Configurable Verification of RISC Processors

Namratha Pashupathy Manjula Devi
nxp6856@rit.edu

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
CONFIGURABLE VERIFICATION OF RISC PROCESSORS

by

Namratha Pashupathy Manjula Devi

A Graduate Paper Submitted

in

Partial Fulfillment

of the

Requirements for the Degree of

MASTER OF SCIENCE

in

Electrical Engineering

Approved by:

PROF. _____________________________________________

(Graduate Paper Advisor - Prof. Mark A. Indovina)

PROF. _____________________________________________

(Department Head - Dr. Sohail A. Dianat)

DEPARTMENT OF ELECTRICAL AND MICROELECTRONIC ENGINEERING

KATE GLEASON COLLEGE OF ENGINEERING

ROCHESTER INSTITUTE OF TECHNOLOGY

ROCHESTER, NEW YORK

MAY, 2017
I would like to dedicate this work to my family, my father Dr Pashupathy, my mother Dr ManjulaDevi, my brother Shamanth, and friends for their love and support during my thesis.
Declaration

I hereby declare that except where specific reference is made to the work of others, that all content of this Graduate Paper are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This Graduate Project is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text.

Namratha Pashupathy Manjula Devi

May, 2017
Acknowledgements

I would like to thank my advisor Professor Mark A. Indovina for his support, guidance, feedback, and encouragement which helped in the successful completion of my graduate research. Special thanks to my colleagues Krunal Mange and Thiago Pinheiro Felix da Silva e Lima for their work, corrections, comments and their active participation in this work. Finally, thanks to Dr. Dorin Patru for providing access to student implementations of the various RISC processors discussed in this paper.
Abstract

The Verification methodology of modern processor designs is an enormous challenge. As processor design complexity increases, an elaborate and sophisticated verification environment has to be employed to identify, assist in debug, and document design bugs. This paper presents a configurable verification environment for RISC processors. The verification environment is developed in SystemVerilog, an IEEE standard that bridges the gap between design and verification by delivering a single language and environment for both. The verification environment will validate the performance of the RISC processors with a micro-architectural model developed in SystemC. The system also comprises of an intelligent Instruction Generator that generates random sequences of instructions. All verification system components are configured using a common RISC processor architecture configuration control file.
Contents

1 Introduction 1
   1.1 Research Goals .............................................. 3
   1.2 Contributions .............................................. 4
   1.3 Organization ................................................ 4

2 Bibliographical Research 6

3 Verification Environment 14
   3.1 Random Instruction Generator ............................ 16
   3.2 Reconfigurable Reference Model ............................ 16
   3.3 Test bench .................................................. 17

4 Processor Architecture 19
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Processor Overview</td>
<td>19</td>
</tr>
<tr>
<td>4.2</td>
<td>Register file (RF)</td>
<td>20</td>
</tr>
<tr>
<td>4.3</td>
<td>Memory Unit (MU)</td>
<td>21</td>
</tr>
<tr>
<td>4.4</td>
<td>Functional Unit (FU)</td>
<td>23</td>
</tr>
<tr>
<td>4.5</td>
<td>I/O Peripheral Unit (IOPU)</td>
<td>23</td>
</tr>
<tr>
<td>4.6</td>
<td>Processor Operation</td>
<td>23</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Data Manipulation Instructions</td>
<td>23</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Data Transfer Instructions</td>
<td>24</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Flow control Instructions</td>
<td>25</td>
</tr>
<tr>
<td>4.6.4</td>
<td>Pipeline Theory</td>
<td>27</td>
</tr>
<tr>
<td>5</td>
<td>Configuration File Specifications</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>Test Methodology</td>
<td>36</td>
</tr>
<tr>
<td>6.1</td>
<td>Test Bench Generator</td>
<td>37</td>
</tr>
<tr>
<td>6.2</td>
<td>Interface</td>
<td>38</td>
</tr>
<tr>
<td>6.3</td>
<td>Driver</td>
<td>39</td>
</tr>
<tr>
<td>6.4</td>
<td>Environment &amp; Compilation Unit</td>
<td>40</td>
</tr>
<tr>
<td>6.5</td>
<td>Test Case</td>
<td>41</td>
</tr>
<tr>
<td>6.6</td>
<td>Test Model</td>
<td>41</td>
</tr>
<tr>
<td>6.7</td>
<td>Testing Harvard Processors</td>
<td>43</td>
</tr>
<tr>
<td>6.8</td>
<td>Testing Von Neumann Processors</td>
<td>44</td>
</tr>
<tr>
<td>7</td>
<td>Result and Discussion</td>
<td>45</td>
</tr>
<tr>
<td>7.1</td>
<td>Modes of operation</td>
<td>46</td>
</tr>
<tr>
<td>7.2</td>
<td>Test Simulation</td>
<td>47</td>
</tr>
</tbody>
</table>
7.2.1 Tests before SR correction ($T1$) ........................................ 47
7.2.2 Tests after SR correction ($T2$) ........................................ 57

8 Conclusion ................................................................. 67
  8.1 Future Work .......................................................... 69

References .................................................................. 71

I RISC Processor Instructions ........................................... 75
  I.1 Data Manipulation Instructions ...................................... 75
  I.2 Data Transfer Instructions ........................................... 79
  I.3 Flow Control Instructions .......................................... 83

II Configuration File ....................................................... 95

III List of Processors ...................................................... 101
  III.1 Processors tested ................................................... 101
  III.2 Flags affected ....................................................... 102

IV Resultant graphs ...................................................... 103
  IV.1 Processor $axt$ ...................................................... 103
  IV.2 Processor $dnm$ ..................................................... 119
  IV.3 Processor $tfl$ ...................................................... 135
  IV.4 Processor $sxs$ ..................................................... 151
  IV.5 Processor $vsk$ ..................................................... 167

V Source Code ............................................................. 183
  V.1 Test bench generator ................................................ 183
V.2 Interface ......................................................... 264
V.3 Driver .............................................................. 268
V.4 Environment ..................................................... 275
V.5 Compilation Unit ............................................... 276
V.6 Test case .......................................................... 278
V.7 Test ................................................................. 279
List of Figures

3.1 Verification Environment System Block Diagram .......................... 15
4.1 Data path block diagram .......................................................... 21
4.2 Memory organization for (a) Von Neumann and (b) Harvard processors .... 22
4.3 Instruction word format for manipulation instructions (a) 12-bit and (b) 14-bit processor ................................................................. 24
4.4 Instruction word format for data transfer instructions (a) 12-bit and (b) 14-bit processor ................................................................. 25
4.5 Instruction word format for JUMP and CALL (a) 12-bit and (b) 14-bit processor 25
4.6 Instruction word format for RET (a) 12-bit and (b) 14-bit processor ........ 26
4.7 Sample Program file to demonstrate pipeline implementation .................. 29
4.8 Pipeline stages ......................................................................... 29
6.1 Test bench components ............................................................. 37
6.2 Command to generate test bench .................................................. 38
6.3 Interface extending into DUT and test bench ................................. 39
7.1 Instruction and Error count for test T1 (mode A) on processor nxp ............ 49
7.2 Classification of Errors found in processor nxp while executing test T1 (mode A) . 50
List of Figures

7.3 Instruction and Error count for test $T1$ (mode $M$) on processor $nxp$ . . . . . . . . 51
7.4 Classification of Errors found in processor $nxp$ while executing test $T1$ (mode $M$) 52
7.5 Instruction and Error count for test $T1$ (mode $MB$) on processor $nxp$ . . . . . . 53
7.6 Classification of Errors found in processor $nxp$ while executing test $T1$ (mode $MB$) 54
7.7 Instruction and Error count for test $T1$ (mode $MD$) on processor $nxp$ . . . . . . 55
7.8 Classification of Errors found in processor $nxp$ while executing test $T1$ (mode $MD$) 56
7.9 Instruction and Error count for test $T2$ (mode $A$) on processor $nxp$ . . . . . . . . 58
7.10 Classification of Errors found in processor $nxp$ while executing test $T2$ (mode $A$) . 59
7.11 Instruction and Error count for test $T2$ (mode $M$) on processor $nxp$ . . . . . . . 60
7.12 Classification of Errors found in processor $nxp$ while executing test $T2$ (mode $M$) 61
7.13 Instruction and Error count for test $T2$ (mode $MB$) on processor $nxp$ . . . . . . . 62
7.14 Classification of Errors found in processor $nxp$ while executing test $T2$ (mode $MB$) 63
7.15 Instruction and Error count for test $T2$ (mode $MD$) on processor $nxp$ . . . . . . . 64
7.16 Classification of Errors found in processor $nxp$ while executing test $T2$ (mode $MD$) 65

IV.1 Instruction and Error count for test $T1$ (mode $A$) on processor $axt$ . . . . . . . . . 103
IV.2 Classification of Errors found in processor $axt$ while executing test $T1$ (mode $A$) . 104
IV.3 Instruction and Error count for test $T1$ (mode $M$) on processor $axt$ . . . . . . . . 105
IV.4 Classification of Errors found in processor $axt$ while executing test $T1$ (mode $M$) 106
IV.5 Instruction and Error count for test $T1$ (mode $MB$) on processor $axt$ . . . . . . . . 107
IV.6 Classification of Errors found in processor $axt$ while executing test $T1$ (mode $MB$) 108
IV.7 Instruction and Error count for test $T1$ (mode $MD$) on processor $axt$ . . . . . . . . 109
IV.8 Classification of Errors found in processor $axt$ while executing test $T1$ (mode $MD$) 110
IV.9 Instruction and Error count for test $T2$ (mode $A$) on processor $axt$ . . . . . . . . . 111
IV.10 Classification of Errors found in processor $axt$ while executing test $T2$ (mode $A$) . 112
IV.32 Classification of Errors found in processor *dnm* while executing test *T2* (mode MD) ................................................................. 134
IV.33 Instruction and Error count for test *T1* (mode A) on processor *tfl* ................................................................. 135
IV.34 Classification of Errors found in processor *tfl* while executing test *T1* (mode A) ......................................................... 136
IV.35 Instruction and Error count for test *T1* (mode M) on processor *tfl* ................................................................. 137
IV.36 Classification of Errors found in processor *tfl* while executing test *T1* (mode M) ......................................................... 138
IV.37 Instruction and Error count for test *T1* (mode MB) on processor *tfl* ................................................................. 139
IV.38 Classification of Errors found in processor *tfl* while executing test *T1* (mode MB) 140
IV.39 Instruction and Error count for test *T1* (mode MD) on processor *tfl* ................................................................. 141
IV.40 Classification of Errors found in processor *tfl* while executing test *T1* (mode MD) ......................................................... 142
IV.41 Instruction and Error count for test *T2* (mode A) on processor *tfl* ................................................................. 143
IV.42 Classification of Errors found in processor *tfl* while executing test *T2* (mode A) ......................................................... 144
IV.43 Instruction and Error count for test *T2* (mode M) on processor *tfl* ................................................................. 145
IV.44 Classification of Errors found in processor *tfl* while executing test *T2* (mode M) ......................................................... 146
IV.45 Instruction and Error count for test *T2* (mode MB) on processor *tfl* ................................................................. 147
IV.46 Classification of Errors found in processor *tfl* while executing test *T2* (mode MB) ......................................................... 148
IV.47 Instruction and Error count for test *T2* (mode MD) on processor *tfl* ................................................................. 149
IV.48 Classification of Errors found in processor *tfl* while executing test *T2* (mode MD) ......................................................... 150
IV.49 Instruction and Error count for test *T1* (mode A) on processor *sxs* ................................................................. 151
IV.50 Classification of Errors found in processor *sxs* while executing test *T1* (mode A) ......................................................... 152
IV.51 Instruction and Error count for test *T1* (mode M) on processor *sxs* ................................................................. 153
IV.52 Classification of Errors found in processor *sxs* while executing test *T1* (mode M) ......................................................... 154
IV.53 Instruction and Error count for test *T1* (mode MB) on processor *sxs* ................................................................. 155
IV.54 Classification of Errors found in processor *sxs* while executing test *T1* (mode MB) ......................................................... 156
IV.55 Instruction and Error count for test *T1* (mode MD) on processor *sxs* ................................................................. 157
List of Figures

IV.56 Classification of Errors found in processor ssx while executing test $T1$ (mode MD) 158
IV.57 Instruction and Error count for test $T2$ (mode A) on processor ssx . . . . . . . . 159
IV.58 Classification of Errors found in processor ssx while executing test $T2$ (mode A) . 160
IV.59 Instruction and Error count for test $T2$ (mode M) on processor ssx . . . . . . . . 161
IV.60 Classification of Errors found in processor ssx while executing test $T2$ (mode M) 162
IV.61 Instruction and Error count for test $T2$ (mode MB) on processor ssx . . . . . . . . 163
IV.62 Classification of Errors found in processor ssx while executing test $T2$ (mode MB) 164
IV.63 Instruction and Error count for test $T2$ (mode MD) on processor ssx . . . . . . . . 165
IV.64 Classification of Errors found in processor ssx while executing test $T2$ (mode MD) 166
IV.65 Instruction and Error count for test $T1$ (mode A) on processor vxk . . . . . . . . 167
IV.66 Classification of Errors found in processor vxk while executing test $T1$ (mode A) 168
IV.67 Instruction and Error count for test $T1$ (mode M) on processor vxk . . . . . . . . 169
IV.68 Classification of Errors found in processor vxk while executing test $T1$ (mode M) 170
IV.69 Instruction and Error count for test $T1$ (mode MB) on processor vxk . . . . . . . . 171
IV.70 Classification of Errors found in processor vxk while executing test $T1$ (mode MB) 172
IV.71 Instruction and Error count for test $T1$ (mode MD) on processor vxk . . . . . . . . 173
IV.72 Classification of Errors found in processor vxk while executing test $T1$ (mode MD) 174
IV.73 Instruction and Error count for test $T2$ (mode A) on processor vxk . . . . . . . . 175
IV.74 Classification of Errors found in processor vxk while executing test $T2$ (mode A) 176
IV.75 Instruction and Error count for test $T2$ (mode M) on processor vxk . . . . . . . . 177
IV.76 Classification of Errors found in processor vxk while executing test $T2$ (mode M) 178
IV.77 Instruction and Error count for test $T2$ (mode MB) on processor vxk . . . . . . . . 179
IV.78 Classification of Errors found in processor vxk while executing test $T2$ (mode MB) 180
IV.79 Instruction and Error count for test $T2$ (mode MD) on processor vxk . . . . . . . . 181
IV.80 Classification of Errors found in processor vxk while executing test $T2$ (mode MD) 182
List of Tables

7.1 Classification of Errors found in processor nxp while executing test T1 (mode A) . 66

I.1 Data Manipulation Instructions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
I.2 Data Transfer Instructions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
I.3 Flow Control Instructions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
Forward

The paper describes a configurable RISC processor test bench and verification environment generator developed as part of a larger Graduate Research project called Project Heliosphere. The overarching goal of Project Heliosphere is to develop a robust, configurable, verification and validation environment to further the study of various RISC processor architectures. The initial phase of this project was undertaken by Krunal Mange (configurable RISC processor random instruction generator), Namratha Pashupathy Manjula Devi (configurable RISC processor test bench and verification environment generator), and Thiago Pinheiro Felix da Silva e Lima (configurable RISC processor model). Indeed I am proud, and humbled by the research work produced by this group of students.

Mark A. Indovina

Rochester, NY USA

10 May 2017
Chapter 1

Introduction

The Verilog Hardware Description Language (HDL) originated at Gateway Design Automation in 1985. With the success of Verilog, Gateway Design Automation was acquired by the Cadence Design Systems in 1989. Through the late 90s, the Verilog Hardware Description Language became the popular HDL used to describe hardware for simulation and synthesis [1]. The first two versions of Verilog, released in 1995 and 2001, were subsequently standardized by the IEEE and had only simple features for creating tests. As design complexity grew, verification issues arose and commercial hardware verification languages like OpenVera and e were created. As an open language, SystemVerilog was started by Accellera and was adopted as an IEEE standard in 2005.

The unification of design and verification tools has given rise to SystemVerilog, a Hardware Description and Verification Language (HDVL) that combines features of HDL such as Verilog and VHDL with features from Hardware Verification Languages, together with features from C and C++. This has led to the improvement in communication between the design and the verification engineers aiding the verification process as the design, verification components, and assertion constructs are all present in the same environment. Traditional direct testing method can
be used to exploit verification environment features given adequate time and resources. However, as the design complexity grows, the time required to verify the design considering all possible input combinations increases and reaching 100% coverage for all the tests could be a laborious process. As an alternative, random tests are adopted for thoroughly exercising complex designs.

A key concept of any modern verification methodology is the layered test bench [1]. Although it can take a considerable amount of time to build a layered test-bench, it helps in dividing the task into smaller sections that can be developed independently. Functional Verification is developed starting with a verification plan which is derived from the hardware specifications and system requirements, and contains a description of what aspects of the design has to verified and what techniques has to be used. The next main task is to create constrained random tests by generating stimulus specific to a test in the verification plan. Functional coverage measures the success of all the tests in the verification plan and helps in evaluating the performance of the design by reporting the percentage of possible stimuli tested.

To satisfy high computational needs, modern processors are made faster, more flexible, and are designed to deliver high performance. The quests for higher performance have produced complex processor designs and elaborate verification methodologies. Random stimulus is essential for complex designs. Although there may be a significant delay before the first test can be run, random testing proves to be a better preference as satisfactory coverage is obtained much faster than in a direct test. Every random test shares a common verification environment resulting in a single constrained-random test bench that finds bugs faster than many directed tests.

In the proposed project, a configurable verification environment is developed that is used for the validation of the range of RISC processors with variable instruction set architectures as provided in one of the digital systems classes in RIT. The verification environment will validate the operation of the RISC processors with a micro-architecture model of a RISC processor implemented in SystemC. The verification environment will provide all the necessary control
1.1 Research Goals

The goal of this research framework is to build a fully viable test bench that validates the RISC processors with the help of the Model and the random Instruction Generator. The goal is achieved with the following objectives:

- To understand the system requirements, the processor architecture, to lay a foundation for the test bench components and to establish a connection between the test components and processor designs.

- To analyze the timing relations between the model and processor for the true comparison of results.

- To develop a system to collect the comparison results and represent them graphically.

A total of eight different processor designs are being testing in this work. The design specifications vary for every processor and they are broadly classified based on their instruction word and processor architecture. There are 12-bit/14-bit processors with Harvard/Von Neumann processor architectures. Two processors, a 12-bit Von Neumann and a 14-bit Harvard are picked from this set and all the significant errors in the implementation of instructions are detected and corrected to provide a standard for the rest of the processors.
1.2 Contributions

The major contributions to the project Heliosphere are as below:

1. A modular test bench is built by developing the test components like driver, interface, etc. in SystemVerilog.

2. Once the test bench framework is established, the test bench components for every processor are generated by the test bench generator written in Perl.

3. All the errors pertaining to the chosen two processors are detected and corrected which consists of implementation errors, register/flag assignment errors and dependency hazards.

4. The result for all the tests of a processor is gathered in a log file and the log file is worked on to derive essential information like the number of times an error occurred and the reason for the occurrence of errors. This information is graphically represented in the form of bar charts for all the processors.

1.3 Organization

The structure of the thesis is as follows:

- Chapter 2: This chapter consists majorly of articles/journals/books that are referred to provide a foundation for building a layered test bench. It also discusses some of the new methodologies and techniques for processor verification.

- Chapter 3: This chapter briefly describes the system architecture, the components and their connection.
• Chapter 4: The processor architecture, data path, theory of operation and processor pipelining are discussed in detail in this chapter.

• Chapter 5: This chapter outlines the structure of a configuration file and briefly describes the parameters used in the file.

• Chapter 6: The test methodology, test bench components and the testing of Harvard and Von Neumann processors are discussed in this chapter.

• Chapter 7: This chapter comprises of the resultant graphs and findings from the tests.

• Chapter 8: The conclusion and possible future work are briefly discussed in this chapter.
Chapter 2

Bibliographical Research

Verification is a major obstacle in creating the final product and it is the most important component of timely market release and success [2]. The main goal of verification is to make sure that the design meets the functional requirements as specified in the specification. There are different varieties of verification methodology options available within the industry [3]. Several articles, journals and books are researched to explore various verification methodologies and to determine the current state of knowledge of the topic under discussion—Verification of RISC processors. Many of them are used as references in this work as they provide ideas and techniques to create and build an efficient environment for processor verification.

[1] is one such reference material which is used extensively throughout the project. It describes the basic language constructs, features and use in detail. It includes several techniques and examples on how to build a basic layered test bench using Object Oriented Programming (OOP). SystemVerilog incorporates OOP, dynamic threads, and interprocess communication [1]. This is evident in the instantiation of several classes, interface in the high level test module which contains the DUT instantiation. SystemVerilog uses verification constructs, such as program blocks, interfaces, and clocking blocks to build a test bench and connect it with the DUT
The SystemVerilog has constrained-random stimulus generation where the user can pick the format of stimulus to be sent to DUT. This work of stimulus generation is done by the Instruction Generator as a part of another work in Perl as the stimulus is more design specific. This book elaborates on all the features of SystemVerilog language and allows the user to create test benches at higher level of abstraction than achieved with any HDL or a programming language such as C [1].

The verification process used in [4] closely relates to the work presented in this paper. It describes the SystemVerilog verification of a 32-bit RISC processor IP core using Open Verification Methodology (OVM). The test bench also has a behavioral model of the processor in it along with the random stimulus generator. The Design Under Test (DUT) of the system is a micro-processor with a 5-stage pipeline with a separate stage for memory data transfers. The processor also supports exceptions. The system verification is done based on three features: Assertion-based verification, Constrained random stimuli generation, and Functional coverage. Assertion based verification creates small blocks of code that contain statements about the expected behavior of the design code [4]. The constrained random stimuli generator allows the user to set some constraints on the stimuli in order to get a relevant stimuli than completely random values. Functional verification checks if certain features are missing in the design. OVM has a library for the user to create transactor-level modeling between test bench components and interfaces where the objects can be created at run time. The test bench consists of layers called Control, Analysis, Operational, Transactor and DUT layer. The driver in the transactor layer drives the DUT and puts a sequence item in predictor present in the analysis layer. The result from the DUT in the monitor is compared with that of the predictor in the scoreboard. The processor behavior is captured in the wave window and the instructions are verified successfully.

The design and verification of 16-bit Harvard RISC processor is discussed in [5]. This processor is designed to embed in Application Specific Integrated Circuits (ASIC) and System on
Chip (SOC) design by replacing the 8051 processor. The processor has a 5-stage pipeline and has functional blocks like ALU, multiplier, and barrel shifter. For program debugging the processor has an on-chip debug logic and the external interface uses JTAG protocol and internal interface uses scan chain [5]. A instruction set simulator is created as a reference model to the processor. The DUT is connected to the verification environment through the DUT ports. These ports/boundary signals are grouped into interfaces, with each port representing interrelated signals that collectively describe an interface protocol supported by the DUT [6]. The processor verification happens in three steps: The simulations of the processor is compared to that of the simulator. In the next step, complex algorithms like ADPCM Vocoder and SOLA (Synchronized Overlap and Add) are used by the simulator to check the pipeline architecture. In the third step, the processor is downloaded on the FPGA to HDL simulations. For the processor core execution test and on-chip debugging, GDB (GNU’s Debugger) server program is required which is listed in the future work of the paper. Another example for a functional simulator can be found in [7] where an existing verification framework for single-core processor is developed to support rapid verification of multi-thread program for multi-core processors.

Multicore processors now dominate the commercial marketplace with inter-core parallelism increasing the performance of computers. As the complexity of the multi-core processors increases with new architectures, they require sophisticated methodologies for verification. The verification methodology discussed in [8] is for heterogeneous (DSP+ARM) multicore processors for multicore SOC. Heterogenous Multicore SOCs consists of complex clusters of processors with multiple levels of caches. The SOC under discussion is the TI Keystone II SOC which integrates an ARM and DSP processing cores. SystemVerilog has a methodology for verifying integrated circuits called the Universal Verification Methodology (UVM). The UVM Class Library provides all the building blocks you need to quickly develop well-constructed, reusable, verification components and test environments [9]. Large number of test cases are written for
verification. Identifying “library functions” is an important step in this methodology just as it is in implementing an effective UVM like flow [8]. Each test has multiple phases like the initialization, configuration, pre-run, test run, post-run, results extraction, and test report phases to be performed in order to complete testing and reporting. Functions used by the processors are loaded in the functions library and is divided into sub-categories specific to the processor architecture. It has general purpose, DSP-specific, and ARM-specific sub-categories each containing code capable of running on any CPU, DSP platforms and ARM platforms respectively. Elements in the code library are used to create the test cases and is mentioned in a test control file which has sequential tasks to control the operation of the DUT. It also has manipulators and checkers whose primary task to observe and report the working of the DUT. The test contains configuration setup which is a group of files where a particular scenario can be setup for the processor and tested as different conditions when paired with test cases. The test is performed with valid combination of test cases and configuration producing a list of valid test combinations to be performed called the Test list. The verification is done by creating tests that target specific features of the DUT. The important reason for using UVM is that the library modules created in one test case can be reused in other test cases where it can customized to the parameter being verified in that test case. [10] is another work where UVM is applied to the verification of an application processor SoC. The proposed test bench can be applied to both IP and top level system. The automatic test generation helps in creating a well structured test bench architecture and reduces complexities with adopting UVM.

Paper [11] presents an example of using UVM in processor verification. High-level modeling method is preferred over traditional RTL model as high level modeling exploits the hardware and software co-design techniques to lessen the development period. High-level modeling method can be roughly divided into architecture model, instruction set model, transaction level model [12]. A transactor level model (TLM) based on UVM is created for the purpose of verifying
'SuperV_EF01 DSP'. The use of TLM is to obtain speed up in simulation and it is reported that TLM model is 20 times faster than the conventional RTL model. In TLM, the communication between modules is done by means of tasks or functions in contrary to descriptive coding in RTL method. Once the TLM of the processor is built, it can direct both RTL design and software development [11]. Both design and verification is presented in this paper and the first step is to design the architecture. The TLM for 'SuperV_EF01 DSP' is then built based on the partition of functional modules. The architecture of TLM has separate caches for data and instructions in the processor. Different phases (build, connect, run and final phase) of the TLM are executed in a top-down order. The main purpose of TLM is to abstract away communication among the modules by so-called transactions: the data or event between two components of a modeled and simulated system [13]. The processor operates in the following stages- instructions fetch, operand fetch, and execution stages in the build phase. All modules are instanced and memory space is given to the variables at the end of build phase. In the connect phase, the modules are connected and the design is linked to the verification system. The run phase starts the simulations and final phase concludes the simulations. The result statistics is obtained in the last phase. Performance analysis data like executive instruction numbers, cache miss rate, instruction level parallel is obtained. These results from the TLM model (reference model) are compared to that of DUT (RTL model) in the scoreboard to see if they match. The work described in [11] is closely related to the work in this paper as the verification framework is basically similar except that the former uses TLM (based on UVM) for the verification methodology.

Another technique to verify processor designs can be FPGA verification which has faster execution speeds making it cost effective, scalable and provides an efficient environment for finding design bugs better than traditional simulators. FPGA verification does not have design visibility and control and hence it becomes challenging to implement the FPGA for verifying a processor design. faster execution speeds enables large number of test cases to be run exploring
every possible stimuli combination which gives a better input coverage. The test case generators follow one of these two techniques: Functional correctness checking generators, consistency checking generators [14]. Functional correctness checking generators have a software reference model and the generated instructions are run on both the reference model and on the DUT and the results are compared. These generators are difficult to generate as the reference model needs to be updated every time the architecture of the DUT is updated. The generated instruction sequence is run on the hardware model several models to obtain a reference in case of consistency checking generators and are best suited for system-level verification and hence is used in this paper as the test case generator. They depend on the FPGA environment for debugging errors and to find a failure, it is crucial that the same events occur consistently. The FPGA implementation of the design differs significantly as compared to simulation and emulation environments [15]. For example, in multi-core processors, the cores can cause the failure to go away if the order of the transaction is changed and can be quite a challenge for FPGA stimulus verification. This is overcome by saving the state of failure for debugging future issues. To get full visibility and control over design, the failing test image can be ported to hardware emulators or logic analyzers which can help in finding the cause of the failure.

The Verification Methodology Manual (VMM) is a verification methodology that enables verification engineers to develop powerful, transaction-level environments for verifying complex designs using SystemVerilog. It contains a manual with rules and recommendation on how to construct and develop test benches. [16] discusses a viable verification platform based on VMM to verify the Fast Bus Interface (FBI) of the network processor (XDNP). There are two major busses in the FBI architecture of the network processor- Push and Pull bus. The 'Push bus' has queues that are aimed at reading FBI data and the 'Pull bus' aimed at writing data to FBI resources. The verification is implemented in three layers- Scenario, Functional and Command layer. The scenario layer are used to generate data to be sent to the driver via transactor. The
functional layer has the scoreboard and has two transaction processors [16] to check the response of the FBI. The purpose of transaction processors is to check the data in and out of the FBI in the scoreboard. By using the VMM, the defects in the FBI are detected and the coverage obtained proves that it is a much better methodology compared to orientation test methods. The work described in [17] is also along the same lines as that of the previous paper where the test bench to a NP (Network Processor) is developed to verify the design with functional coverage models and assertions. The NP is a type of multi-core SOC consisting of a master processor (StrongARM), six microprocessor engines (ME), a SRAM controller, a SDRAM controller and a fast bus interface unit (FBI) [17]. The test bench architecture consists of the generator, driver, monitor and checker. The generator generates stimulus which in this case is an array of random IP packets. The driver sends these packets to the driver which is sent to the DUT. The monitor gathers the output data from the DUT and sends it to the checker to compare it with the data from the functional reference model. This model has a NP micro code that predicts the response of the DUT. The checker compares the DUT and the reference model’s outputs and identifies the number of match and mismatched IP packets. Functional coverage is also done to know whether all ports are involved in sending and receiving packets. Hence, a test bench with functional coverage models and assertions can greatly improve the efficiency of functional verification.

