Automatic Verification of Microprocessor designs using Random Simulation

Anantha Ganesh Karikar Kamath
Abstract

Automatic Verification of Microprocessor designs using Random Simulation

Anantha Ganesh Karikar Kamath

Verification of microprocessor cores has always been a major challenge and a crucial phase in the development of a microprocessor. Increasing chip complexities and decreasing time-to-market windows has led to steady increase in the verification costs. Automatic verification of microprocessor designs based on random simulation helps to quickly capture inconceivable corner cases that would not have been found by manual testing.

This thesis work focuses on the design and implementation of a Co-Simulation testbench platform together with a framework for generating random assembly programs for the functional verification of the OpenRISC Processor, OR1200. A Random Program Generator based on configurable instruction weights is developed to generate large test volumes. These random test programs are used to verify the functional correctness of the Register Transfer Logic model of a processor against a behavioral Instruction Set C Simulator. The simulation results show the effectiveness of this approach. Histograms are used to graphically illustrate the instruction and register coverage statistics.
ACKNOWLEDGMENTS

This is a Master Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Embedded Systems to the Department of Information Technology, Uppsala University, Uppsala, Sweden.

My deepest gratitude to my supervisor Dr. Philipp Rümmer, Programme Director for Master’s programme in Embedded System at Uppsala University, for his guidance, encouragement and patience he demonstrated during my thesis work.

Further I would like to thank my reviewer Dr. Leif Gustafsson, Senior Fellow at the department of Physics and Astronomy, Uppsala University for undertaking my Master’s thesis.

I would also like to thank my thesis coordinator, Prof. Olle Eriksson, for organizing my thesis presentation and for helping me to register my thesis work.

I am also deeply indebted to Mr. Marcus Erlandsson, CTO at ORSoC AB Stockholm, for offering me this thesis opportunity. I would also like to thank other employees at ORSoC AB and especially Mr. Krister Karlsson and Mr. Yann Vernier for their guidance.

Sincere thanks to my friends and family who gave me courage and support throughout this thesis work.
CONTENTS

Acknowledgments i

Contents iii

List of Figures v

Listings vii

1 Introduction 1

2 Background 3
   2.1 Introduction 3
   2.2 Formal and Simulation based Verification Methodolo-
   gies 3
   2.3 Directed and Random verification techniques 5

3 OpenRISC Processor and Tools 7
   3.1 Introduction 7
   3.2 OpenRISC 1200 Soft Processor 7
   3.3 Or1ksim Instruction Set Simulator 9
   3.4 ORPSoC and Icarus RTL Simulator 9
   3.5 Cycle Accurate C++ Model 10
   3.6 GTKWave Waveform Viewer 10

4 Co-Simulation Testbench 11
   4.1 Introduction 11
   4.2 Random Program Generator 11
   4.3 Testbench Controller 13

5 Implementation Details 19
   5.1 Introduction 19
   5.2 Random Program Generator 20
   5.3 Testbench Controller 26
   5.4 Sniffer Modules 30
   5.5 Checker Module 31
   5.6 Coverage Monitor 32

6 Results and Discussion 35
6.1 Introduction 35
6.2 Discrepancies between OR1200 and Or1ksim 35
6.3 The OpenRISC 1200 verification coverage results 39

7 Conclusion and Future Work 45
7.1 Conclusion 45
7.2 Future Work 45

Bibliography 47

Appendices

A Source Code 49

B Example test programs 51
LIST OF FIGURES

1.1 Overview of the Co-Simulation testbench 2
3.1 Overview of the OpenRISC 1200 IP Core 8
4.1 Main components of the Co-Simulation Testbench 11
4.2 Design of Testbench Controller 14
4.3 Register queues 16
5.2 Steps to generate random test program 23
5.3 Conceptual representation of code memory 23
5.4 Markov model for instruction class selection 24
5.5 Random distribution for unequal weights 25
5.6 Random distribution containing equal weights 25
5.7 Constraints in branching and function calls 25
5.8 Start-up sequence of Testbench Controller 27
5.9 Termination due to watchdog timer 28
5.10 Termination due to resource constraints or simulation errors 29
5.11 Normal termination of TBC 30
6.1 Testbench waveform for l.rori instruction 37
6.2 Testbench waveform for illegal load/store instruction 38
6.3 Instruction coverage for a test suite 41
6.4 Instruction histogram for a test program 41
6.5 Instruction histogram for a test program 41
6.6 Register coverage for a test suite 42
6.7 Register coverage for store instruction class 42
6.8 Register coverage for branch and call instructions 43
LISTINGS

Example program illustrating resource collision . . . . . 15
Configurations for the Random Program Generator . . . 20
Configuration file to specify instruction format . . . . . 21
Configuration file to specify instruction weights . . . . . 22
Algorithm to implement weighted random distribution . . . 24
Configurations for the Testbench Controller . . . . . . . 26
Data structure implementing the Register queue . . . . . 32
Code snippet illustrating use of l.mfspr instruction . . . 35
Simulation results for l.mtspr instruction . . . . . . . . . 36
Code snippet for l.rori instruction . . . . . . . . . . . . . 36
Simulation results for l.rori instruction . . . . . . . . . . 36
Simulation results for l.ror instruction . . . . . . . . . . . 37
Simulation results for l.ld or l.sd instruction . . . . . . . 38
Source code containing l.sub instruction . . . . . . . . . 39
Simulation results for l.sub instruction . . . . . . . . . . 39
Run time instruction traces for test program containing
l.sub instruction . . . . . . . . . . . . . . . . . . . . 40
Code snippet for disabling caches . . . . . . . . . . . . . 40
test1.s . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
test2.s . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
run1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
run2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
INTRODUCTION

System on Chips (SOC’s) of today are very large systems, typically consisting of processor clusters (master CPUs, DSPs, hardware accelerators etc), multiple-level data memory and program memory systems (L1, L2, L3 memories and caches), on-chip interconnection systems (buses, crossbars, bridges), DMA\(^1\) controllers, IO’s and various system peripherals. The design of such a SOC is challenging. Typically, it is not the actual implementation, but instead the verification, where most of the project time is spent. It is estimated that roughly 70 percent of the total effort for hardware designs is invested in functional verification (Bin, 2012). As a consequence of this, several different verification methods have been developed over time, such as simulation, hardware emulation, formal verification, and assertion-based verification.

A microprocessor is a central SOC component, in the shape of a control CPU, a DSP, an ASIP\(^2\), or a co-processor. Its design may be verified by executing (lots of) test programs in a simulation testbench. Typically the costs incurred for the verification of a processor design is very high. For example, the verification of a RISC\(^3\) System/6000 processor involved approximately fifteen billion simulation cycles, large engineering teams and hundreds of computers during a year (Aharon et al., 1995). Also substantial amount of money is spent during design re-spins. For example, the Pentium FDIV bug found in the Intel P5 Pentium floating point unit (FPU) and the bug in Intel Sandy Bridge accounted for loss of millions of dollars (Lagoon, 2012).

This thesis work outlines the verification plan for the functional verification of an Open RISC processor core, OR1200 from Opencores\(^4\). The verification methodology suggested here is to compare the actual resulting behavior to the expected behavior, for a certain test program and stimuli input data. The goal of the thesis is to design and implement a Co-Simulation testbench platform together with a framework for generating random assembly programs for the functional verification of an OpenRISC processor. A Random Program Generator based on configurable instruction weights is developed to generate large test volumes. The co-simulation testbench platform interfaces the existing two different models of the processor, i.e. the RTL-model of OR1200

---

1. DMA, Direct Memory Access
2. CPU, Central Processing Unit;
   DSP, Digital Signal Processor;
   ASIP, Application Specific Instruction set Processor
3. RISC, stands for Reduced Instruction Set Computing.
4. OpenCores is the world’s largest site/community for development of hardware IP cores as open source. One can find the source code for different digital hardware projects at, http://OpenCores.org
The RTL-model is simulated using Icarus Verilog which is an open-source Verilog simulation and synthesis tool and the ISS, Or1ksim C simulator. The testbench controller, a module of the co-simulation testbench, executes the test program simultaneously on the two processor models and compares the state of the processor on a cycle-by-cycle basis. The verification results are quantified by implementing a coverage monitor, which measures the verification coverage and builds coverage statistics. Figure 1.1 provides an overview of the co-simulation testbench.

A similar study was performed in (Ahmed, 2010) which applies an OVM framework to provide extensive comparison between the two models of the processor. Although (Ahmed, 2010) provides a good verification platform for the functional verification of OR1200 core, it would be possible to achieve better results using random verification techniques. In this thesis, we investigate the verification results obtained by tests program generated from a random program generator. A random program generator generates assembly programs based on the concept of constrained randomness. This means a combination of randomness and constrained programming rules. The proposed method is advantageous as the random generator can in a very short time create a large number of test programs of arbitrary (however reasonable) size generating enormous test volumes. Another advantage of random testing is that it can over time, produce odd and strange programmes with unusual instruction sequences, that few human programmers will ever think of. Very often these tests are the once that will hit corner and extreme cases, and they can reveal bugs that would not have been found by manual testing.

The next chapter discusses different hardware verification techniques covered during the literature review. The subsequent chapters focus on the design and implementation of the random program generator and the testbench controller. The report is concluded by analyzing the verification and coverage results.
BACKGROUND

2.1 introduction

Design verification is a process to determine if the implemented design accurately represents the intended specification. New process technologies, higher chip complexity and increasing clock frequency cumulate the cost for verifying hardware designs. This has motivated researchers to investigate new approaches to improve and fasten the hardware verification process. This chapter details through the different verification methodologies and approaches used for verifying SOC designs. Based on the literature review, the SOC hardware verification can be broadly divided into two types: pre and post silicon. The thesis work is delimited to the functional verification of SOC’s applied for Pre-silicon designs. Pre-silicon verification employs two major families of solutions: Simulation based Verification and Formal Verification.

2.2 formal and simulation based verification methodologies

Formal verification is a technique that involves generating mathematically proofs for the behavior of the design. It employs assertions based methodologies for the functional verification of the properties of the system. It works aiming to prove or disprove a property of a hardware implementation using logical and mathematical models. Equivalence checking and Property checking form the two major categories of formal verification techniques (Ahmed, 2010). Although this methodology can possibly verify all possible behaviors of the model for all time instances and for all input variations, their extensive computational requirement limits its usability for verifying small to moderate designs.

Simulation based techniques on the contrary are not limited to such constraints. It can be seen as verification through input space sampling. In this method the functional correctness of a hardware model can be checked against higher level references or instruction set simulators. These techniques typically consists of a testbench controller and a DUT\textsuperscript{1}. The testbench controller feeds

\textsuperscript{1}. DUT is an abbreviation for Device Under Test.
input test vectors to the DUT. The next state of the DUT is determined based on its present state and the input from the testbench controller. The computed states of DUT are then verified against expected states of the design. Unless all points are sampled, there exists a possibility that an error escapes the verification process. So in simulation based verification techniques it is necessary to implement coverage monitors to generate coverage statistics. This helps to keep track of the covered scenarios thereby increasing the confidence on device correctness.

