain$. So that when an MFC2 instruction is executed the data will pass the $COPDataOut$ into $RegWrite$ that needs active $COPToMain$ select signal (figure 4.6).

Then when a data is stored from co-processor to data memory, it needs to be placed into the input data of the memory. Hence a multiplexer is required that can choose between $Reg2$, the initial path for writing to data memory in main processor, and $COPDataOut$, the output of co-processor, with select signal $COPToMem$. So that when an SWC2 instruction is executed the data will pass the $COPDataOut$ into the input of data memory that needs active $COPToMem$ select signal (figure 4.7).

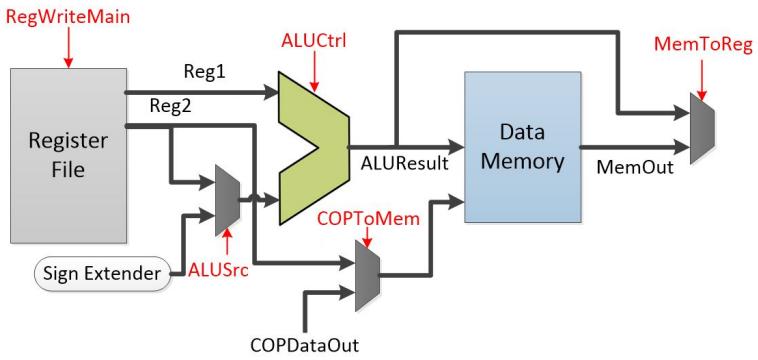



**Figure 4.7:** Extending for SWC

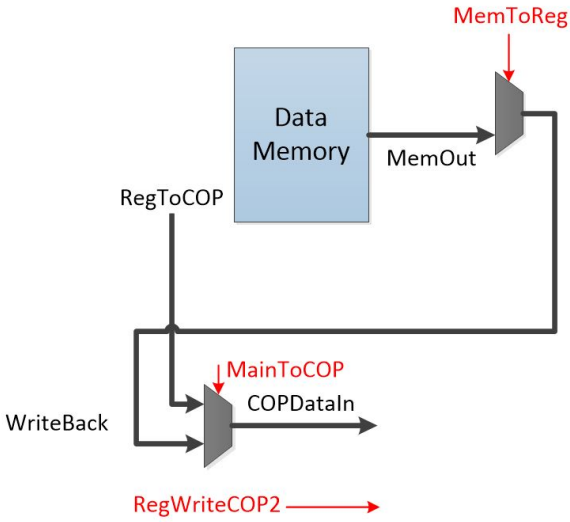



**Figure 4.8:** Extending for LWC

Finally when a data is loaded to co-processor from data memory, it needs to be placed into the output data of the memory. $MemToReg$ select signal needs to be active, so that the data pass to $WriteBack$. However while extending the

single cycle 32 bits MIPS for MTC2, there is an input signal $RegToCOP$ that was added before. So a multiplexer is required that can choose between $RegToCOP$ and $WriteBack$ with select signal $MainToCOP$. So that when an LWC2 instruction is executed the data will pass the $WriteBack$ after $MemToReg$ signal is active and then it will pass to $COPDataIn$ after the $MainToCOP$ signal is active. However this operation needs $RegWriteCOP2$ signal to be activated too so that writing into the co-processor register file is able (figure 4.7).

Lastly, since there are 4 new instructions that is dedicated for the retargeting engine co-processor (iWRITE, iREAD, iAPPLY and iRESET), then $instruction$ signal needs to be connected into the retargeting engine co-processor as well. Figure 4.9 shows the diagram block of an on-chip IJTAG dependability processor hardware.

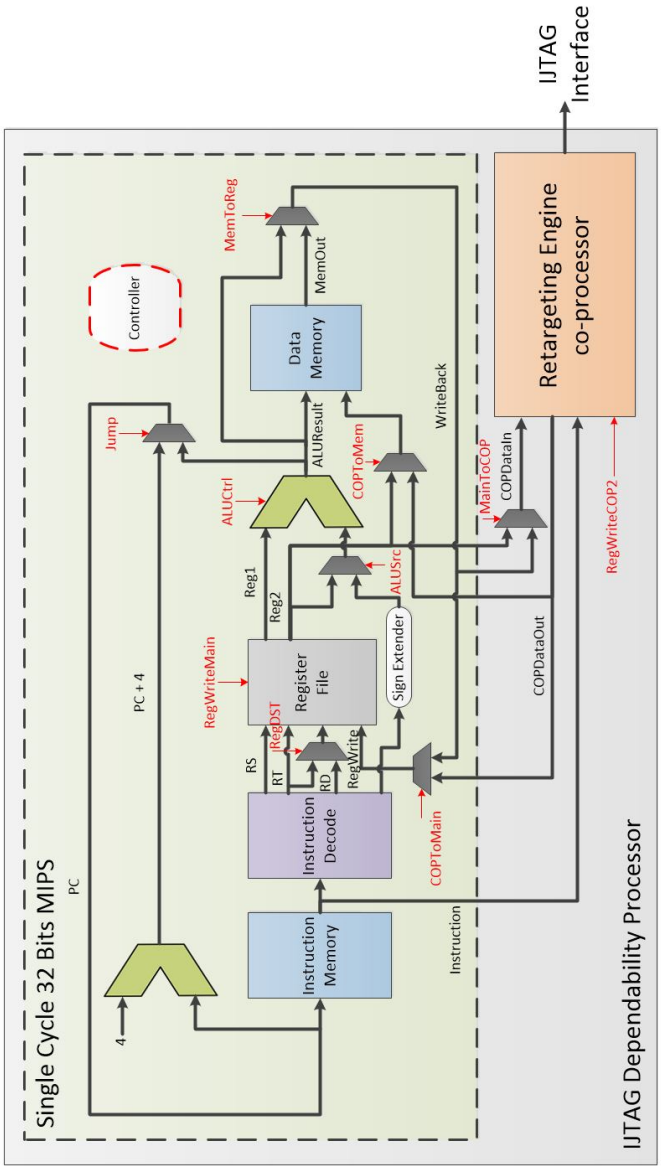



**Figure 4.9:** IJTAG Dependability Processor Block Diagram

## 4.3 Retargeting Engine Wrapper Design

Retargeting is a compulsory operation in order to access embedded instruments on the IJTAG network. This thesis uses an on-chip retargeting engine that has been proposed by [9] for handling the on-chip retargeting. To embed the retargeting engine into a co-processor for single cycle 32 bits MIPS, the design of a wrapper for retargeting engine is required which will be placed in co-processor 2 as explained in section 3.3.

Section 2.2 explained that retargeting engine works by first receiving access requests to one or more instruments. Then it starts to work when the *Concurrent* signal is changed from HIGH to LOW. This condition is important to be considered in order to use the retargeting engine. According to section 3.3, these access requests are instructions. However due to the *Concurrent* signal condition, connecting the retargeting engine directly with the instructions is not possible. For example after the retargeting engine get an access request for writing 0b0010 to $TDR3$, the main processor can not directly provide next access requests (figure 4.10). Because the main processor needs to move the data from main processor to co-processor (explained in section 3.3). On the other hand the retargeting engine has already interpreted it as a start to generate scan vectors due to the change in the concurrent signal (figure 4.10). To solve this problem, it is best to put an instruction buffer in the wrapper. In this manner the retargeting engine wrapper can send the buffered instruction as a group of concurrent instructions into the retargeting engine when iAPPLY instruction is received.

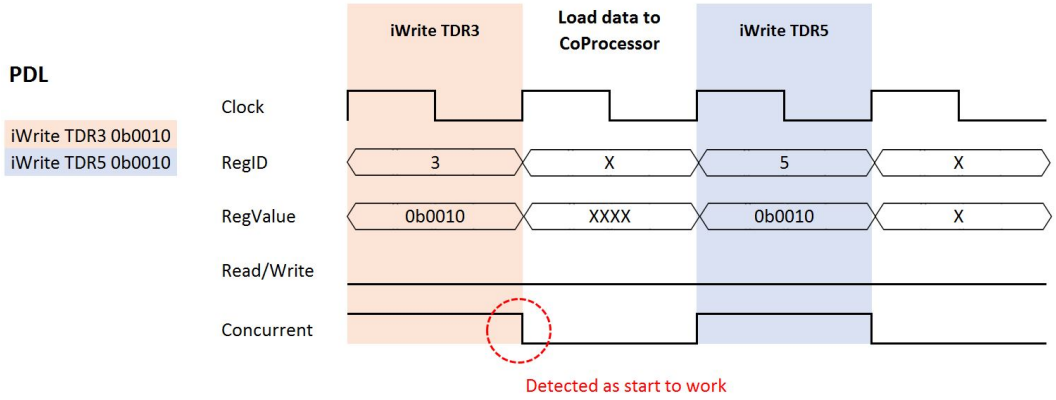



**Figure 4.10:** Concurrency problem on retargeting engine

When retargeting engine has finished accessing embedded instruments, retargeting engine returns *ReadID* and *ReadValue* if there is an access request for reading unordered as explained in section 2.2.3. For example, the order of access requests are reading from $TDR5$ and followed by reading from $TDR3$ (figure 4.11).

Since the connection between instruments on the IJTAG network might be a complex connection, the retargeting engine will generate effective scan vectors for the access requests. However the results may return the data unordered, which in the example is depicted by the return values of $TDR3$ followed by $TDR5$ (figure 4.11).

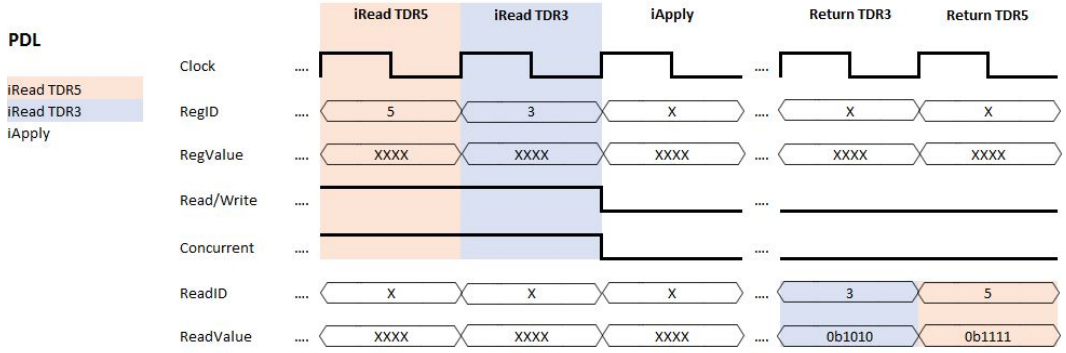



**Figure 4.11:** Retargeting Engine Returns Unordered Data

To solve the unordered returning values, the information in the instruction buffer can be used to arrange these unordered returning values. Moreover in section 3.3 and 4.2, it has been discussed that a register file will be added to the retargeting engine co-processor precisely in the retargeting engine wrapper. Since the returning data will be used in the main processor later, it is better to place the data into the wrapper register file right away. Thus wrapper controller is required to check the returning values according to the instruction buffer and place the returning order in the register file (figure 4.12).

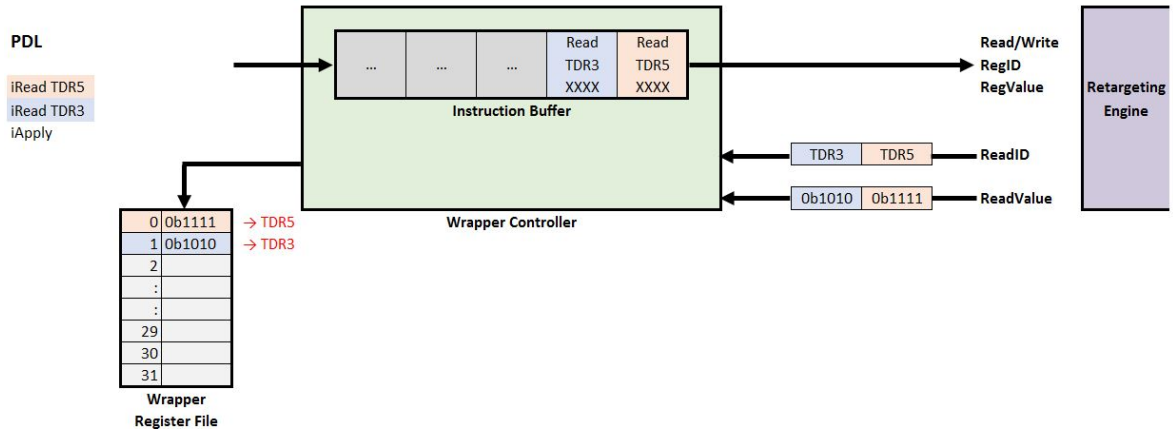



**Figure 4.12:** Arrange The Unordered Returning Values

Summarizing the wrapper design for retargeting engine wrapper, figure 4.13 shows the block diagram of retargeting engine wrapper. It comprises of a register file for holding the data, an instruction buffer to preserve the concurrency, a wrapper

controller for controlling the wrapper and arranging the unordered returning values into the wrapper register file.

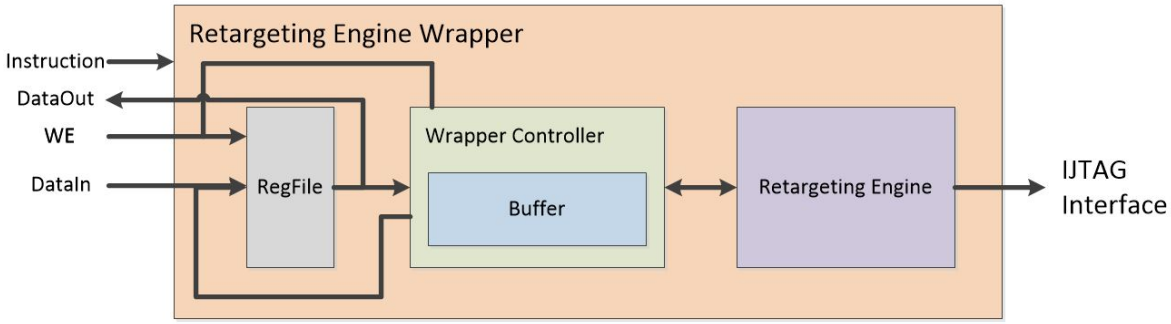



**Figure 4.13:** Retargeting engine wrapper block diagram

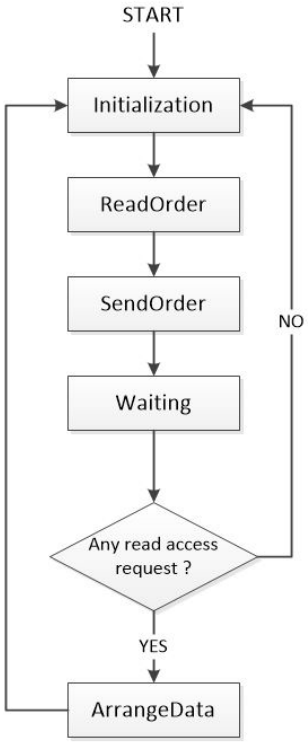



**Figure 4.14:** Retargeting Engine wrapper flow chart

## 4.4 Retargeting Engine Wrapper Workflow

Consider the behavior of the retargeting engine, The works of retargeting engine wrapper comprise of several phase. Figure 4.14 shows the flow chart of retargeting engine wrapper, which are :

1. **Initialization** : reset all the variable and buffer,

2. **ReadOrder** : read an incoming order and put it to buffer,
3. **SendOrder** : send the buffered orders into retargeting engine, if iApply order is detected,
4. **Waiting** : wait the retargeting engine until it finishes accessing embedded instruments,
5. **ArrangeData** : fetch and arrange the incoming data according to the instruction buffer into wrapper register file.

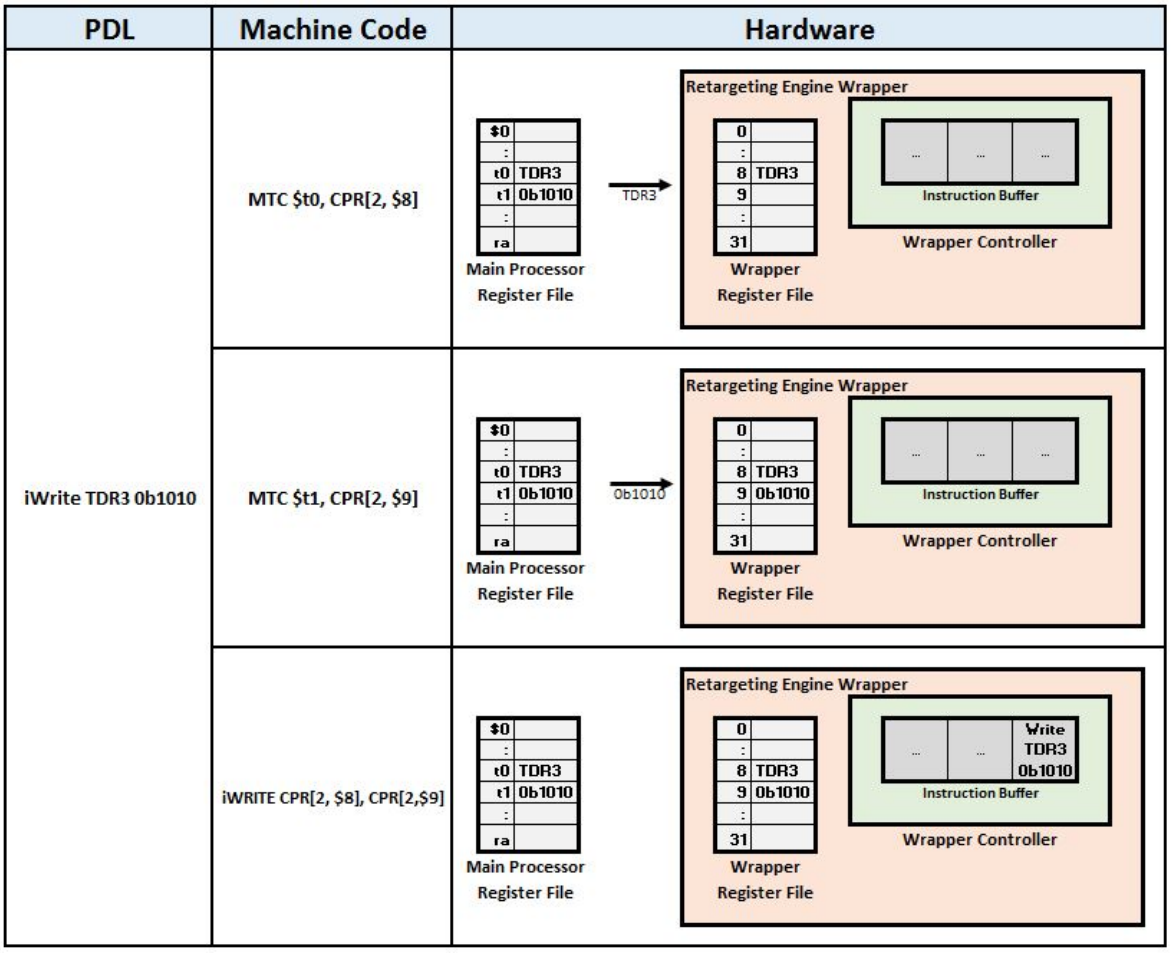



**Figure 4.15:** Reading an iWrite access request

*Initialization* phase resets the instruction buffer and prepares for reading access requests from the incoming instructions in $instruction$ port. While in *ReadOrder* phase, the wrapper is ready to read access requests for specific instructions. For example figure 4.15 shows the steps of an access request for writing 0b0010 into $TDR3$ starts in PDL, machine code and hardware. It starts with the main processor sends the *RegID* followed by *RegValue* and finally followed by an iWRITE instruction. *ReadOrder* phase ends when there is an iAPPLY command, then it changes to *SendOrder* phase. *SendOrder* phase sends the buffered instructions into the re-

targeting engine, in this way access requests can be send concurrently. After all the instructions in the instruction buffer was sent, the wrapper waits until the retargeting engine finishes accessing the embedded instruments in the *Waiting* phase. When the retargeting engine has finished, if there is no access request for reading, the wrapper goes to the *Initialization* phase and sets register 1 in wrapper register file to 1 as an acknowledge signal. However, if there is an access request for reading, the wrapper goes to the *ArrangeData* phase. *ArrangeData* phase reads the incoming data and arranges it according to the instruction buffer into wrapper register file as explained in section 4.3 and sets register 1 in wrapper register file to 1 as an acknowledge signal.