Assertion-Based Verification (ABV) is assuming a significant role in the design validation flow of chip design companies [18]. SystemVerilog Assertions (SVA) are used to validate the behavior of a design and they can be checked by simulation or by a checker tool where certain assumptions are made about the system design. SVA also provides several built-in functions to test for certain design conditions and also provides constructs to collect functional coverage data automatically [19]. The idea of SystemVerilog using assertions sees its roots in software verification methodologies. SystemVerilog assertions use that idea to allow functional verification of the hardware systems [20]. There are two types of SVA- immediate and concurrent assertions.
and the latter is most widely used in functional verification of ASIC designs [21]. The verification methodology here uses the implemented library of Synthesizable SystemVerilog Assertions (SSVA) to implement and verify Open Control Protocol (OCP) monitors on a FPGA design. The structure of concurrent assertions has several layers to define SSVA library. The memory and register profiles of the OCP monitors and the SSVA library are implemented and verified.

With the development of SoC technology and the continuous increase of design complexity, complexity of the IC verification has greatly increased, verification has occupied about 70% of the entire IC development process [22]. To better the verification processor, a method called co-simulation method based on SystemC and SystemVerilog is employed. SystemC can realize transaction-level modeling and high-speed simulation for non-timed or loose timing system-level environment, while base on SystemVerilog you can build a standardized verification platform by UVM, OVM or other verification methods, which can complete verification work flexibly and comprehensively [23]. An interface based on DPI (Direct Programming Interface) mechanism is developed to combine the two languages and to solve the co-simulation problem between SystemC and SystemVerilog for SoC verification. The co-simulation interface presented in this work has smaller code size and increased execution speeds compared to UVM Connect library in [24]. The syntax equivalence is important to identify the co-simulation method between SystemC and SystemVerilog and the basic structuring and modeling of this is elaborated in [25]. The proposed interface sends random stimulus to the SystemC (test model) generate using UVM. Finally, the results of both verification platform and DUT are compared in the scoreboard. With the help of environment configuration unit, protocol conversion unit, stimulus import unit and Timing synchronization unit, the co-simulation between these two systems is successful yielding an interface with better flexibility, faster execution speed and higher portability.
Chapter 3

Verification Environment

The system developed for verification of 12 and 14 bit RISC processors comprises of three major components: the Random Instruction Generator (IG), Reference Model (Model), and Test Bench. The other important files in the system are the Configuration file and processor HDL files. Although the parts of the verification environment are designed in different languages, they are all linked together to form a single system. Figure 3.1 shows the block diagram of the system.

The configuration file is the main input file which has design specifications of all processors and other necessary information. Every processor has a unique configuration file and all the files are grouped under a folder named ‘configuration’. All the major system components, Model, IG and test bench generator extract required information from the configuration file like the name of the processor, architecture, bits, no. of registers, memory size, etc. The processor files are considered as DUTs (Design Under Test) for the system. Only the test bench has access to the processor files and acts like an interface between the DUT and the rest of the system. There are a total of eight 12 and 14 bit RISC processor samples in this course with varying design and architecture. The IG and Model both produce output files that are read by the test bench. There are two processor architecture types- Harvard and Von Neumann; the basic difference between
Figure 3.1: Verification Environment System Block Diagram
them is the type of memory used in each.

3.1 Random Instruction Generator

By extracting processor design specifications from their respective configuration file, the input instruction files namely the program and the data memory are generated by the IG as seen in Figure 3.1. The IG generates random sequences of instructions and is scripted in Perl v5.20. In case of the Harvard architecture, the IG generates two separate memory files whereas for the Von Neumann architecture, a single memory file is generated. The IG can also generate instruction of a particular kind like only the data manipulation instructions, branch instructions, data transfer instructions or any other valid combination can be created. Other salient features of the system include- performing special checks to make sure no illegal instruction is generated which are not part of the design specification and preventing exceptions like a ‘divide by zero’. Apart from generating random sequences of instructions, the IG also executes every instruction and writes the computed result in the respective register files.

3.2 Reconfigurable Reference Model

The reconfigurable processor reference model is implemented in SystemC and has similar properties as that of the hardware under test. It can emulate the working of a processor and its interaction with the memory units by obtaining the specifications from the configuration file. The Model gets the instruction file generated by the IG and computes the result of each instruction present in the program memory. The model then writes the resultant values in the respective register files. The Model also creates other output files like the program counter and the status register output files for every set of instructions. The Model will also check for invalid instructions and
instructions leading to an exception. The Model is built to be a reference for verification of the hardware. The output files created by the Model are read by the test bench and matched with the DUT.

### 3.3 Test bench

The test bench for verification of RISC processors is implemented in SystemVerilog. The environment consists of various modular components such as the Interface, Driver, Environment, Test case and the Test. The DUT is instantiated in the test file and the required input and output from the DUT is interfaced with test bench by means of ports. To model this, the Interface construct of SystemVerilog is used which connects the design blocks with the test bench. The test bench encapsulates the DUT and produces stimulus and captures responses as seen in Fig. All the necessary clock and control are provided by the test bench for the processor operation, particularly the Driver. The tools used for performing simulations are the Cadence Design Systems Incisive Suite. At every clock cycle, the register output values of the processor are collected in different arrays and are used later to match them with the Model and the IG. As every processor has unique design properties, the test bench is generated according to the type of processor indicated in the configuration file. Hence a Perl script was developed to generate the test bench components based on the target processor configuration. All the tasks in the test bench are sensitive to the clock. At every clock, the register arrays of the DUT are compared to that of the Model and the IG. If there is any mismatch, the simulation is stopped and the values of the erroneous instruction from the DUT, the correct values from Model and IG are displayed and stored in a file called the error log file for further analysis. Similar tests are run numerous times and the errors in each test are accumulated in this file. Once a set of tests are complete, the log file is used to interpret the cause and solution to the errors. In Figure 3.1, the error log file, represented as 'C'
and the opcode file 'O' is generated to keep track of the instructions executed through the test. The Error Decode/Report block works on these two files to produce files of '.t' and '.csv' format that are used to represent the results in terms of graphs.

The Instruction Generator and the Model are covered as part of another work. Only the work of generating the test bench and testing the system is discussed in this paper. The purpose of this collaborative work is to efficiently create a complex, robust environment for verification of RISC processors.
Chapter 4

Processor Architecture

The architecture of a processor gives a detailed view of the design and behavior of its basic components and operations. This research work is based on the processors designed in a course, Design of Computer Systems supervised by Dr. Dorin Patru at RIT in 2015. This course had both undergraduate and graduate students with different processor specifications given to each but only the graduate students’ processors are tested in this work. There are a total of eight processors that are considered as DUT in this work each having unique design specifications.

4.1 Processor Overview

The components of the processor architecture consists of the Memory Unit (MU), Functional Unit (FU), Register File (RF) and the I/O-Ps (Input/Output Peripheral) unit as seen in Figure 4.1. The figure represents the data path high level block diagram for the processor system. All processors are designed using the Altera Quartus IDE and the Register Transfer Level (RTL) code is either written in Verilog or VHDL according to the choice of the designer. The processors can be broadly classified based on their architecture as following:
• Harvard, two separate memory units i.e., Program and Data memory.

• Von Neumann, single unified memory unit for both Program and Data storage i.e., Main memory.

The processors can also be classified according to the instruction word size with each having a unique design specification as following:

• 12-bit processors with 6-bit opcode and two 3-bit operands (has 8 general purpose registers).

• 14-bit processors with 6-bit opcode and two 4-bit operands (has 16 general purpose registers).

• 16-bit processors with 6-bit opcode and two 5-bit operands (has 32 general purpose registers).

4.2 Register file (RF)

This unit consists of all the general purpose registers. Each register can be loaded from and outputted to any of the three internal buses (A, B and C) as in Figure 4.1. For the 12, 14 and 16-bit processors the registers are designated as R0-R7, R0-R15 and R0-R31 which totals into eight 12-bit, sixteen 14-bit and thirty-two 16-bit general purpose registers respectively. Only the 12 and 14-bit processors are considered in this work. The registers are reset at the beginning of the simulations and are used to hold computational values required for performing operations like ADD (Addition of two registers), AND (Logical AND between two registers), OR (Logical OR operation), MUL (Multiplication), ROTR (Rotate right), etc.
4.3 Memory Unit (MU)

The Memory Unit comprises of two important memory blocks: Program Memory (PM) and Data Memory (DM) in case of Harvard and only Main Memory (MM) in case of Von Neumann processors. It also comprises of significant registers such as the Program Counter (PC), Instruction Register (IR), Memory Address Base register (MAB), Memory Address Index register (MAX), Effective Memory Address register (MAeff).

In Harvard processors, PM is a ROM (Read-Only Memory) which primarily has all the instructions that are to be executed by the processor and DM is a RAM (Random Access Memory) that holds the data required for loading into or storing from registers respectively in data transfer operations like LOAD and STORE. For Von Neumann architecture processors, there is only one memory, MM, which is an RAM that is initialized at run time with program and data. The MM is divided into a program part of the memory that consists of the opcodes or the instructions to be
executed, and a data part of the memory for data transfer operations. A section of the data memory is also dedicated for I/O operations such as a region that forms the stack. The PC is a register that contains the address of the instruction currently executed and is also used to keep track of the order of the execution of the instructions. In conjunction with the PC, the instruction register (IR) holds the opcode for the current instruction that is being executed. MAeff is calculated by adding the MAB with the MAX which gives the effective address at which the processor should jump to (in case of JUMP and CALL) or load/store values from/to (in case of LOAD or STORE).

The detailed memory organization for Von Neumann and Harvard processors is shown in (a) and (b) of Figure 4.2 respectively. Both architectures have two busses each for address and data and the only difference is in the division of memory unit. A memory of size 00-FF is considered in the figure above to explain the memory division where the Von Neumann processor has a single memory of size FF while the Harvard processor has two memory units each of size FF, each with separate address and data busses for program and data memory.
4.4 Functional Unit (FU)

The functional unit comprises of temporary registers TA (Temporary A) and TB (Temporary B) at its input and it can be loaded with a constant or from any of the three internal busses. At the output side, there are three temporary registers TALUH, TALUL and TSR for holding the computational values temporarily before transferring it to the main registers.

4.5 I/O Peripheral Unit (IOPU)

The registers in this unit are used when input is taken from the outside environment. In this course, a DE0-Nano Development board (Cyclone IV FPGA) is used and the output is displayed on the 8 LEDs of the board. The registers available for these I/O operations are IPAD (Input Peripheral Address Decoder), OPAD (Output Peripheral Address Decoder) and SP (Stack pointer).

4.6 Processor Operation

All instructions generated in this work are listed in the appendix along with its description, syntax and usage. The instructions are classified broadly into three types as listed below.

4.6.1 Data Manipulation Instructions

Instructions that modify the data content in the resultant register are called data manipulation instructions. ADD, SUBC, NOT, XOR, etc., are some of examples for data manipulation instructions. Mathematical operations are performed such as addition, subtraction, multiplication, and division involving two registers or one register with a constant. Shift and rotate operations like shift right, shift left, rotate right, rotate left, rotate right through carry and rotate left through
4.6 Processor Operation

carry are performed. Logical operations such as XOR, NOT are implemented. COPY and SWAP instructions are also implemented where a register value is copied to another or two register values are swapped.

Figure 4.3: Instruction word format for manipulation instructions (a) 12-bit and (b) 14-bit processor

The instruction word (IW) format for data manipulation instructions are presented in Figure 4.3. The manipulation instructions have only one IW that is represented as IW0 in the figure. The first six bits of the IW0 is the opcode and the next bits are 3-bit operands \( R_i \) and \( R_j \) for the 12-bit processor and 4-bit operands for the 14-bit processor.

4.6.2 Data Transfer Instructions

Data Transfer instructions are based on register-memory transfer. LOAD and STORE are the two examples which transfer data from memory to registers and registers to memory respectively. There are two IW represented as IW0 and IW1 in Figure 4.4. The IW0 has the opcode and the operands \( R_i \) and \( R_j \). The IW1 is the address offset which is either 12 or 14-bit. The IW1 and the \( R_i \) together determine the location of the memory from or to which the data is loaded or stored respectively.
4.6 Processor Operation

4.6.3 Flow control Instructions

Flow control instructions, JUMP and CALL, affect the flow of the program. JUMP can be conditional or unconditional; depending on the condition evaluated, either the jump is either performed or the flow of the program is not affected. The CALL instruction initiates a call to a subroutine in the program and the RET is present at the end of the subroutine to go back to the instruction following the CALL in the main program. There are eight conditional JUMP instructions with respect to the status flags like Carry (C), Negative (N), Overflow (V) and Zero (Z) in the status register. The instruction word format for JUMP and CALL is shown in Figure 4.5. RET has only one instruction word IW0 as seen in Figure 4.6.

---

**Figure 4.4:** Instruction word format for data transfer instructions (a) 12-bit and (b) 14-bit processor

**Figure 4.5:** Instruction word format for JUMP and CALL (a) 12-bit and (b) 14-bit processor
The execution of all the instructions is carried out in four machine cycles (MC0, MC1, MC2, MC3): Instruction fetch (IF), Operand fetch (OPF), Execution (EXE) and Write back (WB). In MC0 the instruction to be executed is fetched from the memory and placed in the IR and the PC is incremented by 1 pointing to the next instruction. The instruction is decoded in MC2, where the operands are fetched and the respective operands are saved in the temporary registers TA and TB. In some instructions, like LOAD, STORE, CALL and JUMP, the next instruction is the offset which is the address in the memory at which the data has to be loaded from (in LOAD), stored to (in STORE), or at which the program has to jump to (in case of JUMP and CALL). In the same machine cycle MC1, this offset is stored in the MAB register. There are three addressing modes of operation in case of data transfer and flow control instructions which is indicated by Ri and is stored in MAX in the same cycle. A value of 0 in MAX indicates direct, 1 indicates PC-relative, 2 indicates SP-relative and from 3 onwards its register relative addressing mode. In MC2, the execution of the instruction takes place where the computational values are stored temporarily in TALUH and TALUL. In case of data transfer and flow control instructions, the MAeff is calculated by adding the obtained MAB with the MAX. This gives the effective address in the memory. The last cycle, MC3 is the write back cycle where the computed values in TALUH and TALUL are transferred to the respective registers. The jump conditions are evaluated in the last machine cycle and accordingly the PC is updated. In case of LOAD and STORE instructions, data is either loaded from memory into the target register or data is stored from the target register into memory as determined by the MAeff. This cycle repeats for every
instruction in the program memory. So, to execute one instruction, 4 clock cycles are required which is the same as saying- in one cycle, a quarter of the instruction is executed. Hence, to speed up the execution a concept called pipelining is employed.

### 4.6.4 Pipeline Theory

In pipelining, the execution of different machine cycles from different instructions are overlapped thus increasing the throughput. The presumption is that each of the different machine cycles will use different components of the processor. The maximum theoretical throughput cannot be achieved in practice due to dependencies. A dependency is defined as the relation between the instructions and/or their operands in a program. Dependencies can lead to hazards if left unresolved and can result in an error at runtime. There are two solutions to this problem, the first one is to ‘stall’ the pipeline where later stages (for e.g. IF of next instruction) stop while previous ones (for e.g. WB of current instruction) complete processing. The second and the most efficient solution to this problem is ‘data forwarding’ where the registers involved in the dependency are detected and it can be ‘forwarded’ to the other instruction that needs it without waiting until the WB stage. There are three possible kinds of dependencies which can be classified as- Data, Control and Structural Dependencies.

- **Structural Dependency** arises when two or more instructions try to access same part of the processor during the same machine cycle.

- **Data Dependency** arises between data of the instructions in the flow of operation. There are three types of data dependencies-
  - **RAW**, Read After Write, arises from flow dependency where the current instruction needs the data generated by the previous instruction.
– **WAR, Write After Read**, arises from an anti-dependance where the instruction writes a new value over the one that is still required by the previous instruction.

– **WAW, Write After Write**, arises from an output dependency where the two instructions writing into the same register does not happen in the correct order that they were issued.

- Control Hazards arises when conditional instructions like JUMP start evaluating the condition before the completion of the previous instruction that determines if the jump should be taken or not.

The pipeline implementation of a program sample is presented below. The processor is four stage pipelined with instruction fetch happening constantly at every stage. Simple data manipulation instructions like SUB, ADDC, ADD and SUBC are used to describe the data dependencies/hazards and the possible solutions to overcome them. The speedup of the overall system increases as the pipeline implementation uses less number of machine cycles than the non-pipeline implementation. Figure 4.7 shows the sample list of instructions and the effect it has on the registers after each cycle. Initially, the two registers used in this example are cleared by doing a SUB operation followed by other manipulation instructions.
4.6 Processor Operation

<table>
<thead>
<tr>
<th></th>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SUB R5, R5;</td>
<td>R5=0;</td>
</tr>
<tr>
<td>2</td>
<td>SUB R6, R6;</td>
<td>R6=0;</td>
</tr>
<tr>
<td>3</td>
<td>ADDC R5, 0x1;</td>
<td>R5=0x1=+1d;</td>
</tr>
<tr>
<td>4</td>
<td>SUBC R6, 0x2;</td>
<td>R6=0xFFFFE=-2d;</td>
</tr>
<tr>
<td>5</td>
<td>ADD R6, R5;</td>
<td>R6=0xFFFF=-1d;</td>
</tr>
<tr>
<td>6</td>
<td>SUB R5, R6;</td>
<td>R5=0x2=+2d;</td>
</tr>
</tbody>
</table>

Figure 4.7: Sample Program file to demonstrate pipeline implementation

<table>
<thead>
<tr>
<th>MC</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>P-IC</td>
<td>SUB-IC</td>
<td>SUB-IC</td>
<td>ADDC-IC</td>
<td>SUBC-IC</td>
<td>ADD-IC</td>
<td>SUB-IC</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>P-MC0</td>
<td>SUB-IF</td>
<td>SUB-IF</td>
<td>ADDC-IF</td>
<td>SUBC-IF</td>
<td>ADD-IF</td>
<td>SUB-IF</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>P-MC1</td>
<td>-</td>
<td>SUB-OPF</td>
<td>SUB-OPF</td>
<td>ADDC-OPF</td>
<td>SUBC-OPF</td>
<td>ADD-OPF</td>
<td>SUB-OPF</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>P-MC2</td>
<td>-</td>
<td>-</td>
<td>SUB-EXE</td>
<td>SUB-EXE</td>
<td>ADDC-EXE</td>
<td>SUBC-EXE</td>
<td>ADD-EXE</td>
<td>SUB-EXE</td>
<td>-</td>
</tr>
<tr>
<td>P-MC3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>SUB-WB</td>
<td>SUB-WB</td>
<td>ADDC-WB</td>
<td>SUBC-WB</td>
<td>ADD-WB</td>
<td>SUB-WB</td>
</tr>
<tr>
<td>D/H</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>RAW</td>
<td>RAW</td>
<td>RAW</td>
<td>RAW</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Technique</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>PIB</td>
<td>PIB</td>
<td>FUIB, PIB</td>
<td>FUIB</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 4.8: Pipeline stages

Three cycles are used to fill up the pipeline in the beginning and three more cycles in the end to empty the pipeline. Every column consists of different machine cycles of different instructions that are happening simultaneously which can lead to structural or data dependency that results in a hazard if not resolved. A concept of data forwarding is employed to overcome data dependency.
hazards where the data is forwarded via the internal busses without waiting till the WB stage.

In 4.8, the pipeline starts to fill by performing instruction fetch of the first SUB; in the second machine cycle (MC), the first SUB which is at OPF and the second SUB which is at IF are executing simultaneously. The IF, OPF, EXE and WB stages are performed for each instruction in a similar fashion. In the fourth MC, The ADDC requires the latest value in register R0 to add a constant 1 to it but the register value in R0 has not been written by the first SUB yet (the first SUB is still in the WB stage and the value will be written in the register after this MC). The register at this stage does not have the latest value. The value to be written to the register R0 is present in the temporary register and is forwarded via the Processor Internal Buses (PIB) to the OPF of ADDC. This is one of the data forwarding techniques to avoid hazards and is also an example of RAW dependency. Similar dependency can also be noticed in the fifth MC whereas in the sixth MC, the ADD instruction requires R1 (from the EXE stage of SUBC) and R0 (from the WB stage of ADDC). As earlier the value to be written to R0 is forwarded via the PIB and the value to be written to R1 is forwarded via the Functional Unit Internal Buses (FUIB) as it is still in the execution stage. Both data forwarding techniques PIB and FUIB can be observed in this cycle with a RAW dependency. Control dependencies/hazards can be resolved with the help of a branch predictor or by stalling the pipeline. As the branch predictor concept is out of scope of this course, to overcome control dependency hazards, the instruction fetch of the next instruction is stalled until the JUMP condition is evaluated. Similarly, in case of CALL and RET instructions, the memory unit is used for storing and retrieving the SR and PC value of the instruction to which the CALL subroutine has to return to after the subroutine is complete. Hence, CALL and RET are stalled as the memory is in use for all the four machine cycles and the next instruction fetch cannot simultaneously. In case of the LOAD and STORE instructions, the pipeline can be stalled depending on the processor architecture. As the Harvard processor
uses two different memories for data and program, the pipeline need not be stalled and the data transfer can be performed without any dependencies. In case of the Von Neumann processor, the pipeline has to be stalled as it has a single memory for both data and program. Here, the memory is being used for performing load and store operations and hence cannot be used for the next instruction fetch. The speedup of the example program can be calculated as total non-pipelined cycles divided by the total pipelined cycles which is \[ \frac{(6 \text{ instructions}) \times (4 \text{ cycles each})}{9} = \frac{24}{9} = 2.66 \] and the ideal speedup is 4.
A special file called the configuration file is created for each processor, it contains all the necessary processor design specifications and other essential information required by the verification system (Model, Test Bench Generator, Instruction Generator). The file consists of the following parameters and its requirement in the file is justified below as used by test bench:

- **name_folder**: This is the folder where the main processor file is found. All Verilog files, including those for the processor, memory, and other units if any are placed in their respective folders and collectively placed in the `src` folder of the test bench.

- **name**: indicates the name of the Processor Verilog file and is particularly required by the test environment during instantiation of the DUT (Design Under Test) in the test bench. It is also required for creating log files when the final tests are run to document the results of the respective processor.

- **bits**: this is the instruction word size of the processor and it is required by all the system parts as it is one of the most important parameters of a processor. It can be either 12 (`0xC`) or 14-bit (`0xE`).
• **registers**: denotes the total number of general purpose registers in the architecture and is required by the test bench generator while generating the interface where the registers widths are declared. It can be either 8 (0x8) or 16 (0x10).

• **architecture**: one of the most important parameters for generating the test bench as the format for testing each processor architecture is different. A ’0’ implies its a Harvard architecture and a ’1’ implies its Von Neumann.

• **opcode_size**: denotes the number of bits reserved for the opcode and is generally 6 bits for all processors in this course.

• **operand1_size** and **operand2_size**: denotes the size of the two operands.
  
  - 12-bit: Opcode Size: 6; Operand1: 3; Operand2: 3;
  - 14-bit: Opcode Size: 6; Operand1: 4; Operand2: 4;
  - For flow control instructions, the operand size is given by \((\text{bits}) - (4 + \text{opcode\_size})\).

  Also, the addition of **operand1_size**, **operand2_size** and **opcode_size** should be equal to **bits**.

• **dm_size**: This is the actual size of the memory used in power of 2. \((2^{\text{dm\_size}})\).

• **memory_size**: This is the memory size available for program memory and data memory each in power of 2. In case of Von Neumann, it is the total available size of the initialized RAM.

• **pc_in_pc_relative**: In PC relative addressing mode of the data transfer/control instructions it is important to know what PC value the DUT is considering. If the DUT is considering the current instruction then this parameter is set to 0. If the PC is incremented and it points
to the next instruction then this parameter is set to 1. This is mostly used by the model and the IG to generate the correct results according to how the DUT is designed.

- **SP, Stack_direction and Stack_size:** SP denotes the top of the stack, Stack_direction determines the growth of the stack. It is set to 0 if the stack grows from lower to higher memory address and is set to 1 if it grows from higher to lower memory address. Stack_size is the amount of space occupied by the stack as the percentage of memory. The following addresses are used for the Stack Pointer Top:
  
  - For 12-bit processor: 0x1FEF = 0x1FF-0x10 (Reserved for I/Os)
  - For 14-bit processor: 0x3FEF = 0x3FF-0x10 (Reserved for I/Os)

- **start_mapping and end_mapping:** This represents the boundary in which all the instructions used are declared. The right hand side is the standard and reserved mnemonic used to represent the set of instructions and the left hand side represents the user mnemonic in the processor design. This parameter avoids conflict by mapping different representation of instruction mnemonic into one standard representation.

- **opcode:** Lists all three types of instructions with their respective opcodes. They are divided into data transfer, data manipulation and flow control instructions. Each type has start and end notations which is later used by the IG to generate selective instruction sets that contain only manipulation, only manipulation and branch or only manipulation and data transfer instructions according to the user’s choice.

- **clk_st:** This parameter is extensively used by the test bench generator and is the number of the clock cycles after which the first output is obtained from the processor. The test bench waits for these number of cycles before the simulation starts.
• *name_pm* and *name_dm*: These are the names of the memory files used by the processor. They are required as they have to be listed in the order of their use in the compilation unit that is a group of source files that are compiled together in the src folder of test bench.

• *del_ld* and *del_st*: Denotes if the data transfer instructions are stalled or not. These parameters are set to 1 if the data transfer instructions are stalled and 0 if they are not. Typically, in the Harvard architecture, the LOAD and STORE instructions need not be stalled as there are no structural dependancies related to memory usage but in Von Neumann architecture, due to the presence of single memory, these instructions are stalled. The flow control instructions have to be stalled inevitably in both the architectures. This parameter is required to know the offset between the Model output and the DUT output for a true comparison.

An example of a configuration file (processor *nxp*) is provided in Appendix II.
Chapter 6

Test Methodology

A testbench is a network of verification components designed to check whether the RTL implementation meets the design specification or not [26]. The general steps to verify a design are: To generate inputs, reset DUT, configure the DUT, run the test, capture outputs, verify the correctness, report the errors and provide possible solutions. The test bench is split in layers and components to solve the complexity of DUT and verification systems and also to maintain and reuse the codes [26]. The test bench layers are Signal layer, Command layer, Functional layer and the Scenario layer. The signal layer consists of the DUT and the command layer has the driver. The functional layer is un-used as the checking is done in a separate module. The scenario layer is also un-used as the process of stimulus generation is done by the Instruction Generator outside the test bench. The test bench wraps around the DUT and along with the Model mimics the processor environment. The first step to verification is connecting the test bench and the design. A separate entity is used called the interface to communicate between the test bench and the design. The next steps involve generating stimulus, driving the DUT and capturing the outputs. By doing so, a generalized test bench framework is obtained and the test bench can be customized according to each processor with the help of a Test Bench Generator.
6.1 Test Bench Generator

In Verification infrastructure, the test bench components are encapsulated and instantiated hierarchically and are controlled through an extendable set of phases to initialize, run, and complete each test [27]. A block diagram of the test bench components is shown in Fig 6.1.

**Figure 6.1: Test bench components**

Generating SystemVerilog test bench components automatically can be time saving and gives the user an organized methodology for working with the test bench framework. As all the processors are either 12-bit or 14-bit, the Test Bench Generator is fairly easy to code. The Test Bench Generator reads the input configuration file for design specifications like no. of bits, architecture, no. of registers and stores it in variables. The `Getopt::Long` module is used to implement an extended `getopt` function where it takes arguments from the command line.

The name of the configuration file (`config`), the number of instructions to be printed prior
to the instruction at error (*len*) are the parameters taken from the command line. Help is also included to provide the user with the correct usage of command line options. The default name for configuration file is taken as *configuration.txt*. The correct syntax for generating the test bench is shown in 6.2. Using the Perl regular expressions, the desired parameters are searched for, matched and stored in scalar variables from the configuration file. Four output files are created by the Test Bench Generator: interface, driver, test, and compilation unit (*risc.sv.rtl.f*).

### 6.2 Interface

The Interface construct in SystemVerilog is an intelligent bundle of wires that contain connectivity, synchronization, and optionally, the functionality of the communication between two or more blocks [1]. The interface consists of all the input and output ports from the DUT. The input ports of DUT are only the *Clock_pin* and *Resetn_pin* and the rest of the ports in Figure 6.3 are output ports of DUT.