**literature review**

(Aharon et al., 1995) discusses the design of a model based test generator comprising a formal model of processor architecture tested against a behavioral simulator. In order to generate better quality tests, it uses heuristic database capturing the testing expertise of test engineers. The verification platform succeeded in finding bugs in different derivatives of a PowerPC architecture. (Ahmed, 2010) in his MSc dissertation discusses a Co-Simulation technique to verify RTL model of OpenRISC OR1200 core against a Golden Reference Model implemented in C. It applies an OVM framework to provide extensive comparison between the two models of the processor. (Semeria et al., 2002) also discusses a Co-Simulation technique to verify a processor RTL model using behavioral Golden Reference Model in HDL. An interesting Post-Silicon verification technique is proposed in (Wagner and Bertacco, 2008) which can be extended to Pre-Silicon methods. The idea is to identify an inverse instruction or a procedure for every instruction in the processor’s ISA\(^2\). An initial random state of the processor is assumed and known in prior to generating a random program. The different instructions and their inverse operations make the system to reach a final state equal to its initial state. A bug is discovered if after the execution of the program the system does not reach the initial stage. Since the same system is used to verify its correctness it eliminates the need for a Golden Reference Model. But this approach is limited to test only a subset of the processor’s ISA, since not all instructions have an inverse.

Simulation based verification techniques are either structural or functional in nature. Structural techniques are developed to target the functional components like ALU’s\(^3\), shifter, multiplier, interconnect components that guide the data flow and storage components like the register file. (Corno et al., 2003), (Hudec, 2011), (Wagner and Bertacco, 2008) propose different structural simulation methods. Although structural techniques achieve high fault coverage for the functional components, they yield low fault coverage for pipelined logic and system co-processors (Psarakis et al., 2006). Pipelined logic of a processor typically consists of

---

2. ISA is an acronym for Instruction Set Architecture

3. ALU is an acronym for Arithmetic Logic Unit
pipeline registers, hazard detection and resolution logic including bypass hardware for forwarding and pipeline interlocks. (Psarakis et al., 2006) proposes a methodology which enhances the existing test programs generated for structural verification techniques to achieve high fault coverage for the pipeline logic.

Along with the different methodologies used for hardware verification it is also important to consider the techniques used to generate the test inputs in the verification process. There are two main approaches to achieve this: Directed verification and Random verification. In the following section we will discuss and compare these techniques.

2.3 directed and random verification techniques

Traditionally, simulation-based verification uses directed verification approach. Directed verification techniques\(^4\) are considered 'white-box' techniques in which deterministic test sequences are generated to test the functionality of a design. Considering an example of verifying the functionality of a microprocessor, directed verification can consist of test programs written to test a certain or a small group of related instructions. A self-checking test program consists of four sections; preparation of stimuli input data, preparation of expected result, executing the test program to generate the actual result and comparison of actual and expected result. The advantage of directed testing is that it is a fairly quick and easy method for the very first verification of an instruction. It gives early simulation feedback, so that the most embarrassing bugs can be fixed.

The disadvantage of directed testing is that it is a very slow and tedious method for exhaustive verification. For a common instruction such as add, the number of combinations of input arguments (registers and immediate) can be large. The result can hit several various conditions (overflow, negative/positive, zero, carry). Also, the execution of the instruction may depend on additional control information stored in configuration registers.

Random verification techniques sometimes also known as \textit{Gorilla testing}\(^5\) is a technique of generating random test programs with the purpose of testing any instruction, or any sequence of instructions. It is generally considered as a 'black-box' testing technique as one does not need to rely on the internal details of the model. Considering the previous example of verifying the functional design of the microprocessor, the test programs are generated by a random program generator consisting of pseudo-randomly selected instructions. These random test programs are likely to generate test cases that can verify hidden scenarios unlikely to be covered by manual directed verification techniques. Automatic test

\(4\). Directed verification is sometimes also referred as \textit{happy testing} or \textit{smoke testing}.

\(5\). It is a slang used for torture testing a product to see what level of abuse will cause it to break or fail.
benches are typically set up to generate and verify large volumes of random programs. This can result in massive simulations which helps to increase the confidence on device correctness.

Random test generation can be again classified as either static or dynamic. In static random test generation, the system is initially set to a random state to generate random test programs. These random test programs are used by the test bench controllers or model simulators. Dynamically biased random test generation relies on a simulation feedback to generate random programs on-the-fly yielding high coverage statistics.

literature review

Several dynamic random test generation methods have been proposed in the literature. (Corno et al., 2003), (Hudec, 2011) propose automatic test generation technique for functional verification of processor cores using evolutionary approach that significantly improves the fault coverage. (Corno et al., 2003) exploits MicroGP, an approach which utilizes a Directed Acyclic Graphs (DAG) for representing the sequential flow a program. It associates the generated test programs with fitness values, which are used to probabilistically select the parents for generating new test programs. The fitness value of a test program measures its efficacy and its potential to excite new faults. (Hudec, 2011) discusses a similar evolutionary technique using genetic algorithms to improve the fault coverage from the generated test programs.

Although dynamic techniques can improve the quality of generated test programs, their throughput is generally less than the static generators. Also significant design and implementation effort is needed to develop random generators with dynamic techniques. Modifying the existing generators is also difficult as they are often custom made for specific architectures. In this thesis a random generator using static techniques is implemented together with a coverage monitor to evaluate its performance. The design of the random program generator is separated from the testbench controller which in future is helpful to upgrade or replace with better design implementations.
OPENRISC PROCESSOR AND TOOLS

3.1 introduction

This chapter briefly discusses the OpenRISC OR1200 processor architecture, the functionality of which we aim to verify in this thesis. Latter sections focus on the available simulators and the tools used in this thesis.

3.2 openrisc 1200 soft processor

The OpenRISC project was started in 1999 by a group of Slovenian university students (Baxter, 2011). The project aimed to define and implement free and open source families of RISC microprocessor architectures. As a result of this the OpenRISC 1000 architecture, the first set of specification for a family of 32- and 64-bit RISC/DSP processor was developed. Its open and modular architecture targets medium and high performance application and allows a spectrum of chip and system implementations. The architecture defines the instruction set, the addressing modes, exception handling mechanisms, cache model and cache coherency, memory model, memory management and also the software ABI² (Lampret, 2000). The OpenRISC 1000 architecture is licensed under GNU GPL and can be accessed freely at OpenCores.org. A good overview of open source hardware development can be found in (Baxter, 2011). The author reviews the OpenRISC architecture and its implementations.

The OpenRISC OR1200 is an open source synthesizable CPU implemented and maintained by the developers at OpenCores.org³. The RTL implementation follows a 32-bit scalar RISC with Harvard microarchitecture, 5 stage single issue integer pipeline, virtual memory support and basic DSP capabilities (Damjan Lampret, 2001). The implementation also specifies a register set, cache operation, power management unit, programmable interrupt controller (PIC), debug unit and tick timer. Other peripheral systems can be added using the processor’s implementation of a standardized 32-bit Wishbone bus interface. An overview of the OpenRISC

---

1. RISC is an abbreviation for Reduced Instruction Set Computing.
2. ABI is an abbreviation for Application Binary Interface.
3. OpenCores is currently owned and maintained by ORSoC AB which is a fabless ASIC design and manufacturing services company headquartered in Stockholm, Sweden.
OR1200 core is illustrated in Fig. 3.1.

The OR1200 is intended to be used for embedded, portable media, networking, home entertainment and automotive applications. The latest application includes a product developed at ORSoC AB, to ease connection between different units in a spacecraft. The RTL implementation of OR1200 core is written in Verilog hardware description language and is released under the GNU Lesser General Public License (LGPL). The source code, test software and related scripts can be downloaded from the OpenRISC project subversion (svn) repository.

The verification platform developed in this thesis is implemented using Ubuntu desktop operating system (version 11.04). In order to compile the test programs for OpenRISC OR1200 architecture it is necessary to setup the GNU toolchain\(^4\). The test programs are compiled to ELF\(^5\) file format which can be executed on both the ISS\(^6\) and RTL simulators. It is also possible to use VirtualBox Ubuntu image\(^7\) which makes it easy to get started with the OpenRISC processor system. The test programs can be executed on bare metal, i.e. without using any operating system support, or as a linux application. The latest Linux kernels are ported to the OpenRISC architecture which currently supports statically

---

4. It contains the packages consisting of binutils, GCC and GDB. The instructions to set up the toolchain can be found at [http://OpenCores.org](http://OpenCores.org).

5. ELF is a standard file format. It is an abbreviation for Executable and Linkable Format.

6. ISS is an acronym for Instruction Set Simulator.

7. The current VirtualBox image contains Ubuntu 11.10 Linux distribution with the OpenRISC tool chain installed. It contains all the tools, packages and also the simulators eliminating installation problems. It can be downloaded from [http://OpenCores.org](http://OpenCores.org).
compiled binaries. This thesis work is carried out by executing the test programs on ‘bare metal’.

### 3.3 or1ksim instruction set simulator

Or1ksim is an Instruction Set Simulator (ISS) for the OpenRISC 1000 architecture and serves as a Golden Reference Model for the functional verification of the core. It is an open source simulator and is implemented using C programming language. It can be freely downloaded from the OpenCores website. Jeremy Bennett from Embecosm has lead the contributions to Or1ksim and is currently responsible for the maintenance of the tool. Or1ksim models all major OpenCores peripherals and system controller cores. It also enables easy addition of new peripheral models. The tool uses a configuration file to configure different environments like memory and peripheral device configurations, and processor models. It provides remote debugging facilities via TCP/IP with the GNU Debugger (GDB) using either the GDB Remote Serial Protocol (RSP) or through a modeled JTAG interface. These features makes Or1ksim an effective OpenRISC emulator allowing early code analysis and to evaluate system performance.

Or1ksim can be used as a standalone simulator or as a library. The interactive command-line interface can be used to single step through the code, insert breakpoints and to view register and memory contents at run time. The user manual (Bennett, 2008) describes the installation and configuration of the simulator. It also details the commands needed to work in interactive mode. The Or1ksim source code is modified and the capabilities of interactive command-line interface are extended to interface the ISS with the testbench controller. More implementation details are discussed in section 5.4.

### 3.4 orpsoc and icarus rtl simulator

Icarus Verilog is a Verilog simulation and synthesis tool written by Stephen Williams. It can compile source code written in Verilog (IEEE-1364). The compiler can generate an intermediate format called vvp assembly for batch simulation which can be executed by a vvp command. For synthesis, the compiler generates net lists in the desired format.

The RTL-model of the OpenRISC OR1200 can be simulated using Icarus Verilog. OpenCores.org has created the ORPSoC project which provides a development platform for OpenRISC processor and OpenRISC based SOC’s targeted at specific hardware. A complete documentation of ORPSoC project can be found at (Bax-
The project organization separates the implementation for the reference design from the board specific designs. The implementation includes RTL description of the processor core and related peripherals and the software consisting of CPU drivers and related device drivers. The platform also provides custom tools, intended to be run on the host, for manipulation of binary software images. As the thesis work is limited to verify the functionality of the OpenRISC core the reference design is used for simulation. The section ReferenceDesign in (Baxter, 2010) provides a good overview of the project, illustrating structure of the reference design and details the steps needed to simulate it using the Icarus Verilog Simulator.

