TEST GENERATION OF SIMPLE 16-BIT REDUCED INSTRUCTION SET
COMPUTER (RISC) MICROPROCESSOR

MOHD SYAFIQ BIN SUHAIRI

UNIVERSITI TEKNOLOGI MALAYSIA
UNIVERSITI TEKNOLOGI MALAYSIA

DECLARATION OF THESIS / UNDERGRADUATE PROJECT PAPER AND COPYRIGHT

Author's full name : MOHD SYAFIQ BIN SUHAIRI
Date of birth : 15 MAC 1987
Title : TEST GENERATION OF SIMPLE 16-BIT REDUCED INSTRUCTION SET COMPUTER (RISC) MICROPROCESSOR
Academic Session : 2009/2010

I declare that this thesis is classified as :

☐ CONFIDENTIAL (Contains confidential information under the Official Secret Act 1972)*
☐ RESTRICTED (Contains restricted information as specified by the organisation where research was done)*
☒ OPEN ACCESS I agree that my thesis to be published as online open access (full text)

I acknowledged that Universiti Teknologi Malaysia reserves the right as follows :

1. The thesis is the property of Universiti Teknologi Malaysia.
2. The Library of Universiti Teknologi Malaysia has the right to make copies for the purpose of research only.
3. The Library has the right to make copies of the thesis for academic exchange.

Certified by :

SIGNATURE
870315-09-5075
(NEW IC NO. /PASSPORT NO.)

SIGNATURE OF SUPERVISOR
Dr. OOI CHIA YEE
NAME OF SUPERVISOR

30TH APRIL 2010 30TH APRIL 2010

NOTES : If the thesis is CONFIDENTIAL or RESTRICTED, please attach with the letter from the organization with period and reasons for confidentiality or restriction.
"I acknowledge that I have studied this piece of work and in my opinion it is in accordance with the scope requirement and quality for the purpose of awarding the Bachelor Degree in Engineering (Computer)."

Signature

Name of Supervisor : Dr. Ooi Chia Yee

Date : 30th APRIL 2010
TEST PROGRAM GENERATION OF SIMPLE 16-BIT REDUCED INSTRUCTION SET COMPUTER (RISC) MICROPROCESSOR

MOHD SYAFIQ BIN SUHAIRI

Submitted in fulfilment of the requirements for the award of the degree of Bachelor of Engineering (Computer)

Faculty of Electrical Engineering
Universiti Teknologi Malaysia

APRIL 2010
"I declare that this final year report entitled ‘Test Program Generation of Simple 16-bit Reduced Instruction Set Computer (RISC) Microprocessor’ is the result of my own studies and design except as cited in the references. This work has not been accepted for any degree and is not concurrently submitted in candidature of any other degree."

Signature  : ........................................

Name of Author: MOHD SYAFIQ BIN SUHAIRI

Date : 30th April 2010
Dedicated to my beloved father, mother, brother and sisters, for their support and understandings my life as university student.
ACKNOWLEDGEMENT

First and foremost, I would like to express my gratitude to my supervisor, Dr. Ooi Chia Yee for the guidance and motivation given during the duration of this project. With her constant support and concern, this project is accomplished successfully.

My appreciation also goes to my beloved family who has been supportive to me all these years continuously.

Nevertheless, I would like to thank all the lecturers who have taught me in Universiti Teknologi Malaysia, and all my fellow friends who have provided tips, moral support and views on this project.
ABSTRACT

Testing is useful in manufacturing process to ensure the product that has been produced is in good quality. If the test response is not identical with the correct response, the system is considered as defective and will not be sold to the customers. For this project, a simple 16-bit RISC microprocessor is implemented based on Professor Bruce Jacob’s architecture of University of Maryland. This simple microprocessor has eight instruction sets, an arithmetic logic unit (ALU), several pipelining registers, a data memory, an instruction memory, seven control modules and eight 16-bit general registers. Microprocessor testing is different compared to the common digital circuit testing. This is because processor testing involves the use of instructions as test vectors. So, several test programs are written based on the existing instruction sets to verify the functionality of RISC microprocessor. The simple translator is used to convert the test programs generated by TetraMAX into instructions set form that can be understood by human. Then, assembler is used to convert the test programs into hexadecimal form. This machine code will be loaded to the instruction memory of the simple 16-bit RISC microprocessor. Design Vision software is used to generate netlist for this microprocessor. After that, generated netlist is used in TetraMAX to generate the test programs in ATPG process. Finally, generated test programs are used as external pattern source in fault simulation process and the percentage of fault coverage is evaluated.
ABSTRAK

# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECLARATION</td>
<td></td>
<td>ii</td>
</tr>
<tr>
<td>DEDICATION</td>
<td></td>
<td>iii</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENT</td>
<td></td>
<td>iv</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td></td>
<td>v</td>
</tr>
<tr>
<td>ABSTRAK</td>
<td></td>
<td>vi</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td></td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td></td>
<td>xi</td>
</tr>
<tr>
<td>LIST OF ABBREVIATIONS</td>
<td></td>
<td>xiii</td>
</tr>
<tr>
<td>LIST OF APPENDICES</td>
<td></td>
<td>xiv</td>
</tr>
</tbody>
</table>

1 INTRODUCTION

1.1 Background 1
1.2 Problem Statement 2
1.3 Objectives 3
1.4 Scope of Work 3
1.5 Project Methodology 4
1.6 Design Tools 6

2 LITERATURE REVIEW

2.1 Digital Testing 7
2.2 Build-in Self-Test (BIST) 8
2.3 Software-based Self-Test (SBST) 10
## 3 IMPLEMENTATION OF RISC MICROPROCESSOR

- 3.1 Overview of 16-bit RISC Microprocessor
- 3.2 Instruction Set Architecture (ISA)
- 3.3 Pipeline Implementation
  - 3.3.1 Arithmetic Logic Unit (ALU) module
  - 3.3.2 Sign-Extend module
  - 3.3.3 Left-Shift module
  - 3.3.4 Program Counter module
  - 3.3.5 Control modules
    - 3.3.5.1 CTL 1
    - 3.3.5.2 CTL 2
    - 3.3.5.3 CTL 3
    - 3.3.5.4 CTL 4
    - 3.3.5.5 CTL 5
    - 3.3.5.6 CTL 6
    - 3.3.5.7 CTL 7

## 4 TEST GENERATION OF RISC MICROPROCESSOR

- 4.1 Test Generation Modeling
- 4.2 Test Generation Process
  - 4.2.1 Synthesis using Design Vision
  - 4.2.2 Test Pattern Generation and Fault Coverage Evaluation using TetraMAX
- 4.3 Summary

## 5 TRANSLATOR

- 5.1 Overview
- 5.2 Translator Flow Chart
- 5.3 Restrictions
- 5.4 Summary
6 RESULT AND DISCUSSION 39
6.1 Results 39
  6.1.1 Verification of RISC Microprocessor 39
  6.1.2 Verification of the Translator 41
  6.1.3 Fault Coverage 42
6.2 Discussion 43
6.3 Summary 43

7 CONCLUSION AND RECOMMENDATIONS 44
7.1 Conclusion 44
7.2 Recommendations 44

REFERENCES 46
APPENDIX A-C 48-79
# LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE NO.</th>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Instruction set</td>
<td>13</td>
</tr>
<tr>
<td>5.1</td>
<td>Value assigned to opcode</td>
<td>37</td>
</tr>
<tr>
<td>6.1</td>
<td>Fault coverage and test generation time</td>
<td>42</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE NO.</th>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Test generation model</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Project methodology</td>
<td>5</td>
</tr>
<tr>
<td>2.1</td>
<td>Single stuck-at-fault</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>BIST process</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>BILBO Implementation</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>Test Generation Scheme</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>Instruction format</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>Pipelined Implementation of simple 16-bit microprocessor</td>
<td>15</td>
</tr>
<tr>
<td>3.3</td>
<td>ALU top level module</td>
<td>16</td>
</tr>
<tr>
<td>3.4</td>
<td>Adder module</td>
<td>17</td>
</tr>
<tr>
<td>3.5</td>
<td>NAND module</td>
<td>17</td>
</tr>
<tr>
<td>3.6</td>
<td>Comparator module</td>
<td>18</td>
</tr>
<tr>
<td>3.7</td>
<td>Sign-Extend module</td>
<td>19</td>
</tr>
<tr>
<td>3.8</td>
<td>Left-Shift operation</td>
<td>20</td>
</tr>
<tr>
<td>3.9</td>
<td>Left-Shift module</td>
<td>20</td>
</tr>
<tr>
<td>3.10</td>
<td>Program Counter module</td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>Test Generation modeling</td>
<td>25</td>
</tr>
<tr>
<td>4.2</td>
<td>Constraint module</td>
<td>26</td>
</tr>
<tr>
<td>4.3</td>
<td>Test generation flow chart</td>
<td>27</td>
</tr>
<tr>
<td>4.4</td>
<td>Screenshot for step (a)</td>
<td>28</td>
</tr>
<tr>
<td>4.5</td>
<td>Screenshot for step (c)</td>
<td>28</td>
</tr>
<tr>
<td>4.6</td>
<td>Screenshot for step (b)</td>
<td>29</td>
</tr>
<tr>
<td>4.7</td>
<td>TetraMAX interface</td>
<td>30</td>
</tr>
<tr>
<td>4.8</td>
<td>Screenshot for step (c)</td>
<td>30</td>
</tr>
<tr>
<td>4.9</td>
<td>Screenshot for step (d)</td>
<td>31</td>
</tr>
<tr>
<td>Section</td>
<td>Content</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.10</td>
<td>Screenshot for step (e)</td>
<td>31</td>
</tr>
<tr>
<td>4.11</td>
<td>Screenshot for step (f)</td>
<td>32</td>
</tr>
<tr>
<td>4.12</td>
<td>Screenshot for step (h)</td>
<td>32</td>
</tr>
<tr>
<td>5.1</td>
<td>Translator flow chart</td>
<td>36</td>
</tr>
<tr>
<td>5.2</td>
<td>Translator operation</td>
<td>37</td>
</tr>
<tr>
<td>5.3</td>
<td>Correct and wrong test sequence</td>
<td>38</td>
</tr>
<tr>
<td>6.1</td>
<td>Verification test sequence</td>
<td>40</td>
</tr>
<tr>
<td>6.2</td>
<td>Simulation result</td>
<td>40</td>
</tr>
<tr>
<td>6.3</td>
<td>Test sequence for translator verification</td>
<td>41</td>
</tr>
<tr>
<td>6.4</td>
<td>Translator output file</td>
<td>41</td>
</tr>
<tr>
<td>6.5</td>
<td>Correct and wrong primary input</td>
<td>43</td>
</tr>
</tbody>
</table>
**LIST OF ABBREVIATIONS**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATE</td>
<td>Automatic Test Equipment</td>
</tr>
<tr>
<td>CUT</td>
<td>Circuit-Under-Test</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>BIST</td>
<td>Built-In Self-Test</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computer</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>DFT</td>
<td>Design For Testability</td>
</tr>
<tr>
<td>CISC</td>
<td>Complex Instruction Set Computer</td>
</tr>
<tr>
<td>s-a-0</td>
<td>stuck-at-0</td>
</tr>
<tr>
<td>s-a-1</td>
<td>stuck-at-1</td>
</tr>
<tr>
<td>ALU</td>
<td>Arithmetic Logic Unit</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture</td>
</tr>
<tr>
<td>RRR</td>
<td>Register-Register-Register type instruction</td>
</tr>
<tr>
<td>RRI</td>
<td>Register-Register-Immediate type instruction</td>
</tr>
<tr>
<td>RI</td>
<td>Register-Immediate type instruction</td>
</tr>
<tr>
<td>MUX</td>
<td>Multiplexer</td>
</tr>
<tr>
<td>SBST</td>
<td>Software Based self-test</td>
</tr>
<tr>
<td>ATPG</td>
<td>Automatic Test Pattern Generation</td>
</tr>
<tr>
<td>BILBO</td>
<td>Build-in Logic Block Observer</td>
</tr>
<tr>
<td>LFSR</td>
<td>Linear Feedback Shift Register</td>
</tr>
<tr>
<td>SoC</td>
<td>System-on-Chip</td>
</tr>
</tbody>
</table>
LIST OF APPENDICES

<table>
<thead>
<tr>
<th>APPENDIX</th>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>C Source Code for the Translator</td>
<td>48</td>
</tr>
<tr>
<td>B</td>
<td>RISC Microprocessor Top Level Module</td>
<td>55</td>
</tr>
<tr>
<td>C</td>
<td>Test patterns generated by TetraMAX</td>
<td>59</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

1.1 Background

Testing is very important to ensure the product quality and performance. If the products reach top quality level, the customers trust that products and at the same times it will gain the company profits. Therefore, testing is needed at the end of manufacturing process before the product delivery. Product is considered as defective component if it does not pass the testing process.