The above figure shows how the interface extends into the two blocks, synchronizes and connects the design block and the test bench. The example interface shown in the figure is with respect to a 14-bit processor. The clock is a part of the interface. The *logic* type is used in declaring the signals so it can be driven from procedural statements. The interface is instantiated in the top module and the signals are referred in the test components hierarchically using the instance name as *intf.clk*, *intf.reset*, etc. One of the main advantages of using interface is that when a new signal is added it can be declared only once in the interface and can be accessed.
by higher level modules with the right reference path. Apart from the DUT signals, three set of arrays are declared in the interface. The test bench reads the Model, IG and DUT’s output register values into these arrays. Additional set of arrays are created for every DUT signal (PC, SP, SR, IR1, IR2, IR3, DM_in, DM_out, PM_out) to accumulate their values at every cycle. Here, IR1, IR2 and IR3 are the three instruction registers used in pipelining. PM_out is data out of program memory and is only present in Harvard processors. DM_in and DM_out are data in and out of data memory respectively in both architectures.

6.3 Driver

The bottom layer of the test bench is called the signal layer and consists of DUT and the signals that connect it to the test bench. The layer above this is the command layer. The DUT’s inputs namely Clock_pin and Resetn_pin are driven by the Driver present in the command layer. This layer also has monitor and assertions which are not used in this work. The driver is a class
and a class consists of routines that act on the variables. An instance or an object (\textit{ intf }) of the interface (\textit{ intf_risc }) is created in the driver with type \textit{ virtual}. The use of a virtual interface is to allow objects in a test bench to refer to signals in interface object created using a handle rather than referring it by the actual name. The task of resetting the processor, opening and reading the configuration file and checking the end of program memory file to stop simulations are performed by the driver in this test bench. The instruction set occupies a part of the memory file and the rest is filled with 1’s (F’s in hex). Hence, to find the end of a program memory file, the driver looks for two consecutive F’s (FFF in 12-bit and 3FFF in 14-bit processors) and when this combination is found, the simulations are stopped indicating the end of the program. It also makes sure the previous instruction to the consecutive F’s is not a RET as it would mean the end of a call routine and not the end of the actual program.

\section*{6.4 Environment & Compilation Unit}

The environment is a main class that encapsulates sub-classes like monitor, scoreboard and the driver. Here, as monitor and scoreboards are not used, the environment only consists of the driver. An object (\textit{ drvr }) of the class driver is created in the environment. SystemVerilog also consists of the compilation unit which is a group of source files that are compiled together. Anything declared in this unit is considered global as it can be see by all lower-level blocks. All the test components used by the top level test are listed in the order of their instantiation. As interface is a primary SystemVerilog construct and is instantiated in all the other test bench components, it is listed first followed by the driver, environment, test module, supporting files of the processor namely, memory files, multiplier, divider, counter, processor Verilog file, etc. The test module is instantiated in the end as it instantiates the DUT, test case and interface. The Altera component libraries are included in the same file for the components used by the processor.
6.5 Test Case

The test case is a file of type program and instantiates the environment in it. A program block is similar to a module where it can have code and variables and be instantiated in other modules but in itself cannot have instantiation of modules, interfaces, or other programs [1]. In Verilog, simulations end after all the events are complete or until a $finish is executed. SystemVerilog considers the program block as a test module and the simulation ends when the last statement in every initial block in the program is completed. It can also have a final block that is executed just before the simulation ends. The test case is responsible for the flow of the operation in the top module as all the tasks from respective classes are called here. The tasks are referenced through the environment and are called with the right reference path for e.g., the task ’check’ of the driver is referenced as env.drvr.check(); Three tasks from the driver are referenced in this test case and a $finish in the task ’check’ will end the simulation. A delay of 1000 time units is introduced before simulation is completely stopped. This is done to complete the last instruction in the program memory.

6.6 Test Model

This is the top model of the test bench that comprises of the clock generator, instances of the interface, DUT, and the test case. The clock generator should be in the test module so it can drive both the DUT and the test bench. The test bench events are all clock sensitive and all the signals are driven on the clock’s edge. The DUT is instantiated (as top) in the test module through port mapping. The type of mapping used is the named port mapping where the user does not have to remember the order in which the ports are listed in the DUT, but only the names. The Harvard processors have DM_in (data in) and DM_out (data out) for the data memory and PM_out for the program memory. The Von Neumann processors have only DM_in and DM_out.
The memory component `altsyncram` of the processor has an array `mem_data` each for RAM and ROM that holds the data for the respective memory. The memory files created by the IG is read directly into these arrays using `$readmemh` and is made available to the processor. The program memory is also read into another array for the purpose of use in the test module. The memory files generated are used by IG and Model and after execution of all the operations, they produce output register files available in the respective folders. These files containing general purpose register values, SR values, PC values are read into arrays declared in the interface.

At the positive edge of every clock, the DUT’s output register values namely the registers, SR, PC, IR (IR1, IR2, IR3), DM_in, DM_out and PM_out are read into the respective arrays declared in the interface. A special file `R.t` is created initially to print the cycle by cycle values of the DUT’s output. The obtained IG and Model values are also printed alongside for comparison. Another file `PC.t` is created to keep track of the flow of the program and `opcode.t` for all the opcodes executed in a set of program files in a test. This file is later used for reporting the density of instructions in each test in the report. A very important log file `cplog.t` is created that is an improvised version of the `R.t` file where all the output values of DUT, IG and Model are present and is cleared after every test. This file is very crucial for reporting the errors and the possible cause of them. The structure of the test model for all processors is more or less the same but the major difference is seen between Harvard and Von Neumann processors where the instructions are checked for stalls. The test model has the same structure for all Harvard processors with only changes in the width of the registers depending on if its a 12-bit or a 14-bit. Similarly, the test model is the same for all Von Neumann processors with difference only in the register widths.
6.7 Testing Harvard Processors

As Harvard processors have two separate memory unit for program and data, most of the instructions do not require stalling. Only the flow control instructions (JUMP, CALL and RET) are stalled in the pipeline as these instructions interact with the memory and it cannot be used for the next IF. At every cycle, the processor outputs are read into the arrays but they are not considered during the stalling cycles as the data does not change and is not required. Hence, a flag is set to skip through these stall cycles and the outputs during the stall are not printed in the \texttt{R.t} file. The outputs of Model and IG during instruction LOAD are obtained a cycle before the DUT and hence the indices of comparison have to be adjusted accordingly to match them with the DUT. The output of the processor is obtained 4 clock cycles after its instruction fetch (for e.g. the output of the 2nd instruction is obtained after the 6th clock cycle, output of 3rd instruction is obtained after the 7th cycle). Since the DUT signals are read into arrays at every cycle, the PC and the output obtained at every cycle is not relative to the same instruction. Hence, the PC in the test module has to be adjusted so as to print the outputs next to the correct value of PC in the \texttt{R.t} file. Conditions like two recurring data transfer/flow control instructions, or the occurrence of a data manipulation instruction between two data transfer/flow control instructions in the program memory file are detected and the PC in the test is adjusted to get the actual PC value of the DUT. For all other conditions, the actual PC is calculated as PC-4 in the test.

The DUT outputs are compared with the outputs of both Model and IG and are printed in the \texttt{R.t} file with a ‘*’ next to those outputs that do not match with that of Model or IG. As mentioned earlier, the file \texttt{R.t} is improvised and renamed \texttt{cplog.t} which contains the output values of the instruction at error and few instructions prior to it to detect the possible cause of the error. The number of instructions to be printed in the file prior to the erroneous instructions is determined by ‘\texttt{w}’ and is obtained from the command line while generating the test bench.
Whenever there is a mismatch, a task ‘print_con’ is called which prints the output values of the erroneous instruction and ‘w’ instructions prior to it in cplog.t. The outputs of DUT, IG and Model are matched again in this task and it is also made sure that none of the output has a don’t-care (‘x’) value. This is done by performing reduction XOR on the register bits and if it is equal to ‘x’, it is considered as an undefined register. The Status Register bits are explicitly printed with the respective status flags- Carry (C), Negative (N), Zero (Z) and Overflow (V). The vital information of the error instructions like time, instruction name, opcode and the location of instruction in memory (PC) are printed in cplog.t file.

6.8 Testing Von Neumann Processors

As Von Neumann processors have a single memory unit for both program and data, the Data transfer and the Flow control instructions are stalled as the memory is used by these instructions and cannot be used for the next instruction fetch. The top instantiation of the DUT will have only DM_in and DM_out from the memory along with other signals like reset, clk, PC, SR, etc. The generated memory file will be read into the ’mem_data’ array of the altsyncram component of the RAM. There is no ROM component here as the data memory is an initialized RAM. Similar conditions like two recurring data transfer/flow control instructions, or the occurrence of a data manipulation instruction between two data transfer/flow control instructions in the memory file are detected and the PC in the test is adjusted accordingly. The only difference while testing Von Neumann processors is that the instructions LOAD and STORE are also considered for skipping the stall cycles along with the flow control instructions. Apart from the differences mentioned above, the remaining tasks are performed in the same fashion as that of Harvard processor.
Chapter 7

Result and Discussion

The results and findings are presented in this chapter. The following tasks are performed after simulation to obtain the results, process the various items collected, and represent the resulting data graphically:

• The result from DUT, Model, and IG along with timing information and other necessary details of the instructions are obtained in cplog.t file.

• The opcode values of all the instructions executed in a test are obtained in opcode.t file.

• The required information is extracted from cplog.t and opcode.t by the Error Decode/Report block.

• The extracted information is analyzed by the Error Decode/Report block for the number of occurrences of the instructions in the test, the total number of errors for each instruction and probable cause of the errors.

• The values are printed in respective files in the respective folder. Files of format .csv are also created for the purpose of exporting the values into spreadsheets.
• Using MATLAB, two bar graphs are plotted for every test. One graph depicts the error count and classification of errors for every instruction while the other gives the density of instructions in the test.

### 7.1 Modes of operation

The simulation is performed in four modes of operation:

1. Data manipulation instructions only (mode $M$)
2. Data manipulation and Data transfer instructions (mode $MD$)
3. Data manipulation and Flow control instructions or (mode $MB$)
4. Data manipulation, Data transfer and Flow control instructions or (mode $A$)

From the set of eight processors being tested, two processors, a 12-bit Von Neumann and a 14-bit Harvard are chosen and numerous tests are performed, resulting errors with respect to flag assignments, implementation and dependencies are detected and corrected. These two processors are considered as standards for matching and evaluating the rest of the processors. A set of 1500 tests are performed on the remaining six processors, with each test executing about 1000 instructions. The test is performed in two sets Test 1 ($T1$) and Test 2 ($T2$) with four different operating modes, $M$, $MD$, $MB$ and $A$, each executing 1500 tests. After the completion of each test, cplog.t is generated which contains the output values and necessary information about the errors. This file is cleared when a new test is begun. The errors can be classified as: Dependency, Status flag (SF), Program Counter (PC), Implementation errors and Undefined register(s). Status Flag error occurs due to incorrect implementation of status flags in the processor resulting in a mismatch. Errors related to PC occur when there is a miscalculation in the effective addresses.
7.2 Test Simulation

The tests are performed in two sets, Test 1 ($T_1$) and Test 2 ($T_2$). When the first test $T_1$ is run, it is observed that there are numerous errors pertaining to the Status Register (SR) or Status Flag (SF) for various instructions which masks other critical issues like implementation errors and dependencies. One of the many reasons to these errors can be due to uninitialized SR at reset. Hence, the SR issues are solved in all the processors by initializing them to zero at reset so that there are no undefined flags. Only a few instructions affect the flags in the SR in the standard processors and hence all the other processors are also checked if the flags are implemented in the same fashion as the standard processors. The test is run again after the above mentioned modifications in $T_2$. Hence, the set of tests performed before and after the SR correction are named as $T_1$ and $T_2$ respectively.

7.2.1 Tests before SR correction ($T_1$)

- Mode A

The instruction density and the total number of errors for every instruction executed in $T_1$ (mode A) for the processor $nxp$ is shown in Fig 7.1. Mode A consists of Data manipulation, Data transfer and Flow control instructions. It can be observed from the figure that the instruction NOT and JUMP have the most errors in this particular test and LOAD has the...
least errors as compared to other instructions. The possible reasons for the occurrence of these errors for every instruction are provided in the following figure. The errors are classified as Implementation error, PC error, Undefined register and Status flag error. As mentioned earlier, majority of the errors in every instruction is due to the SF and is evident in Fig 7.2. Also, for instruction JUMP, the PC error dominates over the rest and is due to miscalculation of jump address and can also be also referred as an implementation error in the instruction JUMP. As these errors dominate majority of the tests in $T_1$ they have to be solved in order to view other significant errors. There is a possibility that these errors are also caused due to a dependency issue (Ri or Rj dependency) along with the other errors. Table 7.1 gives a better understanding of the nature of the errors for all the instructions. SF, PC, UR and IE from the table stand for Status flag, Program Counter, Undefined Registers and Implementation errors respectively. From the table it can be observed that the reason for increased PC errors in JUMP can be due to a Ri-dependency and hence it can be concluded that the data forwarding in JUMP is incorrect.
Figure 7.1: Instruction and Error count for test $T1$ (mode $A$) on processor $nxp$
7.2 Test Simulation

Figure 7.2: Classification of Errors found in processor \textit{nxp} while executing test \textit{T1} (mode \textit{A})
• Mode $M$

The instruction density and the total number of errors for every instruction executed in $T1$ (mode $A$) for the processor $nxp$ is shown in Fig 7.3. Mode $M$ consists of only Data manipulation instructions and hence the instruction count for the data transfer (LOAD and STORE) and flow control instructions (JUMP, CALL and RET) is zero. As expected, the SF error count is the highest and is evident in Fig 7.4. Test $T1$ for mode $M$ contain fewer errors compared to mode $A$ as complex instructions (data transfer and flow control) are omitted. From the figure, XOR, OR, SUBC, AND, ADDC and ADD have zero errors with NOT having the maximum errors. The following figure indicates the reason for maximum errors in NOT is due to SF errors and few implementation errors in DIVC and MULC. Note that these figures are created from the table which is generated for every test.

Figure 7.3: Instruction and Error count for test $T1$ (mode $M$) on processor $nxp$
Figure 7.4: Classification of Errors found in processor nxp while executing test $T1$ (mode $M$)
7.2 Test Simulation

- Mode *MB*

This mode consists of only data manipulation along with flow control instructions JUMP and CALL. It can be observed from Figure 7.5 that both NOT and JUMP have the maximum errors compared to other instructions. The errors in NOT are due to SF errors and the errors in JUMP are due to PC errors indicating that there is miscalculation of jump addresses in most places. It can also be inferred from Figure 7.6 that DIVC and MULC have few implementation errors which can either be due to dependencies or incorrect assignments in the instruction cycles.

![Figure 7.5: Instruction and Error count for test T1 (mode MB) on processor nxp](image)
Figure 7.6: Classification of Errors found in processor *nxp* while executing test *T1* (mode *MB*)
• **Mode MD**

This mode consists of data manipulation along with data transfer instructions LOAD and STORE. As there are no flow control instructions, PC errors are less likely to occur unless there are PC errors related to increment / decrement in manipulation and data transfer instructions which will lead to a PC mismatch. Thus it can be concluded from Figure 7.7 that all errors are only present in manipulation instructions as error count for LOAD and STORE is zero. The majority of errors in Figure 7.8 are due to SF and implementation errors. There are zero errors related to PC and very few errors due to undefined registers.

![Figure 7.7: Instruction and Error count for test T1 (mode MD) on processor nxp](image-url)
7.2 Test Simulation

Figure 7.8: Classification of Errors found in processor nxp while executing test T1 (mode MD)
7.2.2 Tests after SR correction ($T2$)

- Mode $A$

   The instruction density and the total number of errors for every instruction executed in $T2$ (mode $A$) for the processor $nxp$ is shown in Fig 7.9. In this test, a reduction in the total number of errors can be observed as most of the insignificant SF errors are detected and corrected. The focus is now drawn to critical errors like Implementation errors which are predominant in this test. All types of instructions are executed in this mode and only a handful of SF errors can be seen in Fig 7.10 as compared to all the modes in test $T1$. A high PC error for JUMP indicates the miscalculation of jump address which is also seen in mode $A$ of test $T1$. The error count for the rest of the errors are all less than 100 which is much lesser than what is observed in test $T1$. Thus, it can be concluded that test $T2$ has significant reduction in the total number of errors compared to $T1$. In order to view other errors in the test mode, the PC issue in JUMP needs to be corrected.
Figure 7.9: Instruction and Error count for test $T2$ (mode $A$) on processor $nxp$
7.2 Test Simulation

Figure 7.10: Classification of Errors found in processor *nxp* while executing test *T2* (mode *A*)
• Mode $M$

The instruction density and the total number of errors for every instruction executed in $T2$ (mode $M$) for the processor $nxp$ is shown in Fig 7.11. The JUMP issue mentioned above is not present here as this test mode executes only manipulation instructions and hence it becomes easier to view the implementation errors for the rest of the instructions. As the error count for all instructions are less compared to their number of occurrences in the test, it can be said that the instructions are erroneous at only a few places. This can be due to the order of the instructions leading to a dependency. The data forwarding does not happen correctly if the dependency is not solved which explains the implementation errors in Figure 7.12 in this test mode.

![Figure 7.11: Instruction and Error count for test $T2$ (mode $M$) on processor $nxp$]
Figure 7.12: Classification of Errors found in processor *nxp* while executing test *T2* (mode *M*)
• Mode \textit{MB}

The instruction density and the total number of errors for every instruction executed in \textit{T2 (mode MB)} for the processor \textit{nxp} is shown in Fig 7.13. The result in this test mode looks similar to that of mode \textit{A}. The highest number of errors in this test mode is due to the PC errors in JUMP as seen from Figure 7.14. Once it is cleared, the remaining errors can be viewed at large.

Figure 7.13: Instruction and Error count for test \textit{T2 (mode MB)} on processor \textit{nxp}
Figure 7.14: Classification of Errors found in processor *nxp* while executing test *T2* (mode *MB*)
• Mode MD

The instruction density and the total number of errors for every instruction executed in T2 (mode MD) for the processor nxp is shown in Fig 7.15. The result in this test mode is similar to that of mode M. It can be noticed from Figure 7.16 that there are fewer number of errors compared to test modes A & MB and mostly has only implementation errors. This can be due to unsolved dependencies or incorrect implementation of the instruction itself.

Figure 7.15: Instruction and Error count for test T2 (mode MD) on processor nxp
Figure 7.16: Classification of Errors found in processor nxp while executing test T2 (mode MD)
Table 7.1: Classification of Errors found in processor *nxp* while executing test *T1* (mode *A*)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Total Occurrence</th>
<th>Errors</th>
<th>Ri-dependency</th>
<th>Rj-dependency</th>
<th>SF</th>
<th>PC</th>
<th>UR</th>
<th>IE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>306</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ADDC</td>
<td>302</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>AND</td>
<td>291</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CALL</td>
<td>275</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>COPY</td>
<td>281</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DIV</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>DIVC</td>
<td>290</td>
<td>31</td>
<td>1</td>
<td>0</td>
<td>24</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>JUMP</td>
<td>244</td>
<td>180</td>
<td>149</td>
<td>0</td>
<td>0</td>
<td>180</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LOAD</td>
<td>240</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>MUL</td>
<td>302</td>
<td>34</td>
<td>5</td>
<td>0</td>
<td>33</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>MULC</td>
<td>293</td>
<td>32</td>
<td>1</td>
<td>0</td>
<td>23</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>NOT</td>
<td>309</td>
<td>197</td>
<td>12</td>
<td>0</td>
<td>196</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>OR</td>
<td>298</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>RET</td>
<td>11</td>
<td>8</td>
<td>6</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ROTL</td>
<td>311</td>
<td>126</td>
<td>1</td>
<td>0</td>
<td>124</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>ROTR</td>
<td>291</td>
<td>114</td>
<td>1</td>
<td>0</td>
<td>113</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>RTLC</td>
<td>289</td>
<td>124</td>
<td>2</td>
<td>0</td>
<td>123</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>RTRC</td>
<td>301</td>
<td>114</td>
<td>0</td>
<td>0</td>
<td>113</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>SHLA</td>
<td>268</td>
<td>115</td>
<td>3</td>
<td>0</td>
<td>114</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>SHLL</td>
<td>320</td>
<td>133</td>
<td>1</td>
<td>0</td>
<td>131</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SHRA</td>
<td>277</td>
<td>117</td>
<td>2</td>
<td>0</td>
<td>115</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>SHRL</td>
<td>291</td>
<td>109</td>
<td>1</td>
<td>0</td>
<td>107</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>STORE</td>
<td>291</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SUB</td>
<td>315</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>SUBC</td>
<td>285</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>SWAP</td>
<td>298</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>XOR</td>
<td>352</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Chapter 8

Conclusion

A configurable SystemVerilog verification environment for 12 and 14 bit RISC processors is developed in this work, and this verification environment is used for the validation of RISC processors with variable instruction set architectures. All verification system components are configured using an input configuration control file and automatically generated by a Test Bench Generator developed in Perl. The verification environment extensively validates the operation of eight RISC processors with the micro-architecture Model of the RISC processor implemented in SystemC, and an intelligent Instruction Generator designed in Perl which generates random instruction sequences. The verification environment provides a robust control and monitoring environment to validate the cycle by cycle operation of the processor and aid in debug in case of failures.

The test bench can be configured based on the following processor characteristics: Processor Architecture (Harvard or Von Neumann), instruction word size (12 or 14-bit), number of registers (8 or 16 registers), and type of instructions. The instruction set is classified into three types: Data manipulation, Data transfer, and Flow control instructions. The Interface, Driver, Environment, Test case, and main Test are the major test bench components used to build the test bench
framework. The Interface connects the processors (DUT) with the test bench, the Driver is used to drive the DUT’s input signals, the Environment encapsulates the Driver and Test case and is responsible for the flow of operation in the top module. The Test module is the top module that comprises of the clock generator, instances of DUT, interface and the test case. It consists of all the tasks required to perform the processor verification.

The Configuration file provides necessary information for the instruction generator to generate instruction sequences. The instructions and data are generated in two memory files: Program and Data memory. The memory files are provided as inputs to the DUT by the test bench. The instruction sequences are also given as inputs to both the Model and the DUT. The Model and DUT produce output values in the respective registers according to the instructions. Concurrently, the Instruction generator also computes the output values. The DUT output values are compared to both the Model and Instruction generator. If the values are found to be dissimilar, it is regarded as an error and is displayed at the end of the test and also represented in the form of a bar graph with errors classified as SF errors, PC errors, Implementation errors and Undefined registers.

As part of the verification process, two processors, each with a unique design, are picked from the processor set and all the errors in them are rectified so as to provide a standard for testing the rest of the processors. Numerous tests are performed on the remaining six processors and are validated based on these two standard processors. The tests are performed in two sets T1 and T2 with four different operating modes, each carrying out 1500 tests with each test executing about 1000 instructions. When there is a mismatch between the DUT and Model or Instruction Generator, the simulation stops and the output values are obtained in a file for the purpose of debugging. The total number of occurrences of each instruction in the test and the error count for every instruction are tabulated in a file and then graphically represented in the form of a bar chart.

The cause of the errors is predicted and the classification of errors is represented in a 3D bar
8.1 Future Work

The research presented in this paper has a lot of scope for future work. Some of the possible ideas that can be developed are presented below:

- Verification of RISC processors with higher bit sizes e.g., 16-bit, 32-bit, or even 64-bit datapath can be implemented with minimal modifications in the verification system design as bit size is configurable.

- In this work, only Verilog processors are verified. A processor designed in VHDL was omitted from the processor set as the component libraries could not be integrated. The verification process can also be extended to VHDL processors as a future work.
• The processor set used in this work is further built to form multi-core processors with unique cache designs. As a future work, multi-core processors with caches can be verified.

• Instruction Set can also be extended to implement SIMD (Single instruction Multiple Data) instructions.

• The error count graphs/table gives a brief understanding of the nature of the errors. Dependency is a major issue along with the mentioned four type of errors. While one can be sure of SF, PC, IE and UR errors, dependancy errors are probable and the user does not really know if a dependency occurred or not. Hence, future work can include elaborate reporting of dependency issues for a comprehensive debug.

• Functional Coverage can be done to create the widest possible range of stimuli with all sort of instruction combinations.
References


[27] Clifford E Cummings and Tom Fitzpatrick. Ovm & uvm techniques for terminating tests, 2011.
### Appendix I

#### RISC Processor Instructions

## I.1 Data Manipulation Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Operation</th>
<th>Syntax</th>
<th>Example</th>
<th>Machine Code for 12 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Addition of two registers</td>
<td>Ri = Rj + Ri</td>
<td>ADD &lt;Ri&gt;, &lt;Rj&gt;;</td>
<td>ADD R6, R7;</td>
<td>&lt;6-bit opcode for ADD&gt; 110 111</td>
</tr>
<tr>
<td>ADDC</td>
<td>Addition of one register and one constant</td>
<td>Ri = Rj + CONSTANT at Rj</td>
<td>ADDC &lt;Ri&gt;, &lt;Constant at Rj field&gt;;</td>
<td>ADDC R2, 1;</td>
<td>&lt;6-bit opcode for ADDC&gt; 010 001</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Example Equation</td>
<td>Instruction Code</td>
<td>Example Code</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>------------------</td>
<td>-----------------</td>
<td>--------------</td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>Subtraction of two registers</td>
<td>$R_i = R_i - R_j$</td>
<td>SUB &lt;Ri&gt;, &lt;Rj&gt;;</td>
<td>SUB R6, R7; SUB&gt; 110 111</td>
<td></td>
</tr>
<tr>
<td>SUBC</td>
<td>Subtraction of one register and one constant</td>
<td>$R_i = R_i - \text{CONSTANT at } R_j$</td>
<td>SUBC &lt;Ri&gt;, &lt;Constant at Rj field&gt;;</td>
<td>SUBC R2, 1; SUBC&gt; 010 001</td>
<td></td>
</tr>
<tr>
<td>MUL</td>
<td>Multiplication of two registers</td>
<td>$R_i = R_j \times R_i$</td>
<td>MUL &lt;Ri&gt;, &lt;Rj&gt;;</td>
<td>MUL R6, R7; MUL&gt; 110 111</td>
<td></td>
</tr>
<tr>
<td>MULC</td>
<td>Multiplication of one register by one constant</td>
<td>$R_i = R_i \times \text{CONSTANT at } R_j$</td>
<td>MULC &lt;Ri&gt;, &lt;Constant at Rj field&gt;;</td>
<td>MULC R2, 3; MULC&gt; 010 011</td>
<td></td>
</tr>
<tr>
<td>DIV</td>
<td>Division of one register by another register</td>
<td>$R_i = R_i / R_j$</td>
<td>DIV &lt;Ri&gt;, &lt;Rj&gt;;</td>
<td>DIV R6, R7; DIV&gt; 110 111</td>
<td></td>
</tr>
<tr>
<td>DIVC</td>
<td>Division of one register by one constant</td>
<td>$R_i = R_i / \text{CONSTANT at } R_j$</td>
<td>DIVC &lt;Ri&gt;, &lt;Constant at Rj field&gt;;</td>
<td>DIVC R2, 3; DIVC&gt; 010 011</td>
<td></td>
</tr>
<tr>
<td>NOT</td>
<td>Logical Negation of one register</td>
<td>Ri = NOT Ri</td>
<td>NOT &lt;Ri&gt;</td>
<td>NOT R7;</td>
<td>&lt;code for NOT&gt; 111 XXX</td>
</tr>
<tr>
<td>--------</td>
<td>---------------------------------</td>
<td>-------------</td>
<td>---------</td>
<td>--------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>XOR</td>
<td>Logical XOR (Exclusive OR) of two registers</td>
<td>Ri = Ri XOR Rj</td>
<td>XOR &lt;Ri&gt;, &lt;Rj&gt;;</td>
<td>XOR R7, R6;</td>
<td>&lt;code for XOR&gt; 111 110</td>
</tr>
<tr>
<td>SHLL</td>
<td>Shift Left Logic of a register by a constant</td>
<td>Ri = Ri &lt;&lt; Constant</td>
<td>SHLL &lt;Ri&gt;, &lt;Constant at Rj field&gt;;</td>
<td>SHLL R7, 0x2;</td>
<td>&lt;code for SHLL&gt; 111 010</td>
</tr>
<tr>
<td>SHRL</td>
<td>Shift Right Logic of a register by a constant</td>
<td>Ri = Ri &gt;&gt; Constant</td>
<td>SHRL &lt;Ri&gt;, &lt;Constant at Rj field&gt;;</td>
<td>SHRL R1, 0x5;</td>
<td>&lt;code for SHRL&gt; 001 101</td>
</tr>
<tr>
<td>SHLA</td>
<td>Shift Left Arithmetic of a register by a constant.</td>
<td>Ri = Ri &lt;&lt; Constant</td>
<td>SHLA &lt;Ri&gt;, &lt;Constant at Rj field&gt;;</td>
<td>SHLA R2, 0x4;</td>
<td>&lt;code for SHLA&gt; 010 100</td>
</tr>
<tr>
<td>SHRA</td>
<td>Shift Right Arithmetic of a register by a constant.</td>
<td>Ri = Ri &gt;&gt; Constant</td>
<td>SHRA &lt;Ri&gt;, &lt;Constant at Rj field&gt;;</td>
<td>SHRA R3, 0x4;</td>
<td>&lt;code for SHRA&gt; 011 100</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Register Value</td>
<td>Immediate Value</td>
<td>Immediate Value</td>
<td>Code</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>----------------</td>
<td>-----------------</td>
<td>-----------------</td>
<td>------</td>
</tr>
<tr>
<td>ROTL</td>
<td>Rotate Left the value of a register by a constant</td>
<td>Ri = Ri ROTL Constant</td>
<td>ROTL &lt;Ri&gt;, &lt;Constant at Rj field&gt;</td>
<td>ROTL R2, 0x4;</td>
<td>&lt;code for ROTL&gt; 010 100</td>
</tr>
<tr>
<td>ROTR</td>
<td>Rotate Right the value of a register by a constant</td>
<td>Ri = Ri ROTR Constant</td>
<td>ROTR &lt;Ri&gt;, &lt;Constant at Rj field&gt;</td>
<td>ROTR R6, 0x2;</td>
<td>&lt;code for ROTR&gt; 110 010</td>
</tr>
<tr>
<td>RTLC</td>
<td>Rotate Left one register through carry by a constant</td>
<td>Ri = Ri RTLC Constant</td>
<td>RTLC &lt;Ri&gt;, &lt;Constant at Rj field&gt;</td>
<td>RTLC R2, 0x1;</td>
<td>&lt;code for RTLC&gt; 010 001</td>
</tr>
<tr>
<td>RTRC</td>
<td>Rotate Right one register through carry by a constant</td>
<td>Ri = Ri RTRC Constant</td>
<td>RTRC &lt;Ri&gt;, &lt;Constant at Rj field&gt;</td>
<td>RTRC R6, 0x7;</td>
<td>&lt;code for RTRC&gt; 110 111</td>
</tr>
<tr>
<td>SWAP</td>
<td>Swap the contents of two registers</td>
<td>Ri = Rj and Rj=Ri</td>
<td>SWAP &lt;Ri&gt;, &lt;Rj&gt;;</td>
<td>SWAP R2, R3;</td>
<td>&lt;code for SWAP&gt; 010 011</td>
</tr>
<tr>
<td>COPY</td>
<td>Copy from register into another</td>
<td>Ri receives Rj value;</td>
<td>CPY &lt;Ri&gt;, &lt;Rj&gt;;</td>
<td>CPY R5, R7;</td>
<td>&lt;code for CPY&gt; 101 111</td>
</tr>
</tbody>
</table>
### 1.2 Data Transfer Instructions

**Table 1.2: Data Transfer Instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Operation</th>
<th>Syntax</th>
<th>Example</th>
<th>Machine Code for 12 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>STORE</strong> - Direct Address Mode</td>
<td>Memory Address specified by this function receives the value of a register</td>
<td>Memory Address = R[Rj]</td>
<td>STORE R0, M[&lt;Rj&gt;, Address];</td>
<td>STORE R0, M[R4, 0x800];</td>
<td><strong>Inst 1:</strong>&lt;code for STORE&gt;000 010&lt;br&gt;**Inst 2:**0100 0000 0000</td>
</tr>
<tr>
<td><strong>LOAD</strong> - Direct Address Mode</td>
<td>One Register receives the value of a Memory Address specified by this function</td>
<td>R[Rj] = Data of the Address</td>
<td>LOAD R0, M[&lt;Rj&gt;, Addr];</td>
<td>LOAD R0, M[R1, 0x860];</td>
<td><strong>Inst 1:</strong>&lt;code for LOAD&gt;000 001&lt;br&gt;**Inst 2:**0100 0000 0000</td>
</tr>
</tbody>
</table>
### STORE - Address + PC

At the Memory Address specified by this function plus the value of PC, stores the value of a register.