3.5 cycle accurate c++ model

The simulation rate of Icarus Verilog, the event-driven simulator, discussed in section 3.4 is far from the simulation rate of the ISS. The simulation speed can be improved by using a C++ cycle accurate model generated using the Verilator tool. This model generated from the OpenRISC RTL-model is capable of running simulations at hundreds of kilohertz range, as opposed to the sub-ten kilohertz range simulation experienced with Icarus Verilog (Baxter, 2011). In spite of this performance advantage the testbench controller is interfaced with the Icarus Verilog Simulator to avoid unforeseen bugs caused during code translation. Thus assuming that the behavior of the Golden Reference Model (ISS) is always correct, one can be certain that the bugs discovered from the verification process exists in the RTL-model.

3.6 gtkwave waveform viewer

It is necessary to log the state of the processor after execution of every single instruction. This can be accomplished by using a standard dump file format of verilog called VCD. The ORPSoC platform discussed in section 3.4 provides configurations to enable these logs. Since there is no built in waveform viewer in Icarus Verilog, we use the GTKWave, an open source tool to examine the waveform. It makes use of the VCD files to generate the waveforms.
4.1 introduction

This chapter discusses the design for the Testbench Controller. A Co-Simulation testbench is implemented to interface two different exiting models of the processor, i.e. the RTL-model and Or1ksim, the ISS. ISS is used as a Golden Reference Model and is assumed to be produce expected results. A Random Program Generator is implemented to generate test programs. Each of the test programs is executed simultaneously by both processor models and their states are compared after execution of each instruction. A Checker module is also implemented as part of the Testbench Controller to log the differences in the states of the two models. The Testbench Controller implementation also includes a coverage monitor module to measure the verification coverage and generate coverage statistics. An overview of the components is presented in 4.1.

In the following sections the design issues involved with each of the components shown in figure 4.1 are discussed. The implementation specific details are covered in the next chapter.

4.2 random program generator

The Random Program Generator is a module to generate test programs\(^1\) by randomly choosing instructions from the OpenRISC ISA\(^2\). It’s design is based on constrained randomness, which involves certain programming rules to generate compilable code. Some of the rules can be,

---

1. The test programs are generated in assembly language and is converted to binary using an assembler.
2. ISA is an abbreviation for Instruction Set Architecture.
• The generator must produce existing instructions with valid (legal) arguments.

• The target address of a branch instruction must be a valid program location.

• A number of branches or subroutine calls must not create an infinite loop.

• After a call to a subroutine there must be a return from subroutine.

• A software loop must have a branch back to the loop start.

• No attempts to pop elements from an empty stack or push elements on a full stack.

The Random Program Generator is implemented as a standalone component and is not part of the testbench controller. This approach allows to upgrade or replace the generator with better design implementations. The generator can also be configured to iteratively produce large volumes of test programs which can be executed in the Co-Simulation testbench in a batch. The initial seed value used by the generator decides the instructions chosen for the test program. So it is important to log the initial seed values to reproduce interesting test programs.

The instruction types, their argument values, their location are largely dependent on the ISA. An instruction configuration file is created consisting of a table with an entry for each instruction to hold the name and format of the instruction. The Random Program Generator uses this configuration file to generate meaningful instructions. The occurrence of an instruction in the test program is controlled by instruction weighting. In this mechanism each instruction is associated with a weight, that indicates the probability of being picked by the generator. A high weight means a high probability and therefore many instances of the instruction in a program. A low weight means low probability and therefore results in few instances of the instruction. A weight of zero forces no instances of the instruction in the test program. Together all weights form a probability distribution function for the instruction set. If a test program that contains only a few selected types of instructions to be generated, the corresponding weights are set to a number larger than zero and the weights of all other instructions are set to zero. The instruction weights are supplied to the Random Program Generator using another configuration file.
literature review

(Wagner et al., 2007) describes a Markov-model-driven random test generator. It abstracts the individual instructions to instruction classes which constitute the vertices of the Markov model. This method can be used to regulate the generation of certain classes of instructions which otherwise requires masking of large number of individual instructions. A template for the design verification of microprocessors and system-on-chips, SOC’s is presented in (Uday Bhaskar et al., 2005). The platform designed in our thesis work closely resembles their proposed template except for the Debugger module, which is not included in our architecture. If inconsistencies are observed between the models, the test programs can be run individually on either models and the existing debugging facilities can be used.

4.3 testbench controller

The Testbench Controller is responsible for the overall simulation progress and control. A detailed design of the Testbench Controller is shown in figure 4.2. The Testbench Controller starts by loading the same test program into the simulators. A software reset causes the processor to load predetermined values into the registers and the data memory. The Testbench Controller controls the simulation progress of the Or1ksim ISS simulator through step commands. The Icarus RTL simulation advances by using the internal clock generator. Sniffer modules are included in the controller to interface the Or1ksim, ISS simulator and the Icarus RTL simulator. After execution of each instruction the sniffer modules collect changes in the processor status and register values. The sniffer module also keep track of the data memory location being addressed during a store operation. The information from the sniffer is updated to the checker module which compares and logs the differences in the processor states. If the number of differences exceed some configured maximum limit, the checker signals the Testbench Controller to end the simulation. In normal scenarios the simulation is completed after execution of the last instruction in the test program. If the test programs are run in a batch, the simulation process is repeated by loading the next program. The Testbench Controller terminates after executing the last test program. The design also includes a coverage monitor which is run offline to build instruction coverage statistics.
4.3.1 Checker Module

As discussed in section 4.3 the Checker module is responsible for comparing the states of the ISS to the RTL-model in order to determine if the two models of the microprocessor have the same behaviour. Since the two processor models run in different levels of abstraction they might generate different results due to pipeline effects. Typically only the RTL-model implements the processor pipeline while the ISS does not. Also as the two models run with a different perception of time they are not perfectly synchronized to each other. These differences cause problems, namely skew and collision, which has to be handled by the checker module.

---

4. Or1k sim ISS simulator is instruction accurate and the Icarus RTL Simulator is cycle accurate.
**Skew Problem**

Even though the two models run the same program in parallel the result of the same instruction may be produced at different points in time. Also during certain periods of the simulation, one model may be slightly ahead of the other model, and therefore produce few more results than the other model. This typically happens after branch instruction, because pipeline refill latencies occur in the RTL-model but not in the ISS. So, the results from the two models are skewed in simulation time. (John L. Hennessy, 2006) lists different reasons for pipeline hazards that can stall the processor.

**Collisions in pipeline**

Besides the skew problem, the RTL-model might generate different results due to collisions in the pipeline. If the RTL-model implements an exposed pipeline then all the instructions are executed in the way they are injected into the pipeline. Whereas in the case of an interlocked pipeline the data dependencies and resource collisions are resolved at run time. A simple example program 4.1 illustrates resource collisions in a processor implementing interlocked pipeline.

```
mac r0, r1, r2 /* Mult-and-acc: 
    r0 = r0 + r1*r2, 2 exe cycles*/
add r0, r11, r12 /* Addition: 
    r0 = r11 + r12, 1 exe cycles*/
```

Source Code Listing 4.1: Example program illustrating resource collision

The mac-instruction (having 2 execute cycles) and the subsequent add instruction (having 1 execute cycle) both attempt to update register r0 in the same cycle. The ISS will typically store two different values for register r0, and therefore the ISS Sniffer will receive two values for register r0. In the RTL-model the two results for register r0 collide. Now, the pipeline may decide to keep the result from the add-instruction (because add happens after mac) and throw away the result from the mac-instruction. In this case the RTL sniffer will receive only one value for register r0 (instead of two). The sequence of instructions causing resource collisions are limited and should be filtered by the checker module.

**Using Register Queues**

The skew problem discussed in 4.3.1 can be solved by using Register Queues. Usage of register queues enables timing, data-flow and ordering control. The idea is to maintain two queues for each of the processor registers. The ISS sniffer pushes register updates
into a queue (ISS register queue) obtained from the Or1ksim ISS simulator and the RTL sniffer uses the second queue (RTL register queue) to push the register values obtained from the Icarus RTL simulator. In order to reduce the complexity and the number of register entries to be checked, only the changes in the register values are pushed. The depth of the queue is determined by the length of the processor pipeline but a typical value of 10 to 20 entries can be used. When the queues reach their capacity, the checker module reads and compares the results of simulation. After comparison identical entries are popped out from the top of the queues. At the end of the simulation the queue pointers are compared and the residual values in the queues are logged for analysis. A similar approach is used for the comparison of data memory locations. Figure 4.3 is an example of a register queue.

![Message Queue Implementation](image)

Figure 4.3: A typical structure of Register queues

### 4.3.2 Coverage Monitor

The Coverage Monitor is responsible for measuring the verification coverage and building coverage statistics. The Coverage Monitor basically contains a number of counters that detect and count the occurrence of certain events. Some examples of verification coverage can be,
• Instruction coverage to measure the occurrences of each instruction.

• Sequence coverage to measure the occurrence of certain sequence of instructions.

• Register coverage to measure the instances it has been used with a particular instruction.

• Branch coverage to measure conditional branches that are taken.

The number of times an instruction occurs in a program code can be different than the number of times the instruction has been executed. The program flow is non-deterministic because of looping, conditional branching, interrupts and other data dependent factors. So the coverage monitor has to use the execution trace instead of the static code to generate coverage statistics.
IMPLEMENTATION DETAILS

5.1 introduction

This chapter details the implementation issues for the Random Program Generator and the Testbench Controller. Later sections illustrate the modifications made to the processor models to interface it to the Testbench Controller.

The Testbench Controller and the Random Program Generator are implemented using C programming language. The source code can be checked out from subversion (SVN) repository

http://svn.orsoc.se/svn/random-generator

Figure 5.1 outlines the organization of the project. For the sake of clarity only the .c files under the trunk\random-testing are listed.

The trunk\random-testing directory contains the implementation for the Random Program Generator and the Testbench Controller. As discussed in section 4.2 the implementation of the Random Program Generator is separated from the Testbench Controller. The source code for the Random Program Generator can be found under random-testing\randomgen, while the files directly under trunk\random-testing constitute for the Testbench Controller. The RTL model of the OR1200 core is saved in trunk\orpsocv2 directory while the Or1ksim ISS model is saved under trunk\or1ksim. Information on setting up the simulators can be found in Sections 3.3 and 3.4. The make all command run from the trunk\random-testing creates test programs under the random-testing\randomgen directory. The number of test programs generated is defined by the variables in the makefile. Then the make compiles the generated test programs. The output of the assembler, i.e. the test programs in elf binary format are stored under the trunk\output directory. The Testbench Controller is run using ./tbc command. Simulation logs are generated under the trunk\log directory. The logs contain the run-time code executed by the ISS and the differences in the states of the simulator. The simulation logs and the elf output files are cleared when make clean or make all is executed.

