IMPLEMENTATION AND VERIFICATION OF A CPU SUBSYSTEM FOR MULTIMODE RF TRANSCEIVERS

by

Waqas Ahmed
<waqasw@kth.se>

Supervisor:

Brandstaetter Siegfried
Infineon Technologies (DICE), Austria

Internal Supervisor:

Ahmed Hemani
Professor, Royal Insitute of Technology (KTH)

A thesis submitted to the faculty of Royal Institute of Technology (KTH) in partial fulfillment of the requirements for the degree of

Masters of System on Chip Design

Department of Information and Communication Technology

Royal Institute of Technology, Sweden

May 2010

ABSTRACT

IMPLEMENTATION AND VERIFICATION OF A CPU SUBSYSTEM FOR MULTIMODE RF TRANSCEIVERS

Waqas Ahmed

Department of ICT

Master of Science

Multimode transceivers are becoming a very popular implementation alternative because of their ability to support several standards on a single platform. For multimode transceivers, advanced control architectures are required to provide flexibility, reusability, and multi-standard support at low power consumption and small die area effort. In such an advance control architecture the CPU Subsystem functions as a central control unit which configures the transceiver and the interface for a particular communication standard.

Open source components are gaining popularity in the market because they not only reduce the design costs significantly but also provide power to the designer due to the availability of the full source code. However, open source architectures are usually available as poorly verified and untested intellectual properties (IPs). Before they can be commercially adapted, an extensive testing and verification strategy is required. In this thesis we have implemented a CPU Subsystem using open source components and performed the functional verification of this Subsystem. The main components of this CPU Subsystem are (i) an open source OpenRISC1200 core, (ii) a memory system, (iii) a triple-layer Sub-bus system and (iv) several Wishbone interfaces. The OpenRISC1200 core was used because it is a 32-bit core ideally suited for applications requiring high performance while having low-cost and low power consumption. The verification of a 5-stage pipeline processor is a challenging task and to the best of our knowledge this is the first attempt to verify the OpenRISC1200 core. The faults identified as a result of the functional verification will not only prove useful for the current project but will likely make the OpenRISC1200 core a more reliable and commercially used processor.

ACKNOWLEDGMENTS

First of all, I would like to thank God, the Almighty, for having made everything possible for me by giving me strength and courage to do this work.

My deepest gratitude to Brandstaetter Siegfried, my supervisor, for his unselfishness, encouragement and guidance and patience he demonstrated during my work.

Further I would like to thank my examiner Professor Ahmed Hemani at the Department of Information and Communication Technology at Royal Insitute of Technology (KTH) for undertaking my Master's thesis.

I am also deeply indebted to Professor Andreas Springer at Institute of Communication Engineering and RF-Systems at Johannes Kepler University for offering me this opportunity.

I would also like to thank the other employees at DICE and especially Dr. Neurauter Burkhard, Dr. Hueber Gernot and Steinmayr Christian for every help during my work.

Sincere thanks to my family, relatives and friends who all gave me courage and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

WIRELESS communication is a rapidly growing division of communication industry in which high quality information can be transferred at high-speed between the portable devices located anywhere in the world. Applications of wireless technology are everywhere including cell phones, home appliances, teleconferencing, satellite communication and much more. However, the development of the wireless systems is a considerable challenge.

Every device that incorporates wireless communication typically comprises of three core components: (i) the transceiver, (ii) the baseband circuit and (iii) the interface between them. The *transceiver* is a mixed-signal part of a wireless system whereas the other two parts are typically digital components. The *baseband* includes digital signal processors (DSPs) and it mostly operates as a part of a complex System on Chip (SoC). The *interface* is an important component mainly used for the communication of data and control-information between the transceiver and the baseband components. This control-information contains the commands to control the transceiver's chains (transmitter and receiver). In earlier days, the interfaces were implemented as an analog component of the wireless systems. However, the integration of a wireless system having an analog interface within a complex SoC is a challenging task. A feasible solution for this problem is to use the digital interfaces which are easier to integrate within the complex systems [1].

An *RF transceiver* typically comprises of a transmitter and a receiver. The transmitter modulates a digital signal, converts it to the analog domain, up-converts it to high frequencies, amplifies the signal and transmits it. The receiver works in the opposite direction of the transmitter. The receiver receives signals from the RF antenna, conditions the low-level signals, down-converts the high frequency signals to a lower intermediate frequency (IF), converts them into the digital domain and demodulates them [2]. The design and implementation of an RF transceiver block for wireless systems is a challenging task.

In recent times, several standards for the wireless communication (e.g. GSM, UMTS, LTE, WLAN) exist in the contemporary market. RF transceivers have to process the signals according to the specifications of these standards. These standards include different applications to provide different services to the customers. Modern communication standards support numerous high speed applications. Previously, RF transceivers were designed to support a single dedicated communication standard where the main emphasis of development was on cost effective solutions [2]. Nevertheless, plentiful services are presently available for the customers. Therefore, developing a transceiver with a single standard support is neither feasible nor beneficial. Hence, the development of a single RF transceiver supporting multiple wireless communication standards is a natural solution e.g., a configurable transceiver to support GSM, UMTS and LTE wireless standards. The main advantages of the multi-standard approach are: (i) small die area, (ii) small PCB area, (iii) less power consumption, (iv) less interconnections and (v) easier to handle. Re-usability of components is key goal of multi-standard solutions. Reusing the hardware components while maintaining a satisfactory performance can significantly reduce the cost (less manpower, less verification efforts) of a development. A single multi-mode transceiver is much more beneficial than several transceivers with single standard support [1, 2].

To support multiple communication standards, an RF transceiver should be reconfigurable so that it's transmitter and receiver chains can be configured to support a particular standard. Hence, the main emphasis in the development of multi-standard transceivers has been directed towards improving hardware reusability, reconfigurability, programmability and flexibility. The capability to support multiple communication standards makes the transceivers very complex. Therefore, a compound logic is needed to control and configure them. This control logic resides within the transceiver and configures it for a particular standard. It is also responsible to control and monitor the communication between the baseband and the transceiver over the interface. This complexity puts more demands on the power and area apprehensions of the transceivers. Further, this control logic itself has to be reconfigurable and flexible for being capable to support the multi-mode transceivers.

Aside the easier integration, the digital interfaces are indispensable to support the contemporary high-speed communication standards. An advance multi-mode transceiver necessitates a high-speed digital interface and a multi talented (reconfigurable, programmable, flexible, intelligent, fast and time accurate) control logic. DICE GmbH & Co KG, Austria is a daughter company of Infineon Technologies, Villach, Austria. It mainly focuses in the designing of innovative, leading-edge Application-Specific Integrated Circuits (ASICs) for the communications industry, particularly for the wireless products. An active group of the company is working for the development of high-speed interfaces and control-architectures for the multimode transceivers. The main objectives of these control architectures are to provide maximum flexibility, reusability and multi-standards (existing and future) support with low power and small die area. A flexible controlling logic is one of the main requirements on the architectures. Programmable digital hardware parts are used in the architectures to provide the maximum flexibility.

The project this report discusses is `the implementation and the verification of a CPU Subsystem`. This CPU Subsystem is a robust component of a control architecture being developed in the company. It operates as a central control unit of this architecture. Its foremost function is to configure the RF transceiver and the interface for a particular communication standard.

Typically costs are important factor in the industry. Therefore, we decided to use open source components to implement this Subsystem. Modern industry is also rapidly shifting towards the cheap open source solutions. The performance, area and power were also significant concerns while implementing the Subsystem.

The goal of the project was to implement a low-cost CPU Subsystem with a satisfactory performance and a comprehensive verification of its correctness.

This report has been structured in chapters for the simplicity and easiness. Brief information about the contents of chapters has given below.

Chapter 2 briefly outlines the environment in which the CPU Subsystem employs. Furthermore, this chapter sheds light on the operations of the CPU Subsystem. It outlines the specifications, constraints and objectives of this project. This chapter also portrays the methodology followed to accomplish these objectives. Further, it briefly discusses some basic concepts necessary to understand the implementation. However, the main emphasis of this chapter is on the development of the CPU Subsystem and its components. Finally, this chapter discusses the Software GNU Tool chain and the creation of the memory initialization file needed for the simulation of the Subsystem.

Chapter 3 gives a short introduction about the basic verification concepts. It discuses the different types of verification and evaluates the possible alternatives to verify the CPU Subsystem. Furthermore, this chapter also discusses the different technologies used for the verification of the CPU Subsystem.

Chapter 4 discusses the verification plans and demonstrates the test benches used to verify the memory system and the Sub-bus system. Further, this chapter describes the framework and development of the test bench used for the functional verification of the OpenRISC1200 core.

Chapter 5 thoroughly discusses the results obtained from the functional verification of the memory system, the Sub-bus system, the OpenRISC1200 core. It also discusses the results from the simulation of the CPU Subsystem. This chapter also outlines: (i) the errors found in the OpenRISC1200 core, and (ii) the various discrepancies found between the Golden Model and the OpenRISC1200 core.

Chapter 6 concludes the thesis, highlights the future work and suggests the possible extensions.

Chapter 2

System Environment and Organization

2.1 Introduction

This chapter focuses on the implementation and the simulation of the CPU Subsystem. Section 2.2 gives an overview about the environment of the CPU Subsystem and its operations in it. This section also outlines the objectives of this thesis and a systematic approach to accomplish them. Section 2.3 introduces the Wishbone interconnection standard which is essential to understand this project. This section also sheds light on the maximum throughput limitations of the Wishbone standard. Section 2.4 explains the implementation of the memory system. Section 2.5 describes the development of a triple-layer Sub-bus system. Section 2.6 introduces the OpenRISC1200 processor used as a central processing unit within the CPU Subsystem. This section briefly describes the main components of the processor and its pipeline architecture. Section 2.7 explains the maximum throughput limitations of the CPU Subsystem. Section 2.8 describes the OpenRISC1200 software tool chain and its installation. This section also discusses the generation of memory initialization file by using this tool chain. Finally, this section summarizes the integration of all components to implement the CPU Subsystem and the simulation of a test program on it.

2.2 System Description

2.2.1 Overview

As discussed earlier, implementing a digital interface is a practical solution to handle the high-speed communication between a complex multimode transceiver and a modern baseband-unit. A flexible and configurable control-architecture is required to control the multimode transceiver's chains (TX/RX), and to maneuver the communication between the transceiver and the baseband-unit. This control-architecture also configures the multimode transceivers to activate a particular standard. The CPU Subsystem operates in a control-architecture being developed to incorporate the multimode RF transceivers. This control-architecture is comprised of specialized adapters, a bus and distribution system, a multicore debug system and the CPU Subsystem. The actual control architecture is confidential and cannot be discussed in detail. However, some of its functionality related to the CPU Subsystem has been summarized below.

2.2.2 Advanced Control Architecture for Multimode RF Transceivers

The advanced control architecture administers the communication and configures the transceiver through macros. These macros are small messages sent by the baseband unit to the control architecture over the digital interface. These *macros* contain the parameters (e.g., channel number, band to be used) required to control the communication, and to configure the different units of the transceiver and the interface. Detail about the transceiver's units is beyond the scope of this report. The CPU Subsystem (Figure [2.1]) is responsible to decode the control macros to extract the control information and store the settings to the different units of the transceiver. The Main-bus system of the architecture is used to write the configuration macros to the memory (RAM) of the CPU subsystem. The central processing unit (CPU) fetches the macros from the RAM and decodes them to extract the control settings. This process is called the *high-level macro processing*. These settings are stored into the transceiver's units (RD/RX unit, RX/TX-PLL etc.) through the Main-bus system of the control architecture. This complete process is called the *pre-configuration*[1] of the transceiver. After the pre-configuration, a time-accurate strobe macro is used to start the *sequencing*[2] of the transceiver's units by using the pre-configured settings. The *strobe macros* have very tough real-time requirements. Hence, they are decoded in the hardware and directly sent to the units.

The macro decoding itself does not have very hard real-time requirements. The CPU Subsystem has a time-period to decode a macro and the decoding must be finished before that time. However, there are other real-time requirements on the chip e.g., the time accurate strobe-macros, the power up/start of the RF chain or the filter etc.

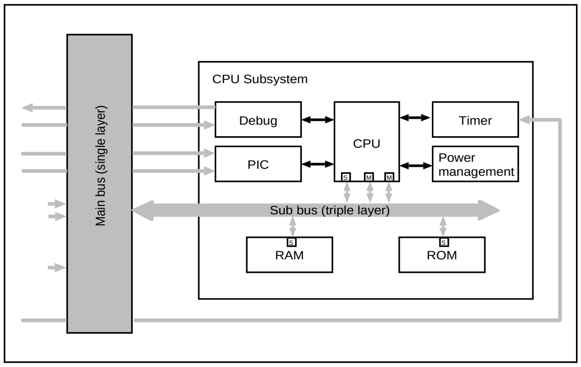


Figure 2.1 CPU subsystem within advanced control architecture.

[1] The macro's decoding.

[2] The configuration of the transceiver by copying the decoded setting to the hardware registers.

2.2.3 Structural Design of the CPU Subsystem

The CPU Subsystem shown in Figure [2.2] is consisted of a processor, a triple-layer Sub-bus, several interfaces and the memories. Details about these components will be given shortly.

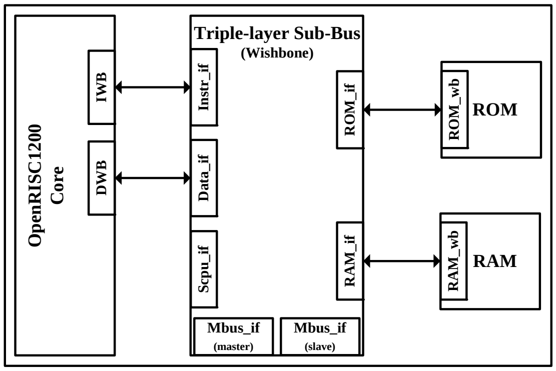


Figure 2.2 The CPU Subsystem.

This project can be divided into two major parts:

1. The implementation of the CPU Subsystem.
2. The verification of the CPU Subsystem and its components.

The implementation part includes the development of: (i) a Sub-bus system, (ii) a memory system and (iii) the interfaces between all the components. Since implementing a processor was beyond the scope of this project, a third party processor was needed which could function as a Central Processing Unit (CPU) in the subsystem. The processor was also required to have the utilities like power management, an interrupt controller, a hardware timer and the debugging facility. After surveying the open source market the OpenRISC1200 processor was a suitable choice [3]. The OpenRISC1200 (OR1200) is a 32-bit open source processor. It is ideally suited for the applications requiring higher performance than 16-bit processors while having low-cost and low power consumption advantage compared to 64-bit processors. Additionally, it also supports all the required utilities. The target applications of the OR1200 processor are: (i) medium and high performance networking, (ii) embedded and automotive systems, (iii) portable and wireless applications, and (iv) consumer electronics. The OR1200 core complies the Wishbone interconnection specifications to interact with the outer world. Therefore, all peripherals have to follow the Wishbone standard to interconnect with the OR1200 core. The *Wishbone* is an open source interconnection standard widely used in the industry. The development of a low cost SoC by using open source components is flourishing in the contemporary industry. The OR1200 core and the Wishbone interconnection standard have been discussed in subsequent sections.

2.2.4 Classification of the Project Objectives

The goal of the thesis is to implement a CPU Subsystem and its exhaustive functional verification. The most important part of any project is to define its scope and objectives in order to identify the requirements. Since this project has two divisions, the objectives can also be divided into two groups: (i) the implementation objectives and (ii) the verification objectives.

Implementation Objectives

The implementation includes the development of the CPU Subsystem using the OR1200 processor. Since the Subsystem has requirements of low power and small die area design, we decided not to use the caches (data and instruction) and the memory management units (data and instruction) of the OR1200 core. The implementation of the CPU Subsystem consisted of the following milestones.

- The implementation of a CPU Subsystem without using the caches and the memory-management of the OR1200 core with the characteristics of: (i) high-performance, (ii) low-power and (iii) small area.
- Achieve the single-cycle execution on the OR1200 core.
- The development of a triple-layer *Sub-bus system* with: (i) single-cycle access, (ii) fixed-priority based arbitration and (iii) the interfaces (Master/Slave) with the Wishbone standard. The implementation should be area and power efficient.
- The implementation of a *memory system* includes a Random Access Memory (RAM) and a Read Only Memory (ROM). The implementation of the Wishbone interfaces for both memories.
- The installation of the OR1200 GNU Toolchain (development toolkit). The generation of the executable-files and the memory initialization files (IHex) for the CPU Subsystem by using the development toolkit.
- The integration of the CPU Subsystem and its simulation by executing sample programs on it.

Verification Objectives

The verification of the CPU Subsystem includes the functional verification of: (i) the OR1200 core, (ii) the Sub-bus system and (iii) the memory system (ROM/RAM). It also includes the simulation-based verification of the CPU Subsystem. For the functional verification of the OR1200 core its Instruction Set Simulator (ISS)[3] was used as a golden model. The verification of the CPU Subsystem consisted of following milestones.

[3] **Or1ksim** is a generic OpenRISC1000 architectural simulator.

- The development of a Bus functional model for the functional verification of the memory system.
- The development of a test bench for the functional verification of the Sub-bus system.
- A simulation based verification of the CPU Subsystem.
- An exhaustive functional verification of the OR1200 core.
- The development of a SystemVerilog based wrapper around the OR1200 core in order to communicate with the core and to access its internal status.
- The development of a *golden model* for the functional verification of the OR1200 core.
  - Compile the ISS to a static library and develop the public interfaces to access it. These public interfaces are used by a system to interact with the ISS library.
  - Develop a SystemC wrapper around the ISS library to access its public interfaces. Provide the Direct Programming Interface (DPI) within the wrapper.
- The development of a reconfigurable and reusable test bench using the Open Verification Methodology (OVM).

Out of Scope

The tasks beyond the scope of this project are listed below:

- The verification of the OR1200 core includes only the troubleshooting of faults. It does not include the alteration in the OR1200 core to rectify them.
- The development of application programs for the macro decoding is not part of work.

2.3 Wishbone Interconnection Standard

2.3.1 Overview

The Wishbone interconnection is an open standard for the behavior of interfaces that describes the protocol to exchange data between the IP (intellectual property) cores. The Wishbone standard does not provide the implementation of the interconnects. The actual connections between the interfaces is up to the designers. The Wishbone interface protocol provides a reliable integration and easier reuse of IPs to develop the large SoCs. All components of the Subsystem have been implemented using the Wishbone interface specifications. Therefore, a brief introduction of the protocol is essential before moving to the implementation. More details can be found in the official Wishbone specification [4].

2.3.2 Wishbone Interface Specifications

The Wishbone interface specification can be used for a point-to-point connection between two cores as well as to implement some kind of bus to connect multiple cores. The Wishbone specification presents the MASTER and the SLAVE interfaces. The Master interface is connected to the Master component that originates a bus transaction. The Slave interface is connected to the component that responds in the bus transaction i.e., the Slave component. The Master and Slave interfaces can be connected to each other in different ways e.g., (i) a point-to-point connection, (ii) a shared bus, (iii) a crossbar bus or (iv) a data flow interconnection. In the Wishbone standard, a suffix (_I or _O) is attached to each signal's name to clearly identify its direction. This identifies whether a signal is an input to a core or an output from a core. For example, (ADR_I) is an input signal while (ADR_O) is an output signal.

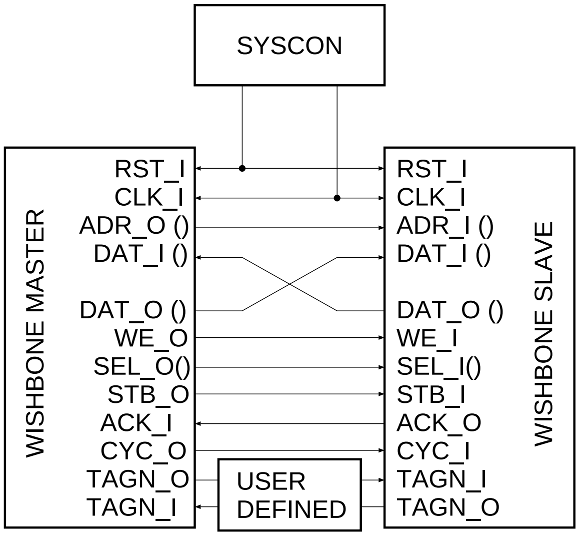


Figure 2.3 Point-to-point connection between the Wishbone Master and Slave.

Figure [2.3] shows a point-to-point connection between a Master and a Slave interface [5]. All timing diagrams (coming later) refer to this connection. Since the Wishbone signals use active-high logic (Rule 2.30), all signals in the CPU Subsystem will also obey this rule [4]. There are some optional signals in the Wishbone interface specification put into service depending on the implementation. These optional signals have not been discussed in this report. A short description about the Wishbone interface signals is given below.

Syscon Signals

`clk_o and rst_o`: The SYSCON module generates the *clock output* (clk_o) and the *reset output* (rst_o) signals for the Master and Slave interfaces. The clk_o signal is the system clock and the rst_o signal is the system reset for the interconnection implementation. The clk_o signal is connected to the clock input (clk_i) signal of the Master and Slave interfaces. The rst_o signal compels the Wishbone interfaces to restart and forces the internal state machines of the interconnection implementation to their initial states. The rst_o signal is connected to the reset input (rst_i) signal of the Master and Slave interfaces.

Signals Common to the Master and Slave Interface

- **`clk_i:`** The *clock input* signal is used to coordinate the internal activities of the Wishbone interconnection. All Wishbone output signals are registered at the rising edge of the clk_i signal. All Wishbone input signals must be stable before the rising edge of the clk_i signal.

- **`rst_i:`** When the *reset input* signal is asserted, the Wishbone interfaces are forced to restart and all internal state machines are switched to their initial states.

- **`dat_i and dat_o:`** The *data input* and *data output* arrays are used to pass binary data. The minimum granularity of the data size is 8-bit with the maximum size of 64-bit. The select output (sel_o) signal is used to select a particular byte of data in the data arrays. The signal dat_i is used to transfer the data from the Slave interface to the Master interface. The signal dat_o is used to transfer data from the Master interface to the Slave interface.

Master Interface Signals

- **`adr_o:`** The *address output* array is used to pass binary addresses from the Master interface to the Slave interface. The higher boundary of the array is specified by the address width of the core. The lower boundary of the array is restricted by the size of the data port and the granularity level.

- **`cyc_o:`** When a Master interface asserts the *cycle output* signal, it indicates that a valid bus transfer is in progress. The signal remains asserted as long as a consecutive bus transfer is valid. For example, in a burst transfer, it is asserted at the first data transfer and remains high until the last data transfer. In a multi-master development, this signal is used to request the arbiter for the bus-access (grant). After getting the grant from the arbiter, a Master interface holds the bus as long as the cyc_o signal is high.

- **`stb_o:`** When the *strobe output* signal is asserted, it indicates a valid data transfer cycle. It certifies that the other interface signals are valid. In response to every stb_o assertion, the Slave interface has to assert either the ack_i, the err_i or the rty_i signal.

- **`ack_i:`** The Slave interface asserts the *acknowledge input* signal in response to the stb_o. Each assertion of the ack_i signal indicates a normal termination of a bus transfer cycle.

- **`err_i:`** The assertion of the *error input* signal indicates an abnormal termination of a bus transfer cycle.

- **`rty_i:`** The assertion of the *retry input* signal indicates that the Slave interface is not ready to send or accept the data. Hence, this cycle should be retried.

- **`sel_o:`** The *select output* array points out where valid data bytes are positioned in the data array. In READ cycles, the sel_o signal indicates the position of valid data bytes in the data input array (dat_i). In WRITE cycle, the sel_o signal indicates the position of valid data bytes in the data output array (dat_o). The sel_o array boundaries depend on the size of the data arrays with bytelevel granularity.

  Example The select output array of 4-bit is needed to indicate the four bytes within a 32-bit data array. Each sel_o bit is corresponding to a particular byte in the dat_i or the dat_o array e.g., the sel_o(0) bit is for the dat_i(7 downto 0) byte, the sel_o(1) bit is for the dat_i(15 downto 8) byte, and so on.

  The Sub-bus system is a 32-bit implementation of the Wishbone standard. Therefore, we need a 4-bit select array to indicate the four bytes of the data arrays.

- **`we_o:`** The *write enable* signal shows whether the current bus cycle is WRITE or READ. The we_o signal is asserted for a WRITE bus cycle and stays low for a READ bus cycle.

Slave Interface Signals

The Slave interface signals have almost the same description like the Master interface signals with the opposite direction. Details about the Slave interface signals can be found in the official Wishbone specification [4].

Wishbone Classic Cycles

The Wishbone classic cycles define the general bus operations, the reset operation, the protocol and how data is organized during the bus transfer. The Master and Slave interfaces are connected through a number of signals. These signals are called the “Bus” which is used to exchange data between the Master and the Slave interfaces. The information on the Bus (address, data, control signals etc.) travels in the form of transactions.

The Wishbone specification uses a *handshake protocol* (Figure [2.4]) for the bus transfers [4]. A Master asserts the strobe signal (stb_o) when ready to transfer. The Slave asserts the terminating signal (ack_i/err_i/rty_i) in response. The terminating signal is sampled at every rising edge of the clock input (clk_i) signal. If the terminating signal is asserted, the strobe signal (stb_o) goes low.

Figure [2.5] shows a Wishbone classical single READ transfer cycle. A Wishbone classical single WRITE transfer cycle is shown in Figure [2.6]. A Wishbone classical bus cycle is initiated by asserting the strobe signal (stb_o) and the cycle signal (cyc_o). The Slave asserts the acknowledge signal (ack_i) for a normal termination. The Slave can insert any number of wait states (WSS) by keeping the acknowledge signal (ack_i) low. The write

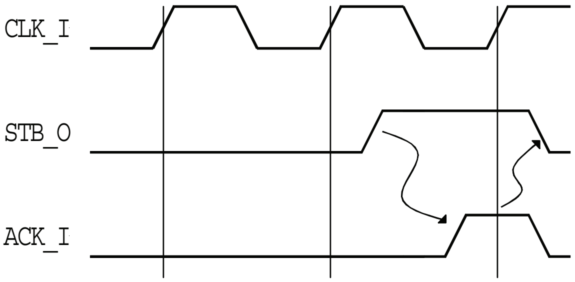


Figure 2.4 Wishbone handshaking protocol.

enable (we_o) signal identifies whether the current transfer cycle is a READ or a WRITE. Each Wishbone classical bus cycle needs to be properly terminated before starting a new one.

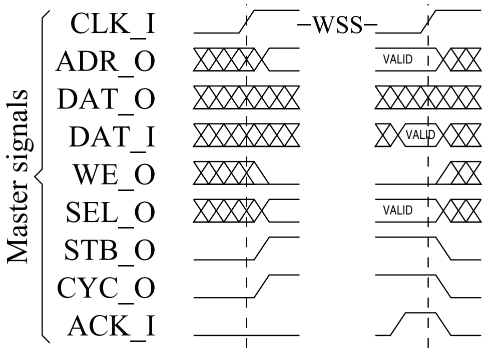


Figure 2.5 Wishbone classical single READ cycle [6].

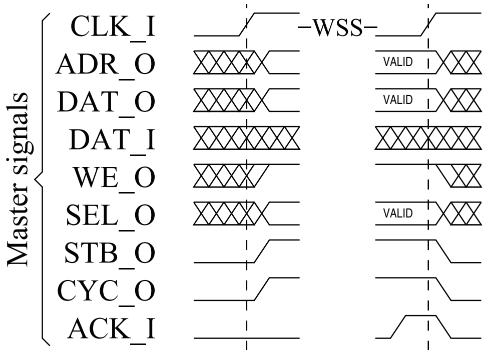


Figure 2.6 Wishbone classical single WRITE cycle [6].

The Wishbone classical bus cycle can be used to get block-style accesses, shown in Figure [2.7]. The cycle signal (cyc_o) remains asserted for the complete burst cycle. The strobe signal (stb_o) is used to control the transfer or to insert wait states. During a block (burst) access, the Master can either start a new transfer by asserting the strobe signal (stb_o) or can insert wait states by keeping it low. This is opposite to the single cycle access where the Slave can insert wait states. In block-style access, the arbitration has already been done and a Master has the ownership of the Slave through the interconnection. Thereby, the Slave is always ready to take a new request.

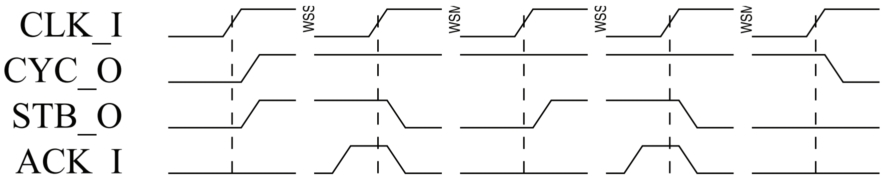


Figure 2.7 Wishbone classical block cycles [6].

The new Revision [B.3] of the Wishbone standard also supports incremental block transfers. However, this is beyond the scope of this report. Details can be found in the official Wishbone specification [4].

2.3.3 Maximum Throughput Constraints on the Wishbone

The maximum throughput from the Wishbone interconnection can be achieved by using asynchronous termination signals (ack_i, err_i, rty_i). But, the asynchronous termination signals result in a *combinatorial loop* [4] i.e., from the Master to the Slave and then from the Slave to the Master, through the INTERCON, Figure [2.8]. The INTERCON is a module that implements the internal logic of the interconnection.

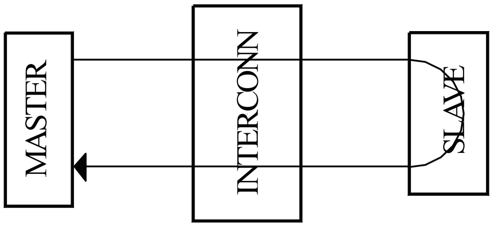


Figure 2.8 Wishbone asynchronous cycle termination path [4].

The simplest solution for this problem is to cut the combinatorial loop by using *synchronous termination* signals. In this case, the Slave has to de-assert its acknowledge signal low after each transfer. Because this approach adds a wait state after every transfer, each transfer can be completed in at least two clock cycles as shown in Figure [2.9]. Consequently, the maximum throughput with synchronous terminating signals is reduced by half because a new bus transfer can be initiated after every second clock cycle.

The *advanced synchronous cycle termination* is an optimum solution to overcome the decreased throughput in which the Slave knows in advance that it is again being addressed. Hence, the Slave keeps the acknowledge signal (ack_i) asserted rather than de-asserting it first and assert it again for the next transfer. The advanced synchronous cycle termination

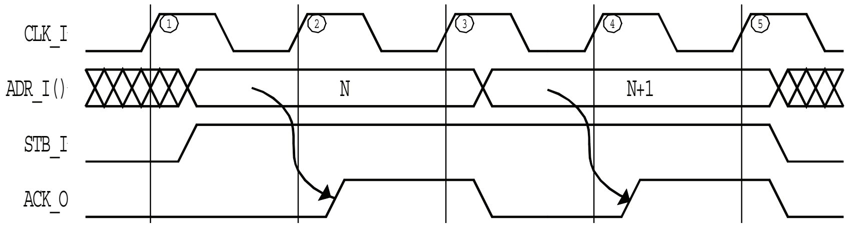


Figure 2.9 Wishbone classic synchronous cycle terminated burst [4].

is a beneficial approach for the large bursts. It needs “burst_length+1” cycles to complete a transfer if there are no wait states, Figure [2.10].

Example An 8-cycle burst needs nine cycles to complete the transfer while it needed sixteen clock cycles with the synchronous cycle termination. This is the throughput increase of 77%.

A single cycle burst is the worst case with the advanced synchronous cycle termination where its throughput is same as the synchronous cycle termination. It means both approaches are same for the single cycle bus transfer.

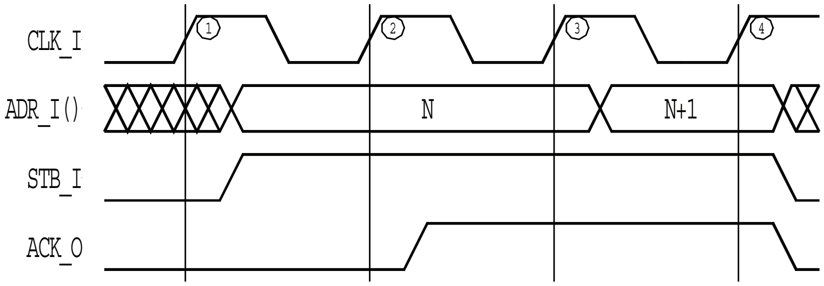


Figure 2.10 Wishbone advanced synchronous terminated burst [4].

We used a technique to increase the throughput for the single-cycle access. The idea behind is to use the asynchronous termination (ack_i, err_i, rty_i) for the WRITE requests and synchronous termination for the READ transfers. Since we do not need the registered data output from the Slave, an asynchronous acknowledgment (ack_i) for the WRITE request can be used. By using this technique, we need one clock cycle for each single cycle WRITE access instead of two clock cycles. However, the READ request still needs two clock cycles for each single cycle access.

To achieve the maximum possible throughput with the Wishbone specifications all components with Wishbone interfaces should follow the technique of “advanced synchronous cycle termination” with the asynchronous termination of WRITE requests and synchronous termination of READ requests. The achieved throughput for the WRITE requests will be a single cycle access. Every READ request will be finished in “burst_length+1” cycles.

2.4 Memory System of the CPU Subsystem

2.4.1 Overview

The *memory system*, used in the CPU Subsystem, consists of a “Read Only Memory” (ROM) and a “Random Access Memory” (RAM). As earlier discussed, the configuration macros are written into the RAM of the CPU Subsystem using the Main-bus of the control architecture. The OR1200 core fetches the macros from the RAM, decodes them, and stores the configuration settings to the different units of the transceiver. Hence, the core needs to run an application to decode the macros. The application is stored into the ROM of the CPU Subsystem. The core fetches the instructions from the ROM and executes them. Both memories are 32-bit word aligned.

2.4.2 Random Access Memory (RAM)

As we know, the OR1200 is a 32-bit processor with Wishbone interfaces for the data and the instruction. Therefore, we needed to implement a 32-bit wide data RAM and a Wishbone interface to access it. Further, the RAM had to be byte addressable in order to support the byte-level granularity of the data arrays. The RAM size and the address-lines are configurable to lessen its power consumption and the area.

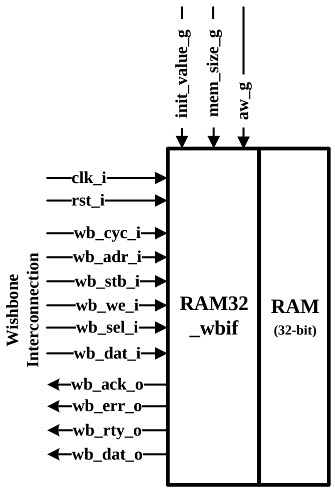


Figure 2.11 32-bit Random Access Memory (RAM).

Figure [2.11] shows the design of the implemented RAM having 32-bit wide data arrays. It is byte-addressable RAM with a Wishbone interface. The input “mem_size_g” determines the size of the RAM. The input “aw_g” controls the address-width needed to access that much size. The empty RAM is initialized with the input value “init_val_g“. As the RAM is a slave component, a Slave Wishbone interface has been implemented to access it. The Wishbone signals have been described before.

Write and Read Operations

We used synchronous termination (ack_i) for READ requests and asynchronous termination (ack_i) for WRITE requests to achieve the maximum throughput from the RAM while breaking the combinatorial loop [2.3.3].

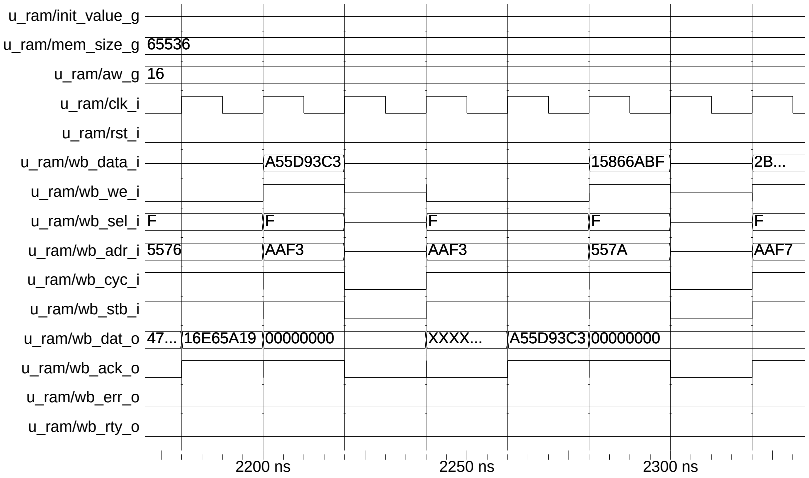


Figure 2.12 Sequential single transfer WRITE/READ.

Figure [2.12] shows a single transfer WRITE and READ operations for the RAM. A Master component initiates the WRITE request at time 2220 ns by asserting (i) the wb_cyc_i, (ii) the wb_stb_i and (iii) the wb_we_i signals. The wb_sel_i signal identifies the valid data bytes in the data arrays (wb_dat_i/wb_dat_o) depending on the operation (WRITE/READ). For instance, in this WRITE operation, all four bytes of 32-bit input data array (wb_dat_i) are valid to be written to the address (wb_adr_i). Since we are using asynchronous acknowledgment for WRITE requests, the wb_ack_i signal has been asserted at time 2220 ns without any delay. Hence, we get a single cycle bus transfer for the WRITE operation. Because of the synchronous acknowledgment for the READ requests at time 2260 ns the acknowledgment wb_ack_i is one clock cycle later (at time 2280 ns) than the request. Hence, the READ operation finishes in two clock cycles.

2.4.3 Read Only Memory (ROM)

The application to decode the configuration macros is stored in the ROM. The core fetches the instructions one by one from the ROM and executes them. Figure [2.13] shows the design of 32-bit ROM implemented with the Wishbone interface. The input “mem_size_g” determines the size of the ROM while the input “aw_g” controls the address width needed to access that much size.

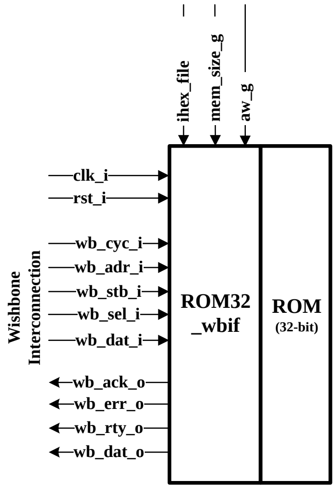


Figure 2.13 32-bit Read Only Memory (ROM).

Read Operation

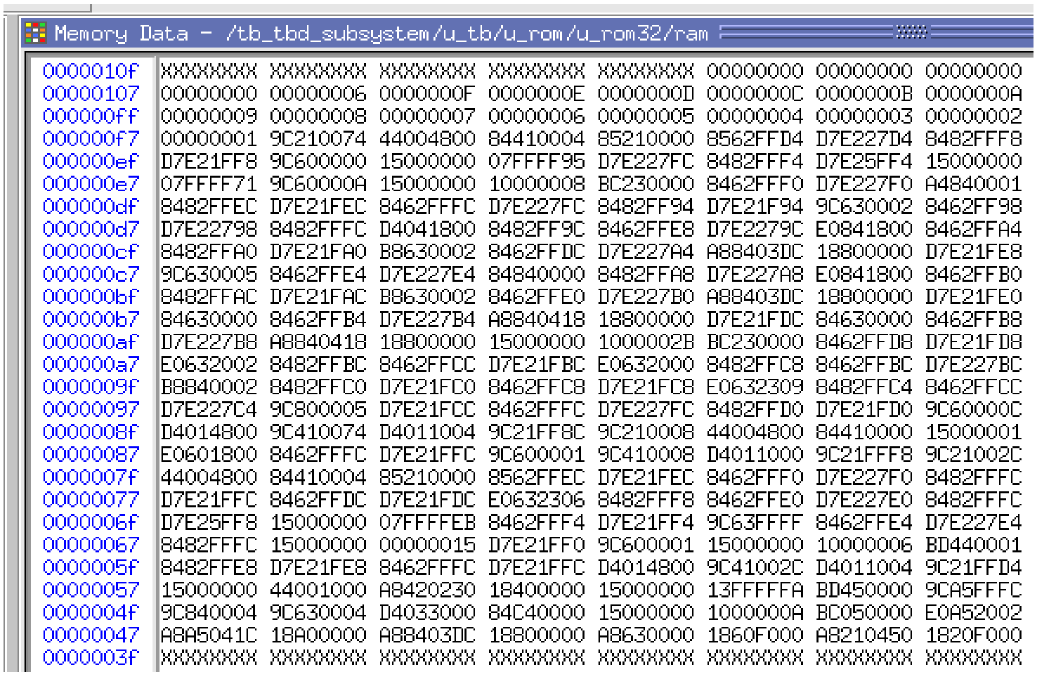
The READ request for the ROM also gets the synchronous acknowledgment. Each READ operation takes at least two clock cycles to finish. Hence, the maximum throughput for the READ operation is similar to the RAM i.e., two clock cycles for every READ access.

Memory Initialization

An application is compiled with the software toolchain to generate a "memory initialization file" for a particular processor. The memory initialization file contains the binary instructions of the application. The input "ihex_file" shown in Figure [2.13] is a reference to the memory initialization file going to be loaded into the ROM. The details about the software toolchain for the OR1200 processor and the generation of memory initialization file are explained in the Section (2.8.2). Here we give a short overview about loading the initialization file into the ROM. The loading of the initialization file has been handled inside the ROM.

After receiving the reset signal the OR1200 core fetches the first binary instruction from a default reset address i.e., 0x00000100. The initialization file must be loaded into the ROM starting from this address so that the first instruction of the applications is stored at the reset address of the core. Figure [2.14] shows a snapshot of an initialized ROM after loading the memory initialization file into it. The first binary instruction (0x1820F000) in the initialization file has been loaded at the address 0x00000040. The OR1200 core always generates a word aligned address[4] for the instruction fetch. Therefore, we have implemented a world aligned ROM. Hence, if the address (0x00000040) is two bits shifted left the resulting address will be the reset address of the OR1200 core (0x00000100).

[4] The last two bits of a word aligned address are zero. The last two bits address the four bytes inside a 32-bit word and there is no meaning of partially fetching a binary instruction.



```
Memory Data - /tb_tbd_subsystem/u_tb/u_rom/u_rom32/ram
0000010f XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX 00000000 00000000 00000000
00000107 00000000 00000006 0000000F 0000000E 0000000D 0000000C 0000000B 0000000A
000000ff 00000009 00000008 00000007 00000006 00000005 00000004 00000003 00000002
000000f7 00000001 9C210074 44004800 84410004 85210000 8562FFD4 D7E227D4 8482FFF8
000000ef D7E21FF8 9C600000 15000000 07FFFF95 D7E227FC 8482FFF4 D7E25FF4 15000000
000000e7 07FFFF71 9C60000A 15000000 10000008 BC230000 8462FFF0 D7E227F0 A4840001
000000df 8482FFEC D7E21FEC 8462FFFC D7E227FC 8482FF94 D7E21F94 9C630002 8462FF98
000000d7 D7E22798 8482FFFC D4041800 8482FF9C 8462FFE8 D7E2279C E0841800 8462FFA4
000000cf 8482FFA0 D7E21FA0 B8630002 8462FFDC D7E227A4 A88403DC 18800000 D7E21FE8
000000c7 9C630005 8462FFE4 D7E227E4 84840000 8482FFA8 D7E227A8 E0841800 8462FFB0
000000bf 8482FFAC D7E21FAC B8630002 8462FFE0 D7E227B0 A88403DC 18800000 D7E21FE0
000000b7 84630000 8462FFB4 D7E227B4 A8840418 18800000 D7E21FDC 84630000 8462FFB8
000000af D7E227B8 A8840418 18800000 15000000 1000002B BC230000 8462FFD8 D7E21FD8
000000a7 E0632002 8482FFBC 8462FFCC D7E21FBC E0632000 8482FFC8 8462FFBC D7E227BC
0000009f B8840002 8482FFC0 D7E21FC0 8462FFC8 D7E21FC8 E0632309 8482FFC4 8462FFCC
00000097 D7E227C4 9C800005 D7E21FCC 8462FFFC D7E227FC 8482FFD0 D7E21FD0 9C60000C
0000008f D4014800 9C410074 D4011004 9C21FF8C 9C210008 44004800 84410000 15000001
00000087 E0601800 8462FFFC D7E21FFC 9C600001 9C410008 D4011000 9C21FFF8 9C21002C
0000007f 44004800 84410004 85210000 8562FFEC D7E21FEC 8462FFF0 D7E227F0 8482FFFC
00000077 D7E21FFC 8462FFDC D7E21FDC E0632306 8482FFF8 8462FFE0 D7E227E0 8482FFFC
0000006f D7E25FF8 15000000 07FFFFEB 8462FFF4 D7E21FF4 9C63FFFF 8462FFE4 D7E227E4
00000067 8482FFFC 15000000 00000015 D7E21FF0 9C600001 15000000 10000006 BD440001
0000005f 8482FFE8 D7E21FE8 8462FFFC D7E21FFC D4014800 9C41002C D4011004 9C21FFD4
00000057 15000000 44001000 A8420230 18400000 15000000 13FFFFFA BD450000 9CA5FFFC
0000004f 9C840004 9C630004 D4033000 84C40000 15000000 1000000A BC050000 E0A52002
00000047 A8A5041C 18A00000 A88403DC 18800000 A8630000 1860F000 A8210450 1820F000
0000003f XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX
```

Figure 2.14 The ROM initialization.

2.5 Triple-layer Sub-bus System

2.5.1 Overview

Current VLSI technology has made it possible to incorporate an extensive number of transistors in a single die area. Therefore, modern systems can accommodate plenty of computational blocks (CPUs, DSPs, IPs) in a single chip to support the modern computation extensive applications. However, the interconnection between the increasing number of components in a SOC is a challenge. Since the traditional serial buses have scalability and bandwidth limitations, we need to find better interconnection methods for the systems having large number of components [7].

The advancement in modern SoCs needs hierarchy of buses in the system. Therefore, a multi layered bus architecture is a better solution to cope the limitations of the traditional buses [8]. Most of the modern buses are following the hierarchical structure to overcome the scalability limitations while providing a higher communication throughput. Moreover, modern hierarchical buses partition the communication domains into different groups of communication layers to achieve the bandwidth's requirement [7].

In this section, we are going to discuss a multi layered bus (also called Crossbar) implemented to connect the components of the CPU Subsystem. All interfaces of this bus comply the Wishbone interconnection standard.

2.5.2 Sub-bus Specifications

The CPU Subsystem includes three Master components and four Slave components. The Master components include: (i) the OR1200 instruction interface, (ii) the OR1200 data interface and (iii) the Main-bus master interface. The Slave components include: (i) the ROM, (ii) the RAM, (iii) the Main-bus slave interface and (iv) the OR1200 slave interface[5]. The development of a scalable and high performance bus architecture to interconnect these components was essential for the Subsystem.

The *Sub-bus* is a triple-layer bus architecture developed with three Master interfaces and four Slave interfaces. The Master components are connected to the Master interfaces. The Slave components are connected to the Slave interfaces of the Sub-bus system. All interfaces employ the Wishbone interconnection standard. The Sub-bus is a simple interconnection architecture which provides high data bandwidth and can support up to single cycle throughput.

The Sub-bus has been implemented by considering the low power and small area requirements. The configurable address lines for the Slave interfaces lessen the area and power consumption of the Sub-bus. Since the Sub-bus is a triple-layer implementation, the Master interfaces can access the Slave interfaces in parallel as long as there is no contention between the Master interfaces on a single Slave interface. If there is a contention, a priority based arbitration protocol has been implemented to serialize the ownership requests. The Sub-bus implements a distributed arbitration method i.e., each Slave interface has its own arbiter to serialize the contention on itself. Each Master interface has been assigned a fixed priority that influences the arbitration. A Master interface with higher priority takes the bus ownership by suspending the current bus transfer. The suspended Master interface resumes the transfer when the higher priority Master interface leaves the ownership. The “Sub-bus master interface” connected to the “Main-bus master interface” owns the highest priority because this interface has to deliver data and get free again and its request should not be delayed to have a predictable response. The Load/Store instructions access the OR1200 data interface during the executio. Hence, the Sub-bus master interface connected to the OR1200 data interface has a higher priority than the Sub-bus master interface connected to the OR1200 instruction interface. Otherwise, a higher priority instruction interface never allows the data interface to access the memories, particularly when the OR1200 core fetches new instruction every cycle.

2.5.3 Sub-bus Architecture

The architecture of the Sub-bus system has been partitioned into two figures for a clear illustration and to easily understand. Figure [2.15] shows the Sub-bus architecture and its internal connections from its Master interfaces to the Slave interfaces. Figure [2.16] shows the internal connections of the Sub-bus from the Slave interfaces to the Master interfaces.

According to the specifications, the Sub-bus contains three Master interfaces to connect three Master components and four Slave interfaces to connect the four Slaves components of the CPU Subsystem. All units of the Sub-bus are individually described below.