There are two main issues that need to be considered during testing, which are testing time and testing speed. Testing time gives a big impact to the product market flow and also to the company financial. For example, if testing time is longer, product and testing cost also increase. Testing speed is also important in testing process. This is because at-speed testing can detect more faults and less faults escaping during testing.

Automatic test equipment (ATE) is a common test device that is widely used for circuit testing. ATE is computer-controlled equipment that tests electronic devices for functionality and performance. For the optimum test performance, the ATE must operate with the same speed or more than the circuit under test (CUT) [3]. But there is one big issue when using ATE, which is the cost. Processor testing must be done at-speed test in order to prevent faults from escaping during testing. So, software-based self test will be used as an alternative to test microprocessor at-speed and at a lower cost.
Software-based self testing is an effective method to test processor at-speed and at a lower testing cost. SBST moves the test functions from external testers to on-chip resources: processors, memories, and on-chip buses [5]. In another hand, it can eliminate area, performance and power consumption overheads. By using this method, test application can be performed at the processor’s standard operating conditions.

1.2 Problem Statement

Nowadays, the number of logic gates in the IC is gradually increasing. As a result, the higher operational frequency is needed. Along with these improvements, the testing process also becomes more challenging.

One of the main issues to be concerned is fault coverage. Fault coverage is defined as the number of faults detected over the total of faults. Low fault coverage affects the product quality. A fault can be detected if the response of the circuit is not equal to the expected responses. If the value of fault coverage is low, the number of faulty products that pass the test will increase. Once this happens, customers will blame the company because of the defective product. At the same time, the company may have to put extra cost for product maintenance.

Microprocessor testing is different compared to the normal digital circuit testing. This is because the processor testing involves the usage of specific instructions to apply test sequences. The instructions need to be written in a correct sequence in order to test the targeted component of the microprocessor. In addition, the understanding of the RTL design of this microprocessor and its instructions set architecture (ISA) are needed. To get the higher fault coverage, software based self-test (SBST) technique is the best choice.
1.3 Objectives

The main cores of this project are:

(a) to derive the test programs for simple 16-bit RISC microprocessor using ATPG to detect faults.

(b) to run the self testing of simple 16-bit RISC microprocessor using the generated test programs in order to verify the functionality of this microprocessors.

(c) to evaluate the fault coverage of the test programs.

1.4 Scope of Work

The project scope is needed to narrow down the project work area. Figure 1.1 shows the overall project. First thing first, the simple 16-bit RISC microprocessor is created based on Professor Bruce Jacob’s design from University of Maryland [2]. Next, the constraint circuit is created to modify the primary input generated by TetraMAX. Lastly, the simple translator is developed to translate binary test programs into instructions set form.

![Test generation model](image-url)
1.5 Project Methodology

To accomplish project, the project methodology shown in Figure 1.2 will be followed. Figure 1.2 shows the flow of this project. At the starting point, the information on designing simple 16-bit RISC microprocessor is obtained from a variety of books borrowed from library, articles and e-books from the internet. After understanding the concept, the implementation of simple 16-bit RISC microprocessor is started. When the design is complete, few simple test programs are written. By using these test programs, the design will be verified to make sure that it works properly.

The next step is to build the simple translator that can translates the test programs into the instructions set form. Once the translator is completed, the simple test programs will be tested to make sure that the translator can work properly without any error. Next, RTL design of a simple 16-bit RISC microprocessor will be used in Design Vision software. This process will produce the netlist for this microprocessor. This netlist will be used in TetraMAX tool to generate test programs and to evaluate the fault coverage of simple 16-bit RISC microprocessor.
Figure 1.2: Project methodology
1.6 Design Tools

Each of the design modules in the RTL modelling is designed using Verilog language with the aids of the Quartus II software. This software is used during design process and verification of the simple RISC microprocessor. For the translator, Borland C++ compiler is used. Design Vision software is used to create netlist for this microprocessor. TetraMAX software is used to generate test patterns and to evaluate the fault coverage.
2.1 Digital Testing

There are two types of digital testing method, which are logic test and electrical test. Logic test method is used to find faults that cause the change of logical behaviour of the circuit. For example, if the systems give logic 0 and the expected value is logic 1, so this situation will be considered as fault. For electrical test, it verifies the correctness of the circuit based on voltage and current level [2].

The common forms of single stuck at faults are stuck-at-0 and stuck-at-1. In order to enable the stuck-at-0, that line need to be set as logic 1. For stuck-at-1, logic 0 will set at that line. There are three properties for single stuck-at-fault according to [13]:

a) Only one line faulty
b) Logic 0 or 1 must be permanently set to represent faulty
c) The fault can be at an input or output of the gate

Figure 2.1 shows a single stuck-at-fault. From that figure, it shows that the faulty can be observed at the output lines if one of the faults in the circuit is activated.
2.2 Build-in Self-Test (BIST)

BIST is one of design-for-testability techniques that allows the circuit to run self-testing. This technique adds additional software and hardware into the integrated circuit to make it more testable. Nowadays, the speed of the microprocessor has reached GHz in order to do multitasking operations. However, ATE cannot beat the speed of the microprocessor. As a result, BIST is widely used as a testing method because it permits at-speed testing. Figure 2.2 shows how a comparator compares the reference signature stored in the ROM with the output response compactor (Signature) during the BIST.

---

**Figure 2.1: Single stuck-at-fault**

**Figure 2.2: BIST Process [2]**
Another type of BIST is Build-in Logic Block Observer (BILBO). BILBO is an additional testability hardware added to the flip-flop, which allows the flip-flop to function in four modes. They can be reconfigured as Linear Feedback Shift Register (LFSR) pattern generator or response compactor, or like the scan chain. Figure 2.3 shows the BILBO implementation. LFSR is used to generate the pseudo-random tests. This method needs a lot of test sequence in order to get high fault coverage.

![BILBO Implementation Diagram](image)

**Figure 2.3: BILBO Implementation [2]**

There are some issues that needs to be considered when implementing BIST:

a) Fault coverage
b) Chip area used by BIST
c) Test time

Advantages of the BIST include the following [12]:

a) Since external hardware to test the circuit is not needed, it can lower the test cost
b) Easier customer support and allow self-testing

Disadvantages of the BIST:

a) It can reduce the access time
b) Cause area overhead since it requires an extra hardware
c) Pin overhead occur, since the BIST circuit needs a path to interface with the outside world.
d) On-chip testing hardware itself can also fail
BIST method is very good in order to get higher fault coverage. This method of testing is not used in this project. But, if the low fault coverage obtained, then this method can be recommended for future work.

2.3 Software-based Self-Test (SBST)

Software-based self-test is recently effective for the manufacturing test and System on Chip (SoCs) [3]. This method can reduce testing cost and enable very good at-speed testing. SBST method helps to utilizes processor's programmability and on-chip resources to eliminate the need of specialized hardware for test pattern generation, application and evaluation, as commonly required in hardware-based schemes [3]. Figure 2.4 shows test generation scheme.

![Figure 2.4: Test generation scheme [3]](image)

Both test code and test data can be called as test programs. Test programs are used to detect faults as many as possible. Test program generation can be performed once and the resultant test routine can be reused for the whole life cycle of the CPU.
There are some advantages of software-based self-test method:

(a) Low-Design-For-Testability (DFT) efforts without area overhead and performance degradation.
(b) Low power consumption for testing process.
(c) Low test cost because of no external hardware is used.
(d) Enable at-speed testing.
CHAPTER 3

IMPLEMENTATION OF RISC MICROPROCESSOR

In this chapter, the full description about RISC microprocessor and each design module involved will be discussed. Test generation modelling is also included in the chapter. All the verilog coding will be shown except verilog coding for control modules.

3.1 Overview of 16-bit RISC Microprocessor

RISC or Reduced Instruction Set Computer is one type of the microprocessor architecture that has highly optimized set of instructions. The standard feature in this type of design is pipelining. This feature can help to run some instructions at the same time. As a result, the execution time for all instruction can be done in a shorter period of time. For this project, the microprocessor is implemented based on the Little Computer (LC-896) developed by Peter Chan from University of Michigan [2]. This microprocessor contains eight registers, one data memory and one instruction memory. Each of the memory modules has 65536 addressable locations. There are five stages involved which are fetch, decode, execute, write and write back. Every stage has registers and known as the pipelined registers.
3.2 Instruction Set Architecture (ISA)

This simple 16-bit microprocessor has eight basic instructions. It consists of three arithmetic instructions, two branching instructions, one bits manipulation instruction and two others instructions for storing and loading data. The length for each instruction is 16-bit size. There are three types of instructions format; RRR type, RRI type, and RI type. “R” stands for register and “I” stands for immediate. Figure 3.1 shows the overview of instruction formats. Table 3.1 shows the simple description about instructions set and assembly code format.