<table>
<thead>
<tr>
<th>STORE R1, M[Rj, Address];</th>
<th>STORE R1, M[R2, 0x405];</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inst 1:</strong></td>
<td><strong>Inst 2:</strong></td>
</tr>
<tr>
<td>&lt;code for STORE&gt;</td>
<td>001 010</td>
</tr>
<tr>
<td></td>
<td>0100 0000</td>
</tr>
<tr>
<td></td>
<td>0101</td>
</tr>
</tbody>
</table>

### LOAD - Address + PC

One Register receives the value of the sum of Memory Address and Program Counter, specified by this function.

<table>
<thead>
<tr>
<th>LOAD R1, M[&lt;Rj&gt;, Address];</th>
<th>LOAD R1, M[R7, 0x204];</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inst 1:</strong></td>
<td><strong>Inst 2:</strong></td>
</tr>
<tr>
<td>&lt;code for LOAD&gt;</td>
<td>001 111</td>
</tr>
<tr>
<td></td>
<td>0100 0000</td>
</tr>
<tr>
<td></td>
<td>0010</td>
</tr>
</tbody>
</table>
### I.2 Data Transfer Instructions

<table>
<thead>
<tr>
<th>STORE - Stack Pointer</th>
<th>At the Memory Address specified by the Stack Pointer Value, stores the value of a register</th>
<th>Memory Address referent to SP value = R[Rj]</th>
<th>STORE R2, M[R3, 0x700];</th>
<th>STORE R2, M[R3, 0x700];</th>
<th>Inst 1: &lt;code for STORE&gt; 010 101 0100 0000 0111</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD - Stack Pointer</td>
<td>One Register receives the value of a Memory Address specified by Program Counter</td>
<td>R[Rj] = Memory Address referent to Stack Pointer value</td>
<td>LOAD R2, M[R9, 0x600];</td>
<td>LOAD R2, M[R9, 0x600];</td>
<td>Inst 1: &lt;code for LOAD&gt; 001 110 0100 0000 0011</td>
</tr>
</tbody>
</table>
### 1.2 Data Transfer Instructions

**STORE** - Address + Value  
At the Memory Address specified by this function plus the value of a specific register, stores the value of a register.

| STORE - Address + Value | At the Memory Address specified by this function plus the value of a specific register, stores the value of a register | Memory Address + R[Ri] = R[Rj] | STORE <Ri>, M[<Rj>, Address]; [1] | ST R3, M[R2, 0x40F]; Inst 1: <code for STORE> 011 010  
Inst 2: 0100 0000 1111 |
|------------------------|-------------------------------------------------------------------------------------------------|---------------------------------|----------------------------------|----------------------------------|

**LOAD** - Address + Value  
One Specific Register receives the value of a Memory Address plus the value of one register specified by this function.

| LOAD - Address + Value | One Specific Register receives the value of a Memory Address plus the value of one register specified by this function | R[Rj] = Data of the ( Memory Address + R[Ri] ) | LD <Ri>, M[<Rj>, Address]; [2] | LOAD R3, M[R7, 0x4FF]; Inst 1: <code for LOAD> 011 111  
Inst 2: 0100 1111 1111 |
|-----------------------|-------------------------------------------------------------------------------------------------|-----------------------------------------------|---------------------------------|----------------------------------|

[1] Obs: Ri can be represented from registers R3 to R7 only for 12-bit processor. For 14-bit processor, from registers R3 to R15 only, and from registers R3 to R31 only for 16-bit processor.

[2] Obs: Ri can be a representation for registers from R3 to R7 only for 12-bit processor,
from registers R3 to R15 only for 14-bit processor and from registers R3 to R31 only for 16-bit processor.

I.3 Flow Control Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Operation</th>
<th>Syntax</th>
<th>Example</th>
<th>Machine Code for 12 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMPU - Jump Unconditional</td>
<td>Jump to a location. This location is determined by the current value of Ri. [3]</td>
<td>PC = New Address;</td>
<td>Code line 1: JMPU @jmp_1 ... Code line 2: @jmp_1 &lt;MNEMONIC&gt;</td>
<td>Code line 1: JMPU @jmp_2 ... Code line 2: @jmp_2 CSUBC R0, 0x1;</td>
<td>&lt;code for JMPU&gt; [xx]&gt;0000&gt; Inst 1: 0000 0011 1111 [4]</td>
</tr>
</tbody>
</table>
### I.3 Flow Control Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Code Line 1</th>
<th>Code Line 2</th>
<th>Instruction 1</th>
<th>Instruction 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMPC - Jump if Carry</td>
<td>Jump to a location determined by the address contained in this function if carry from status bit is set. [3]</td>
<td>Code line 1: JMPC <code>@jmp_1</code> ...</td>
<td>Code line 2: <code>@jmp_1</code> <code>&lt;MNEM&gt;</code></td>
<td>Inst 1: <code>&lt;code for JMPC&gt;</code></td>
<td>Inst 2: <code>0000 0000 1111</code> [5]</td>
</tr>
<tr>
<td>JMPN - Jump if Negative</td>
<td>Jump to a location determined by the address contained in this function if Negative bit from status bit is set. [3]</td>
<td>Code line 1: JMPN <code>@jmp_label</code> ...</td>
<td>Code line 2: <code>@jmp_label</code> <code>&lt;MNEM&gt;</code></td>
<td>Inst 1: <code>&lt;code for JMPN&gt;</code></td>
<td>Inst 2: <code>0000 0000 1111</code> [6]</td>
</tr>
</tbody>
</table>
### I.3 Flow Control Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Code Line 1</th>
<th>Code Line 2</th>
<th>Instruction 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>JMPV - Jump if Overflow</strong></td>
<td>Jump to a location determined by the address contained in this function if Overflow bit from status bit is set. [3]</td>
<td>Code line 1: JMPV @jmp_label ...</td>
<td>Code line 2: @jmp_label &lt;MNEM&gt;</td>
<td>Inst 1: &lt;code for JMPV&gt; &lt;xx&gt;&lt;0010&gt;</td>
</tr>
<tr>
<td><strong>JMPZ - Jump if Zero</strong></td>
<td>Jump to a location determined by the address contained in this function if Zero bit from status bit is set. [3]</td>
<td>Code line 1: JMPZ @jmp_label ...</td>
<td>Code line 2: @jmp_label &lt;MNEM&gt;</td>
<td>Inst 1: &lt;code for JMPZ&gt; &lt;xx&gt;&lt;0001&gt;</td>
</tr>
</tbody>
</table>
### I.3 Flow Control Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Code line 1:</th>
<th>Code line 2:</th>
<th>Inst 1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMPNC</td>
<td>Jump if NOT Carry: Jump to a location determined by the address contained in this function if Carry bit from status bit is zero and overflow, zero and negative bits are all set as ‘1’. [3]</td>
<td>JMPNC @jmp_label</td>
<td>@jmp_label SUBC R4, 0x3;</td>
<td>&lt;code for JMPNC&gt;</td>
</tr>
<tr>
<td>PC = New Address;</td>
<td></td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Code line 2:</td>
<td>Code line 2:</td>
<td>&lt;xx&gt;&lt;0111&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>@jmp_label</td>
<td>@jmp_nc</td>
<td>Inst 2:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;MNEM&gt;</td>
<td>SUBC R4, 0x3;</td>
<td>0000 0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1111</td>
</tr>
</tbody>
</table>

[9]
### I.3 Flow Control Instructions

<table>
<thead>
<tr>
<th>JMPNN - Jump if NOT Negative</th>
<th>Jump to a location determined by the address contained in this function if Negative bit from status bit is zero and carry, overflow and negative bits are all set as '1'. [3]</th>
<th>PC = New Address;</th>
<th>Code line 1: JMPNN @jmp_label ... Code line 2: @jmp_label &lt;MNEM&gt;</th>
<th>Code line 1: JMPNN @jmp_nc; ... Code line 2: @jmp_nn DIVC R3, 0x2;</th>
<th>Inst 1: &lt;code for JMPNN&gt; &lt;xx&gt;&lt;1011&gt; Inst 2: 0000 0000 1111 [10]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Code line 1: JMPNN @jmp_label ... Code line 2: @jmp_label &lt;MNEM&gt;</td>
<td>Code line 1: JMPNN @jmp_nc; ... Code line 2: @jmp_nn DIVC R3, 0x2;</td>
<td>Inst 1: &lt;code for JMPNN&gt; &lt;xx&gt;&lt;1011&gt; Inst 2: 0000 0000 1111 [10]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Code line 1: JMPNN @jmp_label ... Code line 2: @jmp_label &lt;MNEM&gt;</td>
<td>Code line 1: JMPNN @jmp_nc; ... Code line 2: @jmp_nn DIVC R3, 0x2;</td>
<td>Inst 1: &lt;code for JMPNN&gt; &lt;xx&gt;&lt;1011&gt; Inst 2: 0000 0000 1111 [10]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Code line 1: JMPNN @jmp_label ... Code line 2: @jmp_label &lt;MNEM&gt;</td>
<td>Code line 1: JMPNN @jmp_nc; ... Code line 2: @jmp_nn DIVC R3, 0x2;</td>
<td>Inst 1: &lt;code for JMPNN&gt; &lt;xx&gt;&lt;1011&gt; Inst 2: 0000 0000 1111 [10]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Code line 1: JMPNN @jmp_label ... Code line 2: @jmp_label &lt;MNEM&gt;</td>
<td>Code line 1: JMPNN @jmp_nc; ... Code line 2: @jmp_nn DIVC R3, 0x2;</td>
<td>Inst 1: &lt;code for JMPNN&gt; &lt;xx&gt;&lt;1011&gt; Inst 2: 0000 0000 1111 [10]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Code line 1: JMPNN @jmp_label ... Code line 2: @jmp_label &lt;MNEM&gt;</td>
<td>Code line 1: JMPNN @jmp_nc; ... Code line 2: @jmp_nn DIVC R3, 0x2;</td>
<td>Inst 1: &lt;code for JMPNN&gt; &lt;xx&gt;&lt;1011&gt; Inst 2: 0000 0000 1111 [10]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Code line 1: JMPNN @jmp_label ... Code line 2: @jmp_label &lt;MNEM&gt;</td>
<td>Code line 1: JMPNN @jmp_nc; ... Code line 2: @jmp_nn DIVC R3, 0x2;</td>
<td>Inst 1: &lt;code for JMPNN&gt; &lt;xx&gt;&lt;1011&gt; Inst 2: 0000 0000 1111 [10]</td>
</tr>
</tbody>
</table>
### I.3 Flow Control Instructions

<table>
<thead>
<tr>
<th>JMPNV - Jump if NOT Overflow</th>
<th>PC = New Address;</th>
<th>Code line 1: JMPNV @jmp_label … Code line 2: @jmp_label &lt;MNEM&gt;</th>
<th>Code line 1: JMPNV @jmp_nc; … Code line 2: @jmp_nv MULC R3, 0xA;</th>
<th>Inst 1: &lt;code for JMPNV&gt; &lt;xx&gt;&lt;1101&gt; Inst 2: 0000 0000 1111 [11]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jump to a location determined by the address contained in this function if Overflow bit from status bit is zero and carry, zero and negative bits are all set as '1'. [3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **JMPNV** - Jump if NOT Overflow
- **Overflow**
- **PC** - Program Counter
- **Address**
- **JMPNV**
- **@jmp_label**
- **<MNEM>**
- **MULC R3, 0xA;**
### I.3 Flow Control Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Code Line 1</th>
<th>Code Line 2</th>
<th>Inst 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>JMPNZ</strong> -</td>
<td>Jump if NOT</td>
<td>1: JMPNZ @jmp_label</td>
<td>1: JMPNZ @jmp_nc;</td>
<td>&lt;code for JMPNZ&gt;</td>
</tr>
<tr>
<td><strong>Jump if NOT</strong></td>
<td>Zero</td>
<td>...</td>
<td>...</td>
<td>&lt;xx&gt;&lt;1110&gt;</td>
</tr>
<tr>
<td></td>
<td>Jump to a location determined by the address contained in this function if Zero bit from status bit is '0' and the other status bits, Carry, Overflow and Negative are all set as '1'.</td>
<td>2: @jmp_label</td>
<td>2: @jmp_nn</td>
<td>Inst 2: 0000 0000 1111</td>
</tr>
<tr>
<td></td>
<td>PC = New Address;</td>
<td>&lt;MNEM&gt;</td>
<td>MULC R2, 0x9;</td>
<td>[12]</td>
</tr>
</tbody>
</table>
### I.3 Flow Control Instructions

<table>
<thead>
<tr>
<th>CALL</th>
<th>CALL Instruction</th>
<th>PC = Address1 -&gt; Address2</th>
<th>call @label &lt;MNEM&gt;</th>
</tr>
</thead>
</table>
|      | jumps to a specific address and keep the actual address stored in stack. The stored address will be used in return instruction. |                          | Code Line 1: CALL @name; ...

**Code Line 2:** @name MULC R1, 0x1;

**Inst 1:**

&lt;code for CALL&gt;

xxx xxx

**Inst 2:**

0000 1111

1111 [13]
I.3 Flow Control Instructions

### RET
- **Return from the address that was used by CALL mnemonic.**
- It increments the stack pointer and the program counter receives the value of the address stored in stack pointer.

[3] if Ri = 0, the Address contained in this function.
- if Ri = 1, the Address contained in this function plus the Program Counter value.
- if Ri = 2, the Address contained in Stack Pointer.
- if Ri greater than 2, the Address contained in this function plus the value of R[Rj].

[4] JMPU: Two machine codes are present for JMPU, one machine code specific for the instruction code, Jump unconditional, and other machine code that represents the destination address. Rj must be zero.

Obs: The instruction that Jump unconditional is pointing is at address 0x03F and all bits rela-
tive to CNVZ status (Carry, Negative, Overflow and Zero flags) are zero. According to the value of Ri, Program Counter value will change.

[5]JMPC: Two machine codes are present for JMPC, one machine code for the instruction code, Jump if Carry, and other machine code specific for the destination address. The status bits must be 4’b1000, or 0x8 for 12 and 14-bit processors. For 16-bit processor the status bits must be 4’b10000 or 0x10.

Obs: The instruction that Jump if Carry is pointing is at address 0x00F and all bits relative to CNVZ status is 0x8. According to the value of Ri, Program Counter value will change.

[6]JMPN: Two machine codes are present here, one machine code for the instruction code, Jump if Negative, and other for the destination address. The status bits must be 4’b0100, or 0x4 for 12 and 14 bits and 4’b01000, or 0x8 for 16 bits.

Obs: The instruction that Jump if Negative is pointing is at address 0x00F and all bits relative to CNVZ status are 0x4. According to the value of Ri, PC value will change.

[7]JMPV: Two machine codes are present here for this instruction code, one machine code for the instruction code, Jump if Overflow, and other for the destination address. The status bits must be 4’b0010, or 0x2 for 12 and 14 bits and 4’b0100, or 0x4 for 16 bits.

Obs: The instruction that Jump if Overflow is pointing is at address 0x00F and all bits relative to CNVZ status are 0x2. According to the value of Ri, Program Counter value will change.

[8]JMPZ: Two machine codes are present for this instruction, one machine code for the instruction code, Jump if Zero, and other for the destination address. The status bits must be 4’b0001, or 0x1 for 12 and 14 bits and 4’b0010, or 0x2 for 16 bits.
Obs: The instruction that Jump if Zero is pointing is at address 0x00F and all bits relative to CNVZ status are 0x1. According to the value of Ri, Program Counter value will change.

[9] JMPNC: Two machine codes are present for this instruction, one machine code for the instruction code, Jump if Not Carry, and other for the destination address. The status bits must be 4'b0111, or 0x7 for 12 and 14-bit processors. For 16-bit processor the status bits must be 5'b01110 or 0xE.

Obs: The instruction that Jump if Carry is pointing is at address 0x00F and all bits relative to CNVZ status is 0x7. According to the value of Ri, Program Counter value will change.

[10] JMPNN: Two machine codes are present for this instruction, one machine code for the instruction code, Jump if Not Negative and other for the destination address. The status bits must be 4'b1011, or 0xB for 12 and 14-bit processors. For 16-bit processor the status bits must be 5'b10110 or 0x16.

Obs: The second instruction represents the offset address for Jump if Not Negative. It is pointing is at address 0x00F and all bits relative to CNVZ status are 0xB. According to the value of Ri, Program Counter value will change.

[11] JMPNV: Two machine codes are present here, one machine code for the instruction code, Jump if Not Overflow and other for the destination address. The status bits must be 4'b1101, or 0xD for 12 and 14-bit processors. For 16-bit processor the status bits must be 5'b11010 or 0x1A.

Obs: The second instruction represents the offset address for Jump if Not Overflow. It is pointing is at address 0x00F and all bits relative to CNVZ status are 0xD. According to the value of Ri, Program Counter value will change.
[12] JMPNZ: Two machine codes are present, one machine code for the instruction code, Jump if Not Zero and other for the destination address. The status bits must be 4'b1110, or 0xE for 12 and 14-bit processors. For 16-bit processor the status bits must be 5'b11100 or 0x1C.

Obs: The second instruction represents the offset address for Jump if Not Zero. It is pointing is at address 0x00F and all bits relative to CNVZ status are 0xE. According to the value of Ri, PC value will change.

[13] CALL: Two machine codes are present, one for the instruction code CALL and one for the next destination address.

Obs: The instruction that CALL is pointing is at address 0x0FF.
Appendix II

Configuration File

#Configuration File

#name of the folder where the processor files are located
name_folder: 9_nxp

#name of the processor in format <filename.ext> with the extension
name: nxpRISC621pipe_v.v

#number of bits for the processor
bits:0x0E

#number of registers
registers:0x10

#Harvard(0) or Von Neumann (1)
architecture:1

#opcode size
opcode_size:0x6

#define the operand sizes for manipulation instructions, LOAD, STORE, COPY
and SWAP

  #for JUMP, CALL and RET, Ri will be defined by: (number of bits) - (4+opcode_size)

operand1_size: 0x4
operand2_size: 0x4
# data memory address bus size
dm_size: 0xE
# Program and Data Memory Size VN in power of 2
memory_size: 0x0E
# Value of PC used in calculating effective address: '0' if current instruction
pc is used, '1' if next instruction pc is used
pc_in_pc_relative: 1
# operand1_size + operand2_size + opcode_size has to be equal to bits
# Stack Pointer top
SP: 0x3FEF
# Stack growth direction up (i.e. lower to higher address "0") or down (i.e. higher
to lower address "1")
Stack_direction: 1
# Percentage of memory reserved for stack
Stack_size: 2
# opcodes

# instruction mapping: the left hand column represent user mnemonics and right
hand column are reserved mnemonics (with their explanation) reserved by the
test environment

# NOTE: mapping should occur before mnemonic : opcode definition
start_mapping:
ADD : ADD ; Addition eg. ADD Rd, Rs :: Rd <= Rd + Rs

SUB : SUB ; Subtraction eg.

ADDC : ADDC ;

SUBC : SUBC ;

MUL : MUL ;

DIV : DIV ;

MULC : MULC ;

DIVC : DIVC ;

NOT : NOT ;

AND : AND ;

OR : OR ;

XOR : XOR ;

SHLL : SHLL ;

SHRL : SHRL ;

SHLA : SHLA ;

SHRA : SHRA ;

ROTL : ROTL ;

ROTR : ROTR ;

RTLCA : RTLCA ;

RTRC : RTRC ;

COPY : COPY ;

SWAP : SWAP ;

LOAD : LOAD ;

STORE : STORE ;

JUMP : JUMP ;
CALL : CALL ;
RET : RET ;

#Put error reporting if one of the mnemonic is not mapped
end_mapping:
opcode:
start_data_transfer:
LOAD:0x00
STORE:0x01
end_data_transfer:
start_manipulation:
ADD:0x05
SUB:0x06
ADDC:0x07
SUBC:0x08
NOT:0x09
AND:0x0A
OR:0x0B
XOR:0x14
SHLL:0x15
SHRL:0x16
SHLA:0x17
SHRA:0x0C
ROTL:0x18
ROTR:0x0D
RTLCA:0x19
RTRC:0x1A
MUL:0x10
DIV:0x11
MULC:0x12
DIVC:0x13
COPY:0x02
SWAP:0x03
end_manipulation:
start_branch:
JUMP:0x04
CALL:0x1B
RET:0x1C
end_branch:
#clock cycle at which the first instruction output is obtained
clk_st:5
#name of the files used inside processor in format <filename.ext>
#note: use only 'name_pm' for Von Neumann architecture
name_pm: nxpRISC621_ram_1.v
name_dm:
name_div: nxpRISC621_div_1.v
name_mul: nxpRISC621_mul_1.v
name_cnt:
#are you for stalling LOAD, STORE, JUMP, CALL or RET?
#i.e., are you delaying the fetch of next instruction? 1-yes, 0-no
del_ld:1
del_st: 1

EOF:
## Appendix III

### List of Processors

#### III.1 Processors tested

<table>
<thead>
<tr>
<th>Processor</th>
<th>Bits</th>
<th>Architecture</th>
<th>Registers</th>
<th>Configuration File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>pa*</td>
<td>14</td>
<td>Harvard</td>
<td>16</td>
<td>configuration_1.txt</td>
</tr>
<tr>
<td>kxm*</td>
<td>12</td>
<td>Von Neumann</td>
<td>8</td>
<td>configuration_2.txt</td>
</tr>
<tr>
<td>vxk</td>
<td>14</td>
<td>Von Neumann</td>
<td>16</td>
<td>configuration_3.txt</td>
</tr>
<tr>
<td>axt</td>
<td>12</td>
<td>Von Neumann</td>
<td>8</td>
<td>configuration_5.txt</td>
</tr>
<tr>
<td>tfl</td>
<td>12</td>
<td>Harvard</td>
<td>8</td>
<td>configuration_6.txt</td>
</tr>
<tr>
<td>dnm</td>
<td>14</td>
<td>Harvard</td>
<td>16</td>
<td>configuration_7.txt</td>
</tr>
<tr>
<td>sxs</td>
<td>14</td>
<td>Harvard</td>
<td>16</td>
<td>configuration_8.txt</td>
</tr>
<tr>
<td>nxp</td>
<td>14</td>
<td>Von Neumann</td>
<td>16</td>
<td>configuration_9.txt</td>
</tr>
</tbody>
</table>

*Standard processors completely free of errors*
### III.2 Flags affected

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Flags affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD, ADDC</td>
<td>C N V Z*</td>
</tr>
<tr>
<td>ADD, OR, XOR</td>
<td>N Z</td>
</tr>
<tr>
<td>SHRA, SHLL, SHLA, SHRL, RTRC, RTLC, ROTR, ROTL, MUL, MULC, DIV, DVC, NOT</td>
<td>Z</td>
</tr>
</tbody>
</table>

*C-Carry N-Negative V-Overflow Z-Zero*
Appendix IV

Resultant graphs

IV.1 Processor *axt*

![Figure IV.1: Instruction and Error count for test TI (mode A) on processor axt](image)

Figure IV.1: Instruction and Error count for test *TI* (mode A) on processor *axt*
Figure IV.2: Classification of Errors found in processor \textit{axt} while executing test \textit{T1} (mode A)
Figure IV.3: Instruction and Error count for test T1 (mode M) on processor axt
Figure IV.4: Classification of Errors found in processor $a_{xt}$ while executing test $T1$ (mode M)
Figure IV.5: Instruction and Error count for test $T1$ (mode MB) on processor $axt$
Figure IV.6: Classification of Errors found in processor *axt* while executing test *TI* (mode MB)
Figure IV.7: Instruction and Error count for test T1 (mode MD) on processor axt
Figure IV.8: Classification of Errors found in processor $axt$ while executing test $T1$ (mode MD)
Figure IV.9: Instruction and Error count for test $T2$ (mode A) on processor $axt$
Figure IV.10: Classification of Errors found in processor $axt$ while executing test $T2$ (mode A)
Figure IV.11: Instruction and Error count for test $T2$ (mode M) on processor $axt$
Figure IV.12: Classification of Errors found in processor axt while executing test T2 (mode M)
Figure IV.13: Instruction and Error count for test T2 (mode MB) on processor axt
Figure IV.14: Classification of Errors found in processor *axt* while executing test *T2* (mode MB)
Figure IV.15: Instruction and Error count for test T2 (mode MD) on processor *axt*
Figure IV.16: Classification of Errors found in processor \textit{axt} while executing test \textit{T2} (mode MD)
IV.2 Processor \textit{dnm}

Figure IV.17: Instruction and Error count for test \textit{T1} (mode A) on processor \textit{dnm}
Figure IV.18: Classification of Errors found in processor \textit{dnm} while executing test \textit{T1} (mode A)
Figure IV.19: Instruction and Error count for test T1 (mode M) on processor dnm
Figure IV.20: Classification of Errors found in processor dnm while executing test T1 (mode M)
Figure IV.21: Instruction and Error count for test $T_1$ (mode MB) on processor $dnm$
Figure IV.22: Classification of Errors found in processor *dnn* while executing test *T1* (mode MB)
Figure IV.23: Instruction and Error count for test $Tl$ (mode MD) on processor $dnm$
Figure IV.24: Classification of Errors found in processor $dnm$ while executing test $T1$ (mode MD)
Figure IV.25: Instruction and Error count for test T2 (mode A) on processor dnm
Figure IV.26: Classification of Errors found in processor \textit{dnm} while executing test \textit{T2} (mode A)
Figure IV.27: Instruction and Error count for test T2 (mode M) on processor dnm
Figure IV.28: Classification of Errors found in processor *dnm* while executing test *T2* (mode M)
Figure IV.29: Instruction and Error count for test $T2$ (mode MB) on processor $dnm$
Figure IV.30: Classification of Errors found in processor *dnm* while executing test *T2* (mode MB)
Figure IV.31: Instruction and Error count for test T2 (mode MD) on processor $dnm$
Figure IV.32: Classification of Errors found in processor *dnm* while executing test *T2* (mode MD)
IV.3 Processor tfl