Configuration of the Sub-bus System

The configuration generics are used to configure the different units of the Sub-bus. The address-width selection generics are used to configure the address widths of the Masters interfaces and

[5] The original OR1200 implementation does not include this interface.

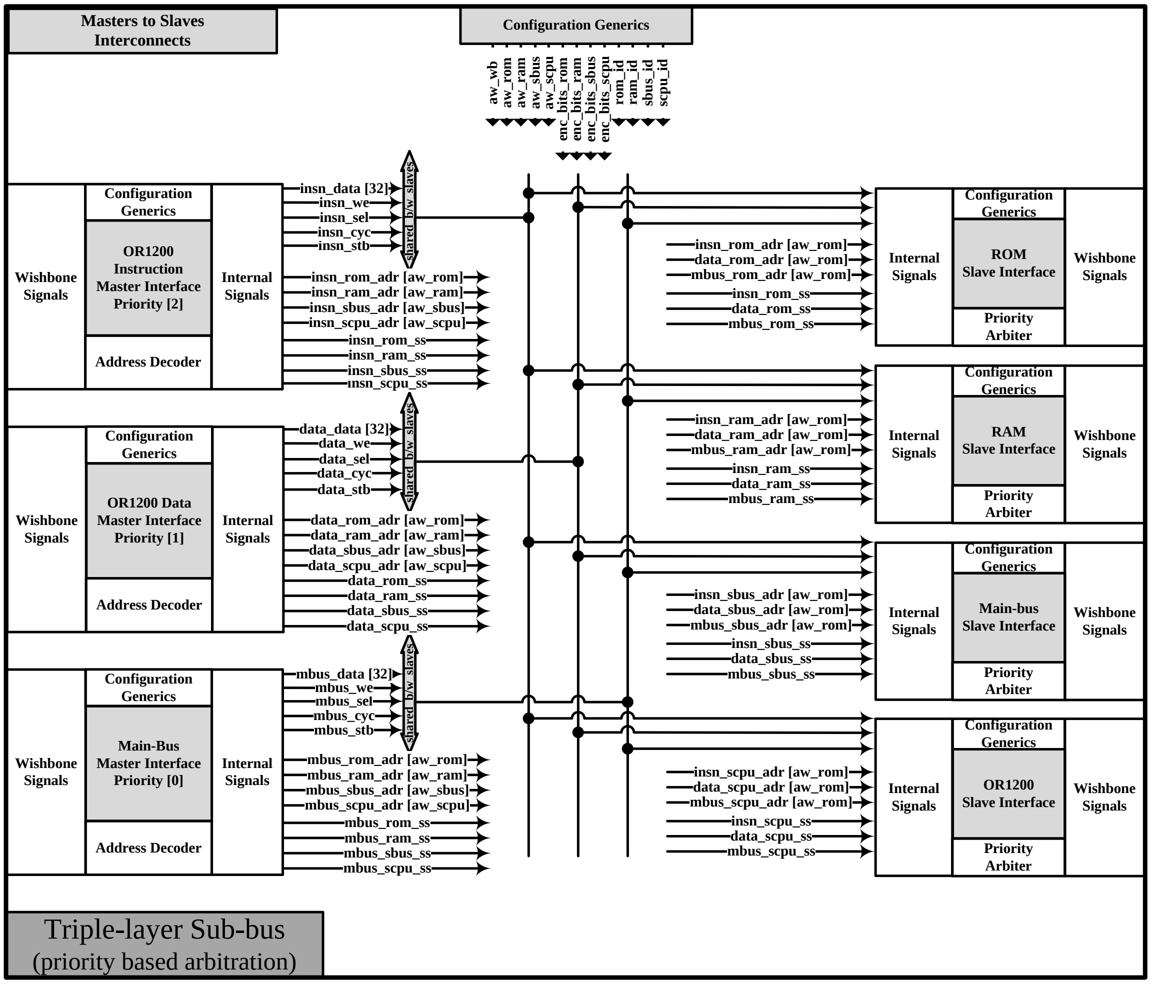


Figure 2.15 Sub-bus architecture (Master to Slave interfaces interconnects).

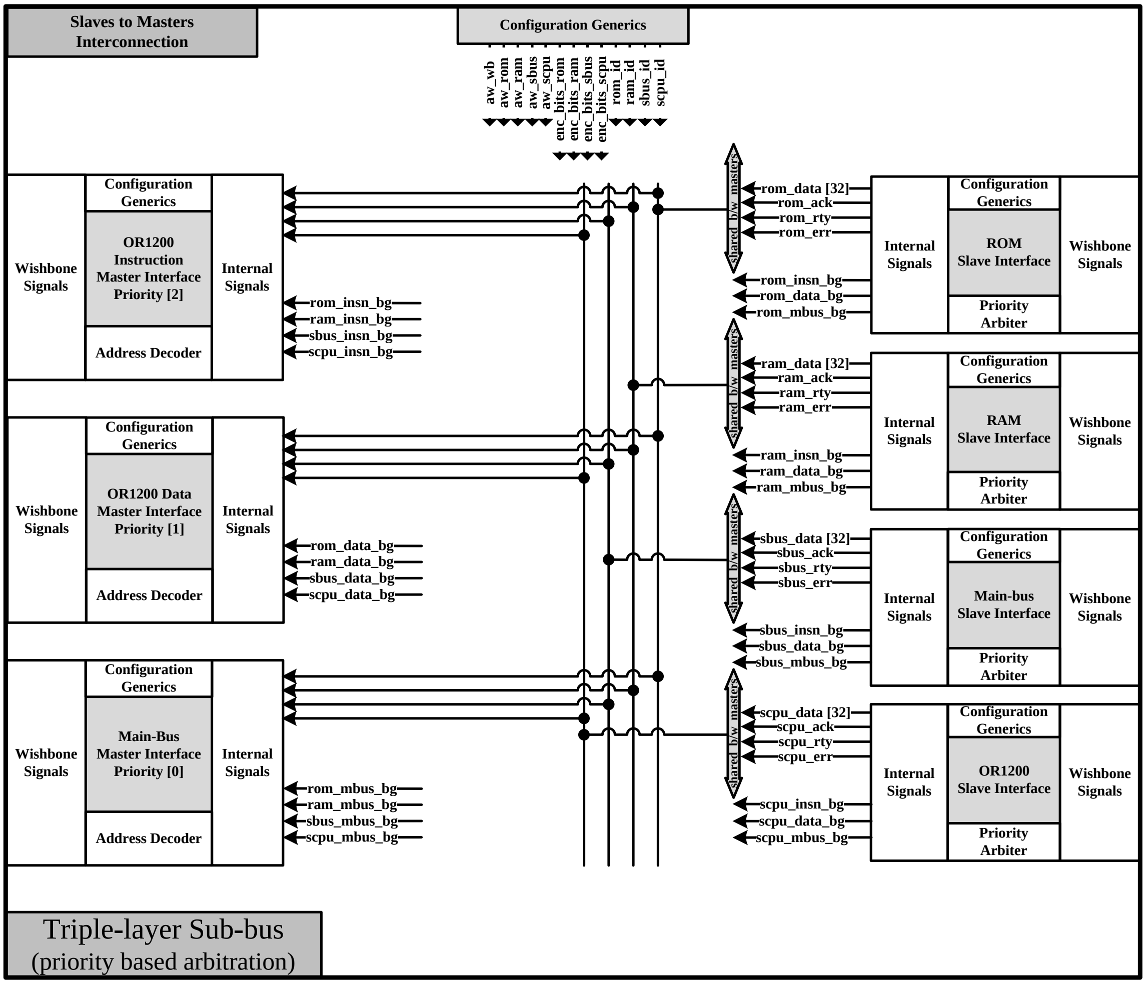


Figure 2.16 Sub-bus architecture (Slave to Master interfaces interconnects).

the Slave interfaces. An optimal width selection for the Slave interfaces significantly reduces the number of address lines inside the Sub-bus architecture which considerably cuts down the area and power consumption of the Sub-bus. The *address decoders* use the encoding-bits selection generics and the slave-identities to select a particular Slave interface. More details will be given while describing the address decoder.

Sub-bus Master Interface

Each Master interface includes (i) the configuration generics, (ii) the Wishbone signals, (iii) the internal signals and (iv) the address decoder. Each unit is individually described below.

Configuration Generics

These generics are used to configure each Sub-bus Master interface. These are used to adjust the width of the Wishbone address[6] and the widths of the internal address lines for each Sub-bus Slave interface.

Example The generic *aw_wb* adjusts the width of the address lines coming to the Master interface from the outer world. The generic *aw_rom* adjusts the width of the internal address lines of the Sub-bus going from the Master interface to the Slave Interface connected to the ROM.

Wishbone Signals

The Wishbone signals of a Master interface are used to connect a Master component to the Sub-bus. A component connected to the Sub-bus must have a Wishbone interface. If the external component is a Master, it must have a Master Wishbone interface to be connected to the Master interface of the Sub-bus. If the external component is a Slave, it must have a Slave Wishbone interface to be connected to the Slave interface of the Sub-bus.

Internal Signals

The internal signals are used for the point-to-point interconnection between the Master interfaces and the Slave interfaces of the Sub-bus. Some internal signals of a Master interface are shared among all the Slave interfaces while other are dedicated for a particular Slave interface.

Address Decoder

An *address decoder* shown in Figure [2.17] is a core component of a Sub-bus Master interface. It is used to decode the incoming address from the Master component. The incoming address includes a specific range of bits to identify the destination Slave interface for a particular request. The configuration generics distinguish the encoding bits in the address and the identity of a requested Slave interface. The decoder includes a comparator for each Sub-bus Slave interface to chop out the encoding bits and compare the value with the Slave interface's identity. The decoder selects a Slave interface if the encoding bits (in the address) hold its identity. For example, if the input address (ms_adr_i) holds the identity (rom_id) in its decoding bits (enc_bits_rom) the Sub-bus Slave interface connected to the ROM will be selected (rom_ss_o). The encoding-bits are always most significant bits (MSBs) of the address.

[6] The address coming from the Master component connected to this Master interface

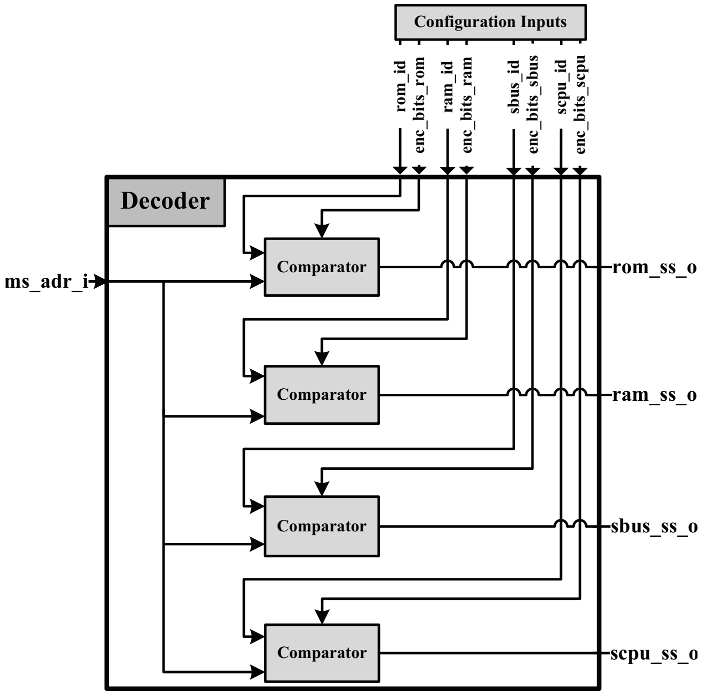


Figure 2.17 Address decoder.

Figure [2.18] shows a waveform of the decoder. The input address is 32-bit wide (aw_g) in which upper 16-bit (enc_bits_slave_g) hold the identities of the Sub-bus Slave interface. The decoder receives an address, compares the 16 MSBs with the Slave identities and asserts the slave-select signal corresponding to the Slave interface having that identity.

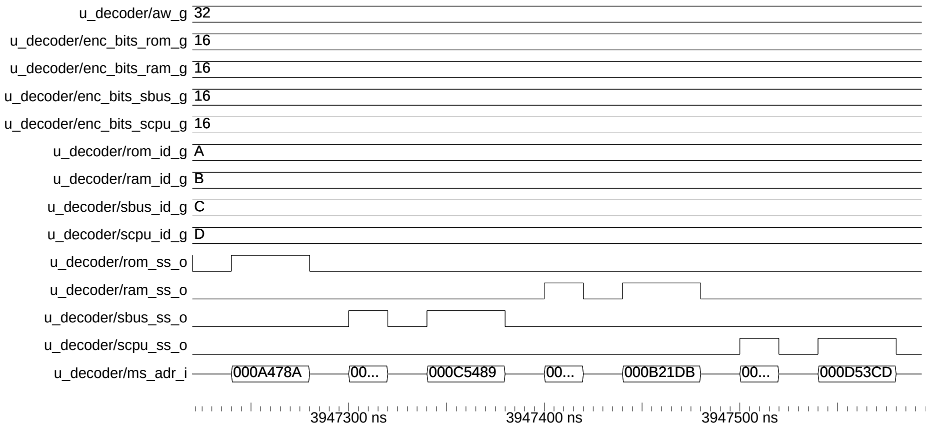


Figure 2.18 Address decoder waveform.

Sub-bus Slave Interface

Each *Sub-bus Slave interface* includes (i) a configuration generic, (ii) the Wishbone signals, (iii) internal signals, and (iv) an arbiter. Each unit has been individually described below.

Configuration Generic
This value is used to configure each Sub-bus Salve interface to adjust the width of the address going to the connected Slave component. It also adjusts the widths of internal address-lines coming from the Sub-bus Master interfaces.

Wishbone Signals
The Wishbone signals are used to connect a Slave component having a Slave Wishbone interface.

Internal Signals
The internal signals are used for the point-to-point interconnection between the Sub-bus Slave interfaces and the Sub-bus Master interface. Some of the internal signals are shared among all Master interfaces while others are dedicated for a particular Master interface.

Arbiter

There is no centralized entity to control the accesses to the Sub-bus Slave interfaces. Each Slave interface itself grants the access requests and implements a fixed priority arbitration protocol (pre-emptive)[7] to handle the contention on its ownership. Each Slave interface contains an arbiter inside (Figure [2.19]) which implements the arbitration protocol and grants the accesses. A Master interface requests the ownership of a Slave interface by asserting its slave select signal (ms_ss_i) for that Slave interface and also the cycle input signal (ms_cyc_i) to indicate the valid bus transfer.

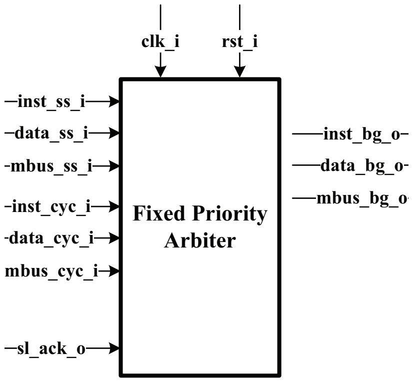


Figure 2.19 Fixed priority arbiter.

Figure [2.20] illustrates the implemented arbitration protocol. A Master interface requests the ownership of a Slave interface by asserting its cycle input (ms_cyc_i) signal and the Slave selec-

[7] Low priority Master cannot block the request of a high priority Master.

tion signal (sl_ss_i) for the requested Slave interface. The Slave interface gives the grant if idle[8]. Otherwise, the requesting Master interface has to compete with the Master interface having the ownership of the Slave interface. A Master interface having the higher priority wins the ownership in the contention. The suspended Master interface waits until the Slave interface is free.

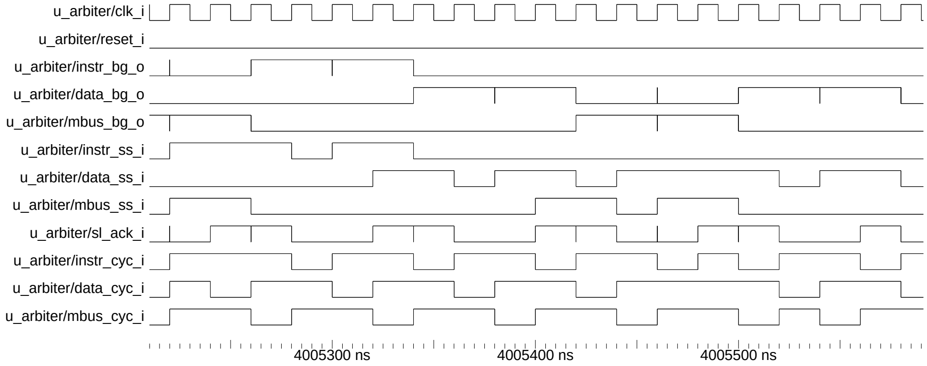


Figure 2.20 Fixed priority based arbitration.

2.5.4 Fundamental Characteristics of Sub-bus

The internal interconnections of the Sub-bus have been divided into shared and dedicated signals, shown in Figure [2.15]. The shared signals of a Master interface are visible amongst all the Slave interfaces while the dedicated signals correspond to a particular Slave interface. When a Master component requests for a bus transaction, the Sub-bus Master interface sets the shared signals and sends the request over the dedicated signals to the requested Sub-bus Slave interface. That Slave interface arbitrates the access request and gives the grant. The shared signals only qualify for the Slave interface which grants the request while other Slave interfaces simply ignore them. The qualified signals are propagated to the Slave component over the Wishbone signals. The connected Slave component sees the request (READ/WRITE) and responds to it accordingly. Despite that the shared signals of a Slave interface are visible amongst all the Master interfaces (Figure [2.16]), only a granted Master interface qualifies these shared signals and sends them to the request initiator.

The triple-layer Sub-bus is very easy to use and simple to handle. Its re-configurability provides the flexibility to employ it according to the system's requirements and to maneuver its area and power utilization. The Sub-bus supports up to single cycle throughput with zero arbitration time when the bus is idle, otherwise one clock cycle at maximum. The Sub-bus also supports block transfer of any size with its maximum throughput.

Even though, the Sub-bus implementation can support single cycle throughput, since it will be connected to the components having Wishbone interfaces, its maximum throughput will be constrained by the Wishbone standard's limitations on the maximum throughput (see Section 2.3.3).

[8] Free, no grant to any Master-interface.

2.6 The OpenRISC1200 Processor

2.6.1 General Description

The OR1200 processor (Figure [2.21]) implements the central processing unit of the CPU Subsystem. The OR1200 is a 32-bit scalar RISC soft-processor with Harvard memory architecture. It has a single-issue 5-stage integer pipeline, virtual memory support and a MAC unit for basic DSP operations. The OR1200 delivers a sustained throughput and supports single-cycle execution for most of its instructions. Its intended target applications include: (i) embedded applications, (ii) Internet and networking applications, (iii) telecoms and wireless applications, and (iv) automotive applications. The OR1200 is a 32-bit implementation of the OpenRISC1000 architecture. The OpenRISC1000 is a latest architecture designed for 32-bit and 64-bit RISC/DSP processors. The emphasis of the architecture is on (i) simplicity, (ii) low power, (iii) high scalability and (iv) high performance of the processors. The OR1200 supports big-endian byte ordering. The OR1200 is an open source processor under the LGPL license. It is developed and being managed by the OpenCores organization. Its verilog model is freely available at OpenCores.

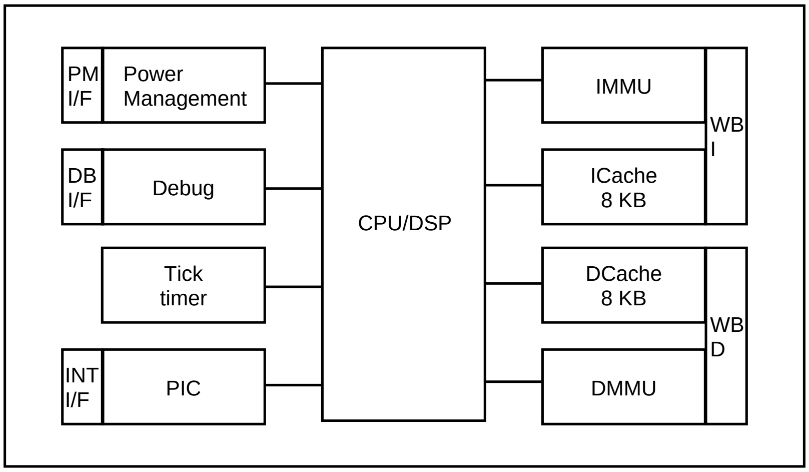


Figure 2.21 The OpenRISC1200 processor.

2.6.2 Performance

The OR1200 supports the system frequency of 250 MHz under worst-case scenario at 0.18 *µm* 6 LM fabrication process. It can execute 250 dhrystone millions of instructions per second (DMIPS) at 250 MHz, the worst-case. It can execute 250 MMAC operations at 250 MHz under worst-case conditions. However, under normal conditions, the OR1200 should provide over 300 dhrystone 2.1 MIPS at 300 MHz and 300 DSP MAC 32x32 operations. This performance is at least 20% more than any other competitor in this class [9].

The power estimation of the OR1200 is less than 1 Watt at full-throttle while less than 500 mW at half-throttle with 250 MHz clock at a 0.18 *µm* process. Its die-area without cache memories is less than 0.5 *Sqmm* at 0.18 *µm* 6 LM fabrication process. A default configuration of the OR1200 has about 1M transistors [9, 10].

2.6.3 The OpenRISC1200 Architecture

Figure [2.22] shows the architecture of the OR1200 core including the central processing unit (CPU/DSP), caches (IC/DC), memory management units (IMMU/DMMU) and other utility components. Many components are optional to implement. A component is implemented and controlled through its corresponding special purpose registers and unit dependent registers. Some important units of the processor will be discussed below. More details can be found in the official documents of the OR1200 core and in its RTL implementation [9–12].

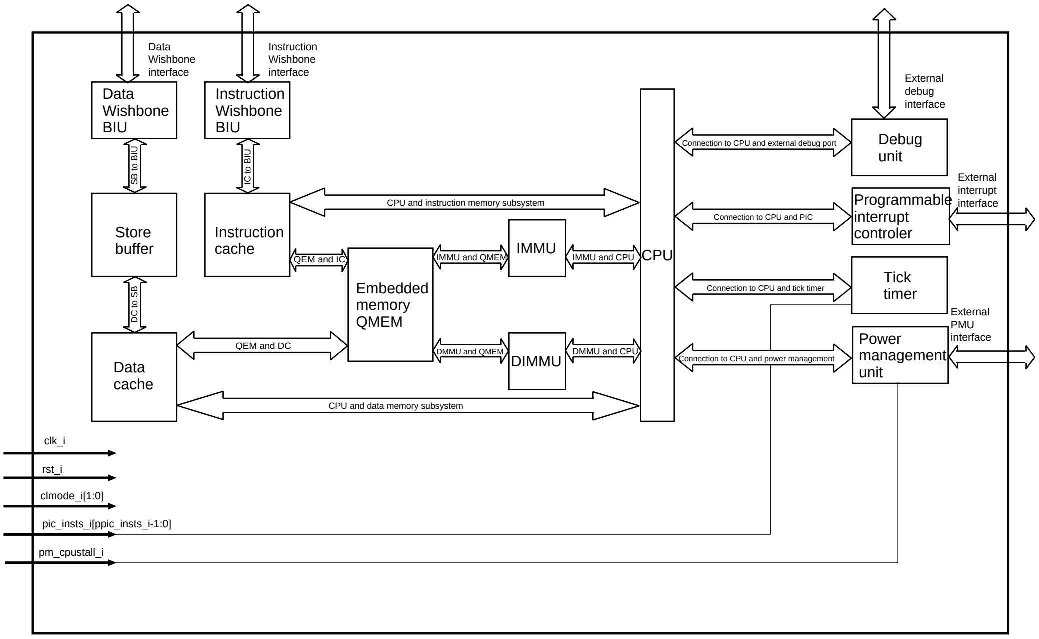


Figure 2.22 The OpenRISC1200 architecture.

Caches and Memory Management

As the OR1200 implements a Harvard memory architecture, it has a Level-1 separate data cache (DC) and a Level-1 instruction cache (IC). It also has a separate data memory management unit (DMMU) and an instruction memory management unit (IMMU). Both caches (IC/DC) are N-way set associative and physically tagged. The DC is scalable from 1 Kbytes to 8 Kbytes. The IC is scalable from 512 Bytes to 8 Kbytes. A default implementation of the processor has direct-mapped 8 Kbytes caches (IC/DC) with 16 Bytes line size (8-byte line size is also supported). The DC operates in write-through mode (only supported). The OR1200 supports an implementation without having caches. However, it affects the throughput of the processor.

The memory management units (IMMU/DMMU) enable the virtual memory support and consist of hash-based translation-lookaside buffers (TLBs). Both translation-lookaside buffers (instruction/data) are direct-mapped with page size of 8 Kbytes. Both TLBs (ITLB/DTLB) are individually scalable from 16 to 128 entries per each way. Both MMUs have a linear address-space with a 32 bits virtual address and a physical address from 24 to 32 bits. A default implementation of the OR1200 core have direct-mapped translation-lookaside buffers (DTLBs/ITLBs) of 64 entries per each way with a fixed page size of 8 Kbytes.

Debug Unit

The OR1200 optionally provides a debug unit to support basic debugging features. The debug unit does not support watch-points, breakpoints and program flow control registers. However, the OR1200 provides a development interface to connect a more advance additional debugging facility.

Tick Timer

The OR1200 core provides a high-resolution hardware timer. The timer is clocked with the RISC clock. It is used by operating systems for task scheduling and precise time measurement. The maximum range of the timer is 2^{32} clock-cycles. The maximum time period between the interrupts is 2^{28} clock-cycles. The timer provides a mask-able interrupt and offers different run-modes: (i) single run, (ii) continuous run or (iii) restart-able.

Programmable Interrupt Handler (PIC)

The OR1200 has an interrupt-controller that receives external interrupts through its interrupt interface and sends them to the Central Processing Unit (CPU). The interrupt controller supports two non-maskable interrupts and 30 maskable interrupts, with two priority levels.

Power Management Unit (PMU)

The OR1200 core has a sophisticated power management unit to control its power consuming functions. The power management unit offers different power down modes: (i) *sleep mode*, (ii) *doze mode*, (iii) *slow and idle mode*, and (iv) *CPU stalling*. The clock frequency of the processor is software-controlled in the slow-and-idle mode. The power consumption can be reduced from 2x to 10x in this mode. In the doze mode, a software running on the core is suspended and the clocks to all RISC internal units are disabled except to the real-time clock and internal timer. Other on-chip blocks can work normally in the doze mode. The power consumption in this mode can be reduced up to 100x. The processor leaves this mode and enters the normal mode when a pending interrupt from an external peripheral occurs. In sleep mode, the real-time clock and periodic timer are the only RISC internal modules that are activated. The OR1200 leaves the sleep mode and enters the normal mode when a pending interrupt from these modules occurs. The power consumption can be reduces up to 200x. The power management unit does not support dynamic clock gating. A more advanced power management utility can be connected to the OR1200 core using the power management interface. The implementation of a power management unit in the core is optional and depends on the designer's requirements.

Quick Embedded Memory (QMEM)

The QMEM implements some time critical functions to achieve a fast and predictable behavior of (i) the soft floating-point unit, (ii) the context switching, (iii) the exception handlers and (iv) the stacks. Since both caches (IC/DC) share the QMEM, the instruction fetch operations affects the performance of the Load/Store operations. The effective throughput of the instruction fetch operation is one instruction per clock cycle. Whereas, data accesses have different effective throughputs for READ and WRITE depending upon the instruction fetch accesses. In absence of an instruction fetch, READ data takes two clock cycles per access while WRITE data takes one clock cycle per access. The QMEM is an optional unit. Its implementation increases the OR1200 size and makes it slower. The QMEM sits behind the memory management units (IMMU/DMMU) so all addresses are physical. Since the IC and the DC are sitting behind the QMEM, the whole design timing might be worse with the QMEM implemented [12].

Store Buffer (SB)

The Store Buffer (SB) is optionally implemented to improve the performance by buffering the CPU's store accesses. The SB is very important for the function prologues because the DC can only work in write-through mode and all stores would have to complete the external write-back writes to the memory. The SB is implemented between the DC and the data Wishbone interface of the OR1200 core. All store accesses are stored into the SB and immediately completed by the CPU. However, the actual external writes are performed later. Hence, the SB masks all data-bus errors related to Stores, but data-bus errors related to Loads are delivered normally. Since the OR1200 core implements a strict-memory model[9], all pending CPU loads will wait until the store buffer is empty [11]. The SB makes the OR1200 implementation bigger, depending upon the number of entries in the SB's FIFO [12].

System Interface

The system interface is used to connect the system signals to the OR1200. The interface is comprised of (i) the system clock, (ii) the system reset and (iii) other system level signals.

Wishbone Interfaces

The OR1200 core is connected to external peripherals and external memories through two interfaces, the data interface (DWB) and the instruction interface (IWB). Both interfaces support only 32 bits bus width and comply with the Wishbone specification Revision [B]. The instruction Wishbone interface (IWB) is used to fetch instructions from the instruction memory (IM). The data Wishbone interface (DWB) is used for the data transfer between data memory (DM) and the processor.

2.6.4 Central Processing Unit (CPU/DSP)

The verification of the OR1200 core was the most important objective of the thesis. Therefore, we must have a detailed information about the internal architecture of the core. In addition to the OR1200's specification, the central processing unit (CPU) is the most important component to understand in detail. The CPU implements the instruction's execution pipeline architecture of the core. We must have a comprehensive knowledge about (i) the architecture of the CPU, (ii) its pipeline execution, (iii) its timing, (iv) register set and (v) the organization of inside units plus their co-ordination during operation. More details about the CPU architecture can be found in the official documents of the OR1200 processor and in its RTL implementation [3, 9–12].

[9] All Load/Store operations are performed in order.

CPU Architecture

The OR1200 CPU/DSP is shown in Figure [2.23]. It implements the 32-bit part of the OpenRISC1000 architecture. A brief introduction about the units of the CPU is presented below. Details about its pipeline architecture will be given in the subsequent section.

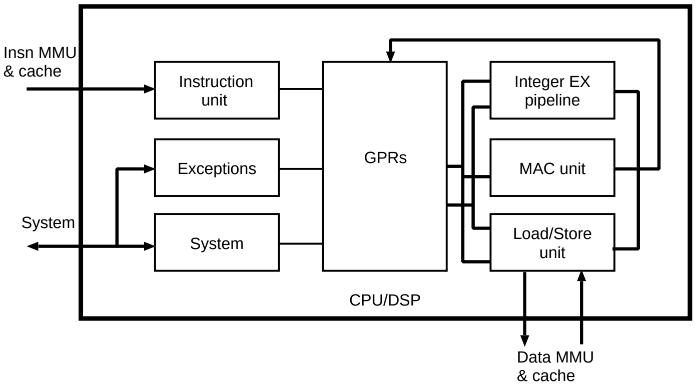


Figure 2.23 Central Processing Unit (CPU/DSP).

Instruction Unit

The *instruction unit* implements the basic instruction pipeline of the OR1200 core. The instruction unit fetches instructions from the memory system and dispatches them to the available execution units (LSU, ALU, MAC unit) while ensuring a precise exception model. The instruction unit also executes the branches and jump instructions. It implements the “OpenRISC Basic Instruction Set” (ORBIS32) of the OpenRISC1000 architecture. The OpenRISC1000 architecture defines five instruction’s formats and two addressing modes: (i) register indirect with displacement and (ii) PC relative. The ORBIS32 instruction set class has 32-bit wide instructions aligned on 32-bit boundaries in the memory and operates on 32-bit and 64-bit data. The instruction set also supports eight custom instructions implemented on demand. An additional co-processor can be attached to the core. All branch/jump instructions are followed by a delay slot while Return-from-Exception (RFE) does not have a delay slot. Most of the OR1200 instructions execute in a single cycle. The instruction multiply takes three clock cycles and the instruction divide takes 32 clock cycles to execute. Both instructions are implemented in the MAC unit. The MAC unit will be discussed later.

Register Set

The OR1200 core implements thirty two 32 bits general purpose registers (GPRs). The GPRs have been implemented as a dual port synchronous memory with 32 words of 32 bits each. The OR1200 contains 32 groups of special purpose registers (SPRs). It also implements unit-dependent registers. Some important SPRs for the verification of the OR1200 are briefly discussed below.

Supervision Register

The supervision register (SR) is a special purpose register which shows the state of the OR1200 processor.

Exception Supervision Registers

There are sixteen exception supervision registers (ESR0-ESR15). It is up to the designer how many ESRs he needs in the implementation. The SR is copied into the ESR register when there is exception. If an implementation has only a single ESR, the exception handler routine has to save it before re-enabling the exception recognition in the SR.

Program Counter Register

The program counter (PC) register stores the address of the next instruction to be executed.

Exception Program Counter Register

The exception program counter (EPCR) is a special purpose register which stores the copy of the PC register when there is an exception. It stores the address of the instruction interrupted by the exception.

Exception Effective Address Registers

There are sixteen exception effective address registers (EEAR0-EEAR15). It is up to the designer how many EEARs he wants in the implementation. When there is exception, the EEAR saves the effective address (EA) generated by the faulting instruction. If an implementation has only one EEAR, the exception handler routine has to save it before re-enabling the exception recognition in the SR.

Load Store Unit (LSU)

The Load/Store unit (LSU) is responsible for transferring data between the GPRs and the internal data Bus of the CPU. The LSU has been implemented as an independent unit in the OR1200 core so that stalls in the memory system only affect master pipeline if there is a data dependency. All Load/Store requests are aligned on 32 bit boundaries. The LSU can execute one load instruction every two clock cycles (assuming a hit in the DC). The execution of store instructions takes one clock cycle (assuming a hit in the DC).

Arithmetic and Logic Unit (ALU)

The ALU is a pipeline unit that implements: (i) the arithmetic instructions, (ii) the compare instructions, (iii) the logical instructions and (iv) the rotate and shift instructions. Most of the ALU instructions execute in a single cycle.

Multiply Accumulate Unit (MAC)

A fully pipelined multiply-accumulate (MAC) unit executes basic DSP operations and MAC instructions. The MAC unit has the ability to accept new MAC operations every clock cycle. It optionally implements multiply and divide instructions. The multiplier in the MAC unit is 32x32 bits. However, the multiply instructions only use the lower 32-bit of the result. The MAC instruction has 32-bit operands and a 48-bit accumulator.

System Unit

It connects all the CPU signals to the system signals except those which are connected through the Wishbone interfaces (IWB/DWB). The system unit also implements the system SPRs e.g., SR.

Exception Unit

The exception unit handles the generated exception in the OR1200 core. These exceptions include: (i) the system calls, (ii) the internal exceptions, (iii) the memory access conditions (e.g., unaligned access/invalid address), (iv) interrupt requests and (v) the internal errors (e.g., unimplemented instructions). Each exception has a defined offset address. The control is transferred to this address if there is an exception.

Example The control is transfered to the address 0x00000600 for an unaligned memory access exception and to the address 0x00000700 for an illegal instruction exception.

2.6.5 OpenRISC1200 Instruction Pipeline

The pipeline architecture is the most important part to understand while verifying a pipelined processor. Pipelining is a technique to divide the instruction's execution into a number of independent steps to improve the throughput of a processor. These independent steps are called pipeline stages. Each pipeline stage ends up in a storage (pipeline registers) of its execution so that the subsequent stages can use the result. The OR1200 core implements a 5-stage integer pipeline. The pipeline stages are shown in Figure [2.24].

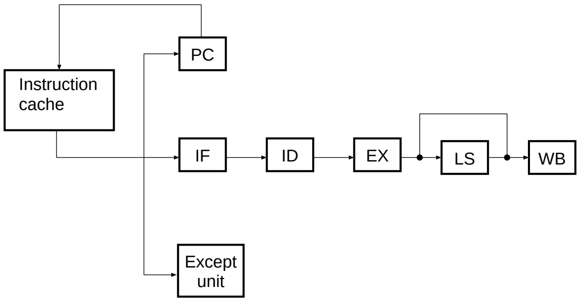


Figure 2.24 The OpenRISC1200 pipeline stages [13].

The GenPC is not an independent pipeline-stage in the OR1200 core. It works in parallel to the Instruction Fetch (IF) i.e., the first pipeline stage. The GenPC generates the next program counter (PC) and sends it to the IMMU (if implemented) to calculate the physical address. If there is no IMMU and IC in the implementation, the PC address is bypassed and directly goes to the instruction memory (IM) via the IWB. The IF stage fetches new instruction from the IC. If no IC is implemented, it fetches the instructions directly from the IM. In the Instruction Decode (ID) stage, the new instruction is decoded to identify (i) the basic type of the instruction, (ii) the instruction's Opcode, (iii) the addresses of the operands to be fetched from the register file, (iv) calculate the immediate if the instruction is with immediate, (v) the address of the data to be loaded/stored, and (vi) the execution unit (LSU/ALU/MAC) for the instruction. The next pipeline stage is the Execution stage (EX). After providing the required input data, an instruction is dispatched to its execution unit to execute. Since the LSU is implemented as an independent unit to not affect the master pipeline of the OR1200, all instructions (except load/store) bypass the Load/Store (LS) pipeline stage. In this stage, the LSU transfers data between the GPRs and the DC (if existed, otherwise, the data memory (DM)). In the Write Back (WB) pipeline stage, the result of an instruction's execution is written back to the register file (RF).

The OR1200 core implements an exception-model parallel to the pipeline-model to handle exceptions in a controlled way. Each pipeline stage (except IF) can generate an exception. The exception-unit implements the exception-model. It inputs the exception signals from different pipeline stages and generates the corresponding exception-vectors to calculate the new program counter (PC). If an instruction in the pipeline results in an exception the next PC will be the exception-vector corresponding to that exception (see Subsection 2.6.4).

We will thoroughly discuss the pipeline execution in next few sections.

Register Level Outlook

Figure [2.25] shows the register level view of the OR1200 pipeline architecture. The register level understanding is very important and a concrete starting point to verify a digital design. Here we are going to discuss 5 important registers of the pipeline architecture. However, many registers are involved in the control-logic and the data-path.

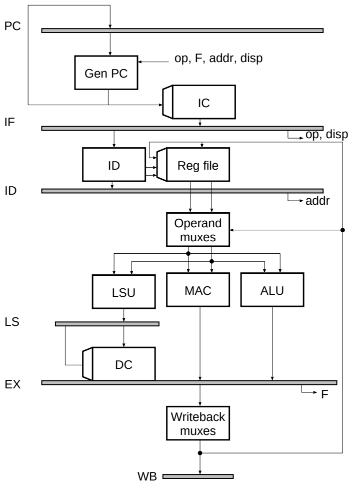


Figure 2.25 Registers abstraction of the OR1200 pipeline [14].

The GenPC calculates the next program counter (PC) and sends the address to the IC/IM. A new instruction from IC/IM is fetched and stored into the `if_insn` register. Since the fields in the OR1200 instructions are fixed, the instruction in the `if_insn` is translated to get the addresses of the source GPRs. The addresses are fed into the register file (RF)[10] to get the source operands (GPRs). The instruction in the `if_insn` moves to the `id_insn` register and a new instruction is fetched into the `if_insn` register.

The fields of the instruction in the ID stage are translated to the control-signals for the next pipeline stages (EX and WB). These control-signals select the input data for the execution units

[10] Register File implements the thirty-two GPRs of 32 bits each.

from the Operand Muxes. These control-signals also select the execution-unit to execute a particular instruction. Each execution-unit has a different execution time depending on the instruction being executed. The LS stage comes on the way for Load/Store instructions. The address of the destination operand is also parsed out from the instruction in the `id_insn` register. This address is registered to be used in the Write Back (WB) stage. When all this done, the `id_insn` instruction shifts to the `ex_insn` register.

The results from the execution-units are fed into the Writeback Muxes. The Writeback Muxes store the calculated result to the Register File (to the destination GPR) and also send it to the Operand Muxes so that the ID stage can use it to handle data dependencies. Finally, the execution of the instruction ends and it moves from the `ex_insn` register to the `wb_insn` register. Currently, there is no use of this register – may be it is for future use.

Behavioral Outlook

Figure [2.26] shows the behavioral model of the OR1200 pipeline. It is clear that the calculation of the next PC (GenPC) and fetching a new instruction (IF stage) are in parallel. The calculation of the next PC follows a procedure. The GenPC block monitors if the exception-model has generated an exception. If YES, the next calculated PC would be the exception-vector corresponding to the generated exception. If NO exception was generated, the GenPC checks for the SPR control-block's implications on the next PC. The SPR control-block generates the address of the next instruction under some specific conditions e.g., the *l.mtspr*[11] instruction writes the PC. Furthermore, if there are no implications from the SPR-block, the GenPC block checks if the instruction in the ID stage is a branch/jump. If YES, the next PC will be the offset address generated by this instruction. Finally, if there was no branch/jump instruction in ID stage, the PC will be simply incremented to the address of the next instruction in the sequence. The updated PC goes to the IC if existed in an implementation. Otherwise, it directly goes to instruction memory (IM). The GenPC stage ends up here.

The instruction at the PC address (in IC/IM) is fetched and stored into a pipeline register (`if_insn`). If no instruction was fetched (error in fetching a new instruction/RFE instruction/branch taken), a default instruction is fed into the `if_insn` register. The instruction in the `if_insn` register is translated to (i) get the addresses of the source operands, (ii) to see if the instruction is with an immediate, and (iii) to identify a branch/jump instruction (to handle the branches/jumps earlier).

The addresses of the source operands are sent to the register file. The register file takes one clock cycle to make the GPRs ready at the inputs of the Operand Muxes (separate muxes for operand-1 and operand-2). Simultaneously, the instruction in the `if_insn` register is moved to the `id_insn` register i.e., to the ID stage. Thus, the register file has been read simultaneous to the ID stage.

When the instruction comes to the ID stage (`id_insn`), its fields are translated to the control-signals for the subsequent pipeline stages (EX/WB). Some important steps of the ID stage are:

1. Get the control-signals for the Operand Muxes to select the source operands for the instruction being decoded. These control-signals select the output of the Operand Muxes from: (i) the input GPRs (from the RF), (ii) the result from the instruction that just finished its execution and entered into the WB stage (data dependency), or (iii) the result from the instruction which has finished the WB stage (data dependencies).

[11] OR1200 instruction to write the SPRs.

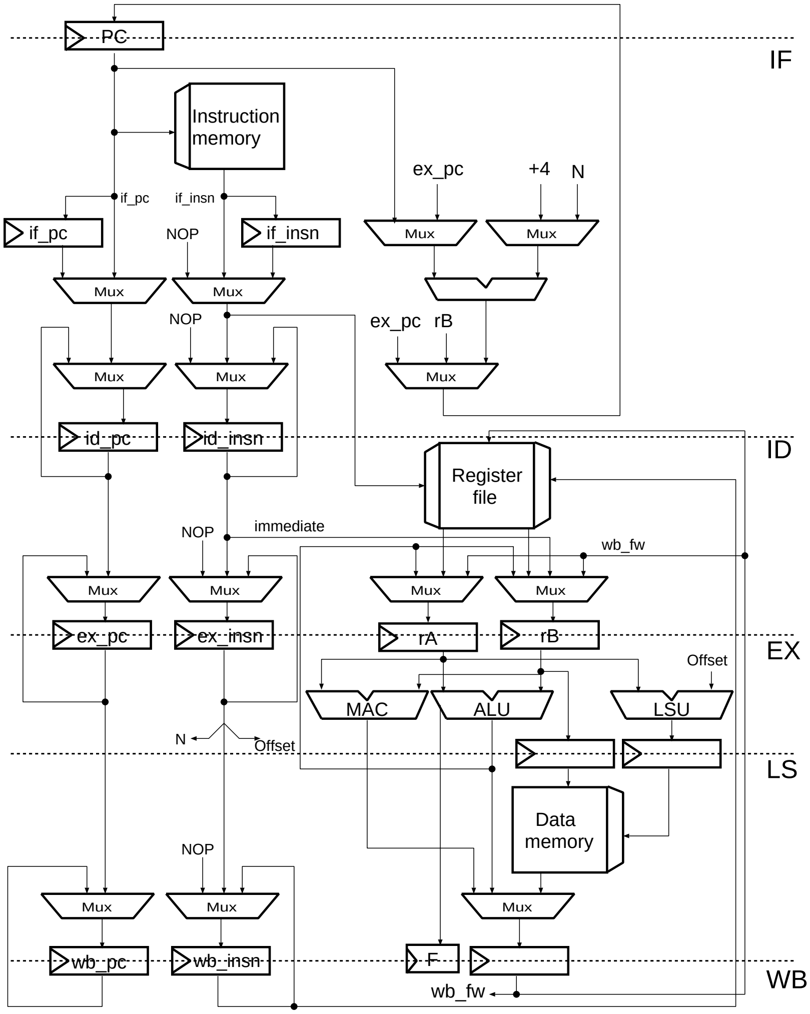


Figure 2.26 Behavioral view of the OpenRISC1200 pipeline [14].

2. Get the control-signals to select the required execution-unit (ALU/MAC/LSU). This selection is based on the instruction's opcode. The opcode is also parsed out and sent to the execution-units. Each execution-unit takes the opcode and executes the instruction if it is valid for it. However, the result only qualifies from a selected execution-unit depending upon the control-signals.

3. Get the control-signals to manage the pipeline's timing. The execution-units have different execution time depending on the instruction being executed. All ALU instructions (except those with immediate) finish the execution in one clock cycle. The LSU take two clock cycles for READ (assuming a hit in the DC) and one clock cycle for the WRITE (assuming a hit in the DC). The MAC unit can accept new instruction every cycle (except multiply and divide). Hence, some logic is needed to cope with these variable execution times. The instruction in the ID stage is translated to know its execution time. This information is sent to a block called Freeze logic which implements the logic to stall the different pipeline stages for the synchronization.

   Example Suppose the instruction in ID stage was a Load, the Freeze logic will know that this instruction will take two clock cycles during execution and it will stall the pipeline stages behind the EX stage for two clock cycles.

4. If the instruction is *l.mtpsr/l.mfspr*, calculate the address of the SPR that is going to be written/read in the EX stage.

5. If the instruction is with immediate, a 16-bit immediate (`id_insn` (15: 0)) is chopped out and sign/zero-extended to 32 bits.

6. Get the control-signal (used in step 5) to see whether the immediate should be sign-extended or zero-extended.

7. Parse out the destination address (in the register file). This address is registered for laterly used in the WB stage.

8. Identify if the instruction is illegal or not implemented.

9. The control-signals (from step 1) select the required source operands from the Operand Muxes. These operands are stored in two registers (rA/rB) and become ready to be used by the execution-units.

When all this done, the instruction in the `id_insn` register is shifted to the `ex_insn` register and the EX stage starts.

All execution-units are fed with the input registers (rA/rB) and the opcode of the instruction and they execute the instruction[12]. For multi-cycle instructions, the Freeze logic stalls the previous and last stages for the execution time. If the instruction is Load/Store, the EX stage also includes the LS stage. Eventually, the results from all execution-units come to the inputs of the Writeback Mux. This result is also fed back to the Operand Muxes to handle the data dependencies. The control-signal selects the correct result (from the selected execution-unit) which goes to the output register

[12] If an instruction does not belong to an execution-unit, the opcode is no-operation (NOP)

of the Writeback Mux and is finally stored into the register file at the destination address (step 8 in ID stage). The written result is also fed back to the Operand Muxes to handle the data dependencies.

The exception-model of the OR1200 functions in parallel to the instruction execution. The exception-block especially looks if the instruction in the EX Stage generates an exception, then calculates the exception-vector and sends it to the GenPC for next PC address calculation. When an exception occurs, the exception-block saves the current context of the CPU by copying (i) the SR to ESR, (ii) ID_PC/EX_PC/WB_PC (depending on the pipeline stage of the instruction which caused the exception) to EPCR and EEAR, and (iii) also sends the control-signal to the SPR-block to update the SR.

2.7 Maximum Throughput Restrictions on Subsystem

A default implementation of the OR1200 core can support single cycle execution for most of its instructions. However, an implementation without caches (IC/DC) has some restrictions on its maximum possible throughput. The OR1200's Wishbone interfaces (IWB/DWB) have registered outputs. It always takes at minimum one clock cycle to initiate a new request (instruction/data) after receiving the acknowledgment of the previous one. Hence, each request takes at least two clock cycles to complete. In order to reduce the area and power consumption of the CPU Subsystem the used implementation of the OR1200 core does not include the caches and memory management units. The OR1200 core cannot support single cycle execution without caches since burst accesses are not possible. Therefore, the maximum throughput of the OR1200 core will be two clock cycles per instruction instead of a single cycle. However, this throughput is only possible if the core can fetch a new instruction without further delays from the memory subsystem or from the bus. The triple-layer Sub-bus does not need any extra cycle. However, the memory subsystem cannot support single cycle access for the READ requests because of the synchronous termination (see Section 2.3.3). Hence, each instruction fetch takes at least three clock cycles in the CPU Subsystem. Consequently, the minimum execution time of a single cycle instruction will be three clock cycles. Thus, the maximum throughput of the OR1200 core in the CPU Subsystem will be three clock cycles for most of its instructions.