Figure 5.1: Project software hierarchy
The source code can be checked out from SVN repository1.
2. Doxygen is an open source documentation system supporting various programming languages. More information can be found at, http://www.stack.nl/~dimitri/doxygen/

The implementation work is documented using Doxygen\(^2\). The html and pdf versions of the source code documentation can be found under trunk\RANDOM-TESTING\docs directory. The Testbench controller and Random Program Generator support different user configurations. The following sections in this chapter discuss these configurations and details the implementation aspects for each of the modules.

### 5.2 random program generator

The design of the Random Program Generator is discussed in section 4.2. The Random Program Generator can be run as a standalone tool to generate large volumes of test programs. Different user configurations are supported which can be listed using the command,

```
./rpg -h
```

Listing 5.1 illustrates these configurations.

-\(l\) <file> : option to specify the instruction log file, this is a optional argument with no default value
-\(c\) <file> : option to specify the configuration instruction file, default file = configinst.s
-\(w\) <file> : option to specify the configuration weight file, default file = configweight.s
-\(i\) <No_of_Instructions> : option to specify number of instructions to be generated in the output program, default value = 100
-\(o\) <output file> : option to specify the name of the output file to be generated, default name = test.s
-\(f\) <Max Procedures> : option to specify the maximum limit on number of proedures, default value = 5
-\(d\) <No of Bytes> : option to specify the maximum number of bytes in data segment, default value = 100
-\(r\) <RANDOM_SEED> : option to specify the random seed for the generation of random markov state
-\(p\) <pattern> : option to specify the pattern for initializing memrory and registers, possible options are RANDOM, SERIAL, ZERO, default value = SERIAL
-\(h\) : lists the available help options

Source Code Listing 5.1: Configurations for the Random Program Generator

A test program named test.s, with 500 instructions and containing a maximum of 10 functions can be generated using the command,

```
./rpg -o test.s -i 500 -f 10
```
The test program can be compiled to ELF file format using the assembler ported for OpenRISC OR1200 architecture.

\[ \text{or32-elf-gcc -o test.o test.s} \]

As discussed in section 4.2 the generator needs two separate configuration files to generate random but meaningful instructions. An example of a configuration file specifying the instruction format is shown in the listing 5.2. Listing 5.3 is an example configuration file to specify instruction weights.

```
instruction l.add
  type = IT_ARITH
  pattern = OPCODE_R_R_R
end

instruction l.addc
  type = IT_ARITH
  pattern = OPCODE_R_R_R
end

instruction l.sub
  type = IT_ARITH
  pattern = OPCODE_R_R_R
end

instruction l.mul
  type = IT_ARITH
  pattern = OPCODE_R_R_R
end

instruction l.div
  type = IT_ARITH
  pattern = OPCODE_R_R_R
end
```

Source Code Listing 5.2: Configuration file to specify instruction format

The execution of the Random Program Generator starts by parsing the user supplied inputs and reading the configuration files. Figure 5.2 illustrates the sequence of tasks involved in generating a random test program. It begins by initializing the data memory and the registers to a predetermined state. This step is important as the simulators starts with a software reset initializing the register and memory values to zeros. Since most of the arithmetic and logic operations operating with zero values do not change the processor state, they would fail to produce any interesting results. The current implementation allows to initialize the registers and data memory with a single constant value, or with serial values, zeros and with random values. An initial seed value is chosen in order
Source Code Listing 5.3: Configuration file to specify instruction weights

to randomize the selection of instructions and register usage. The seed value is either supplied as an user input or is selected as a random number. It’s value is logged into a file which enables one to regenerate interesting test programs.

The test program is made up of a main function and other randomly generated functions. The functions are made up of arbitrary length and placed at random locations to cover maximum code memory location. A conceptual representation is illustrated in Figure 5.3. The implementation prevents recursion and avoids infinite loops caused by random function calls. It is also taken care that the target address of a branch instruction is always a valid program location.

The process of picking random instructions is designed as a two step process. In the first step the Random Program Generator categorizes the instructions into instruction classes and uses a markov model to pick an instruction class at random. Figure 5.4 shows an example of a Markov model. The instruction classes can comprise of arithmetic, logical, branch, move instructions etc, which form the vertices of the model. Once an instruction class is selected at random, the next step is to randomly pick an instruction under this
Figure 5.2: Flow chart illustrating the steps to generate a random test program

1. Begin
2. Read input configuration (No of Instns, Size of DM, Initialization pattern etc)
3. Read instruction configuration file and instruction weights
4. Initialize Registers and Data Memory
5. Choose seed value ($S$) and random functions to generate ($F$)
   - set $f = 0$
6. Select a Random Instruction Class and Pick Random Instruction from selected class
7. $f++$
8. Is $f = F$
   - No
   - Allocate stack, Initialize Registers, Enable interrupts, supervision mode, Initialize Data ptr
   - Yes
9. Generate Main function
10. End

Figure 5.3: Conceptual representation of code memory
class based on the weights specified in the configuration file. After generating an instruction, the next instruction class is probabilistically selected using the markov model based on the weights defined over the state transitions. The registers and immediate values used as arguments to the instructions are also randomly selected. The Random Program Generator keeps track of the generated instructions and continues till the user specified limit is reached.

Source Code Listing 5.4: Algorithm to implement weighted random distribution

Listing 5.4 illustrates the algorithm used to randomly pick the instructions based on preconfigured weights. The algorithm randomly picks the objects defined in the objects array based on the weights defined in the weights array. The algorithm runs for no_of_iterations times which is supplied as an input. The function RandomGenerate uses a discrete uniform distribution function to generate random numbers within a range of (1 to totweight), where totweight is the sum of weights defined in the weights array. Before
initiating the algorithm the weights are sorted in decreasing order by calling the `Reorder_Weights` function. The algorithm when run for sufficiently long iterations implements a weighted random distribution. Figure 5.5 shows the result of the algorithm when run for 20000 iterations and figure 5.6 shows the distribution when some objects are assigned with equal weights.

Although the current implementation of the program generator randomizes many aspects involved in creating test programs, it is bound to limitations. To include function calls in the test program the stack has to set-up which uses architecture specific registers.  

3. register r0 is fixed to zero;  
register r1 is used as stack pointer;  
register r2 is used as frame pointer;  
register r9 is used as Link address register;  

Figure 5.6: Plot showing random distribution containing equal weights
Register \( r13 \) is reserved to simplify accesses to data memory locations. The algorithm to select instruction arguments does not include these special registers. Some limitations are also introduced to avoid infinite loops during branches and subroutine calls. Figure 5.7 illustrates these constraints. The current implementation avoids recursions. Nested function calls are allowed but a call from the callee to the caller is prohibited. To simplify implementation of branch instructions, only forward branches are used. Generation of backward branches are restrained to avoid possible infinite loops or segmentation faults. Appendix B lists some examples of the generated test programs.

5.3 testbench controller

The design of the Testbench Controller is discussed in 4.3. The Testbench Controller handles many user configurations which can be listed using the command,

\[ ./tbc \text{-}h \]

Listing 5.5 illustrates these configurations.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-o \text{ output Dir})</td>
<td>option to specify the output directory</td>
</tr>
<tr>
<td>(-n \text{ No of programs})</td>
<td>option to specify the number of programs to be generated</td>
</tr>
<tr>
<td>(-p \text{ rtl sim path})</td>
<td>option to specify path of icarus rtl sim</td>
</tr>
<tr>
<td>(-e \text{ exename})</td>
<td>option to specify name of exe to execute</td>
</tr>
<tr>
<td>(-l \text{ error log})</td>
<td>option to specify name of file to log errors</td>
</tr>
<tr>
<td>(-i \text{ instruction log})</td>
<td>option to specify name of file to log instructions</td>
</tr>
<tr>
<td>(-h \text{ help})</td>
<td>lists the available help options, example: ( ./tbc \text{-}n 1 \text{-}o ./output )</td>
</tr>
</tbody>
</table>

Source Code Listing 5.5: Configurations for the Testbench Controller

An individual test case can be run by the Testbench Controller by using the command,

\[ ./tbc \text{-}e \text{ test} \]

while a batch of test programs (say 100 programs) can be executed using the command,

\[ ./tbc \text{-}n 100 \]

The Testbench Controller runs as the main thread of the application. It starts the two processor simulators and creates four threads each serving a particular functionality. The threads use POSIX message queues\(^4\) for internal communication which are also created by the main thread. The \textit{or1ksim\_thread} receives messages

\(^4\text{POSIX, is an acronym for Portable Operating System Interface. It is an IEEE Standard 1003.1-1988, which defines API\’s compatible with variants of Unix and other operating systems.}\)
from the Or1ksim ISS simulator and stores the register and memory information into the register queues. The `rtlsim_thread` receives and stores messages from the Icarus RTL simulator into the register queues. When the message queues are full, the `checker_thread` accesses the register queues and compares the result of simulation. To handle unexpected unforeseen scenarios like deadlocks in the Testbench Controller or any existing bugs in the processor model a `watchdog_timer` thread is used. The sequence diagram 5.8 illustrates the start-up sequence of the Testbench Controller.

The `watchdog_timer` thread expects timely updates from the sniffer threads, i.e. `or1ksim_thread` and `rtlsim_thread`, or otherwise halts the simulation. Such scenarios are interesting to analyze as they can reveal serious bugs in the processor models or in the co-simulation testbench. For example, the random program can enter an infinite loop without causing any change in the status register. In such scenarios, the sniffer modules will not receive any updates from the simulators. The `watchdog_timer` thread would then log the simulation results and terminate the co-simulation testbench. The `watchdog_timer` thread executes the following steps for systematic termination,
- closes running instances of the simulators
- logs differences in the register queues needed to analyze the unexpected program termination
- frees dynamically allocated data, closes the message queues and file handlers
- terminate the threads in the order of creation and exit the application

Since the POSIX message queues are created in a virtual file system, it is important to close and delete them after each simulation run. Also the existing implementation forbids multiple instances of the testbench to prevent the message queues from being overwritten.
Figure 5.9 illustrates the termination scenario caused by a watchdog error.

Abnormal terminations can also be caused due to system resource constraints or simulation errors. Some examples of this scenario are lack of memory in the heap, error in creating message queues or usage of null pointers etc. These scenarios cannot be detected by the watchdog timer but have to be handled in a similar way. Figure 5.10 illustrates the termination procedure which ensures logging of necessary information for post simulation analysis.

Under normal circumstances the main thread of Testbench Controller is responsible to log the register information and exit the application. Figure 5.11 illustrates the sequence diagram for normal termination. Even in normal scenarios it is also important to analyse the generated logs as the residual values in the register
queues can reveal serious bugs in the processor models.

Figure 5.11: Sequence diagram illustrating normal termination procedure.

5.4 sniffer modules

Sniffer modules indicated in figure 4.2 act as an interface to the simulators. The Testbench Controller creates POSIX message queues to interact with the simulators. The Or1ksim simulator is modified to send messages to the \textit{or1ksim\_thread} created by the main thread of the Testbench Controller 5.3. The value of the Program counter PC, Processor Status register, General Purpose Registers GPR’s are sent after the execution of each instruction. The values of the addressed data memory location are also sent after a store operation. In case of exceptions the Or1ksim ISS simulator also sends the exception vector address and value of Exception Effective Address Register, EEAR. The EEAR register
is set to the address of the instruction causing exception (Lampret, 2000).