Other than iWRITE instruction, the hardware requirements also requires the retargeting engine co-processor to be able to execute iREAD, iRESET and iAPPLY. Executing iREAD instruction is similar to executing iWRITE instruction but without *RegValue*. On the other hand, executing an iRESET instruction is done by forcing the *RST* port to HIGH in the output of retargeting engine wrapper port. Finally executing an iAPPLY instruction is done by providing a flag for retargeting engine wrapper controller to start sending order into the retargeting engine. It has been explained in section 2.1.2 that accessing embedded instruments take non-deterministic of time. In the hardware side this non-deterministic is handled by *Waiting* phase in wrapper controller, so that the software side also needs to handle this condition too.

## 4.5 Discussion

In this chapter, the hardware design of an on-chip IJTAG dependability processor has been explained. The design based on single cycle 32 bits MIPS that has been described in chapter 3. The hardware design aims to achieve the hardware requirements which are implement retargeting engine co-processor and implement the co-processor type instructions for retargeting engine co-processor. The hardware design starts by extending the single cycle 32 bits MIPS to enable data movements MFC2, MTC2, SWC2 and LWC2 instructions. Then it is followed by enabling the iWRITE, iREAD, iRESET and iAPPLY in the retargeting engine co-processor.

Retargeting engine process a group of concurrent instructions. Thus it is not suitable to connect the retargeting engine into decoded instructions directly, consequently a wrapper design for retargeting engine is required. To make sure the concurrency is preserved, instruction buffer is necessary to queue the instructions before it goes into the retargeting engine. When the retargeting engine finishes accessing embedded instruments, it will return *ReadValue* and *ReadID* unordered as explained in section 2.2.3. Instruction buffer can be used to arrange the unordered returning value that requires a wrapper control. Finally the arranged returning values

can be placed in wrapper register file that has been explained in section 3.3. The hardware of retargeting engine wrapper comprises of a register file, an instruction buffer, a wrapper controller and a retargeting engine.

Until this point, the hardware of an on-chip IJTAG dependability processor has been designed. It extends single cycle 32 bits MIPS by having a retargeting engine wrapper. Looking by how the requirements are answered, there are not much work done for the hardware part. Regardless of the software part, the hardware design should be suffice to execute a dependability application. Appendix A provides the hardware design of an on-chip IJTAG dependability processor and the retargeting engine wrapper.

# Chapter 5

# PDL Cross Compiler

Chapter 3 analyzes the hardware and software co-design for an on-chip IJTAG dependability processor. The results are hardware and software design requirements. This chapter focus on building a PDL cross compiler to achieve the software requirements, which are :
1. Enable to translate PDL syntaxes into MIPS machine code
2. Map PDL commands into co-processor type instructions
3. Use fixed point representation
4. Compensate complex hardware operations by emulating multiplication, division, square root and power operations in the software side.

PDL cross compiler compiles PDL scripting language into a machine code, which in this thesis focus on MIPS machine code only. Building a PDL cross compiler starts with an analysis on how to build a cross compiler for PDL. Then it is followed with a brief explanation about ANTLR tool that is used as compiler framework while building a PDL cross compiler. Afterwards it describes PDL cross compiler design and is closed with a discussion. Additional section is added for explaining how to use the PDL cross compiler. The result of this chapter is software design of a PDL cross compiler.

## 5.1 Analysis on Building PDL Cross Compiler

PDL cross compiler translates a PDL program into MIPS machine code. According to section 3.1, this thesis considers all PDL operations to be PDL level 1 which extends Tcl scripting language. Tcl scripting language is designed to be interpreted than to be compiled. An interpreter directly executes the operations specified in the source program on inputs supplied by the user [1]. Thus the fundamental difference is interpreter does not produce a machine code. On the other hand a compiler takes

the whole source program and generates a machine code that behaves similar to the source program.

Despite Tcl is designed to be interpreted than to be compiled, the needs of a PDL program in MIPS machine code is obvious. Thus this thesis will implement the PDL cross compiler for MIPS. Section 2.3.1 explains the phases of compiler, it comprises of lexical analyzer, syntax analyzer, semantic analyzer, intermediate code generator, code optimizer and code generator. Since a cross compiler is a compiler that targets another platform, this makes the compiler phases are also applied into the cross compiler with different target code in code generator.

There are many ways on implementing a compiler. For example before C programming language is built with C, it was build with assembly code [14]. Why ? because the condition in that time was impossible to use other programming languages. Nowadays there are several programming languages that can be used for implementing a compiler. Using C programming language to build PDL cross compiler can be one of the option, because Tcl interpreter is also built with C [15]. Tcl interpreter is open source, but Tcl interpreter does not generate machine code. It is hard to adopt how Tcl interpreter works and implement an equal C code for PDL cross compiler.

Nowadays there are many compiler frameworks such as ANTLR [16], Beaver [17], YACC [18], etc. It can be used to generate a lexer and a parser from a given grammar file that eases a new language to be designed without designing a lexer and a parser from scratch. From the existing options of compiler frameworks, ANTLR offers good documentation and tutorial. Subsequently this thesis will use ANTLR for compiler framework to build PDL cross compiler.

## 5.2 ANother Tool for Language Recognition

ANother Tool for Language Recognition (ANTLR) is a parser generator. It was developed by Terrence Parr in 1989. ANTLR tool can be used to generate lexers and parsers based on the given grammar file. An ANTLR grammar file conceives structures of a programming language. It is expressed using Extended Backus Naur-Form (EBNF). A production rule for complex tokens can be formed by fundamental tokens, likewise a production rule for complex syntaxes can be formed from less complex production rules. In this way EBNF can be used to define complex programming language grammar with scalable structure.

EBNF supports logical operations for its tokens that is represented with symbols. For example '|' symbol denotes alteration, '?' symbol denotes optional (can be none), '+' symbol denotes 1 or more, '*' symbol denotes 0 or more, etc [19]. For example (listing 5.1), defining a $NUMBER$ with EBNF starts with defining a $DIGIT$

production rule that consists of a single number that ranges from 0 to 9. Then it is followed with defining a $NUMBER$ production rule that consists of at least a $DIGIT$. When EBNF is used for defining a $SCALAR\_ID$, it starts with defining $ALPHABET$ production rule that consists of a single alphabet that ranges from lowercase a-z to uppercase A-Z. Then it is followed with defining a $SCALAR\_ID$ production rule that starts with $ALPHABET$ token and it can be followed with $ALPHABET$, $DIGIT$ or dash ('_') tokens from none to infinity.

**Listing 5.1:** Example of an EBNF grammar file

```
ALPHABET
: [a-zA-Z]
;

DIGIT
: [0-9]
;

NUMBER
: (DIGIT)+
;

SCALAR_ID
: ALPHABET (ALPHABET | DIGIT | '_')*
;
```

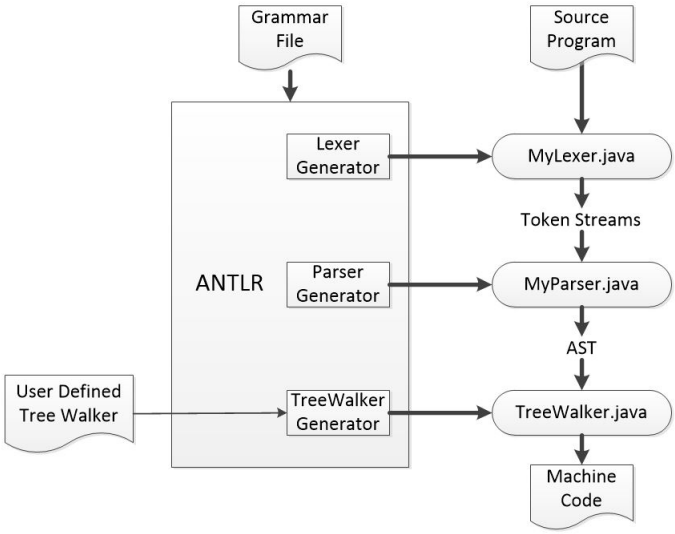



**Figure 5.1:** ANTLR workflow

ANTLR tool works by reading the given grammar file and produces a recognizer (a lexer and a parser) for the given grammar [16]. Since ANTLR is developed in

Java, it generates a lexer and a parser as a java classes (figure 5.1). With the lexer java class, a source program can be extracted into token streams. Then using the parser java class, the extracted token streams are formed into an Abstract Syntax Tree (AST). Moreover ANTLR tool also generates a tree walker to walk the AST that user can define how the AST is processed. In this way user can use the tree walker to generate a machine code for the given source program. This approach is suitable for developing a prototype compiler. There are several compilers have been built using this approach : AADL [20], UDLC [21] and Gaussian Script [22].

## 5.3 PDL Cross Compiler Design

While developing a prototype compiler, intermediate code generator and code optimizer phases can be considered as optional phases. As explained in section 5.2, ANTLR tool is a parser generator that can generate a recognizer based on the given grammar file. The recognizer comprises of a lexer, a parser and a tree parser java classes (figure 5.2). This makes using ANTLR tool is beneficial, especially for developing a prototype compiler. Because it leaves only the code generator phase to be determined in TreeWalker.

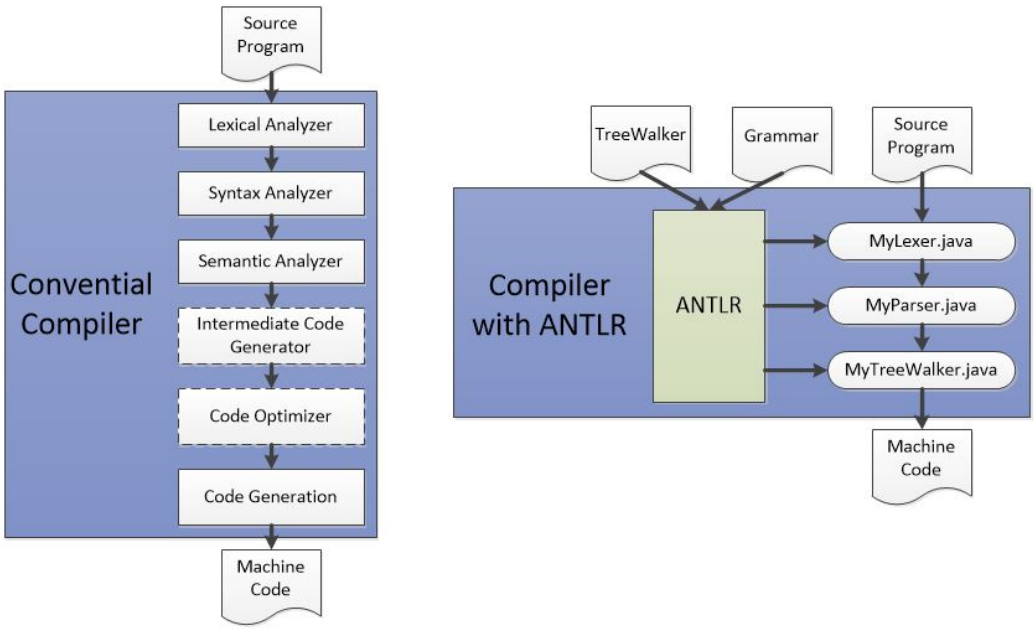



**Figure 5.2:** Comparison of conventional compiler and compiler with ANTLR tool

Figure 5.3 depicts a workflow of PDL cross compiler using ANTLR tool. Since the retargeting engine is used in the hardware design, the PDL cross compiler also needs to process the H-Array. This makes the PDL cross compiler requires an H-

Array, a PDL grammar and a PDL tree walker (figure 5.3). H-Array file is a plain text file that contains a representation of IJTAG network. There is no specific format for H-Array file only separate every instruments (registers) and logical connections with new line as in figure 2.6. For PDL grammar and PDL tree walker, it will be explained on the following sections.

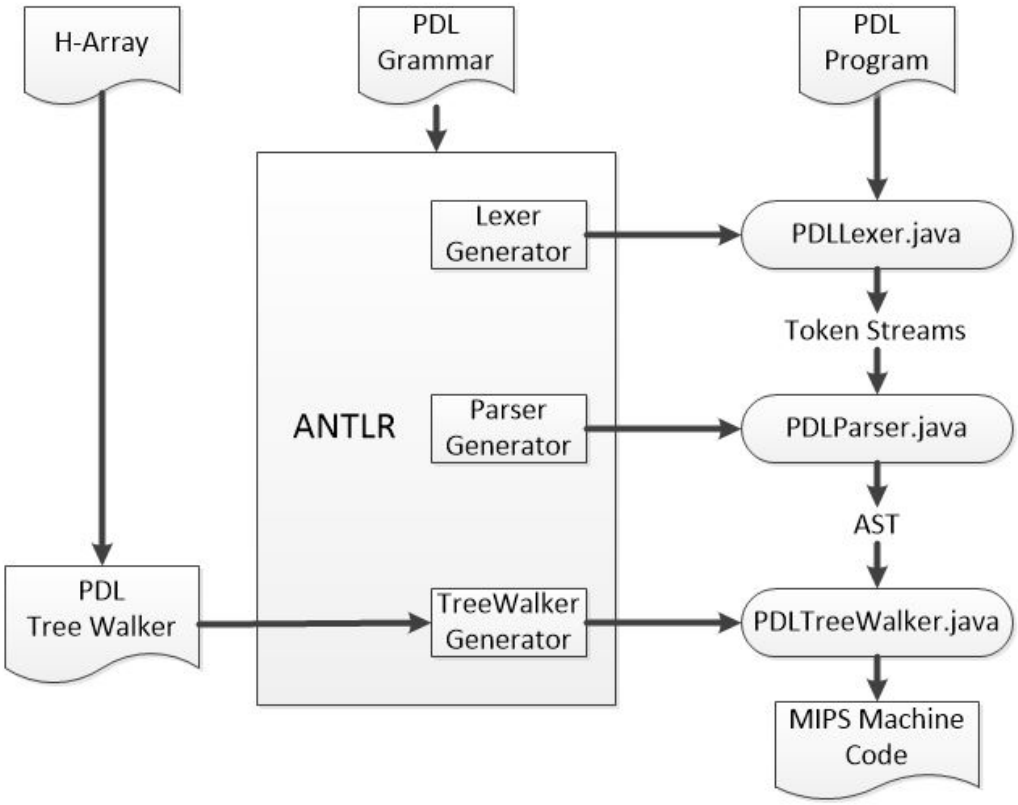



**Figure 5.3:** PDL cross compiler workflow

## 5.4 PDL Grammar

Using PDL level 1 that extends of Tcl scripting language, PDL also inherits the Tcl grammar. This makes PDL grammar consists of a PDL exclusive commands grammar and a Tcl grammar. IEEE 1687 standard provides the grammar for PDL exclusive commands [7]. However this thesis changes several things from PDL grammar provided in the standard. First, section 2.1.2 explains that PDL file begins with defining PDL level within iPDLLevel command. Then it is followed with iProcsForModule command that aims to associate the module in ICL with which subsequent iProcs. Hence re-using the retargeting engine [9] as a co-processor makes iProcsForModule and iProcs are no longer relevant due to H-Array representation.

Second, statements and procedure definitions are placed in the body of main program after iPDLLevel. Most of the programming languages place statements or procedure definitions in the body of main program. But PDL, which based in $module$, places statements and procedures inside an iProc command that is specified for a $module$. For example listing 5.2 shows how TopLevel $module$ calls other $module$s to perform its procedure which is defined within iProc command.

**Listing 5.2:** Example of a PDL script

```
iPDLLevel 1 -version STD_1687_2014;
iProcsForModule TopLevel
iProc Init{}
{
    iCall BISTEngine.Generate
    iCall LogicAnalyzer.Test
}

iProcsForModule BISTEngine
iProc Generate{}
{
    // ...
}

iProcsForModule LogicAnalyzer
iProc Test{}
{
    // ...
}
```

Without iProcsForModule and iProcs, PDL statements and procedure definitions are placed in the body of PDL main program. Listing 5.3 shows a part of PDL grammar that has been changed for this thesis. The *mainProgram* token is the root of the PDL grammar, other tokens will be placed under the *mainProgram* token in the AST. The *mainProgram* expects *c_IPDLLevel* token which represents an IPDLLevel command. Then it expects a statement that is represented by a *statement* token or a procedure definition that is represented by a *c_procDef* token. *statementList* and *procedureList* are used as token collectors for *statement* and *c_procDef* respectively. In this manner the AST can organize which statement or procedure definition that will be visited (walked) first.

Third, this thesis considers all PDL operations to be PDL level 1 as explained before in section 3.1. As depicted in listing 5.3, the *c_iPDLLevel* token comprises of comprises of 'iPDLLevel' token followed with a NUMBER, '-version' and 'STD_1687_2014' tokens. The NUMBER token in *c_iPDLLevel* token is used to represent the PDL level

as in the standard [7]. Although all PDL commands are treated as PDL level 1, this command is necessary to be included for the sake of PDL compliance.

**Listing 5.3:** Main Program of PDL in EBNF

```
grammar PDL;

mainProgram : c_iPDLLevel (statementList+=statement | procedureList+=
    c_procDef)*;

c_iPDLLevel
: 'iPDLLevel' number=NUMBER '-version' 'STD_1687_2014'
;

c_procDef
: 'proc' procName=SCALAR_ID '{' (argumentList+=varDeclaration)* '}' '{' (
    statementList+=statement)* '}'
;

varDeclaration
: varName=SCALAR_ID
;

SCALAR_ID
: ALPHABET ('_' | ALPHABET | DIGIT )*
;

ALPHABET
: [a-zA-Z]
;

NUMBER
: (DIGIT)+
;

DIGIT
: [0-9]
;
```

According to section 3.1, there are 7 PDL commands that will be implemented in software side. These 8 PDL exclusive commands comprise of iPDLLevel, iReset, iWrite, iRead, iApply, iRunLoop and iGetReadData. Listing 5.4 shows PDL grammar for those 7 PDL commands that follows the grammar that is provided in [7].

**Listing 5.4:** Grammar for PDL commands in EBNF [7]

```
c_iPDLLevel
: 'iPDLLevel' num=allNumber '-version' 'STD_1687_2014' #IPDLLevel
```

```
;

c_iWrite
: 'iWrite' hArray=SCALAR_ID num=allNumber
;

c_iRead
: 'iRead' hArray=SCALAR_ID (num=allNumber)?
;

c_iGetReadData
: 'iGetReadData' hArray=SCALAR_ID
;

c_iApply
: 'iApply'
;

c_iReset
: 'iReset'
;

c_iRunLoop
: 'iRunLoop' num=allNumber '-tck'    #iRunLoopTck
| 'iRunLoop' num=allNumber '-sck'    #iRunLoopSck
| 'iRunLoop' '-time' num=allNumber   #iRunLoopTime
;
```

There are no grammar change for these 7 PDL exclusive commands. However for iWrite, iRead and iGetReadData commands, they must refer to a TDR on H-Array instead of ICL (figure 5.4).