2.8 Simulation Framework

2.8.1 Overview

We have discussed all components of the CPU Subsystem. In this section, we will discuss how to interconnect them to build-up the CPU Subsystem. First we will discuss the generation of the "memory initialization file" needed to simulate the Subsystem. There are several formats of memory initialization files. We have opted the "Intel Hexadecimal File format" (IHex) to provide the memory initialization data to the CPU Subsystem. A specific tools-chain is used to generate a memory initialization file for a specific processor. Similarly, a GNU toolchain is used to generate the memory initialization file for the OR1200. In this section, we will discuss: (i) the installation of the OR1200 GNU toolchain, (ii) the generation of the OR1200 memory initialization file (IHex), and (iii) the building up of the Subsystem and its simulation.

2.8.2 OpenRISC1200 GNU Toolchain

Some open source software has been ported to the OR1200 platform e.g., Linux and *μClinux*. For easier software development a GNU toolchain has been also successfully ported to the OR1200 architecture. The tools include:

- GNU binutils-2.18.50,
- GNU GCC-4.2.2,
- GNU GDB-6.8,
- *μClinux*-0.9.29,
- Linux-2.6.24,
- BusyBox-1.7.5 and
- Or1ksim-0.3.0.

More details about the OR1200 GNU toolchain can be found in [15–18].

GNU Toolchain's Installation on Linux (Ubuntu 8.10)

The OR1200 toolchain was developed and being maintained by the `OpenCores` organization. Manually downloading and installing the complete toolchain is tricky. Thanks to OpenCores for developing a script which downloads and installs the latest versions of all tools.

The script (**MOF_ORSOC_TCHN_v5c_or32-elf.sh**) can be downloaded from the OpenCores. The toolchain can be installed on a Windows platform using Cygwin, a Unix-like shell environment. More details can be found on the OpenCores website.

I was working on `Ubuntu 8.10` distribution and it required some standard development tools to be installed before compiling the OR1200 toolchain. It depends on the Linux distribution which tools are pre-installed and which should be installed. I used the `apt-get` package management tool to perform this installation. The given set of `apt-get` commands ensures that the required packages, to build the OR1200 toolchain, have been installed. After downloading the script, we need to perform the given steps to build the OR1200 toolchain. By default, this script installs the toolchain under the current directory but it can be changed. I installed the toolchain under the `/opt` directory. The toolchain requires at least two GB of free space on the local disk to built.

```
sudo apt-get update
sudo apt-get -y install build-essential
sudo apt-get -y install make
sudo apt-get -y install gcc
sudo apt-get -y install g++
sudo apt-get -y install flex
sudo apt-get -y install bison
sudo apt-get -y install patch
sudo apt-get -y install texinfo
sudo apt-get -y install libncurses-dev

/*=========Build the OR1200 GNU Toolchain===========*/
sudo mv "down_load_dir"/MOF_ORSOC_TCHN_v5c_or32-elf.sh /opt
cd /opt
sudo sh MOF_ORSOC_TCHN_v5c_or32-elf.sh

/*====Follow the prompt, when asked by installer====*/
1st time -> Y
2nd time -> N

/*==========Set the Environment variables===========*/
```

```
            gedit ~/.bashrc
            --> Add
              export PATH="$PATH:/opt/or32-elf/bin"
```

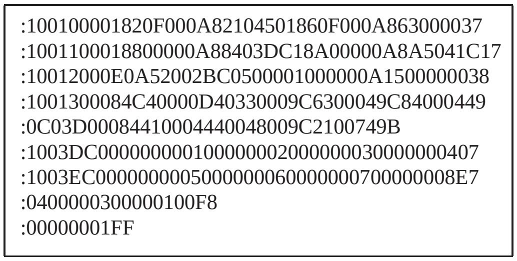
The GNU toolchain has been installed. It is ready to compile and to simulate applications developed for the OR1200 architecture.

Memory Initialization File

While developing applications for microcontrollers or microprocessors, we need to convey the binary information to program them. The memory initialization files are used to provide the binary information of an application to a specific processor architecture e.g., the OR1200.

The `Intel Hex file` (IHex) is a common format for the memory initialization files. It is an ASCII file with lines of text. Each line follows the Intel Hex format and contains one Hex record. These records are hexadecimal numbers representing machine-language code or/and constant data. Intel HEX files are mostly used to transfer the program and data to the ROM/EPROM.

There are three types of Intel HEX files distinguished by their byte orders: (i) 8 bit, (ii) 16 bit and (iii) 32 bit. Figure [2.27] shows a 32 bit Intel HEX file of a sample program executed on the Subsystem.



```
:100100001820F000A82104501860F000A863000037
:1001100018800000A88403DC18A00000A8A5041C17
:10012000E0A52002BC0500001000000A1500000038
:1001300084C40000D40330009C6300049C84000449
:0C03D00084410004440048009C2100749B
:1003DC000000000100000002000000030000000407
:1003EC0000000005000000060000000700000008E7
:0400000300000100F8
:00000001FF
```

Figure 2.27 Intel HEX memory initialization file (IHex).

Each line of the file follows the Intel HEX format and comprises of 6 parts:

1. **`Start code:`** Every line starts with a single character ASCII code (**:**).

2. **`Byte count:`** The first two hex digits, after the start code, show the number of bytes (hex-digit pairs) in the data field e.g. byte count 0x10 or 0x20 identifies 16 or 32 bytes of data respectively.

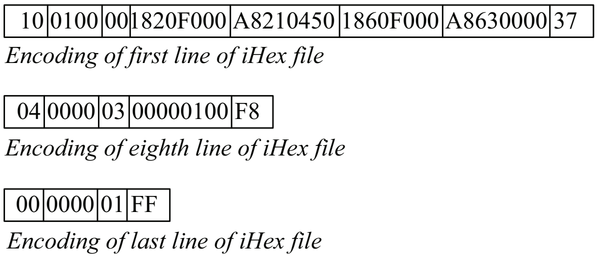
3. **`Address:`** The four hex digits, after the byte count, identify the 16 bit big-endian address of the beginning of data in the memory.

4. **`Record type:`** The two hex digits, after the address, define the type of the data field. There are six types of data fields identified by the record type (00 to 05). The record type **00** identifies that its a data record containing data and a 16-bit address. The record type **01** identifies an end-of-file record and record type **03** identifies a start segment address record.

5. **`Data:`** A sequence of **n** bytes (2n hex-digits) of data, where the byte count specifies **n**.

6. **`Checksum:`** The last two hex-digits are the two's compliment sum of the values in all fields except the start code (:) and the checksum itself.

Figure [2.28] shows the encoding of different lines in the Intel HEX file (Figure [2.27]). The first line contains four data sequences, each of 32 bits. These data sequences can hold either 32-bit instructions or data values.



10	0100	00	1820F000	A8210450	1860F000	A8630000	37

Encoding of first line of iHex file

04	0000	03	00000100	F8

Encoding of eighth line of iHex file

00	0000	01	FF

Encoding of last line of iHex file

Figure 2.28 Encoding of the Intel HEX format.

Generation of the IHex File for the OpenRISC1200 core

A test program in C/C++ is compiled with the OR1200 GNU toolchain to generate the Intel HEX file for the OR1200 architecture. However, compiling a C program with the GNU toolchain requires a few necessary things to do first.

`Linker Script`

A linker script is needed which sets up the memory-map of the application by specifying: (i) the address mapping of the memories, (ii) the place of text and data in the memories, the sizes of these sections, and (iii) the positions and sizes of the stack and heap in the memories. A linker script for the CPU Subsystem has been given in Appendix (A.1.3).

`Startup Script`

Since all initialized data sections go to the RAM, we have to write additional code to initialize these sections. It is better to write a startup script to include explicit initialization code in the application program. Usually, the startup script includes (i) additional code for the stack initialization and (ii) code for copying the initialized data and static variables from the ROM to the RAM. The startup script used to generate the memory initialization file for the OR1200 core, within the CPU Subsystem, has been given in Appendix (A.1.4).

`Makefile`

It is easier and preferred to write a makefile to compile everything together. While compiling C program, we specify the linker script and the startup script to get the executables. In addition to the executables, we can also specify other kind of object files to be generated e.g., (Intel HEX file, .LST file[13]). A sample makefile for the CPU Subsystem has been given in Appendix (A.1.5).

[13] **.lst file** gives information about the memory mapping for different sections. It also contains the disassembly of the .TEXT section (i.e., C code) of the linker script

After generating the executable of a C application the OR1200 architectural simulator (Or1ksim) can be used to execute the program. It is good to know the results of the execution before simulating the program on a real system. To execute the application on the simulator we need to provide the executable file of the application and the configuration file.

Configuration File

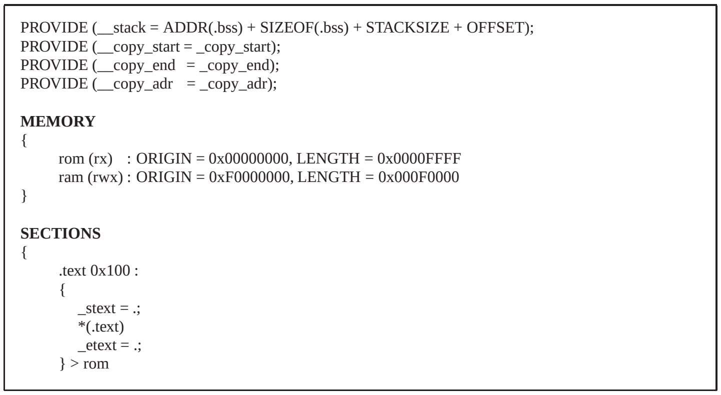
A configuration file is needed to configure the OR1200 architectural simulator (Or1ksim). The configuration file provides the OR1200's settings e.g., implemented peripherals (memories, UART etc.) and the settings of these peripherals (if implemented). This file also includes the configuration settings for the ISS itself. If no configuration file is provided, the ISS uses a default configuration file named `sim.cfg`. More details about the configuration file can be found in the official manuals of the Or1ksim simulator [17, 18].

2.8.3 Simulation Setup for the CPU Subsystem

This section will explain the method to interconnect all components to implement the CPU Subsystem. It will also shed light on converting IHex files to binary data so that it can be loaded into the ROM of the CPU Subsystem. Finally, it will describe the simulation of the CPU Subsystem.

Incorporation of the Subsystem

Interconnecting all components to implement the Subsystem (Figure [2.2]) is straight forward but demands a good care. A correct configuration of the Sub-bus system is a critical part while connecting all components together. The address widths of the Sub-bus Slave interfaces should be equivalent to the connected Slave component's address width. Care must be taken while selecting the identities and the encoding bits for the Sub-bus Slave interfaces to prevent conflicts. These IDs and encoding-bits must comply to the memory-map specified in the linker script (Appendix A.1.3). For clear understanding, a snapshot of the memory-map has been shown in Figure [2.29]. It shows the base address and size of the ROM and the RAM. The section .TEXT shows the place of the application code in the memory.



```
PROVIDE (__stack = ADDR(.bss) + SIZEOF(.bss) + STACKSIZE + OFFSET);
PROVIDE (__copy_start = _copy_start);
PROVIDE (__copy_end   = _copy_end);
PROVIDE (__copy_adr   = _copy_adr);

MEMORY
{
        rom (rx)   : ORIGIN = 0x00000000, LENGTH = 0x0000FFFF
        ram (rwx) : ORIGIN = 0xF0000000, LENGTH = 0x000F0000
}

SECTIONS
{
        .text 0x100 :
        {
            _stext = .;
            *(.text)
            _etext = .;
        } > rom
```

Figure 2.29 Linker script's snapshot.

A correct configuration of the Sub-bus has been shown in Figure [2.30]. In which eight MSBs have been taken as encoding-bits to identify the Sub-bus Slave interfaces. It is important that the identities for the ROM and the RAM are valid for their address spaces. From the linker script, the complete address space of the RAM always contains 0xF0 (240) in its upper eight bits. All addresses holding 0xF0 in their upper eight bits will be forwarded to the RAM. There must not be any other peripheral (connected to the Sub-bus) having the same ID in its encoding-bits (no overlapping or conflict).

```
entity crossbar is
  generic(
      --Generics
      aw_wb_g          : natural := 32;
      aw_rom_g         : natural := 32;
      aw_ram_g         : natural := 32;
      aw_sbus_g        : natural := 32;
      aw_scpu_g        : natural := 32;
      enc_bits_rom_g   : natural := 8;
      enc_bits_ram_g   : natural := 8;
      enc_bits_sbus_g  : natural := 8;
      enc_bits_scpu_g  : natural := 8;
      rom_id_g         : natural := 0;
      ram_id_g         : natural := 240;
      sbus_id_g        : natural := 255;
      scpu_id_g        : natural := 15
    );
  port(
    -- clock and reset
    clk_i   : in std_ulogic;
    reset_i : in std_ulogic;
```

Figure 2.30 Sub-bus configuration's snapshot.

After configuring the Sub-bus system it is simple to connect all components togather, as given in Figure [2.2].

Simulation of the Subsystem

In order to simulate the CPU Subsystem, we need to load the Intel HEX file of a test program (C/C++) into the ROM starting from the reset address of the OR1200 core (see Section 2.4.3). We need to parse the Intel Hex file to take out the data sequences (instruction/data) and load them into the ROM.

Parsing the Intel HEX file has been implemented inside the ROM. We just need to provide the name of the file to the ROM. The ROM will parse this file and load the binary data sequences starting from the reset address of the OR1200 core, Figure [2.27]. It is clear to see that the data sequences in the first line of the sample IHex file has been loaded into the ROM at 0x00000040 address.

Finally, we need a simple test bench to instantiate the CPU Subsystem and to drive the system clock and the system reset signals to it. The system clock for the CPU Subsystem is 100 MHz. When the test bench asserts the system reset the OR1200 core sends the READ request (from address 0x00000100) to its instruction Wishbone-interface (IWB). Since the instruction Master interface (of the Sub-bus) has been connected to IWB, the decoder will translate the requested address (0x00000100). The address qualifies for the Sub-bus Slave interface connected to the ROM. The READ request for address 0x00000100 would be forwarded to the ROM. Thus, the OR1200 core will fetch the first instruction and execute it, and so on. However in the Subsystem, fetching a new instruction will always take at least three clock cycles (see Section 2.7).

In order to access the RAM, the processor will generate a request with an address that qualifies for the Sub-bus Slave interface connected to the RAM. The configuration of Sub-bus is according to the memory-map of the connected Slave components (ROM/RAM). Therefore, an address that qualifies for a Sub-bus Slave interfaces also lies inside the address-space of the connected Slave component.

Chapter 3

Verification Fundamentals

3.1 Introduction

This chapter focuses on different verification approaches, methodologies and technologies available in the market. It also provides a brief introduction about OVM, SystemC library and SystemVerilog direct programming interface (DPI).

3.2 Functional Verification

3.2.1 General Description

The main source of functional errors in a design may be associated to the following:

- ambiguities in product intent
- functional specification ambiguities
- specification are clear but designer misunderstood it
- design implementation errors

The basic objective of *functional verification* is to verify that the initial design implementation is equivalent to the product intent. The functional verification facilitates to identify if any differences exist between (i) the product intent, (ii) the functional specification and (iii) the design implementation. The complexity of functional verification of a design is an NP-hard problem.

3.2.2 Verification Approaches

The main objective of functional verification is to make sure that a design works properly when stimulated on its boundary. Any error in the design is ignored that cannot be stimulated and observed on its boundary. These errors include (i) errors that cannot be activated, (ii) errors that can be activated but never observed and (iii) multiple errors that can potentially hide one another.

Different approaches are used to increase the efficiency and completeness of verification, as described below:

Black-Box Verification

In this approach, a design under verification (DUV) is treated as a black box (closed box) and its implementation have no considerations. The DUV is accessed only through available interfaces and its internal state cannot be accessed. This verification approach lacks controllability and observability which makes it difficult to (i) set up a certain functional state of the design, (ii) isolate a particular functionality and (iii) correlate the output response to an input stimulus. The test bench can be developed in parallel to the design implementation. However, it is not possible to verify a large design as a complete unit due to the discrepancy of the functionality against the controllability and observability.

Gray-Box Verification

In this approach, the DUV is treated as a closed box though its internal structure is known. The input stimulus is applied through external interfaces and its target is to activate the implementation specific features of the DUV e.g., an internal FSM goes through a particular state sequence. This approach is used to increase the verification coverage. A design can be modified to increase the controllability or observability. It is called *design-for-verification* [19]. An example of such modifications is the addition of easily controllable registers to set up a particular internal state of the design.

White-Box Verification

This verification approach offers full controllability and observability of a design such as setting up a particular state or bypassing some internal units. Such verification depends on a particular implementation hence the test bench can be developed once a design is implemented.

Combined Black and White Box Approaches

- **Black/Grey Stimulus and White Observation**

  This approach is a combination of black- and white-box verification approaches. A DUV is simulated using black- or grey-box stimulus and assertions (temporal) are used on the DUV signals (internal and output) for observation (white-box) [19].

- **Lots of Black-Boxes in a Large White-Box**

  This verification approach is used for complex designs. The components of such designs are first verified (black-box). Hence, there is no need to re-verify them once interconnected. The white-box verification approach is used to verify the cooperation of already verified components [19].

3.2.3 Verification Challenges

Functional verification may look like a very simple task at first place but it is very difficult to address. The increasing complexity of designs and shortening time-to-market put more demands on verification engineers to verify complex designs in shorter time. The following challenges must be addressed to achieve the verification closure [20].

- **Verification completeness**
  - Maximizing the part of the design that is verified.
  - Capturing all scenarios that must be verified.
  - Maximizing the use of coverage driven verification methodologies.
- **Verification reusability**
  - Increasing the reusability of a verification environment infrastructure.
  - Use of standardized interfaces or functions.
  - Identification of common functionality in the verification environment that can be reused.
- **Verification efficiency**
  - Minimizing the manual effort to complete a verification project.
  - Use of automated systems.
- **Verification productivity**
  - Maximizing the work produced manually by verification engineers in a given amount of time.
  - Moving to higher levels of abstraction and leveraging reuse concepts.
- **Verification code performance**
  - Maximizing the efficiency of verification programs.
  - Increasing the knowledge of tools and languages used to implement the verification environment.

3.3 Verification Technologies

3.3.1 Overview

Typically, there are three kinds of technologies available to perform functional verification of designs, as given below:

- Simulation-based verification,
- Formal verification and

- Acceleration/Emulation-based verification.

The emulation-based verification is beyond the scope of this report. The following subsections provide an overview of other two technologies.

3.3.2 Simulation-based Verification

All possible behaviors of a system are required to be considered while verifying that the design implementation is correct corresponding to its specification. We can be sure that no design error has escaped if the complete functionality of a design has been covered in the verification. The process in which we examine the behavior of an implementation and increase the confidence in its specification is usually called *validation*.

A simulation is a usual method to discover design errors in the validation. In the *simulation-based verification* next state values of a design are evaluated in terms of its current state and input values. Whereas, the future value assignments are scheduled to design signals by considering signal delays [20]. In this type of verification a verification environment is consisted of a test bench and a design implementation. The test bench is used to apply input values to the DUV. The next state values of the DUV are computed based on these input values. Finally, it is checked whether the computed state is the expected state of the design. All possible input combinations are required to be covered in simulation-based verification to have a full confidence in the design. Therefore, this approach is impractical for designs of a moderate size. As increasing number of inputs exponentially increase input sequences that must be verified. Consequently, we have to reduce the number of input stimuli and as a result design errors possibly remain undetected [21].

3.3.3 Formal Verification

Formal verification is a practical solution to handle limitations of simulation-based verification. In formal verification, the behavior of a design is mathematically proven i.e., an implementation behaves according to its specification for all time instances and for all input variations. Formal verification proves or disproves a given property of a hardware implementation by using logical and mathematical equations and methods [20]. In formal verification, we prove mathematical equations that describe the system. Hence, any property proved by the formal verification holds for all possible input vectors applied to that implementation. The major advantage of formal verification techniques is the ability to make universal statements about a property of a design implementation. These statements hold for all possible input streams without requiring test vectors to be applied. There are two major categories of formal verification techniques, as given below:

- Equivalence checking and
- Property checking.

These approaches are discussed in the following subsections.

Equivalence Checking

Two formal representations of a design implementation (before and after a given transformation) are provided as input to an equivalence-checking tool. This tool creates a canonical representation of each implementation. Since the canonical representation is unique for every Boolean function under an assumed set of conditions (e.g., variable ordering), proving the equivalence of these two representations is typically straightforward. The most common input representations of a design to equivalence-checking tools are RTL and netlist of gates. The development of such a tool for designs of larger sizes is a difficult task. Moreover, creating a canonical representation for very large system is not practical. Therefore, we need to develop some special tricks or even manual intervention may be required to reduce the size of a design to formally verify it. It means that equivalence checking cannot make a large contribution to main challenges of the functional verification.

Property Checking

Property checking is another formal verification approach that is a very powerful technique to address functional verification challenges. Given a formal description of a design implementation (e.g., an RTL description) property checking verifies that a given property described in temporal logic holds for the given implementation [20]. A design property that is to be verified can be formulated as an equation of the design's behavior. The design properties are specified as a set of assertions.

The following advantages of property checking make it more powerful and a suitable technique for the functional verification.

- The properties can be described at any level of the product specification and the design creation. They can be collected incrementally as specification and development proceeds.
- Property checking can be performed in the beginning stages of the design even when a verification environment is not yet available to provide a test stimuli.
- The properties can be used with emulation-based verification and simulation-based verification.
- Property checking provides the coverage collection that is needed to check the verification completeness.
- Property checking is a static technique in which no test bench or logic simulator is required.

There are multiple languages to facilitate property checking, including (i) Property Specification Language (PSL) and (ii) SystemVerilog (properties are defined in the form of assertions).

Limitations of Formal Verification

It is an often-repeated myth that formal verification guarantees the perfectness of systems. Though in reality it can significantly increase the confidence in a design. However, an absolute flawlessness of systems cannot be guaranteed. Since the verification only allows the detection of design faults and does not identify fabrication faults or faults while a system is in use. The verification checks the correctness of statements according to the formal specification of a design which can be incomplete or faulty itself. Moreover, the verification tools may contain faults in their programs. Hence, the formal verification should be taken as an adjunct to but not as a substitute for standard quality assurance methods [21].

3.3.4 Formal Verification vs Simulation-based Verification

The following example illustrates the difference between formal verification and simulation-based verification. We have to show that the Equation (3.1) holds by proving that both of its sides give the same result for all possible input values.

$$(x+1)^2 = x^2 + 2x + 1 \tag{3.1}$$

In a simulation-based verification, this equation is checked using concrete input values for the variable x, as shown in Table (3.1). Although Equation (3.1) holds for all input values of variable x in Table (3.1), the simulation is still not capable of establishing the validity of the equation. In a formal verification, this equation is proven by applying mathematical transformation rules, as shown in Table (3.2) [21].

x	$(x+1)^2$	x^2+2x+1
0	1	1
1	4	4
2	9	9
3	16	16
9	100	100
67	4624	4624
...	...	...

Table 3.1 Simulation values of Equation (3.1).

1.	$(x+1)^2 = (x+1)(x+1)$	definition of square
2.	$(x+1)(x+1) = (x+1)x + (x+1)1$	distributivity
3.	$(x+1)^2 = (x+1)x + (x+1)1$	substitution of 2. in 1.
4.	$(x+1)1 = x+1$	neutral element 1
5.	$(x+1)x = xx + 1x$	distributivity
6.	$(x+1)^2 = xx + 1x + x + 1$	substitution of 4. and 5. in 3.
7.	$1x = x$	neutral element 1
8.	$(x+1)^2 = xx + x + x + 1$	substitution of 7. in 6.
9.	$xx = x^2$	definition of square
10.	$x + x = 2x$	definition of 2x
11.	$(x+1)^2 = x^2 + 2x + 1$	substitution of 9. and 10. in 8.

Table 3.2 Formal proof of Equation (3.1).

3.4 Verification Methodologies

Different technologies and multiple facilities are used during verification activities. To produce an expected outcome for any given project, different methodologies are used to bring together these tools and facilities. Most commonly used types of these methodologies are: (i) assertion-based verification and (ii) coverage-driven verification (CDV). The assertion-based verification focuses on how assertions can be involved throughout the design flow and across multiple tools. The coverage-driven verification is concerned with the best approach for designing and implementing the verification project. Both approaches overlap each other because an assertion can be considered as a coverage point for the coverage analysis [20].

The following subsections briefly describe these two methodologies.

Assertion-based Verification

In this verification methodology, assertions are used as an integral part of the functional verification flow. Main components of this methodology are: (i) identifying main properties to be asserted, (ii) deciding when those properties must be asserted and (iii) verification tools used to confirm asserted properties. The main categories of properties that must be verified are: (i) operating environment assumptions, (ii) verification related assumptions, (iii) design specifications, and (iv) design and implementation decisions. It is not necessary that all properties must be satisfied at all time during a device operation e.g., any device property may fail during the reset sequence. Therefore, such properties may not be asserted during the reset sequence.

Coverage-driven Verification

It is a simulation-based verification approach particularly developed to focus on the productivity and efficiency related challenges faced in any functional verification project (see Section 3.2.3). The coverage-driven approach improves the verification completeness and correctness. The basic idea behind this approach is the random generation of the stimulus, which is the main source of the productivity gained in this methodology. The coverage collection is a necessary part when the stimulus generation is randomized. As in the absence of coverage no information is available about scenarios covered. Some examples of CDV approaches are listed below.

- *Transaction-driven verification:* It allows scenarios to be specified at a higher level of abstraction.
- *Constrained random stimulus generation:* It leads to productivity gains in generating the scenarios.
- *Automatic result checking:* It provides confidence that the design works for all randomly generated scenarios.
- *Coverage collection:* It is a mandatory approach as in the absence of coverage it is not obvious which scenarios have been randomly generated.
- *A directed-test-based verification:* It is also a necessary approach because not all scenarios can be generated efficiently by only using random generation techniques.

3.5 Verification Cycle

A basic intention of any verification project is to achieve the complete verification of all design features in a minimum possible time and within applicable resource limitations. Since these requirements are very important, a verification plan must hit a balance among them. It is even trickier as parameters may change throughout the design flow. For example, (i) the project deadline may be reduced, (ii) project features may be redefined or changed or (iii) the availability of resources may be changed. An interactive process is followed to find a feasible solution that reaches this balance. Figure [3.1] shows the steps involved in creating and executing a verification plan. The basic steps shown in figure are:

1. Building a verification plan,

2. Building a verification environment,
3. Executing the verification environment,
4. Measuring results and
5. Reacting to measurements.

In each iteration of this cycle, the verification completion is checked until it reaches the expected goals.

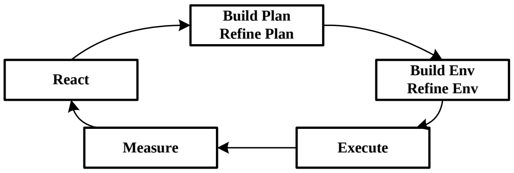


Figure 3.1 Verification cycle.

Building a verification plan is the core of a verification project. It includes following important steps: (i) identification of all actors that are concerned with the project execution, (ii) preparation of planning sessions and planning documents, (iii) brainstorming of product functionalities, (iv) structuring the verification plan, (v) capturing features and attributes, and (vi) formulating the verification environment and the coverage implementation.

3.6 Verification Environment

3.6.1 Introduction

The verification environment must be implemented in a way that it should allow all scenarios in the verification plan to be verified according to the guideline of the target verification methodology. Generally, there can be different verification environment architectures available to achieve this target. This section briefly discusses a verification environment architecture that facilitates the application of the CDV methodology and the assertion-based verification methodology. The OVM, which provides the best outline to accomplish a CDV is also discussed in this section. This section emphasizes on the architectural blocks of a verification environment, how these blocks are generally used in the verification environment and the features that should be supported by each block.

Abstract View of a Verification Environment

A verification environment is connected to a DUV through the boundary signals of that DUV. The boundary signals can be grouped into interfaces that are comprised of multiple ports. Each port represents interconnected signals that jointly describe an interface protocol supported by the DUV. In this way, a DUV is viewed as a block with a number of abstract interfaces suggesting a layered architecture for its verification environment. Figure [3.2] shows a layered architecture of a verification environment in which the lowest layer components directly interact with DUV interfaces.

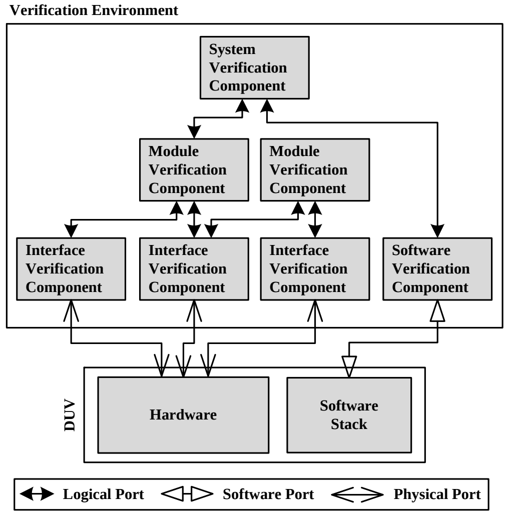

Verification Environment
System Verification Component
Module Verification Component
Module Verification Component
Interface Verification Component
Interface Verification Component
Interface Verification Component
Software Verification Component
DUV
Hardware
Software Stack
Logical Port
Software Port
Physical Port

Figure 3.2 Abstract view of a verification environment.

Each higher layer component deals with increasingly higher levels of verification abstraction that correspond to more complex verification scenarios [20, 22].

This verification environment is structurally comprised of interface verification components (IVCs) and system/module verification components (SVC/MVCs). The IVCs provide abstraction for physical ports to interact with the DUV. The SVCs/MVCs make use of this feature to interact with the DUV at the level of abstraction provided by the IVCs. In this architecture, software verification components are a specific type of IVCs that interact with the software stack of the DUV.

There are two operational modes for every verification component, as given below:

- Active mode and
- Passive mode.

An SVC in an active mode generates transactions for lower layer verification components while an IVC in active operational mode generates transactions at DUV ports. A passive verification component does not include any stimulus generation capability. It only monitors the verification environment traffic. These modes must be correctly implemented when a verification component is reused in the next design integration step.

3.6.2 Interface Verification Component (IVC)

The IVC is used to interact with one or multiple DUV ports that support the same protocol. The IVCs also include supplementary features to monitor and collect coverage information of the physical port they interact with. The architecture of an IVC is geared less towards generating full verification scenarios since concurrent interaction with multiple ports is required for this purpose. However, this architecture is more equipped to give an abstract view of DUV ports to higher layer verification components. They monitor the traffic on DUV ports by protocol checking and coverage collection.

Figure [3.3] shows the architecture of an IVC that contains (i) agent components and (ii) a bus monitor. Each IVC interacts with a DUV port through an agent component that again includes following components:

- A driver,
- A monitor and
- A sequencer.

Details about these components are provided in Section (3.7.3).

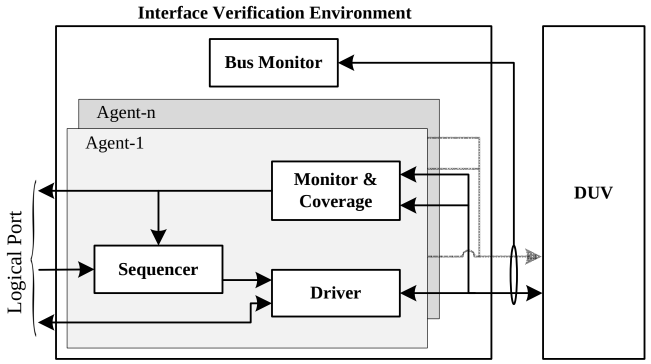


Figure 3.3 Block diagram of the interface verification component.

3.6.3 Module/System Verification Component

A three-layer verification environment is shown in Figure [3.2], which is composed of (i) IVCs, (ii) MVCs and (iii) SVCs. Although, practically a verification environment may have more layers. The SVCs include system level set-up generation functionality and perform end-to-end checking. The internal architecture of MVCs and SVCs is similar because they both interact with higher and lower layer verification components. The architecture of IVCs is different since they interact directly with the DUV ports. The SVCs generally emphasize on the end-to-end behavior of the DUV rather than the behavior of its individual blocks. In this approach it is assumed that smaller blocks have already been verified.

An SVC emphasizes on (i) bugs in modules that can be verified only as a part of the overall system, (ii) wrong assumptions of the designer about the module operation, (iii) wrong wiring between system modules and (iv) problems with module interactions arising from protocol mismatches. Figure [3.4] shows the architecture of a SVC containing multiple agents where each agent provides the same functionality while interacting with a different set of lower layer verification component. Each SVC includes (i) the sequencer, (ii) the verification environment (VE) monitor and coverage collector, and (iii) the DUV monitor and coverage collector.

To provide information about the current state of the DUV, the *VE monitor* interacts with monitors in the lower layer verification components. For example, system monitors track the monitors in

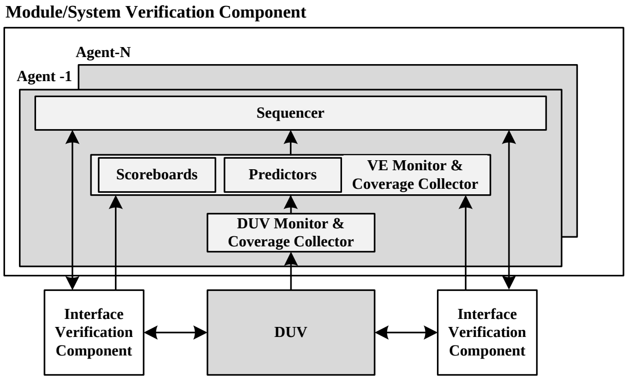

Module/System Verification Component
Agent-N
Agent -1
Sequencer
Scoreboards
Predictors
VE Monitor & Coverage Collector
DUV Monitor & Coverage Collector
Interface Verification Component
DUV
Interface Verification Component

Figure 3.4 Block diagram of the module/system verification component.

the IVSs and in the MVCs. Since internal signals of the DUV cannot be tracked through monitors attached to the DUV ports, a DUV monitor is used to track these internal signals. However, a thin layer of a wrapper between the DUV monitor and the DUV enables the reusability of the verification environment. A combination of both monitors (the VE monitor and the DUV monitor) allows a gray-box verification approach. A sequencer uses the information provided by these monitors to generate end-to-end scenarios. In an SVC, the sequencer is generally responsible for operations including (i) the initialization of the DUV and the verification environment, (ii) the configuration of the DUV and the verification environment, and (iii) end-to-end scenario generation for the DUV verification. The *score boarding* is used to check for potential problems including (i) data values being different than expected, (ii) packets being received when not expected or (iii) a packets not being received when expected. The *coverage collection* is an SVC that focuses on collecting information including (i) the basic traffic of each interface, (ii) the combined effective traffic at all interfaces, (iii) the states of the internal design, (iv) the generated sequences, (v) delay and throughput information (performance information), (vi) the configuration modes, (vii) resets and restarts, and (viii) errors observed and errors injected.

3.7 Open Verification Methodology (OVM)

3.7.1 Introduction

The “Open Verification Methodology” (OVM) is the first language-interoperable open verification methodology that is based on the IEEE standard 1800^{TM}-2005 SystemVerilog Std. It provides a methodology and a supplementary library that allows users to develop modular and reusable verification environments. All components in verification environments interact with each other via standard transaction-level modeling (TLM) interfaces. OVM allows full integration with other commonly used languages. Following are the important features provided by OVM [20, 22].

- **Data Design:** OVM provides an infrastructure for class property abstracting and simplifies the user code by offering facilities for setting, getting and printing the property of variables.
- **Stimulus Generation:** OVM provides specialized classes and infrastructure to enable a fine-

grain control of sequential data streams for module-level and system-level stimulus generation.

- **Building and Running the Verification Environment:** The SystemVerilog OVM Class Library provides base classes for each functional aspect of a verification environment.
- **Coverage Model Design:** The incorporation of a coverage model into reusable open verification components (OVCs) is a good practic. OVM offers a coverage model design to implement coverage models efficiently.
- **Built-in Checking Support:** OVM provides a built-in checking support that is a good practice for incorporating the physical-layer and the functional layer checks into a reusable OVC.

3.7.2 OVM and Coverage Driven Verification (CDV)

OVM offers a framework to achieve a CDV that significantly reduces the time spent on the verification of a design by combining (i) automatic test generation, (ii) self-checking test benches and (iii) coverage metrics. The main purposes of CDV are (i) to reduce the effort and time spent in generating hundreds of tests, (ii) to ensure a thorough verification using up-front goal setting, (iii) to get early error notifications and (iv) to deploy a run-time checking and error analysis to simplify debugging. A CDV flow is different from a directed-testing flow. In CDV, an organized planning process is followed to set up the verification goals. Then a smart test bench is created to generate a legal stimuli and to send it to the DUV. The coverage monitors are added to the environment to measure the verification progress and to identify non-exercised functionality. Checkers are added in the verification environment to observe undesired behaviors of the DUV. After the implementation of a coverage model and a test bench, simulations are launched to achieve the verification. CDV supports directed-testing in addition to the constrained-random verification. Although, it is preferred that most of the work is done through the constrained-random testing before writing a time-consuming deterministic test. This test is used to activate specific scenarios that are very difficult to reach with random generation.

3.7.3 OVM Test bench and Environments

A test bench developed using OVM is comprised of reusable verification environments called OVM verification components (OVCs). An OVC is a configurable verification environment that is ready-to-use for an interface protocol, a module in a design, or a full system. Each OVC contains elements for simulation, protocol testing and coverage collection. To verify an implementation, an OVC is applied to the DUV. The OVC can be an MVC, an SVC or an IVC depending upon its nature of implementation. An OVC can be reused and configured according to its operational requirements. An OVC can be comprised of different types of elements according to its operational needs. These components are briefly described below. However, more details about these components can be found in the official OVM manual [22].

Data Item (Transaction)

A data item can be considered as an input transaction to the DUV in which its fields and characteristics are derived from its specification. For example, the Ethernet protocol specification identifies valid data values and attributes of an Ethernet packet (transaction). A number of meaningful tests can be generated by randomizing data items and sending them to the DUV.

Driver (BFM)

The driver is an active element in an OVC that implements the logic to drive a DUV. It repeatedly receives data items and sends them to the DUV by monitoring and driving DUV signals. For example, a driver emulating the logic to control the read/write signal, the data bus and the address bus for a READ transfer.

Sequencer

The sequencer is used to generate the stimulus for a DUV by controlling the generation of data items. It allows a constrained random generation of data items and on request it sends these data items to the driver. Each sequencer component has an associated sequence library where its associated sequences are stored.

Virtual Sequencer

The virtual sequencer is a special component in an OVM verification environment that is used to create multi-sided verification scenarios. It also synchronizes the timing and data between multiple interfaces. A virtual sequencer offers control over the verification environment for a specific test. It interacts with downstream sequencers and controls their execution of sub-sequences belonging to their sequence libraries. Hence, a virtual sequencer does not need to have a default sequence item type. An executed sub-sequence may belong to the local virtual sequencer's library or to the sequence libraries of any downstream sequencer (connected to the local virtual sequencer through a sequence interface).

Monitor

The monitor is a passive component that only samples DUV signals but does not drive them. Additionally, it performs checking and collects coverage information. The monitor is used to extract signal information over the bus or DUV interfaces. This information is then translated to data items (transactions) which are finally available for other components. There are two types of monitors: (i) agent monitors and (ii) bus monitors. The agent monitor is a local monitor of a specific agent that operates on signals and transactions related to this agent only. The bus monitor is responsible to handle all signals and transactions on the bus or DUV interfaces.

Agent

The agent is a container that is used to name, configure and interconnect a sequencer, a driver and a monitor component. To reduce the work for a test writer, OVM suggests the creation of more abstract containers e.g., multiple agents can be encapsulated within an OVC. An agent can be either a component that initiates transactions to the DUV or a component that reacts to transaction requests. An agent should be configurable to operate in an active mode or a passive mode (only monitors DUV activities).

Environment

The environment is a top-level component of the OVC. It may contain one or more agents and some other components e.g., bus monitors. It enables the customization of topology and behavior of including components through configuration properties. For example, an agent can be changed from an active mode to a passive mode, or a bus monitor can perform checking and collect the coverage information of activities which are not corresponding to any single specific agent.

3.8 OVM Class Library

OVM class library is a SystemVerilog based class library containing all essential components required to implement well-structured, configurable and re-usable verification components and verification environments. The library is consisted of base classes, macros and utilities. The verification environments are developed by hierarchically encapsulating and instantiating all components. These components are controlled by a set of phases to initialize, run and complete the tests. The base class library defines these phases. However, these phases can be extended to meet specific needs of a project. The library provides a vigorous set of built-in features that are needed for the verification e.g., print, copy, etc. Using the OVM library increases the code readability because each parent class predefines its component's functions in the library. Moreover, the class library provides a flexible environment construction (i.e., OVM factory) to facilitate the implementation of verification environments. More details about the OVM library and the OVM factory can be found in the official OVM manual [22].

3.8.1 Transaction-level Modeling (TLM)

All OVM components communicate with each other through standard TLM interfaces. This standard communication infrastructure improves the reusability of components. An OVM component implementing a TLM interface can interact via its interface with any other component that implements this interface. A set of transaction level communication interfaces and channels are provided by OVM for the transaction level interconnection between components. In this way, each component is isolated from changes in other components.

Transaction Level Communication

Transaction level interfaces define a set of methods that take transaction objects as arguments. A set of methods for a particular interface is defined by a TLM *port* while their implementation is provided by a TLM *export*. When a port is connected to an export, calling a port method results in the execution of its implementation provided by the export. OVM provides a *tlm_fifo* channel that enables components to operate independently. This channel implements all TLM interface methods necessary for such a communication. A producer puts the transaction into this FIFO channel while a consumer independently fetches this transaction. This channel provides both blocking and non-blocking interfaces. A special type of TLM communication is provided by OVM which is called *analysis communication*. This communication is for components such as monitors, which may need to generate a stream of transactions without taking care of its targets (whether there is any connected or not). This TLM communication is supported by providing the analysis port, the analysis export and the analysis fifo channel. More details about this communication can be found in the official OVM manual [22].

3.9 SystemC

SystemC is a C++ class library developed to support (i) system level designs, (ii) hardware architectures, (iii) cycle-accurate models for software algorithms, (iv) interfaces of SoCs and (v) executable specifications. The SystemC class library was standardized in 2005 as IEEE 1666™-2005. Currently version 2.2 of the library is currently available while version 3.0 will be available in near future. The later version will be extended to cover the modeling of operating systems and also to support the development of models of embedded software. Moreover, additional libraries can be provided to support a particular design methodology e.g., SystemC Verification Library (SCV). The Open SystemC Initiative (OSCI) developed the SystemC Class Library.

Following are the key features provided by SystemC:

- **sc_module:** It is a C++ class that is appropriate for defining hardware modules that contain parallel processes.
- A method of defining functions that model parallel threads of control within a `sc_module`.
- **sc_port and sc_export:** Two classes representing points of connection for a `sc_module`.
- **sc_interface:** This class tells about software services that are required by a `sc_port` class or provided by a `sc_export` class.
- **sc_prim_channel:** A class representing channel connections.
- **data types:** SystemC supports 2-state and 4-state logic data types.

Recently, the TLM group of OSCI issued the second version of Transaction Level Modeling, which defines an interface that is used to write high-level software models of hardware. More details about SystemC and OSCI TLM-2.0 can be found in their official manuals [23, 24].

3.10 SystemVerilog Direct Programming Interface (DPI)

3.10.1 Overview

This section gives a brief overview about the SystemVerilog DPI. Details about the DPI can be found in the official SystemVerilog manual [25].

SystemVerilog provides an interface to interact with foreign programming languages (such as C/C++). The interface is called SystemVerilog Direct Programming Interface (DPI) and it is particularly to facilitate C language. The DPI makes it much easier to call C functions within SystemVerilog and to call SystemVerilog functions within C. A big advantage of the DPI is the reusability of existing C code without having knowledge of SystemVerilog. By using the DPI, we only need to define a C linkage semantic, though the actual implantation of the foreign programming language remains transparent.

C functions can be called within SystemVerilog by using the `import` “DPI” declaration. These functions and tasks are called *imported* functions and tasks. It is necessary to declare every imported task and function. It is called *import declaration*. To call SystemVerilog functions and tasks within C, they must be specified in the `export` “DPI” declaration within SystemVerilog. These functions and tasks are called *exported* tasks and functions. All DPI functions are supposed to finish their execution instantly in zero simulation time. There is no synchronization mechanism provided by the DPI except data exchange and transfer of control. Only SystemVerilog data types can be passed between SystemVerilog and foreign languages through imported and exported function/task arguments and results. However, there are some restrictions on the notations of these data types. An example of an import declaration and an export declaration is given below. More details can be found in official manual of SystemVerilog [25].

- **`Import Declaration`**

  ```
   import "DPI–C" function int calc_parity (input int a);
  ```

- **`Export Declaration`**

  ```
  export "DPI–C" my_cfunction = function myfunction;
  ```

Chapter 4

Functional Verification of CPU Subsystem

4.1 Introduction

This chapter provides details about the functional verification of the Subsystem and its components. Section 4.2 describes the verification plan and the implemented test bench for the functional verification of the memory system. Section 4.3 gives details about the verification plan and the implemented test bench which was used for the functional verification of the triple-layer Sub-bus system. Section 4.4 outlines the verification plan for the functional verification of the OR1200 core. This section also describes the development of a golden model, which was used as a reference model for the verification of the OR1200 core and developed by using the ISS of this processor. Further, this section also describes the implementation of a SystemC wrapper around the ISS so that the golden model can incorporate the verification environment. The development of a SystemVerilog wrapper around the OR1200 (DUV) will be also described in this section. Section 4.5 describes the development of a verification environment for the functional verification of the OR1200 core by following the OVM. This section also outlines the architecture of this verification environment and describes the test bench (employed inside the verification environment).

4.2 Functional Verification of Memory System

4.2.1 Verification plan

We planned to use SystemVerilog for the constrained random verification of the memory system of the CPU Subsystem. As this memory system includes a ROM and a RAM memory with Wishbone interfaces. The ROM was implemented by using a RAM inside and both memories use same Wishbone interface. Therefore, the functional verification of the RAM (only) will be discussed here. However, we separately need to verify the ROM initialization with an Intel Hex file. The planning of a verification environment for the functional verification of this memory system includes the following key features.

- An *interface* for the structural connectivity between the test bench and the DUV.
- Development of a *bus functional model* complying the Wishbone specifications.

- Development of a *test library* containing the test cases.
- Development of a *test bench*.

These components are discussed below in detail.

Interface

SystemVerilog provides an `interface` construct for:

- modeling the communication between the functional blocks,
- the structural connectivity between the blocks,
- easier migration from the system level designs down to the RTL description and
- easier reusability of the designs by hiding the communication details.

The interface includes *modport* declarations to specify the directions of the ports for different blocks which can be connected through this interface. There are two types of modports specifying the port directions for the Master blocks and the Slave blocks. The interface also contains a *clocking block* for the cycle based semantics where the DUV signals (input and output) are sampled (registered before a clock edge) and synchronized at the clocking events.

Bus Functional Model

The implemented *bus functional model* (BFM) in this verification environment replicates the behavioral model of a Master component with a Wishbone interface. Therefore, this BFM can be used as a Master component complying the Wishbone specifications. This BFM includes a SystemVerilog interface (Section 4.2.1) for the cycle based communication with the memory system (ROM/RAM). This function model implements the following behaviors of a Wishbone Master component.

- **`Idle cycle`**: places an idle cycle.
- **`Single READ request`**: sends a single READ request for a given address, and receive the read data back.
- **`Single WRITE request`**: sends a single WRITE request by supplying the address and data.
- **`Block READ request`**: sends a block of READ requests by providing an array of addresses, and gets the received data array back.
- **`Block WRITE request`**: sends a block of WRITE requests by providing arrays of addresses and data, respectively.

This BFM drives the signals to the DUV and samples them through its interface which complies the Wishbone interconnection standard.

Test Library

This test library (`RAMTest`) is basically a SystemVerilog `program` which contains several types of planned tests for the functional verification of the memory system. The behaviors of the BFM (Section 4.2.1) are used to execute these tests. This library includes the following tests.

1. **`Sequential single WRITE/READ access test`**

   This test generates a sequential address, randomizes a data item and sends the WRITE request to the DUV[1] by using the "single WRITE request" behavior of the BFM. After the successful completion of the WRITE request this test sends a READ request for the same address by using the "single READ request" behavior of the BFM. Then it compares the written data with the received one. If the test passes, this whole procedure is repeated again for next sequential address until it covers the complete address space of the DUV.

2. **`Random single WRITE/READ access test`**

   This test is same as the "sequential single WRITE/READ access test" except that it generates random addresses (instead of sequential ones). It is to test the real-time scenarios where memory accesses are usually for random addresses. This test repeats the testing process for the number of times set by the user.