Figure IV.33: Instruction and Error count for test T1 (mode A) on processor tfl
Figure IV.34: Classification of Errors found in processor tfl while executing test T1 (mode A)
Figure IV.35: Instruction and Error count for test T1 (mode M) on processor tfl
Figure IV.36: Classification of Errors found in processor *tfl* while executing test *T1* (mode M)
Figure IV.37: Instruction and Error count for test T1 (mode MB) on processor tfl
Figure IV.38: Classification of Errors found in processor tfl while executing test $T1$ (mode MB)
Figure IV.39: Instruction and Error count for test T1 (mode MD) on processor tfl
Figure IV.40: Classification of Errors found in processor tfl while executing test T1 (mode MD)
Figure IV.41: Instruction and Error count for test $T_2$ (mode A) on processor $tfl$
Figure IV.42: Classification of Errors found in processor *tfl* while executing test *T2* (mode A)
Figure IV.43: Instruction and Error count for test $T_2$ (mode M) on processor $tfl$
Figure IV.44: Classification of Errors found in processor tfl while executing test T2 (mode M)
Figure IV.45: Instruction and Error count for test T2 (mode MB) on processor tfl
Figure IV.46: Classification of Errors found in processor tfl while executing test T2 (mode MB)
Figure IV.47: Instruction and Error count for test T2 (mode MD) on processor tfl
Figure IV.48: Classification of Errors found in processor *tfl* while executing test *T2* (mode MD)
IV.4 Processor $sxs$

Figure IV.49: Instruction and Error count for test $T_I$ (mode A) on processor $sxs$
Figure IV.50: Classification of Errors found in processor *sxs* while executing test *T1* (mode A)
Figure IV.51: Instruction and Error count for test T1 (mode M) on processor sx5
Figure IV.52: Classification of Errors found in processor *sxs* while executing test *T1* (mode *M*)
Figure IV.53: Instruction and Error count for test TI (mode MB) on processor sxs
Figure IV.54: Classification of Errors found in processor sxs while executing test T1 (mode MB)
Figure IV.55: Instruction and Error count for test $T1$ (mode MD) on processor $sxs$
Figure IV.56: Classification of Errors found in processor sx5 while executing test T1 (mode MD)
Figure IV.57: Instruction and Error count for test T2 (mode A) on processor sxs
Figure IV.58: Classification of Errors found in processor sxs while executing test T2 (mode A)
Figure IV.59: Instruction and Error count for test T2 (mode M) on processor sxs
Figure IV.60: Classification of Errors found in processor sxs while executing test T2 (mode M)
Figure IV.61: Instruction and Error count for test T2 (mode MB) on processor sxs
Figure IV.62: Classification of Errors found in processor $sxs$ while executing test $T2$ (mode MB)
Figure IV.63: Instruction and Error count for test $T_2$ (mode MD) on processor $sxs$
Figure IV.64: Classification of Errors found in processor sx$s$ while executing test $T2$ (mode MD)
IV.5 Processor \( vxk \)

Figure IV.65: Instruction and Error count for test \( T1 \) (mode A) on processor \( vxk \)
Figure IV.66: Classification of Errors found in processor $vxk$ while executing test $T1$ (mode A)
Figure IV.67: Instruction and Error count for test $T_1$ (mode M) on processor $vxk$
Figure IV.68: Classification of Errors found in processor \( vxk \) while executing test \( T1 \) (mode M)
Figure IV.69: Instruction and Error count for test $T1$ (mode MB) on processor $vxk$
Figure IV.70: Classification of Errors found in processor vxk while executing test T1 (mode MB)
Figure IV.71: Instruction and Error count for test T1 (mode MD) on processor vxk
Figure IV.72: Classification of Errors found in processor $vxk$ while executing test $T1$ (mode MD)
Figure IV.73: Instruction and Error count for test T2 (mode A) on processor vxk
Figure IV.74: Classification of Errors found in processor vxk while executing test T2 (mode A)
Figure IV.75: Instruction and Error count for test T2 (mode M) on processor vxk
Figure IV.76: Classification of Errors found in processor vxk while executing test T2 (mode M)
Figure IV.77: Instruction and Error count for test T2 (mode MB) on processor vxk
Figure IV.78: Classification of Errors found in processor \textit{vxk} while executing test \textit{T2} (mode MB)
Figure IV.79: Instruction and Error count for test $T2$ (mode MD) on processor $vxk$
Figure IV.80: Classification of Errors found in processor $vxk$ while executing test $T2$ (mode MD)
Appendix V

Source Code

V.1 Test bench generator

# This code generates a testbench in system verilog according to the configuration file

#!/usr/bin/perl
#!/usr/local/bin/perl
use strict;
use warnings;
use Getopt::Long;

# declarations

my $name = '0'; # name of the processor
my $name_folder = '0'; #name of the folder where all processor files are present
my $bits = '0'; #processor bits
my $reg = '0'; #number of gen.purpose registers
my $bits_n = '0';
my $reg_n = '0';
my $i = '0';
my $readmemh = '$readmemh';
my $fopen = '$fopen';
my $fwrite = '$fwrite';
my $p = '0';
my $q = '0';
my $arch = '0';
my $name_pm = '0';
my $name_dm = '0';
my $name_div = '0';
my $name_mul = '0';
my $name_rot_l = '0';
my $name_rot_r = '0';
my $name_rot_c = '0';
my $name_mem_mux = '0';
my $name_shift_mux = '0';
my $name_shift_arth = '0';
my $name_shift_log = '0';
my $name_add_sub = '0';
my $name_cnt = '0';
my $ext = '0';
my $stall ld = '0';
my $stall st = '0';
my $stall jmp = '0';
my $help = '0';
my $config = 'configuration.txt';
my $ret = '';
my $clk st = '';
my $clk st_1 = '';
my $clk st_2 = '';
my $len = '';

# to get the config file name (−config) from the command line.
    Default set to configuration.txt
# to get the no. of instructions (−len) that the user wants to
display prior to the instruction that has error(s).
# help is included to get correct usage of command line options
.
GetOptions(
    'len=i' => \$len,
    'config=s' => \$config,
    'help!' => \$help,
) or die "Incorrect usage! use '−help' for valid options\n";
59
60
61 # if help is passed
62 if (($help ne '0') || ($config eq ""))
63 {&help;}
64
65
66 # subroutine for help
67 sub help {
68 print "\n";
69 print "******************************************************************************\n";
70 print ">> ERROR: Incorrect Syntax <<\n";
71 print "******************************************************************************\n";
72 print "Correct usage:\n";
73 print "perl test_genr.pl -config <filename.txt >\n";
74 print "******************************************************************************\n";
75 print "\n";
76 die
77 }
78
79
80 # to read the config file and extract req info
81 sub read_config
82 {
83
open (INPUT, "<../$config");
while (<INPUT>){
  next if (/^#/w*%/);
  if (/name:\s*(\w+)\s*/)
    {$name = $1;
     $ext = $2;
     print "EXT: $ext\n";}
  elsif (/name_folder:\s*(\w+)\s*/)
    {$name_folder = $1;}
  elsif (/bits:\s*0x(\w+)\s*/)
    {$bits = $1;
     $bits = hex($bits);}
  elsif (/registers:\s*0x(\w+)\s*/)
    {$reg = $1;
     $reg = hex($reg);}
  elsif (/RET:\s*0x(\w+)\s*/)
    {$ret = $1;}
  elsif (/#RET:\s*0x(\w+)\s*/)
    {$ret = $1;}
  elsif (/architecture:\s*(\d+)\s*/)
    {$arch = $1;}
  elsif (/name_pm:\s*(\w+)\s*/)
    {$name_pm = $1;}
  elsif (/name_dm:\s*(\w+)\s*/)
    {$name_dm = $1;}
}
elsif (/name_div:\s*(\w+)\s*/)
    {$name_div = $1;}
elsif (/name_mul:\s*(\w+)\s*/)
    {$name_mul = $1;}
elsif (/name_cnt:\s*(\w+)\s*/)
    {$name_cnt = $1;}
elsif (/name_rot_l:\s*(\w+)\s*/)
    {$name_rot_l = $1;}
elsif (/name_rot_r:\s*(\w+)\s*/)
    {$name_rot_r = $1;}
elsif (/name_rot_c:\s*(\w+)\s*/)
    {$name_rot_c = $1;}
elsif (/name_mem_mux:\s*(\w+)\s*/)
    {$name_mem_mux = $1;}
elsif (/name_shift_mux:\s*(\w+)\s*/)
    {$name_shift_mux = $1;}
elsif (/name_shift_arth:\s*(\w+)\s*/)
    {$name_shift_arth = $1;}
elsif (/name_shift_log:\s*(\w+)\s*/)
    {$name_shift_log = $1;}
elsif (/name_add_sub:\s*(\w+)\s*/)
    {$name_add_sub = $1;}
elsif (/del_ld:\s*(\d)\s*/)
    { $stall_ld = $1;}
elsif (/del_st:\s*(\d+)\s*/)
134  \{ $stall_st = $1; \}
135  \{ $stall_jmp = $1; \}
136  \{ $clk_st = $1; \}
139  \}
140  \}
141
142  \&read_config;
143
144  \# to generate risc_test.sv file (src folder)
145  sub sv
146  \{ 
147  open (OUTPUT, "../risc/src/risc_test.sv");
148  \&print;
149  \} 
150
151  \# to generate interface.sv file (src folder)
152  sub intf_sv
153  \{ 
154  open (out_intf, "../risc/src/interface.sv");
155  \&print_intf;
156  \} 
157
158
159  # to generate driver.sv file (src folder)
160  sub drv_sv
161  {
162  open (out_drv, ">../risc/src/driver.sv");
163  &print_drv;
164  }
165
166
167  # to generate risc.sv rtl.f file (etc folder)
168  sub rtl_etc
169  {
170  open (out_etc, ">../risc/etc/risc.sv.rtl.f");
171  &print_etc;
172  }
173
174
175
176  # to close all opened files
177  sub close_files
178  {
179  close OUTPUT;
180  close out_intf;
181  close out_drv;
182  }
183
# to print into the risc_test.sv file
sub print {
    $bits_n = $bits - 1;
    $reg_n = $reg - 1;
    $p = $bits_n;
    $q = $p - 5;
    print OUTPUT"<<EOF;
    /*
    *
    * Author:   Namratha
    * Rochester, NY, USA
    */
    'timescale 1ns / 1ns
module test;

reg clk = 0;

reg reset;

reg scan_in0, scan_en, test_mode;

wire scan_out0;

intf_risc intf(clk); // instantiating interface
testcase test_01(intf); // instantiating testcase

if ($stall_jmp == 1 && $stall_ld == 0 && $stall_st == 0) #if only jump, call and ret is stalled

{ print OUTPUT<<EOF;

integer w, i=0, b, j=0, p=0, k=0, m=0, n=0, t, occ=0,

print_flag=0, file_P, file_Q, file_R, file_S, file_T,

jmp_flag=0, g=0, flag=0, flagSR=0;

reg st1, stp1, st2, stp2, st3, stp3, st4, stp4;

reg [{$bits_n:0}] ir3_temp, ir2_temp;
reg [ $bits_n : 0 ] PC_tmp;
EOF
}

else #jump, call, ret, load, store are all stalled
{
print OUTPUT<<EOF;

integer w, i=0, b, j=0, p=0, m=0, k=0, n=0, t, occ=0,
    print_flag=0, file_P, file_Q, file_R, file_S, file_T,
    jmp_flag=0, g=0, flag=0, flagSR=0;

reg st1, stp1, st2, stp2, st3, stp3, st4, stp4;
reg [ $bits_n : 0 ] ir3_temp, ir2_temp;
reg [ $bits_n : 0 ] PC_tmp;
EOF
}

if ($bits == 14)
{
print OUTPUT<<EOF;
reg [ $bits_n : 0 ] PC_j = 14'h0000;

initial
forever #10 clk = ~clk;
else #bits = 12
    }
    print OUTPUT<<EOF;
    reg [ $bits_n : 0 ] PC_j = 12'h000;

    initial
    forever #10 clk = ~clk;
    EOF
}

if ($arch == 0) #harvard
    { print OUTPUT <<EOF
    $name top ( 
    intf.reset ,
    intf.clk ,
    intf.R ,
    intf.SR ,
    intf.PC ,
    intf.SP ,
    intf.IR1 ,
    intf.IR2 ,
    intf.IR3 ,
    intf.PM_out ,
    }
int f . DM_out ,
int f . DM_in ,
int f . SW_pin
);
EOF
}
else  #vonnewmann
{
print OUTPUT <<EOF
$name top ( int f . reset ,
int f . clk ,
int f . R ,
int f . SR ,
int f . PC ,
int f . SP ,
int f . IR1 ,
int f . IR2 ,
int f . IR3 ,
int f . DM_out ,
int f . DM_in ,
int f . SW_pin
);
EOF
}
if ($arch == 0) {
  print OUTPUT<<EOF;

  // reading the memory.t file into the array mem_data of ram component altsyncram in dut
  initial $readmemh("../data_memory.t", test.top.my_ram.
    altsyncram_component.mem_data);

  initial $readmemh("../memory.t", test.top.my_rom.
    altsyncram_component.mem_data);

  EOF
}

else {
  print OUTPUT<<EOF;

  // reading the memory.t file into the array mem_data of ram component altsyncram in dut
  initial $readmemh("../memory.t", test.top.my_ram.
    altsyncram_component.mem_data);

  EOF
}

print OUTPUT<<EOF;
```plaintext
// reading the memory.t file into an array mem_array for use in risc_test.sv file
initial $readmemh("..\memory.t", intf.mem_array);

initial begin
  // gen
  EOF

for ($i=0; $i<$reg; $i++)
  {
    print OUTPUT "$readmemh("..\pl\R$i.t", intf.R$i) ;
    print OUTPUT "_g) ; \n"
  }
print OUTPUT "$readmemh("..\pl\SR.t", intf.SR_g) ;
    $readmemh("..\pl\PC.t", intf.PC_pl) ; \n\n// model
for ($i=0; $i<$reg; $i++)
  {
    print OUTPUT "$readmemh("..\processor\R$i.t", intf.R$i) ;
    print OUTPUT "_arr) ; \n"
  }
print OUTPUT "$readmemh("..\processor\SR.t", intf.SR_arr) ;
    nend \n\n"
print OUTPUT<<EOF;
```

V.1 Test bench generator
initial

begin

file_P = fopen("PC.t", "w");
file_R = fopen("R.t", "w");
file_Q = fopen("log.t", "w");
file_S = fopen("cplog.t", "a");
file_T = fopen("opcode.t", "a");
fwrite(file_R, "Processor, Model, Generator\n");
EOF

if ($bits == 14)
{
print OUTPUT<<EOF;
fwrite(file_R, "PC:\n\tR0\t\t\tR1\t\t\tR2\t\t\tR3\t\t\tR4\t\t\tR5\t\t\tR6\t\t\tR7\t\t\tR8\t\t\tR9\t\t\tR10\t\t\tR11\t\t\tR12\t\t\tR13\t\t\tR14\t\t\tR15\t\t\tSR\t\t\tPC\n"); // to put all the signals in an array
EOF

else #12
{

}
366  print OUTPUT<<EOF;
367  $fwrite ( file_R , "PC:\tR0\t\t\tR1\t\t\tR2\t\t\tR3\t\t\tR4\t\t\tR5\t\t\tR6\t\t\tR7\t\t\tSR\t\t\tPC\n") ;  // to put all the signals in an array
368
369
370  EOF
371  }
372
373  if ( $stall_jmp == 1 && $stall_ld == 0 && $stall_st == 0 ) #if only jump, call and ret is stalled_pa_14
374      {
375  $clk_st_1 = $clk_st -2;
376  $clk_st_2 = $clk_st -1;
377  print OUTPUT<<EOF;
378  for (b=-1; b<=$clk_st_1; b=b+1)
379      begin
380      @(posedge intf.clk);
381      begin
382      intf.IR1_risc[b] = intf.IR1;
383      if ((intfjmp_risc == intf.IR1_risc[b][$p:$q]) || (intf.cal_risc == intf.IR1_risc[b][$p:$q]) || (intf.ret_risc ==
384      intf.IR1_risc[b][$p:$q]))
385      PC_tmp = intf.PC;
386      end


```verilog
t = $clk_st_2;
EOF

else #if all are stalled (load, st, jmp, cal, ret)
{
$clk_st_1 = $clk_st -2;
$clk_st_2 = $clk_st -1;
print OUTPUT<<EOF;
for (b=-1; b<=$clk_st_1; b=b+1)
begin
@ (posedge intf.clk);
begin
intf.IR1_risc[b] = intf.IR1;
if ((intf.load_risc == intf.IR1_risc[b][p:q]) || (intf.st_risc == intf.IR1_risc[b][p:q]) || (intf.jmp_risc == intf.IR1_risc[b][p:q]) || (intf.cal_risc == intf.IR1_risc[b][p:q]) || (intf.ret_risc == intf.IR1_risc[b][p:q]))
PC_tmp = intf_PC;
end
end
t = $clk_st_2;
```

0101 Test bench generator

408  
409  
410  
411  
412  
413  
414  
415  
416  
417  
418  
419  
420  
421  
422  
423  
424  
425  
426  
427  
428  
429  
430  
431  
432  
}

EOF

print OUTPUT<<EOF;
ir3_temp = intf.IR3;
ir2_temp = intf.IR2;
intf.IR3_risc[0] = ir3_temp;
intf.IR2_risc[0] = ir2_temp;
g = g+1;

forever
 @(posedge intf.clk)
 begin

EOF

for ($i=0; $i<$reg; $i++)
 {

print OUTPUT "		intf.R$i"
;
print OUTPUT "_risc[j] = intf.R[$i];
"
}

print OUTPUT<<EOF;
intf.SR_risc[j] = intf.SR;
intf.PC_risc[j] = intf.PC;
intf.SP_risc[j] = intf.SP;
intf.IR1_risc[t] = intf.IR1;
intf.IR2_risc[g] = intf.IR2;
intf.IR3_risc[g] = intf.IR3;

intf.DM_in_risc[j] = intf.DM_in;
intf.DM_out_risc[j] = intf.DM_out;

EOF

if ($stall_jmp == 1 && $stall_ld == 0 && $stall_st == 0) # if only jump, call and ret is stalled_pa_14
{
    print OUTPUT<<EOF;

    // saving the PC before it changes to the (called or jumped location)

    if ( (intf.jmp_risc == intf.IR1_risc[t][p:q]) || (intf.cal_risc == intf.IR1_risc[t][p:q]) || (intf.ret_risc == intf.IR1_risc[t][p:q]))
        PC_tmp = intf.PC;
453 // checking for stalls and accordingly adjusting matching PC of processor and model for matching

454

455 if (occ == 0)
456 begin
457 if (intf.load_risc == intf.IR3_risc[g-1][p:q])
458 begin
459 n = n+1; occ = occ+1;
460 end
461 else if (intf.st_risc == intf.IR3_risc[g-1][p:q])
462 begin
463 n = 0; occ = occ+1;
464 end
465 else if (intf.jmp_risc == intf.IR3_risc[g-1][p:q])
466 begin
467 n = 0; occ = occ+1;
468 end
469 else if (intf.cal_risc == intf.IR3_risc[g-1][p:q])
470 begin
471 n = 0; occ = occ+1;
472 end
473 else if (intf.ret_risc == intf.IR3_risc[g-1][p:q])
474 begin
475 n = 0; occ = occ+1;
476 end
end

else begin if (jmp_flag == 0) print_flag = 0; n = 0; occ = 0; end

// to not print during the cycles required (extra) by jump, call and ret
if (((intf.jmp_risc == intf.IR3_risc[g−3][p:q]) || (intf.cal_risc == intf.IR3_risc[g−3][p:q]) || (intf.ret_risc == intf.IR3_risc[g−3][p:q])) && (intf.IR3_risc[g−2][p:q] == 6'h3F))
begin jmp_flag = 0;
jmp_flag = jmp_flag +1;
print_flag = 1;
end

// to not print an extra cycle as ret has no offset
else if ((intf.ret_risc == intf.IR3_risc[g−2][p:q]) && (intf.IR3_risc[g−1][p:q] == 6'h3F))
begin
jmp_flag = 0;
jmp_flag = jmp_flag+1;
print_flag = 1;
end
else
begin
if (jmp_flag == 1)
begin
if ((intf.jmp_risc == intf.IR3_risc[g][$p:$q]) || (intf.cal_risc == intf.IR3_risc[g][$p:$q]) || (intf.ret_risc == intf.IR3_risc[g][$p:$q]))
j = PC_tmp−2;
else

j = intf.PC−4;
jmp_flag = 0;
print_flag = 1;
end
else
begin
print_flag = 0;
end
end
V.1 Test bench generator

EOF

else # load store and jump are all stalled_kxm_12

print OUTPUT<<EOF;

if ((intf.load_risc == intf.IR1_risc[t][$p:$q]) || (intf.st_risc == intf.IR1_risc[t][$p:$q]) || (intf.jmp_risc == intf.IR1_risc[t][$p:$q]) || (intf.cal_risc == intf.IR1_risc[t][$p:$q]) || (intf.ret_risc == intf.IR1_risc[t][$p:$q]))

PC_tmp = intf.PC;

// checking for stalls and accordingly adjusting matching PC of processor and model for matching

if (occ == 0)

begin

if (intf.load_risc == intf.IR3_risc[g-1][$p:$q])

begin

n = n+1; occ = occ+1;

end

else if (intf.st_risc == intf.IR3_risc[g-1][$p:$q])

begin
n = 0; occ = occ + 1;

else if (intf.jmp_risc == intf.IR3_risc[g-1][Sp:SQ])
begin
n = 0; occ = occ + 1;
end

else if (intf.cal_risc == intf.IR3_risc[g-1][Sp:SQ])
begin
n = 0; occ = occ + 1;
end

else if (intf.ret_risc == intf.IR3_risc[g-1][Sp:SQ])
begin
n = 0; occ = occ + 1;
end

end

else
begin
if (jmp_flag == 0)
print_flag = 0;
n = 0;
occ = 0;
end
end
if (intf.jmp_risc == intf.IR3_risc[g−3][p:q]) || (intf.load_risc == intf.IR3_risc[g−3][p:q]) || (intf.st_risc == intf.IR3_risc[g−3][p:q]) || (intf.cal_risc == intf.IR3_risc[g−3][p:q]) || (intf.ret_risc == intf.IR3_risc[g−3][p:q]) && (intf.IR3_risc[g−2][p:q] == 6’h3F))

begin
jmp_flag = 0;
jmp_flag = jmp_flag + 1;
print_flag = 1;
end

else if ((intf.ret_risc == intf.IR3_risc[g−2][p:q]) && (intf.IR3_risc[g−1][p:q] == 6’h3F))

begin
jmp_flag = 0;
jmp_flag = jmp_flag + 1;
print_flag = 1;
end

else
begin
if (jmp_flag == 1)
begin
if ((intf.load_risc == intf.IR3_risc[g][p:q]) || (intf.st_risc == intf.IR3_risc[g][p:q]) || (intf.jmp_risc == intf.
.IR3_risc[g]($p:$q) || (intf.cal_risc == intf.IR3_risc[g]($p:$q)) || (intf.ret_risc == intf.IR3_risc[g]($p:$q))

584 j = PC_tmp-2;

585 else if ((intf.load_risc == intf.IR2_risc[g]($p:$q)) || (intf.st_risc == intf.IR2_risc[g]($p:$q)) || (intf.jmp_risc ==
intf.IR2_risc[g]($p:$q)) || (intf.cal_risc == intf.IR2_risc[g]($p:$q)) || (intf.ret_risc ==
intf.IR2_risc[g-1]($p:$q) != 6'h3F) && (intf.IR2_risc[g-2]($p:$q] == 6'h3F) )

586 j = PC_tmp-3;

587 else

588 j = intf.PC-4;

589 jmp_flag = 0;

590 print_flag = 1;

591 end

592 else

593 begin

594 print_flag = 0;

595 end

596 end

597

598 EOF

599 }

600

601 if ($bits == 14)
{ 

   print OUTPUT<<EOF;

   if ( print_flag == 0 )
   begin

    // printing the register (gen purpose and status) values in the file R.t

   $fwrite(file_R, " \%h, \%h, \%h;\"t", j, intf.
   PC_pl[m], intf.PC_pl[m]);

   if ( ( j != intf.PC_pl[m] ) )
   begin $fwrite(file_R, ";") ; PC_j = j; flag=1; print_con; end

   //to print the opcodes in a file (if there is a PC error dont print the next instruction)

   if ( flag == 1)
   $fwrite(file_T , "\n") ;

   else

   $fwrite(file_T , "\%2h\n", intf.mem_array[j][p:q]);

   $fwrite(file_R,"@\%h: \%h, \%h, \%h;\"t", j, intf.R0_risc[j],
   intf.R0_arr[j+n], intf.R0_g[k+n]);

   if ( ( intf.R0_risc[j] != intf.R0_arr[j+n]) || (intf.R0_risc[j]
   ) != intf.R0_g[k+n]) || (^intf.R0_risc[j] == 1'bx) )

   begin $fwrite(file_R,"*"); p=1;

   if(^intf.R0_risc[j] == 1'bx) flagSR = 2;
print_con; end

fwrite(file_R, "%h, %h, %h;\t", intf.R1_risc[j], intf.R1_arr[j+n], intf.R1_g[k+n]);
if ((intf.R1_risc[j] != intf.R1_arr[j+n]) || (intf.R1_risc[j] != intf.R1_g[k+n]) || (^intf.R1_risc[j] === 1'bX))
begin fwrite(file_R,"*");
if (^intf.R1_risc[j] === 1'bX) flagSR = 2;
print_con; end

fwrite(file_R, "%h, %h, %h;\t", intf.R2_risc[j], intf.R2_arr[j+n], intf.R2_g[k+n]);
if ((intf.R2_risc[j] != intf.R2_arr[j+n]) || (intf.R2_risc[j] != intf.R2_g[k+n]) || (^intf.R2_risc[j] === 1'bX))
begin fwrite(file_R,"*"); p=1;
if (^intf.R2_risc[j] === 1'bX) flagSR = 2;
print_con; end

fwrite(file_R, "%h, %h, %h;\t", intf.R3_risc[j], intf.R3_arr[j+n], intf.R3_g[k+n]);
begin fwrite(file_R,"*"); p=1;
if (^intf.R3_risc[j] === 1'bX) flagSR = 2;
print_con; end
V.1 Test bench generator

641 $f\text{w}rite(\text{file}_R, " \%h, \%h, \%h;\t", \text{intf.}
\text{R4\_risc}[j], \text{intf.}\text{R4\_arr}[j+n], \text{intf.}\text{R4\_g}[k+n]);

642 \text{if}( (\text{intf.}\text{R4\_risc}[j] != \text{intf.}\text{R4\_arr}[j+n]) || (\text{intf.}\text{R4\_risc}[j]
\text{ }!= \text{intf.}\text{R4\_g}[k+n]) || (^\text{intf.}\text{R4\_risc}[j] == 1'b0))

643 \begin{array}{c}
\text{begin } \$\text{fwrite(\text{file}_R,"*"}; p=1;
\text{if}(^\text{intf.}\text{R4\_risc}[j] == 1'b0) \text{flagSR } = 2;
\text{print}_\text{con}; \text{end}
\end{array}

644

647 $f\text{w}rite(\text{file}_R, " \%h, \%h, \%h;\t", \text{intf.}
\text{R5\_risc}[j], \text{intf.}\text{R5\_arr}[j+n], \text{intf.}\text{R5\_g}[k+n]);

649 \text{if}( (\text{intf.}\text{R5\_risc}[j] != \text{intf.}\text{R5\_arr}[j+n]) || (\text{intf.}\text{R5\_risc}[j]
\text{ }!= \text{intf.}\text{R5\_g}[k+n]) || (^\text{intf.}\text{R5\_risc}[j] == 1'b0))

650 \begin{array}{c}
\text{begin } \$\text{fwrite(\text{file}_R,"*"}; p=1;
\text{if}(^\text{intf.}\text{R5\_risc}[j] == 1'b0) \text{flagSR } = 2;
\text{print}_\text{con}; \text{end}
\end{array}

653

654 $f\text{w}rite(\text{file}_R, " \%h, \%h, \%h;\t", \text{intf.}
\text{R6\_risc}[j], \text{intf.}\text{R6\_arr}[j+n], \text{intf.}\text{R6\_g}[k+n]);

655 \text{if}( (\text{intf.}\text{R6\_risc}[j] != \text{intf.}\text{R6\_arr}[j+n]) || (\text{intf.}\text{R6\_risc}[j]
\text{ }!= \text{intf.}\text{R6\_g}[k+n]) || (^\text{intf.}\text{R6\_risc}[j] == 1'b0))