Table 3.1: Instruction set

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>OpC</th>
<th>Assembly Code</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD (RRR)</td>
<td>Add</td>
<td>000</td>
<td>add rA,rB,rC</td>
<td>rA ← rB+rC</td>
</tr>
<tr>
<td>ADDI (RRI)</td>
<td>Add immediate</td>
<td>001</td>
<td>addi rA,rB,imm</td>
<td>rA ← rB+imm</td>
</tr>
<tr>
<td>NAND (RRR)</td>
<td>Nand</td>
<td>010</td>
<td>nand rA,rB,rC</td>
<td>rA ← ~ (rB&amp;rC)</td>
</tr>
<tr>
<td>LUI (RI)</td>
<td>Load upper immediate</td>
<td>011</td>
<td>lui rA,imm10</td>
<td>rA ← (imm10&lt;&lt;6)</td>
</tr>
<tr>
<td>SW (RRI)</td>
<td>Store word</td>
<td>100</td>
<td>sw rA,rB,imm</td>
<td>rA --&gt; Mem[rB+imm]</td>
</tr>
<tr>
<td>LW (RRI)</td>
<td>Load word</td>
<td>101</td>
<td>lw rA,rB,imm</td>
<td>rA ← Mem[rB+imm]</td>
</tr>
</tbody>
</table>
| BEQ (RRI) | Branch if equal     | 110 | beq rA,rB,imm | If(rA==rB) 
    PC = PC+1+imm  
    else PC = PC+1  |
| JALR (RRI)| Jump and link register | 111 | jalr rA,rB  | PC ← rB  
                      rA ← PC+1 |
3.3 Pipeline Implementation

Figure 3.2 shows the pipelined implementation of simple 16-bit RISC microprocessor that is used for this project. This microprocessor is divided into five different stages which are fetch, decode, execute, memory and write-back stage. By dividing the processor into several stages, the performance and speed for this microprocessor can be increased.
Figure 3.2: Pipeline implementation of simple 16-bit microprocessor [2]
3.3.1 Arithmetic Logic Unit (ALU) module

There are four functions involved in arithmetic logic unit which are addition, comparator, NAND operation and passing value operation. Each of them is designed module by module and connected all together using top level module called ALU. Figure 3.3 shows the verilog coding for the ALU top level module. For this design, each time when instruction is executed, all four functions are performed but only one result from the certain function will be passed to the output. To perform that operation 4-to-1 multiplexor is used. When the selector is 0, the result from the adder will be passed to the output. If the selector is 1, the result from NAND module will be passed to the output and if the selector is equal to 3, the value at line A will be passed to the output. For the information, if the selector is 2, the output is equal to 0.

```verilog
module ALU(A,B,Cin,S,EQ,alu_output);   //ALU
input Cin;
input [1:0] S;
input [15:0] A,B;
output EQ;
output [15:0] alu_output;
wire [15:0] Sum,Out;
reg [15:0] alu_output;

add16 tambah(A, B, Cin, Sum);
nand16 NAND_16(A, B, Out);
comparator16 com(A, B, EQ);

always@(S or Sum or Out or A)       //MUX TO SELECT ALU
OUTPUT
    case(S)
        0 : alu_output = Sum;
        1 : alu_output = Out;
        3 : alu_output = A;
        default : alu_output = 0;
    endcase
endmodule

Figure 3.3: ALU top level module
Figures 3.4-3.6 show the verilog codings for the adder, NAND, and comparator respectively.

module add16(A,B,Cin,Sum);  //ADDER
input [15:0] A,B;    //INPUT
input Cin;
output reg [15:0] Sum;

always@(A,B,Cin)
Sum = A+B+Cin;     //OUTPUT
endmodule

**Figure 3.4: Adder module**

module nand16(A,B,Out);   //NAND
input   [15:0]A,B;
output  [15:0]Out;

assign Out=~(A&B);  //OUTPUT
endmodule

**Figure 3.5: NAND module**
module comparator16(A,B,EQ); //COMPARATOR

input [15:0] A,B;
output EQ;
reg EQ;

always@(A or B)
begin
if (A==B) EQ = 1;
else EQ=0;
end

endmodule

Figure 3.6: Comparator module
3.3.2 Sign-Extend module

The function of this module is to sign extend an eight bits of the immediate value into 16-bit value. Figure 3.7 explains that the first 7-bit of the output is equal to the input immediate value and the last 9-bit which is (Q [15:7]) will be equal to the most significant bit of the immediate input value.

```verilog
module sign_extend(D,Q);  // SIGN-EXTEND
input[6:0] D;
output [15:0]Q;
integer k;
reg [15:0]Q;