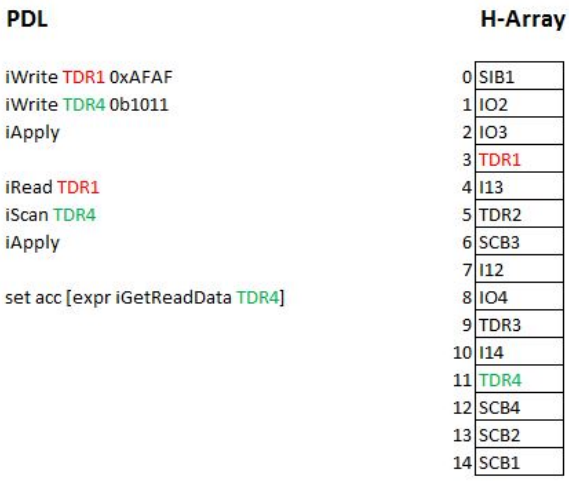



**Figure 5.4:** Referring H-Array in PDL

When it comes grammar for Tcl, experts always come up with fruitless debate [23]. Because Tcl does not understand reserved keywords, every Tcl commands can be redefined and tailored as the user wants (even if, while and for) [15]. This Tcl behaviours are not suitable to be implemented into EBNF form (ANTLR grammar file). That is why there is no grammar for Tcl available. However the needs of PDL cross compiler is obvious, so this thesis build Tcl grammar by reverse engineering the Tcl language specifications in Tcl book [15] that was written by John K. Ousterhout, founder of Tcl.

Every built in Tcl commands have been described on [15]. Hence following the description and the behaviour of each Tcl command, it is possible to produce the Tcl grammar. However this thesis does not cover all of Tcl built in commands, only the fundamental commands that is required to support PDL. The loops are only for and while commands and the conditionals are only if, else if and else without switch command. This thesis also does not implement string operations, since the on-chip IJTAG dependability processor is an embedded processor that does not interact with the user. Summarizing the PDL grammar, it has limitations that comprises of :
**1.** It only implements iPDLLevel, iReset, iRunLoop, iGetReadData, iApply, iWrite and iRead.
**2.** It places statements and procedure definitions in the body of main program without iProc and iProcsForModule.
**3.** It handles all PDL operations in PDL level 1.
**4.** It can not redefine reserved keywords (set, if, else, while, for, proc, incr, expr, pow, sqrt, etc.)
**5.** The loops are only for and while commands.
**6.** The conditionals are only if, else if and else, without switch included.
**7.** No string operations.

## 5.5 PDL Tree Walker

The design of PDL tree walker is limited to the requirements of reading the AST, generated MIPS machine code, hardware limitations and violations of grammar. This thesis divides the PDL tree walker into five java classes which consist of :
**1. Settings.java** : consists of settings and global variable initialization.
**2. MyVisitor.java** : is assigned for walking the AST and instantiates registerHandler, commentHandler and errorHandler.
**3. RegisterHandler.java** : generates MIPS machine code
**4. CommentHandler.java** : generates assembly code with comment for debugging purpose
**5. ErrorHandler.java** : handles violation of grammar and produce an error.

The works in PDL cross compiler focus on Settings, MyVisitor and RegisterHandler java classes. Therefore these three will be discussed in the following subsections. The works in CommentHandler and ErrorHandler java classes are very limited and for a debugging purpose only.

### 5.5.1 Settings.java

Settings java class initializes global variables and necessary settings for the PDL tree walker. It organizes input and output file, size of data memory, and fixed point representation. However settings java class focuses in handling the fixed point representation.

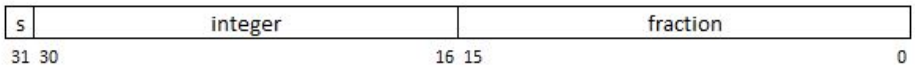



**Figure 5.5:** Q15.16 Fixed Point Representation

As explained in section 3.2.1, fixed point representation in the software side is required to compensate the absence of FPU in the hardware side. This thesis implements Q15.16 fixed point formats which categorizes 32 bit numbers into 1 bit of sign, 15 bits of integer and 16 bits of fraction (figure 5.5). The fractional accuracy of Q15.16 is $0.0000154$. And the integer range of Q15.16 from -32768 to 32767.

Due to fixed point representation, there is a slight change on the behaviour of code generation. Generating code for $A \leftarrow 7 + 9$ normally can be done with 2 Add Immediate (ADDI) instructions, which are :

$$\begin{aligned} A \leftarrow 7 \quad &\equiv \quad \text{ADDI } \$t0, \$0, 7 \\ A \leftarrow 7 + 9 \quad &\equiv \quad \text{ADDI } \$t0, \$t0, 9 \end{aligned}$$

However Q15.16 fixed point representation implements those operations differently, which are : Assigning a number is done by assigning the fractional part first because

$$\begin{aligned} A \leftarrow 7 \quad &\equiv \quad \text{ADDI } \$t0, \$0, 0 \\ &\qquad\quad \text{LUI } \$t0, 7 \\ temp \leftarrow 9 \quad &\equiv \quad \text{ADDI } \$t1, \$0, 0 \\ &\qquad\quad \text{LUI } \$t1, 9 \\ A \leftarrow 7 + 9 \quad &\equiv \quad \text{ADD } \$t0, \$t0, \$t1 \end{aligned}$$

the ADDI instruction is only available to assign 16 bits of LSB. Then for assigning the integer part is done by Load Upper Immediate (LUI) instruction.

### 5.5.2 MyVisitor.java

MyVisitor java class walks the AST and instantiates RegisterHandler, CommentHandler and ErrorHandler java classes. MyVisitor java class handles procedure handling and executing expression that will be explained in this section.

#### Procedure Handling

The purpose of procedure handling is to make sure the procedure calls mapped into correct procedures. The AST of *mainProgram* contains of iPDLLevel, statements and procedure definitions. After checking the PDL Level, MyVisitor java class visits procedure definitions first then followed by statements afterwards. Figure 5.6 depicts how the AST places the procedure definitions before statements although procedure definition *A* is placed after a statement that instantiates procedure *A* (listing 5.5). In the AST, procedure *A* is placed on the left side of the statements. Which means procedure definition of procedure *A* is visited first than the procedure *A* instantiation in the statement.

**Listing 5.5:** AST example of PDL Procedure Definitions

```
iPDLLevel 1 -version STD_1687_2014

set var0 [expr [A 0 1 ... N]]

proc A{arg0 arg1 ... argN}
{
    \\procedure definition of a
}
```

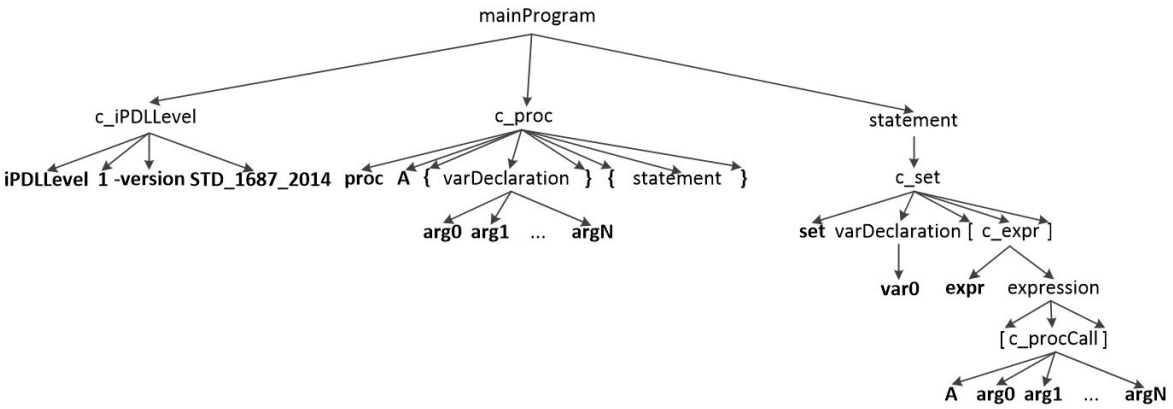



**Figure 5.6:** generated AST from listing 5.5

In the generated MIPS machine code, this approach places procedure *A* in the beginning of instruction list (figure 5.7). To access procedure *A*, the statement needs

to make a jump instruction into an address where procedure *A* begins. In this case procedure calls are always mapped to correct procedures.

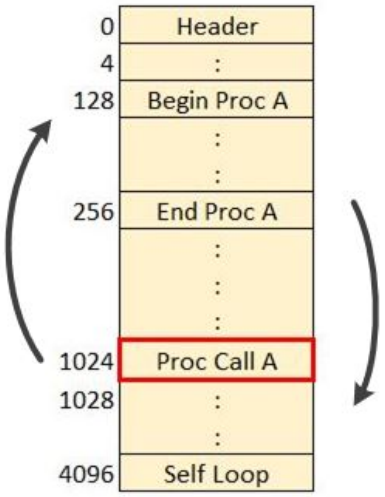



**Figure 5.7:** Example of PDL procedure instantiation

### Executing Expression

AST represents an expression as an operator that has two children on its branches. If there are more than one operators in an expression, one or both children of the main operator may be an operator that also has two children on its branch. These branches can be unlimited depends on the expression. For example $6 * (15 + 7 - 2)$ has an AST that is depicted on figure 5.8.

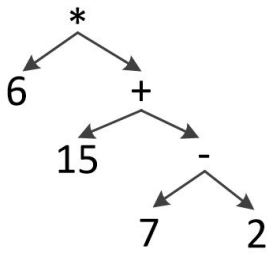



**Figure 5.8:** Example of Expression AST

Designing a compiler to handle expressions need to consider operator precedences. Hence One of the advantage of using ANTLR tool is ANTLR tool supports priority which can be used to handle operator precedences. This priority is specified in the ANTLR grammar file. Listing 5.6 shows the PDL grammar for expression. The top priority is parenthesis which in arithmetic and logic also mean priority. Division

and multiplication are placed higher that addition and subtraction, because multiplication and division are also prioritized than addition and subtraction in arithmetic. Since the basic elements are variables and numbers, it is placed in the very bottom.

**Listing 5.6:** PDL grammar for expression in EBNF

```
expression
: '(' expression ')'                                          #WithParenthesis
| '[' procedureCall=c_procCall ']'                            #ProcCallExpression
| 'sqrt' '(' right=expression ')'                             #Sqrt
| 'pow' '(' left=expression ',' right=expression ')'          #Pow
| left=expression '/' right=expression                        #Div
| left=expression '*' right=expression                        #Mult
| left=expression '-' right=expression                        #Minus
| left=expression '+' right=expression                        #Plus
| left=expression '<<' right=expression                       #ShiftLeft
| left=expression '>>' right=expression                       #ShiftRight
| left=expression '|' right=expression                        #BitwiseOr
| left=expression '&' right=expression                        #BitwiseAnd
| left=expression '^' right=expression                        #BitwiseXor
| left=expression '<' right=expression                        #LowerThan
| left=expression '<=' right=expression                       #LowerThanEqual
| left=expression '>' right=expression                        #GreaterThan
| left=expression '>=' right=expression                       #GreaterThanEqual
| left=expression '==' right=expression                       #Equal
| left=expression '!=' right=expression                       #NotEqual
| left=expression '&&' right=expression                       #LogicalAnd
| left=expression '||' right=expression                       #LogicalOr
| (tok='-')? '$' var=variables                                #Var
| num=allNumber                                               #Number
;
```

To simplify the view of an expression AST, one can transform it into an expression stack. This expression stack always put the left branches first and followed with the right branches then the operation after that. Generating code for an expression also means to process the expression itself, in this case the expression stack. However a complex expression may have its children as an operator with two children on its branches. Therefore, the compiler must find an operator with two non operator children in the AST and process it first. Because the others can not be processed, before both of its children become two non operator children. This sequence of process can be written as ExecuteExpressionStack algorithm (algorithm 1).

ExecuteExpressionStack algorithm aims to execute an expression stack $ExpStack$ until the length of expression stack $L_{ExpStack}$ equals to $1$. This algorithm finds an operator ($ExpStack(n) = operator$) with two non operator children on the expression stack ($ExpStack(n).left \neq operator$ and $ExpStack(n).right \neq operator$) and process

it by generating an equal MIPS machine code for it. Then it substitutes the operator and its two non operator children from the expression stack with a temporary variable $tempN$ instead. Finally decrease the length of expression stack $L_{ExpStack}$ with 2.

**Algorithm 1** Execute Expression Stack $ExpStack$

1: **procedure** EXECUTEEXPRESSIONSTACK($ExpStack$)
2: $L_{ExpStack} \leftarrow Length(ExpStack)$
3: $N \leftarrow 0$
4: **while** $L_{ExpStack} > 1$ **do**
5: find $n$ such that $ExpStack(n) = operator$ and $ExpStack(n).left \neq operator$ and $ExpStack(n).right \neq operator$
6: $LEFT \leftarrow ExpStack(n).left$
7: $RIGHT \leftarrow ExpStack(n).right$
8: $OP \leftarrow ExpStack(n)$
9: generates code for $LEFT\ OP\ RIGHT$
10: substitute $ExpStack(n)$ with $tempN$
11: $N \leftarrow N + 1$
12: $L_{ExpStack} \leftarrow L_{ExpStack} - 2$
13: **end while**
14: **end procedure**

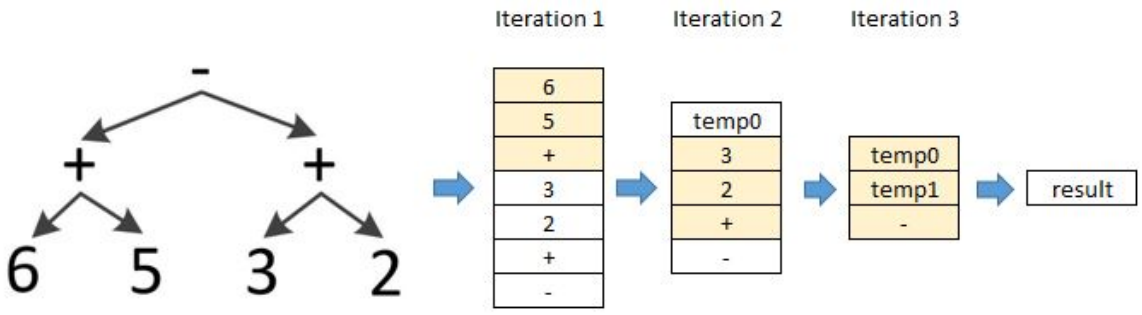



**Figure 5.9:** Example of Expression Stack

An illustration of ExecuteExpressionStack algorithm (algorithm 1) for expression $(6 + 5) - (3 + 2)$ is depicted in figure 5.9. In the iteration 1 there are two operators that have both of its children are non operators. However since process $6 + 5$ is placed more on the top than $3 + 2$, then expression $6 + 5$ get executed first and get substituted with $temp0$. Then in the iteration 2 process $3 + 2$ is executed and get substituted with $temp1$, since it is the only process that has an operator with both of

its children are non operators. In the iteration 3 there is only one operator with both of its children are non operators that is $temp0 - temp1$. Then it leaves only $result$ and the ExecuteExpressionStack algorithm is finished, because the length of expression stack equals to $1$.

### 5.5.3 RegisterHandler.java

RegisterHandler java class is the one that responsible in generating MIPS instructions. The generated MIPS instructions follows the MIPS instruction reference in [4], while generating it follows the way that is explained in [3]. However MIPS architecture is register based. Active variables are stored in the register file. Since register file can only hold limited variables, the rest are stored in the data memory. When a variable is still in the data memory or allocating a new variable that both are going to be used, it needs to be placed in the register file. If there are no empty space in the register file, there is a data in the register file that needs to be moved into the data memory. This limitation requires a register scheduling to organize variables within limited registers.

**Algorithm 2** Regulates turns to use temporary register $TempReg$

```
1: procedure REGISTERSCHEDULING(var,TempReg, n, MEM)
2:     if var ∈ TempReg then
3:         find m such that TempReg(m).id == var.id
4:         TempReg(m) ← var
5:     else if TempReg isfull then
6:         move TempReg(n) to MEM
7:         TempReg(n) ← var.value
8:         n ← n + 1
9:     else
10:        TempReg(n) ← var
11:    end if
12:    if n == 10 then
13:        n ← 0
14:    end if
15: end procedure
```

RegisterScheduling algorithm (Algorithm 2)uses round robin concept where the oldest variable in the register file will be moved into the data memory in order another variable is able to be placed in the register file. RegisterScheduling has a round robin counter $n$ that is initially $0$. When a variable $var$ is assigned to $TempReg$ that has $var$

inside, $var$ is directly assigned into $TempReg(m)$ where $TempReg(m).id$ equals to $var.id$ without any change in round robin counter $n$. When a variable $var$ is assigned to $TempReg$ which has empty slot, $var$ is directly assigned into $TempReg(n)$ (figure 5.10). However if the $TempReg$ is fully occupied with $var \notin TempReg$, $TempReg(n)$ is moved into the memory $MEM$ and $TempReg(n)$ is assigned with $var$. Since index $n$ now holds the newest member in $TempReg$, $n$ is increased by 1. On the other case if the $TempReg$ is not fully occupied and $var \notin TempReg$, $var$ is directly assigned to $TempReg(n)$ and is followed by increasing counter $n$ by 1. Afterwards since the maximum number of available $TempReg$ is 10, $n$ is assigned to $0$ when $n$ reaches 10 (figure 5.11).

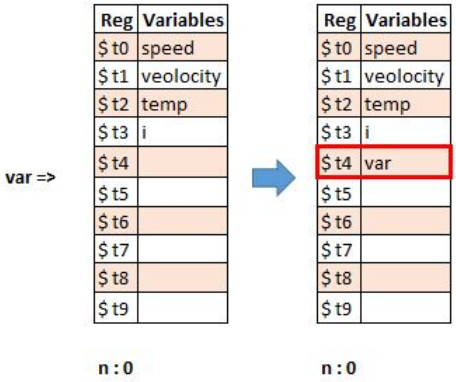



**Figure 5.10:** Example of Assigning variable to registers with empty spot

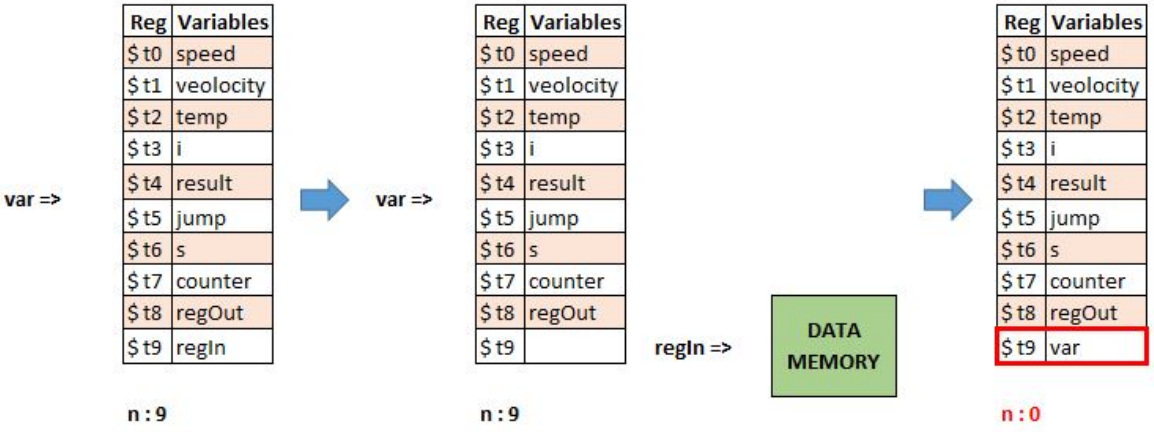



**Figure 5.11:** Example of Assigning variable to fully occupied registers

## 5.6 Map PDL Commands to MIPS Machine Code

Mapping PDL commands into MIPS machine code is done within RegisterHandler java class. As explained in section 3.3, the MIPS machine code includes extended instructions for retargeting engine co-processor. Implementation of common MIPS machine code follows [4] including implementation of MFC, MTC, SWC and LWC. It leaves iWrite, iRead, iReset, iRunLoop, iGetReadData, iApply and iPDLLevel commands. Implementation of iPDLLevel command is not explained since the PDL cross compiler treats all PDL commands into PDL level 1.