3. **`Random block WRITE/READ access test`**

   This test randomly generates arrays of addresses and data, and sends the block WRITE request to the DUV. It uses the "block WRITE request" behavior of the BFM to write this data array to the memory. After a successful completion of the block WRITE request this test sends a block READ request by using the "block READ request" behavior of the BFM. Same address array is supplied to this behavior which was used for writing data array. The received data array is compared with the written data array. If the test passes, the whole process is repeated again with new randomized arrays of addresses and data. This test repeats for the number of times set by the user, who can also set the length of the block.

4.2.2 Test Bench

`RAMTestbench` is the test bench used for the functional verification of the RAM (DUV), which instantiates all components, connects them together and drives the system clock and the system reset signals to these components. This test bench includes:

- a RAM component (DUV),
- a SystemVerilog interface,
- a bus functional model (BFM) and
- a test library (RAMTest).

[1] ROM is a read only memory, these tests are only for the RAM verification.

Figure [4.1] shows the architecture of the RAMTestbench. By considering the BFM as a virtual Wishbone Master component, the test library uses it to run different tests (e.g., Random block WRITE/READ access test) on the DUV for its exhaustive functional verification. The verification results of these tests will be given in Section (5.3).

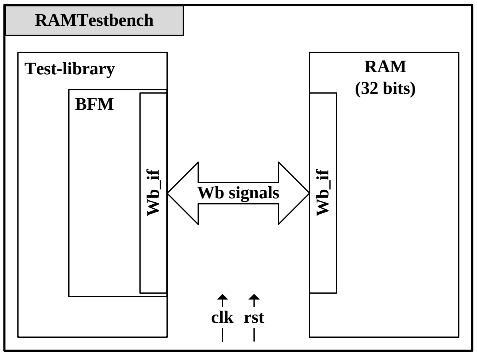


Figure 4.1 RAM Test bench.

4.3 Functional Verification of Triple-layer Sub-bus

4.3.1 Verification plan

We planned a CDV of the Sub-bus system using SystemVerilog based constrained random stimulus generation. Additionally, we decided to implement a coverage model to determine the verification closure. To verify the Sub-bus system we enhanced the existing verification actors those were used in the verification environment of the RAM component. All enhancements in the existing verification environment and new developments are given below.

- The development of a configurable test library (initiator) by enhancing the existing test library (RAMTest) used for the verification of the RAM.
- The implementation of a coverage model.
- The development of a test bench (Sub-bus-Testbench).

These components are discussed below in detail.

Test Library (initiator)

The Sub-bus system has three Master interfaces and four Slave interfaces. Hence, we planned to connect three BFMs (Master components) and four RAMs (Slave components) to these interfaces respectively. As the test library (initiator) uses these BFMs to drive different tests on the DUV, we needed to make this library configurable for each particular BFM. This test library (initiator) is an extension of the RAMTest library which is capable of driving only a single BFM and of generating the addresses for a complete given address space. Thereby, we needed to enhance the RAMTest library in such a way that it can support three BFMs and divide a given address space into four address spaces, one for each Slave component (RAMs).

To have a functioning system, we need a correct configuration of the Sub-bus itself and of the Master and the Slave components connected to it (see Section 2.8.3). To access a Slave component through the Sub-bus system, a Master component has to send an address that qualifies the address space of this Slave component. As the test library is responsible of generating the addresses for the BFMs, it was required to make it configurable so that it can generate the addresses within a specific sub-space of an address space. Hence, this test library (initiator) was required to divide the total address space of the Sub-bus system into four sub-spaces. The addresses generated for a Slave component include the slave-id in the MSBs (configurable). The address space of each connected Slave component is divided into three sub-spaces, one for each BFM. A BFM can only access this particular address space inside the memories. This subdivision of the Slaves' address spaces is necessary to handle the overlapping problem.

Example Figure [4.2] shows a 64 Kbytes RAM having total address space : 32'h00000000 ↦ 32'h0000FFFF. The accessible address space for the BFM-1 is: 32'h00000000 ↦ 32'h00005555, for BFM-2 it is: 32'h00005556 ↦ 32'h0000AAAA, and for BFM-3 the accessible address space is: 32'h0000AAAB ↦ 32'h0000FFFF.

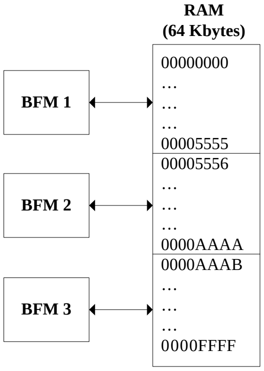


Figure 4.2 RAM address space subdivision.

We need three instances of this test library, which are configured to drive three BFMs. Each instance drives a single BFM and generates all addresses within the accessible address range of its BFM. However, each BFM can randomly access the connected Slaves components over the Sub-bus system. Each instance of the test library (initiator) can execute all tests which are provided by the RAMTest library (see Subsection 4.2.1).

Coverage Model

A *coverage model* was implemented to determine that the DUV has been exposed to a satisfactory variety of the stimuli and it is functioning correctly. We created a database of SystemVerilog bins to store a histogram of the addresses accessed by each BFM. We planned to cover the requested addresses by the BFMs and to cover them at the Slaves' sides too. In this way, we will be able to cross-verify how many times an address was accessed by a BFM and how many times a Slave component correctly responded the requests for this address.

4.3.2 Test bench

This test bench (`Sub-Bus Test-bench`) was used for the functional verification of the Sub-bus system. It instantiates all components those are required for the verification, correctly configures them, connects them together, and drives the system clock and the system reset signals to these components. The including components are

- a Sub-bus system,
- four RAM components,
- three interfaces,
- three bus functional models,
- three test libraries (one for each BFM) and
- a coverage model.

Figure [4.3] shows the architectural look of the test bench that was used for the functional verification of the Sub-bus system. The verification results are given in Section (5.4).

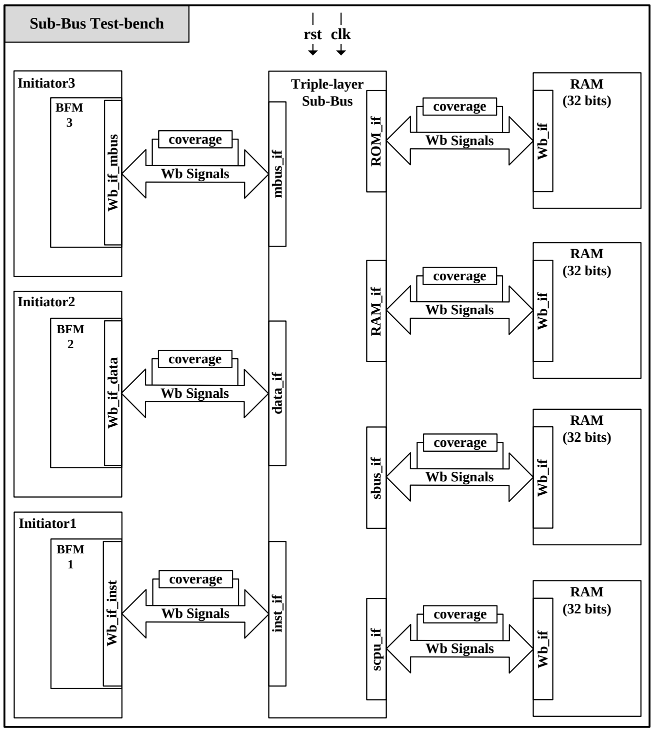


Figure 4.3 Test bench for triple-layer Sub-bus system.

4.4 Functional Verification of OR1200 Core

4.4.1 Verification plan

The functional verification of a heavily pipelined processor is a challenging task. We planned a simulation based verification of the OR1200 core by using the constrained random generation methodology. We planned a grey-box verification approach. Therefore, it was required to monitor the internal signals of the OR1200 core along with a *reference model* for the comparison. We used the architectural simulator (ISS) of the OR1200 core as its golden model. To develop a configurable and reusable verification environment, we planned to follow the OVM. As the verification environment uses SystemVerilog interfaces to communicate with the DUV (OR1200), we implemented a *SystemVerilog wrapper* around the OR1200 core. This wrapper provides interfaces to access the DUV. Details about these developments will be discussed later in this section. The following concerns are the most important to be taken into account while planning the functional verification of the OR1200 core.

- What to verify.
- When to verify.
- How to verify.

These points are discussed below in detail.

What to verify

Since the OR1200 core is a complex implementation and its verification is a challenge, we had to identify the most important aspects those must be verified. These aspects play a vital role in the correct execution of this processor. The correct working of these aspects verifies that the core is correctly operational. The aspects taken into account are listed below.

- Verify, if the OR1200 always generates a correct program counter (PC).
- Verify, if the OR1200 correctly updates its state in its supervision register (SR).
- Verify, if the OR1200 correctly saves its context in case of an exception (ESR/EEAR/EPCR).
- Verify, if the OR1200 always stores correct data to corresponding addresses in the data memory.
- Verify, if the OR1200 correctly stores the execution results in its general purpose registers.

When to verify

To identify the correct time to monitor the DUV's features one must have a thorough understanding of the core's architecture and its working, particularly about the instruction pipeline execution (Section 2.6.5). This task becomes more complicated when the exception model and the variable execution time of different instructions are taken into account. We also need to handle jumps/branches and delay slot executions. Another important side is to consider the freeze logic and flush-pipeline logic of the OR1200 core. These two logics vigorously control the OR1200 pipeline execution. However, all participants those are required to be monitored are `registers`. Hence, they all have enable signals for their update. These register enable signals identify the correct points to monitor these registers. However, along with these enable signals we also need to manage the pipeline's control logic, the exception control logic, the freeze logic and the flush-pipeline logic. Since these logics control the register enables. In pipeline execution, different pipeline stages may operate on different registers or may operate on different parts of a single register. Thus, identifying a correct execution stage to monitor a register is very important.

How to verify

We need a robust verification environment that feeds the instructions to the OR1200 core, handles Load/Store requests from the core, and correctly monitors the important registers of the core. Additionally, it is very important for an exhaustive verification to fill the complete instruction pipeline of the core and account the dependencies between the instructions. In this verification environment, an instruction is first sent to the golden model (ISS). After its execution the status of the golden model is obtained and stored. This instruction is then sent to the DUV and all important registers are monitored when this instruction updates them in different pipeline stages. These registers' values are compared with the status which was received from the ISS. The ISS executes every instruction in zero time while the OR1200 core is a hardware implementation (can be registered) having five instructions in its pipeline. Hence, this verification environment must include a synchronization mechanism between the golden model and the DUV. More details about this verification environment will be given in Section (4.5).

4.4.2 Instruction Set Simulator as a Reference Model

We used the Instruction Set Simulator (ISS) of the OR1200 core as a reference model for the functional verification of the core. This ISS is an architectural simulator, named **`Or1ksim`**, is a generic simulator capable of emulating OpenRISC1000 architecture based systems. It is an open source simulator that can be freely downloaded from the OpenCores. It provides high-level fast architectural simulation for an early code analysis and a performance analysis of systems. It supports all peripherals and system controller cores those are supported by OpenCores. The latest version (0.3.0) of the Or1ksim provides a network socket for remote debugging with a GNU debugger (GDB) support for different environments (OR1K processor model, memory configurations and sizes, configuration of peripheral devices). This new version also offers the choice to either use the simulator standalone or as a library. The new version also includes an OSCI TLM 2.0 interface. This ISS is written in C. Its standard configuration can model the main memory, the CPU, and a numbers of other peripherals [17, 18].

To use the ISS as a golden model, we needed to embed the existing ISS into a SystemC module. The module is also required to support the Direct Programming Interface (DPI) to incorporate the SystemVerilog based verification environment. The development involves several steps, as given below.

- Modify the existing Or1ksim (ISS) library to provide a set of public interfaces to access it.
- Define a SystemC module as a wrapper around this library that can access its public interfaces.

- Implement the DPI support inside this SystemC wrapper so that it can be integrated within the SystemVerilog based verification environment.

These all steps are individually discussed below.

Compiling Or1ksim Library

The installation of the OR1200 GNU toolchain package (Section 2.8.2) includes the Or1ksim simulator but it only works standalone for an early code analysis and a performance analysis of the system. In order to use this simulator as a reference model, we needed to use it as a library with a set of public interfaces to access it. This library can be configured to model OpenRISC1000 architecture based systems. The Ork1sim library, used in this project, was configured to model only the CPU and some generic peripherals. It does not model the main memory, the cache system, the memory management or other peripherals, since our DUV does not include such components. Similar to the standalone implementation of the simulator, a configuration file is used to configure the library to model different components in the system. The Or1ksim (0.3.0) offers the facility to use itself as a library, and provides two upcall functions to call up to the SystemC model of which it is part, to read and write from the peripheral address space. However, we needed to implement an additional upcall function to access the status of the ISS. Further, we also needed to implement a Direct Programming Interfaces (DPI) to access this library within the SystemVerilog based verification environment. These developments are discussed in the subsequent subsections. Here we will discuss how to compile the Or1ksim library with Questasim 6.5 on an Ubuntu 8.10 platform. The involved steps to compile this library are given below.

```
- Download the Or1ksim-0.3.0
- untar it
- Set environment variables:
    CC=/gcc-4.1.2-linux_x86_64/bin/gcc
    CXX=/gcc-4.1.2-linux_x86_64/bin/g++
     Note: The GCC for building the library has to be the same as the
         simulation uses.
- Make a directory Or1ksim-0.3.0, having the sub-directories: /source and /build
- Move the contents of the Or1ksim-0.3.0 to the Or1ksim-0.3.0/source
- Go to the Or1ksim-0.3.0/build
- Configure: ../source/configure --prefix=/home/.../or1ksim-0.3.0/install
        --enable-debug
- make all install
```

When the compilation is finished, we get a header file that defines the public interfaces to the Or1ksim library. This header file should in ../Or1ksim-0.3.0/install/include/or1ksim.h [23].

Using Or1ksim as a Library

In the standalone implementation of the Or1ksim, the `main` function initializes the ISS; after that it stays in a loop and executes the instructions. However, in the library implementation this `main` function is replaced by a series of functions those form the interfaces to the library. The header file (or1ksim.h) contains the declaration of these functions while their implementation is provided in the libtoplevel.c file. These functions are described below.

- **`or1ksim_init (...)`**

  This function initializes the simulator. It has several arguments those are given below.

  Config_file: This file provides the configuration data to the simulator.

  Image_file: This argument is used to pass the program image to the ISS. By default, the ISS takes the .ELF executable format of program images. However, it can also take the .IHex executable format. Since we want to fetch instructions and data from external test bench, an empty .ELF image will be passed in this argument. A sample empty ELF executable file has been given in Appendix (A.2.1) [26].

  To read or write from the peripheral address space the ISS needs to be able to call up to the SystemC model of which it is part. A standard implementation of the ISS library provides two upcall functions to read and write from the peripheral address spaces. These functions are defined by the (i) **upr** and (ii) **upw**, fourth and fifth function parameters of the `or1ksim_init()`. In the golden model we modified these upcall functions according to our requirement. In our implementation the ISS uses the “`upr`” upcall function to read the next instruction from the SystemC model. If this is a Load instruction the same upcall function is used again to read data. However, in case of a Store instruction the “`upw`” upcall function is used to write data up to the SystemC model. Since it was required to access the internal status of the ISS (after every instruction’s execution), a third upcall function (**upcpustatus**) was implemented in the ISS to write its status up to the SystemC model. This ISS status includes: (i) the PC register, (ii) the supervision register (SR), (iii) the exception supervision register (ESR), (iv) the exception program counter register (EPCR), (v) the exception effective address register (EEAR), (vi) all general purpose registers (GPRs) and (vii) the instruction that was just executed on the ISS.

  The implementation of these upcall functions is provided in the SystemC model (C++), while the ISS (C) can access them on demand. The function calls between C and C++ could be awkward. Therefore, upcalls were implemented as *static functions* in the SystemC model. The SystemC model calls the `or1ksim_init()`. To enable the upcall functions for invoking the member functions of this SystemC model a pointer (`class_ptr`) to this SystemC module instance is passed as an argument to these upcall functions. Third argument (`class_ptr`) is the pointer to the SystemC module class that initializes the simulator by calling the `or1ksim_init()`.

```
int or1ksim_init( const char *config_file,
                  const char *image_file,
                  void *class_ptr,
                  unsigned long int (*upr)( void *class_ptr, unsigned long int addr,
                                            unsigned long int mask),
                  void (*upw)( void *class_ptr, unsigned long int addr, unsigned long
                                    int mask, unsigned long int wdata),
                  void (*upcpustatus)( void *class_ptr, void *cpu_statusPtr));
```

Details about the implementation of the upcall functions in the SystemC model can be found in Section (4.4.3).

- **`or1ksim_run (...)`**

  This function is used to run the simulator for a specific period of time, passed in its argument (in seconds). The duration of -1 runs the simulator forever.

  ```
  int or1ksim_run( double duration );
  ```

Beside these functions the or1ksim.h header file also includes some other functions those are beyond the scope of this report. Details about these functions can be found in the official Or1ksim manual [17, 18].

Or1ksim Library and Generic peripherals

The library implementation of the Or1ksim makes provision for any additional peripheral to be implemented externally. Any access (READ/WRITE) to this peripheral's memory map generates the upcall to an external handler.

Generic is a new extension in the Or1ksim to model external peripherals [17, 18]. Any READ or WRITE access to the memory map of an implemented generic component generates an upcall. All peripherals of the Or1ksim are configured in a configuration file (*.cfg*). A new section generic is introduced in this file to describe the external peripherals. Multiple external peripherals can be described by multiple generic sections. Each generic section includes multiple parameters to specify an external peripheral.

```
section generic
    enabled         =          1
    baseaddr        = 0x00000000
    size            = 0x7FFFFFFF
    byte_enabled    =          1
    hw_enabled      =          1
    word_enabled    =          1
    name            = "Gen_dev1"
end
```

The parameters of a **`generic`** component are as given below.

- enabled = 0|1

  The option 1 is to enable and the option 0 is to disable this AT Attachment and AT Attachment Packet Interface (ATA/ATAPI). If you do not specify the value, default is 1 (enabled).

- baseaddr = value

  It is the starting address of this generic peripheral's memory map. Its default value is 0 (not a sensible value). The size of the memory mapped register space is controlled with a parameter i.e., size. It is described below.

- size = value

  This parameter controls the size of the generic peripheral's memory mapped space in bytes. Any access (READ/WRITE) from the ISS to this address space (baseaddr ↦ size-1) will be directed to the external interface (upcall). The value of this parameter should be in power of 2.

- name = “str”

  This string specifies the name of the generic peripheral.

- The generic peripheral can be configured to have support for byte, half-word and word accesses. If the value is 1 (default) the respective support is enabled.

  byte_enabled = 0|1

  hw_enabled = 0|1

  word_enabled = 0|1

Our requirement for the golden model is to generate the upcalls for a complete 32 bit address space (0x0000_0000 ↦ 0xFFFF_FFFF) which is byte-addressable. The maximum size that can be supported by a single generic peripheral is 0x7FFF_FFFF bytes. Hence, three generic peripherals were needed to cover the complete 32 bit address space. With this configuration the golden model (ISS) always generates the upcalls either to READ/WRITE data or to fetch a new instruction. The verification environment feeds the instructions and data to the ISS. A sample configuration file used in this implementation can be found in Appendix (A.2.2).

Modification in ISS

We implemented a third upcall function inside the ISS to write its status up to the SystemC model after the execution of every instruction. More details about this modification can be found in Appendix (A.2.3).

4.4.3 SystemC Wrapper around Reference Model

After modifying the Or1ksim (ISS) and generating the library, we needed to implement a SystemC wrapper around this library so that the reference model can incorporate the verification environment. The key features this SystemC wrapper was required to implement are as given below.

- Provide the implementation of the upcall functions (upr, upw, upcpustatus).
- Call the `or1ksim_init()` function and pass its arguments.
- Run the simulator forever by calling the `or1ksim_run()` function in a thread.
- Provide a Direct Programming Interface (DPI) for these upcall functions to be accessible in the verification environment.
- Implement a synchronization mechanism between the SystemC upcalls and the DPI functions.
- Handle the host machine’s byte order (little-endian/big-endian).
- Provide the implementation to qualify valid data bytes inside the data array by using the selection bits.
- Parse out the required status information of the ISS and make it available to the DPI functions.

Upcalls

Three static member functions were implemented in the SystemC wrapper to provide the implementation of the upcall functions of the Or1ksim library. These static functions take a pointer of the SystemC module's instance which starts the Or1ksim ISS. This pointer is provided as a third argument to the `or1ksim_int` function. Each static function calls inside another C++ class member function. This member function actually provides the implementation of its respective upcall function. When the ISS generates an upcall to its corresponding interface function, it simply calls this static function because its interface function is a pointer to a C++ static function. This static function calls a member function which actually implements the upcall.

The piece of code (given below) is taken from the implementation of this SystemC wrapper class. It gives insight about the upcalls' working in the wrapper. The `staticWriteUpCPUStatus` is a static function of the wrapper. Its pointer was passed to the upcall i.e., `upcpustatus` (in the `or1ksim_init`) for writing up the ISS status. When the ISS generates this upcall, since the `upcpustatus` is a pointer to the `staticWriteUpCPUStatus` function, the ISS in fact calls this static function which actually calls a C++ member function inside (`writeUpCPUStatus`). This member function parses the incoming status information of the ISS and makes it available to its respective DPI function.

```
/*===Access the CPU state after every instruction's exectution===*/
void or1k_sc_module_dpi::staticWriteUpCPUStatus( void *instancePtr, void *cpu_statusPtr){
        or1k_sc_module_dpi *classPtr = (or1k_sc_module_dpi *)instancePtr;
        cpu_state_up *cpu_state_up_ptr = (cpu_state_up *)cpu_statusPtr;
        classPtr->writeUpCPUStatus(cpu_state_up_ptr);
} // staticWriteUpCPUStatus()
```

The definition of the `staticWriteUpCPUStatus` function is given below.

```
static void staticWriteUpCPUStatus(void *instancePtr, void *cpu_statusPtr);
```

Or1ksim_init Initialization

The `or1ksim_init` library function is called within the SystemC wrapper to initialize the ISS. A configuration file, an empty ELF file, the wrapper's own pointer (itself starting the ISS) and the pointers to its static functions are passed as arguments to this library function, as given below.

```
  or1ksim_init( "../simple.cfg",
                "../empty_elf",
                this,
                staticReadUpcall,
                staticWriteUpcall,
                staticWriteUpCPUStatus
  );
```

Direct Programming Interface

As we have discussed in Section (3.10), the implementation of the "*imported DPI functions*" is provided in a SystemC model and it is imported inside a SystemVerilog model by using the **import "DPI"** declaration. On the other hand, the implementation of the "*exported DPI functions*" is provided in a SystemVerilog model and it is exported to a SystemC model by using the **export "DPI"** declaration. In this verification environment we only need imported DPI functions to be called within a SystemVerilog based test bench while their implementation is provided inside the SystemC wrapper of the golden model. We implemented three imported DPI functions in the SystemC wrapper respective to three upcall functions. The hooked-up *member functions*[2] of the SystemC wrapper take data and instructions from these imported DPI functions and feed the ISS with this data and instructions. These member functions also make the simulator's status and data available to these imported DPI function so that it can be sent to the test bench. The definition of these three imported DPI functions in the SystemC wrapper is given below. The implementation of these imported DPI functions is not provided in this report.

```
int sv_readUp(const int rinsn, const int rdata, int *read_addr, int *read_addr_mask);
int sv_writeUp(int *waddr, int *wdata);
int sv_writeStatusUp( cpu_state_ref *iss_status);
```

All imported DPI functions must be registered in the SystemC module by using SC_DPI_REGISTER_CPP_MEMBER_FUNCTION().

```
SC_DPI_REGISTER_CPP_MEMBER_FUNCTION("sv_readUp", &or1k_sc_module_dpi::sv_readUp);
SC_DPI_REGISTER_CPP_MEMBER_FUNCTION("sv_writeUp", &or1k_sc_module_dpi::sv_writeUp);
SC_DPI_REGISTER_CPP_MEMBER_FUNCTION("sv_writeStatusUp",
                                    &or1k_sc_module_dpi::sv_writeStatusUp);
```

All imported DPI functions must be declared in the SystemVerilog environment, as given below.

```
import "DPI-SC" context task sv_readUp(input int rinsn, input int rdata,
                                       output int read_addr, output read_addr_mask);
import "DPI-SC" context task sv_writeUp(output int waddr, output int wdata);
import "DPI-SC" context task sv_writeStatusUp(output iss_cpu_status iss_status);
```

The DPI identifies an imported function by its name only (not by its parameters). Hence, only one copy of overloaded functions can be supported [27].

Note: The composite data types (e.g., structure/union) being transferred through the DPI from SystemC to SystemVerilog (or opposite) make provision for each element to be 32-bit aligned. For example, if a structure contains a char data type (8 bits), 24 bits should be padded to it to make it 32-bit aligned.

[2] The functions which provide the actual implementation of the upcalls.

Golden Model Synchronization

When the ISS starts the execution it fetches the first instruction through an upcall function (`upr`) from the reset address (0x0000_0100). It executes the instruction in zero time and comes up again to fetch the next instruction. As the ISS is running forever in a SystemC thread, it will never give the control to any other process if there is no mechanism to block it. We implemented a SystemC FIFO based mechanism with blocking READ/WRITE to synchronize the system. Four FIFOs of a single element depth were implemented between the hooked-up member functions and the imported DPI functions. With this strategy, when the ISS upcalls to fetch a new instruction it writes the PC address to the pc-fifo and is blocked until the instruction is available in the read-fifo. If this instruction is a Store, the ISS makes an upcall to write data up and it is blocked until the write-fifo is empty. However, if this instruction is a Load, the ISS upcalls to read data and it is blocked until data is present in the read-fifo[3]. After completing the execution of an instruction the ISS upcalls to write its status up and it is blocked until the status-fifo is empty. When the ISS is blocked the control is transferred to other running processes. On the other ends of these FIFOs the test bench uses the imported DPI functions to feed the instructions and data to the ISS to read data and addresses (for the Store instructions) and to get the status of the ISS after the execution of every instruction.

Golden Model Architecture

Figure [4.4] shows the architecture of the *golden model*. The ISS accesses the wrapper functions through its upcalls. The communication between the ISS and the test bench is synchronized by means of SystemC FIFOs. Test bench implemented in SystemVerilog (OVM) accesses these FIFOs through the imported DPI functions.

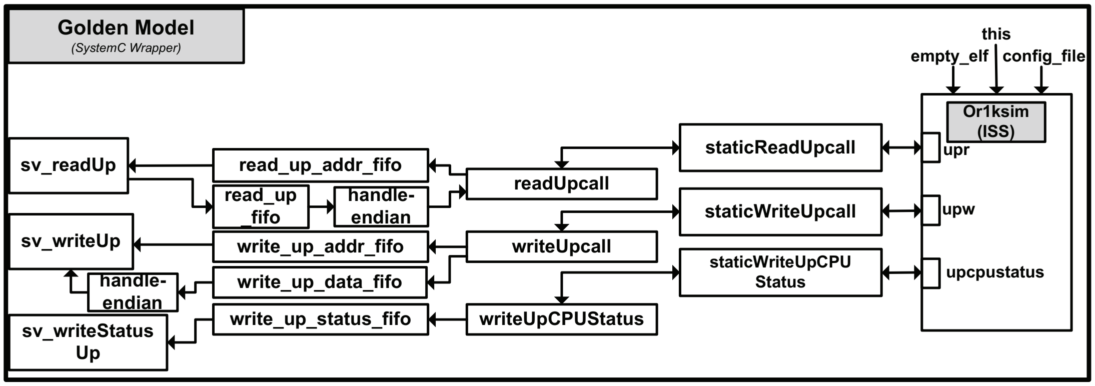


Figure 4.4 Golden model for the verification of the OR1200 core.

How to Compile SystemC Wrapper under Questasim 6.5

To compile the SystemC module we need to include directories which contain (i) header files of the Or1ksim library, (ii) the SystemC library and (iii) the TLM 2.0 library (if used) [23,27]. The option “-DMTI_BIND_SC_MEMBER_FUNCTION” is necessary when compiling a SystemC source file

[3] The read-fifo is used to fetch new instructions plus to read data for the Load instructions.

which uses "SC_DPI_REGISTER_CPP_MEMBER_FUNCTION" to register its member functions as DPI functions (Section 4.4.3). The option "-DSC_INCLUDE_DYNAMIC_PROCESSES" is essential if the TLM 2.0 is used in the model. An example piece of code for compiling a SystemC module (or1k_sc_module_dpi) under Questasim is given below. The directories *sysc_models* and *include* contain SystemC library and header files of the Or1ksim library respectively.

```
 sccom -vv -I ../../sc_golden_model_or1200/sysc_models/
           -I ../../sc_golden_model_or1200/or1ksim-0.3.0/install/include
           -DSC_INCLUDE_DYNAMIC_PROCESSES -g -DMTI_BIND_SC_MEMBER_FUNCTION
            ../../sc_golden_model_or1200/sysc_models/or1k_sc_module_dpi.cpp
```

The SystemC library must be included in the final linking. The linker needs to be directed to find the shared objects needed to compile the Or1ksim library by using these linker's options:
(-Wl, --R, $OR1KSIM_HOME), as shown below.

```
sccom -vv -L../../sc_golden_model_or1200/or1ksim-0.3.0/install/lib -Wl,-R,
             ../../sc_golden_model_or1200/or1ksim-0.3.0/install/lib -lsim -link
```

4.4.4 SystemVerilog Wrapper around OR1200 Core

A SystemVerilog based wrapper was implemented around the OR1200 core (DUV) which includes three SystemVerilog interfaces named as: (i) the *insn-if*, (ii) the *data-if* and (iii) the *status-if*. These are used to access the instruction Wishbone interface, data Wishbone interface and the internal signals of the core respectively. The status-if of this wrapper makes all required internal signals of the OR1200 available at its ports. The internal signals include the status registers (to be monitored) and the control signals (to control the monitoring). The status registers include (i) some important SPRs, (ii) all GPRs and (iii) the program counter (PC). This wrapper also implements a translation block to translate the OR1200's internal signals to a usable form e.g., the GPRs are implemented as a dual-port synchronous memory and their translation to thirty two 32-bit registers is needed. Further, this wrapper also implements a control block to manipulate the internal control signals according to the requirements e.g., delay a control signal for two clock cycles. All components of the verification environment interact with the DUV only through the wrapper's interfaces. The architecture of this wrapper is shown in Figure [4.5].

4.5 Verification Environment for OR1200 Core

4.5.1 Description

We used the OVM to implement a reconfigurable and reusable verification environment for the *simulation based verification* of the OR1200 core. We did a *constrained random generation* of the verification scenarios. We implemented a vibrant coverage model and a scoreboard to assess the verification completeness. Figure [4.6] elaborates the architecture of a verification environment (or1200_tb_top) which was developed by applying OVM and was used for the functional verification of the OR1200 core. This verification environment includes

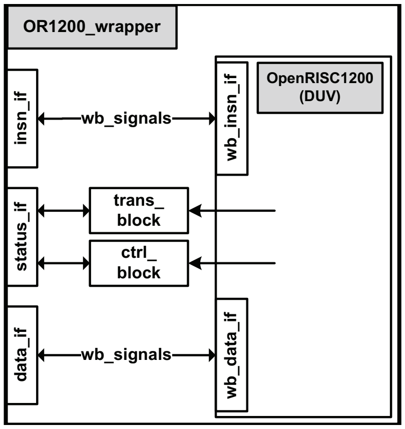


Figure 4.5 SystemVerlog wrapper around the OR1200 core.

- the golden model,
- the DUV wrapper (`or1200_wrapper`),
- the main test bench component (`or1200_tb`),
- the global package (`sv_sc_package`) and
- the test library (`or1200_tb_test_example_inst`).

The golden model and the DUV wrapper have been described in the previous sections. The rest of the components will be described in this section. The main test bench (main TB) is a reconfigurable and reusable component which was developed by following OVM. It interacts with the golden model through its imported DPI functions and uses its physical interfaces to interact with the DUV wrapper. The main TB executes the configurable tests of the test library where all tests are constrained random generation of the scenarios which are comprised of OR1200 instructions. The main TB first sends an instruction to the golden model, writes/reads data (if the instruction is Load or a Store) and receives the ISS status once the instruction has been executed. Further, it sends this instruction to the DUV. While this instruction passes through different pipeline stages in the DUV the main TB keeps eye on the state of the DUV and reacts accordingly. It examines the control state machine of the DUV along with the data-path. The main TB monitors the control signals of the DUV to determine the right time to examine the status of the DUV (e.g., PC, SPRs, etc.) and the execution results (GPRs). It compares the status of the golden model with the DUV status and scoreboards it. The main TB also implements a coverage model to assess the completeness of the verification. Most of the components of the verification environment can be configured according to implementation's requirements. For example, (i) the coverage model or the scoreboard should be implemented or not, (ii) an agent component will operate as a passive component, and (iii) which tests of the test library will be executed.

All components of this verification environment are described in the following subsections.

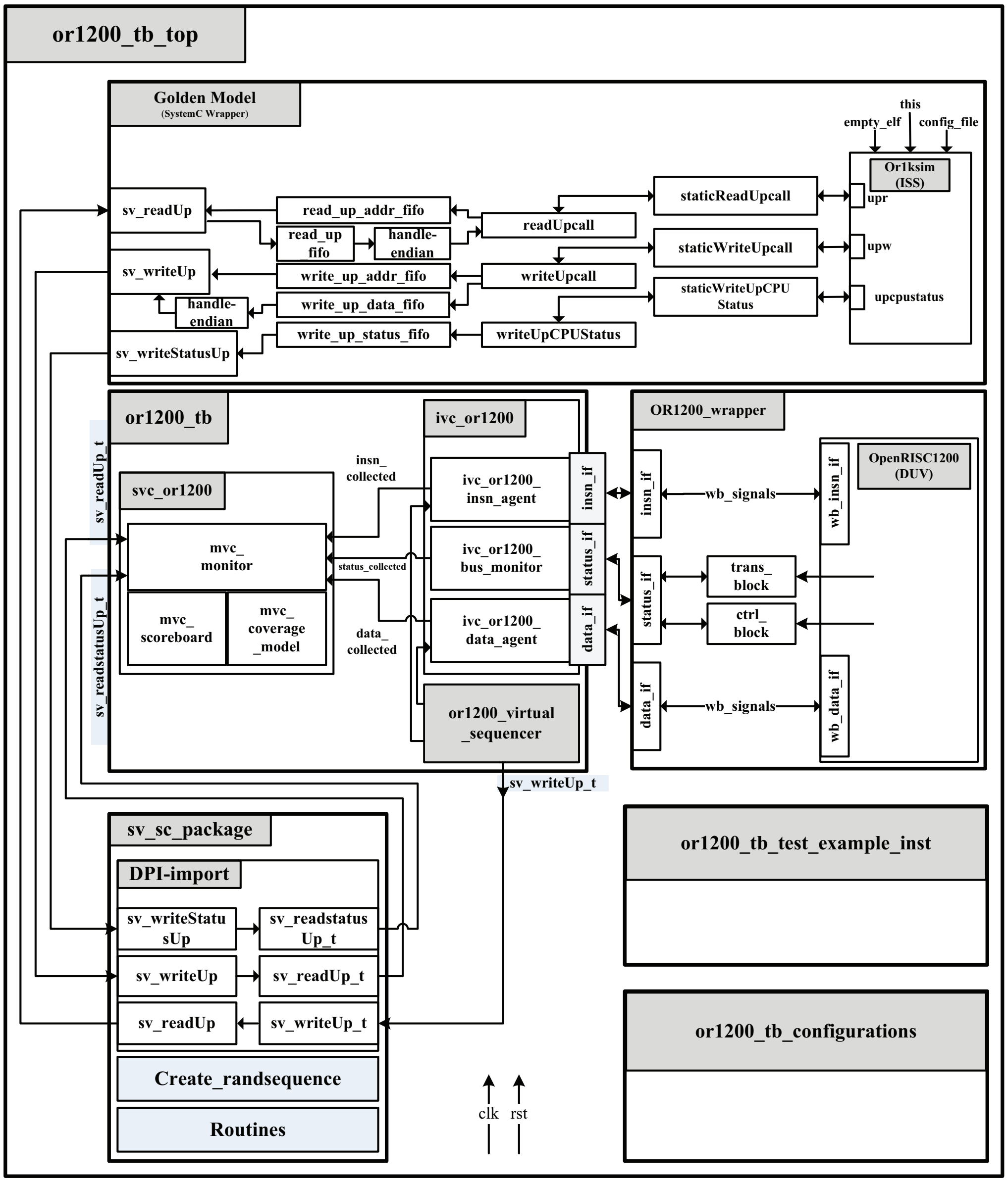


Figure 4.6 Verification environment for the OR1200 core.

4.5.2 Main Test Bench for OR1200 Core

Figure [4.7] shows the structural design of the main TB used for the functional verification of the OR1200 core. It is comprised of three main components:

- the interface verification component (`ivc_or1200`),
- the system/module verification component (`svc_or1200`) and
- the virtual sequencer (`or1200_virtual_sequencer`).

All components inside the main TB interact with each other through standard TLM interfaces.

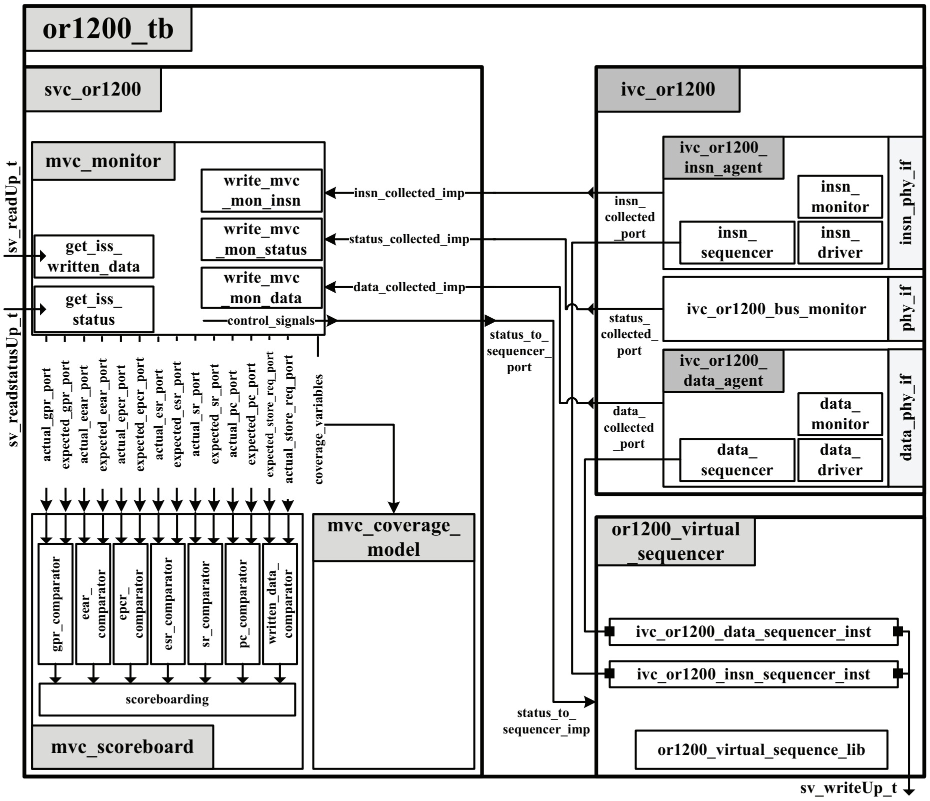


Figure 4.7 Main Test bench for the verification of the OR1200 core.

Interface Verification Component (IVC)

The main TB interacts with the DUV (OR1200) through its interface verification component. This IVC includes (i) three physical interfaces (instruction, status, data), (ii) an instruction agent, (iii) a data agent and (iv) a bus monitor. The instruction, status and data interfaces of the IVC are respectively connected to the instruction, status and data interfaces of the DUV wrapper. The other side of the instruction, status and data interface is respectively connected to the instruction agent, the bus monitor and the data agent of the IVC. The instruction interface is used to send the instructions to the DUV. The status interface is used to read the internal status registers and the control signals of the DUV. The data interface is used to send or receive data of Load or Store accesses from the DUV. Figure [4.8] shows a detailed view of this IVC.

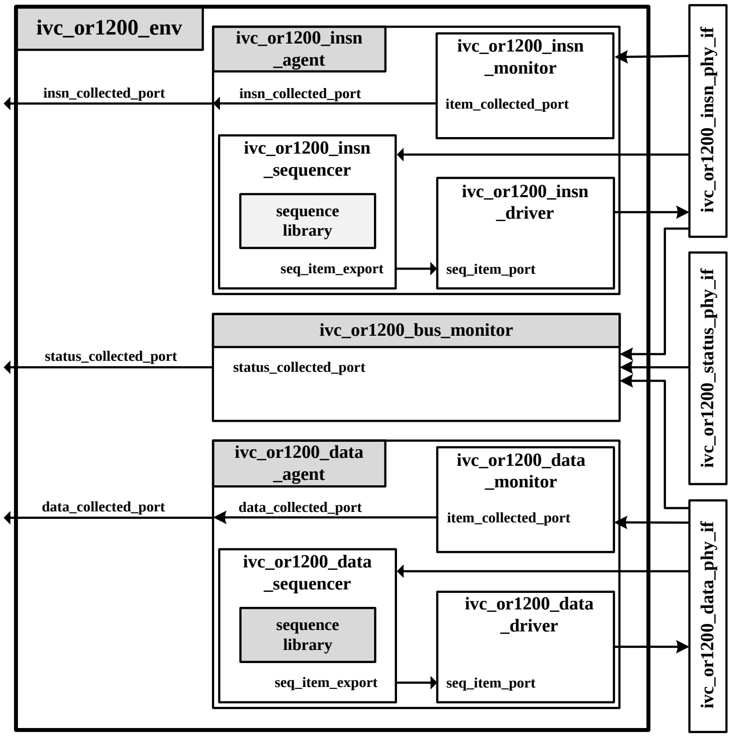


Figure 4.8 Interface verification component.

Physical Interfaces

These interfaces provide the port-level connection to the DUV interfaces and the helper functions for the IVC to read or write the values on these ports. These interfaces implement the Wishbone protocol checking using *concurrent assertions* e.g., the *ack* and the *err* signals must not be asserted together. These *concurrent assertions* are checked throughout the simulation to ensure that the interconnection protocol is always obeyed.

Instruction Agent

This instruction agent contains (i) an instruction driver, (ii) an instruction monitor and (iii) an instruction sequencer. It operates as a Slave component which is connected to the instruction Wishbone interface of the OR1200. On receiving a request from the core, its instruction driver requests a new transaction (instruction) from the instruction sequencer and sends it to the DUV over the instruction interface (`ivc_or1200_insn_phy_if`) by using its helper functions. These transactions are required to be translated to the port level signals. The instruction driver follows the Wishbone interconnection standard. It synchronously asserts the termination signal (i.e., ack, err, rty) for one clock cycle after each request from the DUV. The instruction monitor only reads (does not drive) the signals of the instruction interface when the instruction driver acknowledges a request. After reading the interface signals by using helper functions, this instruction monitor translates them to an instruction transaction and sends this transaction to the *system verification component*, over a TLM port (`insn_collected_port`). An instruction transaction encloses the instruction that is sent to the DUV and the address of this instruction. The instruction driver requests a new instruction from the instruction sequencer. It sends the next transaction (instruction) in the sequence to the driver. These sequences are a constrained random generation of ORBIS32 instructions. The instruction sequencer contains a library which encloses several sequences of instructions those can be generated on demand.

`Data Agent`

The data agent contains (i) a data driver, (ii) a data monitor and (iii) a data sequencer. It operates as a Slave component which is connected to the data Wishbone interface of the OR1200. On receiving a READ request from the DUV, its data driver requests a new transaction (a data item) from the data sequencer and sends it to the DUV over the data interface (`ivc_or1200_data_phy_if`) by using its helper functions. These transactions are required to be translated to the port level signals. The data driver follows the Wishbone standard. It asserts the synchronous termination signal (ack, err, rty) for READ requests while asserting asynchronous termination signal for WRITE requests. These termination signals are asserted for one clock cycle. The data monitor only reads (does not drive) the signals of the data interface when the data driver acknowledges a request. After reading the interface signals, it translates them to a data transaction and sends this transaction to the system verification component over a TLM port (`data_collected_port`). This data transaction encloses the address and the data item along with the write enable (we_i) and the byte select (sel_i) Wishbone signals. On the data driver's request, the data sequencer sends a new transaction (a data item) to the driver. The data sequencer contains a library which encloses several data sequences.

`Bus Monitor`

The bus monitor is used to access the internal control signals and the status registers of the DUV through the status interface of the IVC. It can also access the instruction and data interfaces. This bus monitor reads the OR1200 status signals every cycle, translates them to a status transaction and sends the transaction to the system verification component over a TLM port (`status_collected_port`). This status transaction is comprised of (i) the PC register, (ii) the SR, (iii) the ESR, (iii) the EPCR, (iv) the EEAR, (v) all GPRs and (vi) some important control signals of the OR1200 e.g., pc_we, esr_we, except_start, etc.

System Verification Component (SVC)

The focus of the system verification component is to test the end-to-end behavior of the OR1200 core. This SVC is one step higher at abstraction level than the IVC. It is comprised of the following components:

- the module monitor (`mvc_monitor`),
- the scoreboard (`mvc_scoreboard`) and
- the coverage model (`mvc_coverage_model`).

These components are explained below.

Module Monitor

This module monitor, shown in Figure [4.9], collects the transactions (instruction/data/status) sent from the IVC. It interacts with the golden model to read its status and data along with the store address (in case of Stores). It accesses the golden model by accessing the DPI functions (`sv_writeStatusUp, sv_writeUp`) through the local SystemVerilog tasks (`sv_readstatusUp_t, sv_readUp_t`) respectively. The golden model executes every instruction in zero time while the OR1200 is a 5-stage pipeline processor. Therefore, a synchronization mechanism must be implemented to correctly compare their status and results. This mechanism was implemented in the module monitor using SystemVerilog FIFOs where the depth of each FIFO is four elements. The module monitor receives information from the golden model and stores it into the corresponding FIFO (e.g., SR to SR-fifo, PC to PC-fifo). The main test bench keeps on sending the instructions to the ISS first and then to the DUV. The module monitor keeps on filling its FIFOs by receiving the status and results from the ISS. These FIFOs are full by the time the first instruction executes on the DUV (in the execution pipeline stage). The module monitor takes the status information of the ISS from the top of the FIFOs, parses out the status of the DUV from the transactions (status/data) received from the IVC and sends both information to the scoreboard. The control block (`ctrl`) implements an interactive control logic to monitor the control state machine of the DUV and react accordingly to decide the right time of comparison between the ISS and the DUV statistics. This control block also sends a few control signals (e.g., except_start) to the *virtual sequencer* which are needed for the reactive scenario generation.

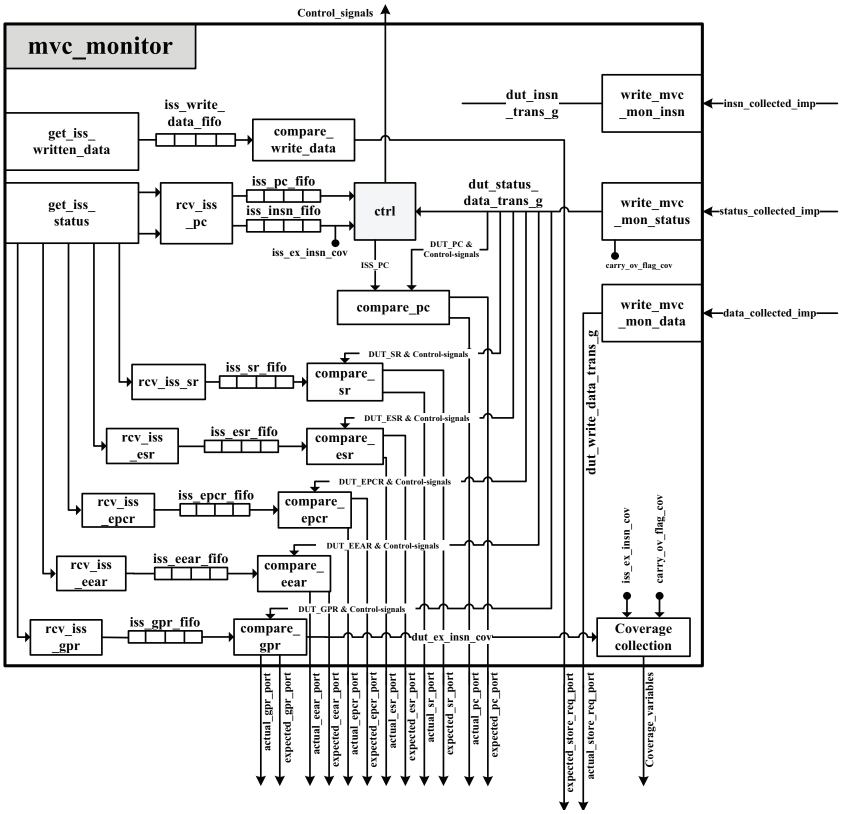


Figure 4.9 Module monitor.