always@(D)
begn
Q[6:0]<= D;
Q[7]<=D[6]; for (k = 8; k<16; k = k+1 ) Q[k]<=D[6];
end //COPY MOST SIGNIFICAT BIT OF IMM INTO Q [15:7]
endmodule
```

**Figure 3.7: Sign-Extend module**

3.3.3 Left-Shift module

Left-shift module will perform six times shifting to the left for the 10-bit immediate value received. Actually, this module will place the 10-bit immediate value to Q [15:6]. The remaining Q [5:0] will be filled with zero. Figure 3.8 illustrate how the left-shift module operates and Figure 3.9 shows the verilog coding for this module.
3.3.4 Program Counter module

Program counter module is a sixteen bit register which holds the address of the next instruction to be executed. If reset is equal to 0, the output Q will be equal to 0. Figure 3.10 shows the verilog coding for program counter.
3.3.5 Control modules

Based on Figure 3.2, there are seven control modules in this simple microprocessor which are labelled as CTL 1 until CTL 7.

3.3.5.1 CTL 1

The input for this module is the content of rT register in MEM/WB. This module controls the WErf line. If any data is to be written in register file, it comes from the RF WRITE DATA register. If the content of rT register equal to 0, WErf will equal to 0. Otherwise, WErf will equal to 1.
3.3.5.2 CTL 2

The input for this module is the content of OP register in EX/MEM. This module controls both the WEdmem line and MUXout line. The WEdmem is only enabled when the opcode is SW. Otherwise, WEdmem is set to 0. The MUXout will choose the output of data memory if the opcode is LW. Otherwise, it will choose the output of ALU OUTPUT register in EX/MEM.

3.3.5.3 CTL 3

The inputs for this module are EQ! (indicate two inputs operand are equal) line and instruction opcode in ID/EX. This module controls MUXpc line, Pstomp line and FUNCalu line. If this module detects BEQ opcode and the operands are equal, MUXpc will choose the value of PC+1+OPERAND0 adder at execute stage. At the same time, the registers at ID/EX and IF/ID will be filled with NOP instructions (STOMP event). Means that, the content of OP and rT registers in ID/EX and instruction register in IF/ID will set to zero. For JALR opcode, MUXpc will choose the the output of MUXalu1 and Pstomp equal to 1. For other instructions, MUXpc will choose the output of PC+1 register and no stomp event occur.

If this module detects ADD, ADDI, SW, LW or BEQ opcodes, the ALU will do addition operation (FUNCalu=0). If NAND opcode is detected, the ALU will do NAND operation (FUNCalu=1). Otherwise, the ALU will do pass operation (FUNCalu=3).

3.3.5.4 CTL 4

This module input is the instruction opcode from OP register in ID/EX. This module controls the MUXimm line. For all instructions that used immediate values, this multiplexor will choose the output of OPERAND0 register in ID/EX. If JALR
opcode is detected, the output of PC+1 adder will be selected. For all others instruction, this multiplexor will choose the output of OPERAND2.

3.3.5.5 CTL 5

The inputs for this module are the register identifier s1 and s2 in ID/EX and three rT registers from the previous three instructions. This control unit handles data forwarding. It controls MUXalu1 and MUXalu2 lines. If the output of rT register in EX/MEM is equal to each output of s1 and s2 registers, the output of ALU OUTPUT register will be selected at both multiplexors. Otherwise, if the output of rT register in MEM/WB is equal to each output of s1 and s2 registers, the output of RF WRITE DATA register in MEM/WB will be selected. Finally, if the output of rT register in WB/END is equal to each output of s1 and s2 registers, the output of RF WRITE DATA register in WB/END will be selected.

3.3.5.6 CTL 6

The input of this module is the opcode currently in the decode stage. This module controls the MUXop0 and MUXs2 lines. For ADDI, LW, SW, BEQ and JALR, MUXop0 will choose the output from Sign-Ext-7 register. Otherwise, it will choose the output of Left-Shift-6 register. MUXs2 will choose the value of rC if ADD and NAND are detected. For all others instruction, it will choose the value of rA
3.3.5.7 CTL 7

The inputs of this module are the content of OP, rA, rB, and rC registers in IF/ID and the outputs of OP and rT registers in ID/EX. The control module’s outputs are the OP and rT fields of the ID/EX register and the Pstall line. If LW opcode is detected at ID/EX register and targets any register that the instruction in decode uses as a source register, a STALL event occurs.

On a pipeline stall, the instructions in the fetch and decode stages are held up, but the rest of the instructions in the pipeline are allowed to move ahead. To fill the created hole, an ADD instruction with target register r0 (rT = 0) is placed in the ID/EX register (ADD r0, X, X). Since r0 is a read only location, this instruction will not have any effect. This module also produces a value for the rT register in ID/EX as follows: if the instruction in IF/ID is a type that targets the register file (ADD, ADDI, NAND, LUI, LW, JALR), the value of rA is passed on to the rT register. For SW and BEQ instructions, the binary value 000 is passed, indicating that the instruction does not store any value in the register file. Again, this works because r0 is a read only target.
CHAPTER 4

TEST GENERATION OF RISC MICROPROCESSOR

4.1 Test Generation Modeling

During test generation process, data memory and instruction memory are assumed as fault free. So, these two memories modules can be removed from the design. Once removed, the original output from the instruction memory will be made primary input to this microprocessor. Then, the original input to the data memory will be made primary output. The original output from data memory will be made primary input. One constraint circuit is added to the microprocessor. This constraint circuit is set as fault free during ATPG process. Figure 4.1 illustrates the test generation model. And Figure 4.2 shows the verilog coding for the constraint circuit.

![Test generation model diagram](image)

Figure 4.1: Test generation modeling
module cct(Primein, out);
input [15:0] Primein;
output [15:0] out;

reg [15:0] out;
wire [2:0] OP;
assign OP=Primein[15:13];

reg [3:0]E4=4'b0000;
reg [2:0]Rc=3'b000;
reg [2:0]Rb=3'b000;
wire [6:0] Rn;
assign Rn=Primein[6:0];

always@(OP or Primein or E4 or Rc or Rb or Rn)
if({OP==0,OP==2})
begin
  out={OP,Primein[12:10],Primein[9:7],E4,Primein[2:0]};
end
else if(OP==7)
begin
  out={OP,Primein[12:10],Primein[9:7],E4,Rc};
end
else if({OP==4,OP==5})
begin
  out={OP,Primein[12:10],Rb,Primein[6:0]};
end
else
  out={OP,Primein[12:10],Primein[9:7],Primein[6:3],Primein[2:0]};
endmodule

Figure 4.2: Constraint module
4.2 Test Generation Process

In this part, the detailed description of the test generation process for the simple RISC microprocessor will be discussed. The process will include how to create the netlist for the RISC microprocessor using Design Vision software, and how to generate test patterns and do fault simulation using TetraMAX software. Figure 4.3 shows the flow chart of the process.

![Flow Chart]

**Figure 4.3: Test generation flow chart**

4.2.1 Synthesis using Design Vision

The following is the procedure to synthesize the RISC microprocessor using Design Vision:

(a) Figure 4.4 was referred. “cd <your project folder name>” was typed in the terminal. This command was used to access the project folder.
(b) After that, to open Design Vision, “design_vision” was typed at the terminal.
(c) Setup option was selected in File menu tab. Figure 4.5 was referred for configuration. Then OK button was clicked.

(d) After that, Read option was selected in File menu tab to read all the design files.
(e) When read process was successful, Compile option was selected from the Design menu tab. Then, without change any setting, OK button was clicked to create
design netlist.
(f) When compile design was finished, Save As option was selected from File menu to save created netlist. The format was changed from AUTO to VERILOG(v) before save file.

4.2.2 Test Pattern Generation and Fault Coverage Evaluation using TetraMAX

The following is the procedure for test pattern generation and fault coverage evaluation of RISC microprocessor using TetraMAX:

(a) “cd <project file folder>” was typed and then ENTER was pressed to enter project folder.
(b) After that, “unsetenv LD_ASSUME KERNEL” was typed and ENTER was pressed. Then, “tmax” was typed to open TetraMAX. Figure 4.6 shows screenshot for step (b). Figure 4.7 shows screenshot of TetraMAX interface.

Figure 4.6: Screenshot for step (b)
(c) Netlist tab was clicked to add netlist file created by Design Vision and also library file. After that, Run button was clicked. Figure 4.8 is a screenshot for this step.
(d) Next, Build tab was clicked. Top level file name was selected at “Top module name” field, then Run button was clicked. Figure 4.9 is screenshot for this step.

![Figure 4.9: Screenshot for step (d)](image)

(d) DRC tab was clicked and without changing any setting, Run button was clicked.

(e) “add faults –module <selected module name>” was typed at TEST field and ENTER was pressed. Figure 4.10 is a screenshot for this step.

![Figure 4.10: Screenshot for step (e)](image)

(f) ATPG tab was clicked. Configuration was liked in Figure 4.11. After that, Full-Seq button was clicked.
Figure 4.11: Screenshot for step (f)

(g) After ATPG process was finished, Write Pat. tab was clicked. Pattern name was written and file format was changed to STIL and OK button was clicked.

(h) Fault Sim. tab was clicked. Configuration was liked in figure 4.12 then Run button was clicked to run the simulation. Note that external pattern source was selected based on Figure 4.12.

Figure 4.12: Screenshot for step (h)
4.3 Summary

First of all, this step-by-step process is a simple instruction to illustrate test generation process for this project. This instruction can be used for fault analysis but some setting and steps maybe different.
CHAPTER 5

TRANSLATOR

5.1 Overview

For this project, translator is very important to convert the binary test programs into instruction set form that can be easily understood by human. Then, the assembler can be used to convert it to hexadecimal form that can be understood by RISC microprocessor. The implementation of the translator will be discussed in this chapter.

5.2 Translator Flow Chart