Instantiating iWrite command requires two data. On the other hand instantiating iRead command requires only one data. In this thesis, the data is loaded in the main processor and is required in the co-processor. Hence it is necessary to move the data from main processor into co-processor. For example a generated MIPS assembly code from a PDL code that instantiates iWrite command to a temperature sensor (tempSensor0) is shown below :

| **PDL** | | **MIPS Assembly** |
|---|---|---|
| iWrite tempSensor0 0x101 | $\equiv$ | ADDI $\$t0$, $\$0$, $257$ |
| | | MTC2 $\$t0$, CPR[2, $\$8$] |
| | | ADDI $\$t1$, $\$0$, $3$ |
| | | MTC2 $\$t1$, CPR[2, $\$9$] |
| | | iWRITE CPR[2,$\$9$], CPR[2,$\$8$] |

The temperature sensor tempSensor0 has *RegID* that equals to 3 in the H-Array and the *RegValue* 0x101 equals to 257. While PDL instantiates iWrite command, the generated code starts with loading the *RegValue* (257) into the register using ADDI instruction. Then it is followed with moving the value into the retargeting engine co-processor using MTC instruction. Subsequently those steps are repeated once more for *RegID* (3). Finally iWRITE instruction is instantiated with specific registers for *RegID* and *RegValue*. In terms of iRead command, it only needs a *RegID* value to be moved into the co-processor. Then the iREAD instruction is instantiated with only *RegID*.

| **PDL** | | **MIPS Assembly** |
|---|---|---|
| iRead tempSensor0 | $\equiv$ | ADDI $\$t0$, $\$0$, $3$ |
| | | MTC2 $\$t0$, CPR[2, $\$8$] |
| | | iREAD CPR[2,$\$8$] |

iRunLoop command is used for waiting instruments to finish its process. iRunLoop has three type of commands, time based ('-time'), system clock based ('-sck')

**Algorithm 3** iRunLoop MIPS instruction generation Algorithm

```
1: procedure IRUNLOOP(m)
2:     if m > 5 then
3:         rem ← mmod4
4:         temp[0] ← 2
5:         temp[1] ← 3
6:         temp[2] ← 0
7:         temp[3] ← 1
8:         i ← 1
9:         for i <= temp[rem] do
10:            Generate NOP
11:            i ← i + 1
12:        end for
13:        if rem == 0 then
14:            counter ← m − 4
15:        else if rem == 1 then
16:            counter ← m − 5
17:        else if rem == 2 then
18:            counter ← m − 2
19:        else
20:            counter ← m − 3
21:        end if
22:        Generate ADDI $at, $0, counter
23:        Generate BEQ $0, $at, 12
24:        Generate ADDI $at, $at, −2
25:        Generate ADDI $at, $at, −2
26:        Generate BEQ $0, $0, −16
27:    else
28:        i ← 1
29:        for i <= m do
30:            Generate NOP
31:            i ← i + 1
32:        end for
33:    end if
```

and test clock based('-tck'). Hence this thesis only implements a single clock which is a system clock. Therefore the implementation of system clock and test clock based use the system clock. The implementation of system clock and test clock based iRunLoop command instantiates an NOP (No Operation). If a big number of clock is requested for an iRunLoop command, branches are added to make the waiting state. Algorithm 3 explains how iRunLoop generate MIPS instruction for a number of clock cycle $m$. If $m$ is less than 5, then $m$ number of NOP instructions are generated. However if $m$ is more than 5, first it needs to calculate the remaining $rem$ after $m$ is divided by $4$. Then $temp$ variable is initialized for compensating the remaining $rem$. Next $temp[rem]$ numbers of NOP are generated. Subsequently $counter$ variable is initialized that will be the total number of loops. Finally loops for MIPS instructions are generated with $counter$ total number of loops.

Implementing time based iRunLoop command also uses algorithm 3. However the time given needs to be divided with the system clock cycle periode to produce total number of clock $m$. Then by following the algorithm 3, it will produce the result for time based iRunLoop command.

iReset and iApply commands are independent commands that can be used anytime. iReset command instantiates an iRESET instruction for resetting the network. On the other hand iApply command instantiates an iAPPLY instruction to trigger the retargetting engine co-processor. However an iApply command will be meaningful if it was preceded with iRead, iWrite or iScan commands. After iAPPLY instruction is given the retargeting engine starts to generate scan vectors and accessing embedded instruments. Accessing embedded instruments may take non-deterministic of time as explained in section 2.2.3. Hence the main processor needs to wait until the retargeting engine is finished which indicated by acknowledge signal that has been explained in section 4.4. So the implementation of iApply command instantiates iAPPLY instruction followed by a loop that moves the acknowledge signal into main processor and keep looping until the acknowledge signal is ready (1). Therefore an example of reading from temperature sensor tempSensor0 with iApply is shown below :

| **PDL** | | **MIPS Assembly** |
|---|---|---|
| iRead tempSensor0 | $\equiv$ | ADDI $\$t0$, $\$0$, $3$ |
| | | MTC2 $\$t0$, CPR[2, $\$8$] |
| | | iREAD CPR[2,$\$8$] |
| iApply | $\equiv$ | iAPPLY |
| | | MFC2 $\$at$, CPR[2,$\$1$] |
| | | BEQ $0$,at, -4 |

## 5.7 Software Emulated Fixed Point Operations

Software emulated fixed point operations are done to compensate the lack of hardware. It is parts of the MyVisitor java class. Accessing the software emulated fixed point operations are done like accessing procedure (figure 5.12). When an expression comprises of multiplication, division, square root or power operations, the main program instantiates a jump into the specific address. Then after the operation has finished, it jumps back to the next address on the main program.

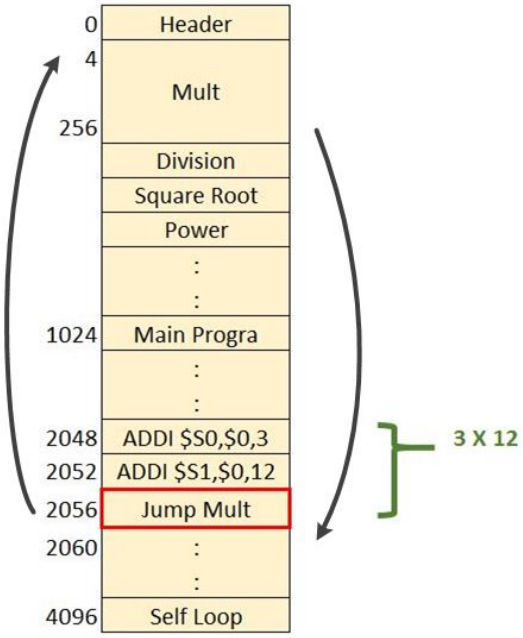



**Figure 5.12:** Accessing software emulated operations

Thus this thesis implements Q15.16 fixed point representation which makes the software emulated operations are also fixed point operations. All the solutions adopt libfixmath.h [24], C library for 16 bits fixed point. Then it was configured for Q15.16 representation. The explanations of software emulated fixed point is available in appendix B

## 5.8 How To Use PDL Cross Compiler

This section explains how to use PDL cross compiler. This tutorial uses Eclipse IDE version Neon Milestone 3 that is available on [25]. First make sure that you have ANTLR tool (available on [16]) and Java Developmnet Kit (available on [26]). Then open the PDL cross Compiler Project and make sure there are two java project : ANTLR and PDLCompiler (figure 5.13).

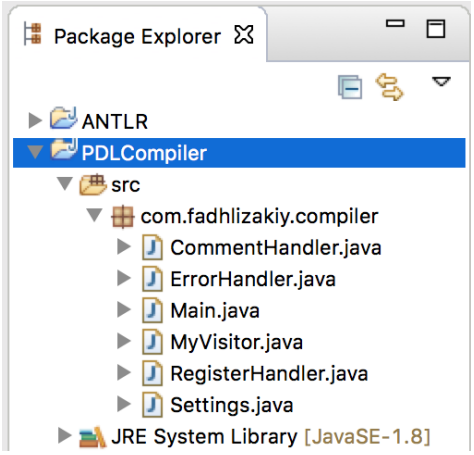



**Figure 5.13:** PDL Cross Compiler package

Open settings.java class. Then sets input and output files. Initially the input files are 'file.pdl' and 'file.harray' and the the output file is 'instruction.data'. 'file.pdl' is where the user can write the PDL code. 'file.harray' is used for providing H-Array from IJTAG network. And 'instruction.data' is the only output file which consists of hexadecimal streams of MIPS machine code. The output file will be used further for simulation. Other than input and output files, there are settings for memory and fixed point representations (figure 5.14). However it is better not to change the fixed point representation, because the PDL cross compiler has not been tested other than Q15.16.

```
1  package com.fadhlizakiy.compiler;
2
3  public class Settings {
4
5      // Memory Variable
6      private int maxMemory = 255;
7
8      // Fixed Point
9      private int wordLength = 32;
10     private int fraction = 16;
11
12     // Input File
13     private String inputHArray = "file.harray";
14     private String inputFile = "file.pdl";
15
16     // Output File
17     private String outputFile = "instruction.data";
```

**Figure 5.14:** PDL cross Compiler settings

Click play, after typing the PDL or set PDL file into settings.java and providing H-Array file. The transcript will show 'done' message if everything goes well. On the other hand it will show error message that needs to be taken care of if there is an error or grammar violations. Finally the output file is generated and ready to use.

## 5.9 Discussion

In this chapter, the design of PDL cross Compiler has been explained. PDL cross compiler uses ANTLR tool which can be used to generate lexer, parser and tree walker from the given grammar and user defined tree walker. This approach is suitable for developing prototype compiler, because it leaves the code generation step only. the design of PDL cross compiler begins with analyzing the needs of PDL cross compiler which are PDL grammar and PDL tree walker. The PDL grammar consists of PDL exclusive grammar and Tcl grammar. The PDL exclusive grammar is available on IEEE 1687 standard [7]. However since there is no Tcl grammar available, it is fulfilled with reverse engineer the Tcl commands which has been described in Tcl book [15]. A PDL tree walker consists of five java classes : Settings.java, MyVisitor.java, RegisterHandler.java, CommentHandler.java and ErrorHandler.java. However this thesis focuses only on Settings.java, MyVisitor.java and RegisterHandler.java. It only uses CommentHandler.java and ErrorHandler.java for testing and debugging.

Settings.java organizes input and output files, size of data memory and fixed point representation. Highlighted works of settings.java is fixed point representation that uses Q15.16 fixed point representation. MyVisitor.java is assigned for walking the AST. While walking the AST MyVisitor.java instantiates RegisterHandler.java, CommentHandler.java and ErrorHandler.java. Highlighed works of MyVisitor.java are procedure handler and expression execution. RegisterHandler.java generates MIPS machine code that applies register scheduling algorithm.

In chapter 3, it has been explained that several operations to be emulated on the software in order to keep the hardware simple such as multiplication, division, square root and power. With Q15.16 fixed point representation makes those operations need to be implemented on fixed point representation as well. This thesis adopted 16 bits fixed points algorithm from libfixmath [24] to implement software emulated fixed point operations that is explained in appendix B.

Until this point, the IJTAG dependability processor and PDL cross compiler has been designed. The user provides an H-Array file and a PDL program and compiles it with PDL cross compiler. Then the PDL cross compiler will produce a MIPS machine code. Next the MIPS machine code is given into the IJTAG dependability processor as instruction memory (figure 5.15).

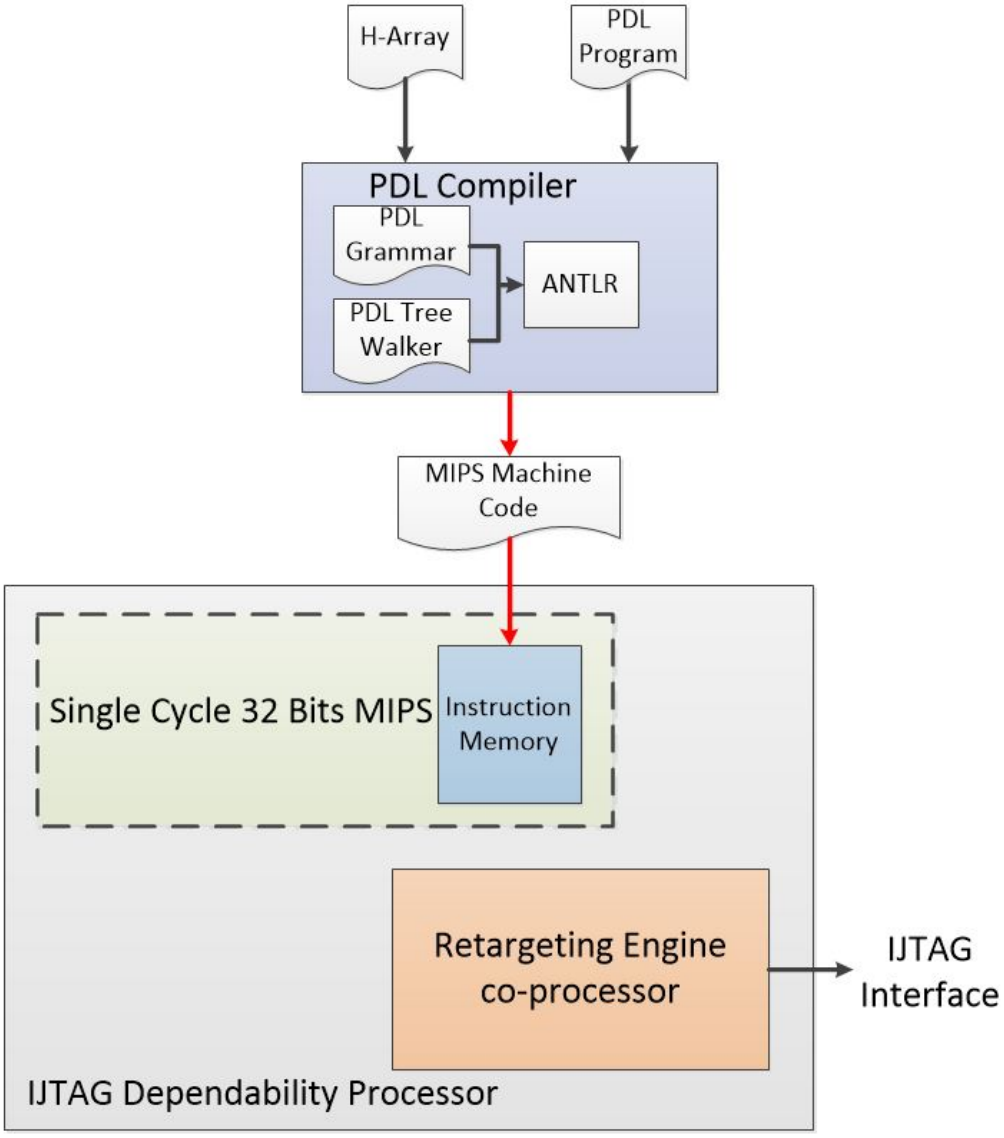



**Figure 5.15:** HW-SW IJTAG Dependability Processor Workflow

# Chapter 6

# Experimental Results

In chapter 4 and chapter 5, the hardware an on-chip IJTAG dependability processor and the software of PDL cross compiler has been discussed. In this chapter both works in software and hardware are combined to execute a dependability application. The works in this chapter is divided into two parts : benchmark test and dependability application test.

## 6.1 Benchmark Test

Benchmark test is an act of running a set of program to asses the performance of an object. In our case, it is a toolchain from compiler into the processor. This thesis uses benchmark test for verifying the correctness of the PDL toolchain (PDL cross compiler to an on-chip IJTAG dependability processor). There are several embedded processor benchmark, however only MiBench that is accessible to academic research. MiBench is an open source embedded processor benchmark that was developed in University of Michigan by [27]. miBench provides C code and the output file for verifying the processor. Hence this thesis uses MiBench for verifying the PDL toolchain. Since the PDL cross compiler implements Q15.16 fixed point representation, the benchmark code is implemented in MATLAB using fixed point representations. Then the MATLAB generated output will be a reference for further comparison.

The workflow for the benchmark test is shown in figure 6.1. It starts with creating equal MATLAB code for the benchmark code followed by generating the MATLAB output file for comparison. Then equal PDL code is provided to PDL cross compiler for generating MIPS machine code. Next this MIPS machine code is loaded into the on-chip IJTAG dependability processor in Modelsim. In this case, 'PRINT' instruction and debug ports ($Ack$ and $Data$) are added into the IJTAG dependability processor for debugging purpose only. When 'PRINT $\$t0$' instruction is executed, the $Ack$ port

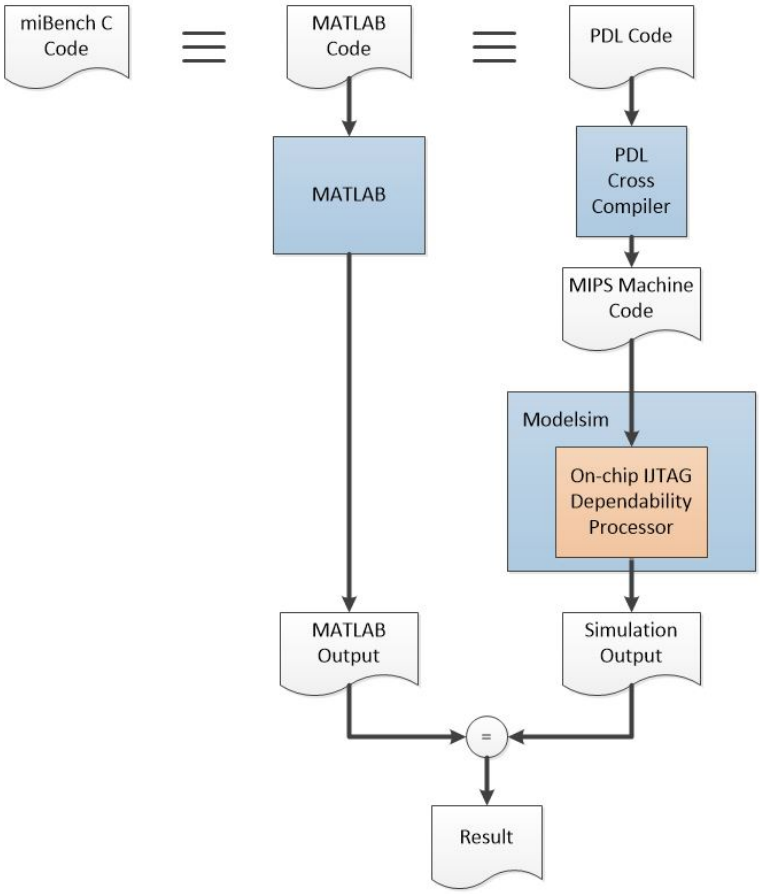



**Figure 6.1:** Benchmark Test Workflow

will be active and data in register $\$t0$ are loaded into $Data$ port. In this case, the data can be collected from the processor into an output file. Finally, output file from processor is compared with the data generated from MATLAB.