656 \begin{array}{c}
\text{begin } \$\text{fwrite(\text{file}_R,"*"}; p=1;
\text{if}(^\text{intf.}\text{R6\_risc}[j] == 1'b0) \text{flagSR } = 2;
\text{print}_\text{con}; \text{end}
\end{array}
$fwrite(file_R, "\%h, \%h, \%h;\t", intf.
    R7_risc[j], intf.R7_arr[j+n], intf.R7_g[k+n]);
662 begin fwrite(file_R,"*"), p=1;
663 if(^intf.R7_risc[j] === 1'b) flagSR = 2;
664 print_con; end
665
666 $fwrite(file_R, "\%h, \%h, \%h;\t", intf.
    R8_risc[j], intf.R8_arr[j+n], intf.R8_g[k+n]);
668 begin fwrite(file_R,"*"), p=1;
669 if(^intf.R8_risc[j] === 1'b) flagSR = 2;
670 print_con; end
671
672 $fwrite(file_R, "\%h, \%h, \%h;\t", intf.
    R9_risc[j], intf.R9_arr[j+n], intf.R9_g[k+n]);
674 begin fwrite(file_R,"*"), p=1;
675 if(^intf.R9_risc[j] === 1'b) flagSR = 2;
676 print_con; end
677
$fwrite(file_R, " \%h, \%h, \%h;\t", intf.
    R10_risc[j], intf.R10_arr[j+n], intf.R10_g[k+n]);

if ( (intf.R10_risc[j] != intf.R10_arr[j+n]) || (intf.
    R10_risc[j] != intf.R10_g[k+n]) || (^intf.R10_risc[j] === 1'
    bx))
begin $fwrite(file_R, " * "); p=1;
if (^intf.R10_risc[j] === 1'bx) flagSR = 2;
print_con; end

$fwrite(file_R, " \%h, \%h, \%h;\t", intf.
    R11_risc[j], intf.R11_arr[j+n], intf.R11_g[k+n]);

    R11_risc[j] != intf.R11_g[k+n]) || (^intf.R11_risc[j] === 1'
    bx))
begin $fwrite(file_R, " * "); p=1;
if (^intf.R11_risc[j] === 1'bx) flagSR = 2;
print_con; end

$fwrite(file_R, " \%h, \%h, \%h;\t", intf.
    R12_risc[j], intf.R12_arr[j+n], intf.R12_g[k+n]);

    R12_risc[j] != intf.R12_g[k+n]) || (^intf.R12_risc[j] === 1'
    bx))
begin $fwrite(file_R, " * "); p=1;
if (^intf.R12_risc[j] === 1'bx) flagSR = 2;
V.1 Test bench generator

694 print_con; end

695

696 $fwrite(file_R, " \%h, \%h, \%h;\t", intf.
    R13_risc[j], intf.R13_arr[j+n], intf.R13_g[k+n]);

    R13_risc[j] != intf.R13_g[k+n]) || (^intf.R13_risc[j] === 1'
    bx) )

698 begin $fwrite(file_R,"*"); p=1;

699 if(^intf.R13_risc[j] === 1'bx) flagSR = 2;

700 print_con; end

701

702 $fwrite(file_R, " \%h, \%h, \%h;\t", intf.
    R14_risc[j], intf.R14_arr[j+n], intf.R14_g[k+n]);

703 if ( (intf.R14_risc[j] != intf.R14_arr[j+n]) || (intf.
    R14_risc[j] != intf.R14_g[k+n]) || (^intf.R14_risc[j] === 1'
    bx) )

704 begin $fwrite(file_R,"*"); p=1;

705 if(^intf.R14_risc[j] === 1'bx) flagSR = 2;

706 print_con; end

707

708 $fwrite(file_R, " \%h, \%h, \%h;\t", intf.
    R15_risc[j], intf.R15_arr[j+n], intf.R15_g[k+n]);

    R15_risc[j] != intf.R15_g[k+n]) || (^intf.R15_risc[j] === 1'
    bx) )
begin $fwrite (file_R,"*"); p=1;
711 if (^intf.R15_risc[j] === 1'bX) flagSR = 2;
712 print_con; end
713
714 fwrite (file_R, "\%h, \%h, \%h;\t", interf.
715 SR_risc[j], interf.SR_arr[j+n], interf.SR_g[k+n]);
716 if ( ( interf.SR_risc[j] != interf.SR_arr[j+n]) || ( interf.SR_risc[j]
717 != interf.SR_g[k+n]) || (^ interf.SR_risc[j] === 1'bX) )
718 begin $fwrite (file_R,"*"); flagSR = (flagSR == 2) ? flagSR : 1;
719 print_con; end
720 fwrite (file_P, "@\%h j=\%h \n",k,j);
721 k = k+1;
722 fwrite (file_R, "\n\n") ;
723 m=m+1;
724 end
725
726 j = j+1; g = g+1; t = t+1;
727 end
728 end
729 task print_con;
730 begin
731 for (w=$len; w>=0; w=w-1)
begin
if ((w == 0) && (flag != 1)) // all errors except PC error
begin
if (flagSR == 1)
\$display ("A Mismatch Occurred S @\%4h! $name", intf.PC_pl[m-w]);
else if (flagSR == 2)
\$display ("A Mismatch Occurred R @\%4h! $name", intf.PC_pl[m-w]);
else
\$display ("A Mismatch Occurred @\%4h! $name", intf.PC_pl[m-w]);
\$display ("\tTime :\%t ps\n", $time);
if (flagSR == 1)
\$fwrite (file_S, "A Mismatch Occurred S @\%4h! $name", intf.PC_pl[m-w]);
else if (flagSR == 2)
\$fwrite (file_S, "A Mismatch Occurred R @\%4h! $name", intf.PC_pl[m-w]);
else
\$fwrite (file_S, "A Mismatch Occurred @\%4h! $name", intf.PC_pl[m-w]);
751 \$fwrite (file_S, "\\tTime :\%t ps\n", \$time);
752
753 p = 0;
754 end
755 if ((w == 0) && (flag == 1)) // when PC error is present
756 begin
757 \$display ("A Mismatch Occurred PC @\%4h! $name", intf.PC_pl[m–w]);
758 \$display ("\\tTime :\%t ps\n", \$time);
759
760 \$fwrite (file_S, "A Mismatch Occurred PC @\%4h! $name", intf.
    PC_pl[m–w]);
761 \$fwrite (file_S, "\\tTime :\%t ps\n", \$time);
762
763 if (intf.mem_array[j][p:q] == intf.ret_risc) begin
764 \$display (" Instruction @\%4h : \%h", intf.PC_pl[m–1], intf.
    mem_array[j]);
765 \$fwrite (file_S, " Instruction @\%4h : \%h\n", intf.PC_pl[m
    –1], intf.mem_array[j]); end
766 else begin
767 \$display (" Instruction @\%4h : \%h", intf.PC_pl[m–2], intf.
    mem_array[j–1]);
768 \$fwrite (file_S, " Instruction @\%4h : \%h\n", intf.PC_pl[m
    –2], intf.mem_array[j–1]); end
769
770 \$display
     ("---------------------------------------------------------------");
771 \$display (" Processor Model Generator \n");
772
773
774 \$fwrite (file_S, 
     "---------------------------------------------------------------\n") ;
775 \$fwrite (file_S, " Processor Model Generator \n");
776
777
778 j = intf.PC_pl[m-w];
779 \$display ("PC %4h %h %h *", PC_j, intf.PC_pl[m-w], intf.
PC_pl[m-w]);
780 \$fwrite (file_S, "PC %4h %h %h \n", PC_j, intf.PC_pl[m-w]
), intf.PC_pl[m-w]);
781 \$display
     ("---------------------------------------------------------------")
 ;
782 \$fwrite (file_S, 
     "---------------------------------------------------------------\n")
 ;
783 \$display ("\n");
784 \$fwrite (file_S, "\n");
785 p = 0;
\$finish;
end

j = intf$PC_pl[m-w];

\$display ("\tInstruction @\%4h : \%h", intf$PC_pl[m-w], intf$mem_array[j]);
\$display ("−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−");
\$display (" Processor Model Generator\n");

\$fwrite (file_S, "\tInstruction @\%4h : \%h\%\n", intf$PC_pl[m-w], intf$mem_array[j]);
\$fwrite (file_S, "−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−\n");
\$fwrite (file_S, " Processor Model Generator\n");

if ((intf$mem_array[j][p:q] == intf$load_risc) && ((intf$mem_array[j-1][p:q]!= intf$cal_risc) && (intf$mem_array[j-1][p:q]! = intf$jmp_risc) && (intf$mem_array[j-1][p:q]! = intf$st_risc) && (intf$mem_array[j-1][p:q]! = intf$load_risc) 
  || ((j==0) && (intf$mem_array[j][p:q] == intf$load_risc))
) 
  n=1; else n=0;
if ((intf.R0_risc[j] != intf.R0_arr[j+n]) || (intf.R0_risc[j] != intf.R0_g[k+n-w]) || (^intf.R0_risc[j] === 1 'bx)) begin
    \display("R0 \%h \%h \%h *", intf.R0_risc[j], intf.R0_arr[j+n], intf.R0_g[k+n-w]);
    \fwrite(file_S, "R0 \%h \%h \%h \n", intf.R0_risc[j], intf.R0_arr[j+n], intf.R0_g[k+n-w]); end
else begin
    \display("R0 \%h \%h \%h", intf.R0_risc[j], intf.R0_arr[j+n], intf.R0_g[k+n-w]);
    \fwrite(file_S, "R0 \%h \%h \%h\n", intf.R0_risc[j], intf.R0_arr[j+n], intf.R0_g[k+n-w]); end

if ((intf.R1_risc[j] != intf.R1_arr[j+n]) || (intf.R1_risc[j] != intf.R1_g[k+n-w]) || (^intf.R1_risc[j] === 1 'bx)) begin
    \display("R1 \%h \%h \%h *", intf.R1_risc[j], intf.R1_arr[j+n], intf.R1_g[k+n-w]);
    \fwrite(file_S, "R1 \%h \%h \%h \n", intf.R1_risc[j], intf.R1_arr[j+n], intf.R1_g[k+n-w]); end
else begin
    \display("R1 \%h \%h \%h", intf.R1_risc[j], intf.R1_arr[j+n], intf.R1_g[k+n-w]);
    \fwrite(file_S, "R1 \%h \%h \%h\n", intf.R1_risc[j], intf.R1_arr[j+n], intf.R1_g[k+n-w]); end
  $display ("R2 \%h \%h \%h *", intf.R2_risc[j], intf.R2_arr[j+n], intf.R2_g[k+n-w]);
  $fwrite (file_S, "R2 \%h \%h \%h 
", intf.R2_risc[j], intf.R2_arr[j+n], intf.R2_g[k+n-w]);
end

else begin
  $display ("R2 \%h \%h \%h", intf.R2_risc[j], intf.R2_arr[j+n], intf.R2_g[k+n-w]);
  $fwrite (file_S, "R2 \%h \%h \%h\n", intf.R2_risc[j], intf.R2_arr[j+n], intf.R2_g[k+n-w]);
end

  $display ("R3 \%h \%h \%h *", intf.R3_risc[j], intf.R3_arr[j+n], intf.R3_g[k+n-w]);
  $fwrite (file_S, "R3 \%h \%h \%h\n", intf.R3_risc[j], intf.R3_arr[j+n], intf.R3_g[k+n-w]);
end

else begin
  $display ("R3 \%h \%h \%h", intf.R3_risc[j], intf.R3_arr[j+n], intf.R3_g[k+n-w]);
\$fwrite (file_S, "R3 \%h \%h \%h\n", intf.R3_risc[j], intf.R3_arr[j+n], intf.R3_g[k+n-w]); end

\$display ("R4 \%h \%h \%h *"); intf.R4_risc[j], intf.R4_arr[j+n], intf.R4_g[k+n-w]);
\$fwrite (file_S, "R4 \%h \%h \%h\n", intf.R4_risc[j], intf.R4_arr[j+n], intf.R4_g[k+n-w]); end

else begin
\$display ("R4 \%h \%h \%h
", intf.R4_risc[j], intf.R4_arr[j+n], intf.R4_g[k+n-w]);
\$fwrite (file_S, "R4 \%h \%h \%h\n", intf.R4_risc[j], intf.R4_arr[j+n], intf.R4_g[k+n-w]); end

if ((intf.R5_risc[j] != intf.R5_arr[j+n]) || (intf.R5_risc[j] != intf.R5_g[k+n-w]) || (^intf.R5_risc[j] === 1'bX)) begin
\$display ("R5 \%h \%h \%h *"); intf.R5_risc[j], intf.R5_arr[j+n], intf.R5_g[k+n-w]);
\$fwrite (file_S, "R5 \%h \%h \%h\n", intf.R5_risc[j], intf.R5_arr[j+n], intf.R5_g[k+n-w]); end

else begin
\$Display \("R5 \%h \%h \%h\), intf.R5_risc[j], intf.R5_arr[j+n], intf.R5_g[k+n-w];

\$fwrite (file_S, "R5 \%h \%h \%h\n", intf.R5_risc[j], intf.R5_arr[j+n], intf.R5_g[k+n-w]); end

\$Display \("R6 \%h \%h \%h *\), intf.R6_risc[j], intf.R6_arr[j+n], intf.R6_g[k+n-w];
\$fwrite (file_S, "R6 \%h \%h \%h *\n", intf.R6_risc[j], intf.R6_arr[j+n], intf.R6_g[k+n-w]); end

else begin
\$Display \("R6 \%h \%h \%h\), intf.R6_risc[j], intf.R6_arr[j+n], intf.R6_g[k+n-w];
\$fwrite (file_S, "R6 \%h \%h \%h\n", intf.R6_risc[j], intf.R6_arr[j+n], intf.R6_g[k+n-w]); end

if ((intf.R7_risc[j] != intf.R7_arr[j+n]) || (intf.R7_risc[j] != intf.R7_g[k+n-w]) || (^intf.R7_risc[j] === 1'b0)) begin
\$Display \("R7 \%h \%h \%h *\), intf.R7_risc[j], intf.R7_arr[j+n], intf.R7_g[k+n-w];
\$fwrite (file_S, "R7 \%h \%h \%h *\n", intf.R7_risc[j], intf.R7_arr[j+n], intf.R7_g[k+n-w]); end
else begin
\$display("R7 \%h \%h \%h", intf.R7_risc[j], intf.R7_arr[j+n], intf.R7_g[k+n–w]);
\$fwrite(file_S, "R7 \%h \%h \%h\n", intf.R7_risc[j], intf.R7_arr[j+n], intf.R7_g[k+n–w]); end

\$display("R8 \%h \%h \%h *
", intf.R8_risc[j], intf.R8_arr[j+n], intf.R8_g[k+n–w]);
\$fwrite(file_S, "R8 \%h \%h \%h\n", intf.R8_risc[j], intf.R8_arr[j+n], intf.R8_g[k+n–w]); end

else begin
\$display("R8 \%h \%h \%h", intf.R8_risc[j], intf.R8_arr[j+n], intf.R8_g[k+n–w]);
\$fwrite(file_S, "R8 \%h \%h \%h\n", intf.R8_risc[j], intf.R8_arr[j+n], intf.R8_g[k+n–w]); end

\$display("R9 \%h \%h \%h *
", intf.R9_risc[j], intf.R9_arr[j+n], intf.R9_g[k+n–w]);
\$fwrite(file_S, "R9 \%h \%h \%h\n", intf.R9_risc[j], intf.R9_arr[j+n], intf.R9_g[k+n–w]); end
else begin

\$display ("R9 \%h \%h \%h", intf.R9_risc[j], intf.R9_arr[j+n], intf.R9_g[k+n-w]);
\$fwrite (file_S, "R9 \%h \%h \%h\n", intf.R9_risc[j], intf.R9_arr[j+n], intf.R9_g[k+n-w]); end

begin
\$display ("R10 \%h \%h \%h *", intf.R10_risc[j], intf.R10_arr[j+n], intf.R10_g[k+n-w]);
\$fwrite (file_S, "R10 \%h \%h \%h \n", intf.R10_risc[j], intf.R10_arr[j+n], intf.R10_g[k+n-w]); end
else begin
\$display ("R10 \%h \%h \%h", intf.R10_risc[j], intf.R10_arr[j+n], intf.R10_g[k+n-w]);
\$fwrite (file_S, "R10 \%h \%h \%h\n", intf.R10_risc[j], intf.R10_arr[j+n], intf.R10_g[k+n-w]); end

begin
890 \$display("R11 \%h \%h \%h *", intf.R11_risc[j], intf.R11_arr[j+n], intf.R11_g[k+n-w]);
891 \$fwrite(file_S, "R11 \%h \%h \%h *\n", intf.R11_risc[j], intf.R11_arr[j+n], intf.R11_g[k+n-w]); end
892
893 else begin
894 \$display("R11 \%h \%h \%h", intf.R11_risc[j], intf.R11_arr[j+n], intf.R11_g[k+n-w]);
895 \$fwrite(file_S, "R11 \%h \%h \%h\n", intf.R11_risc[j], intf.R11_arr[j+n], intf.R11_g[k+n-w]); end
896
begin
898 \$display("R12 \%h \%h \%h *", intf.R12_risc[j], intf.R12_arr[j+n], intf.R12_g[k+n-w]);
899 \$fwrite(file_S, "R12 \%h \%h \%h *\n", intf.R12_risc[j], intf.R12_arr[j+n], intf.R12_g[k+n-w]); end
900
901 else begin
902 \$display("R12 \%h \%h \%h", intf.R12_risc[j], intf.R12_arr[j+n], intf.R12_g[k+n-w]);
903 \$fwrite(file_S, "R12 \%h \%h \%h\n", intf.R12_risc[j], intf.R12_arr[j+n], intf.R12_g[k+n-w]); end

begin

$display("R13 \%h \%h \%h *", intf.R13_risc[j], intf.R13_arr[j+n], intf.R13_g[k+n-w]);

fwrite(file_S, "R13 \%h \%h \%h\n", intf.R13_risc[j], intf.R13_arr[j+n], intf.R13_g[k+n-w]); end

else begin

$display("R13 \%h \%h \%h", intf.R13_risc[j], intf.R13_arr[j+n], intf.R13_g[k+n-w]);

fwrite(file_S, "R13 \%h \%h \%h\n", intf.R13_risc[j], intf.R13_arr[j+n], intf.R13_g[k+n-w]); end


begin

$display("R14 \%h \%h \%h *", intf.R14_risc[j], intf.R14_arr[j+n], intf.R14_g[k+n-w]);

fwrite(file_S, "R14 \%h \%h \%h\n", intf.R14_risc[j], intf.R14_arr[j+n], intf.R14_g[k+n-w]); end

else begin

$display("R14 \%h \%h \%h", intf.R14_risc[j], intf.R14_arr[j+n], intf.R14_g[k+n-w]);
\$fwrite (file_S, "R14 \%h \%h \%h\n", intf.R14_risc[j],
    intf.R14_arr[j+n], intf.R14_g[k+n+w]); end

    begin
\$display ("R15 \%h \%h \%h *", intf.R15_risc[j], intf.R15_arr[j+n], intf.R15_g[k+n-w]);
\$fwrite (file_S, "R15 \%h \%h \%h \%h\n", intf.R15_risc[j],
    intf.R15_arr[j+n], intf.R15_g[k+n-w]); end

else begin
\$display ("R15 \%h \%h \%h", intf.R15_risc[j], intf.R15_arr[j+n], intf.R15_g[k+n-w]);
\$fwrite (file_S, "R15 \%h \%h \%h\n", intf.R15_risc[j],
    intf.R15_arr[j+n], intf.R15_g[k+n-w]); end

// assigning the individual status bits to a variable and to display
st1 = intf.SR_risc[j][7]; st2 = intf.SR_risc[j][6]; st3 = intf.SR_risc[j][5]; st4 = intf.SR_risc[j][4];
stp1 = intf.SR_g[k+n-w][7]; stp2 = intf.SR_g[k+n-w][6]; stp3 = intf.SR_g[k+n-w][5]; stp4 = intf.SR_g[k+n-w][4];
if ((intf.SR_risc[j] != intf.SR_arr[j+n]) || (intf.SR_risc[j] != intf.SR_g[k+n-w]) || (^intf.SR_risc[j] === 1 'bx)) begin
\$display("SR \%h \%h \%h *", intf.SR_risc[j], intf.SR_arr[j+n], intf.SR_g[k+n-w]);
\$fwrite(file_S, "SR \%h \%h \%h \n", intf.SR_risc[j], intf.SR_arr[j+n], intf.SR_g[k+n-w]); end

else begin
\$display("SR \%h \%h \%h", intf.SR_risc[j], intf.SR_arr[j+n], intf.SR_g[k+n-w]);
\$fwrite(file_S, "SR \%h \%h \%h \n", intf.SR_risc[j], intf.SR_arr[j+n], intf.SR_g[k+n-w]); end

\$display("PC %4h %5h %5h", j, intf.PC_pl[m-w], intf.PC_pl[m-w]);
\$fwrite(file_S, "PC %4h %5h %5h \n", j, intf.PC_pl[m-w], intf.PC_pl[m-w]);

if ((intf.SR_risc[j] != intf.SR_arr[j+n]) || (intf.SR_risc[j] != intf.SR_g[k+n-w]) || (^intf.SR_risc[j] === 1 'bx))
begin
\$display("Status Bits (processor): SR=\%h; C=\%d, N=\%d, V=\%d, Z=\%d", intf.SR_risc[j-w], st1, st2, st3, st4);
\$display("Status Bits (expected): SR=\%4h; C=\%d, N=\%d, V=\%d, Z=\%d", intf.SR_g[k+n-w], stp1, stp2, stp3, stp4);
948 \$fwrite (file_S, "Status Bits (processor): SR=\%h; C=\%d, N=\%d, V=\%d, Z=\%d\n", intf.SR_risc[j−w], st1, st2, st3, st4);
949 \$fwrite (file_S, "Status Bits (expected): SR=\%4h; C=\%d, N=\%d, V=\%d, Z=\%d\n", intf.SR_g[k+n−w], stp1, stp2, stp3, stp4);
950 end
951 else begin
952 \$display ("Status Bits: C=\%d, N=\%d, V=\%d, Z=\%d", st1, st2, st3, st4);
953 \$fwrite (file_S, "Status Bits: C=\%d, N=\%d, V=\%d, Z=\%d\n", st1, st2, st3, st4); end
954
955 \$display
("\n
\n");
956 \$fwrite (file_S,
"\n
\n"");
957 end
958 end
959 \$finish;
960 end
961 endtask
962
963 initial
964 #23000 \$finish;
endmodule

EOF

else {
    print OUTPUT<<EOF;

if (print_flag == 0)
    begin
        // printing the register (gen purpose and status) values in the file R.t
        $fwrite(file_R, " \%h, \%h, \%h;\\t", j, intf.
            PC_pl[m], intf.PC_pl[m]);
        if ( ( j != intf.PC_pl[m]) )
            begin $fwrite(file_R,"*"); PC_j=j; flag=1; print_con; end

    // to print the opcodes in a file (if there is a PC error dont print the next instruction)
    if (flag == 1)
        $fwrite(file_T, "\\n");
    else
        $fwrite(file_T, "\%2h\\n", intf.mem_array[j][p:q]);
987 `fwrite(file_R,"@\%h: \%h, \%h, \%h;\t", j, intf.R0_risc[j],
       intf.R0_arr[j+n], intf.R0_g[k+n]); //add feature that will
       enable multiple stalls
988 if ( (intf.R0_risc[j] != intf.R0_arr[j+n]) || (intf.R0_risc[j]
       != intf.R0_g[k+n]) || (^intf.R0_risc[j] === 1'b1) )
989 begin fwrite(file_R,"*"), p=1;
990 if (^intf.R0_risc[j] === 1'b1) flagSR = 2;
991 print_con; end
992
993 fwrite(file_R, " \%h, \%h, \%h;\t", intf.
       R1_risc[j], intf.R1_arr[j+n], intf.R1_g[k+n]);
994 if ( (intf.R1_risc[j] != intf.R1_arr[j+n]) || (intf.R1_risc[j]
       != intf.R1_g[k+n]) || (^intf.R1_risc[j] === 1'b1) )
995 begin fwrite(file_R,"*"), p=1;
996 if (^intf.R1_risc[j] === 1'b1) flagSR = 2;
997 print_con; end
998
999 fwrite(file_R, " \%h, \%h, \%h;\t", intf.
       R2_risc[j], intf.R2_arr[j+n], intf.R2_g[k+n]);
1000 if ( (intf.R2_risc[j] != intf.R2_arr[j+n]) || (intf.R2_risc[j]
       != intf.R2_g[k+n]) || (^intf.R2_risc[j] === 1'b1))
1001 begin fwrite(file_R,"*"), p=1;
1002 if (^intf.R2_risc[j] === 1'b1) flagSR = 2;
1003 print_con; end
1004
$fwrite(file_R, " \%h, \%h, \%h;\t", intf.
    R3_risc[j], intf.R3_arr[j+n], intf.R3_g[k+n]);

if ( (intf.R3_risc[j] != intf.R3_arr[j+n]) || (intf.R3_risc[j]
    != intf.R3_g[k+n]) || (^intf.R3_risc[j] == 1'bx) )
begin fwrite(file_R, "*"); p=1;
if (^intf.R3_risc[j] == 1'bx) flagSR = 2;
print_con; end

fwrite(file_R, " \%h, \%h, \%h;\t", intf.
    R4_risc[j], intf.R4_arr[j+n], intf.R4_g[k+n]);

    != intf.R4_g[k+n]) || (^intf.R4_risc[j] == 1'bx) )
begin fwrite(file_R,"*"); p=1;
if (^intf.R4_risc[j] == 1'bx) flagSR = 2;
print_con; end

fwrite(file_R, " \%h, \%h, \%h;\t", intf.
    R5_risc[j], intf.R5_arr[j+n], intf.R5_g[k+n]);

if ( (intf.R5_risc[j] != intf.R5_arr[j+n]) || (intf.R5_risc[j]
    != intf.R5_g[k+n]) || (^intf.R5_risc[j] == 1'bx) )
begin fwrite(file_R,"*"); p=1;
if (^intf.R5_risc[j] == 1'bx) flagSR = 2;
print_con; end

1022
fwrite(file_R, " \%h, \%h, \%h;\t", intf.
    R6_risc[j], intf.R6_arr[j+n], intf.R6_g[k+n]);

if ( ( intf.R6_risc[j] != intf.R6_arr[j+n]) || (intf.R6_risc[j]
    != intf.R6_g[k+n]) || (^intf.R6_risc[j] === 1'bx) )
begin fwrite(file_R,"*"'); p=1;
if(^intf.R6_risc[j] === 1'bx) flagSR = 2;
print_con; end

fwrite(file_R, " \%h, \%h, \%h;\t", intf.
    R7_risc[j], intf.R7_arr[j+n], intf.R7_g[k+n]);

if ( ( intf.R7_risc[j] != intf.R7_arr[j+n]) || (intf.R7_risc[j]
    != intf.R7_g[k+n]) || (^intf.R7_risc[j] === 1'bx) )
begin fwrite(file_R,"*"'); p=1;
if(^intf.R7_risc[j] === 1'bx) flagSR = 2;
print_con; end

fwrite(file_R, " \%h, \%h, \%h;\t", intf.
    SR_risc[j], intf.SR_arr[j+n], intf.SR_g[k+n]);

if ( ( intf.SR_risc[j] != intf.SR_arr[j+n]) || (intf.SR_risc[j]
    != intf.SR_g[k+n]) || (^intf.SR_risc[j] === 1'bx) )
begin fwrite(file_R,"*"'); flagSR = (flagSR == 2) ? flagSR : 1;
    print_con; end

fwrite(file_P, "@\%h j=\%h \n",k,j);
k = k + 1;
fwrite(file_R, "\n\n\n");
if (j != intf.PC_pl[m])
begin
fwrite(file_Q, "@\%h, perl= \%h, testbench= \%h\n", m, intf
    .PC_pl[m], j);
end
m=m+1;
end
j = j+1; g = g+1; t = t+1;
end
task print_con;
begin
for (w=$len; w>=0; w=w-1)
begin
if ((w == 0) && (flag != 1)) // all errors except PC error
begin
if (flagSR == 1)
$display("A Mismatch Occurred S @\%4h! $name", intf.PC_pl[m-
    w]);
else if (flagSR == 2)
$display("A Mismatch Occurred R @\%4h! $name", intf.PC_pl[m-
    w]);
1063  else
1064  \$display ("A Mismatch Occurred @\%4h! $name", intf.PC_pl[\m-w]);
1065
1066  \$display ("\\tTime :\% t ps\n", \$time);
1067
1068  if (flagSR == 1)
1069  \$fwrite (file_S, "A Mismatch Occurred S @\%4h! $name", intf.
  PC_pl[\m-w]);
1070  else if (flagSR == 2)
1071  \$fwrite (file_S, "A Mismatch Occurred R @\%4h! $name", intf.
  PC_pl[\m-w]);
1072  else
1073  \$fwrite (file_S, "A Mismatch Occurred @\%4h! $name", intf.
  PC_pl[\m-w]);
1074
1075  \$fwrite (file_S, "\\tTime :\% t ps\n", \$time);
1076
1077
1078  p = 0;
1079  end
1080  if ((w == 0) && (flag == 1)) // when PC error is present
1081  begin
1082  \$display ("A Mismatch Occurred PC @\%4h! $name", intf.PC_pl[\m-
  w]);
V.1 Test bench generator

1083 \$display ("\tTime :\%t ps\n",\$time);

1084

1085 \$fwrite (file_S, "A Mismatch Occurred PC @%4h! $name", intf.
    PC_pl[m–w]);

1086 \$fwrite (file_S, "\tTime :\%t ps\n",\$time);

1087

1088 if (intf.mem_array[j][p:q] == intf.ret_risc) begin

1089 \$display (" Instruction @%4h : \%h", intf.PC_pl[m–1], intf.
    mem_array[j]);

1090 \$fwrite (file_S, " Instruction @%4h : \%h\n", intf.PC_pl[m
    –1], intf.mem_array[j]); end