The initial implementation of the RTL sniffer used GDB Remote Serial Protocol, RSP, to communicate with the GDP stub implemented in ORPSoC (Baxter, 2010). This method did not require any modifications in the existing RTL model. The existing implementation of the GDB stub interfaces the verilog simulator using VPI\(^5\) which in turn accesses the JTAG debug module external to the processor model. To retrieve the processor state information, the debug module stalls the processor after every instruction flushing the processor pipeline. Since this method is slow\(^6\) and does not simulate a real working scenario of a processor an alternative approach is used. The RTL model is modified to write state information into a POSIX pipe whenever the enable and write control signals for the registers/data memory are asserted. The information is read and manipulated by a sniffer module implemented in rtsniffer.c shown in figure 5.1. The sniffer module passes the state information using message queues to the rtsim_thread created by the main thread of the Testbench Controller 5.3. This approach, although requiring some modifications in the RTL model it does not alter the functional or timing specification of the RTL simulation. Moreover since it bypasses the RSP and JTAG communication ports it significantly improves the simulation time\(^7\).

The source code modification for the simulators to interface with the Testbench Controller are discussed in section A.

### 5.5 checker module

The Checker module is responsible to compare the states of the two processor models after execution of each instruction and is implemented as a thread, checker_thread. The Sequence diagram 5.8 illustrates the creation of the checker_thread. As discussed in section 4.3.1 Register queues are used to handle the skew problem. The sniffer modules update the Register queues and sends a message to the Checker module when they reach their capacity. The depth of the Register queue can be determined by the length of the processor pipeline but a typical value of 10 to 20 entries is sufficient. The current implementation uses a queue length of 16. Listing 5.6 shows the data structure used to implement the Register queue. The Register queue, regQueue, is implemented as a singly linked list of regElem. Each of the queue is associated with a counting semaphore and a mutex used to block the sniffer threads, i.e. or1ksim_thread and rtsim_thread, until the Checker module completes the consistency checks.

---

5. VPI, Verilog Procedural Interface is a part of the IEEE 1364 Programming Language Interface standard. It allows behavioral Verilog code to invoke C functions, and C functions to invoke standard Verilog system tasks.

6. The co-simulation run for a test program consisting of 50 instructions took approximately 8 minutes.

7. The co-simulation run for a test program consisting of 50 instructions took less than a minute.
Typedef struct reg_elem {
    int value;
    struct reg_elem* nextElem;
} regElem;

/// This data type represents the register file
typedef struct {
    int address;
    int size;
    pthread_mutex_t mutex;
    sem_t semaphore;
    regElem* data;
} regQueue;

The sniffer threads when blocked cannot receive messages from
the simulators and thereby are prevented from updating the Regis-
ter queues. The simulators though can update the processor states
until the POSIX message queues reach their maximum capacity.8
There can be scenarios when the two processor models produce
many inconsistencies causing the Register queue to be filled with
unequal values. In such cases the Register queues remain filled,
blocking the sniffer threads indefinitely. Such scenarios are de-
tected by the Checker module which then logs the Register queue
entries and terminates the simulation. The sequence diagram 5.10
illustrates this use case.

5.6 coverage monitor

As discussed in section 4.3.2 the Coverage Monitor module is
required to measure verification coverage which would consist of
building different coverage statistics. The current implementation
is limited to cover the number of instructions tested in each of
the test programs or in a test suite (consisting of number of test
programs). It is also possible to measure register coverage with
respect to a particular instruction used in a test program or in a
test suite.

The c file coverage.c as shown in 5.1 implements the coverage
module. It generates instruction statistics from the run time code
generated by the ISS simulator. These statistics are stored in the
log directory which has a directory structure as shown in figure
5.12. Coverage results for each of the test programs are separated
into different sub-folders. Further the results for each instruction
class is saved into different files as shown in 5.12. PHP scripts
are used to parse the coverage files and an open source JavaScript charting library, *Highcharts*\(^9\), is used to draw interactive instruction histograms. The results generated from the coverage module are detailed in the next chapter under the section 6.3.

---

9. Highcharts currently supports line, spline, area, areaplume, column, bar, pie and scatter chart types. More information can be found at, [http://www.highcharts.com/](http://www.highcharts.com/).
RESULTS AND DISCUSSION

6.1 introduction

This chapter summarizes the functional verification results obtained from the Co-Simulation platform for OR1200 core developed in this thesis work. It first presents the discrepancies found between OR1200 and Or1ksim ISS simulator. The next section details on the verification coverage results.

6.2 discrepancies between or1200 and or1ksim

6.2.1 l.mtspr implemented as l.nop in RTL model

The instruction l.mtspr belongs to the ORBIS32-II instruction class of the OpenRISC1000 architecture (Lampret, 2000). The instruction is used to move the contents from a general purpose register to a special purpose register. The instruction requires the processor to be in supervision mode. An instruction to change the content of Present Program Counter, PPC can be written as,

\[\begin{align*}
l.mfspr & \quad r3, r0, 16 \\
l.addi & \quad r3, r0, 460 \\
l.mtspr & \quad r0, r3, 16
\end{align*}\]

Source Code Listing 6.1: Code snippet illustrating use of l.mfspr instruction

The simulation results logged by the checker is illustrated in the listing 6.2.

The simulation results show the differences in the value of Program Counter. The l.mtspr instruction causes the Or1ksim, ISS simulator, to jump to an instruction pointed by the new value of PC while the RTL simulator implements the instruction as a l.nop and continues with the program execution. In some scenarios depending on the new value stored in PC after l.mtspr instruction, the execution in ISS simulator could end up in an infinite loop or could cause a segmentation fault. Such scenarios can be easily reproduced by loading random values to the PPC register.
Register r 3

or1k sim          rtl sim
901d

PC

or1k sim          rtl sim
1cc     238c
1d0     2390
1d4
1d0
1d4
1d0
1d4

SR

or1k sim          rtl sim
9005

Source Code Listing 6.2: simulation results for \texttt{l.mtspr} instruction

6.2.2 \textit{logical error in \texttt{l.rori}}

The instruction \texttt{l.rori} belongs to the ORBIS32-II instruction class of the OpenRISC1000 architecture (Lampret, 2000). The instruction is used to rotate contents of a general purpose register by a number of bit positions specified by 6 bit immediate value. An instruction sequence illustrated in 6.3 is generated in a test program. Listing 6.4 shows the results logged by checker.

\begin{verbatim}
1.addi   r3,r0,7
1.rori   r4,r3,1
\end{verbatim}

Source Code Listing 6.3: Code snippet for \texttt{l.rori} instruction

Register r 4

or1k sim          rtl sim
80000003        3

Source Code Listing 6.4: Simulation results for \texttt{l.rori} instruction

The simulation result logged by the checker module shows residual values generated for register \texttt{r4}. From the logs it is clear that the RTL model implements the \texttt{l.rori} instruction as a shift right instruction instead of a rotate instruction. After the rotate instruction register \texttt{r4} is expected to contain value of 80000003.
The implementation in the ISS is found to be correct as per the specification. While the RTL model contains value of 3 as a result of a shift operation.

Figure 6.1 shows the testbench waveforms\(^1\) showing the value of register \(r4\) after the shift operation.

![Figure 6.1: Testbench waveform showing value of register r3 being shifted instead of a rotate operation. The result of the operation is stored in register r4](image)

6.2.3 Illegal-instruction exception raised for \(l.ror\) instruction

The instruction \(l.ror\) belongs to the ORBIS32-II instruction class of the OpenRISC1000 architecture (Lampret, 2000). The instruction is used to rotate contents of a general purpose register by a number of bit specified by another general purpose register. The RTL model generates an illegal instruction exception when \(l.ror\) is generated in the test programs. Listing 6.5 illustrates the result of the simulation.

<table>
<thead>
<tr>
<th>SR</th>
<th>rtl sim</th>
<th>rtl sim</th>
</tr>
</thead>
<tbody>
<tr>
<td>orlk sim</td>
<td>9005</td>
<td>18001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EEAR</th>
<th>rtl sim</th>
<th>rtl sim</th>
</tr>
</thead>
<tbody>
<tr>
<td>orlk sim</td>
<td>2384</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E Vector</th>
<th>rtl sim</th>
<th>rtl sim</th>
</tr>
</thead>
<tbody>
<tr>
<td>orlk sim</td>
<td>700</td>
<td></td>
</tr>
</tbody>
</table>

Source Code Listing 6.5: Simulation results for \(l.ror\) instruction

The program counter is set to 0x700 when an illegal instruction is found in the instruction stream (Lampret, 2000). The simulation results show that the exception vector 0x700 is set only for RTL
model. Also it can be verified that the value of EEAR, Exception Effective Address Register, points to the address of the exception causing instruction.

6.2.4 Illegal-instruction exception raised for load/store instructions

The instruction \texttt{l.ld} and \texttt{l.sd} belongs to the ORBIS32-II instruction class of the OpenRISC1000 architecture (Lampret, 2000) and are used to perform double word load and store memory operations. The RTL model generates an illegal instruction exception when they are generated in the test programs. The usage of the instruction \texttt{l.lws} produces simulation results shown in listing 6.6. Figure 6.2 is the testbench waveform showing the contents of current program counter, next program counter and EEAR register after the faulting load/store instructions. These values match with the co-simulation results.

\begin{verbatim}
SR
orlk sim	rtl sim
9005
18001

EEAR
orlk sim	rtl sim
0
2384

E Vector
orlk sim	rtl sim
700
\end{verbatim}

Source Code Listing 6.6: Simulation results for \texttt{l.ld} or \texttt{l.sd} instruction

![Figure 6.2: Testbench waveform showing value of program counter and EEAR register after executing illegal load/store instruction.](image)
6.2.5 **Multiple updates for l.sub instruction**

The instruction `l.sub` belongs to the ORBIS32-II instruction class of the OpenRISC1000 architecture (Lampret, 2000) and is used to perform subtraction operation. Listing 6.7 shows an example of a subtraction operation followed by addition with carry operation (`l.addic`). The simulation results are shown in the second half of the listing. It is also interesting to discuss the run time traces generated by the testbench controller which is also shown in 6.8. According to the architecture specification (Lampret, 2000) the `l.sub` instruction does not alter the Carry Flag. But the simulation results 6.8 contradicts the specification. Moreover the results show that the RTL model causes multiple updates to the status register (the overflow flag is initially set and then reset) resulting in residual values in the register queue.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Function</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>l.addi</td>
<td>r3, r0, 1</td>
<td></td>
</tr>
<tr>
<td>l.addi</td>
<td>r4, r0, 2</td>
<td>orlk sim</td>
</tr>
<tr>
<td>l.sub</td>
<td>r5, r3, r4</td>
<td>rtl sim</td>
</tr>
<tr>
<td>l.addic</td>
<td>r6, r3, 3</td>
<td>8805</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8c05</td>
</tr>
</tbody>
</table>

Source Code Listing 6.7: The left half of the listing shows code snippet for `l.sub` instruction, while the right half illustrates the simulation results.