### 6.1.1 Benchmark Applications

MiBench provides several applications for benchmarking, however this thesis only uses basic math package from MiBench for benchmarking. It comprises of square root operation, converting degree to radian and converting radian to degree. Here is the MATLAB code and equal PDL code :

**Listing 6.1:** MATLAB code for MiBench Basic Math Benchmarking

```
function fix = toFix(val)
    fix = fi(val,1,32,16);

function rad = fixDeg2rad(deg)
    temp = toFix(3.1416)*toFix(deg)/toFix(180);
    rad = toFix(temp);
```

```
function deg = fixRad2deg(rad)
    temp = toFix(180)*toFix(rad)/toFix(3.1416);
    deg = toFix(temp);

fileID = fopen('exp.txt','w');

%% SQRT test
for I = 0:0.01:10
    fixI = toFix(I);
    temp = toFix(sqrt(fixI));
    Res = bin2dec(temp.bin);

    %%%%%%%%%% PRINT TO FILE %%%%%%%%%%
    fprintf(fileID,'%d\n',Res);
end

%% Angle Conversion : Deg to Rad

for I = 0:1:360
    temp = fixDeg2rad(I);
    Res = bin2dec(temp.bin);

    %%%%%%%%%% PRINT TO FILE %%%%%%%%%%
    fprintf(fileID,'%d\n',Res);
end

%% Angle Conversion : Rad to Deg
I = toFix(0);
while I <= 6.2832
    temp = fixRad2deg(I);
    Res = bin2dec(temp.bin);

    %%%%%%%%%% PRINT TO FILE  %%%%%%%%%%
    fprintf(fileID,'%d\n',Res);
end
fclose(fileID);
```

**Listing 6.2:** PDL code for MiBench Basic Math Benchmarking

```
iPDLLevel 1 -version STD_1687_2014

for{set i 0}{$i <= 10}{incr i 0.01}
{
        print [expr sqrt($i)]
}

for{set i 0}{$i <= 360}{incr i}
```

```
{
        print [expr 3.1416*$i/180]
}

for{set i 0}{$i <= 6.2832}{incr i 0.0175}
{
        print [expr 180*$i/3.1416]
}
```

### 6.1.2 Benchmark Simulations

Error detection is accomplished by subtracting the data from MATLAB and simulator output file. Table 6.1 shows the error from subtracting both output file for each section. It shows that no error is detected for square root operation and conversion from degree to radian. However there is an error for conversion from radian to degree (figure 6.2). The maximum value of error is $0.0036$ . According to the algorithm for converting radian to degree in listing 6.1, it is a multiplication radian value with 180 and followed by a division with $\pi$. $\pi$ is a an irrational number which is usually approximated into $3.14159$. This makes operation that utilize $\pi$ is susceptible to error. Focusing on the error in conversion from radian to degree on figure 6.2, it can be observed that the error is periodical (figure 6.3). This error is not an accumulated error and ruins the calculation. Therefore it can be concluded as a computational error due to approximation for fixed point representation and $\pi$. Hence the IJTAG dependability does not perform perfectly, it can only be used for basic mathematical operations.

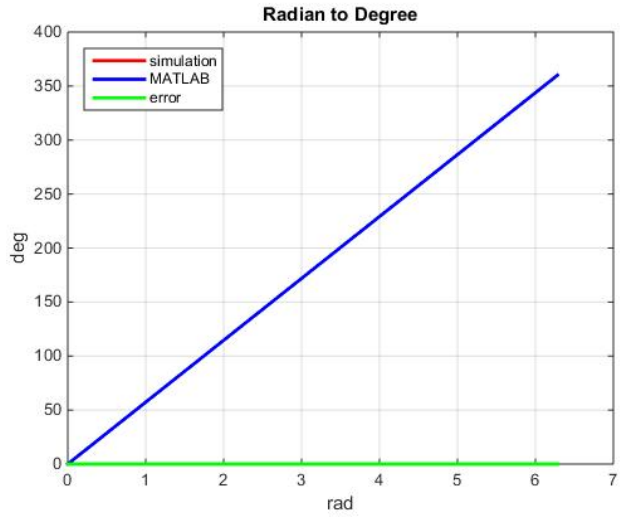



**Figure 6.2:** Result of Conversion from Rad to Degree

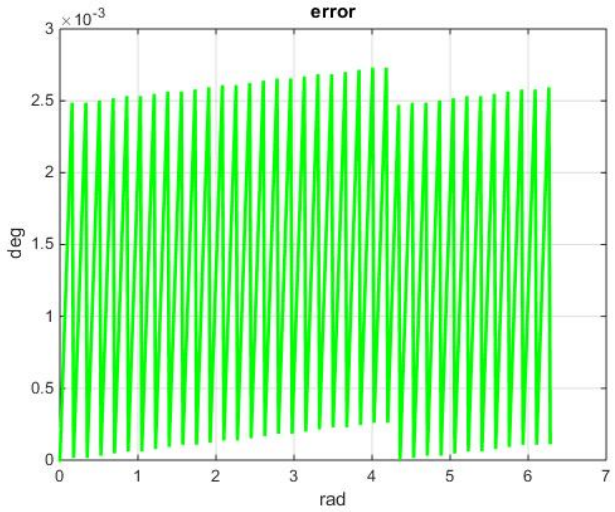



**Figure 6.3:** Error of Conversion from Rad to Degree

**Table 6.1:** MiBench benchmark test error report

| Section | Max Error |
|---|---|
| Square Root | $0$ |
| Convert Deg to Rad | $0$ |
| Convert Rad to Deg | $0.0036$ |

## 6.2 Dependability Application Test

The dependability application, that is used for this test, is acceleration factor calculation from a temperature sensor. This work has been proposed by [28] as a part of lifetime estimation of a circuit. Higher acceleration factor means the circuit is under stressful condition which reduces the lifetime estimation. [28] defines an acceleration factor $AF$ as multiplication between temperature acceleration factor $AF_T$ with voltage acceleration factor $AF_V$ :

$$AF = AF_T \times AF_V$$
$$AF_T = e^{\frac{E_a}{k} \times (\frac{1}{T_S} - \frac{1}{T_O})}$$
$$AF_V = e^{\beta \times (V_S - V_O)}$$

Where $E_a$ denotes activation energy (normally on 0.7 eV), $k$ denotes Boltzman constant ($8.62 \times 10^{-5}$), $T_S$ denotes stress temperature (usually on $393°K$), $T_O$ denotes operating temperature (usually ranges from $333 - 393°K$), $\beta$ is a constant derived experimentally (usually $3.2$), $V_S$ denotes stress voltage (usually $1.1V$) and $V_O$ denotes operating voltage (usually $1V$). Consider the specification in the acceleration factor equation, it leaves $AF_T$ to be calculated since $AF_V$ is a constant.

### 6.2.1 Dependability Application Setup

The experimental setup for executing the dependability application uses Virtex 7 VC707 Field Programable Gate Array (FPGA) evaluation board from Xilinx (figure 6.4). In order to perform the task, the hardware design needs to be synthesized with Xilinx ISE into a bit file. Afterwards the synthesized bit file is downloaded into the evaluation board for testing. For reading the data from FPGA, it uses chipscope analyzer from Xilinx. This chipscope analyzer is connected to $Ack$ and $Data$ debug ports. So that when 'PRINT' instruction is executed, it raises the $Ack$ debug port and the data, that is going to be read, is available in the $Data$ debug port.

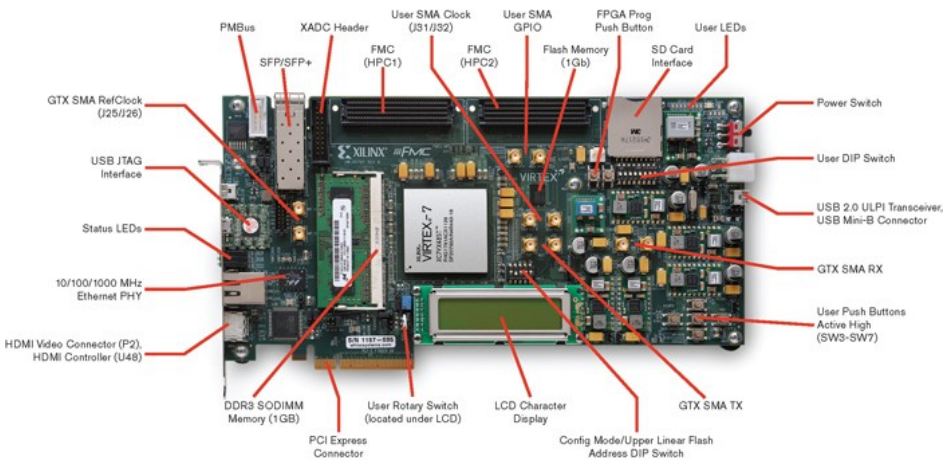



**Figure 6.4:** Virtex 7 VC707 [6]

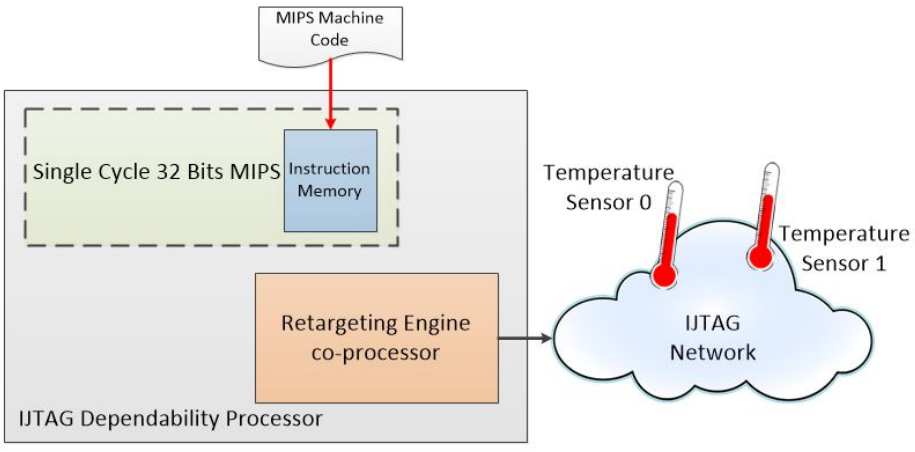



**Figure 6.5:** Abstract of Dependability Application Test IJTAG Network

In this setup, there are two dummy temperature sensors that will be accessed (figure 6.5). Each temperature sensor is connected to 4 switches in the FPGA to give a value into it. The switches give a value multiplied by 8 for each temperature

sensor, so activating the switches with $0x1000$ produces a value of $64°C$. Accessing the dummy temperature sensor is done by first writing 0x101 into it. Afterwards it is available in the next concurrent IJTAG access. The synthesis report shows that the worst-case paths are from the clock into the co-Processor temporary buffer that takes $16.244ns$. However the Virtex 7 Datasheet [6] explains that Virtex-7 system clock is 200MHz. Therefore clockDivider is used to make the system clock of an on-chip IJTAG dependability processor is 50 MHz.

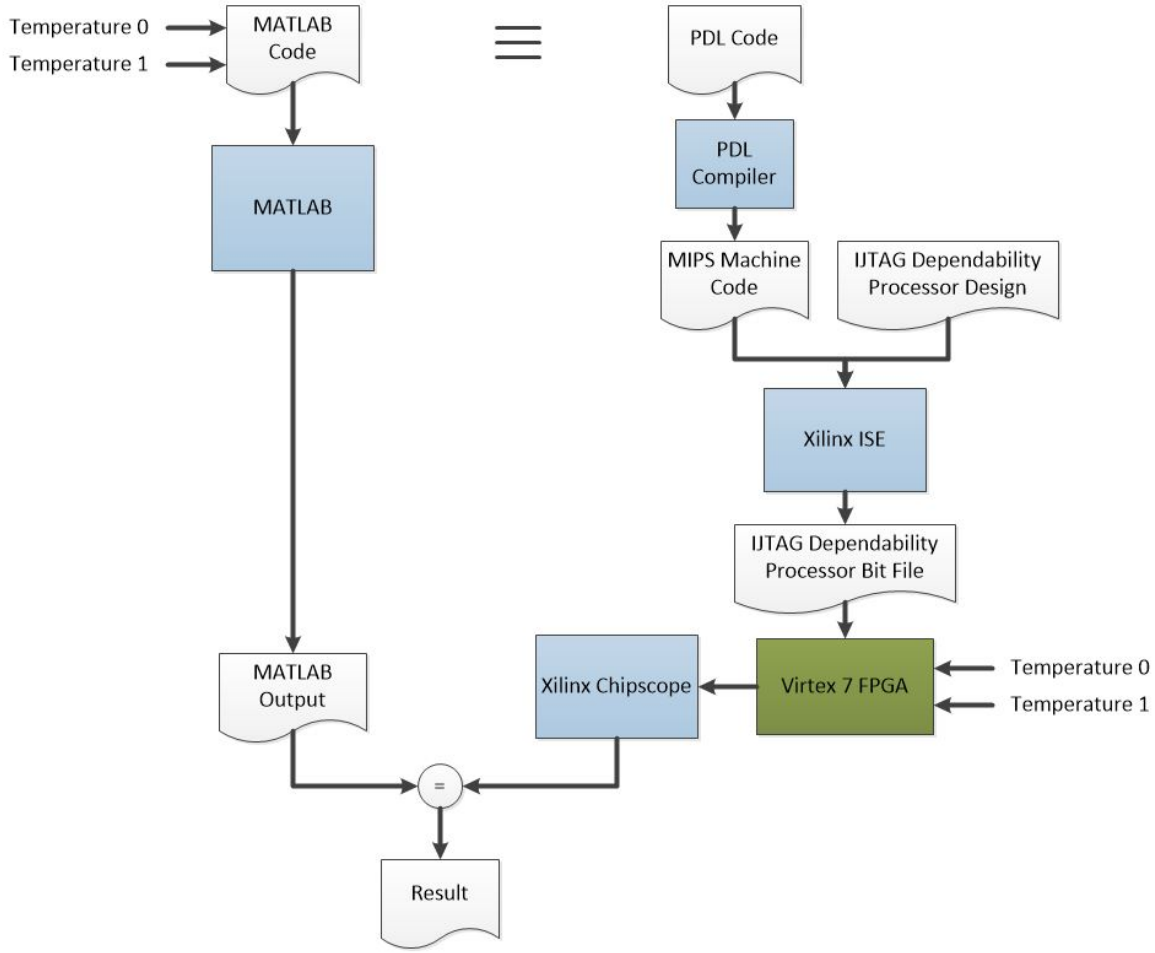



**Figure 6.6:** Dependability Application Test Setup

The overview of the setup is depicted on figure 6.6. First the acceleration factor procedure is calculated on MATLAB for specific temperature values. Afterwards it is simulated on MATLAB and generates output file for reference. Next an equal PDL code is compiled into MIPS machine code. Both IJTAG dependability processor design and MIPS machine code is synthesized with Xilinx ISE and generates IJTAG dependability processor bit file. Then the generated bit file is downloaded to Virtex 7 FPGA and the debug ports $Ack$ and $Data$ can be monitored by Xilinx Chipscope. While the Virtex 7 FPGA holds the IJTAG dependability processor bit file, the switches can be changed to assign values for the temperature sensors. Those

changes can be monitored in the Xilinx chipscope and compared with the MATLAB generated reference.

The PDL for calculating acceleration factor is available in listing 6.4. This code will execute the dependability application once. The calculation of acceleration factor is implemented as procedure. The main code begins with writing 0x101 to the temperature sensors using iWrite commands and is followed with iApply command. Afterwards iRunLoop command is given for waiting the temperature sensors to finish reading the temperature. Next to fetch the temperature values, iRead commands are added followed by iApply command. iGetReadData is used to move the data from co-processor to the main processor for further calculation. After the temperature values are available on the main processor, it instantiates accelerationFactor procedures for calculating the acceleration factor. Finally to make the results available for chipscope analyzer, print commands are used to trigger the chipscope for capturing the data.

**Listing 6.3:** MATLAB code for Acceleration Factor Calculation

```
function result = accFactor(deg)
    to = deg + 273;
    e = toFix(2.7182);
    eaPerK = toFix(81.20649);
    ts = toFix(0.2544);
    AFv = toFix(1.377);

    result = toFix(eaPerK*(ts - toFix(100/to)));
    result = toFix(double(e)^double(result));
    result = toFix(AFv*result);
```

**Listing 6.4:** PDL code for Acceleration Factor Calculation

```
iPDLLevel 1 -version STD_1687_2014

proc accFactor{temp}
{
        set to [expr ($temp << 0x10) + 273]
        set e 2.7182
        set eaPerK 81.20649
        set ts 0.2544
        set AFv 1.377


        set result [expr $eaPerK*($ts-(100/$to))]
        set result [expr pow($e,$result)]
        set result [expr $AFv * $result]

        return $result
```

```
}

iWrite TempSensor0 0x101
iWrite TempSensor1 0x101
iApply

iRead TempSensor0
iRead TempSensor1
iApply

set temp0 [iGetReadData TempSensor0]
set AF0 [expr [accFactor $temp0]]

set temp1 [iGetReadData TempSensor1]
set AF1 [expr [accFactor $temp1]]

print [expr $AF0]
print [expr $AF1]
```

### 6.2.2 Dependability Application FPGA evaluation

Figure 6.7 shows the data captured in chipscope while the temperature sensors are $24°C$ and $32°C$. The MATLAB result for $24°C$ is $0.0171$ that is similar to the FPGA result. It also the same for $32°C$, where both MATLAB and FPGA results show $0.0352$. All possibilities has been tested and it shows an equal result. It shows that IJTAG dependability processor has successfully implemented the dependability application for calculating acceleration factor.

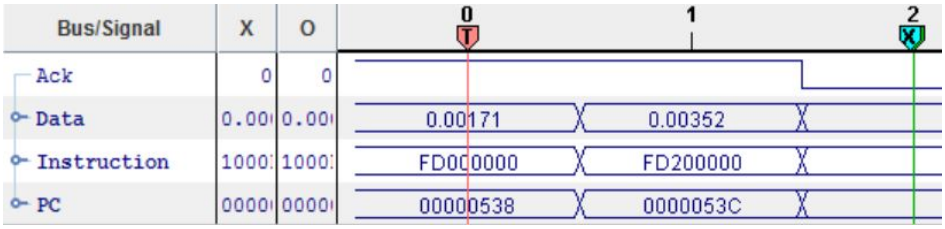



**Figure 6.7:** Chipscope result for $24°C$ and $32°C$

The synthesis report shows that the worst-case paths are from the clockDivider into the co-Processor temporary buffer that takes $16.244ns$. The area utilization for the IJTAG dependability processor is available on table 6.2.

**Table 6.2:** Synthesis Report for Area

| **Properties** | **Area** |
|---|---|
| Slice Registers | 26,450 |
| Slice LUTs | 19,503 |
| Occupied Slices | 7,532 |
| LUT Flip Flop Pairs | 27,101 |

## 6.3 Discussion

This chapter explains two works : benchmark test and dependability application test. Benchmark test is used for verifying the PDL toolchain (from PDL cross compiler into an On-Chip IJTAG dependability processor). It was done by using MiBench open source processor benchmark [29]. From several benchmark that MiBench offers, only basic math packages that are feasible and relevant with the functionality of IJTAG dependability processor and PDL cross compiler. It comprises of calculating square root operations, degree to radian conversion and radian to degree conversion. The result shows that it has no error except radian to degree conversion. The error is caused by division with $\pi$. Due to being an irrational number, $\pi$ is approximated into $3.1416$ in the fixed point representation. This makes operations that utilize $\pi$ are susceptible to errors. However the error is a periodic error (figure 6.3), this kind of error does not get accumulated that ruins the calculations later. Therefore, this error is acceptable.