Figure 5.1 shows the flow chart of the translator to be coded in C programming. Firstly, the translator reads the test program that has been saved in text file. For example, “000010100000011” is one of the machine code that translator read from the text file. Each bits of it will be assigned to op, op1, op2, a, a1, a3, b, b1, b2, lw, lw1, lw2, lw3, lw4, lw5 and lw6 string’s variables from left to right respectively. Then, all the variables will be converted into integer. “op” and “op1” are left shifted two and one times respectively. Both of “op” and “op1” are added and stored in variable called “opcode”. There are eight types of instructions which are ADD, ADDI, NAND, LW, SW, LUI, BEQ and JALR. Table 5.1 shows the value assigned for each opcodes. For this case, opcode = 0 = ADD. After that, (a, b, lw4) and (a1, b1, lw5) are left shifted two and one times respectively. Variables “finala”, “finalb” and “finalc” are equal to the value of registers A, B an C. Results for
“a+a1+a2”, “b+b1+b2” and “lw+lwl+lw2” will be stored in “finala”, “finalb” and “finalc” correspondingly. “ADD r1,r2,r3” is the output for this example. Finally, the complete form of the instruction will be saved to output file. The name for the output file can be set manually when the translator ran. The translator will proceed to the next test sequence and ran the process based on Figure 5.1 until the end of file is reached. Figure 5.2 illustrate the operation of the translator and the instruction set that can be formed.
Figure 5.1: Translator flow chart
Table 5.1: Value assigned to opcode

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Value assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>0</td>
</tr>
<tr>
<td>addi</td>
<td>1</td>
</tr>
<tr>
<td>nand</td>
<td>2</td>
</tr>
<tr>
<td>lui</td>
<td>3</td>
</tr>
<tr>
<td>sw</td>
<td>4</td>
</tr>
<tr>
<td>lw</td>
<td>5</td>
</tr>
<tr>
<td>beq</td>
<td>6</td>
</tr>
<tr>
<td>jalr</td>
<td>7</td>
</tr>
</tbody>
</table>

Figure 5.2: Translator operation
5.3 Restrictions

This translator cannot handle the test program that has space between each bit of the machine code and at the end of test sequence. If this happened, the translator will produce the wrong instruction. Figure 5.3 shows the invalid test programs form. Another restriction is the “ENTER” key cannot be pressed after the last test sequence is written in the text file.

Figure 5.3: Correct and wrong test sequence

5.4 Summary

This chapter has described a simple translator which is able to perform its functions correctly and fast. This translator can be improved by overcoming the restrictions that has been mentioned in Section 5.3. As a result, this translator can be more capable to be used in future.
CHAPTER 6

RESULT AND DISCUSSION

In this chapter the verification result of the simple 16-bit microprocessor, the functionality of the translator and the fault coverage for this microprocessor is shown. The functionality of the constraint circuit will be discussed.

6.1 Results

6.1.1 Verification of RISC Microprocessor

During the verification process, the instructions in Figure 6.1 are used to check the functionality of this simple microprocessor. After all the instructions are executed, the content of registers r2, r3, r5 and r6 should be liked in Figure 6.1. Figure 6.2 shows the simulation result for this simple microprocessor. “test” output is used to show that the content of r6 is stored at address 7. The result shows that this processor work without any problem.
Figure 6.1: Verification test sequence

Figure 6.2: Simulation result
6.1.2 Verification of the Translator

One of the test sequences shown in Figure 6.3 is used to verify the functionality of the translator. By right, the translator should produce the instructions as shown in Figure 6.3. Figure 6.4 shows the text file that is created by the translator for the test sequences in Figure 6.3. As a result, this has proved that the translator work very well.

| 1001011111101000 | sw r5,r0,68 |
| 1001101111011000 | sw r6,r0,58 |
| 0110110001011011 | lui r3,5b |
| 0111111100000101 | lui r7,305 |
| 1010100110110011 | lw r2,0,33 |
| 1000010100011000 | sw r1,r0,18 |
| 1001001101111110 | sw r4,r0,7e |
| 0100000111110111 | nand r0,r3,r7 |
| 0101110100101000 | nand r7,r2,r0 |
| 1001010101001100 | sw r5,r0,4c |
| 1001010011111110 | sw r5,r0,7e |
| 1001110111111111 | sw r7,r0,7f |
| 0111101000000001 | lui r6,201 |
| 0011001000011110 | addi r4,r4,1e |
| 0000000000100110 | add r0,r0,r6 |

Figure 6.3: Test sequence for translator verification

Figure 6.4: Translator output file
6.1.3 Fault Coverage

Table 6.1 shows the value of fault coverage and test generation time for simple 16-bit RISC microprocessor.

Table 6.1: Fault coverage and test generation time

<table>
<thead>
<tr>
<th></th>
<th>Fault Coverage</th>
<th>Test generation time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original 16-bit RISC processor</td>
<td>66.03%</td>
<td>Approx 30 minutes</td>
</tr>
</tbody>
</table>
6.2 Discussion

During test generation process, a constraint circuit which is also known as virtual circuit is used. This constraint circuit is used to prevent TetraMAX from generating invalid test patterns. Figure 6.5 shows the correct and wrong test patterns. That example is based on instruction “add r6,r6,r4”.

![Generated by tetramax](image)

<table>
<thead>
<tr>
<th>Generated by tetramax</th>
<th>Correct form of input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prin=0001101100010100</td>
<td>PI=0001101100000100</td>
</tr>
</tbody>
</table>

(a) Wrong

(b) Correct

**Figure 6.5: Correct and wrong primary input**

This microprocessor is designed module by module for easy debugging process. In other words, we can easily add fault to the selected test module by simply use this syntax “add faults –module <name of target module>” during test mode in TetraMAX software.

6.3 Summary

First of all, the functionality of RISC microprocessor and the translator can be concluded functioning as expected. The value of fault coverage is evaluated. The value is low but it doesn’t mean that this method of testing is not good.
CHAPTER 7

CONCLUSION AND RECOMMENDATIONS

7.1 Conclusion

Software-based self-test is very effective method for testing of components in System-on-Chips (SoCs) such as memory and processor [5]. In addition, this method of testing also can reduce cost and also enables at-speed testing. During the testing process, the memory modules are assumed as fault free and not included during test generation process. At the end of this project, test programs for simple 16-bit RISC microprocessor are generated and the fault coverage is evaluated. Based on the obtained result, this method of testing is not to bad and effective enough to evaluate fault coverage and also in detecting faults. Lastly, constraint circuit is needed when using this type of testing method in order to get valid test result.

7.2 Recommendations

For the translator, it can be modified so that it can handle both either the test patterns has gap or space between each bits or without space test patterns which are in STIL format. Current translator can support for STIL format test patterns and not for WGL format.
Finally for the testing, in order to get the better fault coverage, design-for-testability (DFT) modelling is suggested [1]. This method will make the loop inside the microprocessor more controllable and observable. On the other, this method will also help to reduce test generation time because more primary inputs and primary outputs pins are assigned to replace the removed registers. As a result, the time to propagate fault to the primary output will be shorter.
REFERENCES


APPENDIX A

C Source Code for the Translator

#include <stdio.h>
#include <string.h>

#define LINE_LENGTH 80

main()
{
    FILE *input;  /* declare the input file pointer */
    FILE *output;
    char line[80];
    char readfile[80];
    char writefile[80];
    int op;
    int op1;
    int op2;
    int a;
    int a1;
    int a3,a2;
    int b;
    int b1;
    int b2;
    int lw;
    int lw1;
int lw2;
int lw3;
int lw4;
int lw5;
int lw6;
int opcode, finala, finalb, finalc, Imm;

printf("Please insert readfile name:");
scanf("%s", &readfile);
printf("Please insert writefile name:");
scanf("%s", &writefile);

input = fopen(readfile, "r"); /* open the file for reading */
output = fopen(writefile, "w"); /* open the file for writing */
fprintf(output, "\0");

while (fgets(line, 80, input) != NULL)
{

sscanf(line, "%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c", &op, &op1, &op2, &a, &a1, &a3, &b, &b1, &b2, &lw, &lw1, &lw2, &lw3, &lw4, &lw5, &lw6);

printf("%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c", op, op1, op2, a, a1, a3, b, b1, b2, lw, lw1, lw2, lw3, lw4, lw5, lw6);

op=atoi(&op);
op1=atoi(&op1);
op2=atoi(&op2);
a=atoi(&a);
a1=atoi(&a1);
a2=atoi(&a3);
printf("%d\n", a2);
b=atoi(&b);
b1=atoi(&b1);
b2=atoi(&b2);
lw=atoi(&lw);
lw1=atoi(&lw1);
lw2=atoi(&lw2);
lw3=atoi(&lw3);
lw4=atoi(&lw4);
lw5=atoi(&lw5);
lw6=atoi(&lw6);
a=a<<2;
a1=a1<<1;
finala=a+a1+a2;
printf("%d\n",a);
printf("%d\n",a1);
printf("%d\n",finala);
op=op<<2;
op1=op1<<1;
opcode=op+op1+op2;

if(opcode==4)  /*SW detected*/
{

    lw=lw<<6;
lw1=lw1<<5;
lw2=lw2<<4;
lw3=lw3<<3;
lw4=lw4<<2;
lw5=lw5<<1;
    Imm=lw+lw1+lw2+lw3+lw4+lw5+lw6;

    fprintf(output,"sw\%d,r0,%x\n",finala,Imm);
}
}  
else if(opcode==0)  /*ADD detected*/
{
    b=b<<2;
    b1=b1<<1;
    finalb=b+b1+b2;
    lw4=lw4<<2;
    lw5=lw5<<1;
    finalc=lw4+lw5+lw6;
    fprintf(output,"add\tr%d,r%d,r%d\n",finala,finalb,finalc);
}
else if(opcode==1)  /*ADDI detected*/
{
    b=b<<2;
    b1=b1<<1;
    finalb=b+b1+b2;
    lw=lw<<6;
    lw1=lw1<<5;
    lw2=lw2<<4;
    lw3=lw3<<3;
    lw4=lw4<<2;
    lw5=lw5<<1;
    Imm=lw+lw1+lw2+lw3+lw4+lw5+lw6;
    fprintf(output,"addi\tr%d,r%d,%x\n",finala,finalb,Imm);
}
else if(opcode==2)  /*NAND detected*/
{
    b=b<<2;

b1 = b1 << 1;
finalb = b + b1 + b2;
lw4 = lw4 << 2;
lw5 = lw5 << 1;
finalc = lw4 + lw5 + lw6;

fprintf(output, "nand\tr%d, r%d, r%d\n", finala, finalb, finalc);
}
else if (opcode == 3) /* LUI detected */
{
  b = b << 9;
b1 = b1 << 8;
b2 = b2 << 7;
lw = lw << 6;
lw1 = lw1 << 5;
lw2 = lw2 << 4;
lw3 = lw3 << 3;
lw4 = lw4 << 2;
lw5 = lw5 << 1;
Imm = b + b1 + b2 + lw + lw1 + lw2 + lw3 + lw4 + lw5 + lw6;

fprintf(output, "lui\tr%d, %x\n", finala, Imm);
}
else if (opcode == 5) /* LW detected */
{
lw = lw << 6;
lw1 = lw1 << 5;
lw2 = lw2 << 4;
lw3 = lw3 << 3;
lw4 = lw4 << 2;
lw5 = lw5 << 1;
Imm = lw + lw1 + lw2 + lw3 + lw4 + lw5 + lw6;

fprintf(output, "lw\tr%d,r0,%x\n", finala, Imm);
}