6.2.6 **Error in cache configuration for ORPSoC**

The RTL-model in ORPSoC can be configured using `refdesign-or1ksim.cfg` configuration file. The current implementation does not configure the caches (data and instruction cache) from the input given in the configuration files. This causes differences in the simulation results for the processor status register. This configuration error can be avoided by programmatically disabling the caches as illustrated in the listing 6.9.

6.3 **the openrisc 1200 verification coverage results**

The coverage monitor implemented is used to generate graphical results as explained in section 5.6. Coverage results generated for a test suite consisting of 50 test programs can be accessed at, [http://home.student.uu.se/anka2609/index.php](http://home.student.uu.se/anka2609/index.php).

Figure 6.3 shows the coverage statistics of a test suite consisting of 200 test programs with each program consisting of 100 random instructions. The graphs are drawn by grouping the instructions
or1ksim: sprs 0 : 2314 and size : 0
or1ksim: sprs 0 : 2318 and size : 1
or1ksim: reg 3 : 1 and size : 0
or1ksim: sprs 0 : 231c and size : 2
or1ksim: reg 4 : 2 and size : 0
or1ksim: sprs 0 : 2320 and size : 3
or1ksim: sprs 1 : 8405 and size : 0
or1ksim: reg 5 : ffffffff and size : 0
or1ksim: sprs 0 : 2324 and size : 4
or1ksim: sprs 1 : 8005 and size : 1
or1ksim: reg 6 : 5 and size : 0
or1ksim: sprs 0 : 2328 and size : 5
or1ksim_thread:or1ksim_thread exiting
watchdog_timer: received LAST_INST_SIM
checker_thread: received LAST_INST_SIM

rtlsim_thread: sprs 0 : 2314 and size : 0
rtlsim_thread: reg 3 : 1 and size : 0
rtlsim_thread: sprs 0 : 2318 and size : 1
rtlsim_thread: reg 4 : 2 and size : 0
rtlsim_thread: sprs 0 : 231c and size : 2
rtlsim_thread: sprs 1 : 8805 and size : 0
rtlsim_thread: reg 5 : ffffffff and size : 0
rtlsim_thread: sprs 0 : 2320 and size : 3
rtlsim_thread: sprs 1 : 8c05 and size : 1
rtlsim_thread: sprs 1 : 8405 and size : 2
rtlsim_thread: reg 6 : 5 and size : 0
rtlsim_thread: sprs 0 : 2324 and size : 4
rtlsim_thread: sprs 1 : 8005 and size : 3
rtlsim_thread: sprs 0 : 2328 and size : 5

rtlsim_thread: rtlsim_thread exiting
checker_thread: received LAST_INST_RTL
checker_thread:checker thread exiting
watchdog_thread: received LAST_INST_RTL
watchdog_thread: watchdog thread exiting
tbc_main:tbc exiting

Source Code Listing 6.8: run time instruction traces for test program containing l.sub

#disable data and instruction cache
l.mfspr r3,r0,17
l.addi r4,r0,0xffe7
l.and r3,r3,r4
l.mtspr r0,r3,17

Source Code Listing 6.9: Code snippet for disabling caches

into instruction classes. It is possible to generate similar charts for any particular test program. For example the instruction coverage for test1.s is shown in figure 6.4 and 6.5. The figure in
Figure 6.3: The chart illustrates Instruction coverage for a test suite consisting of 200 test programs.

Figure 6.4: The chart illustrates instruction coverage grouped in classes for an example test program, test1.s.

Figure 6.5: The chart illustrates arithmetic instruction coverage for an example test program, test1.s.

The left provides coverage in terms of instruction classes. A more detailed instruction coverage, as shown in the right figure for logical instruction class, can be viewed by clicking the bar of a particular instruction class.

Figure 6.6 illustrates the register coverage statistics generated for three test programs, test1.s, test2.s and test3.s. Any arbitrary number of test programs and instructions can be selected from a drop-down list provided on the webpage. The graphs provide an interactive interface allowing to further filter the results for a particular program or programs. It is interesting to see and verify register coverage for certain instruction classes. For example, register coverage obtained for all the store instructions is shown.
Figure 6.6: The chart illustrates interactive register coverage for three test programs.

Figure 6.7: The chart illustrates coverage statistics for store instruction class.
in figure 6.7. The figure illustrates that \( r13 \) and \( r1 \) were the only registers used to generate the store instructions. Section 5.2 lists the architecture specific registers reserved for specific operations. Register \( r13 \) and \( r1 \) are dedicated registers, \( r13 \) being a pointer to data section and \( r1 \) is reserved as a stack pointer. Thus the store operation which comprises of data and stack memory accesses are limited to use these dedicated registers. Another interesting

Figure 6.8: The chart illustrates coverage statistics for branch and call instructions.

coverage is for branch and call instructions. Figure 6.8 illustrates the coverage obtained for these instruction classes. The high coverage obtained for register \( r9 \) is justified as it is dedicated as a Link Address Register. From figures 6.6, 6.7 and 6.8 it can also be inferred that the coverage obtained for register \( r0 \) is always zero. This is because the architecture specifies \( r0 \) as a zero register\(^2\). Also figure coverage for register \( r1 \) and \( r2 \) obtained in figure 6.6 can be justified for their use in allocating and deallocating the stack.

The current implementation can only generate coverage statistics for the basic instruction set and floating point extensions of the OpenRISC core. OpenRISC DSP/Vector instruction sets are not generated and hence do produce any coverage statistics.

\(^2\) The value of the register is forced to 0 (Lampret, 2000).
CONCLUSION AND FUTURE WORK

7.1 conclusion

This thesis work was carried out to provide a co-simulation framework for dynamic functional verification of an OpenRISC processor core, OR1200. To achieve this goal a Random Assembly Program Generator and a Co-simulation testbench platform was implemented. The verification results discussed in section 6.2 substantiates the correctness of the Testbench Controller. Furthermore the discrepancies found in the RTL-model are verified against the test bench waveforms. The inconsistencies are brought into notice of ORSoC AB\(^1\) and are reported in bugzilla\(^2\). We expect that the inconsistencies found in this thesis work will be updated in the specification documents and the design of OR1200 core will be checked for correctness.

The coverage results discussed in section 6.3 proves that the large volumes of random test programs can be generated based on the configurable instruction weights. The histograms can easily be used to generate instruction, register and branch coverage statistics either for individual test programs or for large test volumes.

As discussed in section 4.2 the Random Program Generator runs independent of the Testbench Controller. The provision to control the generation of a particular class of instruction offers more flexibility to the generator. Also the possibility to regenerate interesting test programs based on the input seed values avoids the need to save large test volumes.

7.2 future work

The Co-Simulation test bench implemented as a part of this thesis work provides basic instruction and register coverage statistics. Extending the coverage monitor module to include sequence coverage, will provide interesting statistics on occurrence of certain sequence of instructions. The configuration files can be supplemented with

---

1. ORSoC AB is the owner and maintainer of Opencores [http://orsoc.se/](http://orsoc.se/)

2. Bugzilla is an open source bug tracking tool. The bugs for the open RISC OR1200 core can be found at, [http://bugzilla.opencores.org/](http://bugzilla.opencores.org/), under the ORPSoC category. (Bug Id: 95, 96, 97 and 98)
register weights to force usage of certain registers that have poor coverage statistics.

The existing implementation of the coverage monitor does not filter the discrepancies resulting from resource collisions in the pipeline. So it is required to manually analyze the simulation logs to exclude these discrepancies. Also it is difficult to analyze the simulation logs to identify the fault causing instruction or instruction sequence. This method is laborious and needs careful examination of the logs. Implementing an automatic fault detection module to locate the faulty instruction or minimizing the search space could subsequently reduce the manual effort required to track the bug.

The Random Program Generator can be extended to include a feedback loop from the coverage monitor to increase the coverage statistics after each simulation run. This method would decrease the throughput of the generator but at the cost of producing better programs.

In the current implementation the memory values are compared against their virtual addresses. Though this method sufficiently tests the functional behavior of the processor, the implementation can be extended to include the memory management units and possibly even the processor cache designs. The existing implementation of Or1ksim, ISS simulator does not support Hardware architecture\(^3\) and also since the cache model in OR1200, implements a physically indexed physically tagged design (Lampret, 2000) it is difficult to include these modules.

The Test Bench Controller can be redesigned to run efficiently on a multiprocessor or multi-core systems. This can help in improving the simulation speed.
Aharon, A.; Goodman, Dave; Levinger, Moshe; Lichtenstein, Yossi; Malka, Yossi; Metzger, Charlotte; Molcho, Moshe; Shurek, Gil; and Metzger, Yossi Malka Charlotte. 1995. *Test Program Generation for Functional Verification of PowerPC Processors in IBM*. Cited on pp. 1 and 4.


Psarakis, Mihalis; Gizopoulos, Dimitris; Hatzimihail, Miltiadis; Paschalis, Antonis; Raghunathan, Anand; and Ravi, Srivaths. 2006. *Systematic Software-Based Self-Test for Pipelined Processors*. Cited on pp. 4 and 5.

Semeria, Luc; Seawright, Andrew; Mehra, Renu; Ng, Daniel; Ekanayake, Arjuna; and Pangrle, Barry. June 2002. *RTL C-Based Methodology for Designing and Verifying a Multi-Threaded Processor*. Cited on p. 4.


The Test Bench Controller and the Random Program Generator are implemented in c and their source code can be checked out from subversion (SVN) repository\(^1\) using the command,

\[
\text{svn co http://svn.orsoc.se/svn/random-generator}
\]

Figure 5.1 outlines the organization of the project. For the sake of clarity only the \$.c files under the \texttt{trunk/random-testing} are listed. The implementation work is documented using Doxygen\(^2\). The html and pdf versions of the source code documentation can be found under \texttt{trunk/random-testing/docs} directory. The Coverage statistics are generated using \texttt{trunk/random-testing/coverage.c}. The HTML code and the scripts used to generate the instruction histograms can be accessed under \texttt{trunk/random-testing/www} directory.

The source code for Or1ksim, ISS simulator, found in the directory \texttt{trunk/or1ksim} contains the modifications to interface it with the Test Bench Controller. The source code for the unmodified version of the ISS simulator can be checked out using the command,

\[
\text{svn co http://opencores.org/ocsvn/openrisc/openrisc/trunk/or1ksim}
\]

The changes made to the RTL-model can be found in the directory \texttt{trunk/orpsocv2/rtl/verilog}. When the \texttt{write} control signals for the register and memory modules are asserted their values are written onto a Posix pipe. This value is read using a RTLsniffer module implemented in \texttt{trunk/random-testing/rtlsniffer.c}. The unmodified version of the RTL model can be checked out using the command,

\[
\text{svn co http://opencores.org/ocsvn/openrisc/openrisc/trunk/orpsocv2}
\]

The changes made to the processor models can be listed by comparing the unmodified version\(^3\) with the version present in the directory structure in figure Figure 5.1.