To collect the verification coverage the module monitor sends the instructions those are executed on the golden model and on the OR1200 core to an implemented coverage model. Additionally, it sends a few status flags of the OR1200 core which are essential for a satisfactory coverage collection. These flags include (i) the *carry flag*, (ii) the *overflow flag* and (iii) the *conditional branch flag*.

Scoreboard

The scoreboard receives the status registers and data along with the address (for Stores) from the module monitor through standard TLM ports. It receives the status of the golden model (`expected_*_port`) and the status of the DUV (`actual_*_port`). It implements an individual comparator for each stakeholder in the status and data transactions e.g., PC, SR, address to store data, etc. After comparison the scoreboarding is executed to generate the final report for each stakeholder.

Coverage Model

To assess the verification closure, a coverage model was implemented to determine that the DUV has been exposed to a satisfactory variety of stimulis. We created a database of "SystemVerilog coverage points" to generate a histogram of instructions those have been executed on the ISS and on the DUV. This coverage model creates a database on the bases of the following key features.

- The total number of instructions being executed on the ISS.
- The total number of instructions being executed on the DUV.
- The type of instructions being executed on the ISS.
- The type of instructions being executed on the DUV.
- Verify that every instruction reads or writes to all its legal operand e.g., (i) ADD uses all 32 GPRs as source1, source2 and destination. (ii) A Jump instruction takes all legal immediate values.
- Verify that each instruction which can modify the status flags (carry, overflow, and branch flag) properly sets and clears the corresponding flags. For example, ADD correctly sets and resets the carry and overflow flags.
- The coverage of 32-bit address space through the OR1200's program counter (PC) register.
- The cross coverage of three consecutive instructions in the OR1200 pipeline to observe the dependencies between the instructions.

Virtual Sequencer

The verification environment contains a virtual sequencer to synchronize the timing and data between (i) the golden model (ISS), (ii) the instruction interface and (iii) the data interface. The instruction sequencer generates sequences of instructions. The data sequencer generates sequences of data. There is no co-ordination between these sequencers. This co-ordination is necessary to control the sequence generation on the instruction and data interfaces. Moreover, we need to send the instruction and data transactions to the golden model first and then to the OR1200 core (DUV). Therefore, a controlling body must be implemented at a higher level to allow the fine control of the verification environment for a particular test. The virtual sequencer contains the instances of the instruction sequencer and the data sequencer along with a virtual sequence library. This library encloses the virtual sequences which are executed on the virtual sequencer and control the coordination between the instruction sequencer, the data sequencer and the golden model. The virtual sequences are a constrained random generation of the scenarios (a sequence of instruction types e.g., ADD, MUL, etc.). When the OR1200 sends an instruction fetch request the virtual sequencer picks the next instruction in the sequence (e.g., ADD) and generates its constrained random transaction (binary code of ADD instruction e.g., 0xe0841800). The transaction is generated on the instruction sequencer by using the local sequence library of the instruction sequencer. Before sending this transaction to the instruction driver the virtual sequencer first sends it to the golden model. If this instruction is a Load, the virtual sequencer also provides a randomized data to the golden model. The golden model finishes execution and sends the status and result back to the module monitor. After this the virtual sequencer allows the instruction sequencer to send this instruction's transaction to the instruction driver. If the instruction is a Load, the virtual sequencer uses the same data sent to the golden model and generates a constrained data transaction on the data sequencer by using its local sequence library (data sequence library). When this instruction is executed on the DUV and sends a READ request, this data transaction is sent to the data driver. The virtual sequencer also implements a complex mechanism to offer interactive behavior by using control signals of the DUV received from the module monitor. One instance of this mechanism is to stop sending instructions to the golden model (sending instructions to the DUV never stops) if an exception has been signaled in the OR1200 pipeline. It is because the OR1200 instruction pipeline is flushed and following instructions will never be executed. Whereas, the golden model (ISS) executes instructions at once in zero simulation time as we feed it instruction before sending to the OR1200 core (DUV).

Chapter 5

Results

5.1 Introduction

This chapter summarizes the results obtained from the simulation of the CPU Subsystem. It presents the functional verification results of the memory system and the Sub-bus system. Further, this chapter presents the verification results of the OR1200 core obtained by applying the verification environment developed for the functional verification of the OR1200 core (see Section 4.4). These results can be divided into three categories, as given below.

1. **Errors:** All errors found in the OR1200 core.
2. **Discrepancies:** All found discrepancies between the OR1200 core and its instruction set simulator (used as a golden model).
3. **Coverage results:** The verification coverage results achieved from the functional verification of the OR1200 core.

5.2 CPU Subsystem Simulations Results

5.2.1 Overview

After interconnecting all components of the Subsystem, we run a test program on it to see its basic functional correctness. For this purpose, the memory initialization file (IHex) of the test program is first generated using the OR1200 Tool chain and then loaded into the ROM of the CPU Subsystem. The Subsystem is required to execute this binary encoded file correctly. The test program is also executed on the OR1200 ISS to get execution results (given below) for cross-testing.

Test program's execution result on the ISS = **0x0037_5F00**.

The used test program is given in Appendix (A.1.1). Additionally, a copy of its disassembly file is given in Appendix (A.1.2). Some important execution results of the application program are presented in the following subsection.

5.2.2 Execution Results

The following are the most important aspects while executing an application program on the Subsystem.

1. The Subsystem should fetch correct instructions from correct addresses inside the ROM.
2. The Subsystem should calculate the correct execution result and store it back to the memory.

Correct Instruction Fetch

After receiving the reset signal, the CPU Subsystem should fetch the first instruction correctly from its reset address (0x0000_0100) inside the ROM. Figure [5.1] shows a waveform of the OR1200 instruction interface (Wishbone) that fetches instructions from the ROM and feeds them to the CPU. In this waveform the reset signal is de-asserted at time 100 ns. The OR1200 sends its first READ request by asserting the iwb_cyc_o and iwb_stb_o signals at time 115 ns from the address 0x0000_0100. It gets the instruction 0x1820_F000 back (at time 125 ns) that is stored inside the ROM at the address 0x0000_0100 (see Section 2.14). This instruction (`l.movhi` = 0x1820_F000) is the first instruction in the disassembly of the test program (see Appendix A.1.2). The OR1200 core then keeps on fetching and executing new instructions correctly.

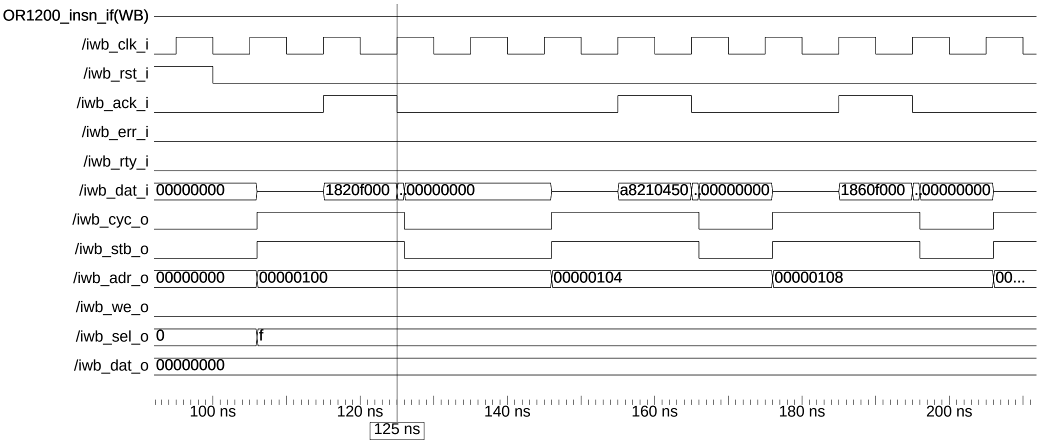


Figure 5.1 CPU Subsystem's correct instruction fetch.

Correct Execution Result

Figure [5.2] shows the waveform of the instruction and data interfaces of the OR1200 core. At time 17765 ns a “load word zero” instruction (`l.lwz` = 0x8482_FFF4) is fed into the processor. At time 17795 ns a “store word” instruction (`l.sw` = 0xD7E2_27FC) is fed into the processor. At time 17815 ns the instruction `l.lwz` loads the calculated result (0x0037_5F00) of the test program from the stack (over the data interface). At time 17845 ns the instruction `l.sw` stores the result to the memory (over the data interface). The stored result is **0x0037_5F00** which is the correct result calculated from the ISS.

The waveform also confirms our use of synchronous termination (ack, err, rty) for READ transfers and asynchronous termination for WRITE transfers. This is the solution to get the maximum throughput when using the Wishbone standard (see Section 2.3.3).

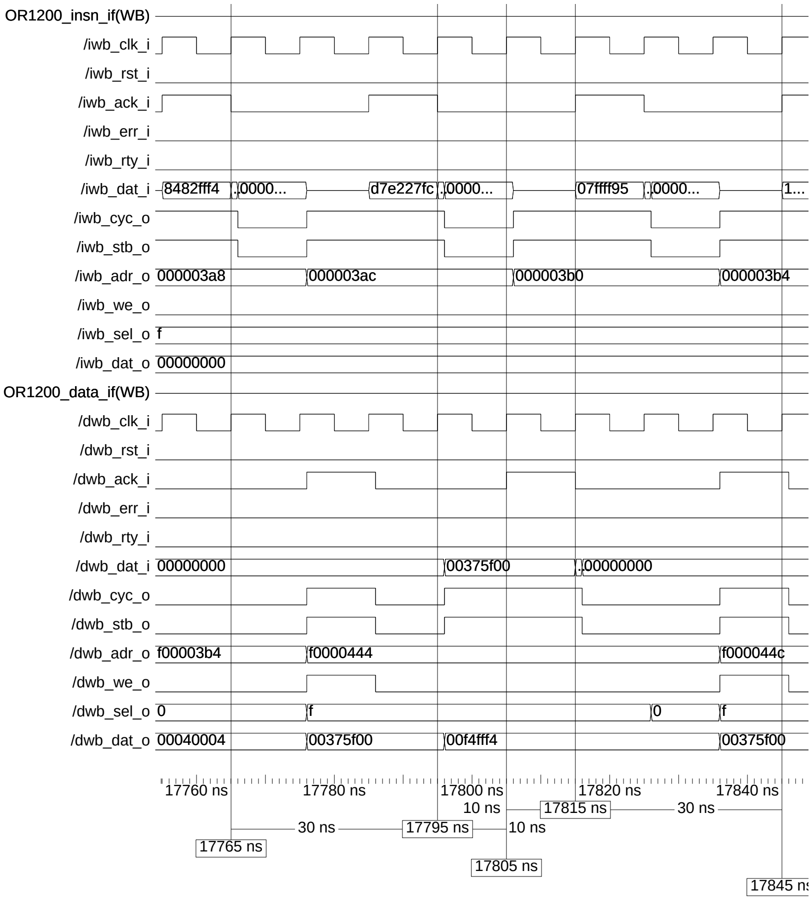


Figure 5.2 CPU Subsystem's execution result.

5.2.3 Maximum Throughput Results

As discussed in Section (2.7), the maximum achievable throughput of the CPU Subsystem is three clock cycles per instruction. Figure [5.3] shows a waveform of the instruction interface of the OR1200 core. It confirms that most of the instructions are fed into the core after every three clock cycles (i.e., 30 ns). However, the LOAD/STORE instructions need more time to execute on the CPU Subsystem. The LOAD instructions take five clock cycles (i.e., 50 ns) because of the synchronous termination and STORE instructions take four clock cycles (i.e., 40 ns) because of the asynchronous termination.

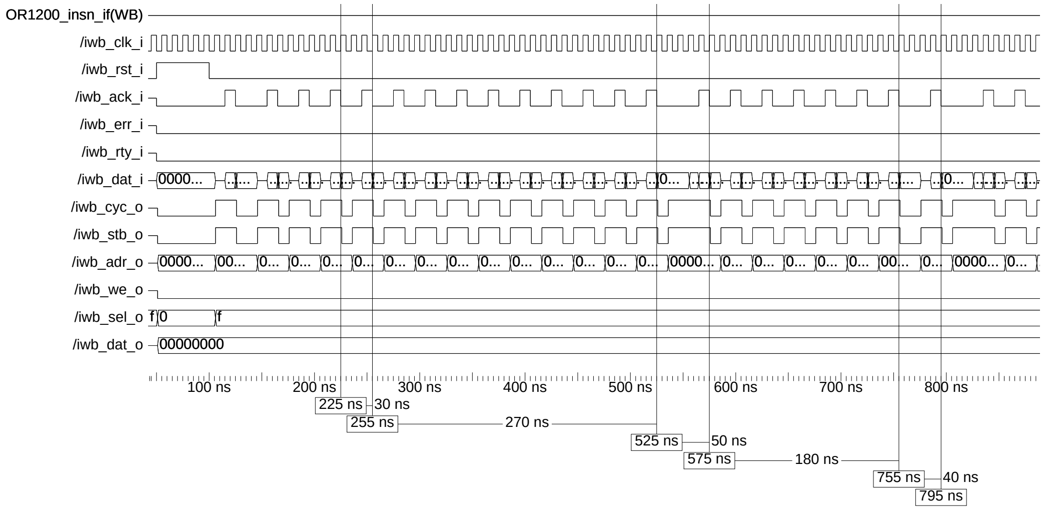


Figure 5.3 CPU Subsystem's maximum throughput results.

5.3 Memory System Verification Results

5.3.1 Overview

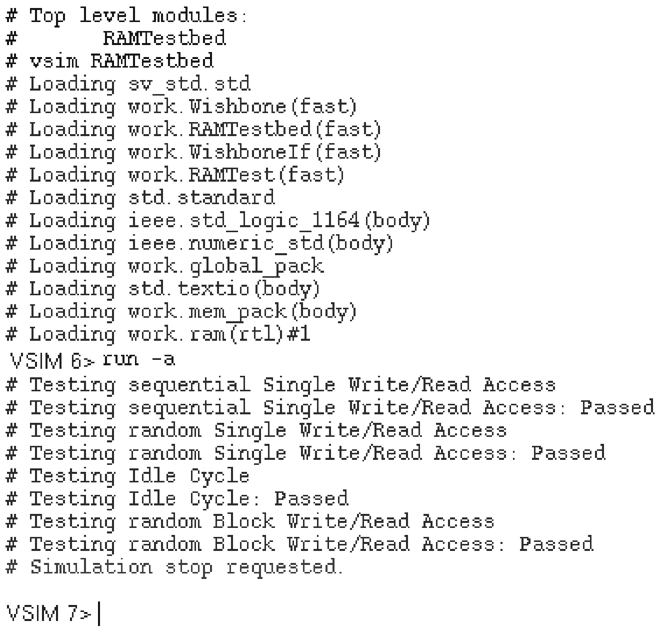
The memory system of the CPU Subsystem includes a ROM and a RAM memory with Wishbone interfaces (see Section 4.2). The ROM is implemented by using a RAM inside and both memories use the same Wishbone interface. Hence, only the verification of the RAM is adequate. However, we separately verified the ROM initialization with an Intel Hex file (see Section 2.4.3). The simulation results of the RAM verification are presented in this section.

5.3.2 RAM Verification Results

We discussed the verification plan and the test bench used for the functional verification of the RAM in Section (4.2). Figure [5.4] presents the successful completion of all tests designed for the verification of the RAM component.

Sequential Single WRITE/READ Access Test Result

This test was designed to sequentially cover the complete address space of the RAM. It first writes a data value to an address, then reads from the same address and finally compares both data values (see Section 4.2.1). Figure [5.5] shows that the test bench asserts a WRITE request (at time 26290 ns) to the RAM with a randomly generated data (0x7B30_911F) and a sequential address (0x0000_020D). As all WRITE accesses get an asynchronous termination, this transfer finishes in the same clock cycle. Then the test bench asserts a READ request (at time 26310 ns) to the RAM from the same address used in the previous WRITE transfer (0x0000_020D). As all READ accesses get a synchronous termination, data is available to the test bench one clock cycle later (at time 26320 ns). Then the test bench compares both data values (written and read). As this test is passed, the next WRITE request is sent to the RAM (at time 26340 ns) with the next sequential



```
# Top level modules:
#       RAMTestbed
# vsim RAMTestbed
# Loading sv_std.std
# Loading work.Wishbone(fast)
# Loading work.RAMTestbed(fast)
# Loading work.WishboneIf(fast)
# Loading work.RAMTest(fast)
# Loading std.standard
# Loading ieee.std_logic_1164(body)
# Loading ieee.numeric_std(body)
# Loading work.global_pack
# Loading std.textio(body)
# Loading work.mem_pack(body)
# Loading work.ram(rtl)#1
VSIM 6> run -a
# Testing sequential Single Write/Read Access
# Testing sequential Single Write/Read Access: Passed
# Testing random Single Write/Read Access
# Testing random Single Write/Read Access: Passed
# Testing Idle Cycle
# Testing Idle Cycle: Passed
# Testing random Block Write/Read Access
# Testing random Block Write/Read Access: Passed
# Simulation stop requested.

VSIM 7>|
```

Figure 5.4 Tests' execution of the functional verification of RAM.

address (0x0000_020E) and a new random data value. The test covers the complete address space of the RAM.

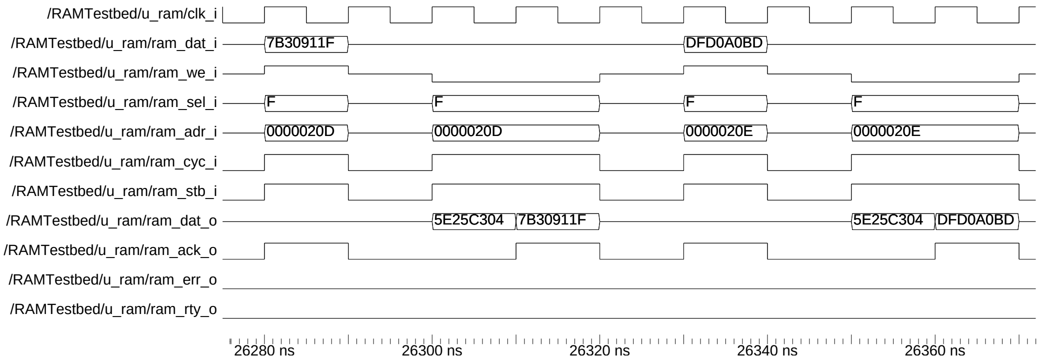


Figure 5.5 Sequential single WRITE/READ access result.

Random Single WRITE/READ Access Test Result

This test was designed to randomly cover the address space of the RAM. It first writes a data value to a random address, then reads from the same address and finally compares both data values (see Section 4.2.1). Figure [5.6] shows that the test bench asserts a WRITE request (at time 877490 ns) to the RAM with a random generated address (0x44DC_BB76) and a random data value (0x081B_E479). As all WRITE accesses get an asynchronous termination, this transfer finishes in the same clock cycle. Then the test bench asserts a READ request (at time 877510 ns) to the RAM from the same address used in the previous WRITE transfer (0x44DC_BB76). As all

READ accesses get a synchronous termination, data is available to the test bench one clock cycle later (at time 877520 ns). The test bench compares both data values (written and read). As this test is passed, the next WRITE request is sent to the RAM (at time 877540 ns) with new random address (0xE8AA_CE1C). This test was repeated 2^{28} times to get an exhaustive verification completeness.

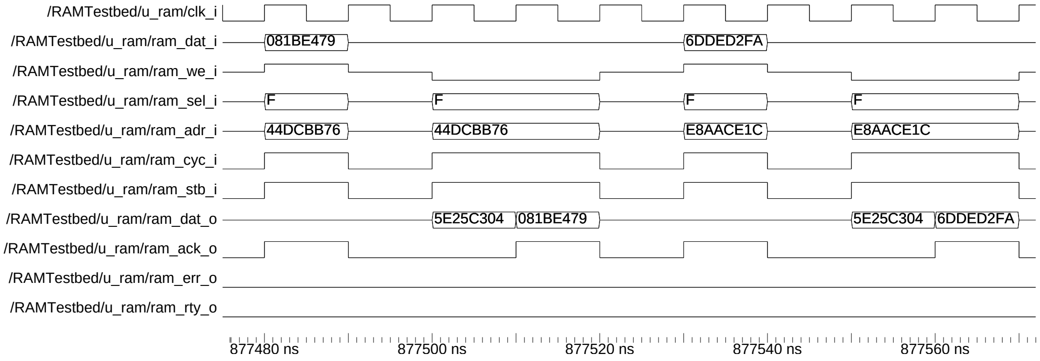

/RAMTestbed/u_ram/clk_i
/RAMTestbed/u_ram/ram_dat_i
081BE479
6DDED2FA
/RAMTestbed/u_ram/ram_we_i
/RAMTestbed/u_ram/ram_sel_i
F
F
F
F
/RAMTestbed/u_ram/ram_adr_i
44DCBB76
44DCBB76
E8AACE1C
E8AACE1C
/RAMTestbed/u_ram/ram_cyc_i
/RAMTestbed/u_ram/ram_stb_i
/RAMTestbed/u_ram/ram_dat_o
5E25C304
081BE479
5E25C304
6DDED2FA
/RAMTestbed/u_ram/ram_ack_o
/RAMTestbed/u_ram/ram_err_o
/RAMTestbed/u_ram/ram_rty_o
877480 ns
877500 ns
877520 ns
877540 ns
877560 ns

Figure 5.6 Random single WRITE/READ access result.

Random Block WRITE/READ Access Test Result

In this test, the test bench sends a block WRITE request to the RAM with randomly generated arrays of addresses and data. Then the test bench sends a block READ request for the same addresses. The received data array is then compared with the written data array (see Section 4.2.1). If the test passes, the next block WRITE request is sent to the RAM with new randomly generated arrays of addresses and data. Figure [5.7] shows a block WRITE request. Figure [5.8] shows a block READ request corresponding to the previous block WRITE transfer. The length of the blocks is randomly generated. It is important to note that by using our approach (all WRITE accesses get an asynchronous termination and all READ accesses get a synchronous termination) a WRITE access of eight blocks length takes exactly eight clock cycles to finish. It was taking nine clock cycles by using the "advanced synchronous cycle termination" approach [4]. However, a READ access of eight blocks length still takes 16 clock cycles because of the synchronous termination limitations. For an exhaustive verification coverage this test was repeated 2^{20} times with random block lengths.

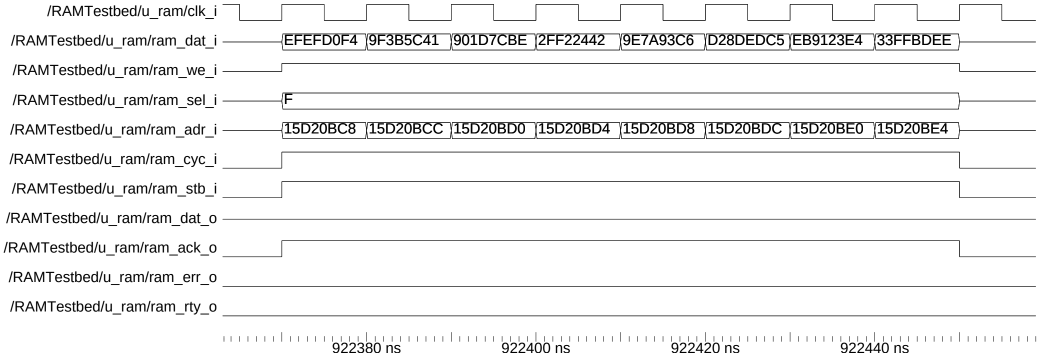

/RAMTestbed/u_ram/clk_i
/RAMTestbed/u_ram/ram_dat_i
EFEFD0F4
9F3B5C41
901D7CBE
2FF22442
9E7A93C6
D28DEDC5
EB9123E4
33FFBDEE
/RAMTestbed/u_ram/ram_we_i
/RAMTestbed/u_ram/ram_sel_i
F
/RAMTestbed/u_ram/ram_adr_i
15D20BC8
15D20BCC
15D20BD0
15D20BD4
15D20BD8
15D20BDC
15D20BE0
15D20BE4
/RAMTestbed/u_ram/ram_cyc_i
/RAMTestbed/u_ram/ram_stb_i
/RAMTestbed/u_ram/ram_dat_o
/RAMTestbed/u_ram/ram_ack_o
/RAMTestbed/u_ram/ram_err_o
/RAMTestbed/u_ram/ram_rty_o
922380 ns
922400 ns
922420 ns
922440 ns

Figure 5.7 Random block WRITE access result.

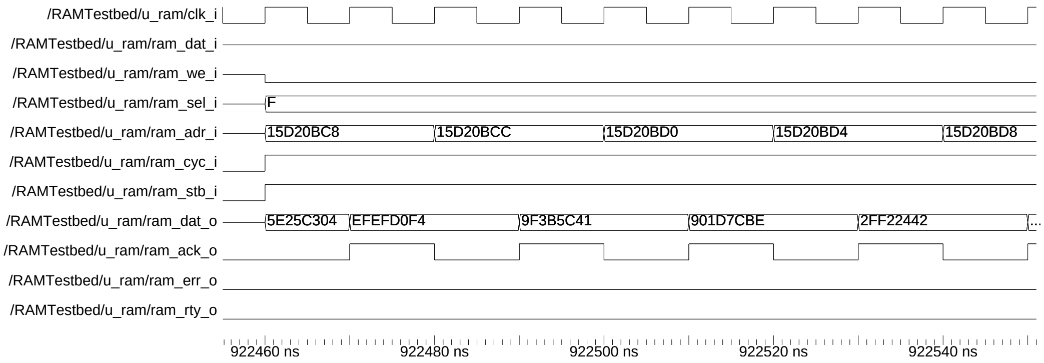


Figure 5.8 Random block READ access result.

5.4 Sub-Bus System Verification Results

5.4.1 Overview

We did a constrained random verification of the Sub-bus system. A detailed verification plan and the test bench used for the functional verification of the triple-layer Sub-bus system is discussed in Section (4.3). In this section we present the results and the achieved verification coverage of the Sub-bus system.

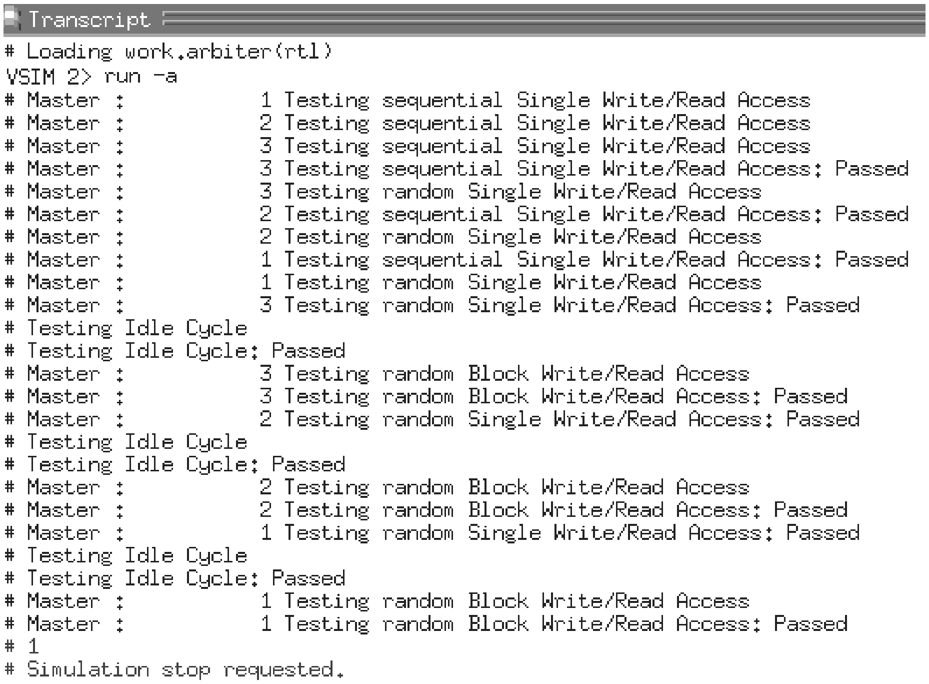
5.4.2 Tests Stimuli Execution

As we have discussed in Section (2.5), the Sub-bus system has three Master interfaces and four Slave interfaces. The Sub-bus system applies a priority based arbitration where each Master interface has a fixed priority. The "Main-bus Master interface (mbus_if)" has the highest priority and the "instruction interface (insn_if)" has the lowest priority. We have connected three BFMs to three Master interfaces for the functional verification of the Sub-bus system. The BFMs emulate the behavior of Master components having Wishbone interface. Figure [4.3] shows the used test bench. Master-1 is the BFM-1 which is connected to the instruction interface (insn_if) of the Sub-bus system. Master-2 is the BFM-2 connected to the data interface (data_if) of the Sub-bus system. Master-3 is the BFM-3 connected to the Main-bus Master interface (mbus_if) of the Sub-bus system. Hence, Master-3 is the highest priority Master component while Master-1 is the lowest priority Master component. Four RAM components are connected to four Slave interfaces of the Sub-bus system.

Figure [5.9] shows that each Master component executes the test stimuli designed for the verification of the RAM component (see Section 4.3). Since it is a priority based Bus system, a higher priority Master finishes its tests before lower priority Masters. Master-3 blocks Master-2 and Master-1. Master-2 blocks Master1. Master-1 finishes last. A higher priority Master blocks the lower priority Master if they access the same Slave component.

5.4.3 Sub-Bus Verification Coverage Results

Figure [5.10] presents the address coverage of all Master components within each Slave component and the address coverage of each Slave component accessed by each Master component. This



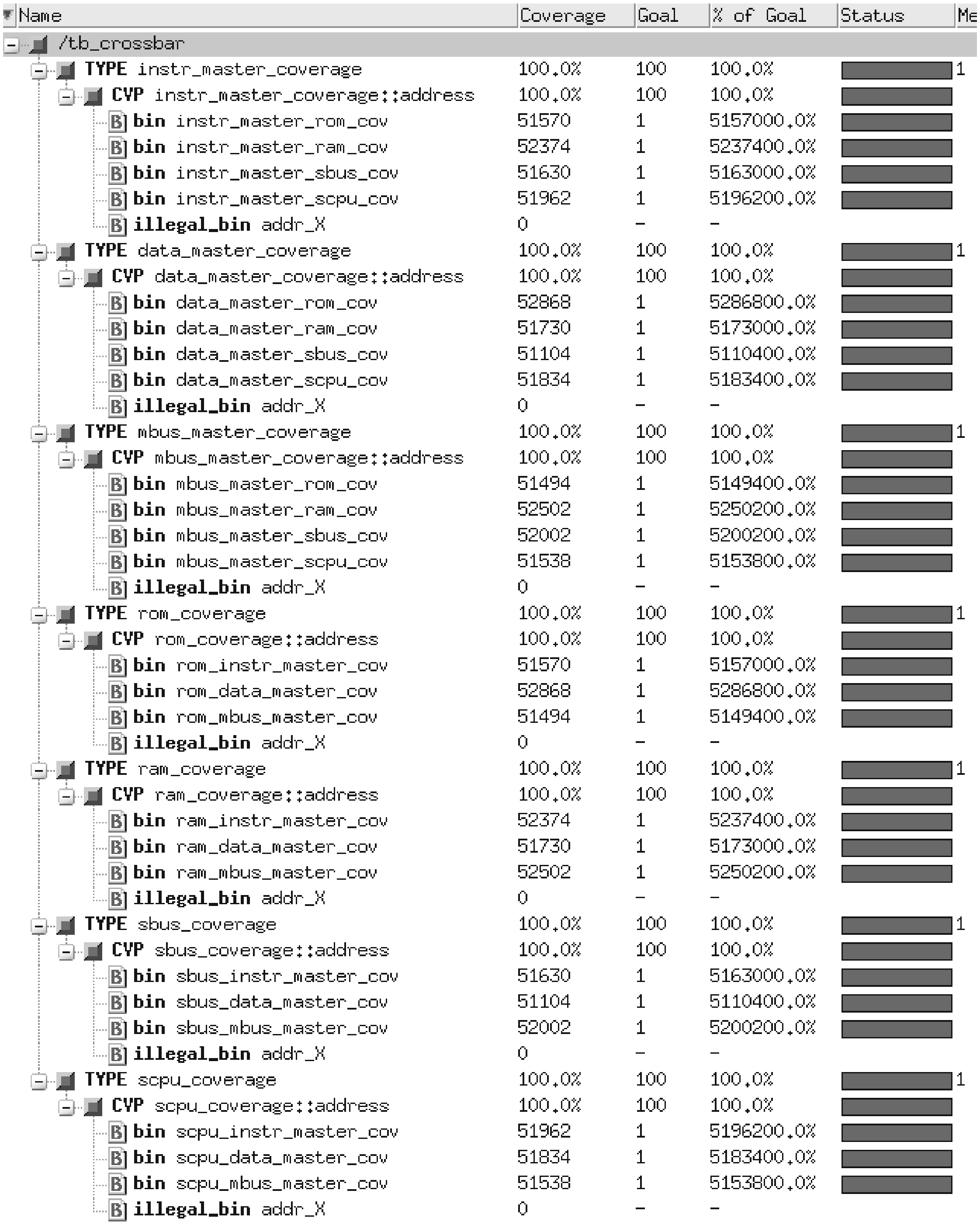
```
# Loading work.arbiter(rtl)
VSIM 2> run -a
# Master :          1 Testing sequential Single Write/Read Access
# Master :          2 Testing sequential Single Write/Read Access
# Master :          3 Testing sequential Single Write/Read Access
# Master :          3 Testing sequential Single Write/Read Access: Passed
# Master :          3 Testing random Single Write/Read Access
# Master :          2 Testing sequential Single Write/Read Access: Passed
# Master :          2 Testing random Single Write/Read Access
# Master :          1 Testing sequential Single Write/Read Access: Passed
# Master :          1 Testing random Single Write/Read Access
# Master :          3 Testing random Single Write/Read Access: Passed
# Testing Idle Cycle
# Testing Idle Cycle: Passed
# Master :          3 Testing random Block Write/Read Access
# Master :          3 Testing random Block Write/Read Access: Passed
# Master :          2 Testing random Single Write/Read Access: Passed
# Testing Idle Cycle
# Testing Idle Cycle: Passed
# Master :          2 Testing random Block Write/Read Access
# Master :          2 Testing random Block Write/Read Access: Passed
# Master :          1 Testing random Single Write/Read Access: Passed
# Testing Idle Cycle
# Testing Idle Cycle: Passed
# Master :          1 Testing random Block Write/Read Access
# Master :          1 Testing random Block Write/Read Access: Passed
# 1
# Simulation stop requested.
```

Figure 5.9 Tests' execution of the functional verification of Sub-bus system.

coverage model was implemented to clearly see the verification completeness of the Sub-bus system by means of addresses accesses (see Section 4.3).

Example The address coverage of the instruction Master depicts that it has accessed the ROM component 51570 times, the RAM component 52374 times, the SBUS component 51630 times and the SCPU component 51962 times. The address coverage of the Slave components verifies the access of the instruction Master.

We have achieved 100% functional verification coverage of the Sub-bus system by means of addresses (each address was accessed).



Name	Coverage	Goal	% of Goal	Status	Me
/tb_crossbar					
TYPE instr_master_coverage	100.0%	100	100.0%	█	1
CVP instr_master_coverage::address	100.0%	100	100.0%	█	
bin instr_master_rom_cov	51570	1	5157000.0%	█	
bin instr_master_ram_cov	52374	1	5237400.0%	█	
bin instr_master_sbus_cov	51630	1	5163000.0%	█	
bin instr_master_scpu_cov	51962	1	5196200.0%	█	
illegal_bin addr_X	0	-	-		
TYPE data_master_coverage	100.0%	100	100.0%	█	1
CVP data_master_coverage::address	100.0%	100	100.0%	█	
bin data_master_rom_cov	52868	1	5286800.0%	█	
bin data_master_ram_cov	51730	1	5173000.0%	█	
bin data_master_sbus_cov	51104	1	5110400.0%	█	
bin data_master_scpu_cov	51834	1	5183400.0%	█	
illegal_bin addr_X	0	-	-		
TYPE mbus_master_coverage	100.0%	100	100.0%	█	1
CVP mbus_master_coverage::address	100.0%	100	100.0%	█	
bin mbus_master_rom_cov	51494	1	5149400.0%	█	
bin mbus_master_ram_cov	52502	1	5250200.0%	█	
bin mbus_master_sbus_cov	52002	1	5200200.0%	█	
bin mbus_master_scpu_cov	51538	1	5153800.0%	█	
illegal_bin addr_X	0	-	-		
TYPE rom_coverage	100.0%	100	100.0%	█	1
CVP rom_coverage::address	100.0%	100	100.0%	█	
bin rom_instr_master_cov	51570	1	5157000.0%	█	
bin rom_data_master_cov	52868	1	5286800.0%	█	
bin rom_mbus_master_cov	51494	1	5149400.0%	█	
illegal_bin addr_X	0	-	-		
TYPE ram_coverage	100.0%	100	100.0%	█	1
CVP ram_coverage::address	100.0%	100	100.0%	█	
bin ram_instr_master_cov	52374	1	5237400.0%	█	
bin ram_data_master_cov	51730	1	5173000.0%	█	
bin ram_mbus_master_cov	52502	1	5250200.0%	█	
illegal_bin addr_X	0	-	-		
TYPE sbus_coverage	100.0%	100	100.0%	█	1
CVP sbus_coverage::address	100.0%	100	100.0%	█	
bin sbus_instr_master_cov	51630	1	5163000.0%	█	
bin sbus_data_master_cov	51104	1	5110400.0%	█	
bin sbus_mbus_master_cov	52002	1	5200200.0%	█	
illegal_bin addr_X	0	-	-		
TYPE scpu_coverage	100.0%	100	100.0%	█	1
CVP scpu_coverage::address	100.0%	100	100.0%	█	
bin scpu_instr_master_cov	51962	1	5196200.0%	█	
bin scpu_data_master_cov	51834	1	5183400.0%	█	
bin scpu_mbus_master_cov	51538	1	5153800.0%	█	
illegal_bin addr_X	0	-	-		

Figure 5.10 Sub-bus verification coverage results.

5.5 OpenRISC1200 Error Reports

5.5.1 Overview

This section presents all errors and faults we find in the OR1200 core. A detailed report is given for every fault.

5.5.2 Extend Half Word with Sign (l.exths) Instruction

The instruction `l.exths` belongs to the ORBIS32-II instruction class of the OpenRISC1000 architecture. The following instructions belong to the family of the `l.exths`.

- Extend Byte with Sign (l.extbs)
- Extend Byte with Zero (l.extbz)
- Extend Half Word with Sign (l.exths)
- Extend Half Word with Zero (l.exthz)

The same inconsistency is found between the OR1200 core and the ISS for all these instructions. These instructions are properly implemented and correctly working in the ISS but the OR1200 core does not contain their implementations. This report is for the instruction `l.exths` only, though it is applicable to other mentioned instructions.

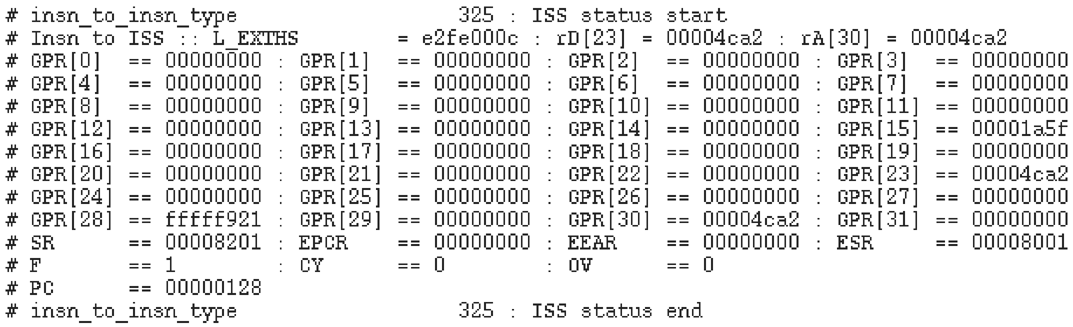
Description

The execution result of the instruction `l.exths` is placed into GPR rD. In execution, bit 15 of GPR rA is placed into the higher-order 16 bits of GPR rD. The low-order 16 bits of rA are copied into low-order 16 bits of rD. The bit encoding of the instruction `l.exths` is given below. More details can be found in the OpenRISC1000 architecture manual [11].

<table>
<tr><th colspan="32">l.exths</th></tr>
<tr><td>31</td><td>.</td><td>.</td><td>.</td><td>.</td><td>26</td><td>25</td><td>.</td><td>.</td><td>.</td><td>21</td><td>20</td><td>.</td><td>.</td><td>.</td><td>16</td><td>15</td><td>.</td><td>.</td><td>.</td><td>.</td><td>10</td><td>9</td><td>.</td><td>.</td><td>6</td><td>5</td><td>4</td><td>3</td><td>.</td><td>.</td><td>0</td></tr>
<tr><td colspan="6">Opcode 0x38</td><td colspan="5">D</td><td colspan="5">A</td><td colspan="6">Reserved</td><td colspan="4">Opcode 0x0</td><td colspan="2">Reserved</td><td colspan="4">Opcode 0xc</td></tr>
<tr><td colspan="6">6 bits</td><td colspan="5">5 bits</td><td colspan="5">5 bits</td><td colspan="6">6 bits</td><td colspan="4">4 bits</td><td colspan="2">2 bits</td><td colspan="4">4 bits</td></tr>
</table>

ISS Implementation of `l.exths`

The ISS implementation of the instruction `l.exths` works correctly. The ISS status and the results after the execution of an instruction `l.exths` (0xe2fe_000c) on the ISS are given in Figure [5.11]. The results show that bit 15 of register rA (GPR[30]) is correctly placed into the higher-order 16 bits of register rD (GPR[23]). The low-order 16 bits of register rA are also correctly copied into the low-order 16 bits of register rD.



```
# insn_to_insn_type                  325 : ISS status start
# Insn to ISS :: L_EXTHS        = e2fe000c : rD[23] = 00004ca2 : rA[30] = 00004ca2
# GPR[0]  == 00000000 : GPR[1]  == 00000000 : GPR[2]  == 00000000 : GPR[3]  == 00000000
# GPR[4]  == 00000000 : GPR[5]  == 00000000 : GPR[6]  == 00000000 : GPR[7]  == 00000000
# GPR[8]  == 00000000 : GPR[9]  == 00000000 : GPR[10] == 00000000 : GPR[11] == 00000000
# GPR[12] == 00000000 : GPR[13] == 00000000 : GPR[14] == 00000000 : GPR[15] == 00001a5f
# GPR[16] == 00000000 : GPR[17] == 00000000 : GPR[18] == 00000000 : GPR[19] == 00000000
# GPR[20] == 00000000 : GPR[21] == 00000000 : GPR[22] == 00000000 : GPR[23] == 00004ca2
# GPR[24] == 00000000 : GPR[25] == 00000000 : GPR[26] == 00000000 : GPR[27] == 00000000
# GPR[28] == fffff921 : GPR[29] == 00000000 : GPR[30] == 00004ca2 : GPR[31] == 00000000
# SR      == 00008201 : EPCR    == 00000000 : EEAR    == 00000000 : ESR     == 00008001
# F       == 1        : CY      == 0        : OV      == 0
# PC      == 00000128
# insn_to_insn_type                  325 : ISS status end
```

Figure 5.11 Execution results of `l.exths` on the ISS.

OR1200 Implementation of `l.exths`

It is mentioned above that the instruction `l.exths` and its other family instructions are not implemented in the OR1200 core. However, it would be interesting to know what happens if this instruction is executed on the OR1200 core. Perhaps this instruction would provide correct results on the ISS. Some pieces of the OR1200 implementation, interesting for the instruction `l.exths`, are given below.

The `l.exths` is an ALU instruction. The signal alu_op (in the code) contains the ALU opcode which is the last four bits of an ALU instruction. Hence, in the instruction decode (id_insn) stage, an instruction `l.exths` sets the alu_op = 0xC (from `l.exths` bit encoding).

```
/*********or1200_ctrl.v*********/
// Decode of alu_op

always @(posedge clk or posedge rst) begin
  if (rst)
    alu_op <= #1 `OR1200_ALUOP_NOP;
  else if (!ex_freeze & id_freeze | flushpipe)
    alu_op <= #1 `OR1200_ALUOP_NOP;
  else if (!ex_freeze) begin
    case (id_insn[31:26])   // synopsys parallel_case
      // ALU instructions except the one with immediate
      `OR1200_OR32_ALU:
        alu_op <= #1 id_insn[3:0];
```

The `l.exths` is an ALU instruction but no ALU opcode is defined for this instruction in the OR1200 implementation, as given below. However, an ALU opcode (OR1200_ALUOP_MOVHI) for another instruction is defined with the same value (4’d12 = 0xC) as for the instruction `l.exths` (from `l.exths` bit encoding).

```
/******or1200_defines.v******/
`define OR1200_ALUOP_MOVHI   4'd12
```

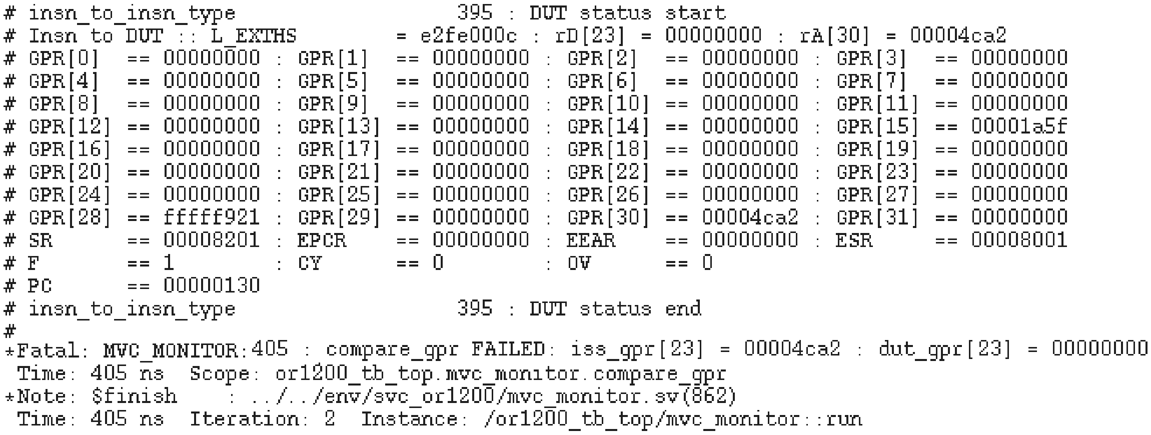
As discussed above, the instruction `l.exths` sets the alu_op to 0xC, which is an opcode of the instruction `l.movhi`. Therefore, the instruction `l.exths` actually results in the execution of another instruction i.e., `l.movhi`. The instruction `l.exths` does not set the macrc_op flag in the instruction decode (ID) stage. Hence, in the execution stage (EX) operand ‘b’ is shifted left by 16 bits in the OR1200_ALUOP_MOVHI implementation (given below). The bit encoding of the instruction `l.movhi` is different. If the instruction is `l.movhi`, the 16-bit immediate value is zero-extended, shifted left by 16 bits and placed into a GPR rD.

```
/******0r1200_alu.v******/
    `OR1200_ALUOP_MOVHI : begin
        if (macrc_op) begin
          result = mult_mac_result;
        end
        else begin
          result = b << 16;
        end
    end
```

Simulation Results of `l.exths`

Figure [5.13] shows the simulation waveform of the OR1200 core. It shows that the instruction `l.exths` (0xe2fe_000c) is in the execution stage (ex_insn) at time 375 ns, where alu_op = 0xC, macrc_op = 0x0, operand a = 0x0000_4ca2 and operand b = 0x0000_0000. The execution result of the instruction `l.movhi` is 0x0000_0000, as macrc_op = 0x0.