else if (opcode == 6) /* BEQ detected */
{

b = b << 2;
b1 = b1 << 1;
finalb = b + b1 + b2;

lw = lw << 6;
lw1 = lw1 << 5;
lw2 = lw2 << 4;
lw3 = lw3 << 3;
lw4 = lw4 << 2;
lw5 = lw5 << 1;
Imm = lw + lw1 + lw2 + lw3 + lw4 + lw5 + lw6;

fprintf(output, "beq\tr%d,r%d,%x\n", finala, finalb, Imm);
}
else if (opcode == 7) /* JALR detected */
{

b = b << 2;
b1 = b1 << 1;
finalb = b + b1 + b2;

fprintf(output, "jalr\tr%d,r%d\n", finala, finalb);
else

    fprintf(output,"ERROR!!");

}
fclose(input);
fclose(output);
module full
sel1,test1,intest,WEdat,stall,W40,SRC1,STRout,test,SRC2,FUCalusel,W1,W43,clk,
OP1,OP2,rst,RTw24,MUXoutsel,IR,DATout,inputWD1,Outw21,LS,SE,OP0,aluout,
WD1,WD2,PC1,opcode1,opcode2,r0,r1,r2,r3,r4,r5,r6,r7);

input clk,rst,sel1;
input [15:0] intest;
supply0 gnd;
output [15:0] PC1,test,test1,STRout,W40,W43,W1,OP1,OP2,IR,LS,SE,OP0,aluout,WD1,WD2,inp
utWD1,Outw21,DATout,r0,r1,r2,r3,r4,r5,r6,r7;
output [2:0] opcode1,opcode2,RTw24,SRC1,SRC2;
output [1:0] FUCalusel;
output MUXoutsel,WEdat,stall;
wire [15:0] w1,w2,w3,w4,w5,w8,w10,w11,w18,w19,w21,w25,w26,w27,w30,w31,w32,w33,w38 ,w39,w40,w42,w43,w46,w48,w50,w52,w60;
wire [2:0] w12,w13,w14,w15,w17,w24,w28,w29,w34,w35,w47;
wire [1:0] w6,w36,w37,w41,w45;
wire w7,w9,w16,w20,w22,w44,w49,w51;
// FETCH STAGE
mux3to1 a1(w6, w3, w1, w2, w4);
Program_Counter a2(clk, rst, w4, w5, w7);
inst_mem(w5, w8);
add_one_pc(w5, w2);

// IF/ID
Instruction_Register(clk, w7, w9, w8, w10);
PC_register a3(clk, w5, w11, w7);

// DECODE STAGE
ctl7(w10[15:13], w10[12:10], w10[9:7], w10[2:0], w12, w13, w14, w15, w7);
mux_2to1_3bit_rf a4(w10[12:10], w10[2:0], w16, w17);
left_shift6 a8(w10[9:0], w18);
mux_2to1_16bit_rf(w18, intest, sel1, w60);
sign_extend a9(w10[6:0], w19);
ctl_6 a10(w14, w16, w20);
mux_2to1_16bit_rf a11(w60, w19, w20, w21);
memory_testing(w22, w25, w24, w10[9:7], w17, clk, w27, w26, r0, r1, r2, r3, r4, r5, r6, r7);

// ID/EX
threebit_register a13(clk, w14, w12, w9);
threebit_register a14(clk, w15, w13, w9);
threebit_register a15(clk, w10[9:7], w28, w9);
threebit_register a16(clk, w17, w29, w9);
PC_register a17(clk, w11, w30, w9);
eightbit_register a18(clk, w21, w31, w9);
eightbit_register a19(clk, w26, w32, w9);
eightbit_register a20(clk, w27, w33, w9);
// EXECUTE STAGE
ctl_5(w28,w29,w34,w24,w35,w37,w36);
mux_4to1_8bit2 a22(w36,w32,w38,w25,w39,w40);
mux_4to1_8bit2 a23(w37,w39,w25,w38,w33,w1);
mux3to1 a24(w41, w31, w42 ,w40, w43);
add_one_pc a25(w30,w42);
ctl_4 a26(w12,w41);
add16 a27(w42,w31,gnd,w3);
ctl_3 a28(w12,w44,w45,w6,w9);
ALU (w43,w1,gnd,w45,w44,w46);

// EX/MEM
threebit_register2 a30(clk,w12,w47);
threebit_register2 a31(clk,w13,w34);
PC_register2 a32(clk,w30);
eightbit_register2 a33(clk,w40,w48);
eightbit_register2 a34(clk,w46,w39);

// MEMORY STAGE
data_mem(clk,w39,w48,w49,w50,test,test1);
ctl_2 a35(w47,w49,w51);
mux_2to1_16bit_rf a36(w50,w39,w51,w52);

// MEM/WB
threebit_register2 a37(clk,w34,w24);
eightbit_register2 a38(clk,w52,w25);

// WRITEBACK STAGE
ctl_1(w24,w22);

// WB/END
eightbit_register2 a39(clk,w25,w38);
threebit_register2 a40(clk,w24,w35);
assign IR=w10;
assign PC1=w5;
assign LS=w18;
assign SE=w19;
assign OP0=w31;
assign opcode1=w12;
assign opcode2=w47;
assign aluout=w39;
assign WD1=w25;
assign WD2=w38;
assign MUXoutsel=w51;
assign inputWD1=w52;
assign Outw21=w21;
assign RTw24=w15;
assign DATout=w50;
assign OP2=w32;
assign OP1=w33;
assign W40=w40;
assign W1=w1;
assign W43=w43;
assign FUCalusel=w45;
assign SRC1=w10[9:7];
assign SRC2=w17;
assign WEdat=w49;
assign STRout=w48;
assign stall=w7;
endmodule
Test patterns generated by Tetramax

STIL 1.0 { Design 2005; }
Header {
Title " TetraMAX (TM) Z-2007.03-SP5-i071016_155731 STIL output";
Date "Wed Mar 10 16:33:13 2010";
History {
Ann {* Uncollapsed Stuck Fault Summary Report *}
Ann {* ---------------------------------------------- *}
Ann {* fault class code #faults *}
Ann {* ---------------------------------------------- *}
Ann {* Detected DT 10510 *}
Ann {* Possibly detected PT 461 *}
Ann {* Undetectable UD 94 *}
Ann {* ATPG untestable AU 35 *}
Ann {* Not detected ND 5362 *}
Ann {* ---------------------------------------------- *}
Ann {* total faults 16462 *}
Ann {* test coverage 66.03% *}
Ann {* ---------------------------------------------- *}
Ann {* *}
Ann {* Pattern Summary Report *}
Ann {* ---------------------------------------------- *}
Ann {* #internal patterns 16 *}
Ann {* #full_sequential patterns 16 *}
Ann {* ---------------------------------------------- *}
Ann {* *}
Ann {* rule severity #fails description *}
Ann {* ---- ------ ------ ---------------------------------------------- *}
Ann {* B7 warning 7 undriven module output pin *}
Ann {* B8 warning 10 unconnected module input pin *}
Ann {* B9 warning 6 undriven module internal net *}
Ann {* B10 warning 12 unconnected module internal net *}
Ann {* C16 warning 16 nonscan cell port unable to capture *}
Ann {* C25 warning 1 unstable cell clock input connected from multiple sources (nomask) *}
Ann {* *}
Ann {* clock_name     off period LE_time TE_time meas_time usage *}
Ann {* -------------------- ----------------- ----------------- -------------------- *}
Ann {* clock   0    100ns  50   80    40  reset nonscan_DFF *}
Ann {* *}
Ann {* There are no constraint ports *}
Ann {* There are no equivalent pins *}
Ann {* There are no net connections *}

Signals {
  "clock" In; "Prin[15]" In; "Prin[14]" In; "Prin[13]" In; "Prin[12]" In; "Prin[11]" In;
  "Prin[10]" In; "Prin[9]" In; "Prin[8]" In; "Prin[7]" In; "Prin[6]" In; "Prin[5]" In;
  "Prin[4]" In; "Prin[3]" In; "Prin[2]" In; "Prin[1]" In; "Prin[0]" In; "data_keluar[15]" In;
  "data_keluar[14]" In; "data_keluar[13]" In; "data_keluar[12]" In; "dataKeluar[11]" In;
  "data_keluar[10]" In; "data_keluar[9]" In; "data_keluar[8]" In; "data_keluar[7]" In;
  "data_keluar[6]" In; "data_keluar[5]" In; "data_keluar[4]" In; "data_keluar[3]" In;
  "data_keluar[2]" In; "data_keluar[1]" In; "data_keluar[0]" In; "O[15]" Out; "O[14]" Out;
  "O[13]" Out; "O[12]" Out; "O[11]" Out; "O[10]" Out; "O[9]" Out; "O[8]" Out;
  "O[7]" Out;
  "O[6]" Out; "O[5]" Out; "O[4]" Out; "O[3]" Out; "O[2]" Out; "O[1]" Out; "O[0]" Out;
  "Primout[15]" Out; "Primout[14]" Out; "Primout[13]" Out; "Primout[12]" Out;
  "Primout[11]" Out;
  "Primout[10]" Out; "Primout[9]" Out; "Primout[8]" Out; "Primout[7]" Out;
  "Primout[6]" Out;
  "Primout[5]" Out; "Primout[4]" Out; "Primout[3]" Out; "Primout[2]" Out;
  "Primout[1]" Out;
  "Primout[0]" Out;
}

SignalGroups {
  "_pi" = "clock" + "Prin[15]" + "Prin[14]" + "Prin[13]" + "Prin[12]" +
    "data_keluar[15]" + "data_keluar[14]" + "data_keluar[13]" + "data_keluar[12]" +
    "data_keluar[11]" + "data_keluar[10]" + "data_keluar[9]" + "data_keluar[8]" +
    "data_keluar[7]" + "data_keluar[6]" + "data_keluar[5]" + "data_keluar[4]" +
    "data_keluar[3]" + "data_keluar[2]" + "data_keluar[1]" + "data_keluar[0]";
  #signals=33
  "_in" = "clock" + "Prin[15]" + "Prin[14]" + "Prin[13]" + "Prin[12]" +
    "data_keluar[15]" + "data_keluar[14]" + "data_keluar[13]" + "data_keluar[12]" +
    "data_keluar[11]" + "data_keluar[10]" + "data_keluar[9]" + "data_keluar[8]" +
    "data_keluar[7]" + "data_keluar[6]" + "data_keluar[5]" + "data_keluar[4]" +
    "data_keluar[3]" + "data_keluar[2]" + "data_keluar[1]" + "data_keluar[0]";
  #signals=33


"_default_Clk0_Timing_" = "clock";

Timing {
  WaveformTable "_default_WFT_" {
    Period '100ns';
    Waveforms {
      "_default_In_Timing_" { 0 { '0ns' D; } }
      "_default_In_Timing_" { 1 { '0ns' U; } }
      "_default_In_Timing_" { Z { '0ns' Z; } }
      "_default_In_Timing_" { N { '0ns' N; } }
      "_default_Clk0_Timing_" { P { '0ns' D; '50ns' U; '80ns' D; } }
      "_default_Out_Timing_" { X { '0ns' X; } }
      "_default_Out_Timing_" { H { '0ns' X; '40ns' H; } }
      "_default_Out_Timing_" { T { '0ns' X; '40ns' T; } }
      "_default_Out_Timing_" { L { '0ns' X; '40ns' L; } }
    }
  }
}

ScanStructures {
  // Uncomment and modify the following to suit your design
  // ScanChain "chain_name" { ScanIn "chain_input_name"; ScanOut "chain_output_name"; }
}

PatternBurst "_burst_" {
  PatList { "_pattern_" {

PatternExec {
    PatternBurst "_burst_";
}

Procedures {
    "capture_clock" {
        W "_default_WFT_";
        C { "Prin[9]"=0; "Prin[14]"=0; "data_keluar[10]"=0; "Prin[0]"=0; "data_keluar[3]"=0;
            "Prin[15]"=0; "data_keluar[11]"=0; "Prin[1]"=0; "data_keluar[4]"=0;
            "data_keluar[12]"=0;
            "Prin[2]"=0; "data_keluar[5]"=0; "data_keluar[13]"=0; "Prin[3]"=0;
            "data_keluar[6]"=0;
            "data_keluar[14]"=0; "clock"=0; "Prin[4]"=0; "data_keluar[7]"=0;
            "data_keluar[15]"=0;
            "Prin[5]"=0; "Prin[10]"=0; "data_keluar[8]"=0; "Prin[6]"=0; "Prin[11]"=0;
            "data_keluar[9]"=0;
            "data_keluar[0]"=0; "Prin[7]"=0; "Prin[12]"=0; "data_keluar[1]"=0;
            "Prin[8]"=0; "Prin[13]"=0;
            "data_keluar[2]"=0; } } 
        "forcePI": V { "_pi"=r33 # ; "_po"=r32 X ; }
        "measurePO": V { "_po"=r32 # ; }
        "pulse": V { "clock"=P; "_po"=r32 X ; }
    }

    "capture" {
        W "_default_WFT_";
        C { "Prin[9]"=0; "Prin[14]"=0; "data_keluar[10]"=0; "Prin[0]"=0; "data_keluar[3]"=0;
            "Prin[15]"=0; "data_keluar[11]"=0; "Prin[1]"=0; "data_keluar[4]"=0;
            "data_keluar[12]"=0;
            "Prin[2]"=0; "data_keluar[5]"=0; "data_keluar[13]"=0; "Prin[3]"=0;
            "data_keluar[6]"=0;
            "data_keluar[14]"=0; "clock"=0; "Prin[4]"=0; "data_keluar[7]"=0;
            "data_keluar[15]"=0;
            "Prin[5]"=0; "Prin[10]"=0; "data_keluar[8]"=0; "Prin[6]"=0; "Prin[11]"=0;
            "data_keluar[9]"=0;
            "data_keluar[0]"=0; "Prin[7]"=0; "Prin[12]"=0; "data_keluar[1]"=0;
            "Prin[8]"=0; "Prin[13]"=0;
            "data_keluar[2]"=0; } } 
        "forcePI": V { "_pi"=r33 # ; "_po"=r32 X ; }
        "measurePO": V { "_po"=r32 # ; }
    }

    // Uncomment and modify the following to suit your design
    // load_unload {
    //     V { "clock" = 0; } // force clocks off and scan enable pins active
    //     Shift { V { _si=#; _so=#; "clock" = P; } } // pulse shift clocks
    // }
}

MacroDefs {
"test_setup" {
  W "_default_WFT_";
  V { "clock"=0; }
}

Pattern "._pattern_" {
  W "_default_WFT_";
  "precondition all Signals": C { "_pi"=r33 0 ; "_po"=r32 X ; }
  Macro "test_setup";
  Ann {* full_sequential *}
  "pattern 0": V { "_pi"=0100001011000100101101000111101; }
                V { "_po"=HLLLLLLLLLHLLLHXXXXXXXXXXXXXXXXXXX; }
                C { "_po"=r32 X; }