1. Please contact ORSoC AB for access privileges
2. Doxygen is an open source documentation system supporting various programming languages. More information can be found at, \url{http://www.stack.nl/~dimitri/doxygen/}
3. The latest version of the simulators can be downloaded from the OpenCores SVN repository at, \url{http://opencores.org/websvn/listing/openrisc}
EXAMPLE TEST PROGRAMS

Listing B.1 and B.2 refers to randomly generated test programs test1.s and test2.s. The run time code generated for each of the test programs are illustrated in B.3 and B.4 respectively. From the listings we can see that the run time code can be much larger than the static code as in the case of test1.s or can be much smaller as in the case of test2.s. The program counter value listed in the run time code is used for post simulation analysis.

1  /**
2  /* @file test1.s
3  *
4  * @brief this file was generated from random generator program
5  *
6  * @date Mon Jun  4 21:28:55 2012
7  *
8  */
9
10 #start of data segment
11 .global data
12 .data
13 .align 4
14 data:
15 .4byte 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
16 .4byte 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
17 .4byte 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31
18 .4byte 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42
19 .4byte 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53
20 .4byte 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64
21 .4byte 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75
22 .4byte 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86
23 .4byte 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97
24 .4byte 98, 99
#start of code segment

```
.text
.align 4

.space 6773828

PROC0:
#allocate frame
1.sw   -8(r1), r2
1.addi r2,r1,0
1.sw   -4(r1),r9
1.addi r1,r1,-12
1.msync
1.mfspr r23,r20,26

#deallocate frame
1.ori  r1,r2,0
1.lwz  r2,-8(r1)
1.lwz  r9,-4(r1)
1.jr   r9
1.nop

.space 27064

PROC1:
#allocate frame
1.sw   -8(r1), r2
1.addi r2,r1,0
1.sw   -4(r1),r9
1.addi r1,r1,-12
1.jal  PROC0
1.bnf  5
1.lhz  r6,16(r13)
1.msync
1.sw   20(r13),r20
1.cmov r16,r27,r23
1.sh   20(r13),r26
1.mfspr r22,r0,16
1.addi r22,r22,12
1.jr   r22
1.lwz  r15,16(r13)
1.cmov r8,r10,r25
1.psync
1.sfgeui r30,0

#deallocate frame
1.ori  r1,r2,0
1.lwz  r2,-8(r1)
1.lwz  r9,-4(r1)
1.jr   r9
1.nop
```
.space 167064
PROC2:
    #allocate frame
    l.sw        -8(r1), r2
    l.addi      r2, r1, 0
    l.sw        -4(r1), r9
    l.addi      r1, r1, -12
    l.lbs       r5, 12(r13)
    l.mfspr     r29, r25, 54
    l.mfspr     r27, r14, 74
    l.mfspr     r15, r27, 69
    l.psync
    #deallocate frame
    l.ori       r1, r2, 0
    l.lwz       r2, -8(r1)
    l.lwz       r9, -4(r1)
    l.jr         r9
    l.nop

.global main
.global first
.global last
.space 4384
main:
    #allocate frame
    l.sw        -8(r1), r2
    l.addi      r2, r1, 0
    l.sw        -4(r1), r9
    l.addi      r1, r1, -12
    #enable interrupts and supervision bit
    l.mfspr     r3, r0, 17
    l.ori       r3, r3, 5
    l.mtspr     r0, r3, 17
    #disable data and instruction cache
    l.mfspr     r3, r0, 17
    l.addi      r4, r0, 0xffe7
    l.and       r3, r3, r4
    l.mtspr     r0, r3, 17
    #make r13 to point to data section
    l.movhi     r13, hi(data)
    l.ori       r13, r13, lo(data)
    #initialize registers
    l.addi      r3, r0, 3
    l.addi      r4, r0, 4
    l.addi      r5, r0, 5
    l.addi      r6, r0, 6
    l.addi      r7, r0, 7
    l.addi      r8, r0, 8
    l.addi      r10, r0, 10
    l.addi      r11, r0, 11
    l.addi      r12, r0, 12
132 l.addi   r14, r0, 14
133 l.addi   r15, r0, 15
134 l.addi   r16, r0, 16
135 l.addi   r17, r0, 17
136 l.addi   r18, r0, 18
137 l.addi   r19, r0, 19
138 l.addi   r20, r0, 20
139 l.addi   r21, r0, 21
140 l.addi   r22, r0, 22
141 l.addi   r23, r0, 23
142 l.addi   r24, r0, 24
143 l.addi   r25, r0, 25
144 l.addi   r26, r0, 26
145 l.addi   r27, r0, 27
146 l.addi   r28, r0, 28
147 l.addi   r29, r0, 29
148 l.addi   r30, r0, 30
149 l.addi   r31, r0, 31
150
151     first:
152     l.nop
153     l.lbs r27, 8(r13)
154     l.jal PROC1
155     l.sfgt s r17, r12
156     l.sfgt u r3, r19
157     l.bf 68
158     l.sb 4(r13), r30
159     l.sb 0(r13), r23
160     l.lhz r25, 8(r13)
161     l.mov hi r29, hi(PROC1)
162     l.ori r29, r29, lo(PROC1)
163     l.jalr r29
164     l.lwz r19, 12(r13)
165     l.csync
166     l.jal PROC1
167     l.sb 12(r13), r12
168     l.sfgt ui r29, 71
169     l.addi r26, r29, 28
170     l.psyc
171     l.mfspr r8, r8, 42
172     l.lbs r5, 4(r13)
173     l.ff1 r6, r27
174     l.psyc
175     l.mac r5, r15
176     l.andi r8, r24, 24
177     l.mfspr r16, r15, 89
178     l.psyc
179     l.lbs r17, 12(r13)
180     l.mfspr r19, r17, 60
181     l.lhz r14, 8(r13)
182     l.muli r15, r30, 26
183     l.cmov r24, r20, r8
184     l.lbs r6, 12(r13)
185     l.sh 20(r13), r17
186     l.jal PROC2
187 l.lhs r8,20(r13)
188 l.andi r17,r15,14
189 l.lhz r18,12(r13)
190 l.muli r30,r31,32
191 l.addi r18,r25,15
192 l.msync
193 l.csync
194 l.sw 24(r13),r26
195 l.movhi r26,hi(PROC2)
196 l.ori r26,r26,lo(PROC2)
197 l.jalr r26
198 l.sfleui r10,3
199 l.csync
200 l.movhi r3,34
201 l.msync
202 l.csync
203 l.div r16,r10,r14
204 l.cmov r24,r21,r10
205 l.sfltuui r15,52
206 l.lhs r22,24(r13)
207 l.sfgeui r29,80
208 l.cmov r10,r18,r3
209 l.jal PROC2
210 l.sb 8(r13),r4
211 l.lbs r14,24(r13)
212 l.lwz r15,4(r13)
213 l.mfspr r8,r12,77
214 l.jal PROC1
215 l.sh 20(r13),r16
216 l.sfneui r31,28
217 l.sw 16(r13),r4
218 l.muli r24,r20,5
219 l.ori r12,r19,11
220 l.lhs r5,12(r13)
221 l.csync
222 l.lbs r18,16(r13)
223 l.mfspr r10,r15,57
224 l.mfspr r18,r20,41
225 l.csync
226 l.csync
227 l.cmov r26,r29,r26
228 l.addi r27,r10,92
229 l.cmov r30,r21,r23
230 l.addi r8,r3,87
231 l.sh 12(r13),r6
232 last:
233 l.nop
234
235 #deallocate frame
236 l.ori r1,r2,0
237 l.lwz r2,8(r1)
238 l.lwz r9,4(r1)
239 l.jr r9
240 l.nop

Source Code Listing B.1: example test program test1.s
/* @file test2.s */
* @brief this file was generated from random generator program *
* @date Mon Jun 4 21:28:55 2012 *
*/

#start of data segment
.globl data
.data
.align 4
data:
.globl PROC0
.data
.globl main
.globl first
.globl last

#start of code segment
.text
.align 4
.space 2548880
.PROC0:
#allocate frame
l.sw -8(r1), r2
l.addi r2,r1,0
l.sw -4(r1),r9
l.addi r1,r1,-12
l.sw 16(r13),r19
l.macrc r8
l.j 1
l.msync
l.lbz r28,12(r13)
l.lhs r20,4(r13)
l.srli r10,r10,59
#deallocate frame
l.ori r1,r2,0
l.lwz r2,-8(r1)
l.lwz r9,-4(r1)
l.jr r9
l.nop
.globl main
.globl first
.globl last
main:

# allocate frame
1. sw -8(r1), r2
l.addi r2, r1, 0
1. sw -4(r1), r9
l.addi r1, r1, -12

# enable interrupts and supervision bit
l.mfspr r3, r0, 17
l.ori r3, r3, 5
l.mtspr r0, r3, 17

# disable data and instruction cache
l.mfspr r3, r0, 17
1. addi r4, r0, 0xffe7
1. and r3, r3, r4
1. mtspr r0, r3, 17

# make r13 to point to data section
l.movhi r13, hi(data)
l.ori r13, r13, lo(data)

# initialize registers
1. addi r3, r0, 3
1. addi r4, r0, 4
1. addi r5, r0, 5
1. addi r6, r0, 6
1. addi r7, r0, 7
1. addi r8, r0, 8
1. addi r10, r0, 10
1. addi r11, r0, 11
1. addi r12, r0, 12
1. addi r14, r0, 14
1. addi r15, r0, 15
1. addi r16, r0, 16
1. addi r17, r0, 17
1. addi r18, r0, 18
1. addi r19, r0, 19
1. addi r20, r0, 20
1. addi r21, r0, 21
1. addi r22, r0, 22
1. addi r23, r0, 23
1. addi r24, r0, 24
1. addi r25, r0, 25
1. addi r26, r0, 26
1. addi r27, r0, 27
1. addi r28, r0, 28
1. addi r29, r0, 29
1. addi r30, r0, 30
1. addi r31, r0, 31
first:

l.nop
l.add r17, r22, r20
l.lbs r15, 20(r13)

l.bnf 82
l.addic r16, r17, 88
l.addic r17, r5, 4
l.jal PROC0
l.mfspr r23, r18, 69
l.sfgeu r27, r16
l.sub r16, r10, r5
l.movhi r18, hi(PROC0)
l.or r18, r18, lo(PROC0)
l.jalr r18
l.mfspr r3, r17, 89
l.or r22, r18, r30
l.mfspr r6, r31, 86
l.sh 8(r13), r17
l.csync
l.sfne r6, r10
l.addic r21, r6, 18
l.rori r23, r10, 1
l.sb 12(r13), r10
l.msync
l.jal PROC0
l.muli r10, r3, 3
l.psync
l.lbs r15, 4(r13)

l.addi r16, r17, 87
l.lbz r12, 8(r13)

l.movhi r28, 12
l.sb 0(r13), r14
l.sfleui r10, 67
l.addi r4, r26, 56
l.msync
l.csync
l.msync
l.movhi r30, 46