After the execution of the instruction `l.exths` on the OR1200 core, the ISS status was compared with the DUV results and a mismatch was found as shown in Figure [5.12].



```
# insn_to_insn_type                  395 : DUT status start
# Insn to DUT :: L_EXTHS         = e2fe000c : rD[23] = 00000000 : rA[30] = 00004ca2
# GPR[0]  == 00000000 : GPR[1]  == 00000000 : GPR[2]  == 00000000 : GPR[3]  == 00000000
# GPR[4]  == 00000000 : GPR[5]  == 00000000 : GPR[6]  == 00000000 : GPR[7]  == 00000000
# GPR[8]  == 00000000 : GPR[9]  == 00000000 : GPR[10] == 00000000 : GPR[11] == 00000000
# GPR[12] == 00000000 : GPR[13] == 00000000 : GPR[14] == 00000000 : GPR[15] == 00001a5f
# GPR[16] == 00000000 : GPR[17] == 00000000 : GPR[18] == 00000000 : GPR[19] == 00000000
# GPR[20] == 00000000 : GPR[21] == 00000000 : GPR[22] == 00000000 : GPR[23] == 00000000
# GPR[24] == 00000000 : GPR[25] == 00000000 : GPR[26] == 00000000 : GPR[27] == 00000000
# GPR[28] == fffff921 : GPR[29] == 00000000 : GPR[30] == 00004ca2 : GPR[31] == 00000000
# SR      == 00008201 : EPCR    == 00000000 : EEAR    == 00000000 : ESR     == 00008001
# F       == 1        : CY      == 0        : OV      == 0
# PC      == 00000130
# insn_to_insn_type                  395 : DUT status end
#
*Fatal: MVC_MONITOR:405 : compare_gpr FAILED: iss_gpr[23] = 00004ca2 : dut_gpr[23] = 00000000
 Time: 405 ns  Scope: or1200_tb_top.mvc_monitor.compare_gpr
*Note: $finish    : ../../env/svc_or1200/mvc_monitor.sv(862)
 Time: 405 ns  Iteration: 2  Instance: /or1200_tb_top/mvc_monitor::run
```

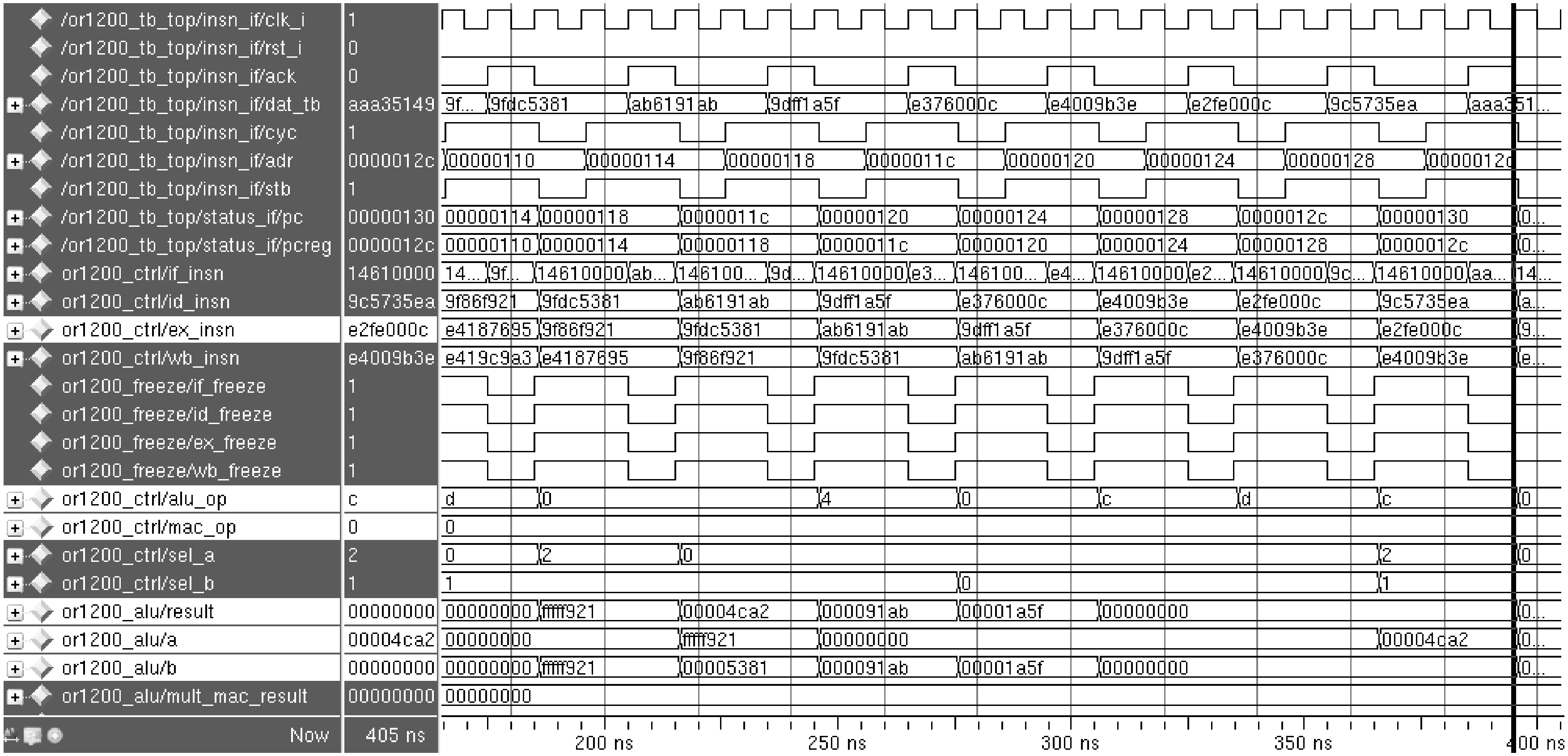
Figure 5.12 Results mismatch of `l.exths` from the OR1200 core and the ISS.

Conclusion

The instruction `l.exths` and its other family instructions are implemented and working correctly in the ISS. Whereas, these instructions are not implemented in the OR1200 core. It is very important to notice that the execution of these instructions in the OR1200 core actually executes the instruction `l.movhi`, as discussed above. This is an implementation fault in the OR1200 core because if an unimplemented instruction executes on the processor, it should generate an illegal exception. However, when `l.exths` is executed the OR1200 executes the instruction `l.movhi` instead of generating an exception. Hence, incorrect execution results are calculated without knowing that an unimplemented instruction has been executed on the processor.

In the presence of this error, we can not include the instruction `l.exths` and its other family instructions in the main verification test of the OR1200 core.

Several benchmark programs were compiled using OR32 C/C++ compiler but it did not generate sign/zero extended instructions. This means that the compiler does not either implement these instructions or often generate them. This is the reason why this error stayed unidentified before. However, the OR32 assembler could assemble code that uses these instructions.



```
/or1200_tb_top/insn_if/clk_i          1
/or1200_tb_top/insn_if/rst_i          0
/or1200_tb_top/insn_if/ack            0
/or1200_tb_top/insn_if/dat_tb         aaa35149
/or1200_tb_top/insn_if/cyc            1
/or1200_tb_top/insn_if/adr            0000012c
/or1200_tb_top/insn_if/stb            1
/or1200_tb_top/status_if/pc           00000130
/or1200_tb_top/status_if/pcreg        0000012c
or1200_ctrl/if_insn                   14610000
or1200_ctrl/id_insn                   9c5735ea
or1200_ctrl/ex_insn                   e2fe000c
or1200_ctrl/wb_insn                   e4009b3e
or1200_freeze/if_freeze               1
or1200_freeze/id_freeze               1
or1200_freeze/ex_freeze               1
or1200_freeze/wb_freeze               1
or1200_ctrl/alu_op                    c
or1200_ctrl/mac_op                    0
or1200_ctrl/sel_a                     2
or1200_ctrl/sel_b                     1
or1200_alu/result                     00000000
or1200_alu/a                          00004ca2
or1200_alu/b                          00000000
or1200_alu/mult_mac_result            00000000
Now                                   405 ns
```

Figure 5.13 Simulation results of `l.exths` on the OR1200 core.

5.5.3 Add Signed and Carry (l.addc) Instruction

The content of GPR rA and GPR rB are added with the carry flag (SR[CY]). The result is then placed into GPR rD.

l.addc								
31 26	25 . . . 21	20 . . . 16	15 . . . 11	10	9	8	7 . . 4	3 . . 0
Opcode 0x38	D	A	B	Reserved	Opcode 0x0		Reserved	Opcode 0x1
6 bits	5 bits	5 bits	5 bits	1 bits	2 bits		4 bits	4 bits

Problem Discussion

Different scenarios need to be discussed to understand the problem with the instruction `l.addc`.

(A) First of all, we need to know how the instruction `l.addc` is implemented in the OR1200 core. Following piece of code shows the implementation of the instructions `l.add` and `l.addc` in the OR1200 ALU.

```
/******or1200_alu.v******/

  assign {cy_sum, result_sum} = a + b;
  `ifdef OR1200_IMPL_ADDC
  assign {cy_csum, result_csum} = a + b + {32'd0, carry};
  `endif
```

(B) We also need to know how the result is generated in the ALU for the write-back stage. The following piece of code generates this result (only the cases corresponding to the instructions `l.add` and `l.addc` are taken). It should be noted that the sensitivity list of the *always* block does not contain result_csum (`l.addc` result from (A)).

```
/******or1200_alu.v******/

  always @(alu_op or a or b or result_sum or result_and or macrc_op or
       shifted_rotated or mult_mac_result) begin
    `ifdef OR1200_CASE_DEFAULT
    casex (alu_op)    // synopsys parallel_case
    `else
    casex (alu_op)    // synopsys full_case parallel_case
    `endif
    `OR1200_ALUOP_ADD : begin
        result = result_sum;
    end
    `ifdef OR1200_IMPL_ADDC
    `OR1200_ALUOP_ADDC : begin
      result = result_csum;
    end
    `endif
  end
```

(C) Next we need to know is how the carry flag is generated (in the ALU) from the cy_sum and the cy_csum signals (from (A)). The piece of code (given below) shows that a combinatorial logic generates the carry flag (cyforw) and a write enable signal (cy_we) for the supervision register (SR).

```
/******or1200_alu.v******/

  always @(alu_op or cy_sum
    `ifdef OR1200_IMPL_ADDC
    or cy_csum
    `endif
        ) begin
        casex (alu_op)     // synopsys parallel_case
         `ifdef OR1200_IMPL_CY
         `OR1200_ALUOP_ADD : begin
             cyforw = cy_sum;
             cy_we = 1'b1;
          end
         `ifdef OR1200_IMPL_ADDC
         `OR1200_ALUOP_ADDC: begin
          cyforw = cy_csum;
          cy_we = 1'b1;
          end
          `endif
          `endif
         default: begin
          cyforw = 1'b0;
          cy_we = 1'b0;
         end
        endcase
  end
```

(D) Further, we need to know how the carry bit is updated in the SR. The following piece of code gives this detail. The carry flag (cyforw) and the write enable signal (cy_we) from (C) first updates the carry bit in to_sr that further updates the SR register. It is very important to note that the freeze logic is not considered when updating the SR and also the carry flag.

```
/******or1200_sprs.v******/

//
// Write enables for SR
//
  assign sr_we = (write_spr && sr_sel) | (branch_op == `OR1200_BRANCHOP_RFE) |
                   flag_we | cy_we;

//
// Supervision register
//
  always @(posedge clk or posedge rst)
    if (rst)
      sr <= #1 {1'b1, `OR1200_SR_EPH_DEF, {`OR1200_SR_WIDTH-3{1'b0}}, 1'b1};
    else if (except_started) begin
      sr[`OR1200_SR_SM]  <= #1 1'b1;
      sr[`OR1200_SR_TEE] <= #1 1'b0;
      sr[`OR1200_SR_IEE] <= #1 1'b0;
      sr[`OR1200_SR_DME] <= #1 1'b0;
      sr[`OR1200_SR_IME] <= #1 1'b0;
    end
    else if (sr_we)
      sr <= #1 to_sr[`OR1200_SR_WIDTH-1:0];

  /*It is clear that SR is updated from TO_SR, and then we need to know how
  TO_SR is being updated.*/

  assign to_sr[`OR1200_SR_CY] =
    (branch_op == ‘OR1200_BRANCHOP_RFE) ? esr[‘OR1200_SR_CY] :
    cy_we ? cyforw : (write_spr && sr_sel) ? spr_dat_o[‘OR1200_SR_CY] :
    sr[‘OR1200_SR_CY];
```

(E) The next thing we need to understand is how the input carry flag is generated for the `l.addc` instruction. The following code shows that the carry flag is combinatorially generated from the SR register's carry bit.

```
/******or1200_sprs.v******/
  //
  // Carry alias
  //
  assign carry = sr[‘OR1200_SR_CY];
```

(F) Further, it is interesting to see how the write enable signal for the register file is generated in the OR1200 core. The code given below shows that the register file write enable signal (rf_we) is controlled by the freeze logic for the write-back pipeline stage (wb_freeze). The register file can be written only when the write-back stage is not frozen i.e., the signal wb_freeze is low. Note that we are not considering the register file write enable from the SPRS but only from the CPU control.

```
/******or1200_rf.v******/
  //
  // RF write enable is either from SPRS or normal from CPU control
  //
  always @(posedge rst or posedge clk)
    if (rst)
      rf_we_allow <= #1 1'b1;
    else if (~wb_freeze)
      rf_we_allow <= #1 ~flushpipe;

  assign rf_we = ((spr_valid & spr_write) | (we & ~wb_freeze)) & rf_we_allow & supv |
       (|rf_addrw));
```

Errors with Instruction `l.addc`

As we have established a good understanding of the `l.addc` implementation in the OR1200, we now discuss the found errors with this instruction.

<u>Error in the OR1200 core</u>

This result indicates an implementation error we found in the OR1200 ALU. It can be seen in the waveform (Figure [5.16]) that an instruction `l.addi` (0x9d72_a73f) is in the execution stage (ex_insn) at time 975075 ns. The destination register for this instruction is rD[11], register operand one is rA[18] (0x0eeb_4c9e) and the immediate value is 0xffff_a73f. The (carry) flag is zero at the time of execution of this instruction (at time 975075 ns). The execution results of the `l.addi` implementation in (A) and (B) are given below:

```
At time = 975075 ns.
    cy_sum      = 1
    result_sum  = 0x0eea_f3dd
    cy_csum     = 1
    result_csum = 0x0eea_f3dd (The ‘carry’ is not updated yet.)
    carry       = 0
    result      = 0x0eea_f3dd

As the carry flag is registered in the SR (in (D)), it is updated one clock
cycle later. Hence, at time = 975085 ns.
    carry       = 1
    cy_csum     = 1
    result_csum = 0x0eea_f3de
```

The carry flag is first updated in to_sr (in (D)) in the same clock cycle (at time 975075 ns) and then in the register SR in the next clock cycle (at time 975085 ns). It means that the carry flag coming from (D) into the ALU (in (A)) is updated in the next clock cycle.

The next instruction executed (ex_insn) on the OR1200 core is `l.addc` (0xe399_7001) at time 975105 ns. The destination register is rD[28], register operand one is rA[25] (0x4d7b_f415) and register operand two is rB[14] (0x0674_0760). The carry flag is set by the time of execution of this instruction (at time 975105 ns). The results after the execution of this instruction are given below.

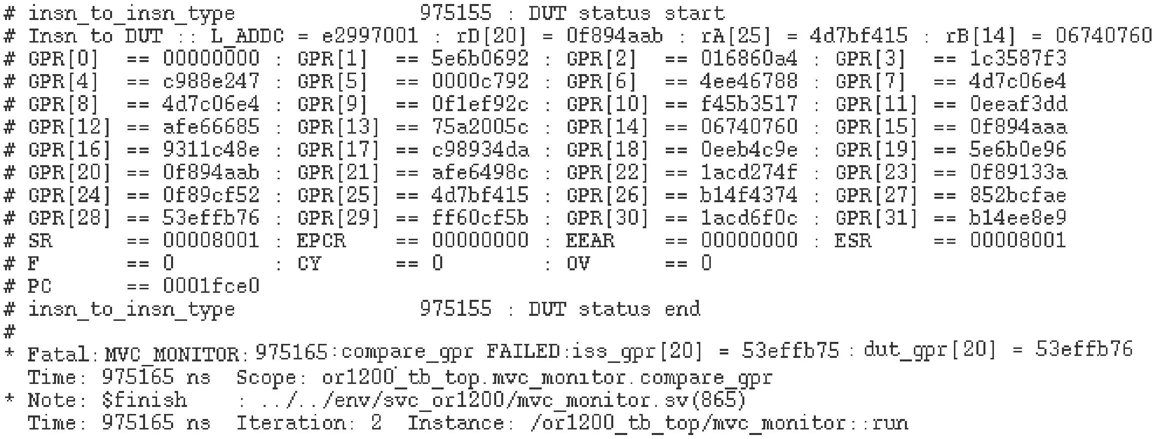
```
At time = 975105 ns.
    cy_sum      =   0
    result_sum  =   0x53effb75
    cy_csum     =   0
    result_csum =   0x53effb76 (The ‘carry’ is not updated yet.)
    result      =   0x53effb76

Since the carry flag is updated one clock cycle later (in (D)).
Hence, at time = 975115 ns.
    cy_sum      =   0
    result_sum  =   0x53effb75
    cy_csum     =   0
    result_csum =   0x53effb75
    result      =   0x53effb76
```

It should be noted that once the carry flag is updated (at time 975115 ns), the result_csum is changed in this execution. As it is not included in the sensitivity list of the *`always`* block in (B), it does not effect the result. Hence, a correct result (0x53ef_fb76) is stored into the destination register rD[28] at time 975135 ns. It takes several cycles to write the register file because the write enable signal (rf_we) is controlled by the write-back freeze logic (wb_freeze) as described in (F).

The next instruction executed (ex_insn) on the DUV is again `l.addc` (0xe299_7001) at time 975135 ns. The destination register is rD[20], register operand one is rA[25] (0x4d7b_f415) and register operand two is rB[14] (0x0674_0760). The carry flag is zero by the time of execution of this instruction (at time 975135 ns). The results after the execution of this instruction are as follow.

```
    cy_sum      =   0
    result_sum  =   0x53effb75
    cy_csum     =   0
    result_csum =   0x53effb75
    result      =   0x53effb76
```

Note that the only difference between the current instruction (0xe299_7001) and the previous instruction (0xe399_7001) is the destination operand (previous = rD[28] and current = rD[20]). The rest of the bits in both instructions are exactly the same. In the ALU implementation (in (B)) the destination operand is not included in the sensitivity list of the *always* block. Therefore, the result can not take the new calculated value of result_csum. Consequently, the destination register gets a wrong value (0x53ef_fb76) at time 975165 ns. Figure [5.14] shows a mismatch between the OR1200 core and its golden model (ISS) because of this wrong value. This means that the temporal result must be included in the sensitivity list of the *always* block (in (B)).



```
# insn_to_insn_type                  975155 : DUT status start
# Insn to DUT :: L_ADDC = e2997001 : rD[20] = 0f894aab : rA[25] = 4d7bf415 : rB[14] = 06740760
# GPR[0]  == 00000000 : GPR[1]  == 5e6b0692 : GPR[2]  == 016860a4 : GPR[3]  == 1c3587f3
# GPR[4]  == c988e247 : GPR[5]  == 0000c792 : GPR[6]  == 4ee46788 : GPR[7]  == 4d7c06e4
# GPR[8]  == 4d7c06e4 : GPR[9]  == 0f1ef92c : GPR[10] == f45b3517 : GPR[11] == 0eeaf3dd
# GPR[12] == afe66685 : GPR[13] == 75a2005c : GPR[14] == 06740760 : GPR[15] == 0f894aaa
# GPR[16] == 9311c48e : GPR[17] == c98934da : GPR[18] == 0eeb4c9e : GPR[19] == 5e6b0e96
# GPR[20] == 0f894aab : GPR[21] == afe6498c : GPR[22] == 1acd274f : GPR[23] == 0f89133a
# GPR[24] == 0f89cf52 : GPR[25] == 4d7bf415 : GPR[26] == b14f4374 : GPR[27] == 852bcfae
# GPR[28] == 53effb76 : GPR[29] == ff60cf5b : GPR[30] == 1acd6f0c : GPR[31] == b14ee8e9
# SR      == 00008001 : EPCR    == 00000000 : EEAR    == 00000000 : ESR     == 00008001
# F       == 0        : CY      == 0        : OV      == 0
# PC      == 0001fce0
# insn_to_insn_type                  975155 : DUT status end
#
* Fatal: MVC_MONITOR: 975165:compare_gpr FAILED:iss_gpr[20] = 53effb75 : dut_gpr[20] = 53effb76
  Time: 975165 ns  Scope: or1200_tb_top.mvc_monitor.compare_gpr
* Note: $finish    : ../../env/svc_or1200/mvc_monitor.sv(865)
  Time: 975165 ns  Iteration: 2  Instance: /or1200_tb_top/mvc_monitor::run
```

Figure 5.14 Results mismatch of `l.addc` from the OR1200 and the ISS.

Error in the ISS

Besides the errors in the OR1200 core we find that the ISS also has a problem with the carry flag implementation. This scenario is discussed here. The ISS implementation of the instruction `l.addc` is given below.

```
/******execgen.c******/

case 0x1:
      /* Not unique: real mask fffffffffc00030f and current mask fc00000f differ - do
          final check */
      if((insn & 0xfc00030f) == 0xe0000001) {
        /* Instruction: l.addc */
        {
          uorreg_t a, b, c;
          /* Number of operands: 3 */
          a = (insn >> 21) & 0x1f;
          #define SET_PARAM0(val) cpu_state.reg[a] = val
          #define PARAM0 cpu_state.reg[a]
          b = (insn >> 16) & 0x1f;
          #define PARAM1 cpu_state.reg[b]
          c = (insn >> 11) & 0x1f;
          #define PARAM2 cpu_state.reg[c]
          {             /* "l_addc" */
            orreg_t temp1, temp2, temp3;
            int8_t temp4;
            temp2 = (orreg_t)PARAM2;
            temp3 = (orreg_t)PARAM1;
            temp1 = temp2 + temp3;
            if(cpu_state.sprs[SPR_SR] & SPR_SR_CY)
              temp1++;
            SET_PARAM0(temp1);
            SET_OV_FLAG_FN (temp1);
            if (ARITH_SET_FLAG) {
              if(!temp1)
                cpu_state.sprs[SPR_SR] |= SPR_SR_F;
              else
                cpu_state.sprs[SPR_SR] &= ~SPR_SR_F;
            }
            if ((uorreg_t) temp1 < (uorreg_t) temp2)
              cpu_state.sprs[SPR_SR] |= SPR_SR_CY;
            else
              cpu_state.sprs[SPR_SR] &= ~SPR_SR_CY;

            temp4 = temp1;
            if (temp4 == temp1)
              or1k_mstats.byteadd++;
          }
          #undef SET_PARAM
          #undef PARAM0
          #undef PARAM1
          #undef PARAM2

          if (do_stats) {
            current->insn_index = 177;   /* "l.addc" */
            analysis(current);
          }
          cpu_state.reg[0] = 0; /* Repair in case we changed it */
        }
      } else { /* Invalid insn */
        {
          l_invalid ();

          if (do_stats) {
            current->insn_index = -1;   /* "???" */
            analysis(current);
          }
          cpu_state.reg[0] = 0; /* Repair in case we changed it */
        }
      }
      break;
```

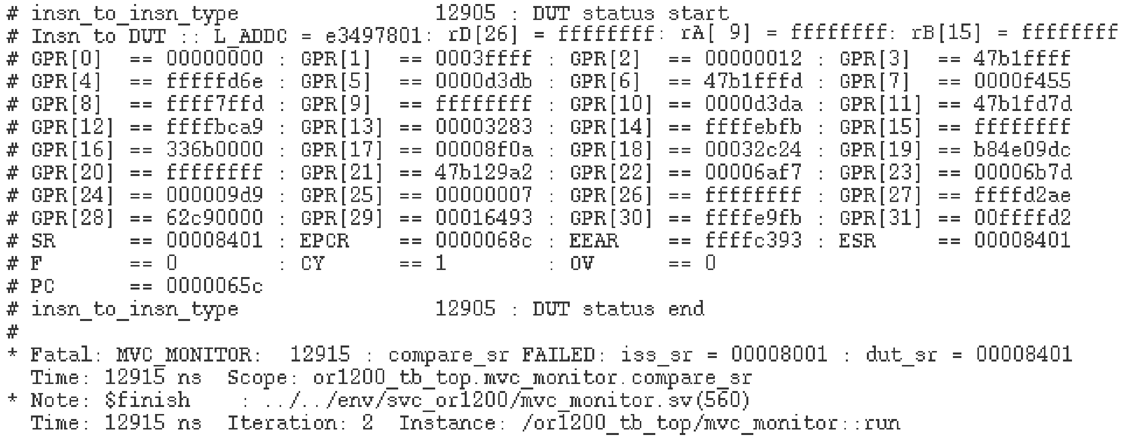
In this scenario, the first instruction sent to the ISS is `l.addc` (0xe149_2801) with the destination register rD[10], the register operand one rA[9] (0xffff_ffff) and the register operand two rB[5] (0x0000_d3db). After the execution of this instruction the destination register gets the result (0x0000_d3da) and the carry flag (CY) is set. The execution results and status registers of the ISS are as given below.

```
# L_ADDC = 0xe1492801 : rD[10] = 0x0000d3da : rA[9] = 0xffffffff : rB[5] = 0x0000d3db
# SR == 0x00008401 : EPCR == 0x0000068c : EEAR == 0xffffc393 : ESR == 0x00008401
# F == 0 : CY == 1 : OV == 0 # PC == 0x00000650
```

The next instruction sent to the ISS is again `l.addc` (0xe349_7801) with the destination register rD[26], the register operand one rA[9] (0xffff_ffff) and the register operand two rB[15] (0xffff_ffff). By considering the ISS implementation of the instruction `l.addc` (given above) there are following results.

```
temp2 = 0xffffffff
temp3 = 0xffffffff
temp1 = 0xfffffffe
```

However, a correct value of `temp1` is 0x1_ffff_fffe but since it is a 32-bit register it only contains 32 bits (0xffff_fffe). Since the carry flag is set (CY = 1), the value of `temp1` is incremented (0xffffffff). Moreover, the values in `temp1` and `temp2` are equal. Hence, the carry bit in the supervision register (SR) is cleared. So the value in the SR is incorrect because a correct result of this execution leads to a set carry bit. The mismatch between the SR of the ISS and the OR1200 is shown in Figure [5.15].



```
# insn_to_insn_type                  12905 : DUT status start
# Insn to DUT :: L_ADDC = e3497801: rD[26] = ffffffff: rA[ 9] = ffffffff: rB[15] = ffffffff
# GPR[0]  == 00000000 : GPR[1]  == 0003ffff : GPR[2]  == 00000012 : GPR[3]  == 47b1ffff
# GPR[4]  == fffffd6e : GPR[5]  == 0000d3db : GPR[6]  == 47b1fffd : GPR[7]  == 0000f455
# GPR[8]  == ffff7ffd : GPR[9]  == ffffffff : GPR[10] == 0000d3da : GPR[11] == 47b1fd7d
# GPR[12] == ffffbca9 : GPR[13] == 00003283 : GPR[14] == ffffebfb : GPR[15] == ffffffff
# GPR[16] == 336b0000 : GPR[17] == 00008f0a : GPR[18] == 00032c24 : GPR[19] == b84e09dc
# GPR[20] == ffffffff : GPR[21] == 47b129a2 : GPR[22] == 00006af7 : GPR[23] == 00006b7d
# GPR[24] == 000009d9 : GPR[25] == 00000007 : GPR[26] == ffffffff : GPR[27] == ffffd2ae
# GPR[28] == 62c90000 : GPR[29] == 00016493 : GPR[30] == ffffe9fb : GPR[31] == 00ffffd2
# SR      == 00008401 : EPCR    == 0000068c : EEAR    == ffffc393 : ESR     == 00008401
# F       == 0        : CY      == 1        : OV      == 0
# PC      == 0000065c
# insn_to_insn_type                  12905 : DUT status end
#
* Fatal: MVC_MONITOR:  12915 : compare_sr FAILED: iss_sr = 00008001 : dut_sr = 00008401
  Time: 12915 ns  Scope: or1200_tb_top.mvc_monitor.compare_sr
* Note: $finish    : ../../env/svc_or1200/mvc_monitor.sv(560)
  Time: 12915 ns  Iteration: 2  Instance: /or1200_tb_top/mvc_monitor::run
```

Figure 5.15 Results mismatch of `l.addc` from the OR1200 core and the ISS.

Conclusion

The instruction `l.addc` implementation in the OR1200 core was found to be erroneous because it does not include the result of the instruction in the sensitivity list of the *always* block (in (B)). Moreover, it is very important to note that the carry flag in the OR1200 core is not controlled by the freeze logic, whereas the update of the destination register is controlled. It leads to an update of `carry` in (A) before the result is stored and a wrong carry could be added. A synthesis tool will automatically add the temporal result to the sensitivity list of the *always* block with a warning. However, the implementation will still not work because the carry flag is not controlled by the freeze logic.

Further, the ISS implementation to update the carry flag is same for all instructions (as for `l.addc`). This implementation has a problem and generates an incorrect result. It means that all instructions (e.g., `l.add`, `l.addi`, `l.addic` etc.) with this implementation to update the carry flag in the ISS may lead to a wrong carry. When this carry is added the calculation result goes wrong.

Several benchmark programs were compiled using OR32 C/C++ compiler but it did not generate the instruction `l.addc`. This means that the compiler does not either implement this instruction or often generate it. This is the reason why this error stayed unidentified before. However, the OR32 assembler could assemble code that uses this instruction.

Consequently, the instruction `l.addc` is excluded from the main verification test of the OR1200 core.

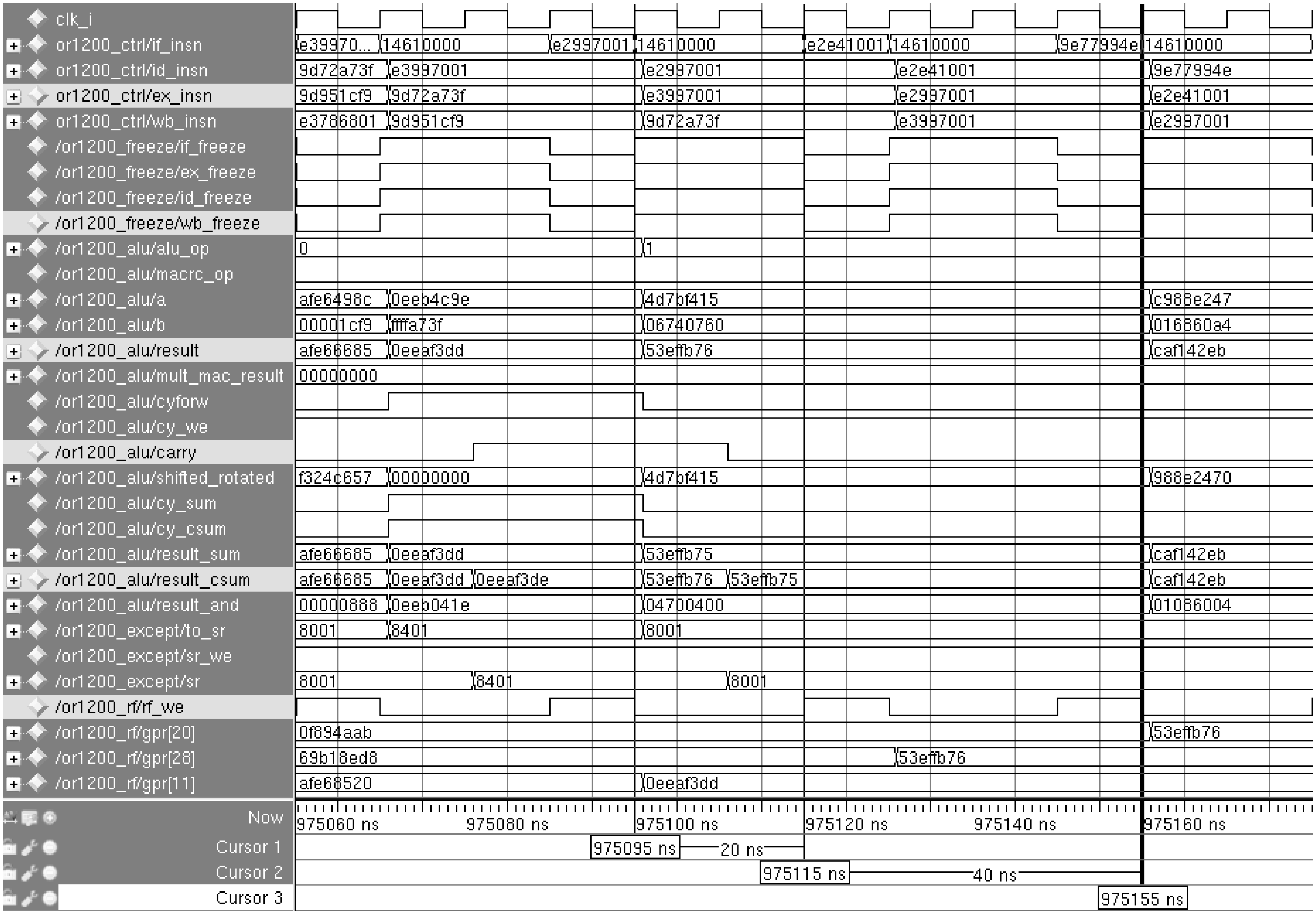


Figure 5.16 Simulation results of `l.addc` on the OR1200 core.

5.5.4 Divide Signed (l.div) Instruction

The content of GPR rA is divided by the content of GPR rB. The result is then placed into GPR rD. Both rA and rB are treated as signed operands. If the divisor is zero the carry flag (SR[CY]) is set.

l.div							
31 26	25 . . . 21	20 . . . 16	15 . . . 11	10	9 8	7 . . 4	3 . . 0
Opcode 0x38	D	A	B	Reserved	Opcode 0x3	Reserved	Opcode 0x9
6 bits	5 bits	5 bits	5 bits	1 bits	2 bits	4 bits	4 bits

ISS implementation of `l.div`

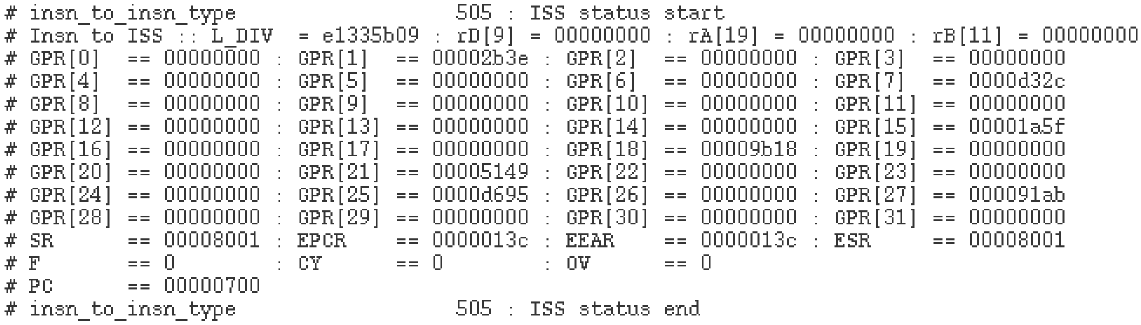
The ISS implementation of the instruction `l.div` is given below. It shows if a divisor (temp3) is zero then the instruction is going to generate an illegal exception. Whereas from the instruction's description (given above), it is required to set the carry flag.

```
/******execgen.c******/

   if((insn & 0xfc00030f) == 0xe0000309) {
     /* Instruction: l.div */
     {
       uorreg_t a, b, c;
       /* Number of operands: 3 */
       a = (insn >> 21) & 0x1f;
       #define SET_PARAM0(val) cpu_state.reg[a] = val
       #define PARAM0 cpu_state.reg[a]
       b = (insn >> 16) & 0x1f;
       #define PARAM1 cpu_state.reg[b]
       c = (insn >> 11) & 0x1f;
       #define PARAM2 cpu_state.reg[c]
       {              /* "l_div" */
         orreg_t temp3, temp2, temp1;

         temp3 = PARAM2;
         temp2 = PARAM1;
         if (temp3)
           temp1 = temp2 / temp3;
         else {
           except_handle(EXCEPT_ILLEGAL, cpu_state.pc);
           return;
         }
```

Figure [5.17] shows the execution results of the instruction `l.div` (0xe133_5b09) executed on the ISS. The divisor (rB[11]) is zero and the instruction generates an illegal exception (PC = 0x0000_0700) instead of setting the carry flag (CY).

OR1200 implementation of `l.div`

The instructions `l.div` and `l.divu` are optional to implement in the OR1200 core and take 32 clock cycles to execute. Both instructions do not drive the carry flag in the OR1200 core. Whereas from their description, both instructions are required to set the carry flag if the divisor is zero.

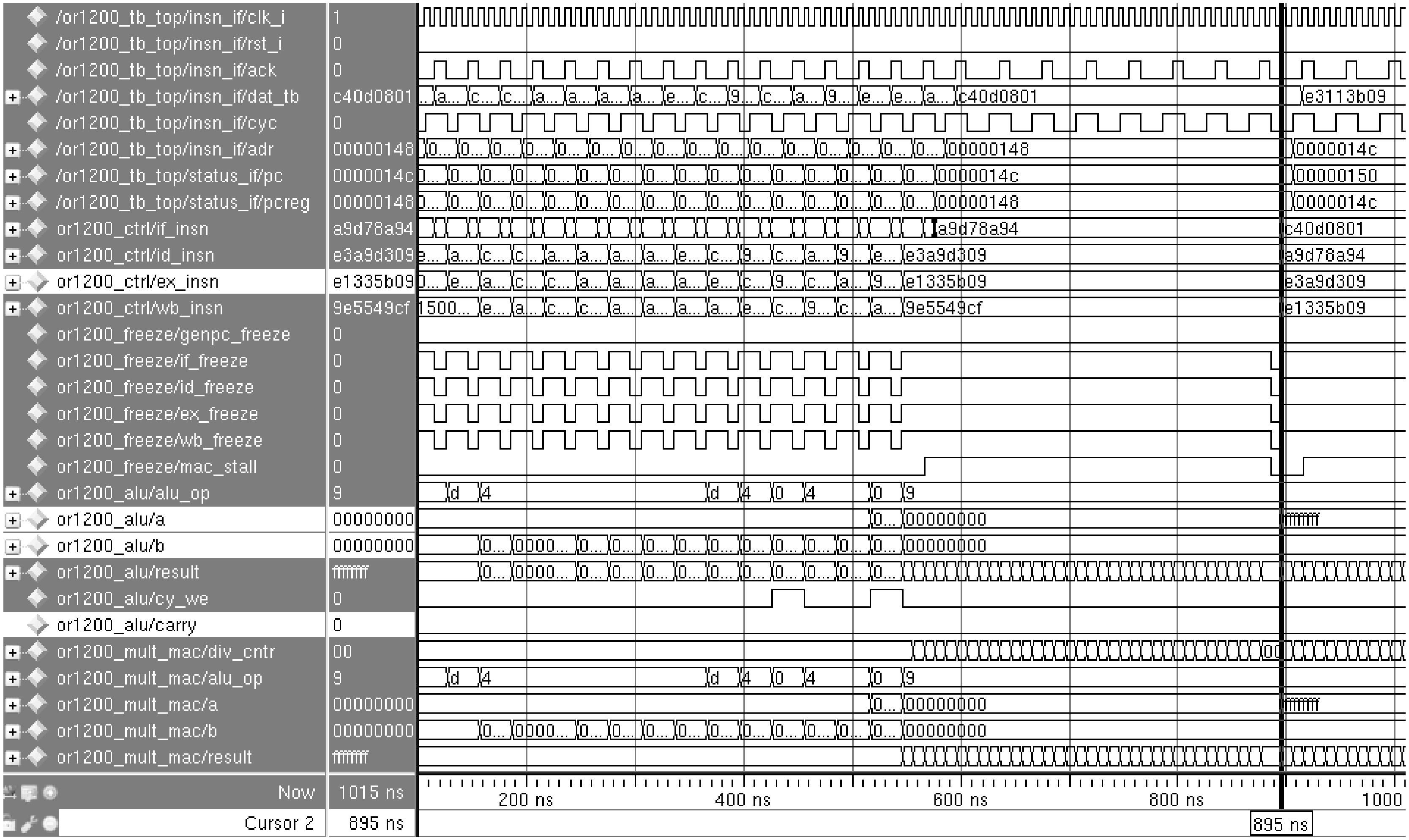


```
# insn_to_insn_type                     505 : ISS status start
# Insn to ISS :: L_DIV  = e1335b09 : rD[9] = 00000000 : rA[19] = 00000000 : rB[11] = 00000000
# GPR[0]  == 00000000 : GPR[1]  == 00002b3e : GPR[2]  == 00000000 : GPR[3]  == 00000000
# GPR[4]  == 00000000 : GPR[5]  == 00000000 : GPR[6]  == 00000000 : GPR[7]  == 0000d32c
# GPR[8]  == 00000000 : GPR[9]  == 00000000 : GPR[10] == 00000000 : GPR[11] == 00000000
# GPR[12] == 00000000 : GPR[13] == 00000000 : GPR[14] == 00000000 : GPR[15] == 00001a5f
# GPR[16] == 00000000 : GPR[17] == 00000000 : GPR[18] == 00009b18 : GPR[19] == 00000000
# GPR[20] == 00000000 : GPR[21] == 00005149 : GPR[22] == 00000000 : GPR[23] == 00000000
# GPR[24] == 00000000 : GPR[25] == 0000d695 : GPR[26] == 00000000 : GPR[27] == 000091ab
# GPR[28] == 00000000 : GPR[29] == 00000000 : GPR[30] == 00000000 : GPR[31] == 00000000
# SR      == 00008001 : EPCR    == 0000013c : EEAR    == 0000013c : ESR     == 00008001
# F       == 0        : CY      == 0        : OV      == 0
# PC      == 00000700
# insn_to_insn_type                     505 : ISS status end
```

Figure 5.17 Instruction `l.div` generates an illegal exception at the ISS.

Conclusion

A division by zero does not generate any compilation error by the OR1200 C/C++ compiler. In the ISS, the instructions `l.div` and `l.divu` generate an illegal exception if a divisor is zero and there is no effect on the carry flag. Whereas, in the OR1200 core, such instruction neither generates an illegal exception nor sets the carry flag. This is a clear mismatch between the ISS implementation and the OR1200 implementation of the instructions `l.div` and `l.divu`. It should be also noted that both instructions do not effect the carry flag when a divisor is zero (neither in the ISS nor in the OR1200 core). This is a mismatch between the implementation of both instructions and their specification provided in the OpenRISC1000 architecture manual [11]. Figure [5.18] shows a waveform that clearly depicts this mismatch. At time 555 ns, the instruction `l.div` (0xe133_5b09) is in the execution stage with operand one (a = 0x0000_0000) and operand two (b = 0x0000_0000). The divisor is zero but the carry flag is not set. Additionally, the waveform confirms that the instruction `l.div` takes 32 clock cycles to execute. Consequently, the instructions `l.div` and `l.divu` are excluded from the main verification test of the OR1200 core.



/or1200_tb_top/insn_if/clk_i	1
/or1200_tb_top/insn_if/rst_i	0
/or1200_tb_top/insn_if/ack	0
/or1200_tb_top/insn_if/dat_tb	c40d0801
/or1200_tb_top/insn_if/cyc	0
/or1200_tb_top/insn_if/adr	00000148
/or1200_tb_top/status_if/pc	0000014c
/or1200_tb_top/status_if/pcreg	00000148
or1200_ctrl/if_insn	a9d78a94
or1200_ctrl/id_insn	e3a9d309
or1200_ctrl/ex_insn	e1335b09
or1200_ctrl/wb_insn	9e5549cf
or1200_freeze/genpc_freeze	0
or1200_freeze/if_freeze	0
or1200_freeze/id_freeze	0
or1200_freeze/ex_freeze	0
or1200_freeze/wb_freeze	0
or1200_freeze/mac_stall	0
or1200_alu/alu_op	9
or1200_alu/a	00000000
or1200_alu/b	00000000
or1200_alu/result	ffffffff
or1200_alu/cy_we	0
or1200_alu/carry	0
or1200_mult_mac/div_cntr	00
or1200_mult_mac/alu_op	9
or1200_mult_mac/a	00000000
or1200_mult_mac/b	00000000
or1200_mult_mac/result	ffffffff
Now	1015 ns
Cursor 2	895 ns

Figure 5.18 Simulation results of `l.div` on the OR1200 core.

5.5.5 Find Last 1 (l.fl1) Instruction

Starting from the MSB, the position of the last ‘1’ bit in GPR rA is placed into a GPR rD. The instruction checks for ‘1’ bit in rA and decrements the count for every zero bit until the last ‘1’ bit is found. If the last ‘1’ bit is found in the MSB, 32 is written into rD. If the last ‘1’ bit is found in LSB, 1 is placed into rD. If no ‘1’ bit is found, zero is placed into rD.

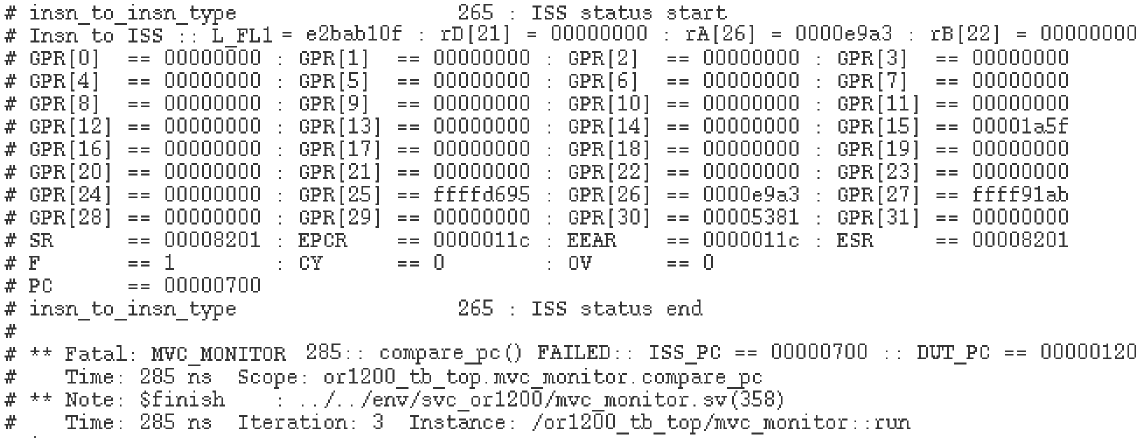
<table>
<tr><th colspan="32">l.fl1</th></tr>
<tr><td>31</td><td>.</td><td>.</td><td>.</td><td>.</td><td>26</td><td>25</td><td>.</td><td>.</td><td>.</td><td>21</td><td>20</td><td>.</td><td>.</td><td>.</td><td>16</td><td>15</td><td>.</td><td>.</td><td>.</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>.</td><td>.</td><td>4</td><td>3</td><td>.</td><td>.</td><td>0</td></tr>
<tr><td colspan="6">Opcode 0x38</td><td colspan="5">D</td><td colspan="5">A</td><td colspan="5">B</td><td>Reserved</td><td colspan="2">Opcode 0x1</td><td colspan="4">Reserved</td><td colspan="4">Opcode 0xf</td></tr>
<tr><td colspan="6">6 bits</td><td colspan="5">5 bits</td><td colspan="5">5 bits</td><td colspan="5">5 bits</td><td>1 bits</td><td colspan="2">2 bits</td><td colspan="4">4 bits</td><td colspan="4">4 bits</td></tr>
</table>

ISS implementation of `l.fl1`

The ISS implementation of the instruction `l.fl1` (given below) shows that it is an invalid instruction which is not implemented in the ISS so far. Whenever the instruction is `l.fl1`, the ISS is going to generate the exception of an illegal instruction.

```
/*********execgen.c*********/

      case 0x1:
        /* Not unique: real mask ffffffffc00030f and current mask fc00030f differ - do
            final check */
        if((insn & 0xfc00030f) == 0xe000010f) {
          /* Instruction: l.fl1 */
          {
            uorreg_t a, b;
            /* Number of operands: 2 */
            a = (insn >> 21) & 0x1f;
            #define SET_PARAM0(val) cpu_state.reg[a] = val
            #define PARAM0 cpu_state.reg[a]
            b = (insn >> 16) & 0x1f;
            #define PARAM1 cpu_state.reg[b]
            l_invalid ();
            #undef SET_PARAM
            #undef PARAM0
            #undef PARAM1

            if (do_stats) {
              current->insn_index = 198;   /* "l.fl1" */
              analysis(current);
            }
            cpu_state.reg[0] = 0; /* Repair in case we changed it */
          }
        } else {
          /* Invalid insn */
          {
            l_invalid ();

            if (do_stats) {
              current->insn_index = -1;   /* "???" */
              analysis(current);
            }
            cpu_state.reg[0] = 0; /* Repair in case we changed it */
          }
        }
        break;
```

Figure [5.19] shows the execution results of the `l.fl1` instruction (0xe2ba_b10f) from the ISS. Since it is an illegal instruction, it generated an illegal exception (PC = 0x0000_0700).



```
# insn_to_insn_type                  265 : ISS status start
# Insn to ISS :: L_FL1 = e2bab10f : rD[21] = 00000000 : rA[26] = 0000e9a3 : rB[22] = 00000000
# GPR[0]  == 00000000 : GPR[1]  == 00000000 : GPR[2]  == 00000000 : GPR[3]  == 00000000
# GPR[4]  == 00000000 : GPR[5]  == 00000000 : GPR[6]  == 00000000 : GPR[7]  == 00000000
# GPR[8]  == 00000000 : GPR[9]  == 00000000 : GPR[10] == 00000000 : GPR[11] == 00000000
# GPR[12] == 00000000 : GPR[13] == 00000000 : GPR[14] == 00000000 : GPR[15] == 00001a5f
# GPR[16] == 00000000 : GPR[17] == 00000000 : GPR[18] == 00000000 : GPR[19] == 00000000
# GPR[20] == 00000000 : GPR[21] == 00000000 : GPR[22] == 00000000 : GPR[23] == 00000000
# GPR[24] == 00000000 : GPR[25] == ffffd695 : GPR[26] == 0000e9a3 : GPR[27] == ffff91ab
# GPR[28] == 00000000 : GPR[29] == 00000000 : GPR[30] == 00005381 : GPR[31] == 00000000
# SR      == 00008201 : EPCR    == 0000011c : EEAR    == 0000011c : ESR     == 00008201
# F       == 1        : CY      == 0        : OV      == 0
# PC      == 00000700
# insn_to_insn_type                  265 : ISS status end
#
# ** Fatal: MVC_MONITOR 285:: compare_pc() FAILED:: ISS_PC == 00000700 :: DUT_PC == 00000120
#    Time: 285 ns  Scope: or1200_tb_top.mvc_monitor.compare_pc
# ** Note: $finish    : ../../env/svc_or1200/mvc_monitor.sv(358)
#    Time: 285 ns  Iteration: 3  Instance: /or1200_tb_top/mvc_monitor::run
```

Figure 5.19 Execution results of `l.fl1` on the ISS.