                V { "clock"=P; }
                V { "_pi"=00001010101001001011010001001101; }
                V { "_po"=LHHHLLLLLLHLHLLLLLLLLLLLLLLLLLLLLL; }
                C { "_po"=r32 X; }
                V { "clock"=P; }
                V { "_pi"=00110111101011000110011100111001; }
                V { "_po"=LHLLLLLHHHLHLLLLLLLLLLLLLLLLLLLLL; }
                C { "_po"=r32 X; }
                V { "clock"=P; }
                V { "_pi"=00011111101011010001100100101001; }
                V { "_po"=LHHHHHHHLLLLLLLLLLLLLLLLLLLLL; }
                C { "_po"=r32 X; }
                V { "clock"=P; }
                V { "_pi"=01001100100111101010001000110110; }
                V { "_po"=HLLLHLLLLLLLHLLLLLLLLL; }
                C { "_po"=r32 X; }
                V { "clock"=P; }
                V { "_pi"=0000000101011101110100010100101011; }
                V { "_po"=LLLLLLLLHLLLLHLLLLLLLLLLLLLLLLL; }
                C { "_po"=r32 X; }
                V { "clock"=P; }
                V { "_pi"=000000001001110111010000010010100; }
                V { "_po"=LHHHLLLLLLLLLLLLLLLLLLLLL; }
                C { "_po"=r32 X; }
                V { "clock"=P; }
                V { "_pi"=00100001101111011010100101010010; }
                V { "_po"=LHHHLLLLLLLHLLLLLLLLLLLLLLLLLL; }
                C { "_po"=r32 X; }
                V { "clock"=P; }
                V { "_pi"=00101111110100000100011110101010; }
                V { "_po"=LHHHHHHHHLLLLLLLLLLLLLLLLLLL; }
                C { "_po"=r32 X; }
                V { "clock"=P; }
                V { "_pi"=01000001110000010101100011001001; }
                V { "_po"=HLLLLLLLLŁLHLLLLLLLLLLLLLLLLLL; }
                C { "_po"=r32 X; }
                V { "clock"=P; }
}

V { "_pi"=00100001011000100101101000111101; }
V { "_po"=HLLLLLLLLLHLLLHXXXXXXXXXXXXXXXXXXX; }
C { "_po"=r32 X; }
V { "clock"=P; }

V { "_po"=HLLLLLLLLLLHHHLLLLLLLLLLLLLLLLL; }
C { "_po"=r32 X; }
V { "_po"=P; }
V { "_pi"=00000010110011001001001100110001; }
V { "_po"=LLLLLLLLHHHLLLLLLLLLLLLLLLLL; }
C { "_po"=r32 X; }
V { "_po"=P; }
V { "_pi"=010011011101001001011111100; }
V { "_po"=LLLLLLLLHHHLLLLLLLLLLLLLLLLL; }
C { "_po"=r32 X; }
V { "_po"=P; }
V { "_pi"=00101111011100111111111100; }
"end 1 measurePO": V {
"_po"=LHLHLLLLLLLLLLLLLHLLLLLHHHLLL; }
C { "_po"=r32 X; }
Ann {* full_sequential *}
"pattern 2": V { "_pi"=0100101001100100011010100111001100; }
V { "_po"=LLLLLLLLHHHLLLLLLLLLLLLLLLLL; }
C { "_po"=r32 X; }
V { "_po"=P; }
V { "_pi"=01111111010000000100110011111110; }
V { "_po"=LHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH; }
C { "_po"=r32 X; }
V { "_po"=P; }
V { "_pi"=00111110011101011001010011100101; }
V { "_po"=LHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH; }
C { "_po"=r32 X; }
V { "_po"=P; }
V { "_pi"=00000000011101000001101011011100; }
V { "_po"=LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL; }
C { "_po"=r32 X; }
V { "_po"=P; }
V { "_pi"=0000100100011111111111111100; }
V { "_po"=LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL; }
C { "_po"=r32 X; }
V { "_po"=P; }
V { "_pi"=01001111110001001101100110101101; }
V { "_po"=LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL; }
C { "_po"=r32 X; }
V { "_po"=P; }
V { "_pi"=011011001100011111111111111100; }
V { "_po"=HHHLLLLLLHHHLLLLLHHHLLLLLLLLLLLLLLL; }
C { "_po"=r32 X; }
V { "_po"=P; }
V { "_pi"=011011001100011111111111111100; }
V { "_po"=HHHLLLLLLHHHLLLLLHHHLLLLLLLLLLLLLLL; }
C { "_po"=r32 X; }
V { "_po"=P; }

"end 2 measurePO": V {
  "_po"=LHLHHHHLHHLLLHHHLLHLLLHLLLLLLL; }
C { "_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=001011101101001111001100000100101; }
"pattern 3": V { "_pi"=0000111100011100011000110111000; }
V { "_po"=LLLLHHHLLLLLLHHHLLLHLLLHLLLHLLLLLLL; }
C { "_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=00000101001110110100110001000001; }
V { "_po"=LHLHHHHHLHLLLLLLLLLLLLLLLLLLLL; }
C { "_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=001111010110000011000101111001; }
V { "_po"=LHHHHHHHLHLLLLLLLLLLLLLLLLLLL; }
C { "_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=010111011000111000110110111100; }
V { "_po"=HLHLLLLLLLLHHHLLLHLLLHLLLHLLLHLLLHLLL; }
C { "_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=01011101001010001111111001011111; }
V { "_po"=HLHHHHHHLHLLLLLLLLLLLLLLLLLLL; }
C { "_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=00000000011011111001111000110101; }
V { "_po"=LHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH; }
C { "_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=010011110111101001100101111001; }
V { "_po"=HLHLLLLLLLHHLHLLLLLLLLLLLLLLLLLLL; }
C { "_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=0100101001111101001010001111111; }
V { "_po"=HLHLLLLLLLHHLHLLLLLLLLLLLLLLLLLLL; }
C { "_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=0110011011010110011001000001101; }
V { "_po"=HHLHLHLLLLHHLHLLLLLLLLLLLLLLLLLLL; }
C { "_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=0110011011010110011001000001101; }
V { "_po"=HHLHLHLLLLHHLHLLLLLLLLLLLLLLLLLLL; }
C { "_po"=r32 X; }
v { ".pi"=01001111100011101110011010101111; }
v { ".po"=HLLLHHHHLLLLHHHLHLLLLLLLLLLLLLLL; }
c { ".po"=x32 x; }
v { "clock"=p; }
v { ".pi"=0111000011001101111100011010011; }
v { ".po"=HHHLLLHHLHHLHLLLLLLLLLLLLLLLLLL; }
c { ".po"=x32 x; }
v { "clock"=p; }
v { ".pi"=0111001000011010001111101101101; }
v { ".po"=HHHLLLHHLHHLHLLLLLLLLLLLLLLLLLLL; }
c { ".po"=x32 x; }
v { "clock"=p; }
v { ".pi"=001001111011100100100100011111; }
"end 4 measurePO": v {
    ".po"=LHLLLHHLHLLLHLHLLLLHHHHHHHHH; }
c { ".po"=x32 x; }
ann {* full_sequential *}
"pattern 5": v { ".pi"=00001011011111101001000110; }
v { ".po"=LLLLHHLHLLLLLHHHHLLLLLLLLLLLLLLLL; }
c { ".po"=x32 x; }
v { "clock"=p; }
v { ".pi"=0001010111111110101111101101; }
v { ".po"=LLLLHHLHLLLLLHHHLLLLLLLLLLLLLLLL; }
c { ".po"=x32 x; }
v { "clock"=p; }
v { ".pi"=0011011010000000011110100100011; }
v { ".po"=LHHLHHLHLHLLLLLLLLLLLLLLLLLLLLLLL; }
c { ".po"=x32 x; }
v { "clock"=p; }
v { ".pi"=001111011111111001000000011111; }
v { ".po"=LHHLHHLHHLHLLLLLLLLLLLLLLLLLLLLLL; }
c { ".po"=x32 x; }
v { "clock"=p; }
v { ".pi"=0011110111111110011000001101011; }
v { ".po"=LHHLHHLHHLHLLLLLLLLLLLLLLLLLLLLLLL; }
c { ".po"=x32 x; }
v { "clock"=p; }
v { ".pi"=00111111111111110111000011010111; }
v { ".po"=LHHLHHLHHLHLLLLLLLLLLLLLLLLLLLLLLL; }
c { ".po"=x32 x; }
v { "clock"=p; }
v { ".pi"=0101010101011111101101011111; }
v { ".po"=HLHLHHLHHLHLLLLLLLLLLLLLLLLLLLLL; }
c { ".po"=x32 x; }
v { "clock"=p; }
v { ".pi"=00100101011000000011011000000101; }
v { ".po"=LHHLHHLHHLHLLLLLLLLLLLLLLLLLLLLLLL; }
c { ".po"=x32 x; }
v { "clock"=p; }
v { ".pi"=0001001111111010010011011100; }
v { ".po"=LHHLHHLHHLHLLLLLLLLLLLLLLLLLLLLLLL; }
c { ".po"=x32 x; }
v { "clock"=p; }
v { ".pi"=0010000011001011111111011101101; }
}
V { ";_po"=LHLLLLHHLLLLHLLLLLLLLLLLLLLLLLL; } 
C { "_po"=r32 X; } 
V { "clock"=P; } 
V { "_pi"=001011111010000010010111001000; } 
V { ";_po"=LHLLLLHHLLLLLHLLLLLLLLLLLLLLLLLL; } 
C { ";_po"=r32 X; } 
V { "clock"=P; } 
V { "_pi"=000101001000100011010101111111; } 
V { ";_po"=LLLLLLLLLLLLLLLLLLLLLLLLLLLLLL; } 
C { ";_po"=r32 X; } 
V { "clock"=P; } 
V { "_pi"=0100101110010001101100011010001; } 
V { ";_po"=HLLHLHLLLLLLLLHLLLLLLLLLL; } 
C { ";_po"=r32 X; } 
V { "clock"=P; } 
V { "_pi"=001101110101111010111101000101; } 
V { ";_po"=HLLLLLHHHHHLHHLHHLHHLHHLHHLHHLHHLHHLH; } 
C { ";_po"=r32 X; } 
V { "clock"=P; } 
V { "_pi"=01000101001001110001100000110011; } 
V { ";_po"=HLLLLLLLLLLLLLLLLLLLLLLLLLLLL; } 
C { ";_po"=r32 X; } 
V { "clock"=P; } 
V { "_pi"=00011000110100011010110101100101; } 
V { ";_po"=HLLLLLHHHHHHLHHLHHLHHLHHLHHLHHLHHLH; } 
C { "_po"=LHLLLLLLHLLLLLHLLLLLHHLHHLHHLHHLHHLH; } 
Ann { * full_sequential * } 
"pattern 6": V { ";_pi"=010001011110101100000001100000; } 
V { ";_po"=HLLLLLLLLLLLLLLLLLLLLLLLLLLLL; } 
C { ";_po"=r32 X; } 
V { "clock"=P; } 
V { "_pi"=010001001001011100110010001111; } 
V { ";_po"=HLLLLLLLLLLLLLLLLLLLLLLLLLLLL; } 
C { ";_po"=r32 X; } 
V { "clock"=P; } 
V { "_pi"=00110000110100011010110110100101; } 
V { ";_po"=LHLLLLLHLHLLLLLLLLLLLLLLLLLLLLL; } 
C { "_po"=r32 X; } 
V { "clock"=P; } 
V { "_pi"=00111010001001000110101010101111; } 
V { ";_po"=LHLLLLLHLHLLLLLHLLLLLLLLLLLLLLLLLL; } 
C { ";_po"=r32 X; } 
V { "clock"=P; } 
V { "_pi"=000111011000110100101110110011011; } 
V { ";_po"=LHLLLLLHLHLLLLLHLLLLLLLLLLLLLLLLLL; } 
C { ";_po"=r32 X; } 
V { "clock"=P; } 
V { "_pi"=000111011000110100101110110011011; } 
V { ";_po"=LHLLLLLHLHLLLLLHLLLLLLLLLLLLLLLLLL; }
C { ".po"="r32 X; } 
V { "clock"="P; } 
V { ".pi"=0000000100111011011010101; } 
V { ".po"=LLLLLHLLLLLHLLLLLHLLLLLLllllllll; } 
C { ".po"="r32 X; } 
V { "clock"="P; } 
V { ".pi"=0100101100100110111011001110101; } 
V { ".po"=LLLLLHLLLLLHLLLLLHLLLLLllllllllll; } 
C { ".po"="r32 X; } 
V { "clock"="P; } 
V { ".pi"=0100101100111101001110101110110; } 
V { ".po"=LLLLLHLLLLLHLLLLLHLLLLLllllllllll; } 
C { ".po"="r32 X; } 
V { "clock"="P; } 
V { ".pi"=0011100100010110110111101110111; } 
V { ".po"=LLLLLHLLLLLHLLLLLHLLLLLllllllllll; } 
C { ".po"="r32 X; } 
V { "clock"="P; } 
V { ".pi"=0011100010010110110101111011110; } 
V { ".po"=LLLLLHLLLLLHLLLLLHLLLLLllllllllll; } 
C { ".po"="r32 X; } 
V { "clock"="P; } 
V { ".pi"=0011111000101111011100111011110; } 
V { ".po"=LLLLLHLLLLLHLLLLLHLLLLLllllllllll; } 
C { ".po"="r32 X; } 
V { "clock"="P; } 
V { ".pi"=0011111000101111011100111011110; } 
V { ".po"=LLLLLHLLLLLHLLLLLHLLLLLllllllllll; } 
C { ".po"="r32 X; } 
V { "clock"="P; } 
V { ".pi"=0011111000101111011100111011110; } 
V { ".po"=LLLLLHLLLLLHLLLLLHLLLLLllllllllll; } 
C { ".po"="r32 X; } 
V { "clock"="P; } 
V { "end 6 measurePO": V { 
"_po"=LLLLLHLLLLLHLLLLLHLLLLLllllllllll; } 
C { ".po"="r32 X; } 
Ann {* full_sequential *} 
"pattern 7": V { ".pi"=00001110010100000111001110101011; } 
V { ".po"=LLLLLHLLLLLHLLLLLHLLLLLllllllllll; } 
C { ".po"="r32 X; } 
V { "clock"="P; } 
V { ".pi"=0010111011101110111011011110111; } 
V { ".po"=LLLLLHLLLLLHLLLLLHLLLLLllllllllll; } 
C { ".po"="r32 X; } 
V { "clock"="P; } 
V { ".pi"=0010111011011110110111011110111; } 
V { ".po"=LLLLLHLLLLLHLLLLLHLLLLLllllllllll; } 
C { ".po"="r32 X; } 
V { "clock"="P; } 
V { ".pi"=0001011101111011011111110111101; } 
V { ".po"=LLLLLHLLLLLHLLLLLHLLLLLllllllllll; } 
C { ".po"="r32 X; } 
V { "clock"="P; } 
V { ".pi"=0001011101111101110111101110111; } 
V { ".po"=LLLLLHLLLLLHLLLLLHLLLLLllllllllll; } 
C { ".po"="r32 X; } 
V { "clock"="P; }
V { "_po"=HLLHHHLLLLLHLLHLLLLLLLLLLLLLLLL; }  
C { "_po"=r32 X; }  
V { "clock"=P; }  