l.ori r25, r15, 29
l.msync
l.muli r27, r26, 71
l.divu r29, r29, r31
l.mfspr r19, r4, 96
l.cmov r6, r19, r23
l.srli r26, r29, 50
l.sub r21, r12, r25
l.sh 4(r13), r21
l.sh 8(r13), r22
l.sub r26, r30, r7
l.movhi r8, 9
l.sh 24(r13), r5
l.addc r30, r3, r23
l.mac i r15, 7
l.jal PROC0
Source Code Listing B.2: example test program test2.s
1 PPC:6a8768  l.lbs r27  0x8(r13)  
2 PPC:6a876c  l.jal  -42941 
3 PPC:6a8770  l.sfgts r17  r12 
4 PPC:67e878  l.sw  -8(r1)  r2  
5 PPC:67e87c  l.addi r2  r1  0x0 
6 PPC:67e880  l.sw  -4(r1)  r9 
7 PPC:67e884  l.addi r1  r1  -12 
8 PPC:67e888  l.jal  -6781 
9 PPC:67e88c  l.bnf  0x5 
10 PPC:677e94  l.sw  -8(r1)  r2  
11 PPC:677e98  l.addi r2  r1  0x0 
12 PPC:677e9c  l.sw  -4(r1)  r9 
13 PPC:677ea0  l.addi r1  r1  -12 
14 PPC:677ea4  l.msync 
15 PPC:677ea8  l.mfspr r23  r20  0x1a 
16 PPC:677eac  l.ori r1  r2  0 
17 PPC:677eb0  l.lwz r2  -8(r1) 
18 PPC:677eb4  l.lwz r9  -4(r1) 
19 PPC:677eb8  l.jr  r9 
20 PPC:677ebe  l.nop  0 
21 PPC:67e890  l.lhz r6  0x10(r13) 
22 PPC:67e894  l.msync 
23 PPC:67e898  l.sw  0x14(r13)r20 
24 PPC:67e89c  l.cmov r16  r27  r23 
25 PPC:67e8a0  l.sh  0x14(r13)r26 
26 PPC:67e8a4  l.mfspr r22  r0  0x10 
27 PPC:67e8a8  l.addi r22  r22  0xc 
28 PPC:67e8ac  l.jr  r22 
29 PPC:67e8b0  l.lwz r15  0x10(r13) 
30 PPC:67e8b0  l.lwz r15  0x10(r13) 
31 PPC:67e8b4  l.cmov r8  r10  r25 
32 PPC:67e8b8  l.psnc 
33 PPC:67e8bc  l.sfguir300x0 
34 PPC:67e8c0  l.ori r1  r2  0 
35 PPC:67e8c4  l.lwz r2  -8(r1) 
36 PPC:67e8c8  l.lwz r9  -4(r1) 
37 PPC:67e8cc  l.jr  r9 
38 PPC:67e8d0  l.nop  0 
39 PPC:67e8d4  l.sfgtu r3  r19 
40 PPC:67e8d8  l.bf  0x44 
41 PPC:67e8dc  l.sb  0x4(r13)r30 
42 PPC:67e8d8  l.sb  0x0(r13)r23 
43 PPC:67e8e4  l.lhz r25  0x8(r13) 
44 PPC:67e8e8  l.movhi r29  0x67 
45 PPC:67e8ec  l.ori r29  r29  0xe878 
46 PPC:67e8f0  l.jalr  r29 
47 PPC:67e8f4  l.lwz r19  0xc(r13) 
48 PPC:67e8f8  l.sw  -8(r1)  r2 
49 PPC:67e8fc  l.addi r2  r1  0x0 
50 PPC:67e900  l.sw  -4(r1)  r9 
51 PPC:67e904  l.addi r1  r1  -12 
52 PPC:67e908  l.jal  -6781 
53 PPC:67e90c  l.bnf  0x5 
54 PPC:67e910  l.sw  -8(r1)  r2
56 PPC:67e8a0  l.sh  0x14(r13)r26
57 PPC:67e8a4  l.mfspr r22  r0  0x10
58 PPC:67e8a8  l.addi  r22  r22  0xc
59 PPC:67e8ac  l.jr  r22
60 PPC:67e8b0  l lwz  r15  0x10(r13)
61 PPC:67e8b0  l lwz  r15  0x10(r13)
62 PPC:67e8b4  l.cmov  r8  r10  r25
63 PPC:67e8b8  l.psnc
64 PPC:67e8bc  l.sfguir300x0
65 PPC:67e8c0  l.ori  r1  r2  0
66 PPC:67e8c4  l lwz  r2  -8(r1)
67 PPC:67e8c8  l lwz  r9  -4(r1)
68 PPC:67e8cc  l.jr  r9
69 PPC:67e8d0  l nop  0
70 PPC:6a8798  l.csnc
71 PPC:6a879c  l jal  -42953
72 PPC:6a87a0  l sb  0xc(r13)r12
73 PPC:67e878  l sw  -8(r1)  r2
74 PPC:67e87c  l.addi  r2  r1  0x0
75 PPC:67e880  l sw  -4(r1)  r9
76 PPC:67e884  l.addi  r1  r1  -12
77 PPC:67e888  l jal  -6781
78 PPC:67e88c  l bnf  0x5
79 PPC:67e994  l sw  -8(r1)  r2
80 PPC:67e9b8  l.addi  r2  r1  0x0
81 PPC:67e9c  l sw  -4(r1)  r9
82 PPC:67ea0  l.addi  r1  r1  -12
83 PPC:67ea4  l msync
84 PPC:67ea8  l.mfspr  r23  r20  0x1a
85 PPC:67eac  l.ori  r1  r2  0
86 PPC:67eb0  l lwz  r2  -8(r1)
87 PPC:67eb4  l lwz  r9  -4(r1)
88 PPC:67eb8  l jr  r9
89 PPC:67ebc  l nop  0
90 PPC:67e90  l lhz  r6  0x10(r13)
91 PPC:67e94  l msync
92 PPC:67e98  l sw  0x14(r13)r20
93 PPC:67e9c  l cmov  r16  r27  r23
94 PPC:67ea0  l sh  0x14(r13)r26
95 PPC:67ea4  l mfspr  r22  r0  0x10
96 PPC:67ea8  l.addi  r22  r22  0xc
97 PPC:67eac  l.jr  r22
98 PPC:67eb0  l lwz  r15  0x10(r13)
99 PPC:67eb0  l lwz  r15  0x10(r13)
100 PPC:67eb4  l.cmov  r8  r10  r25
101 PPC:67eb8  l.psnc
102 PPC:67ebc  l.sfguir300x0
103 PPC:67ec0  l.ori  r1  r2  0
104 PPC:67ec4  l lwz  r2  -8(r1)
105 PPC:67ec8  l lwz  r9  -4(r1)
106 PPC:67ecc  l jr  r9
107 PPC:67ed0  l nop  0
108 PPC:6a87a4  l.sfgtuir290x47
109 PPC:6a87a8  l.addi  r26  r29  0x1c
110 PPC:6a87ac  l.psnc
0x2a
0x3c
0x1a
0xc
0x36
0x4a
0x45
0x13
0x20
0xf
0x6a
0x756c
0x8
0x6a
0x6a
0x6a
0x6a
166: PPC:6a75a0 l.nop 0
167: PPC:6a8820 l.csync
168: PPC:6a8824 l.movhi r3 0x22
169: PPC:6a8828 l.msinc
170: PPC:6a882c l.csync
171: PPC:6a8830 l.div r16 r10 r14
172: PPC:6a8834 l.cmov r24 r21 r10
173: PPC:6a8838 l.sfiltu150x34
174: PPC:6a883c l.lhs r22 0x18(r13)
175: PPC:6a8840 l.sfgeuir290x50
176: PPC:6a8844 l.cmov r10 r18 r3
177: PPC:6a8848 l.jal -1207
178: PPC:6a884c l.sb 0x8(r13)r4
179: PPC:6a756c l.sw -8(r1) r2
180: PPC:6a7570 l.addi r2 r1 0x0
181: PPC:6a7574 l.sw -4(r1) r9
182: PPC:6a7578 l.addi r1 r1 -12
183: PPC:6a757c l.lbs r5 0xc(r13)
184: PPC:6a7580 l.mfspr r29 r25 0x36
185: PPC:6a7584 l.mfspr r27 r14 0x4a
186: PPC:6a7588 l.mfspr r15 r27 0x45
187: PPC:6a758c l.psinc
188: PPC:6a7590 l.ori r1 r2 0
189: PPC:6a7594 l.lwz r2 -8(r1)
190: PPC:6a7598 l.lwz r9 -4(r1)
191: PPC:6a759c l.jr r9
192: PPC:6a75a0 l.nop 0
193: PPC:6a8850 l.lbs r14 0x18(r13)
194: PPC:6a8854 l.lwz r15 0x4(r13)
195: PPC:6a8858 l.mfspr r8 r12 0x4d
196: PPC:6a885c l.jal -43001
197: PPC:6a8860 l.sh 0x14(r13)r16
198: PPC:67e878 l.sw -8(r1) r2
199: PPC:67e87c l.addi r2 r1 0x0
200: PPC:67e880 l.sw -4(r1) r9
201: PPC:67e884 l.addi r1 r1 -12
202: PPC:67e888 l.jal -6781
203: PPC:67e88c l.bn 0x5
204: PPC:67e994 l.sw -8(r1) r2
205: PPC:67e9a0 l.sh 0x14(r13)r26
206: PPC:67e9a0 l.mfspr r22 r0 0x10
207: PPC:67e9a8 l.addi r22 r22 0xc
208: PPC:67e9ac l.jr r22
209: PPC:67eb0 l.lwz r15 0x10(r13)
210: PPC:67eb0 l.lwz r15 0x10(r13)
211: PPC:67eb4 l.cmov r8 r10 r25
212: PPC:67eb8 l.psinc
213: PPC:67ebc l.sfgeuir300x0
214: PPC:67ec0 l.ori r1 r2 0
215: PPC:67ec4 l.lwz r2 -8(r1)
216: PPC:67ec8 l.lwz r9 -4(r1)
217: PPC:67ecc l.jr r9
218: PPC:67ed0 l.nop 0
219: PPC:6a8864 l.sfnei r31 0x1c

63
Source Code Listing B.3: Run time code for test program test1.s

1 PPC:3a8b64 l.add r17 r22 r20
2 PPC:3a8b68 l.lbs r15 0x14(r13)
3 PPC:3a8b6c l.bnf 0x52
4 PPC:3a8b70 l.addic r16 r17 0x58
5 PPC:3a8cb4 l.movhi r15 0x28
6 PPC:3a8cb8 l.msync
7 PPC:3a8cbc l.mfspr r19 r0 0x10
8 PPC:3a8cc0 l.addi r19 r19 0xc
9 PPC:3a8cc4 l.jr r19
10 PPC:3a8cc8 l.macrc r28
11 PPC:3a8cc8 l.macrc r28
12 PPC:3a8ccc l.sw 0xc(r13) r15
13 PPC:3a8cd0 l.lhz r26 0x18(r13)
14 PPC:3a8cd4 l.nop 0

Source Code Listing B.4: Run time code for test program test2.s

64