The dependability application test evaluates the IJTAG dependability processor to execute dependability application while accessing embedded instruments on the IJTAG network. This thesis implements acceleration factor calculation that has been proposed by [28], as a part of lifetime estimation of a circuit. Acceleration factor calculation requires temperature from the temperature sensors, which in this setup uses two temperature sensors. The setup for this test uses Virtex 7 VC707 FPGA from Xilinx. The temperature sensor is connected into the switches, so the user can change the value of the temperature sensor. The test begins with making equal MATLAB and PDL code from acceleration factor equation. The MATLAB code is simulated for reference and the PDL code is compiled into MIPS machine code. Afterwards IJTAG dependability processor design and compiled MIPS machine code is synthesized into a bit file that will be uploaded into the FPGA. Finally reading the data from FPGA is done using Xilinx Chipscope Analyzer that starts capturing the data when $Ack$ debug port is active. The result shows that there is no error for these dependability application tests. The timing constraints shows that the worst-case path is from clockDivider into co-Processor temporary buffer that takes $16.244ns$.

Finally the hardware and software designs for IJTAG dependability processor

successfully executes the dependability application. Although there are some errors and limitations that has been discussed in chapter 4 and chapter 5, it has successfully executed the dependability application for calculating acceleration factor. The next chapter will conclude all the work and provides future work that can be done to improve the on-chip IJTAG dependability processor and PDL cross compiler. The setup environment is available in appendix C

# Chapter 7

# Conclusions & Future Works

## 7.1 Conclusions

In this thesis the design of an on-chip IJTAG dependability processor has been proposed for executing dependability application. The works can be categorized into two parts : hardware and software parts. The hardware part expands the design of single cycle 32 bit MIPS processor [5] with integrating retargeting engine [9] as its co-processor. The reason of using single cycle 32 bits MIPS is to make the hardware as simple as possible. Therefore real number operations are compensated with fixed point representation in the software side along with complex hardware operations.

The software part starts with designing PDL cross compiler for MIPS that has not been proposed before. The PDL cross compiler is built using ANTLR tool [16] which is suitable for designing a prototype compiler, because it leaves code generation step to handle. In order to do use ANTLR tool, PDL grammar and PDL tree walker is required. PDL grammar consists of PDL exclusive grammar (available on [7]) and Tcl grammar that is built by reverse engineering the Tcl Book [15] due to no Tcl grammar availability [23]. The PDL tree walker walks the AST and generates MIPS machine code out of it. PDL tree walker organizes fixed point representation, procedure handling, instantiating self loop, register scheduling and execute expressions. The software uses Q15.16 fixed point representation to compensate the absence of FPU in the hardware side. Software emulated fixed point operations are implemented by adopting libfixmath.h [24] for Q15.16 fixed point representation. Despite the fact that there is no PDL cross compiler available, this thesis successfully implements a compiler for PDL with ANTLR tool. This thesis also provides Tcl grammar that is also unavailable by reverse engineering the Tcl Book.

Having complete design of IJTAG dependability processor from hardware and hardware parts. Then it needs to be evaluated to do its main purpose which is to execute dependability application. Before that the complete toolchain from PDL cross

compiler to an on-chip IJTAG dependability processor design needs to be verified with a benchmark test. Basic math packages from MiBench [29] is used for the benchmark test, because MiBench offers open source benchmarking for embedded processor. The benchmark tests consist of calculating square root operations, degree to radian conversion and radian to degree conversion. After that the complete toolchain is evaluated to execute a dependability application. This thesis uses acceleration factor calculation [28] for the dependability application test. The setup uses Virtex 7 VC707 FPGA from Xilinx and chipscope analyzer to monitor the output. The test shows that the IJTAG dependability processor has successfully executed the dependability application for acceleration factor calculation. It is verified by comparing the MATLAB and FPGA results. It can be concluded that the design of the on-chip IJTAG dependability processor is suitable to execute the dependability application using the IJTAG network. This solution will eases reliability engineers for developing dependability application, because increasing number of connected embedded instruments on the IJTAG network will not alter the dependability hardware. Only the software side that needs to be configured to fit the IJTAG network.

## 7.2 Future Works

There are several improvements that can be done to improve the on-chip IJTAG dependability processor. Those improvements are :

1. Test for different processor
2. Implement the rest of PDL commands
3. Improve the PDL cross compiler to be able to target other machine
4. Improve the PDL cross compiler to be able to use or to emulate C library for re-usability
5. Implement optimize code phase in PDL cross compiler
6. Use register allocation algorithm such as register coloring for optimum use of registers
7. Add complex number handler if necessary

# Bibliography


[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[2] M. K. Jain, M. Balakrishnan, and A. Kumar, "Asip design methodologies: survey and issues," in *VLSI Design, 2001. Fourteenth International Conference on*, 2001, pp. 76–81.

[3] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*, 4th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[4] "MIPS Architecture for Programmers Volume II-A: The MIPS32 Instruction Set Manual." [Online]. Available: https://imgtec.com/?do-download=4287

[5] V. P. Rubio and V. P. Rubio, "A fpga implementation of a mips risc processor for computer architecture education by," 2004.

[6] "VC707 Evaluation Board for the Virtex-7 FPGA." [Online]. Available: http://www.xilinx.com/support/documentation/boards_and_kits/vc707/ug885_VC707_Eval_Bd.pdf

[7] "IEEE Standard for Access and Control of Instrumentation Embedded within a Semiconductor Device," IEEE Std 1687-2014, 2014.

[8] "MIPS register." [Online]. Available: https://msdn.microsoft.com/en-us/library/ms253512(v=vs.90).aspx

[9] A. Ibrahim and H. G. Kerkhoff, "Analysis and Design of an On-Chip Retargeting Engine for IEEE 1687 Networks," *European Test Symposium*, April 2016.

[10] "Cross Compilation." [Online]. Available: https://www.gnu.org/savannah-checkouts/gnu/automake/manual/html_node/Cross_002dCompilation.html

[11] C. Ortega-Sanchez, “Minimips: An 8-bit mips in an fpga for educational purposes,” in *2011 International Conference on Reconfigurable Computing and FPGAs*, Nov 2011, pp. 152–157.

[12] K. J. Lee and G. Choi, “Design of a fault-tolerant microprocessor: a simulation approach,” in *Fault-Tolerant Systems, 1997. Proceedings., Pacific Rim International Symposium on*, Dec 1997, pp. 161–166.

[13] “FPU core - OpenCores.org.” [Online]. Available: http://opencores.org/project, fpu100

[14] B. W. Kernighan, *The C Programming Language*, 2nd ed., D. M. Ritchie, Ed. Prentice Hall Professional Technical Reference, 1988.

[15] J. K. Ousterhout, *Tcl and the Tk Toolkit*. Addison-Wesley, 1994, missing.

[16] “ANother Tool for Language Recognition.” [Online]. Available: http://antlr.org/

[17] “Beaver - a LALR Parser Generator.” [Online]. Available: http://beaver.sourceforge.net/

[18] “Yet Another Compiler-Compiler.” [Online]. Available: http://dinosaur.compilertools.net/yacc/index.html

[19] “Antlr Cheat Sheet.” [Online]. Available: https://theantlrguy.atlassian.net/wiki/display/ANTLR3/ANTLR+Cheat+Sheet

[20] H. Jiang, X. Wu, Y. Dong, and F. Zhang, “Implementing the compiler of aadl behavior annex using antlr,” in *2011 IEEE 2nd International Conference on Software Engineering and Service Science*, July 2011, pp. 191–195.

[21] C. Hu, R. Zhang, T. Wei, R. Wei, S. Li, and Y. Cheng, “Implementing the compiler of udlc,” in *2010 Ninth International Conference on Grid and Cloud Computing*, Nov 2010, pp. 383–387.

[22] T. Wei, R. Zhang, X. Su, S. Chen, and L. Li, “Gaussianscripteditor: An editor for gaussian scripting language for grid environment,” in *2009 Eighth International Conference on Grid and Cooperative Computing*, Aug 2009, pp. 39–44.

[23] “BNF for Tcl.” [Online]. Available: http://wiki.tcl.tk/1643

[24] “Libfixmath.” [Online]. Available: https://code.google.com/archive/p/libfixmath/

[25] “Eclipse Download Page.” [Online]. Available: https://eclipse.org/downloads/

[26] “Java Development Kit Download Page.” [Online]. Available: http://www.oracle.com/technetwork/java/javase/downloads/index.html

[27] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, “Mibench: A free, commercially representative embedded benchmark suite,” in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, ser. WWC ’01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 3–14. [Online]. Available: http://dx.doi.org/10.1109/WWC.2001.15

[28] Y. Zhao and H. G. Kerkhoff, “A genetic algorithm based remaining lifetime prediction for a vliw processor employing path delay and iddx testing,” in *2016 International Conference on Design and Technology of Integrated Systems in Nanoscale Era (DTIS)*, April 2016, pp. 1–4.

[29] “mibench home page.” [Online]. Available: http://vhosts.eecs.umich.edu/mibench//

# Appendix A

# An On-Chip IJTAG Dependability Processor

## A.1 IJTAG Dependability Processor

```
--------------------------------------------------------------------------------
-- Processor.vhd
--
-- Top Level entity of the single cycle MIPS processor
--
--------------------------------------------------------------------------------
-- Mochammad Fadhli Zakiy
-- University of Twente
-- 2016
--------------------------------------------------------------------------------
-- Design based on :
-- 1. http://chris.sagedy.com/projects/ecec490_fa08/
-- 2. Computer Organization and Design. Patterson & Hennessy
-- 3. MIPS Architecture for Programmers Volume II-A: The MIPS Instruction Set
--    Manual. 2015
--------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use work.ConstantsPkg.all;
use work.ComponentsPkg.all;

entity Processor is
```

```
  generic (
    InstructionMemContents : T_MemoryArray := (others => (others => '0'));
    DataMemContents        : T_MemoryArray := (others => (others => '0'))
    );
  port (
    in_Clock   : in std_logic;
    in_Reset   : in std_logic;
    in_SO      : in std_logic;
    out_SI     : out std_logic;
    out_RST    : out std_logic;
    out_Sel    : out std_logic;
    out_CE     : out std_logic;
    out_SE     : out std_logic;
    out_UE     : out std_logic;
    out_TCK    : out std_logic
    );
end Processor;

architecture structural of Processor is

  -- Processor
  signal PC : std_logic_vector(31 downto 0) := (others => '0');
  signal memAddress : std_logic_vector(31 downto 0);

  -- Instruction Memory
  signal Instruction : std_logic_vector(31 downto 0);

  -- Instruction Decoder
  signal Opcode   : std_logic_vector(5 downto 0);
  signal RS       : std_logic_vector(4 downto 0);
  signal RT       : std_logic_vector(4 downto 0);
  signal RD       : std_logic_vector(4 downto 0);
  signal Shamt    : std_logic_vector(4 downto 0);
  signal Funct    : std_logic_vector(5 downto 0);
  signal IAddress : std_logic_vector(15 downto 0);
  signal JAddress : std_logic_vector(25 downto 0);

  -- Controller
  signal RegDst       : std_logic;
```

```
signal Branch       : std_logic;
signal MemRead      : std_logic;
signal MemWrite     : std_logic;
signal ALUOp        : std_logic_vector(3 downto 0);
signal MemtoReg     : std_logic;
signal ALUSrc       : std_logic_vector(1 downto 0);
signal RegWriteMain : std_logic;
signal RegWriteCOP  : std_logic;
signal Jump         : std_logic;
signal RegJump      : std_logic;
signal DsttoSrc     : std_logic_vector(1 downto 0);
signal MaintoCOP    : std_logic;
signal COPtoMem     : std_logic;
signal COPtoReg     : std_logic;
signal RegtoCOP     : std_logic;
signal Ack          : std_logic;

-- Co-Processor Selector
signal Sel0         : std_logic;
signal Sel1         : std_logic;
signal Sel2         : std_logic;
signal Sel3         : std_logic;

-- Register File : Main Processor
signal WriteRegMain    : std_logic_vector(4 downto 0);
signal ReadReg1        : std_logic_vector(4 downto 0);
signal ReadDataMain1   : std_logic_vector(31 downto 0);
signal ReadDataMain2   : std_logic_vector(31 downto 0);
signal WriteBackData   : std_logic_vector(31 downto 0);
signal WriteBackCommon : std_logic_vector(31 downto 0);
signal WriteBackMain   : std_logic_vector(31 downto 0);

-- Register File Common
signal RegDataCOP        : std_logic_vector(31 downto 0);
signal DataOutCOP2       : std_logic_vector(31 downto 0);

-- Sign Extender
signal ExtendedAddressMain   : std_logic_vector(31 downto 0);
signal ExtendedShift         : std_logic_vector(31 downto 0);
```

```
  signal ExtendedAddressCOP    : std_logic_vector(31 downto 0);

  -- ALU
  signal InputA      : std_logic_vector(31 downto 0);
  signal InputB      : std_logic_vector(31 downto 0);
  signal ALUResult   : std_logic_vector(31 downto 0);
  signal ZeroFlag    : std_logic;

  -- Data Memory
  signal MemoryData : std_logic_vector(31 downto 0);
  signal MemDataIn  : std_logic_vector(31 downto 0);

begin

  memAddress <= "00000000000000000000000"& ALUResult(memLength+1 downto 0);

  -------------------------------------------------------------------------
  -- Instantiate the instruction memory.
  -------------------------------------------------------------------------
  InstructionMemory : Memory
    generic map (
      DefaultContents => InstructionMemContents
      )
    port map (
      in_Clock   => in_Clock,
      in_Reset   => in_Reset,
      in_Address => PC,
      in_Data    => (others => '0'),
      in_WriteEn => '0',
      in_ReadEn  => '1',
      out_Data   => Instruction
      );

  -------------------------------------------------------------------------
  -- Instantiate the instruction decoder.
  -------------------------------------------------------------------------
  InstructionDecode : InstructionDecoder
    port map (
      in_Instruction => Instruction,
```

```
    out_Opcode     => Opcode,
    out_RS         => RS,
    out_RT         => RT,
    out_RD         => RD,
    out_Shamt      => Shamt,
    out_Funct      => Funct,
    out_IAddress   => IAddress,
    out_JAddress   => JAddress
    );

-------------------------------------------------------------------------
-- Instantiate the main control block.
-------------------------------------------------------------------------
Control : Controller
  port map (
    in_Opcode            => Opcode,
    in_Funct             => Funct,
    in_Format            => RS,
    debug_Ack            => Ack,
    out_RegDst           => RegDst,
    out_Branch           => Branch,
    out_MemRead          => MemRead,
    out_MemWrite         => MemWrite,
    out_ALUOp            => ALUOp,
    out_MemtoReg         => MemtoReg,
    out_ALUSrc           => ALUSrc,
    out_RegWriteMain     => RegWriteMain,
    out_RegWriteCOP      => RegWriteCOP,
    out_Jump             => Jump,
    out_RegJump          => RegJump,
    out_DsttoSrc         => DsttoSrc,
    out_MaintoCOP        => MaintoCOP,
    out_COPtoMem         => COPtoMem,
    out_COPtoReg         => COPtoReg,
    out_RegtoCOP         => RegtoCOP
    );

-------------------------------------------------------------------------
-- Instantiate the register file.
```

```
-------------------------------------------------------------------------------

WriteRegMainMux : Mux2to1
  generic map (
    data  => 5
    )
  port map (
    in_Data0 => RT,
    in_Data1 => RD,
    out_Data => WriteRegMain,
    Sel      => RegDst
    );

WriteBackDataMux : Mux2to1
  generic map (
    data  => 32
    )
  port map (
    in_Data0 => ALUResult,
    in_Data1 => MemoryData,
    out_Data => WriteBackData,
Sel      => MemtoReg
    );

WriteBackCommonMux : Mux2to1
  generic map (
    data  => 32
    )
  port map (
    in_Data0 => WriteBackData,
    in_Data1 => RegDataCOP,
    out_Data => WriteBackCommon,
Sel      => COPtoReg
    );

WriteBackMainMux : Mux2to1
  generic map (
    data  => 32
    )
```

```
    port map (
      in_Data0 => WriteBackCommon,
      in_Data1 => ReadDataMain2,
      out_Data => WriteBackMain,
      Sel      => RegtoCOP
      );

  ReadReg1Mux : Mux4to1
    generic map (
      data  => 5
      )
    port map (
      in_Data00 => RS,
      in_Data01 => RT,
      in_Data10 => RD,
      in_Data11 => RS,
      out_Data  => ReadReg1,
  Sel       => DsttoSrc
      );

  SelMux : Decoder2to4
    port map (
      in_Sel    => Opcode(1 downto 0),
      out_Data0 => Sel0,
      out_Data1 => Sel1,
      out_Data2 => Sel2,
      out_Data3 => Sel3
      );

  RegFileMain : RegisterFile
    port map (
      in_Clock    => in_Clock,
      in_Reset    => in_Reset,
      in_ReadReg1 => ReadReg1,
      in_ReadReg2 => RT,
      in_WriteReg => WriteRegMain,
      in_Data     => WriteBackMain,
      in_WriteEn  => RegWriteMain,
      out_Data1   => ReadDataMain1,
```

```
    out_Data2   => ReadDataMain2
    );

-------------------------------------------------------------------------------
-- Instantiate the Co-Processor(s)
-------------------------------------------------------------------------------

COP2 : CoProcessor2
  port map (
    in_Clock    => in_Clock,
    in_Reset    => in_Reset,
    in_Instr    => Instruction,
    in_Data     => WriteBackMain,
    in_Reg      => WriteRegMain,
    in_MTC      => MaintoCOP,
    in_RWC      => RegWriteCOP,
    in_Sel      => Sel2,
    in_SO       => in_SO,
    out_Data    => DataOutCOP2,
    out_SI      => out_SI,
    out_RST     => out_RST,
    out_Sel     => out_Sel,
    out_CE      => out_CE,
    out_SE      => out_SE,
    out_UE      => out_UE,
    out_TCK     => out_TCK
  );

-------------------------------------------------------------------------------
-- Instantiate the sign extender 16 to 32.
-------------------------------------------------------------------------------
SignExtend16to32 : SignExtender
  generic map (
    InputWidth  => 16,
    OutputWidth => 32
    )
  port map (
    in_Data  => IAddress,
    out_Data => ExtendedAddressMain
```

```
    );

-------------------------------------------------------------------------------
-- Instantiate the sign extender 11 to 32.
-------------------------------------------------------------------------------
SignExtend11to32 : SignExtender
  generic map (
    InputWidth  => 11,
    OutputWidth => 32
    )
  port map (
    in_Data  => IAddress(10 downto 0),
    out_Data => ExtendedAddressCOP
    );

-------------------------------------------------------------------------------
-- Instantiate the sign extender 5 to 32.
-------------------------------------------------------------------------------
ExtendedShift <= "00000000000000000000000000" & IAddress(10 downto 6);

-------------------------------------------------------------------------------
-- Instantiate the ALU.
-------------------------------------------------------------------------------
InputA <= ReadDataMain1;

InputBMux : Mux4to1
  generic map (
    data  => 32
    )
  port map (
    in_Data00 => ReadDataMain2,
    in_Data01 => ExtendedAddressCOP,
    in_Data10 => ExtendedAddressMain,
    in_Data11 => ExtendedShift,
    out_Data  => InputB,
    Sel       => ALUSrc
    );

ALU : ALU32
```

```
  port map (
    in_Operation => ALUOp,
    in_A         => InputA,
    in_B         => InputB,
    out_Result   => ALUResult,
    out_Zero     => ZeroFlag
    );

-------------------------------------------------------------------------
-- Instantiate data memory.
-------------------------------------------------------------------------
MemDataInMux : Mux2to1
  generic map (
    data  => 32
    )
  port map (
    in_Data0 => ReadDataMain2,
    in_Data1 => RegDataCOP,
    out_Data => MemDataIn,
    Sel      => COPtoMem
    );

RegDataCOPMux : Mux4to1
  generic map (
    data  => 32
    )
  port map (
    in_Data00 => (others => '0'),
    in_Data01 => (others => '0'),
    in_Data10 => DataOutCOP2,
    in_Data11 => (others => '0'),
    out_Data  => RegDataCOP,
    Sel       => Opcode(1 downto 0)
    );

DataMemory : Memory
  generic map (
    DefaultContents => DataMemContents
    )
```

```
    port map (
      in_Clock   => in_Clock,
      in_Reset   => in_Reset,
      in_Address => memAddress,
      in_Data    => MemDataIn,
      in_WriteEn => MemWrite,
      in_ReadEn  => MemRead,
      out_Data   => MemoryData
      );

  ------------------------------------------------------------------------
  -- Update the program counter.
  ------------------------------------------------------------------------
  ThePC : ProgramCounter
  port map(
      in_Clock      => in_Clock,
      in_Reset      => in_Reset,
      in_Opcode     => Opcode,
      in_Jump       => Jump,
      in_RegJump    => RegJump,
      in_Branch     => Branch,
      in_ZeroFlag   => ZeroFlag,
      in_JAddress   => JAddress,
      in_ExtAddress => ExtendedAddressMain,
      in_ALUResult  => ALUResult,
      out_PC        => PC
      );

end structural;
```