V { "_pi"=001001110101010101101100110001100; }  
V { "_po"=LHHHHHLHLLLLLLLLLLLLLLLLLLLLLLLL; }  
C { "_po"=r32 X; }  
V { "clock"=P; }  
V { "_pi"=001111101000010100000001011010100; }  
V { "_po"=HLLHHHLLLLLHLLLLLLLLLLLLLLLLLLLL; }  
C { "_po"=r32 X; }  
V { "clock"=P; }  
V { "_pi"=010011011011010001111111010100; }  
V { "_po"=HLLHHHLLLLLHLLLLLLLLLLLLLLLLLLLL; }  
C { "_po"=r32 X; }  
V { "clock"=P; }  
V { "_pi"=001011010100000001000000010101011; }  
"end 9 measurePO": V {  
"_po"=LHLHHLHHLHHLLLLHHHHHHHHHHHHH; }  
C { "_po"=r32 X; }  
Ann { * full_sequential * }  
"pattern 10": V { "_pi"=000010111001100011011000001100; }  
V { "_po"=LLLLHHHLLLLLLLLLLLLLLLLLLLLLLLL; }  
C { "_po"=r32 X; }  
V { "clock"=P; }  
V { "_pi"=0011110101011010010201011011001101100; }  
V { "_po"=LHHHHHLLLLLLLLLLLLLLLLLHHHHHHL; }  
C { "_po"=r32 X; }  
V { "clock"=P; }  
V { "_pi"=010011011011010001111111010100; }  
V { "_po"=HLLHHHLLLLLHLLLLLLLLLLLLLLLLLLLL; }  
C { "_po"=r32 X; }  
V { "clock"=P; }  
V { "_pi"=0101100110110101110110011000110001; }  
V { "_po"=HLLHHHLLLLLHLLLLLLLLLLLLLLLLLLLL; }  
C { "_po"=r32 X; }  
V { "clock"=P; }  
V { "_pi"=010011001101101100111001100110001; }  
V { "_po"=HLLHHHLLLLLHLLLLLLLLLLLLLLLLLLLL; }  
C { "_po"=r32 X; }  
V { "clock"=P; }  
V { "_pi"=010011001101101101101110011001110; }  
V { "_po"=HLLLHHHLLLHLLLLLLLLLLLLLLLLLLLL; }  
C { "_po"=r32 X; }  
V { "clock"=P; }  
V { "_pi"=000001101111110101101000010100; }  
V { "_po"=LLLLHHHLLLLLLLLLLLLLLLLLHHHHHHL; }  
C { "_po"=r32 X; }  
V { "clock"=P; }  
V { "_pi"=0100100110101101101101101101001011; }  
V { "_po"=HLLHHHLLLLLHLLLLLLLLLLLLLLLLLLLL; }  
C { "_po"=r32 X; }  
V { "clock"=P; }  
V { "_pi"=0101100011111011010110100010100111; }  
V { "_po"=LHLHLLLHLLLLLLLLLLLLLLLLLLLLL; }  
C { "_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=00111000000101110011111100010102; }
V { "_po"=LHHHLLLHLLHLHLLHLLLLLHLLHLH; }  
C { "_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=0111010001111010000111011101001010; }  
"end 10 measurePO": V {
  "_po"=LHHHLLLHLLHLHLLHLLLLLHLLHLH;
}
C { "_po"=r32 X; }
Ann {* full_sequential *}
"pattern 11": V { "_pi"=01001011111101000111000110110100; }
V { "_po"=HLLLHLLLHLLLLLHLHLLLLLHLLLHLH;
}
C { "_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=0011010001011011110111111000101000; }
V { "_po"=LHHHLLHLHLLHHLLHLLLLLHLLLHLH;
}
C { "_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=0011101001011101111011000110010100; }
V { "_po"=LHHHLLLLHHLHLLLLLHHLLLLHLH;
}
C { "_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=01001010010001010001010101010; }
V { "_po"=HLLLHLLLLLHHLHLLLLLHLLLLHLH;
}
C { "_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=010000101000110001010100100010110; }
V { "_po"=HLLLLLLLLLHHLHLLLLLLLLLLHLH;
}
C { "_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=0100100110110100101010100101010; }
V { "_po"=HLLLLLLLLHHLHLLHLHLLLLLLLLL;
}
C { "_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=00100000111110111101011111110001010; }
V { "_po"=LHHHLLLLHHLHLLLHLLLLLHLLLHLH;
}
C { "_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=00101110100111000011011010100101001; }
V { "_po"=HLLLLLLLLHHLHLLLLLHLHLLLLLHLH;
}
C { "_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=0100101001010101010101010101010; }
V { "_po"=HLLLHLLLHHLHLLLHLLLLLHLLLHLH;
}
C { "_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=0100101001010101010101010101010; }
V { "_po"=HLLLLLLHLHLLLHLLLHLLLLLHLLLHLH;
}
C { "_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=0011010010000101010101010101010; }
V { "_po"=LHHHHLHLLLLLHLLLHLLLLLHLLLHLH;
}
C { "_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=0100101010011000010101110110010100; }
V { "_po"=HLLLHLLLLLHHLHLLLLLHLLLHLH;
}
V { "_pi"=0100101100111111010011010001010; } V { "_po"=HLLLHLLLHLLLLLLLLLLLLLLLLLLLLL; } C { "_po"=r32 X; } V { "clock"=P; } V { "_pi"=010011101111111111110011010100000; } V { "_po"=HLLLHHHLLLLLHLLLLLLLLLLLLLLLLL; } C { "_po"=r32 X; } V { "clock"=P; } V { "_pi"=001111010000000001100110110010001; } V { "_po"=LHHHHHLLLLLLLLLLHLHLLHHHLH; } C { "_po"=r32 X; } V { "clock"=P; } V { "_pi"=0011100100000111100010011000101; } V { "_po"=LLHHLLHHLLLLLLLLHHHHLLLLHHHLH; } C { "_po"=r32 X; } V { "clock"=P; } V { "_pi"=00000000001110111101110110101; } "end 11 measurePO" : V { "_po"=LLLLLLLLLLLLLLLLHLLHLLLHLLHHHLL; } C { "_po"=r32 X; } Ann {* full_sequential *} "pattern 12" : V { "_pi"=000001011100010111010001100010101; } V { "_po"=LLLLHLHHHLLLLLHHLHLLHHHHLHHHLL; } C { "_po"=r32 X; } V { "clock"=P; } V { "_pi"=01000010111110001001100010101; } V { "_po"=LLLLHLLLLHHHHLLLLHLHLLLHHLH; } C { "_po"=r32 X; } V { "clock"=P; } V { "_pi"=001111100100110011010001010; } V { "_po"=HLLLHLLLHLLLLLHHHLLLLLLLLLHLLL; } C { "_po"=r32 X; } V { "clock"=P; } V { "_pi"=01111110011001100110000101100; } V { "_po"=HHHHHLLLLHHHLLLLLLLLLLLLLLLLLLL; } C { "_po"=r32 X; } V { "clock"=P; } V { "_pi"=0000001010001001100100100000; } V { "_po"=LLLLLHLHLLLLHHXXXXXXXXXXXXXX; } C { "_po"=r32 X; } V { "clock"=P; } V { "_pi"=00001000111001011010001011111; } V { "_po"=LLLLLHLLLHLLLHLLLHLLLLLHLLLHLLL; } C { "_po"=r32 X; } V { "clock"=P; } V { "_pi"=010011101111101111100110010011010; } V { "_po"=HLLLHLLLHLLHLLLLLHHHHHLLLLLHLLL; } C { "_po"=r32 X; } V { "clock"=P; } V { "_pi"=0010100100011000101101110100111; }
V { "_po"=LHLHLLLLLLLLLLLLLLLLLLLLLLLLLLL; }  
C { "_po"=r32 X; }  
V { "clock"=P; }  
V { "_pi"=00001101100100111001101000100011; }  
V { "_po"=LHLHLLLLLLLLLLLLLLLLLLLLLLLLLLL; }  
C { "_po"=r32 X; }  
V { "clock"=P; }  
V { "_pi"=00010111000011111111100110101111; }  
"end 12 measurePO": V {  
"_po"=LHLHLLLLLLLLLLLLLLLLLLLLLLLLLLL; }  
C { "_po"=r32 X; }  
Ann { * full_sequential *}

"pattern 13": V { "_pi"=01001101000011010101011011010111; }  
V { "_po"=HLLLHLLLLHHHLLLLLLLLLLLLLLLLLLL; }  
C { "_po"=r32 X; }  
V { "clock"=P; }  
V { "_pi"=00110010001000010100010001000000; }  
V { "_po"=LHLHLLLLLHHHLLLLLLLLLLLLLLLLLLL; }  
C { "_po"=r32 X; }  
V { "clock"=P; }  
V { "_pi"=0100110110010110110111011101011; }  
V { "_po"=HLLLHLLLLHHHLLLLLLLLLLLLLLLLLLL; }  
C { "_po"=r32 X; }  
V { "clock"=P; }  
V { "_pi"=0000100001001100010001011001011; }  
V { "_po"=LHLHLLLLLHHHLLLLLLLLLLLLLLLLLLL; }  
C { "_po"=r32 X; }  
V { "clock"=P; }  
V { "_pi"=00000100001000110111011001001011; }  
V { "_po"=LHLHLLLLLHHHLLLLLLLLLLLLLLLLLLL; }  
C { "_po"=r32 X; }  
V { "clock"=P; }  
V { "_pi"=010000010100001011000010110000101; }  
V { "_po"=HLLLHLLLLLHHHLLLLLLLLLLLLLLLLLLL; }  
C { "_po"=r32 X; }  
V { "clock"=P; }  
V { "_pi"=01000100001001010010011001010101; }  
V { "_po"=HLLLHLLLLLHHHLLLLLLLLLLLLLLLLLLL; }  
C { "_po"=r32 X; }  
V { "clock"=P; }  
V { "_pi"=00000100001100000111111111101110; }  
V { "_po"=LHLHLLLLLHHHLLLLLLLLLLLLLLLLLLL; }  
C { "_po"=r32 X; }  
V { "clock"=P; }  
V { "_pi"=01000100110010100100110010011010; }  
V { "_po"=HLLLHLLLLLHHHLLLLLLLLLLLLLLLLLLL; }  
C { "_po"=r32 X; }  
V { "clock"=P; }  
V { "_pi"=00010010000101110011111010010100; }
"end 13 measurePO": V {
"_po"=LHLLLHLLHLLLLLLLLHHHHLHHHHHHHHH; }
C { 
"_po"=r32 X; }
Ann {* full_sequential *}
"pattern 14": V {
"_pi"=0000011001101110000001010110101011; }
V {
"_po"=LLLLLLLHLLLLLLLLLLLLLLLLLLLLLL; }
C { 
"_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=00110011000000010110011111101; }
V { "_po"=LHLLHHHLHHHHHHHHLHLHHHLHHHHHHL; }
C { 
"_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=0100100110000111010111011101101; }
V { "_po"=LHLLLLHHHHHHHHHHHHHHHHHHHHHHHHH; }
C { 
"_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=0110100100011000100010011001101; }
V { "_po"=LHLLHHHLHHHHHHHHHHHHHHHHHHHHHHH; }
C { 
"_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=001111111010011101011110111101; }
V { "_po"=LHLLHHHLHHHHHHHHLHLHHHLHHHHHHH; }
C { 
"_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=001010011000011101011101101101; }
V { "_po"=LHLLHHHLHHHHHHHHLHLHHHLHHHHHHH; }
C { 
"_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=01101000011110101101101100011; }
V { "_po"=LHLLHHHLHHHHHHHHLHLHHHLHHHHHHH; }
C { 
"_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=001101000110001011011101011101; }
V { "_po"=LHLLHHHLHHHHHHHHLHLHHHLHHHHHHH; }
C { 
"_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=010010011001100010011001100110; }
V { "_po"=LHLLHHHLHHHHHHHHLHLHHHLHHHHHHH; }
C { 
"_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=0001000011000011110011000100011; }
V { "_po"=LHLLHHHLHHHHHHHHLHLHHHLHHHHHHH; }
C { 
"_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=000101111011100000001100101110100; }
V { "_po"=LHLLHHHLHHHHHHHHLHLHHHLHHHHHHH; }
C { 
"_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=010010001100100100010001001000; }
V { "_po"=LHLLHHHLHHHHHHHHLHLHHHLHHHHHHH; }
C { 
"_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=00000001110111111001101100001; }
V { "_po"=LHLLHHHLHHHHHHHHLHLHHHLHHHHHHH; }
C { 
"_po"=r32 X; }
V { "clock"=P; }
C { 
"_po"=r32 X; 
} 
V { 
"clock"=P; 
} 
V { 
"_pi"=0100111000110111110100001001101; 
} 
V { 
"_po"=HLLLLLHLLLLHLLLLLHLLLLLLLLLLLLLLLL; 
} 
C { 
"_po"=r32 X; 
} 
V { 
"clock"=P; 
} 
V { 
"_pi"=011001100011001110000011101101; 
} 
V { 
"_po"=HLLLLLHLLLLHLLLLLHLLLLLLLLLLLLLLLL; 
} 
C { 
"_po"=r32 X; 
} 
V { 
"clock"=P; 
} 
V { 
"_pi"=001101001000011111111001010111; 
} 
V { 
"_po"=LHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH; 
} 
C { 
"_po"=r32 X; 
} 
Ann { 
* full_sequential * 
} 
"pattern 15": V { 
"_pi"=000000111011111000110000101010111; 
} 
V { 
"_po"=LLLLLHLLLLLHLLLLLHLLLLLHLLLLLHLLLLL; 
} 
C { 
"_po"=r32 X; 
} 
V { 
"clock"=P; 
} 
V { 
"_pi"=000001011001101101100000000000; 
} 
V { 
"_po"=LLLLLHLLLLLHLLLLLHLLLLLHLLLLLHLLLLL; 
} 
C { 
"_po"=r32 X; 
} 
V { 
"clock"=P; 
} 
V { 
"_pi"=01010001100001001111100000111010; 
} 
V { 
"_po"=HLLLLLHLLLLLHLLLLLHLLLLLHLLLLLHLLLLL; 
} 
C { 
"_po"=r32 X; 
} 
V { 
"clock"=P; 
} 
V { 
"_pi"=00000100110011000110010011100001; 
} 
V { 
"_po"=LLLLLHLLLLLHLLLLLHLLLLLHLLLLLHLLLLL; 
} 
C { 
"_po"=r32 X; 
} 
V { 
"clock"=P; 
} 
V { 
"_pi"=0100001001110011110010110101101; 
} 
V { 
"_po"=HLLLLLHLLLLLHLLLLLHLLLLLHLLLLLHLLLLL; 
} 
C { 
"_po"=r32 X; 
} 
V { 
"clock"=P; 
} 
V { 
"_pi"=00100101111101100101110011001101; 
} 
V { 
"_po"=LHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH; 
} 
C { 
"_po"=r32 X; 
}
V { "clock"=P; }
V { "_pi"=010010100010011110001000010000110; }
V { "_po"=HLLHLHLLLHLLHHHLLLHLLLLLLLLLLLLLLL; }
C { "_po"=r32 X; }
V { "clock"=P; }
V { "_pi"=001001010000110010010101010011110; }
"end 15 measurePO": V {
"_po"=LHLLHLHLLLHLLLLHLLHHHHLHLLHLLH; }
}