OR1200 implementation of `l.fl1`

There is no implementation of the instruction `l.fl1` in the OR1200 core. However, when we send an instruction `l.fl1` to the OR1200 core, another instruction is executed instead. This instruction is “Find First 1” (`l.ff1`)[1] and its implementation in the OR1200 core is given below.

```
/*******or1200_defines.v*******/
`define OR1200_ALUOP_FF1   4'd15

/********or1200_alu.v*******/
`ifdef OR1200_CASE_DEFAULT
  casex (alu_op)     // synopsys parallel_case
`else
  casex (alu_op)     // synopsys full_case parallel_case
`endif
    `OR1200_ALUOP_FF1: begin
       result = a[0] ? 1 : a[1] ? 2 : a[2] ? 3 : a[3] ? 4 : a[4] ? 5 : a[5] ?
        6 : a[6] ? 7 : a[7] ? 8 : a[8] ? 9 : a[9] ? 10 : a[10] ? 11 : a[11] ?
        12 : a[12] ? 13 : a[13] ? 14 : a[14] ? 15 : a[15] ? 16 : a[16] ? 17 :
        a[17] ? 18 : a[18] ? 19 : a[19] ? 20 : a[20] ? 21 : a[21] ? 22 : a[22]
        ? 23 : a[23] ? 24 : a[24] ? 25 : a[25] ? 26 : a[26] ? 27 : a[27] ? 28 :
        a[28] ? 29 : a[29] ? 30 : a[30] ? 31 : a[31] ? 32 : 0;
    end
```

Within the `l.ff1` implementation, OR1200_ALUOP_FF1 is defined as 4’d15 (0xF) which is actually an ALU opcode (alu_op). The ALU opcode is described by the last 4 bits of an ALU instruction. The bit encoding of the instruction `l.fl1` shows that the last four bits are same for both

[1] **`l.ff1:`** Starting from the LSB, the position of the first ’1’ bit in GPR rA is placed into GPR rD. This instruction checks for ’1’ bit and increments the count for every zero bit. If the last ’1’ bit is found in the MSB, 32 is written into rD. If the last ’1’ bit is found in the LSB, 1 is placed into rD. If no ’1’ bit is found, zero is placed into rD.

instructions i.e., 0xF. The only difference in the bit encoding of both instructions is in bits [9:8]. Hence, the implementation shown above executes for both instructions. This means that even if the instruction `l.fl1` is not implemented in the OR1200 core, it does not generate an illegal exception. Moreover, it gives a wrong result since it finds the position of the first '1' instead of the last '1'.

It can be seen from the waveform in Figure [5.20] that the instruction `l.fl1` (0xe2ba_b10f) is executed on the OR1200 core at time 315 ns. Register operand one is 0x0000_e9a3 and the ALU opcode is 0xF (alu_op). The calculated result is 0x1 i.e., the first '1' is found in LSB. This is a correct result for the instruction `l.ff1` but an incorrect result for the instruction `l.fl1`. The correct result for the instruction `l.fl1` is 16 i.e., the last '1' bit in register operand one (0x0000_e9a3). It means that even if the instruction `l.fl1` is not implemented in the OR1200 core, the instruction `l.ff1` is executed instead.

Conclusion

The instruction `l.fl1` is neither implemented in the ISS nor in the OR1200 core. However, when this instruction is sent to the OR1200 core, the instruction `l.ff1` is executed instead of generating an illegal instruction exception. As a result, the execution of an unimplemented instruction is never reported. Hence, the instruction `l.fl1` is excluded from the main verification test of the OR1200 core.

Several benchmark programs were compiled using OR32 C/C++ compiler but it did not generate the instruction `l.ff1`. This means that the compiler does not either implement this instruction or often generate it. This is the reason why this error stayed unidentified before. However, the OR32 assembler could assemble code that uses this instruction.

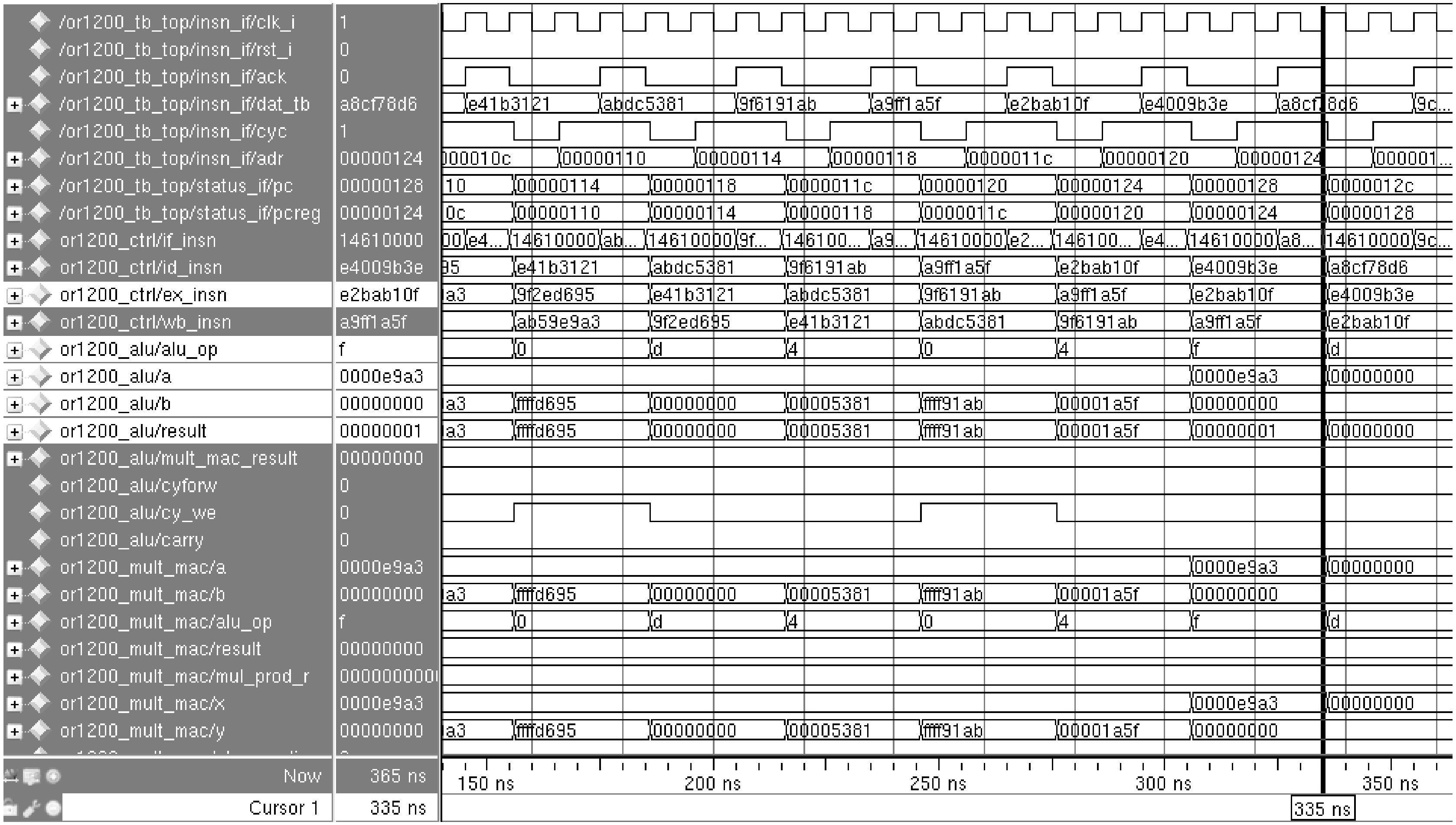

/or1200_tb_top/insn_if/clk_i 1
/or1200_tb_top/insn_if/rst_i 0
/or1200_tb_top/insn_if/ack 0
/or1200_tb_top/insn_if/dat_tb a8cf78d6
/or1200_tb_top/insn_if/cyc 1
/or1200_tb_top/insn_if/adr 00000124
/or1200_tb_top/status_if/pc 00000128
/or1200_tb_top/status_if/pcreg 00000124
or1200_ctrl/if_insn 14610000
or1200_ctrl/id_insn e4009b3e
or1200_ctrl/ex_insn e2bab10f
or1200_ctrl/wb_insn a9ff1a5f
or1200_alu/alu_op f
or1200_alu/a 0000e9a3
or1200_alu/b 00000000
or1200_alu/result 00000001
or1200_alu/mult_mac_result 00000000
or1200_alu/cyforw 0
or1200_alu/cy_we 0
or1200_alu/carry 0
or1200_mult_mac/a 0000e9a3
or1200_mult_mac/b 00000000
or1200_mult_mac/alu_op f
or1200_mult_mac/result 00000000
or1200_mult_mac/mul_prod_r 00000000
or1200_mult_mac/x 0000e9a3
or1200_mult_mac/y 00000000
Now 365 ns
Cursor 1 335 ns
150 ns
200 ns
250 ns
300 ns
350 ns
335 ns

Figure 5.20 Simulation results of `l.fl1` on the OR1200 core.

5.5.6 Multiply Immediate Signed and Accumulate (l.maci) Instruction

The content of GPR rA is multiplied with a sign extended immediate. The result is then truncated to 32 bits and added to the registers MACHI and MACLO (MAC accumulator). All operands are treated as signed integers.

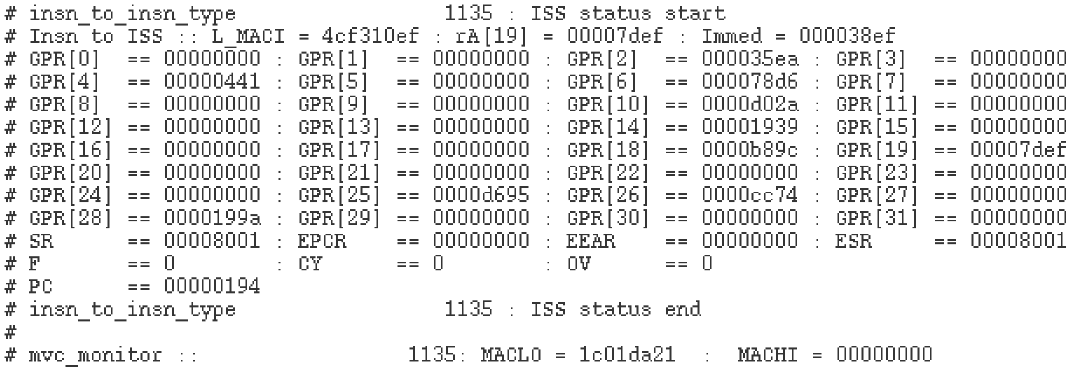
l.maci																															
31	.	.	.	.	26	25	.	.	.	21	20	.	.	.	16	15	.	.	.	11	10	.	.	.	.	.	.	.	.	.	0
Opcode 0x13						Immediate					Reserved					A					Immediate										
6 bits						5 bits					5 bits					5 bits					11 bits										

ISS implementation of `l.maci`

The `l.maci` implementation in the ISS (given below) shows that the operand 'a' (register rA) is taken from bits [20:16] of an instruction (insn[20:16]). The bit encoding of `l.maci` tells that bits [20:16] are not used (reserved). This is a considerable discrepancy between the specifications of `l.maci` and its implementation in the ISS.

```
/**********execgen.c**********/
  case 0x13:
    /* Not unique: real mask fffffffffc000000 and current mask fc000000 differ – do final
        check */
    if((insn & 0xfc000000) == 0x4c000000) {
      /* Instruction: l.maci */
      {
        uorreg_t a, b;
        /* Number of operands: 2 */
        a = (insn >> 16) & 0x1f;
        #define SET_PARAM0(val) cpu_state.reg[a] = val
        #define PARAM0 cpu_state.reg[a]
        b = (insn >> 0) & 0x7ff;
        b |= ((insn >> 21) & 0x1f) << 11;
        if(b & 0x00008000) b |= 0xffff8000;
        #define PARAM1 b
        {           /* "l_mac" */
          uorreg_t lo, hi;
          LONGEST l;
          orreg_t x, y;

          lo = cpu_state.sprs[SPR_MACLO];
          hi = cpu_state.sprs[SPR_MACHI];
          x = PARAM0;
          y = PARAM1;
        /*   PRINTF ("[%"PRIxREG",%"PRIxREG"]\t", x, y); */
          l = (ULONGEST)lo | ((LONGEST)hi << 32);
          l += (LONGEST) x * (LONGEST) y;

          /* This implementation is very fast – it needs only one cycle for mac.  */
          lo = ((ULONGEST)l) & 0xFFFFFFFF;
          hi = ((LONGEST)l) >> 32;
          cpu_state.sprs[SPR_MACLO] = lo;
          cpu_state.sprs[SPR_MACHI] = hi;
        /*   PRINTF ("(%"PRIxREG",%"PRIxREG"\n", hi, lo); */
        }
        #undef SET_PARAM
        #undef PARAM0
        #undef PARAM1
        if (do_stats) {
          current->insn_index = 106;    /* "l.maci" */
          analysis(current);
        }
      }
    } else {
      /* Invalid insn */
      {
        l_invalid ();

        if (do_stats) {
          current->insn_index = -1;    /* "???" */
          analysis(current);
        }
      }
    }
    break;
```

Figure [5.21] shows that the instruction `l.maci` (0x4cf3_10ef) is sent to the ISS. According to the implementation of the instruction `l.maci`, operand one is rA[19] and the immediate value is 0x0000_38ef. The register rA is taken from bits [20:16] of the instruction and the 16-bit immediate is taken from bits [25:21] and bits [10:0] (0x0000_38ef). However, according to the bit encoding of the instruction `l.maci`, bits [20:16] are not used (reserved) and the register rA should have been taken from bits [15:11] i.e., rA[2].



```
# insn_to_insn_type                 1135 : ISS status start
# Insn to ISS :: L_MACI = 4cf310ef : rA[19] = 00007def : Immed = 000038ef
# GPR[0]  == 00000000 : GPR[1]  == 00000000 : GPR[2]  == 000035ea : GPR[3]  == 00000000
# GPR[4]  == 00000441 : GPR[5]  == 00000000 : GPR[6]  == 000078d6 : GPR[7]  == 00000000
# GPR[8]  == 00000000 : GPR[9]  == 00000000 : GPR[10] == 0000d02a : GPR[11] == 00000000
# GPR[12] == 00000000 : GPR[13] == 00000000 : GPR[14] == 00001939 : GPR[15] == 00000000
# GPR[16] == 00000000 : GPR[17] == 00000000 : GPR[18] == 0000b89c : GPR[19] == 00007def
# GPR[20] == 00000000 : GPR[21] == 00000000 : GPR[22] == 00000000 : GPR[23] == 00000000
# GPR[24] == 00000000 : GPR[25] == 0000d695 : GPR[26] == 0000cc74 : GPR[27] == 00000000
# GPR[28] == 0000199a : GPR[29] == 00000000 : GPR[30] == 00000000 : GPR[31] == 00000000
# SR      == 00008001 : EPCR    == 00000000 : EEAR    == 00000000 : ESR     == 00008001
# F       == 0        : CY      == 0        : OV      == 0
# PC      == 00000194
# insn_to_insn_type                 1135 : ISS status end
#
# mvc_monitor ::                 1135: MACLO = 1c01da21  :  MACHI = 00000000
```

Figure 5.21 Execution results of `l.maci` on the ISS.

OR1200 implementation of `l.maci`

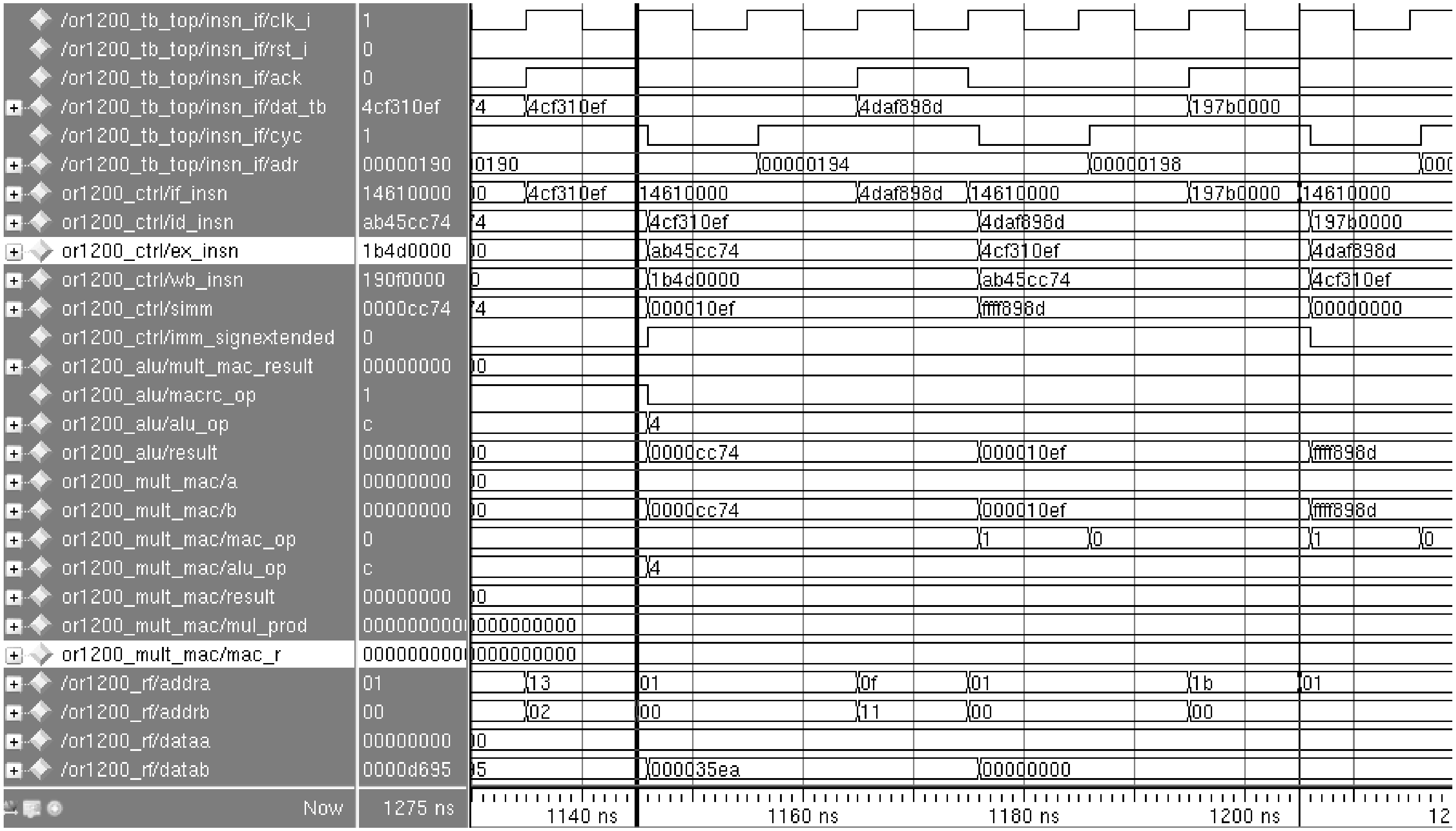
The OR1200 implementation for the instruction `l.maci` is given below. It shows that the GPR used in the execution of this instruction is taken from bits [20:16]. The immediate value is taken from bits [15:0] of the instruction. The waveform in Figure [5.22] shows the execution of the instruction `l.maci` (0x4cf3_10ef). At time 1155 ns, this instruction is in the instruction decode stage (id_insn). The immediate value (simm), later used in the execution stage (at time 1185 ns), is taken from the instruction's bits [15:0] (0x0000_10ef). Note that the immediate value is not taken according to the bit encoding of the instruction `l.maci` which specifies bits [25:21] and bits [10:0] for the immediate value.

```
/******or1200_ctrl.v******/
//
// Register file read addresses
//
assign rf_addra = if_insn[20:16];
assign rf_addrb = if_insn[15:11];
assign rf_rda   = if_insn[31];
assign rf_rdb   = if_insn[30];

//
// Decode of imm_signextend
//
always @(id_insn) begin
  case (id_insn[31:26])   // synopsys parallel_case

  // l.maci
`ifdef OR1200_MAC_IMPLEMENTED
  `OR1200_OR32_MACI:
    imm_signextend = 1'b1;
`endif

//
// Sign/Zero extension of immediates
//
assign simm = (imm_signextend == 1'b1) ? {{16{id_insn[15]}}, id_insn[15:0]} : {{16'b0},
    id_insn[15:0]};
```

Conclusion

A very clear difference between the `l.maci` implementation in the ISS and the OR1200 core is on the bit encoding of the immediate value. The ISS takes the immediate value from bits [25:21] and bits [10:0] while the OR1200 core takes the immediate value from bits [15:0] of an instruction. Despite that, GPR rA is taken from the instruction bits [20:16] in the ISS and the OR1200 core. This is not according to the instruction's bit encoding given in the OpenRISC1000 architecture manual where bits [15:11] are specified for GPR rA. In the presence of these differences, it is not possible to include this instruction in the main verification test of the OR1200 core.

Several benchmark programs were compiled using OR32 C/C++ compiler but it did not generate the instruction `l.maci`. This means that the compiler does not either implement this instruction or often generate it. This is the reason why this error stayed unidentified before. However, the OR32 assembler could assemble code that uses this instruction.



5.5.7 Multiply Immediate Signed (l.muli) Instruction

The content of GPR rA is multiplied with an immediate value. The result is then truncated to a 32-bit value and placed into GPR rD.

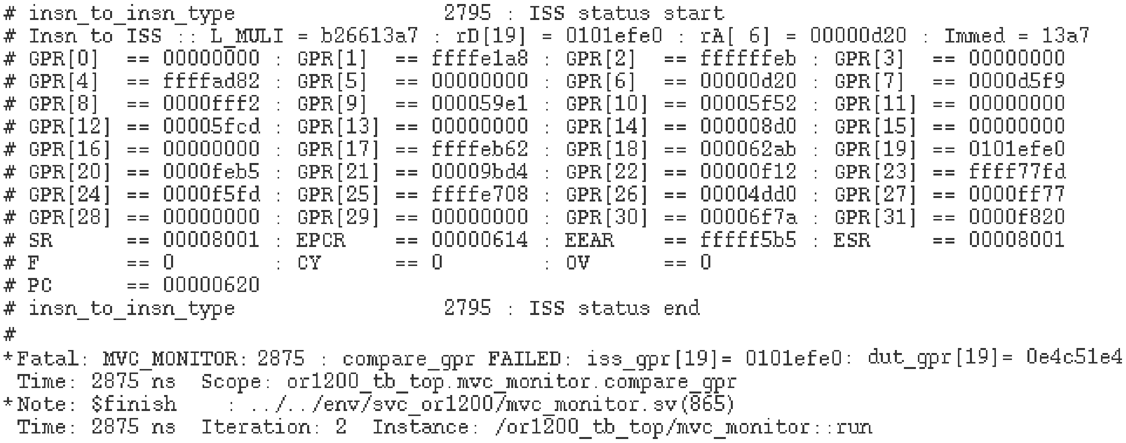
<table>
<tr><th colspan="32">l.muli</th></tr>
<tr><td>31</td><td>.</td><td>.</td><td>.</td><td>.</td><td>26</td><td>25</td><td>.</td><td>.</td><td>.</td><td>21</td><td>20</td><td>.</td><td>.</td><td>.</td><td>16</td><td>15</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>0</td></tr>
<tr><td colspan="6">Opcode 0x2c</td><td colspan="5">D</td><td colspan="5">A</td><td colspan="16">Immediate</td></tr>
<tr><td colspan="6">6 bits</td><td colspan="5">5 bits</td><td colspan="5">5 bits</td><td colspan="16">16 bits</td></tr>
</table>

Problem Discussion

The `l.muli` implementation in the ISS is working correctly. However, its implementation in the OR1200 core has an error. The problem belongs to the freeze logic implementation for the instruction. The instruction `l.muli` is a multicycle instruction but it is not controlled by the freeze logic (for the number of execution cycles), as the OR1200 implementation shows below.

```
/******or1200_defines.v******/
// ALU instructions multicycle field in machine word
`define OR1200_ALUMCYC_POS     9:8

/******or1200_ctrl.v******/
//
// Decode of multicycle
//
always @(id_insn) begin
  case (id_insn[31:26])   // synopsys parallel_case

    // l.sb
    `OR1200_OR32_SB:
      multicycle = `OR1200_TWO_CYCLES;

    // ALU instructions except the one with immediate
    `OR1200_OR32_ALU:
      multicycle = id_insn[`OR1200_ALUMCYC_POS];

/*
ALU instructions (l.add/l.mul etc.) have op-code in bits {[31:26],[9:8],[3:0]}.
OR1200_ALUMCYC_POS is bits [9:8] of an instruction that is commonly 2'b11 for
multicycle instructions (e.g., l.mul, l.div etc.) and 2'b00 for single cycle
instructions (e.g., l.add, l.sub, l.or etc.).
*/

    // Single cycle instructions
    default: begin
      multicycle = `OR1200_ONE_CYCLE;
    end
  endcase
end
```

The multicycle signal shows the number of clock cycles that an instruction (in the ID-stage (id_insn)) should take to complete its execution (in the EX-stage). The instruction `l.muli` basically executes the implementation of the instruction `l.mul` in the OR1200 core. For multicycle instructions bits [9:8] (OR1200_ALUMCYC_POS) are explicitly specified as an opcode. This opcode is used to specify the number of cycles (multicycle signal) that an instruction takes in its execution. But the `l.muli` instruction's bits [9:8] are not explicitly specified as an opcode. The waveform in Figure [5.24] shows an instruction `l.muli` (0xb266_13a7) in the instruction decode stage (id_insn) at time 2815 ns. Even if it is a multicycle instruction, the multicycle signal is low. The instruction `l.muli` takes three clock cycles to execute (executes as `l.mul`). Hence, its execution result is available after three clock cycles. Since the execution stage of the instruction `l.muli` is not controlled by the freeze logic for multicycles and it is treated as a single cycle instruction, an incorrect result (0x0e4c_51e4) is taken at time 2865 ns. For a multicycle instruction of three clock cycles a correct result is calculated and available at time 2875 ns (mac_prod_r = 0x0101_efe0). Figure [5.23] shows a mismatch between the ISS and the OR1200 core results after the execution of the same `l.muli` instruction.



```
# insn_to_insn_type                  2795 : ISS status start
# Insn to ISS :: L_MULI = b26613a7 : rD[19] = 0101efe0 : rA[ 6] = 00000d20 : Immed = 13a7
# GPR[0]  == 00000000 : GPR[1]  == ffffe1a8 : GPR[2]  == ffffffeb : GPR[3]  == 00000000
# GPR[4]  == ffffad82 : GPR[5]  == 00000000 : GPR[6]  == 00000d20 : GPR[7]  == 0000d5f9
# GPR[8]  == 0000fff2 : GPR[9]  == 000059e1 : GPR[10] == 00005f52 : GPR[11] == 00000000
# GPR[12] == 00005fcd : GPR[13] == 00000000 : GPR[14] == 000008d0 : GPR[15] == 00000000
# GPR[16] == 00000000 : GPR[17] == ffffeb62 : GPR[18] == 000062ab : GPR[19] == 0101efe0
# GPR[20] == 0000feb5 : GPR[21] == 00009bd4 : GPR[22] == 00000f12 : GPR[23] == ffff77fd
# GPR[24] == 0000f5fd : GPR[25] == ffffe708 : GPR[26] == 00004dd0 : GPR[27] == 0000ff77
# GPR[28] == 00000000 : GPR[29] == 00000000 : GPR[30] == 00006f7a : GPR[31] == 0000f820
# SR      == 00008001 : EPCR    == 00000614 : EEAR    == fffff5b5 : ESR     == 00008001
# F       == 0        : CY      == 0        : OV      == 0
# PC      == 00000620
# insn_to_insn_type                  2795 : ISS status end
#
*Fatal: MVC_MONITOR: 2875 : compare_gpr FAILED: iss_gpr[19]= 0101efe0: dut_gpr[19]= 0e4c51e4
 Time: 2875 ns  Scope: or1200_tb_top.mvc_monitor.compare_gpr
*Note: $finish    : ../../env/svc_or1200/mvc_monitor.sv(865)
 Time: 2875 ns  Iteration: 2  Instance: /or1200_tb_top/mvc_monitor::run
```

Figure 5.23 Results mismatch of `l.muli` from the OR1200 core and the ISS.

Conclusion

The instruction `l.muli` is not working correctly in the OR1200 core. It is a multicycle instruction but not controlled by the freeze logic. Therefore, an incorrect result is selected because of an incorrect selection time. Hence, it is not possible to include this instruction in the main verification test of the OR1200 core.

Several benchmark programs were compiled using OR32 C/C++ compiler but it did not generate the instruction `l.muli`. This means that the compiler does not either implement this instruction or often generate it. This is the reason why this error stayed unidentified before. However, the OR32 assembler could assemble code that uses this instruction.

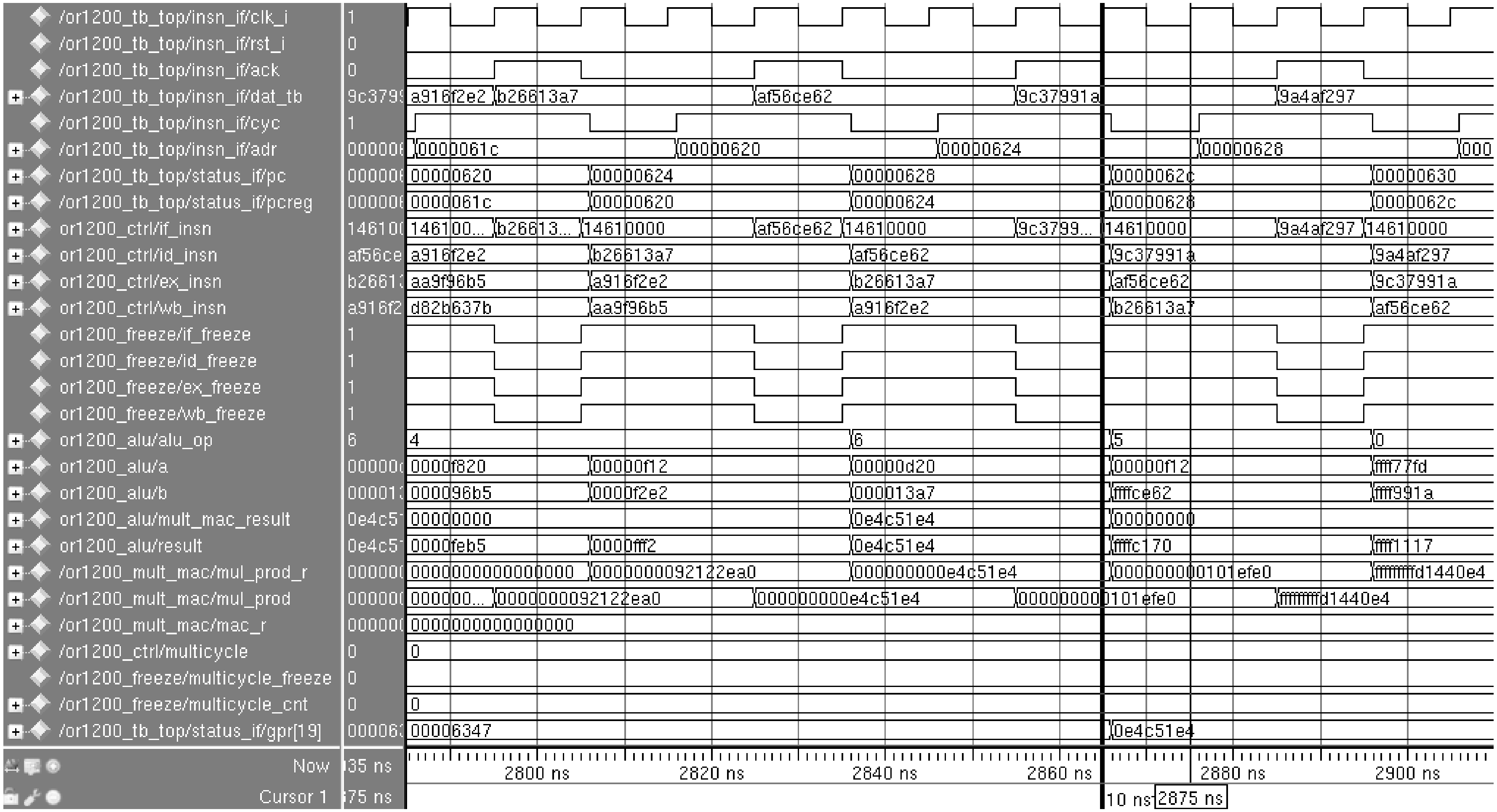

/or1200_tb_top/insn_if/clk_i
/or1200_tb_top/insn_if/rst_i
/or1200_tb_top/insn_if/ack
/or1200_tb_top/insn_if/dat_tb
/or1200_tb_top/insn_if/cyc
/or1200_tb_top/insn_if/adr
/or1200_tb_top/status_if/pc
/or1200_tb_top/status_if/pcreg
or1200_ctrl/if_insn
or1200_ctrl/id_insn
or1200_ctrl/ex_insn
or1200_ctrl/wb_insn
or1200_freeze/if_freeze
or1200_freeze/id_freeze
or1200_freeze/ex_freeze
or1200_freeze/wb_freeze
or1200_alu/alu_op
or1200_alu/a
or1200_alu/b
or1200_alu/mult_mac_result
or1200_alu/result
/or1200_mult_mac/mul_prod_r
/or1200_mult_mac/mul_prod
/or1200_mult_mac/mac_r
/or1200_ctrl/multicycle
/or1200_freeze/multicycle_freeze
/or1200_freeze/multicycle_cnt
/or1200_tb_top/status_if/gpr[19]
Now
Cursor 1
2800 ns
2820 ns
2840 ns
2860 ns
2880 ns
2900 ns
10 ns
2875 ns

Figure 5.24 Simulation results of `l.muli` on the OR1200 core.

5.5.8 Multiply Unsigned (l.mulu) Instruction

The content of GPR rA is multiplied by the content of GPR rB. The result is then truncated to 32 bits and placed into GPR rD. All operands are treated as signed integers.

l.mulu							
31 26	25 . . . 21	20 . . . 16	15 . . . 11	10	9 8	7 . . 4	3 . . 0
Opcode 0x38	D	A	B	Reserved	Opcode 0x3	Reserved	Opcode 0xb
6 bits	5 bits	5 bits	5 bits	1 bits	2 bits	4 bits	4 bits

Problem Discussion

The instruction `l.mulu` is neither implemented in the ISS nor in the OR1200 core. It belongs to the ORBIS32-I instruction class of the OpenRISC1000 architecture and all instructions belonging to this class are compulsory to implement [11]. The OR1200 core and the ISS both are supposed to generate an illegal exception for any illegal or unimplemented instruction but the OR1200 core does not generate an illegal exception in case of the instruction `l.mulu`. This means that the execution of an illegal or unimplemented instruction is not reported. It is an implementation fault in the OR1200 core where the instruction `l.mulu` is executed as an instruction that moves the content of GPR rB to a destination register (rD). The fault can be seen within the OR1200 implementation given below, which shows that 0xb (OR1200_ALUOP_IMM = 4'd11) is an ALU opcode to move an immediate value to a destination register. As the instruction `l.mulu` also contains the opcode 0xb in its last four bits, the implementation to move an immediate value to a GPR is executed instead and generates a wrong result. Hence, the instruction `l.mulu` is not taken as an illegal or an unimplemented instruction even though it is not implemented in the OR1200 core. The waveform in Figure [5.25] shows that the instruction `l.mulu` (0xe147_930b) is in the execution stage at time 2845 ns. The destination register is rD[10], the register operand one is rA[7] (0x0000_d5f9) and the register operand two is rB[18] (0x0000_62ab). When the instruction `l.mulu` is in the execution stage, the ALU opcode (alu_op) is 0xb which is actually the last four bits of an ALU instruction (insn[3:0]). Further, operand "/or1200_alu/b" (rB[18]) is first placed into the result at time 2845 ns and then stored into rD[10] (0x0000_62ab) at time 2885 ns. Despite that, neither an except_illegal signal is reported nor the next program counter (PC) is an illegal exception vector (0x0000_0700).

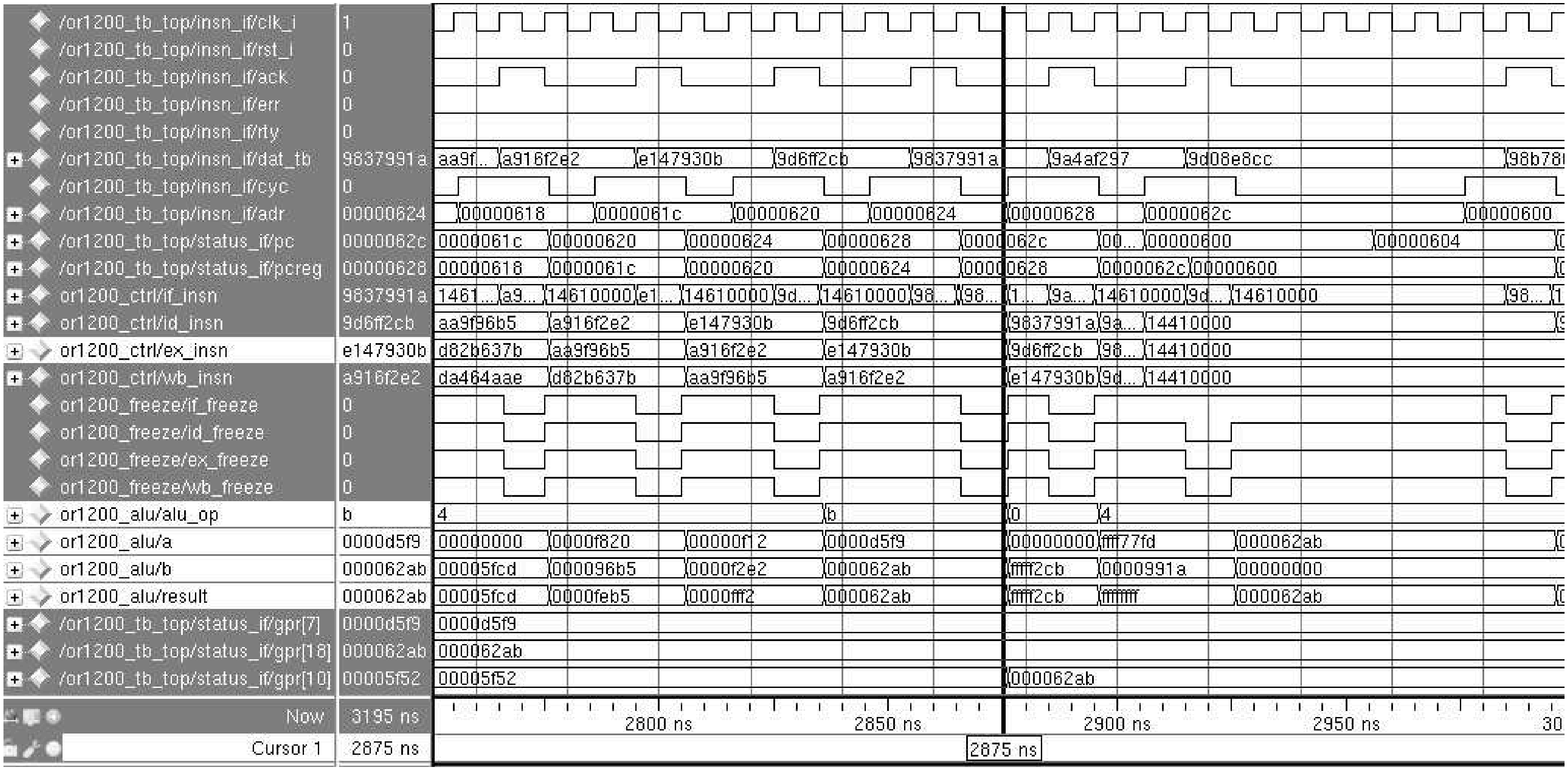
```
/*******or1200_defines.v*****/
//
// ALUOPs
//
`define OR1200_ALUOP_IMM   4'd11

/*********or1200_ctrl.v*******/
//
// Decode of alu_op
//
always @(posedge clk or posedge rst) begin
  if (rst)
    alu_op <= #1 `OR1200_ALUOP_NOP;
  else if (!ex_freeze & id_freeze | flushpipe)
    alu_op <= #1 `OR1200_ALUOP_NOP;
  else if (!ex_freeze) begin
    case (id_insn[31:26])   // synopsys parallel_case
```

```
      // ALU instructions except the one with immediate
      `OR1200_OR32_ALU:
        alu_op <= #1 id_insn[3:0];

/*******or1200_alu.v*****/

//
// Central part of the ALU
//

`ifdef OR1200_CASE_DEFAULT
  casex (alu_op)     // synopsys parallel_case
`else
  casex (alu_op)     // synopsys full_case parallel_case
`endif
    `OR1200_ALUOP_IMM : begin
        result = b;
    end
```

Conslusion

The instruction `l.mulu` is neither implemented in the ISS nor in the OR1200 core. It belongs to the ORBIS32-I instruction class of the OpenRISC1000 architecture and all instructions belonging to this class are compulsory to implement [11]. Despite that, the OR1200 core does not generate an illegal exception but executes a wrong implementation instead of the instruction `l.mulu`. Hence, this instruction cannot be included in the main verification test of the OR1200.

Several benchmark programs were compiled using OR32 C/C++ compiler but it did not generate the instruction `l.mulu`. This means that the compiler does not either implement this instruction or often use it. This is the reason why this error stayed unidentified before. However, the OR32 assembler could assemble code that uses this instruction.



/or1200_tb_top/insn_if/clk_i	1
/or1200_tb_top/insn_if/rst_i	0
/or1200_tb_top/insn_if/ack	0
/or1200_tb_top/insn_if/err	0
/or1200_tb_top/insn_if/rty	0
/or1200_tb_top/insn_if/dat_tb	9837991a
/or1200_tb_top/insn_if/cyc	0
/or1200_tb_top/insn_if/adr	00000624
/or1200_tb_top/status_if/pc	0000062c
/or1200_tb_top/status_if/pcreg	00000628
or1200_ctrl/if_insn	9837991a
or1200_ctrl/id_insn	9d6ff2cb
or1200_ctrl/ex_insn	e147930b
or1200_ctrl/wb_insn	a916f2e2
or1200_freeze/if_freeze	0
or1200_freeze/id_freeze	0
or1200_freeze/ex_freeze	0
or1200_freeze/wb_freeze	0
or1200_alu/alu_op	b
or1200_alu/a	0000d5f9
or1200_alu/b	000062ab
or1200_alu/result	000062ab
/or1200_tb_top/status_if/gpr[7]	0000d5f9
/or1200_tb_top/status_if/gpr[18]	000062ab
/or1200_tb_top/status_if/gpr[10]	00005f52
Now	3195 ns
Cursor 1	2875 ns

Figure 5.25 Simulation results of `l.mulu` on the OR1200 core.

5.5.9 Unimplemented Overflow Flag (OV)

The overflow flag is the 11^{th} bit of the SR register (considering the initial index to be zero). According to the OR1200 architectural manual a number of instructions (e.g., `l.add`, `l.sub`, `l.mul` etc.) can alter this flag. But if we look into the implementation of the OR1200 ALU and the SR (given below), there is no implementation of the overflow flag.

```
/*****or1200_define.v*****/
`define OR1200_SR_OV   11 // Unused

/*****or1200_sprs.v******/
//
// What to write into SR
//
assign to_sr[`OR1200_SR_FO:`OR1200_SR_OV] =
    (branch_op == `OR1200_BRANCHOP_RFE) ? esr[`OR1200_SR_FO:`OR1200_SR_OV] :
    (write_spr && sr_sel) ? {1'b1, spr_dat_o[`OR1200_SR_FO-1:`OR1200_SR_OV]}:
    sr[`OR1200_SR_FO:`OR1200_SR_OV];
//
// Supervision register
//
always @(posedge clk or posedge rst)
  if (rst)
    sr <= #1 {1'b1, `OR1200_SR_EPH_DEF, {`OR1200_SR_WIDTH-3{1'b0}}, 1'b1};
  else if (except_started) begin
    sr[`OR1200_SR_SM]  <= #1 1'b1;
    sr[`OR1200_SR_TEE] <= #1 1'b0;
    sr[`OR1200_SR_IEE] <= #1 1'b0;
    sr[`OR1200_SR_DME] <= #1 1'b0;
    sr[`OR1200_SR_IME] <= #1 1'b0;
  end
  else if (sr_we)
    sr <= #1 to_sr[`OR1200_SR_WIDTH-1:0];

/*****or1200_alu.v******/
  casex (alu_op)    // synopsys parallel_case
`else
  casex (alu_op)    // synopsys full_case parallel_case
`endif
    `OR1200_ALUOP_ADD : begin
        result = result_sum;
    end
`ifdef OR1200_IMPL_ADDC
    `OR1200_ALUOP_ADDC : begin
        result = result_csum;
    end
`endif
    `OR1200_ALUOP_SUB : begin
        result = a - b;
    end
`ifdef OR1200_MULT_IMPLEMENTED
`ifdef OR1200_IMPL_DIV
    `OR1200_ALUOP_DIV,
    `OR1200_ALUOP_DIVU,
`endif
    `OR1200_ALUOP_MUL : begin
        result = mult_mac_result;
    end
`endif
```

5.6 Discrepancies Between OR1200 and Golden Model

5.6.1 Overview

This section presents the acquired results corresponding to the discrepancies between the OR1200 core and its ISS. These discrepancies are as given below.

- An instruction is implemented in the OR1200 core but not in the ISS or vice versa.
- An instruction is implemented in the OR1200 core and in the ISS but its behavior is not the same.
- An instruction is not working correctly in the ISS.

All found discrepancies between the DUV and the ISS are presented in the following subsections.

5.6.2 Jump Register and Link (l.jalr) and Jump Register (l.jr) Instructions

The effective address for these jump instructions is the content of GPR rB. Both instructions have a delay slot. The program unconditionally jumps to this effective address. In case of the instruction `l.jalr`, the address of the instruction after the delay slot instruction is placed into the link register (GPR 9). The link register is not allowed to be used as rB in the instruction `l.jalr`.

l.jalr																															
31	.	.	.	.	26	25	.	.	.	.	.	.	.	.	16	15	.	.	.	11	10	.	.	.	.	.	.	.	.	.	0
Opcode 0x12						Reserved										B					Reserved										
6 bits						10 bits										5 bits					11 bits										

l.jr																															
31	.	.	.	.	26	25	.	.	.	.	.	.	.	.	16	15	.	.	.	11	10	.	.	.	.	.	.	.	.	.	0
Opcode 0x11						Reserved										B					Reserved										
6 bits						10 bits										5 bits					11 bits										

ISS implementation of `l.jalr` and `l.jr`

The ISS implementation of both instructions is provided in Appendix (A.3). The implementation does not include any exception handling if these instructions try to jump to an unaligned address. Both instructions set the pc_delay (the address of the delay slot instruction) with the content of GPR rB. The instruction `l.jalr` additionally stores the address of the instruction after the delay slot instruction in the link register (GPR 9).