## A.2 Retargeting Engine Co-Processor

```
------------------------------------------------------------------------
-- Retargeting Engine CoProcessor.vhd
--
-- Co-Processor for Retargeting Engine. The Only Co-Processor that connected
-- with IJTAG interface
--
```

```
--------------------------------------------------------------------------------
-- Mochammad Fadhli Zakiy
-- University of Twente
-- 2016
--------------------------------------------------------------------------------
-- Design based on :
-- 1. http://chris.sagedy.com/projects/ecec490_fa08/
-- 2. Computer Organization and Design. Patterson & Hennessy
-- 3. MIPS Architecture for Programmers Volume II-A: The MIPS Instruction Set
--    Manual. 2015
--------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use work.ConstantsPkg.all;
use work.ComponentsPkg.all;

entity CoProcessor2 is
  generic(
    MaxConcurrent : integer := 10
  );
  port (
    in_Clock : in  std_logic;
    in_Reset : in  std_logic;
    in_Instr : in  std_logic_vector(31 downto 0);
    in_Data  : in  std_logic_vector(31 downto 0);
    in_Reg   : in  std_logic_vector(4 downto 0);
    in_MTC   : in  std_logic;
    in_RWC   : in  std_logic;
    in_Sel   : in  std_logic;
    in_SO    : in  std_logic;
    out_Data : out std_logic_vector(31 downto 0);
    out_SI   : out std_logic;
    out_RST  : out std_logic;
    out_Sel  : out std_logic;
    out_CE   : out std_logic;
    out_SE   : out std_logic;
    out_UE   : out std_logic;
```

```
    out_TCK  : out std_logic
  );
end CoProcessor2;

architecture behavioral of CoProcessor2 is
-- Co-Processor State
type T_CommandQueue is array(MaxConcurrent-1 downto 0)
of std_logic_vector (64 downto 0);
type T_DataOutQueue is array(31 downto 0)
of std_logic_vector (63 downto 0);

type state_type is (readOrder, sendOrder, working, getData, sendData);
signal state       : state_type;

signal CommandQueue : T_CommandQueue;
signal DataOutQueue : T_DataOutQueue;

-- Instruction Decoder
signal Opcode   : std_logic_vector(5 downto 0);
signal Format   : std_logic_vector(4 downto 0);
signal RT       : std_logic_vector(4 downto 0);
signal RD       : std_logic_vector(4 downto 0);
signal Shamt    : std_logic_vector(4 downto 0);
signal Funct    : std_logic_vector(5 downto 0);
signal IAddress : std_logic_vector(15 downto 0);
signal JAddress : std_logic_vector(25 downto 0);

-- Co-Processor signal
signal command     : std_logic_vector(4 downto 0);
signal Data1       : std_logic_vector(31 downto 0);
signal Data2       : std_logic_vector(31 downto 0);
signal RegOut      : std_logic_vector(4 downto 0);
signal WriteEnable : std_logic;
signal DataOut     : std_logic_vector(31 downto 0);

-- Register signal
signal WriteRegCOP2  : std_logic_vector(4 downto 0);
signal WriteBackCOP2 : std_logic_vector(31 downto 0);
signal RegWriteCOP2  : std_logic;
```

```
-- Retargeting Engine signal
signal RegValue    : std_logic_vector(31 downto 0);
signal RegId       : std_logic_vector(31 downto 0);
signal Concurrent  : std_logic;
signal ReadWrite   : std_logic;
signal ReadValue   : std_logic_vector(31 downto 0);
signal ReadId      : std_logic_vector(31 downto 0);
signal RequestedId : std_logic_vector(31 downto 0);
signal error       : std_logic;
signal ACK         : std_logic;

-- IJTAG signal
signal SO          : std_logic;
signal SI          : std_logic;
signal RST         : std_logic;
signal Sel         : std_logic;
signal CE          : std_logic;
signal SE          : std_logic;
signal UE          : std_logic;
signal TCK         : std_logic;
begin

out_Data <= Data1;

-------------------------------------------------------------------------------
-- Instantiate the instruction decoder.
-------------------------------------------------------------------------------
  InstructionDecode2 : InstructionDecoder
    port map (
      in_Instruction => in_Instr,
      out_Opcode     => Opcode,
      out_RS         => Format,
      out_RT         => RT,
      out_RD         => RD,
      out_Shamt      => Shamt,
      out_Funct      => Funct,
      out_IAddress   => IAddress,
      out_JAddress   => JAddress
```

```
      );

-------------------------------------------------------------------------
-- Instantiate the register file for COP 2
-------------------------------------------------------------------------
  RegWriteCOP2    <= (in_RWC AND in_Sel) OR WriteEnable;

  RegFileCOP2 : RegisterFile
    port map (
      in_Clock    => in_Clock,
      in_Reset    => in_Reset,
      in_ReadReg1 => RT,
      in_ReadReg2 => RD,
      in_WriteReg => WriteRegCOP2,
      in_Data     => WriteBackCOP2,
      in_WriteEn  => RegWriteCOP2,
      out_Data1   => Data1,
      out_Data2   => Data2
      );

  WriteRegCOP2Mux : Mux2to1
    generic map (
      data  => 5
      )
    port map (
      in_Data0 => in_Reg,
      in_Data1 => RegOut,
      out_Data => WriteRegCOP2,
      Sel      => WriteEnable
      );

  WriteBackCOP2Mux : Mux2to1
    generic map (
      data  => 32
      )
    port map (
      in_Data0 => DataOut,
      in_Data1 => in_Data,
      out_Data => WriteBackCOP2,
```

```
  Sel      => in_MTC
      );

-------------------------------------------------------------------------------
-- Instantiate Retargeting Engine
-------------------------------------------------------------------------------
  Retargeting : RetargetingEngine
    port map (
      in_Clock        => in_Clock,
      in_Reset        => in_Reset,
      in_RegValue     => RegValue,
      in_RegId        => RegId,
      in_Concurrent   => Concurrent,
      in_ReadWrite    => ReadWrite,
      in_SO           => SO,
      out_ReadValue   => ReadValue,
      out_ReadId      => ReadId,
      out_RequestedId => RequestedId,
      out_error       => error,
      out_ACK         => ACK,
      out_SI          => SI,
      out_RST         => RST,
      out_Sel         => Sel,
      out_CE          => CE,
      out_SE          => SE,
      out_UE          => UE,
      out_TCK         => TCK
      );

-------------------------------------------------------------------------------
-- Co-Processor 2 State Machine
-------------------------------------------------------------------------------
  stateMachine:process(in_Clock, in_Reset, Format)
    variable commandCounter : integer := 0;
    variable totalRead      : integer range 0 to 31 := 0;
    variable readCounter    : integer range 0 to 31 := 0;
    variable sendCounter    : integer range 0 to 31 := 0;
    variable workCounter    : integer := 0;
    variable I              : integer range 0 to 31 := 0;
```

```
begin
if(in_Reset='1') then
  RegValue   <= (others => '0');
  RegId      <= (others => '0');
  ReadWrite  <= '0';
  Concurrent <= '0';
  WriteEnable<= '0';
  RegOut    <= (others => '0');
  DataOut   <= (others => '0');
  state <= readOrder;
  commandCounter := 0;
  totalRead      := 0;
  readCounter    := 0;
  sendCounter    := 0;
  workCounter    := 0;


elsif(rising_edge(in_Clock)) then
  if (state = readOrder) then
      RegValue    <= (others => '0');
      RegId       <= (others => '0');
      ReadWrite   <= '0';
      Concurrent  <= '0';
      WriteEnable <= '0';
      RegOut      <= (others => '0');
      DataOut     <= (others => '0');

      if (in_Sel = '1') then
          case Format is
              when FORMAT_IWRITE =>
                  CommandQueue(commandCounter) <= Format(0) & Data1 & Data2;

                  state <= readOrder;
                  commandCounter := commandCounter + 1;

              when FORMAT_IREAD =>
                  CommandQueue(commandCounter) <= Format(0) & Data1 & Data2;
                  DataOutQueue(readCounter) <= Data1 & x"00000000";
```

```
                state <= readOrder;
                commandCounter := commandCounter + 1;
                readCounter := readCounter + 1;

            when FORMAT_IAPPLY =>
                RegValue    <= CommandQueue(workCounter)(31 downto 0);
                RegId       <= CommandQueue(workCounter)(63 downto 32);
                ReadWrite   <= CommandQueue(workCounter)(64);
                Concurrent  <= '1';

                state       <= sendOrder;
                workCounter := workCounter + 1;
                totalRead   := readCounter;


            when FORMAT_IRESET =>
                state          <= readOrder;
                commandCounter := 0;
                totalRead   := 0;
                readCounter := 0;
                sendCounter := 0;
                workCounter := 0;

            when others =>
                state <= readOrder;
        end case;
     else
        state <= readOrder;
    end if;

elsif (state = sendOrder) then
    WriteEnable   <= '0';
    RegOut        <= (others => '0');
    DataOut       <= (others => '0');

    if(workCounter < commandCounter) then
        RegValue    <= CommandQueue(workCounter)(31 downto 0);
        RegId       <= CommandQueue(workCounter)(63 downto 32);
        ReadWrite   <= CommandQueue(workCounter)(64);
```

```
            Concurrent  <= '1';

            state       <= sendOrder;
            workCounter := workCounter + 1;
        else
            RegValue    <= (others => '0');
            RegId       <= (others => '0');
            ReadWrite   <= '0';
            Concurrent  <= '0';

            state       <= working;
            workCounter    := 0;
            commandCounter := 0;
        end if;

    elsif (state = working) then
        RegValue       <= (others => '0');
        RegId          <= (others => '0');
        ReadWrite      <= '0';
        Concurrent     <= '0';
        WriteEnable    <= '0';
        RegOut        <= (others => '0');
        DataOut       <= (others => '0');
        if(ACK ='1') then
            if (readCounter > 0) then
                state       <= getData;
            else
                WriteEnable   <= '1';
                RegOut  <= "00001";
                DataOut <= x"00000001";
                state    <= readOrder;
            end if;
        end if;

    elsif (state = getData) then
        RegValue       <= (others => '0');
        RegId          <= (others => '0');
        ReadWrite      <= '0';
        Concurrent     <= '0';
```

```
        WriteEnable    <= '0';
        RegOut         <= (others => '0');
        DataOut        <= (others => '0');

        for I in 0 to 31 loop
            if (ReadId = DataOutQueue(I)(63 downto 32)) then
                DataOutQueue(I)(31 downto 0) <= ReadValue;
            end if;
        end loop;

        if(readCounter = 1) then
            state          <= sendData;
        else
            state          <= getData;
            readCounter := readCounter - 1;
        end if;

    else
        if (sendCounter < totalRead) then
            RegValue       <= (others => '0');
            RegId          <= (others => '0');
            ReadWrite      <= '0';
            Concurrent     <= '0';
            RegOut         <= std_logic_vector(to_unsigned(sendCounter + 8, 5));
            DataOut        <= DataOutQueue(sendCounter)(31 downto 0);
            WriteEnable    <= '1';
            state          <= sendData;

            sendCounter    := sendCounter + 1;

        else
            RegValue       <= (others => '0');
            RegId          <= (others => '0');
            ReadWrite      <= '0';
            Concurrent     <= '0';
            RegOut         <= "00001";
            DataOut        <= x"00000001";
            WriteEnable    <= '1';
            state          <= readOrder;
```

```
            sendCounter    := 0;
            readCounter    := 0;

        end if;
    end if;
  end if;
  end process;
end behavioral;
```

# Appendix B

# Software Emulated Fixed Point Operations

## B.1 Emulated Fixed Point Multiplication

In order to emulate fixed point multiplication operation, one must understand how the multiplication works. Multiplication of two 32 bits numbers will produce a 64 bits number (figure B.1). Hardware multiplication usually divides this 64 bits number into two 32 bits numbers: $HI$ and $LO$. However this 64 bits number can be truncated into a 32 bits number by taking 16 bits LSB of $HI$ and put it is as the MSB of the result. Then followed with taking 16 bits MSB of $LO$ and put it as the LSB of the result. This solution is restricted for small numbers only. Multiplication that produces more than 16 bits integer will be truncated into 16 bits integer that produce incorrect result.

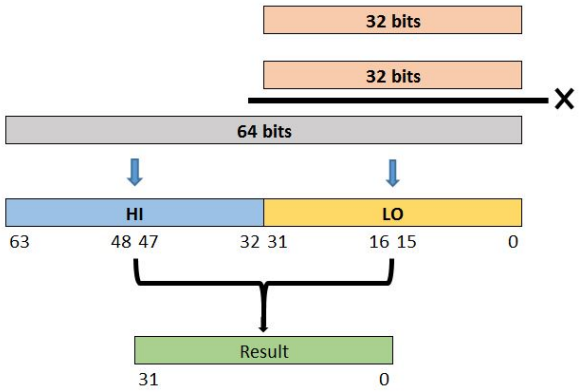



**Figure B.1:** Hardware multiplication concept

Full fixed point multiplication algorithm can be seen on algorithm 4. It begins with assigning the multiplier $m$ and multiplicand $n$ into positive values. It will raise the $negM$ and/or $negN$ flags into $1$ respectively if the multiplier $m$ and/or multiplicand $n$

are negative values. $negR$ flags are raised to 1 if either $negM$ or $negN$ flags are raised. Then the algorithm moves to calculate $AC$, $AD$, $CB$ and $BD$ by using a $MultiplicationLoop$ algorithm. Next $AD\_CB$ is calculated from addition of $AD$ and $CB$, this $AD\_CB$ is an intersect value of $HI$ and $LO$. Because $HI$ is an addition between $AC$ and 16 bits MSB of $AD\_CB$. On the other hand $LI$ is an addition between $BD$ and 16 bits LSB of $AD\_CB$. Addition between $BD$ and 16 bits LSB of $AD\_CB$ might produce an overflow which increases the value of $HI$. Since $HI$ and $LO$ are represented separately, so if the value of $LO$ is less than $0$ (overflow detected), the value of $HI$ is increased by 1. Truncating the $HI$ and $LO$ into a 32 bits number by taking 16 bits LSB of $HI$ then put it is as the MSB of the result and taking 16 bits LSB of $LO$ then put it is as the MSB of the result. Back to $negR$ flag, if it is raised then the result is turned to negative value. Finally the result of fixed point multiplication is ready. These steps are depicted on figure B.2.

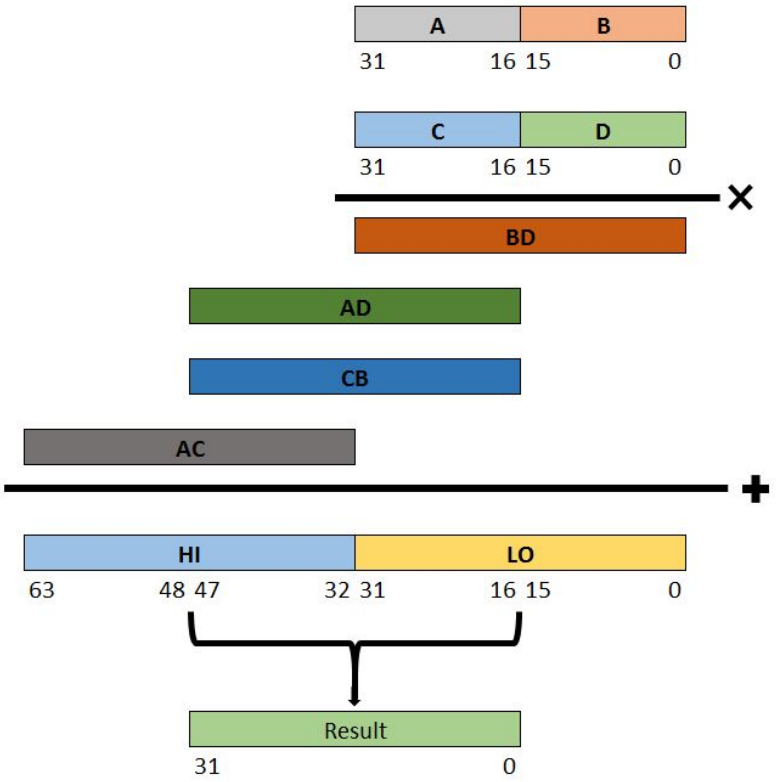



**Figure B.2:** Example of Expression AST

**Algorithm 4** Fixed Point Multiplication Algorithm

```
1: procedure MULTIPLICATIONLOOP(m,n)
2:     result ← 0
3:     while m do
4:         if m & 1 then
5:             result ← result + n
6:         end if
7:         n ← n << 1
8:         m ← m >> 1
9:     end while
10:    return result
11: end procedure
12: procedure FIXEDPOINTMULTIPLICATION(m,n)
13:    result ← 0
14:    negM ← 0
15:    negN ← 0
16:    negR ← 0
17:    if m < 0 then
18:        negM ← 1
19:        m ← −m
20:    end if
21:    if n < 0 then
22:        negN ← 1
23:        n ← −n
24:    end if
25:    negR ← negM ⊕ negN
26:    A ← (m >> 16) & 65535
27:    B ← m >> & 65535
28:    C ← (n >> 16) & 65535
29:    D ← n & 65535
30:    AC ← MultiplicationLoop(A,C)
31:    AD ← MultiplicationLoop(A,D)
32:    CB ← MultiplicationLoop(C,B)
33:    BD ← MultiplicationLoop(B,D)
34:    AD_CB ← AD + CB
35:    HI ← AC + ((AD_CB >> 16) & 65536)
36:    LO ← BD + (AD_CB << 16)
```

37: **if** $LO < 0$ **then**
38: $HI \leftarrow HI + 1$
39: **end if**
40: $result \leftarrow (HI << 16) \mid (LO >> 16)$
41: **if** $negR$ **then**
42: $result \leftarrow -result$
43: **end if**
44: return $result$
45: **end procedure**

## B.2 Emulated Fixed Point Division

In order to emulate fixed point division operation, one must understand how it works. Full fixed point division algorithm can be seen on algorithm 5. It begins with checking the denominator $b$. If $b$ is $0$, it should return errors. However on this thesis, the ErrorHandler java class has not been developed perfectly, so it will return $0$ instead. Then the algorithm moves to assign $bitFlag$ with $0\text{x}10000$ which is the first integer bit on Q15.16 fixed point representation. This $bitFlag$ will be used for detecting whether $a$ is still available for division. Next the algorithm moves to assign the numerator $a$ and denominator $b$ into positive values. It will raise the $negA$ and/or $negB$ flags into $1$ respectively if the numerator $a$ and/or denominator $b$ are negative values. $negR$ flags are raised to 1 if either $negA$ or $negB$ flags are raised. Then the algorithm shifts $b$ to left by 1 bit and shifts $bitFlag$ to left by 1 bit when the value of $b$ is less than $a$. This operations are necessary to make the values of $b$ and $a$ divisible. If $b$ has became negative and the value of $a$ is bigger than equal to $b$, then try to set $a$ as subtraction of $a$ with $b$ and sets $result$ with or operation between $result$ with $bitFlag$. And followed with shifting $b$ value to right by 1 bit and shifting $bitFlag$ to right by 1 bit. Next while bitwise and operation between $bitFlag$ and $a$ is not $0$ and the value of $a$ is bigger than equal to $b$, then try to set $a$ as subtraction of $a$ with $b$ and sets $result$ with or operation between $result$ with $bitFlag$. And followed with shifting $a$ value to left by 1 bit and shifting $bitFlag$ to right by 1 bit. Back to $negR$ flag, if it is raised then the result is turned to negative value. Finally the result of fixed point division is ready.