1091 else begin

1092 \$display (" Instruction @%4h : \%h", intf.PC_pl[m–2], intf.
    mem_array[j –1]);

1093 \$fwrite (file_S, " Instruction @%4h : \%h\n", intf.PC_pl[m
    –2], intf.mem_array[j –1]); end

1094

1095 \$display ("
    −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−");

1096 \$display (" Processor Model  Generator\n");

1097

1098

1099 \$fwrite (file_S, "
    −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−\n");

1100 \$fwrite (file_S, " Processor Model  Generator\n");
j = intf.PC_pl[m-w];

$display ("PC %4h I %h *", PC_j, intf.PC_pl[m-w], intf.
PC_pl[m-w]);

fwrite (file_S, "PC %4h I %h *\n", PC_j, intf.PC_pl[m-w]
], intf.PC_pl[m-w]);

$display ("%n"),

fwrite (file_S, "%n")

p = 0;

$finish;
end

j = intf.PC_pl[m-w];

$display ("Instruction @%4h : %h", intf.PC_pl[m-w], intf.
mem_array[j]);

$display ("%n", Processor Model Generator\n"
1118 \$fwrite (file_S, "\\tInstruction @%4h : %h\n", intf.PC_pl[m-w], intf.mem_array[j]);

1119 \$fwrite (file_S, "
------------------------------------------------------------------------\n")

1120 \$fwrite (file_S, " Processor Model Generator\n")

1121


1123 n=1; else n=0;

1124

1125 if ((intf.R0_risc[j] != intf.R0_arr[j+n]) || (intf.R0_risc[j] != intf.R0_g[k+n-w]) || (^intf.R0_risc[j] === 1'bX)) begin

1126 \$display ("R0 \%h \%h \%h *", intf.R0_risc[j], intf.R0_arr[j+n], intf.R0_g[k+n-w]);

1127 \$fwrite (file_S, "R0 \%h \%h \%h \n", intf.R0_risc[j], intf.R0_arr[j+n], intf.R0_g[k+n-w]); end

1128 else begin

1129 \$display ("R0 \%h \%h \%h", intf.R0_risc[j], intf.R0_arr[j+n], intf.R0_g[k+n-w]);
1131 \$fwrite (file_S, "R0 \%h \%h \%h\n", intf.R0_risc[j], intf.
R0_arr[j+n], intf.R0_g[k+n-w]); end

1132

1133 if ((intf.R1_risc[j] != intf.R1_arr[j+n]) || (intf.R1_risc[j]
!= intf.R1_g[k+n-w]) || (^intf.R1_risc[j] == 1'b)) begin

1134 \$display ("R1 \%h \%h \%h *", intf.R1_risc[j], intf.R1_arr[
       j+n], intf.R1_g[k+n-w]);

1135 \$fwrite (file_S, "R1 \%h \%h \%h\n", intf.R1_risc[j], intf.
       R1_arr[j+n], intf.R1_g[k+n-w]); end

1136

1137 else begin

1138 \$display ("R1 \%h \%h \%h", intf.R1_risc[j], intf.R1_arr[j+n], intf.
       R1_g[k+n-w]);

1139 \$fwrite (file_S, "R1 \%h \%h \%h\n", intf.R1_risc[j], intf.
       R1_arr[j+n], intf.R1_g[k+n-w]); end

1140

1141 if ((intf.R2_risc[j] != intf.R2_arr[j+n]) || (intf.R2_risc[j]
!= intf.R2_g[k+n-w]) || (^intf.R2_risc[j] == 1'b)) begin

1142 \$display ("R2 \%h \%h \%h *", intf.R2_risc[j], intf.R2_arr[
       j+n], intf.R2_g[k+n-w]);

1143 \$fwrite (file_S, "R2 \%h \%h \%h\n", intf.R2_risc[j], intf.
       R2_arr[j+n], intf.R2_g[k+n-w]); end

1144

1145 else begin
\$display ("R2 \%h \%h \%h", intf.R2_risc[j], intf.R2_arr[j+n], intf.R2_g[k+n-w]);
1147 $fwrite (file_S, "R2 \%h \%h \%h\n", intf.R2_risc[j], intf.R2_arr[j+n], intf.R2_g[k+n-w]); end

1148

1149 if ((intf.R3_risc[j] != intf.R3_arr[j+n]) || (intf.R3_risc[j] != intf.R3_g[k+n-w]) || (^intf.R3_risc[j] == l'bx)) begin
1150 $display ("R3 \%h \%h \%h *", intf.R3_risc[j], intf.R3_arr[j+n], intf.R3_g[k+n-w]);
1151 $fwrite (file_S, "R3 \%h \%h \%h \n", intf.R3_risc[j], intf.R3_arr[j+n], intf.R3_g[k+n-w]); end

1152

1153 else begin
1154 $display ("R3 \%h \%h \%h", intf.R3_risc[j], intf.R3_arr[j+n], intf.R3_g[k+n-w]);
1155 $fwrite (file_S, "R3 \%h \%h \%h\n", intf.R3_risc[j], intf.R3_arr[j+n], intf.R3_g[k+n-w]); end

1156

1158 $display ("R4 \%h \%h \%h *", intf.R4_risc[j], intf.R4_arr[j+n], intf.R4_g[k+n-w]);
1159 $fwrite (file_S, "R4 \%h \%h \%h \n", intf.R4_risc[j], intf.R4_arr[j+n], intf.R4_g[k+n-w]); end

1160
else begin
  $display("R4 \%h \%h \%h", intf.R4_risc[j], intf.R4_arr[j+n], intf.R4_g[k+n-w]);
  $fwrite(file_S, "R4 \%h \%h \%h\n", intf.R4_risc[j], intf.R4_arr[j+n], intf.R4_g[k+n-w]);
end

if ((intf.R5_risc[j] != intf.R5_arr[j+n]) || (intf.R5_risc[j] != intf.R5_g[k+n-w]) || (^intf.R5_risc[j] == 1'bX)) begin
  $display("R5 \%h \%h \%h *", intf.R5_risc[j], intf.R5_arr[j+n], intf.R5_g[k+n-w]);
  $fwrite(file_S, "R5 \%h \%h \%h*
", intf.R5_risc[j], intf.R5_arr[j+n], intf.R5_g[k+n-w]);
end else begin
  $display("R5 \%h \%h \%h", intf.R5_risc[j], intf.R5_arr[j+n], intf.R5_g[k+n-w]);
  $fwrite(file_S, "R5 \%h \%h \%h\n", intf.R5_risc[j], intf.R5_arr[j+n], intf.R5_g[k+n-w]);
end

  $display("R6 \%h \%h \%h *", intf.R6_risc[j], intf.R6_arr[j+n], intf.R6_g[k+n-w]);
  $fwrite(file_S, "R6 \%h \%h \%h*
", intf.R6_risc[j], intf.R6_arr[j+n], intf.R6_g[k+n-w]);
end
else begin

$display("R6 \%h \%h \%h", intf.R6_risc[j], intf.R6_arr[j+n], intf.R6_g[k+n-w]);

fwrite(file_S, "R6 \%h \%h \%h\n", intf.R6_risc[j], intf.R6_arr[j+n], intf.R6_g[k+n-w]); end

if ((intf.R7_risc[j] != intf.R7_arr[j+n]) || (intf.R7_risc[j] != intf.R7_g[k+n-w]) || (^intf.R7_risc[j] == 1'b0)) begin

$display("R7 \%h \%h \%h *", intf.R7_risc[j], intf.R7_arr[j+n], intf.R7_g[k+n-w]);

fwrite(file_S, "R7 \%h \%h \%h *\n", intf.R7_risc[j], intf.R7_arr[j+n], intf.R7_g[k+n-w]); end

else begin

$display("R7 \%h \%h \%h", intf.R7_risc[j], intf.R7_arr[j+n], intf.R7_g[k+n-w]);

fwrite(file_S, "R7 \%h \%h \%h\n", intf.R7_risc[j], intf.R7_arr[j+n], intf.R7_g[k+n-w]); end

// assigning the individual status bits to a variable and to display

st1 = intf.SR_risc[j][11]; st2 = intf.SR_risc[j][10]; st3 = intf.SR_risc[j][9]; st4 = intf.SR_risc[j][8];
stp1 = intf.SR_g[k+n−w][11]; stp2 = intf.SR_g[k+n−w][10]; stp3
    = intf.SR_g[k+n−w][9]; stp4 = intf.SR_g[k+n−w][8];

if ((intf.SR_risc[j] != intf.SR_arr[j+n]) || (intf.SR_risc[j]
    != intf.SR_g[k+n−w]) || (^intf.SR_risc[j] === 1'bxx)) begin
    $display("SR %h %h %h *", intf.SR_risc[j], intf.SR_arr[
        j+n], intf.SR_g[k+n−w]);
    $fwrite(file_S, "SR %h %h %h\n", intf.SR_risc[j], intf.
        SR_arr[j+n], intf.SR_g[k+n−w]); end

else begin
    $display("SR %h %h %h", intf.SR_risc[j], intf.SR_arr[j+n],
        intf.SR_g[k+n−w]);
    $fwrite(file_S, "SR %h %h %h\n", intf.SR_risc[j], intf.
        SR_arr[j+n], intf.SR_g[k+n−w]); end

$display("PC %3h %h %h", j, intf.PC_pl[m−w], intf.PC_pl[m−w]);

fwrite(file_S, "PC %3h %h %h\n", j, intf.PC_pl[m−w],
        intf.PC_pl[m−w]);

if ((intf.SR_risc[j] != intf.SR_arr[j+n]) || (intf.SR_risc[j]
    != intf.SR_g[k+n−w]) || (^intf.SR_risc[j] === 1'bxx)) begin
\$display ("Status Bits (processor): SR=\%h; C=\%d, N=\%d, V=\%d, Z=\%d", intf.SR_risc[j-w], st1, st2, st3, st4);
\$display ("Status Bits (expected): SR=\%4h; C=\%d, N=\%d, V=\%d, Z=\%d", intf.SR_g[k+n-w], stp1, stp2, stp3, stp4);
\$fwrite (file_S, "Status Bits (processor): SR=\%h; C=\%d, N=\%d, V=\%d, Z=\%d\n", intf.SR_risc[j-w], st1, st2, st3, st4);
\$fwrite (file_S, "Status Bits (expected): SR=\%4h; C=\%d, N=\%d, V=\%d, Z=\%d\n", intf.SR_g[k+n-w], stp1, stp2, stp3, stp4);
end
else begin
\$display ("Status Bits: C=\%d, N=\%d, V=\%d, Z=\%d", st1, st2, st3, st4);
\$fwrite (file_S, "Status Bits: C=\%d, N=\%d, V=\%d, Z=\%d\n", st1, st2, st3, st4); end

\$display ("
------------------------------------------------------------------------\n\n"
);
\$fwrite (file_S, "
------------------------------------------------------------------------\n\n"
);
end

$finish;
end
endtask
initial
#23000 \$finish;

endmodule
EOF

# to generate the interface according to the config file
sub print_intf
{
print out_intf "interface intf_risc (input clk);\\n\\n";
print out_intf <<EOF;
logic reset;
logic [\$bits_n:0] R [\$reg_n:0];
logic [\$bits_n:0] SR;
logic [\$bits_n:0] PC;
logic [\$bits_n:0] SP;
logic [\$bits_n:0] IR1;
logic [\$bits_n:0] IR2;
logic [\$bits_n:0] IR3;
logic [\$bits_n:0] PM_out;
logic [\$bits_n:0] DM_out;
logic [\$bits_n:0] DM_in;
logic [4:0] SW_pin;

logic [7:0] bits_risc; // bits of the processor
logic [7:0] no_reg; // no of registers available in processor
logic [7:0] load_risc; // to input the opcode used by processor for load (from config file)
logic [7:0] st_risc; // store
logic [7:0] jmp_risc; // jump
logic [7:0] cal_risc; // call
logic [7:0] ret_risc; // return

logic [2:0] del_ld; // to input the delay until next instruction fetch for: load
logic [2:0] del_st; // store
logic [2:0] del_jmp; // jump
logic [2:0] del_cal; // call
logic [2:0] del_ret; // return

logic [19:0] PC_proc [0:16384]; // to make sure the PC sequence is correct
logic [19:0] PC_pl [0:16384];
1268 logic [$bits_n:0] DM_out_intf [0:16384]; // added to keep track of two consecutive FFFFs
1269 logic [$bits_n:0] PM_out_intf [0:16384]; // added to keep track of two consecutive FFFFs
1270 EOF
1271
1272
1273 printf out_intf "\n";
1274 printf out_intf "// model\n";
1275 for ($i=0; $i<$reg; $i++)
1276 {
1277 printf out_intf "\tlogic [19:0] R$i";
1278 printf out_intf "_arr [0:16384];\n";
1279 }
1280 printf out_intf "\tlogic [19:0] SR_arr [0:16384];\n\n\n// generator\n";
1281
1282
1283 for ($i=0; $i<$reg; $i++)
1284 {
1285 printf out_intf "\tlogic [19:0] R$i";
1286 printf out_intf "_g [0:16384];\n";
1287 }
1288 printf out_intf "\tlogic [19:0] SR_g [0:16384];\n\n\n// processor\n";
V.1 Test bench generator

1289
1290 for ($i=0; $i<$reg; $i++)
1291 {
1292 print out_intf "\tlogic [$bits_n:0] R$i";
1293 print out_intf "_risc [0:16384];\n";
1294 }
1295 print out_intf <<EOF;
1296
1297
1298 logic [$bits_n:0] SR_risc [0:16384];
1299 logic [$bits_n:0] PC_risc [0:16384];
1300 logic [$bits_n:0] SP_risc [0:16384];
1301
1302 logic [$bits_n:0] IR1_risc [0:16384];
1303 logic [$bits_n:0] IR2_risc [0:16384];
1304 logic [$bits_n:0] IR3_risc [0:16384];
1305
1306 logic [$bits_n:0] DM_in_risc [0:16384];
1307 logic [$bits_n:0] DM_out_risc [0:16384];
1308 logic [$bits_n:0] mem_array [0:16384];
1309
1310 endinterface
1311
1312
1313 EOF
# to print into risc.sv.rtl file in etc folder

sub print_etc
{
print out_etc <<EOF;
+nowarnTFNPC
+nowarnIWFA
+nowarnSVTL
src/timescale.sv
src/interface.sv
src/scoreboard.sv
src/driver.sv
src/monitor.sv
src/env.sv
src/testcase.sv
EOF
if ($ext eq 'vhd')
{
print out_etc <<EOF;
src/4_ajf/220 pack.vhd
src/4_ajf/altera_mf_components.vhd
src/4_ajf/altera_mf.vhd
src/4_ajf/220 model.vhd
}
if ($arch == 0) #harvard
{
print_out_etc "src/$name_folder/$name_pm.$ext

src/$name_folder/$name_dm.$ext"
}
else #von neumann
{
print_out_etc "src/$name_folder/$name_pm.$ext"
}
if ($name_div ne '0' && $name_mul ne '0')
{
print_out_etc <<EOF;
src/$name_folder/$name_div.$ext
src/$name_folder/$name_mul.$ext
EOF
if ($name_cnt ne '0')
{
print_out_etc "src/$name_folder/$name_cnt.$ext"
}
if ($name_rot_l ne '0' && $ext eq 'v')
{
print_out_etc "src/$name_folder/$name_rot_l.$ext"
}
if ($name_rot_r ne '0' && $ext eq 'v')
{
print_out_etc "src/$name_folder/$name_rot_r.$ext"
}
if ($name_rot_c ne '0' && $ext eq 'v')
{
print_out_etc "src/$name_folder/$name_rot_c.$ext"
}
if ($name_mem_mux ne '0')
1363  \{ print out etc "src/$name_folder/$name_mem_mux.$ext\n"; \}
1364  if ($name_shift_mux ne '0' && $ext eq 'v')
1365  \{ print out etc "src/$name_folder/$name_shift_mux.$ext\n"; \}
1366  if ($name_shift_arth ne '0' && $ext eq 'v')
1367  \{ print out etc "src/$name_folder/$name_shift_arth.$ext\n"; \}
1368  if ($name_shift_log ne '0' && $ext eq 'v')
1369  \{ print out etc "src/$name_folder/$name_shift_log.$ext\n"; \}
1370  if ($name_add_sub ne '0')
1371  \{ print out etc "src/$name_folder/$name_add_sub.$ext\n"; \}
1372  print out etc <<EOF;
1373  src/$name_folder/$name.$ext
1374  src/risc_test.sv
1375  //
1376  //Add source files above
1377  //
1378  //
1379  //Uncomment for TSMC 180nm
1380  -v .. /maiee/lib/tsmc-0.18/verilog/tsmc18.v
1381  //
1382  //Uncomment for TSMC 65nm
1383  // -v /classes/ee620/maiee/lib/synopsys/TSMC_tcbc65/TSMCHOME/
1384  //   digital/Front_End/verilog/tcbn65lp_200a/tcbn65lp.v
1385  //
1386  //Uncomment for SAED 90nm

V.1 Test bench generator
// Uncomment for SAED 32nm
//
SAED32_EDK/lib/stdcell_rvt/verilog/saed32nm.v

//
// add altera component libraries

–v/tools/altera/9.0/quartus/eda/sim_lib/220model.v
–v/tools/altera/9.0/quartus/eda/sim_lib/altera_mf.v

//
// librescan

//
EOF

# to print the content of driver into driver.sv

sub print_drv
{

print out_drv<<EOF;

class driver;

scoreboard sb;
}
V.1 Test bench generator

```plaintext
1409  integer  j=0, k=0, c, file_0, fp, rp;
1410  integer  len, i, z=0;
1411  string  line, temp;
1412  integer  limit;
1413
1414  virtual  intf_risc  intf;
1415  function  new(virtual  intf_risc  intf, scoreboard  sb);
1416     this.intf = intf;
1417     this.sb = sb;
1418     endfunction
1419
1420  task  reset;
1421  begin
1422     intf.reset = 1'b0;
1423     @(negedge  intf.clk);
1424     intf.reset = 1'b1;
1425     sb.count_stopsim = 0;
1426  end
1427  endtask
1428
1429
1430  task  config_fileopen();
1431  begin
1432
1433     fp = $fopen("../configuration.txt", "r");
```

if (fp == 0)
begin
$display("Error: can not open the file");
$finish;
end

while (!$feof(fp))
begin
if ($fgets(line, fp))
begin

len = line.len();

for (i = 0; i < len; i++)
begin
if (line.substr(i, i+5-1) == "bits:")
begin

temp = line.substr(7,8);

intf.bits_risc = temp.atohex();

intf.bits_risc = intf.bits_risc -1;

if ((intf.bits_risc+1) == 12)
limit = 12'hFFF;
else if ((intf.bits_risc+1) == 14)
limit = 14'h3FFF;
else if ((intf.bits_risc+1) == 16)
1459   limit = 16'hFFFF;
1460   end
1461
1462   if ( line.substr(i, i+10-1) =="registers:" )
1463     begin
1464       temp = line.substr(12,13);
1465       intf.no_reg = temp.atohex();
1466       intf.no_reg = intf.no_reg−1;
1467       //\$display(intf.no_reg);
1468     end
1469
1470   if ( line.substr(i, i+5-1) =="LOAD:" ) // strip the line off white spaces*
1471     begin
1472       temp = line.substr(7,8);
1473       intf.load_risc = temp.atohex();
1474       //\$display(intf.load_risc);
1475     end
1476
1477   if ( line.substr(i, i+6-1) =="STORE:" )
1478     begin
1479       temp = line.substr(8,9);
1480       intf.st_risc = temp.atohex();
1481       //\$display(intf.st_risc);
1482     end
if (line.substr(i, i+5-1) == "JUMP:")
begin
  temp = line.substr(7, 8);
  intf.jmp_risc = temp.atohex();
  //\$display(intf.jmp_risc);
end

if (line.substr(i, i+5-1) == "CALL:")
begin
  temp = line.substr(7, 8);
  intf.cal_risc = temp.atohex();
  //\$display(intf.cal_risc);
end

if (line.substr(i, i+4-1) == "RET:")
begin
  temp = line.substr(6, 7);
  intf.ret_risc = temp.atohex();
  //\$display(intf.ret_risc);
end

if (line.substr(i, i+7-1) == "del ld:")
begin
  temp = line.substr(7, 8);
intf.del_ld = temp.atoi();
//\$display(intf.del_ld);
end

if ( line.substr(i, i+7-1) =="del_st:"
begin
temp = line.substr(7,8);
intf.del_st = temp.atoi();
//\$display(intf.del_st);
end

if ( line.substr(i, i+8-1) =="del_jmp:"
begin
temp = line.substr(8,9);
intf.del_jmp = temp.atoi();
//\$display(intf.del_jmp);
end

if ( line.substr(i, i+8-1) =="del_cal:"
begin
temp = line.substr(8,9);
intf.del_cal = temp.atoi();
//\$display(intf.del_cal);
end
if (line.substr(i, i+8-1) == "del_ret:")
begin
  temp = line.substr(8, 9);
  intf.del_ret = tempatoi();
  //\$display(intf.del_ret);
end
der
declose(fp);

dost
endost
EOF

if ($arch == 0) #harvard
{
  print out_drv<<EOF;
task check; //when the simulation sees two FFF's in the
            instruction, it stops simulation
forever
begin
  @(posedge intf.clk)
begin
intf.PM_out_intf[z] = intf.PM_out;

begin
    sb.count_stopsim = sb.count_stopsim+1;
end
if (sb.count_stopsim == 1)
begin
    @(posedge intf.clk);
    $display("\n\n—— Simulation complete with no errors !——\n\n");
    $finish;
end
end

z = z+1;
endtask
endclass
EOF
else #von neumann
{
    print outDrv<<EOF;
    task check; //when the simulation sees two FFF's in the
    instruction, it stops simulation
    forever
    begin
    @(posedge intf.clk)
    begin
    intf.DM_out_intf[z] = intf.DM_out;

    if ((intf.DM_out_intf[z] == limit) && (intf.DM_out_intf[z−1] ==
    limit) && (intf.DM_out_intf[z−2] == limit) && (intf.
    DM_out_intf[z−3] == limit) && (intf.DM_out_intf[z−4] == limit
    ) && (intf.DM_out_intf[z−5] == limit) && (intf.DM_out_intf[z
    −6][p:$q] != 6'h$ret ) )
    begin
    sb.count_stopsim = sb.count_stopsim+1;
    end
    if (sb.count_stopsim == 1)
    begin
    @(posedge intf.clk);
    $display("\n\n——Simulation complete with no errors
    !———\n\n") ;
\$finish;
end
end
ez = z+1;
end
endtask
class
endclass
EOF

V.2 Interface

```verilog
interface intf_risc (input clk);

logic reset;
logic [11:0] R [7:0];
logic [11:0] SR;
logic [11:0] PC;
logic [11:0] SP;
logic [11:0] IR1;
logic [11:0] IR2;
logic [11:0] IR3;
logic [11:0] PM_out;
logic [11:0] DM_out;
logic [11:0] DM_in;
logic [4:0] SW_pin;

logic [7:0] bits_risc; // bits of the processor
logic [7:0] no_reg;   // no of registers available in processor
logic [7:0] load_risc; // to input the opcode used by processor for load (from config file)
logic [7:0] st_risc;  // store
logic [7:0] jmp_risc; // jump
```
logic [7:0] cal_risc; // call
logic [7:0] ret_risc; // return

logic [2:0] del_ld;  // to input the delay until next
  instruction fetch for: load
logic [2:0] del_st;  // store
logic [2:0] del_jmp; // jump
logic [2:0] del_cal; // call
logic [2:0] del_ret; // return

logic [19:0] PC_proc [0:16384]; // to make sure the PC
  sequence is correct
logic [19:0] PC_pl [0:16384];

logic [11:0] DM_out_intf [0:16384]; // added to keep track of
  two consecutive FFFFs
logic [11:0] PM_out_intf [0:16384]; // added to keep track of
  two consecutive FFFFs

// model
logic [19:0] R0_arr [0:16384];
logic [19:0] R1_arr [0:16384];
logic [19:0] R2_arr [0:16384];
logic [19:0] R3_arr [0:16384];
logic [19:0] R4_arr [0:16384];
V.2 Interface

// generator

 logic [19:0] R0_g [0:16384];
 logic [19:0] R1_g [0:16384];
 logic [19:0] R2_g [0:16384];
 logic [19:0] R3_g [0:16384];
 logic [19:0] R4_g [0:16384];
 logic [19:0] R5_g [0:16384];
 logic [19:0] R6_g [0:16384];
 logic [19:0] R7_g [0:16384];
 logic [19:0] SR_g [0:16384];

// processor

 logic [11:0] R0_risc [0:16384];
 logic [11:0] R1_risc [0:16384];
 logic [11:0] R2_risc [0:16384];
 logic [11:0] R3_risc [0:16384];
 logic [11:0] R4_risc [0:16384];
 logic [11:0] R5_risc [0:16384];
V.2 Interface

68  logic [11:0] R6_risc [0:16384];
69  logic [11:0] R7_risc [0:16384];
70
71
72  logic [11:0] SR_risc [0:16384];
73  logic [11:0] PC_risc [0:16384];
74  logic [11:0] SP_risc [0:16384];
75
76  logic [11:0] IR1_risc [0:16384];
77  logic [11:0] IR2_risc [0:16384];
78  logic [11:0] IR3_risc [0:16384];
79
80  logic [11:0] DM_in_risc [0:16384];
81  logic [11:0] DM_out_risc [0:16384];
82  logic [11:0] mem_array [0:16384];
83
84  endinterface
class driver;

scoreboard sb;

integer j=0, k=0, c, file_0, fp, rp;

integer len, i, z=0;

string line, temp;

integer limit;


virtual intf_risc intf;

function new(virtual intf_risc intf, scoreboard sb);

dropdown

task reset;

begin

intf.reset = 1'b0;

@negedge intf.clk ;

intf.reset = 1'b1;

sb.count_stopsim = 0;

end
dropdown
task config_fileopen();
begin

fp = $fopen("./configuration.txt", "r");
if (fp == 0)
begin
$display("Error: can not open the file");
$finish;
end

while(!$feof(fp))
begin
if ($fgets(line, fp))
begin

len = line.len();

for (i = 0; i < len; i++)
begin
if ( line.substr(i, i+5-1) =="bits:" )
begin
 temp = line.substr(7,8);
 intf.bits_risc = temp.atohex();
 intf.bits_risc = intf.bits_risc -1;
if (((intf.bits_risc+1) == 12)
limit = 12'hFFF;
else if ((intf.bits_risc+1) == 14)
  limit = 14'h3FFF;
else if ((intf.bits_risc+1) == 16)
  limit = 16'hFFFF;
end

if (line.substr(i, i+10−1) == "registers:")
begin
  temp = line.substr(12,13);
  intf.no_reg = temp.atohex();
  intf.no_reg = intf.no_reg−1;
  // $display(intf.no_reg);
end

if (line.substr(i, i+5−1) == "LOAD:") // strip the line off white spaces*
begin
  temp = line.substr(7,8);
  intf.load_risc = temp.atohex();
  // $display(intf.load_risc);
end

if (line.substr(i, i+6−1) == "STORE:")
begin
73 temp = line.substr(8,9);
74 intf.st_risc = temp.atohex();
75 // $display(intf.st_risc);
76 end
77
78 if ( line.substr(i, i+5-1) == "JUMP:" )
79 begin
80 temp = line.substr(7,8);
81 intf.jmp_risc = temp.atohex();
82 // $display(intf.jmp_risc);
83 end
84
85 if ( line.substr(i, i+5-1) == "CALL:" )
86 begin
87 temp = line.substr(7,8);
88 intf.cal_risc = temp.atohex();
89 // $display(intf.cal_risc);
90 end
91
92 if ( line.substr(i, i+4-1) == "RET:" )
93 begin
94 temp = line.substr(6,7);
95 intf.ret_risc = temp.atohex();
96 // $display(intf.ret_risc);
97 end
if ( line.substr(i, i+7-1) == "del_ld:" )
begin
    temp = line.substr(7,8);
    intf.del_ld = tempatoi();
    // $display(intf.del_ld);
end

if ( line.substr(i, i+7-1) == "del_st:" )
begin
    temp = line.substr(7,8);
    intf.del_st = tempatoi();
    // $display(intf.del_st);
end

if ( line.substr(i, i+8-1) == "del_jmp:" )
begin
    temp = line.substr(8,9);
    intf.del_jmp = tempatoi();
    // $display(intf.del_jmp);
end

if ( line.substr(i, i+8-1) == "del_cal:" )
begin
    temp = line.substr(8,9);

```
intf.del_cal = tempatoi();

// $display(intf.del_cal);
end

if (line.substr(i, i+8-1) == "del_ret:")
begin

temp = line.substr(8,9);
intf.del_ret = tempatoi();

// $display(intf.del_ret);
end
end

$fclose(fp);

end
endtask
task check; // when the simulation sees two FFF's in the
 instruction, it stops simulation

forever
begin
begin

@posedge intf.clk)
begin

V.3 Driver

```vhdl
147  intf.DM_out_intf[z] = intf.DM_out;
148
150    begin
151      sb.count_stopsim = sb.count_stopsim + 1;
152    end
153  if (sb.count_stopsim == 1)
154    begin
155      @(posedge intf.clk);
156      $display("\n\n−−−−−−−Simulation complete with no errors
       !−−−−−−−−\n\n");
157      $finish;
158    end
159  end
160  z = z + 1;
161  end
162 endtask
163
164 endclass
```
# V.4 Environment