## B.3 Emulated Fixed Point Square Root

In order to emulate fixed point square root operation, one must understand how it works. Full fixed point square root algorithm can be seen on algorithm 6. It begins with assigning the value of $HI$ with $0$ and the value of $LO$ to radicand $x$. Due

**Algorithm 5** Fixed Point Division Algorithm

```
1: procedure FIXEDPOINTDIVISION(a,b)
2:     result ← 0
3:     negA ← 0
4:     negB ← 0
5:     negR ← 0
6:     bitFlag ← 0x10000
7:     if b == 0 then
8:         return result
9:     end if
10:    if a < 0 then
11:        negA ← 1
12:        a ← −a
13:    end if
14:    if b < 0 then
15:        negB ← 1
16:        b ← −b
17:    end if
18:    negR ← negA ⊕ negB
19:    while b < a do
20:        b ← b << 1
21:        bitFlag ← bitFlag << 1
22:    end while
23:    if b & 0x80000000 then
24:        if a >= b then
25:            result ← result | bitFlag
26:            a ← a − b
27:        end if
28:        b ← b >> 1
29:        bitFlag ← bitFlag >> 1
30:    end if
31:    while bitFlag && a do
32:        if a >= b then
33:            result ← result | bitFlag
34:            a ← a − b
35:        end if
36:        a ← a << 1
37:        bitFlag ← bitFlag >> 1
38:    end while
```

39: **if** $negR$ **then**
40: $result \leftarrow -result$
41: **end if**
42: return $result$
43: **end procedure**

to Q15.16 fixed point representation, $wordLength$ is assigned with $32$ and $frac$ is assigned with $16$. If the radicand $x$ is less than $0$, it should return complex value on the end. However on this thesis, there is no complex number representation yet. Therefore handling complex number will return $0$ instead. Next for $i$ starts from $0$ to $i$ is less than $24$ ($wordLength - (frac >> 1)$) do square root calculation. Square root calculation comprises of :
1. assigning $HI$ with addition of shifted $HI$ value to the left by 2 bits with 2 bits MSB of $LO$,
2. assigning $LO$ with shifted $LO$ value to the left by 2 bits,
3. assigning $result$ with shifted $result$ value to the left by 1 bit,
4. assigning $div$ with addition of shifted $result$ value to the left by 1 bit with $1$,
5. assigning $HI$ with subtraction of $HI$ with $div$ and assigning $result$ with addition of $result$ with $1$ if the value of $HI$ is bigger than equal to the value of $div$,
6. Lastly, assigning $i$ with addition of $i$ with $1$.
Finally the result of fixed point square root is ready.

## B.4 Emulated Fixed Point Power

In order to emulate fixed point power operation, one must understand how it works. Full fixed point power algorithm can be seen on algorithm 7. It begins with assigning the value of $result$ with $0$x$10000$, due to Q15.16 fixed point representation. If the power $y$ is less than $0$, $y$ is turned into negative value and $negR$ flag is raised to $1$. Then the algorithm moves to assign $powReal$ with shifted $y$ to right by 16 for get rid of the fractional part and assigning $temp$ with the value of base $x$. Then while the value of $powReal$ is not zero do the power operation for integer part which comprises of :
1. assigning $result$ with fixed point multiplication between $result$ and $temp$, if the and bitwise operation between $powReal$ and $1$ is not $0$,
2. assigning $powReal$ with shifted $powReal$ to left by 1 bit,
3. assigning $temp$ with fixed point multiplication between $temp$ and $temp$.
After that the algorithm moves to assign $powFrac$ with shifted $y$ to left by 16 for

**Algorithm 6** Fixed Point Square Root Algorithm

```
1: procedure FIXEDPOINTSQUAREROOT(x)
2:     result ← 0
3:     HI ← 0
4:     LO ← x
5:     wordLength ← 32
6:     frac ← 16
7:     if x < 0 then
8:         return result
9:     end if
10:    i ← 0
11:    for i < wordLength − (frac >> 1) do
12:        HI ← (HI << 2) + ((LO >> 30) & 3)
13:        LO ← LO << 2
14:        result ← result << 1
15:        div ← (result << 1) + 1
16:        if HI >= div then
17:            HI ← HI − div
18:            result ← result + 1
19:        end if
20:        i ← i + 1
21:    end for
22:    return result
23: end procedure
```

getting rid of the integer part and assigning $temp$ with fixed point operation of base $x$. Then while the value of $powFrac$ is not zero do the power operation for fractional part which comprises of :
1. assigning $result$ with fixed point multiplication between $result$ and $temp$, if the and bitwise operation between $powFrac$ and $0$x$80000000$ is not $0$,
2. assigning $powFrac$ with shifted $powReal$ to right by 1 bit,
3. assigning $temp$ with fixed point square root of $temp$.
Back to $negR$ flag, if it is raised then the result is assigned with fixed point division between $0$x$10000$ and $result$. Because negative value of $y$ on the beginning means a division between $1$ with $result$.Finally the result of fixed point power is ready.

**Algorithm 7** Fixed Point Power Algorithm

```
1: procedure FIXEDPOINTPOWER(x,y)
2:     result ← 0x10000
3:     negR ← 0
4:     if y < 0 then
5:         y ← −y
6:         negR ← 1
7:     end if
8:     powReal ← y >> 16
9:     temp ← x
10:    while powReal do
11:        if powReal & 1 then
12:            result ← FixedPointMultiplication(result,temp)
13:        end if
14:        powReal ← powReal >> 1
15:        temp ← FixedPointMultiplication(temp,temp)
16:    end while
17:    powFrac ← y << 16
18:    temp ← FixedPointSquareRoot(temp)
19:    while powFrac do
20:        if powFrac & 0x80000000 then
21:            result ← FixedPointMultiplication(result,temp)
22:        end if
23:        powFrac ← powFrac << 1
24:        temp ← FixedPointSquareRoot(temp)
25:    end while
26:    if negR then
27:        result ← FixedPointDivision(0x10000,result)
28:    end if
29:    return result
30: end procedure
```

# Appendix C

# Setup Environment

## C.1 XilinxTopLevel

```
--------------------------------------------------------------------------------
-- xilinxTopLevel.vhd
--
-- Testbench for 32 bits PDL-MIPS processor for xilinx
--
--------------------------------------------------------------------------------
-- Mochammad Fadhli Zakiy
-- University of Twente
-- 2016
--------------------------------------------------------------------------------
-- Design based on :
-- 1. http://chris.sagedy.com/projects/ecec490_fa08/
-- 2. Computer Organization and Design. Patterson & Hennessy
-- 3. MIPS Architecture for Programmers Volume II-A: The MIPS Instruction Set
--    Manual. 2015
--------------------------------------------------------------------------------

LIBRARY IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use work.ComponentsPkg.all;
use work.ConstantsPkg.all;
use std.textio.all;
use IEEE.std_logic_textio.all;
Library UNISIM;
use UNISIM.vcomponents.all;
```

```
entity xilinxTopLevel is
port
(
    clock_n    : IN  std_logic;
    clock_p    : IN  std_logic;
    in_Reset   : IN  std_logic;
    in_temp0   : IN  std_logic_vector(3 downto 0);
    in_temp1   : IN  std_logic_vector(3 downto 0);
);
end xilinxTopLevel;

architecture test of xilinxTopLevel is

------------------------------------------------------------------------
-- Component instantiation
------------------------------------------------------------------------
    component ICON
    PORT (
        CONTROL0 : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0));
    end component;

    component ILA
    PORT (
        CONTROL : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0);
        CLK : IN STD_LOGIC;
        TRIG0 : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
        TRIG1 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        TRIG2 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        TRIG3 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        TRIG4 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        TRIG5 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        TRIG6 : IN STD_LOGIC_VECTOR(31 DOWNTO 0));
    end component;

------------------------------------------------------------------------
-- Function
------------------------------------------------------------------------
    impure function InitRomFromFile (RomFileName : in string)
```

```
    return T_MemoryArray is
        FILE romfile : text is in RomFileName;
        variable RomFileLine1 : line;
        variable RomFileLine2 : line;
        variable RomFileLine3 : line;
        variable RomFileLine4 : line;
        variable rom : T_MemoryArray;
        variable mem1 : std_logic_vector(31 downto 0);
        variable mem2 : std_logic_vector(31 downto 0);
        variable mem3 : std_logic_vector(31 downto 0);
        variable mem4 : std_logic_vector(31 downto 0);
    begin
        for i in T_MemoryArray'range loop
            readline(romfile, RomFileLine1);
            readline(romfile, RomFileLine2);
            readline(romfile, RomFileLine3);
            readline(romfile, RomFileLine4);
            hread(RomFileLine1, mem1);
            hread(RomFileLine2, mem2);
            hread(RomFileLine3, mem3);
            hread(RomFileLine4, mem4);
            rom(i):=mem4&mem3&mem2&mem1;
        end loop;
        return rom;
    end function;

-------------------------------------------------------------------------
-- Signal instantiation
-------------------------------------------------------------------------
-- keep unoptimized for memory signal
attribute keep : string;

-- memory instantiation
signal InstructionMemContents : T_MemoryArray := InitRomFromFile
("instruction.data");
signal DataMemContents        : T_MemoryArray := InitRomFromFile
("memory.data");

-- keep memory
```

```
attribute keep of InstructionMemContents : signal is "true";
attribute keep of DataMemContents        : signal is "true";

-- clock
signal diffClock : std_logic;
signal sysClock  : std_logic;

-- IJTAG Interface
signal SO  : std_logic;
signal SI  : std_logic;
signal RST : std_logic;
signal Sel : std_logic;
signal CE  : std_logic;
signal SE  : std_logic;
signal UE  : std_logic;
signal TCK : std_logic;

-- Debug Interface
signal CONTROL0 : STD_LOGIC_VECTOR(35 DOWNTO 0);
Signal TRIG0    : STD_LOGIC_VECTOR(0 DOWNTO 0);
Signal TRIG1    : STD_LOGIC_VECTOR(31 DOWNTO 0);
Signal TRIG2    : STD_LOGIC_VECTOR(31 DOWNTO 0);
Signal TRIG3    : STD_LOGIC_VECTOR(31 DOWNTO 0);
Signal TRIG4    : STD_LOGIC_VECTOR(31 DOWNTO 0);
Signal TRIG5    : STD_LOGIC_VECTOR(31 DOWNTO 0);
Signal TRIG6    : STD_LOGIC_VECTOR(31 DOWNTO 0);

-- Debug Signal
signal sigAck  : STD_LOGIC;
signal sigData : STD_LOGIC_VECTOR(31 DOWNTO 0);
signal sigInst : STD_LOGIC_VECTOR(31 DOWNTO 0);
signal sigPC   : STD_LOGIC_VECTOR(31 DOWNTO 0);
signal sigA    : STD_LOGIC_VECTOR(31 DOWNTO 0);
signal sigB    : STD_LOGIC_VECTOR(31 DOWNTO 0);
signal sigALU  : STD_LOGIC_VECTOR(31 DOWNTO 0);

begin

    TRIG0(0) <= sigAck;
```

```
    TRIG1    <= sigData;
    TRIG2    <= sigInst;
    TRIG3    <= sigPC;
    TRIG4    <= sigA;
    TRIG5    <= sigB;
    TRIG6    <= sigALU;

------------------------------------------------------------------------
-- Instantiate clock
------------------------------------------------------------------------
    IBUFDS_inst : IBUFDS
    generic map (
        DIFF_TERM => FALSE,
        IBUF_LOW_PWR => TRUE,
        IOSTANDARD => "DEFAULT")
    port map (
        O  => diffClock,
        I  => clock_p,
        IB => clock_n
    );

------------------------------------------------------------------------
-- Instantiate Clock Divider
------------------------------------------------------------------------
    theClockDiv : ClockDiv
    generic map (
        divider => 2
    )
    port map (
        in_Clock  => diffClock,
        in_Reset  => in_Reset,
        out_Clock => sysClock
    );

------------------------------------------------------------------------
-- Instantiate ICON
------------------------------------------------------------------------
    ICON_inst : ICON
    port map (
```

```
        CONTROL0 => CONTROL0
    );

-------------------------------------------------------------------------------
-- Instantiate ILA
-------------------------------------------------------------------------------
    ILA_inst : ILA
    port map (
        CONTROL => CONTROL0,
        CLK => sysClock,
        TRIG0 => TRIG0,
        TRIG1 => TRIG1,
        TRIG2 => TRIG2,
        TRIG3 => TRIG3,
        TRIG4 => TRIG4,
        TRIG5 => TRIG5,
        TRIG6 => TRIG6
    );

-------------------------------------------------------------------------------
-- Instantiate the processor.
-------------------------------------------------------------------------------
    MIPS_Processor : DebugProcessor
    generic map (
        InstructionMemContents => InstructionMemContents,
        DataMemContents => DataMemContents
    )
    port map (
        in_Clock   => sysClock,
        in_Reset   => in_Reset,
        in_SO      => SO,
        in_temp0   => in_temp0,
        in_temp1   => in_temp1,
        debug_Ack  => sigAck,
        debug_Data => sigData,
        debug_Inst => sigInst,
        debug_PC   => sigPC,
        debug_A    => sigA,
        debug_B    => sigB,
```

```
        debug_ALU  => sigALU,
        out_SI     => SI,
        out_RST    => RST,
        out_Sel    => Sel,
        out_CE     => CE,
        out_SE     => SE,
        out_UE     => UE,
        out_TCK    => TCK
    );
end test;
```

## C.2 Xilinx Top Level UCF

```
# PlanAhead Generated physical constraints

NET "clock_n"  LOC = E18;
NET "clock_p"  LOC = E19;
NET "in_Reset" LOC = AV39;
NET "in_temp0[0]"  LOC = AV30;
NET "in_temp0[1]"  LOC = AY33;
NET "in_temp0[2]"  LOC = BA31;
NET "in_temp0[3]"  LOC = BA32;
NET "in_temp1[0]"  LOC = AW30;
NET "in_temp1[1]"  LOC = AY30;
NET "in_temp1[2]"  LOC = BA30;
NET "in_temp1[3]"  LOC = BB31;
NET "Data[0]"  LOC = AM22;
NET "Data[1]"  LOC = AL22;
NET "Data[2]"  LOC = AJ20;
NET "Data[3]"  LOC = AJ21;
NET "Data[4]"  LOC = AM21;
NET "Data[5]"  LOC = AL21;
NET "Data[6]"  LOC = AK22;
NET "Data[7]"  LOC = AJ22;
NET "Data[8]"  LOC = AL20;
NET "Data[9]"  LOC = AK20;
NET "Data[10]" LOC = AK23;
NET "Data[11]" LOC = AJ23;
```

```
NET "Data[12]" LOC = AN21;
NET "Data[13]" LOC = AP22;
NET "Data[14]" LOC = AP23;
NET "Data[15]" LOC = AN23;
NET "Data[16]" LOC = AM23;
NET "Data[17]" LOC = AN24;
NET "Data[18]" LOC = AY24;
NET "Data[19]" LOC = BB22;
NET "Data[20]" LOC = BA22;
NET "Data[21]" LOC = BA25;
NET "Data[22]" LOC = AY25;
NET "Data[23]" LOC = AY22;
NET "Data[24]" LOC = AY23;
NET "Data[25]" LOC = AV24;
NET "Data[26]" LOC = AU24;
NET "Data[27]" LOC = AW21;
NET "Data[28]" LOC = AV21;
NET "Data[29]" LOC = AT24;
NET "Data[30]" LOC = AR24;
NET "Data[31]" LOC = AU21;
NET "Ack"      LOC = AT21;

# PlanAhead Generated IO constraints

NET "clock_n" IOSTANDARD = LVDS;
NET "clock_p" IOSTANDARD = LVDS;
NET "in_Reset"  IOSTANDARD = LVCMOS18;
NET "in_temp0[0]"  IOSTANDARD = LVCMOS18;
NET "in_temp0[1]"  IOSTANDARD = LVCMOS18;
NET "in_temp0[2]"  IOSTANDARD = LVCMOS18;
NET "in_temp0[3]"  IOSTANDARD = LVCMOS18;
NET "in_temp1[0]"  IOSTANDARD = LVCMOS18;
NET "in_temp1[1]"  IOSTANDARD = LVCMOS18;
NET "in_temp1[2]"  IOSTANDARD = LVCMOS18;
NET "in_temp1[3]"  IOSTANDARD = LVCMOS18;
NET "Data[0]"   IOSTANDARD = LVCMOS18;
NET "Data[1]"   IOSTANDARD = LVCMOS18;
NET "Data[2]"   IOSTANDARD = LVCMOS18;
NET "Data[3]"   IOSTANDARD = LVCMOS18;
```

```
NET "Data[4]"   IOSTANDARD = LVCMOS18;
NET "Data[5]"   IOSTANDARD = LVCMOS18;
NET "Data[6]"   IOSTANDARD = LVCMOS18;
NET "Data[7]"   IOSTANDARD = LVCMOS18;
NET "Data[8]"   IOSTANDARD = LVCMOS18;
NET "Data[9]"   IOSTANDARD = LVCMOS18;
NET "Data[10]"  IOSTANDARD = LVCMOS18;
NET "Data[11]"  IOSTANDARD = LVCMOS18;
NET "Data[12]"  IOSTANDARD = LVCMOS18;
NET "Data[13]"  IOSTANDARD = LVCMOS18;
NET "Data[14]"  IOSTANDARD = LVCMOS18;
NET "Data[15]"  IOSTANDARD = LVCMOS18;
NET "Data[16]"  IOSTANDARD = LVCMOS18;
NET "Data[17]"  IOSTANDARD = LVCMOS18;
NET "Data[18]"  IOSTANDARD = LVCMOS18;
NET "Data[19]"  IOSTANDARD = LVCMOS18;
NET "Data[20]"  IOSTANDARD = LVCMOS18;
NET "Data[21]"  IOSTANDARD = LVCMOS18;
NET "Data[22]"  IOSTANDARD = LVCMOS18;
NET "Data[23]"  IOSTANDARD = LVCMOS18;
NET "Data[24]"  IOSTANDARD = LVCMOS18;
NET "Data[25]"  IOSTANDARD = LVCMOS18;
NET "Data[26]"  IOSTANDARD = LVCMOS18;
NET "Data[27]"  IOSTANDARD = LVCMOS18;
NET "Data[28]"  IOSTANDARD = LVCMOS18;
NET "Data[29]"  IOSTANDARD = LVCMOS18;
NET "Data[30]"  IOSTANDARD = LVCMOS18;
NET "Data[31]"  IOSTANDARD = LVCMOS18;
NET "Ack"       IOSTANDARD = LVCMOS18;

NET "in_Reset" CLOCK_DEDICATED_ROUTE = FALSE;
```