```plaintext
class environment;
driver drv;
    scoreboard sb;
    monitor mntr;
    virtual intf_risc intf;

function new(virtual intf_risc intf);
    this.intf = intf;
    sb = new();
    drv = new(intf, sb);
    mntr = new(intf, sb);

    // fork
    //      drv.check();
    //      join_none

endfunction

endclass
```
### V.5 Compilation Unit

1. +nowarnTFNPC
2. +nowarnIWFA
3. +nowarnSVTL

4. `src/timescale.v`
5. `src/interface.sv`
6. `src/scoreboard.sv`
7. `src/driver.sv`
8. `src/monitor.sv`
9. `src/env.sv`
10. `src/testcase.sv`
11. `src/2_kxm/kxmRISC621_ram1.v`
12. `src/2_kxm/kxmRISC_div.v`
13. `src/2_kxm/kxmRISC_mult.v`
14. `src/2_kxm/kxm621_count.v`
15. `src/2_kxm/kxmRISC621_v.v`
16. `src/risc_test.sv`

17. /*
18. // Add source files above
19. /*
20. /*
21. // Uncomment for TSMC 180nm
22. --v ../maieee/lib/tsmc-0.18/verilog/tsmc18.v
23. /*
// Uncomment for TSMC 65nm
-vg /classes/ee620/maieee/lib/synopsys/TSMC_tcbc65/TSMCHOME/
digital/Front_End/verilog/tcbn65lp_200a/tcbn65lp.v

// Uncomment for SAED 90nm
-vy /classes/ee620/maieee/lib/synopsys/SAED_EDK90nm/
   Digital_Standard_Cell_Library/verilog/

// Uncomment for SAED 32nm
-vy /classes/ee620/maieee/lib/synopsys/SAED_EDK32-28nm/
   SAED32_EDK/lib/stdcell_rvt/verilog/saed32nm.v

// add altera component libraries
-v /tools/altera/9.0/quartus/eda/sim_lib/220model.v
-v /tools/altera/9.0/quartus/eda/sim_lib/altera_mf.v

//
+librescan
//
V.6 Test case

program testcase(intf_risc intf);
environment env = new(intf);

initial
begin
env.drvr.config_fileopen();
env.drvr.reset();
env.drvr.check();

#1000;

end
endprogram
V.7 Test

```vhdl
/*
 * Author: Namratha
 * Rochester, NY, USA
 */

'timescale 1ns / 1ns

module test;

reg clk = 0;
reg reset;
reg scan_in0, scan_en, test_mode;
wire scan_out0;

intf_risc intf(clk); // instantiating interface
testcase test_01(intf); // instantiating testcase

integer w, i=0, b, j=0, p=0, m=0, k=0, n=0, t, occ=0,
    print_flag=0, file_P, file_Q, file_R, file_S, file_T,
```
jmp_flag = 0, g = 0, flag = 0, flagSR = 0;

reg st1, stp1, st2, stp2, st3, stp3, st4, stp4;
reg [11:0] ir3_temp, ir2_temp;
reg [11:0] PC_tmp;
reg [11:0] PC_j = 12'h000;

initial
forever #10 clk = ~clk;

kxmRISC621_v top (intf.reset,
intf.clk,
intf.R,
intf.SR,
intf.FC,
intf.SP,
intf.IR1,
intf.IR2,
intf.IR3,
intf.DM_out,
intf.DM_in,
intf.SW_pin
);

// reading the memory.t file into the array mem_data of ram

component altsyncram in dut
initial $readmemh("../memory.t", test.top.my_ram.
           altsyncram_component.mem_data);

// reading the memory.t file into an array mem_array for use in
    risc_test.sv file
initial $readmemh("../memory.t", intf.mem_array);

initial
begin
  // gen
  $readmemh("../pl/R0.t", intf.R0_g);
  $readmemh("../pl/R1.t", intf.R1_g);
  $readmemh("../pl/R2.t", intf.R2_g);
  $readmemh("../pl/R3.t", intf.R3_g);
  $readmemh("../pl/R4.t", intf.R4_g);
  $readmemh("../pl/R5.t", intf.R5_g);
  $readmemh("../pl/R6.t", intf.R6_g);
  $readmemh("../pl/R7.t", intf.R7_g);
  $readmemh("../pl/SR.t", intf.SR_g);
  $readmemh("../pl/PC.t", intf.PC_pl);

  // model
  $readmemh("../processor/R0.t", intf.R0_arr);
  $readmemh("../processor/R1.t", intf.R1_arr);
  $readmemh("../processor/R2.t", intf.R2_arr);
$readmemh("../processor/R3.t", intf.R3_arr);
$readmemh("../processor/R4.t", intf.R4_arr);
$readmemh("../processor/R5.t", intf.R5_arr);
$readmemh("../processor/R6.t", intf.R6_arr);
$readmemh("../processor/R7.t", intf.R7_arr);
$readmemh("../processor/SR.t", intf.SR_arr);
end

initial

begin
file_P = $fopen("PC.t", "w");
file_R = $fopen("R.t", "w");
file_Q = $fopen("log.t", "w");
file_S = $fopen("cplog.t", "a");
file_T = $fopen("opcode.t", "a");
fwrite(file_R, "Processor, Model, Generator\n");
fwrite(file_R, "PC: \tR0\t\tR1\t\tR2\t\tR3\t\tR4\t\tR5\t\tR6\t\tR7\t\tSR\t\tPC\n"); // to put all the signals in an array

for (b=-1; b<=3; b=b+1)
begin
92 @(posedge intf.clk);
93 begin
94 intf.IR1_risc[b] = intf.IR1;
95 if ((intf.load_risc == intf.IR1_risc[b][11:6]) || (intf.st_risc == intf.IR1_risc[b][11:6]) || (intf.jmp_risc == intf.IR1_risc[b][11:6]) || (intf.cal_risc == intf.IR1_risc[b][11:6]) || (intf.ret_risc == intf.IR1_risc[b][11:6]))
96 PC_tmp = intf.PC;
97 end
98 end
99 t = 4;
100
101 ir3_temp = intf.IR3;
102 ir2_temp = intf.IR2;
103 intf.IR3_risc[0] = ir3_temp;
104 intf.IR2_risc[0] = ir2_temp;
105
106 g = g+1;
107
108 forever
109 @(posedge intf.clk)
110 begin
111
112 intf.R0_risc[j] = intf.R[0];
113 intf.R1_risc[j] = intf.R[1];
\texttt{114 intf.R2\_risc[j] = intf.R[2];}
\texttt{115 intf.R3\_risc[j] = intf.R[3];}
\texttt{117 intf.R5\_risc[j] = intf.R[5];}
\texttt{118 intf.R6\_risc[j] = intf.R[6];}
\texttt{119 intf.R7\_risc[j] = intf.R[7];}
\texttt{120 intf.SR\_risc[j] = intf.SR;}
\texttt{121 intf.PC\_risc[j] = intf.PC;}
\texttt{122 intf.SP\_risc[j] = intf.SP;}
\texttt{123}
\texttt{124 intf.IR1\_risc[t] = intf.IR1;}
\texttt{125 intf.IR2\_risc[g] = intf.IR2;}
\texttt{126 intf.IR3\_risc[g] = intf.IR3;}
\texttt{127}
\texttt{128 intf.DM\_in\_risc[j] = intf.DM\_in;}
\texttt{129 intf.DM\_out\_risc[j] = intf.DM\_out;}
\texttt{130}
\texttt{131}
\texttt{132 if ((intf.load\_risc == intf.IR1\_risc[t][11:6]) || (intf.st\_risc}
\texttt{ == intf.IR1\_risc[t][11:6]) || (intf.jmp\_risc == intf.IR1\_risc[t}
\texttt{][11:6]) || (intf.cal\_risc == intf.IR1\_risc[t]
\texttt{][11:6]) || (intf.ret\_risc == intf.IR1\_risc[t][11:6]))}
\texttt{133 PC\_tmp = intf.PC;}
\texttt{134}
\texttt{135}
// checking for stalls and accordingly adjusting matching PC of processor and model for matching

if (occ == 0)
    begin
        if (intf.load_risc == intf.IR3_risc[g-1][11:6])
            begin
                n = n+1; occ = occ+1;
            end
    else if (intf.st_risc == intf.IR3_risc[g-1][11:6])
            begin
                n = 0; occ = occ+1;
            end
    else if (intf.jmp_risc == intf.IR3_risc[g-1][11:6])
            begin
                n = 0; occ = occ+1;
            end
    else if (intf.cal_risc == intf.IR3_risc[g-1][11:6])
            begin
                n = 0; occ = occ+1;
            end
    else if (intf.ret_risc == intf.IR3_risc[g-1][11:6])
            begin
                n = 0; occ = occ+1;
            end
    end
else begin
  if (jmp_flag == 0)
    print_flag = 0;
  n = 0;
  occ = 0;
end

if (((intf.jmp_risc == intf.IR3_risc[g-3][11:6]) || (intf.load_risc == intf.IR3_risc[g-3][11:6]) || (intf.st_risc == intf.IR3_risc[g-3][11:6]) || (intf.cal_risc == intf.IR3_risc[g-3][11:6]) || (intf.ret_risc == intf.IR3_risc[g-3][11:6]))
    && (intf.IR3_risc[g-2][11:6] == 6'h3F))
begin
  jmp_flag = 0;
  jmp_flag = jmp_flag + 1;
  print_flag = 1;
end

else if ((intf.ret_risc == intf.IR3_risc[g-2][11:6]) && (intf.IR3_risc[g-1][11:6] == 6'h3F))
begin
  jmp_flag = 0;
  jmp_flag = jmp_flag + 1;
print_flag = 1;

end

else

begin

if (jump_flag == 1)

begin

if ((intf.load_risc == intf.IR3_risc[g][11:6]) || (intf.st_risc == intf.IR3_risc[g][11:6]) || (intf jmp_risc == intf.IR3_risc[g][11:6]) || (intf.cal_risc == intf.IR3_risc[g][11:6]) || (intf.ret_risc == intf.IR3_risc[g][11:6]))

j = PC_tmp - 2;

else if ((intf.load_risc == intf.IR2_risc[g][11:6]) || (intf.st_risc == intf.IR2_risc[g][11:6]) || (intf jmp_risc == intf.IR2_risc[g][11:6]) || (intf.cal_risc == intf.IR2_risc[g][11:6]) || (intf.ret_risc == intf.IR2_risc[g][11:6]) && (intf.IR2_risc[g-1][11:6] != 6'h3F) && (intf.IR2_risc[g-2][11:6] == 6'h3F))

j = PC_tmp - 3;

else

j = intf.PC - 4;

jmp_flag = 0;

print_flag = 1;

end

else
\begin{verbatim}
begin
print_flag = 0;
end

if (print_flag == 0)
begin
    // printing the register (gen purpose and status) values in the file R.t
    fwrite(file_R, " %h, %h, %h ;	", j, intf.PC_pl[m], intf.PC_pl[m]);
    if ( (j != intf.PC_pl[m]) )
begin
    fwrite(file_R,"*"); PC_j=j; flag=1; print_con; end

    // to print the opcodes in a file (if there is a PC error dont print the next instruction)
    if (flag == 1)
fwrite (file_T, "\n");
else
fwrite (file_T, "%2h\n", intf.mem_array[j][11:6]);

fwrite(file_R,"@%h: %h, %h;\t", j, intf.R0_risc[j], intf.R0_arr[j+n], intf.R0_g[k+n]); //add feature that will enable multiple stalls
\end{verbatim}
```c
if ( ( intf.R0_risc[j] != intf.R0_arr[j+n]) || (intf.R0_risc[j] != intf.R0_g[k+n]) || (^intf.R0_risc[j] === 1'b) )
begin $fwrite (file_R,"*"); p=1;
if (^intf.R0_risc[j] === 1'b) flagSR = 2;
print_con; end

fwrite (file_R, " %h, %h, %h;\t", intf.R1_risc[j], intf.R1_arr[j+n], intf.R1_g[k+n]);
if ( ( intf.R1_risc[j] != intf.R1_arr[j+n]) || (intf.R1_risc[j] != intf.R1_g[k+n]) || (^intf.R1_risc[j] === 1'b) )
begin $fwrite (file_R,"*"); p=1;
if (^intf.R1_risc[j] === 1'b) flagSR = 2;
print_con; end

fwrite (file_R, " %h, %h, %h;\t", intf.R2_risc[j], intf.R2_arr[j+n], intf.R2_g[k+n]);
if ( ( intf.R2_risc[j] != intf.R2_arr[j+n]) || (intf.R2_risc[j] != intf.R2_g[k+n]) || (^intf.R2_risc[j] === 1'b) )
begin $fwrite (file_R,"*"); p=1;
if (^intf.R2_risc[j] === 1'b) flagSR = 2;
print_con; end

fwrite (file_R, " %h, %h, %h;\t", intf.R3_risc[j], intf.R3_arr[j+n], intf.R3_g[k+n]);
```
if ( (intf.R3_risc[j] != intf.R3_arr[j+n]) || (intf.R3_risc[j] != intf.R3_g[k+n]) || (^intf.R3_risc[j] !== 1'bx) )

begin $fwrite(file_R,"*"); p=1;
if (^intf.R3_risc[j] === 1'bx) flagSR = 2;
print_con; end

fwrite(file_R, " %h, %h, %h;\t", intf.R4_risc[j], intf.R4_arr[j+n], intf.R4_g[k+n]);


begin $fwrite(file_R,"*"); p=1;
if (^intf.R4_risc[j] === 1'bx) flagSR = 2;
print_con; end

fwrite(file_R, " %h, %h, %h;\t", intf.R5_risc[j], intf.R5_arr[j+n], intf.R5_g[k+n]);

if ( (intf.R5_risc[j] != intf.R5_arr[j+n]) || (intf.R5_risc[j] != intf.R5_g[k+n]) || (^intf.R5_risc[j] !== 1'bx) )

begin $fwrite(file_R,"*"); p=1;
if (^intf.R5_risc[j] === 1'bx) flagSR = 2;
print_con; end

fwrite(file_R, " %h, %h, %h;\t", intf.R6_risc[j], intf.R6_arr[j+n], intf.R6_g[k+n]);
begin $fwrite(file_R,"*"); p=1;
if(^intf.R6_risc[j] === 1’bx) flagSR = 2;
print_con; end

fwrite(file_R, " %h, %h, %h; \t", intf.R7_risc[j], intf.R7_arr[j+n], intf.R7_g[k+n]);
begin $fwrite(file_R,"*"); p=1;
if(^intf.R7_risc[j] === 1’bx) flagSR = 2;
print_con; end

fwrite(file_R, " %h, %h, %h; \t", intf.SR_risc[j], intf.SR_arr[j+n], intf.SR_g[k+n]);
begin $fwrite(file_R,"*"); flagSR = (flagSR == 2) ? flagSR : 1;
print_con; end

fwrite(file_P, "@%h j=%h \n",k,j);
k = k+1;
fwrite(file_R,"\n\n");
if (j != intf.PC_pl[m])
begin
fwrite(file_Q, "@%h, perl= %h, testbench= %h\n", m, intf.PC_pl[m], j);
end
m=m+1;
end

j = j+1; g = g+1; t = t+1;
end
end
task print_con;
begint
for (w=4; w>=0; w=w-1)
begin
if (((w == 0) && (flag != 1)) // all errors except PC error
begin
if (flagSR == 1)
$display ("A Mismatch Occurred S @%4h! kxmRISC621_v", intf.
PC_pl[m-w]);
else if (flagSR == 2)
$display ("A Mismatch Occurred R @%4h! kxmRISC621_v", intf.
PC_pl[m-w]);
else
293 $display ("A Mismatch Occurred @%4h! kxmRISC621_v", intf.PC_pl[m-w]);

294

295 $display ("\tTime :%t ps\n",$time);

296

297 if (flagSR == 1)
298 $fwrite (file_S, "A Mismatch Occurred S @%4h! kxmRISC621_v",
           intf.PC_pl[m-w]);
299 else if (flagSR == 2)
300 $fwrite (file_S, "A Mismatch Occurred R @%4h! kxmRISC621_v",
            intf.PC_pl[m-w]);
301 else
302 $fwrite (file_S, "A Mismatch Occurred @%4h! kxmRISC621_v", intf
            .PC_pl[m-w]);
303
304 $fwrite (file_S, "\tTime :%t ps\n",$time);

305
306
307 p = 0;
308 end
309
310 if ((w == 0) && (flag == 1)) // when PC error is present
311 begin
312 $display ("A Mismatch Occurred PC @%4h! kxmRISC621_v", intf.
            PC_pl[m-w]);
313 $display (" Time :%t ps\n",$time);
$fwrite (file_S, "A Mismatch Occurred PC @%4h! kxmRISC621_v", intf.PC_pl[m-w]);

$fwrite (file_S, " Time :%t ps\n", $time);

if (intf.mem_array[j][11:6] == intf.ret_risc) begin
  $display("Instruction @%4h : %h", intf.PC_pl[m-1], intf.
           mem_array[j]);
  $fwrite (file_S, " Instruction @%4h : %h\n", intf.PC_pl[m-1],
           intf.mem_array[j]);
  end
else begin
  $display("Instruction @%4h : %h", intf.PC_pl[m-2], intf.
           mem_array[j-1]);
  $fwrite (file_S, " Instruction @%4h : %h\n", intf.PC_pl[m-2],
           intf.mem_array[j-1]);
  end

$display('′−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−′)

$display("Processor Model Generator\n");

$fwrite (file_S, "
                                                                                          \n");

$fwrite (file_S, " Processor Model Generator\n");


j = intf.PC_pl[m-w];

$display("PC %4h %h %h *");PC_j, intf.PC_pl[m-w], intf.PC_pl[m-w]);

fwrite (file_S, "PC %4h %h %h *
", PC_j, intf.PC_pl[m-w],
intf.PC_pl[m-w]);

$display("-------------");

fwrite (file_S, "
-------------

");

$display("\n");

fwrite (file_S, "\n");

p = 0;

$finish;

end

j = intf.PC_pl[m-w];

$display("\tInstruction @%4h : %h", intf.PC_pl[m-w], intf.
mem_array[j]);

$display("-------------");

$display(" Processor Model Generator\n");

fwrite (file_S, "\tInstruction @%4h : %h\n", intf.PC_pl[m-w],
intf.mem_array[j]);
$\texttt{fwrite}\ (\texttt{file\_S},\ "\n")$

$\texttt{fwrite}\ (\texttt{file\_S},\ "\ Processor\ Model\ Generator\n")$

if ($(\texttt{intf}\.\texttt{mem\_array}[j][11:6] == \texttt{intf}\.\texttt{load\_risc}) \&\& (\texttt{intf}\.\texttt{mem\_array}[j-1][11:6]!\neq \texttt{intf}\.\texttt{cal\_risc}) \&\& (\texttt{intf}\.\texttt{mem\_array}[j-1][11:6]!\neq \texttt{intf}\.\texttt{jump\_risc}) \&\& (\texttt{intf}\.\texttt{mem\_array}[j-1][11:6]!\neq \texttt{intf}\.\texttt{store\_risc}) \&\& (\texttt{intf}\.\texttt{mem\_array}[j-1][11:6]!\neq \texttt{intf}\.\texttt{load\_risc})) \mid (j==0) \&\& (\texttt{intf}\.\texttt{mem\_array}[j][11:6] == \texttt{intf}\.\texttt{load\_risc}))$

n=1; \text{ else } n=0;

if ($(\texttt{intf}\.\texttt{R0\_risc}[j] \neq \texttt{intf}\.\texttt{R0\_arr}[j+n]) \mid (\texttt{intf}\.\texttt{R0\_risc}[j] \neq \texttt{intf}\.\texttt{R0\_g}[k+n-w]) \mid (^{\texttt{intf}\.\texttt{R0\_risc}[j] \neq 1'bx})$) begin

$\texttt{display}\ ("R0 \%h \%h \%h *", \texttt{intf}\.\texttt{R0\_risc}[j], \texttt{intf}\.\texttt{R0\_arr}[j+n], \texttt{intf}\.\texttt{R0\_g}[k+n-w]);$

$\texttt{fwrite}\ (\texttt{file\_S},\ "R0 \%h \%h \%h *\n", \texttt{intf}\.\texttt{R0\_risc}[j], \texttt{intf}\.\texttt{R0\_arr}[j+n], \texttt{intf}\.\texttt{R0\_g}[k+n-w]);$ end

else begin

$\texttt{display}\ ("R0 \%h \%h \%h", \texttt{intf}\.\texttt{R0\_risc}[j], \texttt{intf}\.\texttt{R0\_arr}[j+n], \texttt{intf}\.\texttt{R0\_g}[k+n-w]);$

$\texttt{fwrite}\ (\texttt{file\_S},\ "R0 \%h \%h \%h\n", \texttt{intf}\.\texttt{R0\_risc}[j], \texttt{intf}\.\texttt{R0\_arr}[j+n], \texttt{intf}\.\texttt{R0\_g}[k+n-w]);$ end


if (( intf.R1_risc[j] != intf.R1_arr[j+n]) || (intf.R1_risc[j] != intf.R1_g[k+n-w]) || (^intf.R1_risc[j] === 1'bx)) begin
  $display ("R1 %h %h %h *", intf.R1_risc[j], intf.R1_arr[j+n], intf.R1_g[k+n-w]);
  $fwrite (file_S, "R1 %h %h %h *
", intf.R1_risc[j], intf.R1_arr[j+n], intf.R1_g[k+n-w]); end
else begin
  $display ("R1 %h %h %h", intf.R1_risc[j], intf.R1_arr[j+n], intf.R1_g[k+n-w]);
  $fwrite (file_S, "R1 %h %h %h\n", intf.R1_risc[j], intf.R1_arr[j+n], intf.R1_g[k+n-w]); end

if (( intf.R2_risc[j] != intf.R2_arr[j+n]) || (intf.R2_risc[j] != intf.R2_g[k+n-w]) || (^intf.R2_risc[j] === 1'bx)) begin
  $display ("R2 %h %h %h *", intf.R2_risc[j], intf.R2_arr[j+n], intf.R2_g[k+n-w]);
  $fwrite (file_S, "R2 %h %h %h *
", intf.R2_risc[j], intf.R2_arr[j+n], intf.R2_g[k+n-w]); end
else begin
  $display ("R2 %h %h %h", intf.R2_risc[j], intf.R2_arr[j+n], intf.R2_g[k+n-w]);
  $fwrite (file_S, "R2 %h %h %h\n", intf.R2_risc[j], intf.R2_arr[j+n], intf.R2_g[k+n-w]); end
if ((intf.R3_risc[j] != intf.R3_arr[j+n]) || (intf.R3_risc[j] != intf.R3_g[k+n-w]) || (^intf.R3_risc[j] === 1'b0)) begin
  display("R3 %h %h %h *", intf.R3_risc[j], intf.R3_arr[j+n], intf.R3_g[k+n-w]);
  fwrite(file_S, "R3 %h %h %h *
", intf.R3_risc[j], intf.R3_arr[j+n], intf.R3_g[k+n-w]);
end
else begin
  display("R3 %h %h %h", intf.R3_risc[j], intf.R3_arr[j+n],
          intf.R3_g[k+n-w]);
  fwrite(file_S, "R3 %h %h %h
", intf.R3_risc[j], intf.R3_arr[j+n], intf.R3_g[k+n-w]); end

  display("R4 %h %h %h *", intf.R4_risc[j], intf.R4_arr[j+n],
          intf.R4_g[k+n-w]);
  fwrite(file_S, "R4 %h %h %h
", intf.R4_risc[j], intf.R4_arr[j+n],
          intf.R4_g[k+n-w]);
end
else begin
  display("R4 %h %h %h", intf.R4_risc[j], intf.R4_arr[j+n],
          intf.R4_g[k+n-w]);
fwrite (file_S, "%R4 %h %h %h\n", intf.R4_risc[j], intf.R4_arr[j+n], intf.R4_g[k+n-w]); end

if ((intf.R5_risc[j] != intf.R5_arr[j+n]) || (intf.R5_risc[j] != intf.R5_g[k+n-w]) || (^intf.R5_risc[j] == 1'bX)) begin
$display ("R5 %h %h %h *", intf.R5_risc[j], intf.R5_arr[j+n], intf.R5_g[k+n-w]);
fwrite (file_S, "%R5 %h %h %h\n", intf.R5_risc[j], intf.R5_arr[j+n], intf.R5_g[k+n-w]); end
}
else begin
$display ("R5 %h %h %h", intf.R5_risc[j], intf.R5_arr[j+n], intf.R5_g[k+n-w]);
fwrite (file_S, "%R5 %h %h %h\n", intf.R5_risc[j], intf.R5_arr[j+n], intf.R5_g[k+n-w]); end

$display ("R6 %h %h %h *", intf.R6_risc[j], intf.R6_arr[j+n], intf.R6_g[k+n-w]);
fwrite (file_S, "%R6 %h %h %h\n", intf.R6_risc[j], intf.R6_arr[j+n], intf.R6_g[k+n-w]); end
}
else begin
$display ("R6 %h %h %h", intf.R6_risc[j], intf.R6_arr[j+n],
    intf.R6_g[k+n−w]);

fwrite (file_S, "R6 %h %h %h\n", intf.R6_risc[j], intf.
    R6_arr[j+n], intf.R6_g[k+n−w]); end

if ((intf.R7_risc[j] != intf.R7_arr[j+n]) || (intf.R7_risc[j]
    != intf.R7_g[k+n−w]) || (^ intf.R7_risc[j] === 1'bx)) begin
$display ("R7 %h %h %h *", intf.R7_risc[j], intf.R7_arr[j+n]
    , intf.R7_g[k+n−w]);
fwrite (file_S, "R7 %h %h %h *\n", intf.R7_risc[j], intf.
    R7_arr[j+n], intf.R7_g[k+n−w]); end

else begin
$display ("R7 %h %h %h", intf.R7_risc[j], intf.R7_arr[j+n],
    intf.R7_g[k+n−w]);
fwrite (file_S, "R7 %h %h %h\n", intf.R7_risc[j], intf.
    R7_arr[j+n], intf.R7_g[k+n−w]); end

// assigning the individual status bits to a variable and to
display
st1 = intf.SR_risc[j][11]; st2 = intf.SR_risc[j][10]; st3 =
    intf.SR_risc[j][9]; st4 = intf.SR_risc[j][8];
stp1 = intf.SR_g[k+n−w][11]; stp2 = intf.SR_g[k+n−w][10]; stp3
    = intf.SR_g[k+n−w][9]; stp4 = intf.SR_g[k+n−w][8];
if ((intf.SR_risc[j] != intf.SR_arr[j+n]) || (intf.SR_risc[j] != intf.SR_g[k+n-w]) || (^intf.SR_risc[j] === 1'b0)) begin
    $display("SR %h %h %h *", intf.SR_risc[j], intf.SR_arr[j+n], intf.SR_g[k+n-w]);
    $fwrite(file_S, "SR %h %h %h \n", intf.SR_risc[j], intf.SR_arr[j+n], intf.SR_g[k+n-w]);
end

else begin
    $display("SR %h %h %h", intf.SR_risc[j], intf.SR_arr[j+n], intf.SR_g[k+n-w]);
    $fwrite(file_S, "SR %h %h %h \n", intf.SR_risc[j], intf.SR_arr[j+n], intf.SR_g[k+n-w]);
end

$display("PC %3h %h %h", j, intf.PC_pl[m-w], intf.PC_pl[m-w]);
$fwrite(file_S, "PC %3h %h %h \n", j, intf.PC_pl[m-w], intf.PC_pl[m-w]);

if ((intf.SR_risc[j] != intf.SR_arr[j+n]) || (intf.SR_risc[j] != intf.SR_g[k+n-w]) || (^intf.SR_risc[j] === 1'b0)) begin
    $display("Status Bits (processor): SR=%h; C=%d, N=%d, V=%d, Z=%d", intf.SR_risc[j-w], st1, st2, st3, st4);
    $display("Status Bits (expected): SR=%4h; C=%d, N=%d, V=%d, Z=%d", intf.SR_g[k+n-w], stp1, stp2, stp3, stp4);
end
$\texttt{fwrite (file\_S, "Status Bits (processor): SR=\%h; C=\%d, N=\%d, V =\%d, Z=\%d\n", int\_f} \cdot \texttt{SR\_risc[J-w], st1, st2, st3, st4);}$

$\texttt{fwrite (file\_S, "Status Bits (expected): SR=\%4h; C=\%d, N=\%d, V =\%d, Z=\%d\n", int\_f\cdot SR\_g[k+n-w], stp1, stp2, stp3, stp4);}$

end

else begin

$\texttt{display ("Status Bits: C=\%d, N=\%d, V=\%d, Z=\%d", st1, st2, st3, st4);}$

$\texttt{fwrite (file\_S, "Status Bits: C=\%d, N=\%d, V=\%d, Z=\%d\n", st1, st2, st3, st4);}$

end

$\texttt{display ("
---------------------------------------------------------------------\n\n")}$

$\texttt{fwrite (file\_S, "
---------------------------------------------------------------------\n\n")}$

end

$\texttt{finish;}$

end

$\texttt{initial}$

$\texttt{#23000 \$finish;}$

$\texttt{endtask}$

$\texttt{initial}$

$\texttt{#23000 \$finish;}$
endmodule