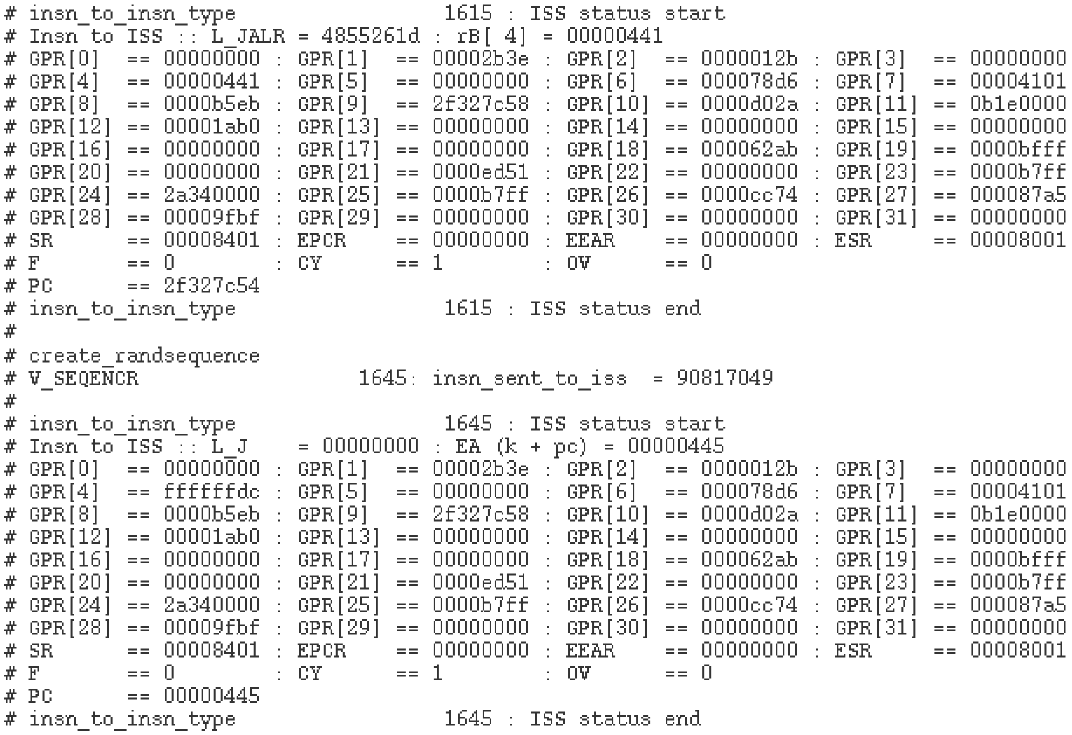
The implementation of the upcall function (`generic_read_word`) is given below. This upcall is used to fetch a new instruction or data (for Loads) from the address space of a generic device. The implementation shows that whenever there is an unaligned word access to fetch a new instruction or to load a new data word, this upcall gives an error on the standard output. Further, an unaligned access forbids the call to the `ext_read()` function (upcall to the public interface of the ISS to read from an external peripheral). Hence, the upcall is never generated and `generic_read_word` returns 0x0000_0000, which is taken by the ISS as the next instruction to execute. According to the ORBIS32-I instruction class, 0x0000_0000 is a simple jump instruction (`l.j`) followed by a delay slot. The PC address calculation for the next instruction (the delay slot instruction) is PC + 4. Hence, the next PC is also an unaligned address.

```
/*********** generic.c ***********/

static uint32_t generic_read_word (oraddr_t addr, void *dat)
{
  struct dev_generic *dev = (struct dev_generic *) dat;

  if (!config.ext.class_ptr)
    {
      fprintf (stderr, "Full word read from disabled generic device\n");
      return 0;
    }
  else if (addr >= dev->size)
    {
      fprintf (stderr, "Full word read  out of range for generic device %s "
         "(addr %" PRIxADDR ")\n", dev->name, addr);
      return 0;
    }
  else if (0 != (addr & 0x3))
    {
      fprintf (stderr,
         "Unaligned full word read from 0x%" PRIxADDR " ignored\n",
         addr);
      return 0;
    }
  else
    {
      unsigned long wordaddr = (unsigned long int) (addr + dev->baseaddr);
      return (uint32_t) htoml (ext_read (wordaddr, 0xffffffff));
    }
}       /* generic_read_word() */
```

Note: The effective address (EA) for other jump instructions (e.g., `l.bnf`, `l.jal` etc.) is not calculated from the content of any GPR. It is calculated from an immediate value which is shifted right 2 bits (word aligned) before calculating an EA. Hence, these instructions always generate a word aligned access.

Results

As seen in Figure [5.26], the instruction `l.jalr` (0x4855_261d) is sent to the ISS when requesting an instruction from address 0x2f32_7c50. The instruction uses the content of GPR rB[4] (0x0000_0441) as an EA for the jump. After the execution of this instruction the PC address for the delay slot instruction is PC + 4 (0x2f32_7c54). The address of the next instruction to be executed after the delay slot instruction is 0x0000_0441 (EA). The link register (GPR r9) is set to 0x2f32_7c58. It is the address of the instruction after the delay slot instruction. The ISS then makes an upcall (`generic_read_word`) to fetch the next instruction to be executed in the delay slot. The given transcript shows that the next instruction sent to the ISS (at time 1645 ns) is a “load byte signed” (`l.lbs`) instruction (0x9081_7049). After the execution of this instruction in the delay slot, the next instruction is fetched from 0x0000_0441 (PC). This results in an unaligned address access. As discussed earlier, the upcall (`generic_read_word`) returns zero in response to an unaligned address access that is actually the instruction `l.j`. The next PC address for the delay slot of this new jump instruction is PC + 4 (0x00000441 + 4 = 0x00000445). As the new calculated PC is again an unaligned address, it will result in another jump instruction generated by the upcall (`generic_read_word`). This new jump instruction will again generate an unaligned PC address (PC + 4 = 0x00000449) for its delay slot instruction. This is again an unaligned address and it will result in another jump instruction. This never-ending loop of calculating the unaligned PC address and generation of the jump instruction by the upcall (`generic_read_word`) never stops. Consequently, the ISS never generates the upcall needed to fetch a new instruction from the test bench. This can be seen from the received ISS status after sending the instruction `l.lbs` (0x9081_7049) at time 1645 ns. The status shows that the virtual sequencer (`V_SEQENCR`) sent the delay slot instruction `l.lbs` to the ISS. However, the status returned from the ISS shows that it was the instruction `l.j` (0x0000_0000) instead of the instruction `l.lbs`. It is because of the instruction `l.jalr` which jumped to an unaligned address (0x0000_0441). Further, the unchanged time (1645 ns) confirms that the ISS is stuck in a never-ending loop.

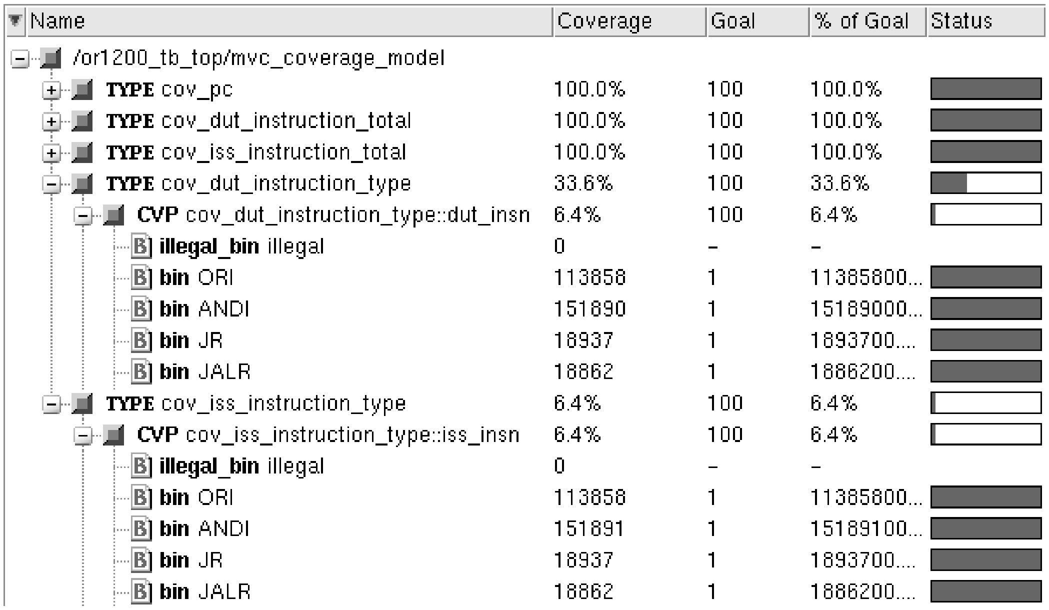


```
# insn_to_insn_type                  1615 : ISS status start
# Insn to ISS :: L_JALR = 4855261d : rB[ 4] = 00000441
# GPR[0]  == 00000000 : GPR[1]  == 00002b3e : GPR[2]  == 0000012b : GPR[3]  == 00000000
# GPR[4]  == 00000441 : GPR[5]  == 00000000 : GPR[6]  == 000078d6 : GPR[7]  == 00004101
# GPR[8]  == 0000b5eb : GPR[9]  == 2f327c58 : GPR[10] == 0000d02a : GPR[11] == 0b1e0000
# GPR[12] == 00001ab0 : GPR[13] == 00000000 : GPR[14] == 00000000 : GPR[15] == 00000000
# GPR[16] == 00000000 : GPR[17] == 00000000 : GPR[18] == 000062ab : GPR[19] == 0000bfff
# GPR[20] == 00000000 : GPR[21] == 0000ed51 : GPR[22] == 00000000 : GPR[23] == 0000b7ff
# GPR[24] == 2a340000 : GPR[25] == 0000b7ff : GPR[26] == 0000cc74 : GPR[27] == 000087a5
# GPR[28] == 00009fbf : GPR[29] == 00000000 : GPR[30] == 00000000 : GPR[31] == 00000000
# SR      == 00008401 : EPCR    == 00000000 : EEAR    == 00000000 : ESR     == 00008001
# F       == 0        : CY      == 1        : OV      == 0
# PC      == 2f327c54
# insn_to_insn_type                  1615 : ISS status end
#
# create_randsequence
# V_SEQENCR                   1645: insn_sent_to_iss  = 90817049
#
# insn_to_insn_type                  1645 : ISS status start
# Insn to ISS :: L_J    = 00000000 : EA (k + pc) = 00000445
# GPR[0]  == 00000000 : GPR[1]  == 00002b3e : GPR[2]  == 0000012b : GPR[3]  == 00000000
# GPR[4]  == ffffffdc : GPR[5]  == 00000000 : GPR[6]  == 000078d6 : GPR[7]  == 00004101
# GPR[8]  == 0000b5eb : GPR[9]  == 2f327c58 : GPR[10] == 0000d02a : GPR[11] == 0b1e0000
# GPR[12] == 00001ab0 : GPR[13] == 00000000 : GPR[14] == 00000000 : GPR[15] == 00000000
# GPR[16] == 00000000 : GPR[17] == 00000000 : GPR[18] == 000062ab : GPR[19] == 0000bfff
# GPR[20] == 00000000 : GPR[21] == 0000ed51 : GPR[22] == 00000000 : GPR[23] == 0000b7ff
# GPR[24] == 2a340000 : GPR[25] == 0000b7ff : GPR[26] == 0000cc74 : GPR[27] == 000087a5
# GPR[28] == 00009fbf : GPR[29] == 00000000 : GPR[30] == 00000000 : GPR[31] == 00000000
# SR      == 00008401 : EPCR    == 00000000 : EEAR    == 00000000 : ESR     == 00008001
# F       == 0        : CY      == 1        : OV      == 0
# PC      == 00000445
# insn_to_insn_type                  1645 : ISS status end
```

Figure 5.26 Problem with `l.jalr` and `l.jr` in the ISS.

Conclusion

For a simple test, in addition to the instructions `l.jalr` and `l.jr`, we generated the instructions (`l.ori` and `l.andi`) with word aligned immediate values to get a world aligned content of all GPRs. The coverage results of this test (Figure [5.27]) shows that both instructions (`l.jalr` and `l.jr`) work correctly in the OR1200 core when using aligned accesses. It should be noted that the OR1200 core never generates unaligned accesses to the instruction memory. It uses 30 MSBs of the PC register to fetch a new instruction thus the address is always a word aligned. Hence, in the presence of the discrepancy we discussed with the instructions `l.jr` and `l.jalr` it is not possible to include them in the main verification test of the OR1200 core.



Name	Coverage	Goal	% of Goal	Status
/or1200_tb_top/mvc_coverage_model				
TYPE cov_pc	100.0%	100	100.0%	
TYPE cov_dut_instruction_total	100.0%	100	100.0%	
TYPE cov_iss_instruction_total	100.0%	100	100.0%	
TYPE cov_dut_instruction_type	33.6%	100	33.6%	
CVP cov_dut_instruction_type::dut_insn	6.4%	100	6.4%	
illegal_bin illegal	0	-	-	
bin ORI	113858	1	11385800...	
bin ANDI	151890	1	15189000...	
bin JR	18937	1	1893700....	
bin JALR	18862	1	1886200....	
TYPE cov_iss_instruction_type	6.4%	100	6.4%	
CVP cov_iss_instruction_type::iss_insn	6.4%	100	6.4%	
illegal_bin illegal	0	-	-	
bin ORI	113858	1	11385800...	
bin ANDI	151891	1	15189100...	
bin JR	18937	1	1893700....	
bin JALR	18862	1	1886200....	

Figure 5.27 Verification coverage results of `l.jalr` and `l.jr`.

5.6.3 Add Immediate Signed and Carry (l.addic) Instruction

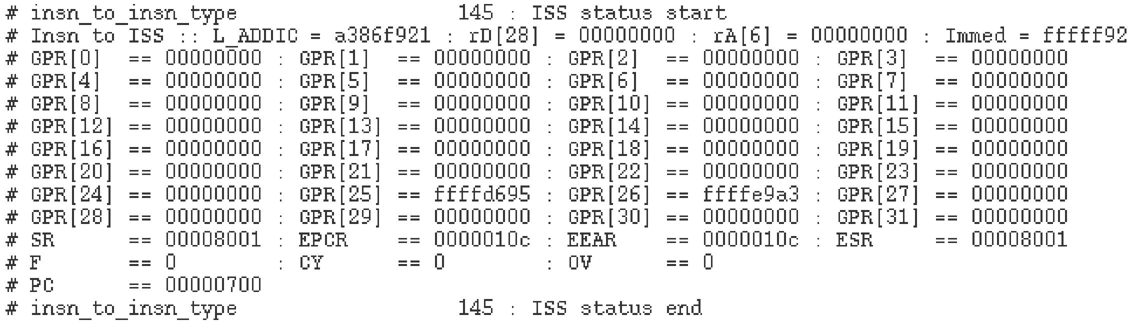
A sign extended immediate value is added to the content of GPR rA. The carry flag (SR[CY]) is also added and finally the result is stored into GPR rD.

l.addic			
31 26	25 . . . 21	20 . . . 16	15 0
Opcode 0x28	D	A	Immediate
6 bits	5 bits	5 bits	16 bits

Problem Discussion

The instruction `l.addic` is not implemented in the ISS. It generates an illegal exception[2] when executing this instruction, as shown in Figure [5.28].

[2] The next instruction is fetched from the exception vector (0x0000_0700).



```
# insn_to_insn_type                  145 : ISS status start
# Insn to ISS :: L_ADDIC = a386f921 : rD[28] = 00000000 : rA[6] = 00000000 : Immed = fffff92
# GPR[0]  == 00000000 : GPR[1]  == 00000000 : GPR[2]  == 00000000 : GPR[3]  == 00000000
# GPR[4]  == 00000000 : GPR[5]  == 00000000 : GPR[6]  == 00000000 : GPR[7]  == 00000000
# GPR[8]  == 00000000 : GPR[9]  == 00000000 : GPR[10] == 00000000 : GPR[11] == 00000000
# GPR[12] == 00000000 : GPR[13] == 00000000 : GPR[14] == 00000000 : GPR[15] == 00000000
# GPR[16] == 00000000 : GPR[17] == 00000000 : GPR[18] == 00000000 : GPR[19] == 00000000
# GPR[20] == 00000000 : GPR[21] == 00000000 : GPR[22] == 00000000 : GPR[23] == 00000000
# GPR[24] == 00000000 : GPR[25] == ffffd695 : GPR[26] == ffffe9a3 : GPR[27] == 00000000
# GPR[28] == 00000000 : GPR[29] == 00000000 : GPR[30] == 00000000 : GPR[31] == 00000000
# SR      == 00008001 : EPCR    == 0000010c : EEAR    == 0000010c : ESR     == 00008001
# F       == 0        : CY      == 0        : OV      == 0
# PC      == 00000700
# insn_to_insn_type                  145 : ISS status end
```

Figure 5.28 Instruction `l.addic` generates an illegal exception at the ISS.

Conclusion

Since the instruction `l.addic` is not implemented in the ISS (golden model), we have to exclude it from the exhaustive verification of the OR1200 core. The instruction is implemented in the OR1200 core but its correctness is not proven because it is not included in the exhaustive verification test. Since the carry flag is not controlled by the freeze logic in the OR1200 core (see Subsection 5.5.3), this problem should also be considered for the instruction `l.addic`. Figure [5.29] shows a correctly working example of the instruction `l.addic` on the OR1200 core. At time 195 ns, the instruction `l.addic` (0xa386_f921) is in the execution stage (ex_insn). The destination register is rD[28], register operand one is rA[6] (0x0000_0000) and the immediate value is 0xffff_f921. The carry flag is zero at the time this instruction executes. After the execution of this instruction the correct result (0xffff_f921) is stored into the destination register rD[28] at time 225 ns. At time 225 ns another instruction `l.addic` (0xa3dc_5381) is in the execution stage. The destination register is rD[30], register operand one is rA[28] (0xffff_f921) and the immediate vlaue is 0x0000_5381. The carry flag is zero. After the execution of this instruction the correct result (0x0000_4ca2) is stored into the destination register rD[30] at time 255 ns. The carry flag is correctly set.

Since the instruction `l.addic` is not implemented in the ISS, it is excluded from the main verification test of the OR1200 core.

Several benchmark programs were compiled using OR32 C/C++ compiler but it did not generate the instruction `l.addic`. This means that the compiler does not either implement this instruction or often generate it. This is the reason why this error stayed unidentified before. However, the OR32 assembler could assemble code that uses this instruction.

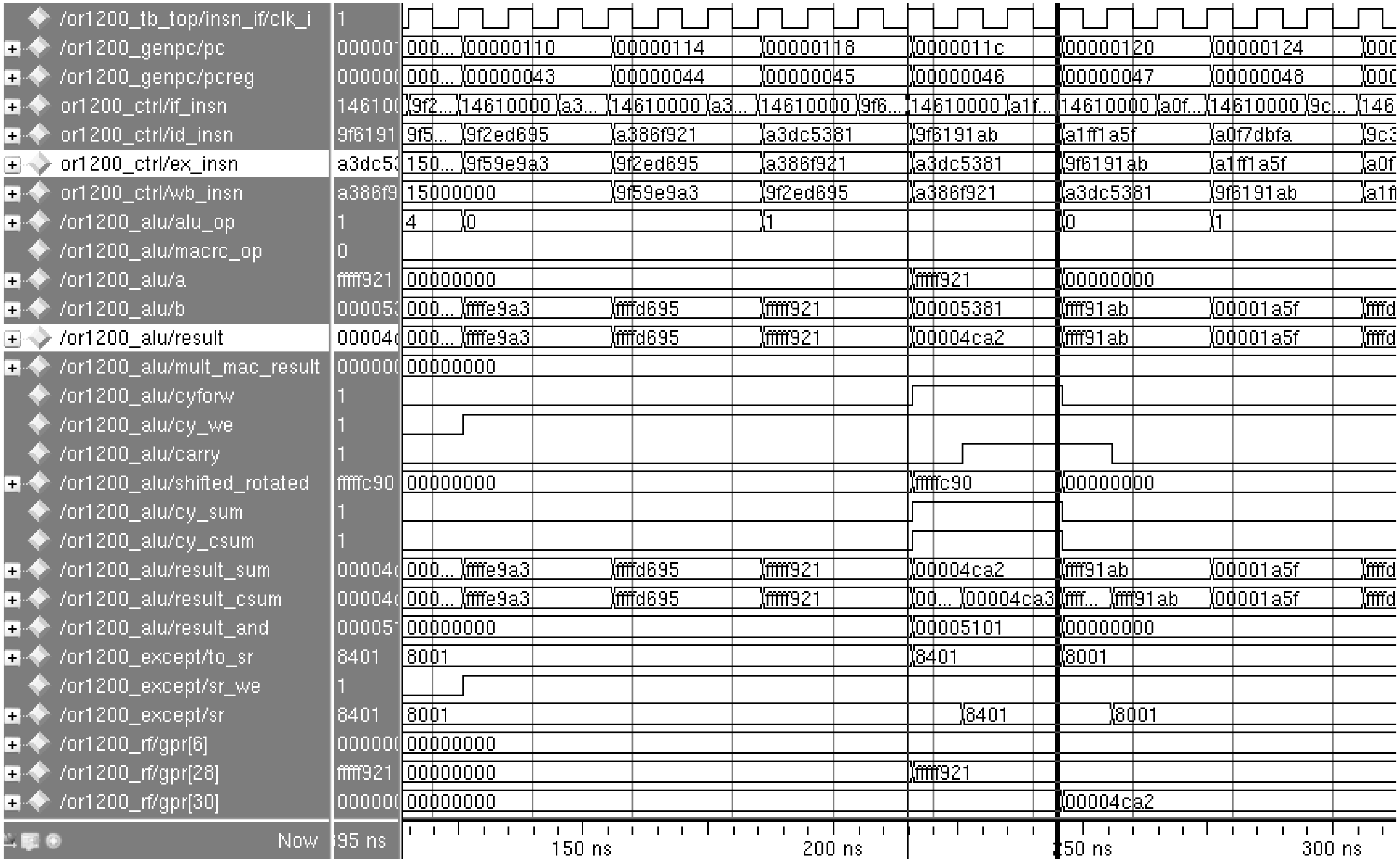


Figure 5.29 Simulation results of `l.addic` on the OR1200 core.

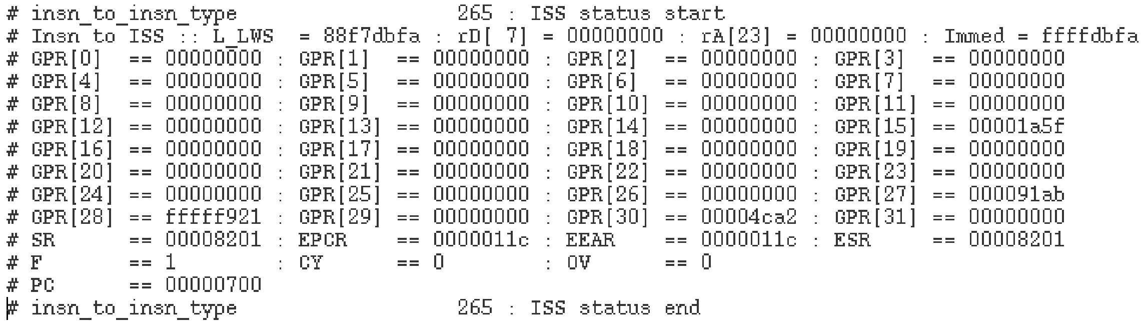
5.6.4 Load Single Word and Extend with Sign (l.lws) Instruction

A sign extended immediate value is added to the content of GPR rA to calculate the effective address (EA). A word in the memory is addressed and loaded into GPR rD using this EA.

<table>
<tr><th colspan="32">l.lws</th></tr>
<tr><td>31</td><td>.</td><td>.</td><td>.</td><td>.</td><td>26</td><td>25</td><td>.</td><td>.</td><td>.</td><td>21</td><td>20</td><td>.</td><td>.</td><td>.</td><td>16</td><td>15</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>0</td></tr>
<tr><td colspan="6">Opcode 0x22</td><td colspan="5">D</td><td colspan="5">A</td><td colspan="16">Immediate</td></tr>
<tr><td colspan="6">6 bits</td><td colspan="5">5 bits</td><td colspan="5">5 bits</td><td colspan="16">16 bits</td></tr>
</table>

ISS implementation of `l.lws`

The ISS implementation shows that the instruction `l.lws` is an invalid instruction which is not implemented in the ISS so far. Whenever this instruction is sent to the ISS, it generates an illegal exception. Figure [5.30] shows that when the instruction `l.lws` (0x88f7_dbfa) was sent to the ISS, it generated an illegal exception (the next instruction is fetched from 0x0000_0700).



```
# insn_to_insn_type                  265 : ISS status start
# Insn to ISS :: L_LWS  = 88f7dbfa : rD[ 7] = 00000000 : rA[23] = 00000000 : Immed = ffffdbfa
# GPR[0]  == 00000000 : GPR[1]  == 00000000 : GPR[2]  == 00000000 : GPR[3]  == 00000000
# GPR[4]  == 00000000 : GPR[5]  == 00000000 : GPR[6]  == 00000000 : GPR[7]  == 00000000
# GPR[8]  == 00000000 : GPR[9]  == 00000000 : GPR[10] == 00000000 : GPR[11] == 00000000
# GPR[12] == 00000000 : GPR[13] == 00000000 : GPR[14] == 00000000 : GPR[15] == 00001a5f
# GPR[16] == 00000000 : GPR[17] == 00000000 : GPR[18] == 00000000 : GPR[19] == 00000000
# GPR[20] == 00000000 : GPR[21] == 00000000 : GPR[22] == 00000000 : GPR[23] == 00000000
# GPR[24] == 00000000 : GPR[25] == 00000000 : GPR[26] == 00000000 : GPR[27] == 000091ab
# GPR[28] == fffff921 : GPR[29] == 00000000 : GPR[30] == 00004ca2 : GPR[31] == 00000000
# SR      == 00008201 : EPCR    == 0000011c : EEAR    == 0000011c : ESR     == 00008201
# F       == 1        : CY      == 0        : OV      == 0
# PC      == 00000700
# insn_to_insn_type                  265 : ISS status end
```

Figure 5.30 Instruction `l.lws` generates an illegal exception at the ISS.

OR1200 implementation of `l.lws`

The instruction `l.lws` is not implemented in the OR1200 core. Therefore, whenever it is sent to the OR1200 core, it generates an illegal instruction exception. The waveform in Figure [5.31] shows that the instruction `l.lws` (0x88f7_dbfa) is in the execution stage at time 315 ns in the OR1200 core. The except_illegal signal is high which indicates that an illegal instruction is in the execution stage. An illegal exception results in accessing the next instruction from the illegal_exception vector (0x0000_0700). The instruction `l.lws` is a load instruction and supposed to be executed on the LSU. However, since it is not implemented in the OR1200 core, the lsu_op (LSU opcode) is a NOP at time 315 ns.

Conclusion

The instruction `l.lws` is neither implemented in the ISS nor in the OR1200 core. It belongs to the ORBIS32-I instruction class of the OpenRISC1000 architecture and all instructions belonging to this class are compulsory to implement [11]. This instruction cannot be included in the main verification test of the OR1200 core.

Several benchmark programs were compiled using OR32 C/C++ compiler but it did not generate the instruction `l.lws`. This means that the compiler does not either implement this instruction or often generate it. This is the reason why this error stayed unidentified before. However, the OR32 assembler could assemble code that uses this instruction.

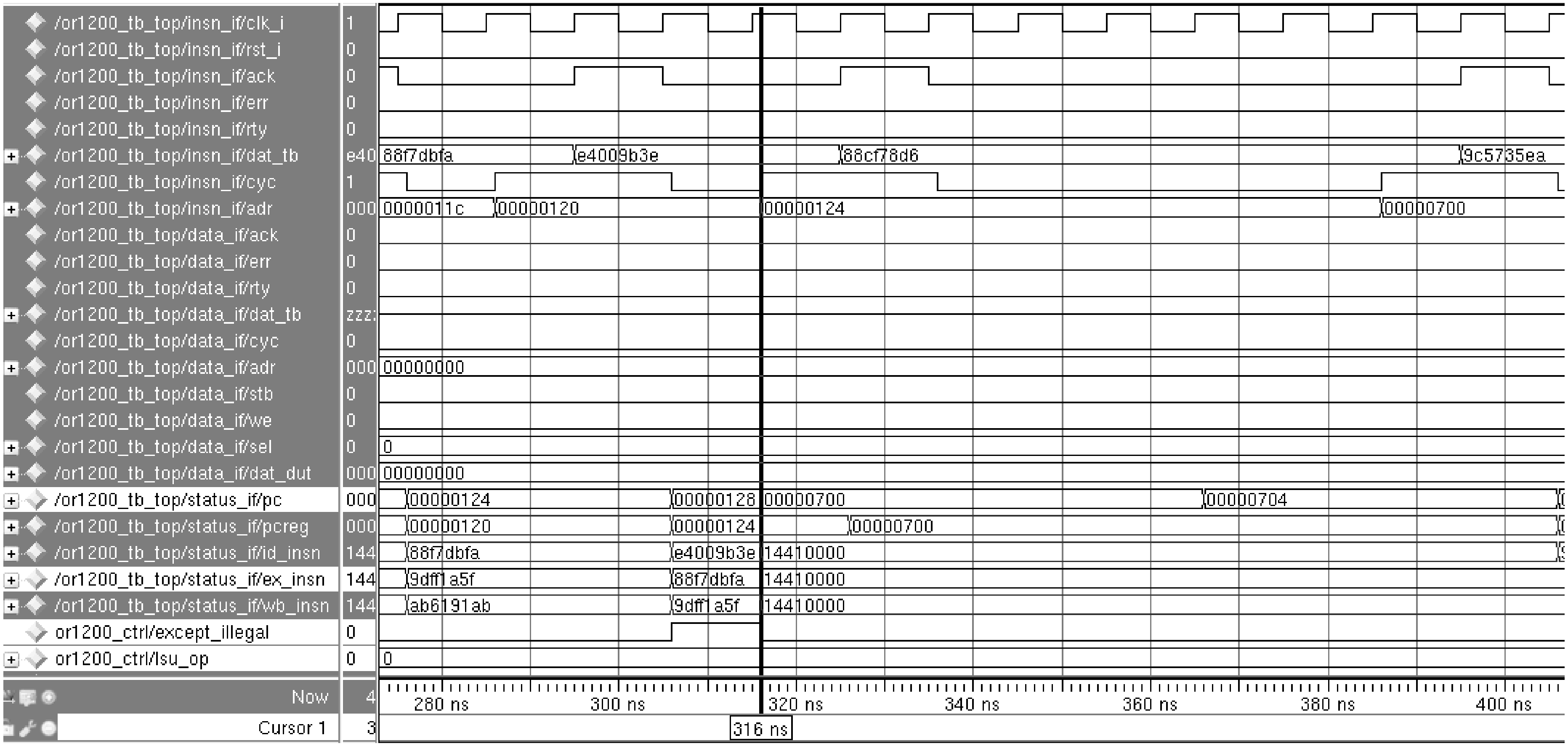

/or1200_tb_top/insn_if/clk_i 1
/or1200_tb_top/insn_if/rst_i 0
/or1200_tb_top/insn_if/ack 0
/or1200_tb_top/insn_if/err 0
/or1200_tb_top/insn_if/rty 0
/or1200_tb_top/insn_if/dat_tb e40 88f7dbfa e4009b3e 88cf78d6 9c5735ea
/or1200_tb_top/insn_if/cyc 1
/or1200_tb_top/insn_if/adr 000 0000011c 00000120 00000124 00000700
/or1200_tb_top/data_if/ack 0
/or1200_tb_top/data_if/err 0
/or1200_tb_top/data_if/rty 0
/or1200_tb_top/data_if/dat_tb zzz
/or1200_tb_top/data_if/cyc 0
/or1200_tb_top/data_if/adr 000 00000000
/or1200_tb_top/data_if/stb 0
/or1200_tb_top/data_if/we 0
/or1200_tb_top/data_if/sel 0 0
/or1200_tb_top/data_if/dat_dut 000 00000000
/or1200_tb_top/status_if/pc 000 00000124 00000128 00000700 00000704
/or1200_tb_top/status_if/pcreg 000 00000120 00000124 00000700
/or1200_tb_top/status_if/id_insn 144 88f7dbfa e4009b3e 14410000
/or1200_tb_top/status_if/ex_insn 144 9dff1a5f 88f7dbfa 14410000
/or1200_tb_top/status_if/wb_insn 144 ab6191ab 9dff1a5f 14410000
or1200_ctrl/except_illegal 0
or1200_ctrl/lsu_op 0 0
Now 4
280 ns
300 ns
320 ns
340 ns
360 ns
380 ns
400 ns
Cursor 1 3
316 ns

Figure 5.31 Simulation results of `l.lws` on the OR1200 core.

5.6.5 MAC Read and Clear (l.macrc) Instruction

The instruction `l.macrc` is used for the MAC unit synchronization. When all instructions in the MAC unit's pipeline are completed, the content of the MAC accumulator (MACHI, MACLO) is stored into GPR rD and the accumulator is cleared.

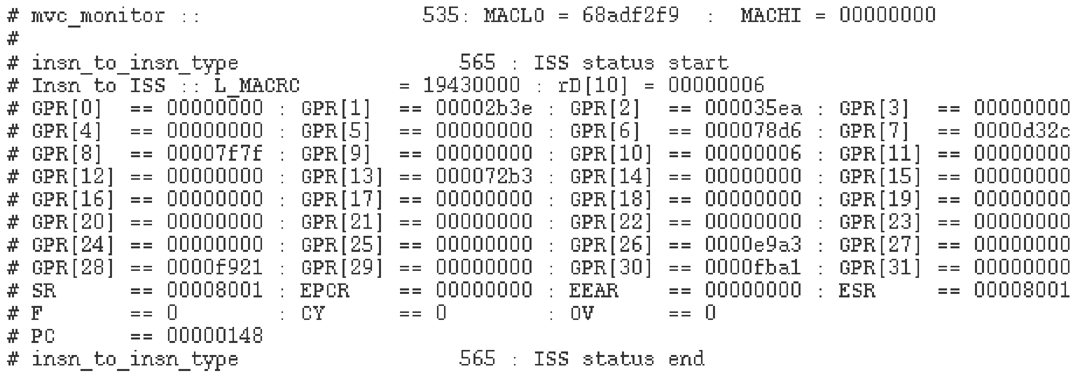
<table>
<tr><th colspan="32">l.macrc</th></tr>
<tr><td>31</td><td>.</td><td>.</td><td>.</td><td>.</td><td>26</td><td>25</td><td>.</td><td>.</td><td>.</td><td>21</td><td>20</td><td>.</td><td>.</td><td>17</td><td>16</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>0</td></tr>
<tr><td colspan="6">Opcode 0x6</td><td colspan="5">D</td><td colspan="4">Reserved</td><td colspan="17">Opcode 0x10000</td></tr>
<tr><td colspan="6">6 bits</td><td colspan="5">5 bits</td><td colspan="4">4 bits</td><td colspan="17">17bits</td></tr>
</table>

ISS implementation of `l.macrc`

The ISS implementation of the instruction `l.macrc` is given below. It shows that the values in the registers MACHI and MACLO are stored in a long variable (64-bit) which is then shifted right 28 times. This shifting is a problem since there is no reason to perform it.

```
/**********execgen.c**********/

case 0x1:
      /* Not unique: real mask fffffffffc01ffff and current mask fc010000 differ - do
          final check */
      if((insn & 0xfc01ffff) == 0x18010000) {
        /* Instruction: l.macrc */
        {
          uorreg_t a;
          /* Number of operands: 1 */
          a = (insn >> 21) & 0x1f;
          #define SET_PARAM0(val) cpu_state.reg[a] = val
          #define PARAM0 cpu_state.reg[a]
          {              /* "l_macrc" */
            uorreg_t lo, hi;
            LONGEST l;
            /* No need for synchronization here -- all MAC instructions are 1 cycle long.
                 */
            lo =  cpu_state.sprs[SPR_MACLO];
            hi =  cpu_state.sprs[SPR_MACHI];
            l = (ULONGEST) lo | ((LONGEST)hi << 32);
            l >>= 28;
            //PRINTF ("<%08x>\n", (unsigned long)l);
            SET_PARAM0((orreg_t)l);
            cpu_state.sprs[SPR_MACLO] = 0;
            cpu_state.sprs[SPR_MACHI] = 0;
          }
          #undef SET_PARAM
          #undef PARAM0

          if (do_stats) {
            current->insn_index = 6;   /* "l.macrc" */
            analysis(current);
          }
          cpu_state.reg[0] = 0; /* Repair in case we changed it */
        }
      } else {
        /* Invalid insn */
        {
          l_invalid ();
          if (do_stats) {
            current->insn_index = -1;    /* "???" */
            analysis(current);
          }
          cpu_state.reg[0] = 0; /* Repair in case we changed it */
        }
      }
      break;
    }
    break;
```

Figure [5.32] shows that the instruction `l.macrc` (0x1943_0000) is sent to the ISS and the destination register is rD[10]. By the time this instruction is executed, the register MACHI is 0x0000_0000 and the register MACLO is 0x68ad_f2f9. After the execution of this instruction the destination register rD[10] is 0x0000_0006 while the correct result is 0x68ad_f2f9. This is an implementation fault since the result is shifted right 28 times.



```
# mvc_monitor ::                  535: MACLO = 68adf2f9  :  MACHI = 00000000
#
# insn_to_insn_type                  565 : ISS status start
# Insn to ISS :: L_MACRC        = 19430000 : rD[10] = 00000006
# GPR[0]  == 00000000 : GPR[1]  == 00002b3e : GPR[2]  == 000035ea : GPR[3]  == 00000000
# GPR[4]  == 00000000 : GPR[5]  == 00000000 : GPR[6]  == 000078d6 : GPR[7]  == 0000d32c
# GPR[8]  == 00007f7f : GPR[9]  == 00000000 : GPR[10] == 00000006 : GPR[11] == 00000000
# GPR[12] == 00000000 : GPR[13] == 000072b3 : GPR[14] == 00000000 : GPR[15] == 00000000
# GPR[16] == 00000000 : GPR[17] == 00000000 : GPR[18] == 00000000 : GPR[19] == 00000000
# GPR[20] == 00000000 : GPR[21] == 00000000 : GPR[22] == 00000000 : GPR[23] == 00000000
# GPR[24] == 00000000 : GPR[25] == 00000000 : GPR[26] == 0000e9a3 : GPR[27] == 00000000
# GPR[28] == 0000f921 : GPR[29] == 00000000 : GPR[30] == 0000fba1 : GPR[31] == 00000000
# SR      == 00008001 : EPCR    == 00000000 : EEAR    == 00000000 : ESR     == 00008001
# F       == 0        : CY      == 0        : OV      == 0
# PC      == 00000148
# insn_to_insn_type                  565 : ISS status end
```

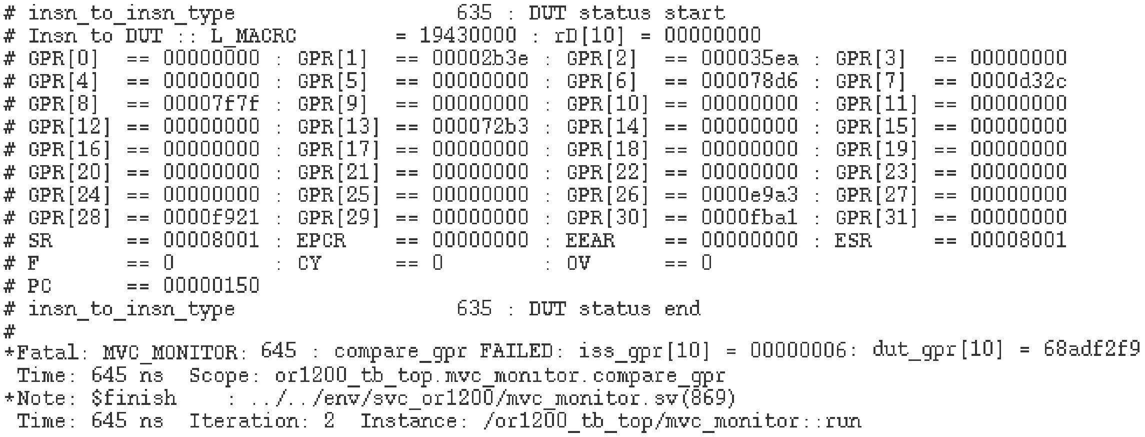
Figure 5.32 Execution results of `l.macrc` on the ISS.

The same instruction `l.macrc` (0x1943_0000) is correctly executed on the OpenRISC1200 core. Figure [5.33] shows the mismatch between the wrong result from the ISS and a correct result from the OR1200 core.

Conclusion

The instruction `l.macrc` is not working correctly in the golden model (ISS), therefore it is excluded from the main verification test of the OR1200 core. The instruction is working correctly in the OR1200 core as it generated a correct result. However, its correctness is not proven because it is not included in the exhaustive verification test.

Several benchmark programs were compiled using OR32 C/C++ compiler but it did not generate the instruction `l.macrc`. This means that the compiler does not either implement this instruction or often use it. This is the reason why this error stayed unidentified before. However, the OR32 assembler could assemble code that uses this instruction.



```
# insn_to_insn_type                  635 : DUT status start
# Insn to DUT :: L_MACRC        = 19430000 : rD[10] = 00000000
# GPR[0]  == 00000000 : GPR[1]  == 00002b3e : GPR[2]  == 000035ea : GPR[3]  == 00000000
# GPR[4]  == 00000000 : GPR[5]  == 00000000 : GPR[6]  == 000078d6 : GPR[7]  == 0000d32c
# GPR[8]  == 00007f7f : GPR[9]  == 00000000 : GPR[10] == 00000000 : GPR[11] == 00000000
# GPR[12] == 00000000 : GPR[13] == 000072b3 : GPR[14] == 00000000 : GPR[15] == 00000000
# GPR[16] == 00000000 : GPR[17] == 00000000 : GPR[18] == 00000000 : GPR[19] == 00000000
# GPR[20] == 00000000 : GPR[21] == 00000000 : GPR[22] == 00000000 : GPR[23] == 00000000
# GPR[24] == 00000000 : GPR[25] == 00000000 : GPR[26] == 0000e9a3 : GPR[27] == 00000000
# GPR[28] == 0000f921 : GPR[29] == 00000000 : GPR[30] == 0000fba1 : GPR[31] == 00000000
# SR      == 00008001 : EPCR    == 00000000 : EEAR    == 00000000 : ESR     == 00008001
# F       == 0        : CY      == 0        : OV      == 0
# PC      == 00000150
# insn_to_insn_type                  635 : DUT status end
#
*Fatal: MVC_MONITOR: 645 : compare_gpr FAILED: iss_gpr[10] = 00000006: dut_gpr[10] = 68adf2f9
 Time: 645 ns  Scope: or1200_tb_top.mvc_monitor.compare_gpr
*Note: $finish    : ../../env/svc_or1200/mvc_monitor.sv(869)
 Time: 645 ns  Iteration: 2  Instance: /or1200_tb_top/mvc_monitor::run
```

Figure 5.33 Results mismatch of `l.macrc` from the OR1200 core and the ISS.

5.6.6 Rotate Right (l.ror) Instruction

The contents of GPR rA are rotated right by the number of bit positions specified by GPR rB. The result is then placed into GPR rD.

<table>
<tr><th colspan="32">l.ror</th></tr>
<tr><td>31</td><td>.</td><td>.</td><td>.</td><td>.</td><td>26</td><td>25</td><td>.</td><td>.</td><td>.</td><td>21</td><td>20</td><td>.</td><td>.</td><td>.</td><td>16</td><td>15</td><td>.</td><td>.</td><td>.</td><td>11</td><td>10</td><td>9</td><td>.</td><td>.</td><td>6</td><td>5</td><td>4</td><td>3</td><td>.</td><td>.</td><td>0</td></tr>
<tr><td colspan="6">Opcode 0x38</td><td colspan="5">D</td><td colspan="5">A</td><td colspan="5">B</td><td colspan="1">Reserved</td><td colspan="4">Opcode 0x3</td><td colspan="2">Reserved</td><td colspan="4">Opcode 0x8</td></tr>
<tr><td colspan="6">6 bits</td><td colspan="5">5 bits</td><td colspan="5">5 bits</td><td colspan="5">5 bits</td><td colspan="1">1 bits</td><td colspan="4">4 bits</td><td colspan="2">4 bits</td><td colspan="4">4 bits</td></tr>
</table>

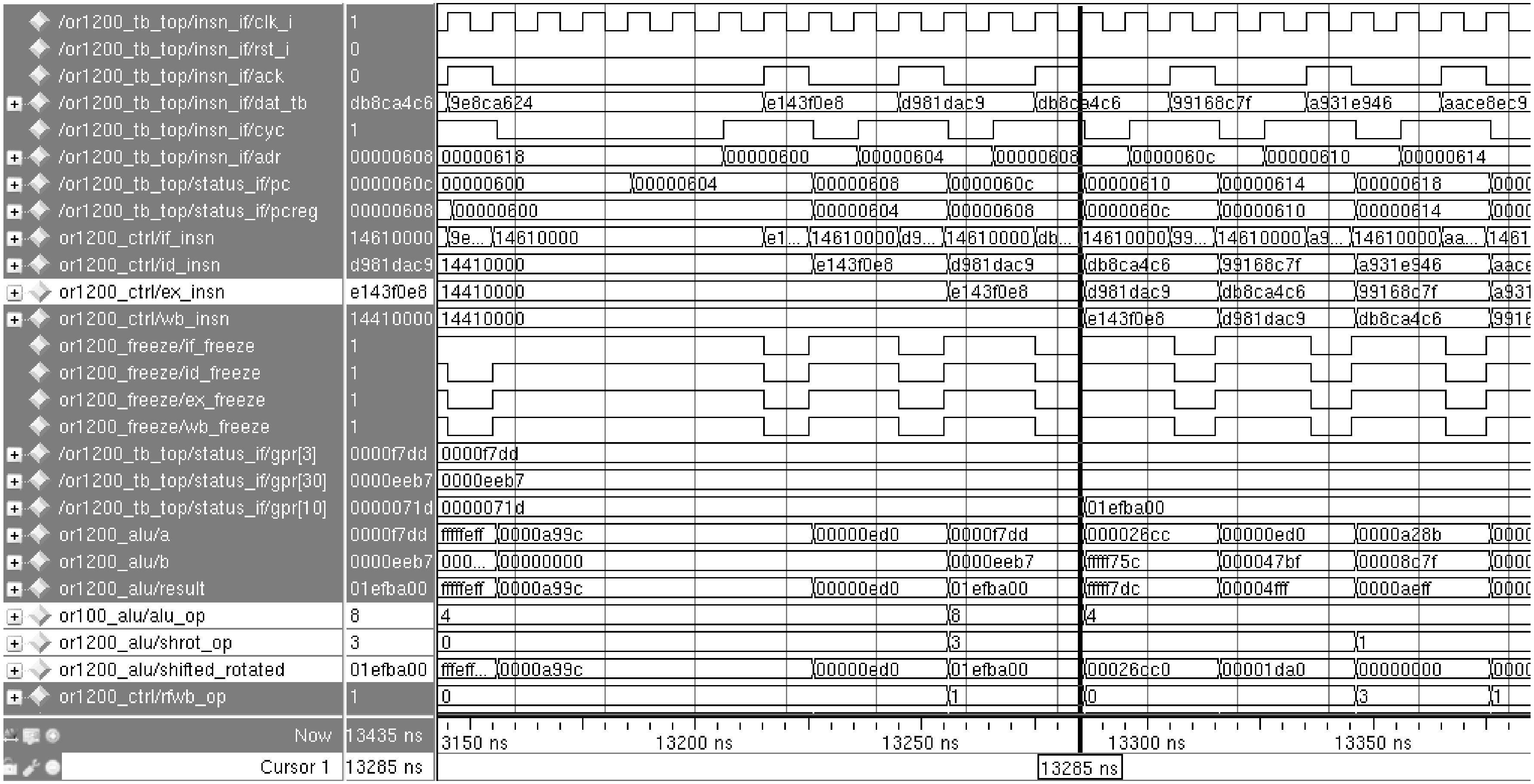
Problem Discussion

The instruction `l.ror` is not implemented in the ISS that is why it results in an illegal exception. The instruction is implemented in the OR1200 core and working correctly. When this instruction was implemented in the OR1200 core, the OR32 C/C++ compiler did not generate rotate instructions. However, the OR32 assembler could assemble code that uses rotate instructions. It means that rotate instructions must be inserted manually. By default the implementation of rotate instructions is disabled in the OR1200 core to save area and to increase the clock frequency [12]. A simulation of the instruction `l.ror` is shown in Figure [5.34] to show that it is working correctly in the OR1200 core. The waveform shows that the instruction `l.ror` (0xe143_f0e8) is in the execution stage (ex_insn) at time 13265 ns. The destination register is rD[10], register operand one is rA[3] (0x0000_f7dd) and register operand two is rB[30] (0x0000_eeb7). Operand 'a' (rA[3]) is rotated right by the number of bit positions specified in operand 'b' (rB[30]). At time 13265 ns the correct result (0x01ef_ba00) is calculated and the destination register (rD[10]) is finally updated at time 13295 ns.

Conclusion

The instruction `l.ror` is not implemented in the golden model (ISS), therefore it is excluded from the main verification test of the OR1200 core. The instruction is working correctly in the OR1200 core as it generated a correct result. However, its correctness is not proven because it is not included in the exhaustive verification test.

Since the OR32 C/C++ compiler does not generate rotate instructions, the instruction `l.ror` is not often used. This is the reason why this error stayed unidentified before.



/or1200_tb_top/insn_if/clk_i	1
/or1200_tb_top/insn_if/rst_i	0
/or1200_tb_top/insn_if/ack	0
/or1200_tb_top/insn_if/dat_tb	db8ca4c6
/or1200_tb_top/insn_if/cyc	1
/or1200_tb_top/insn_if/adr	00000608
/or1200_tb_top/status_if/pc	0000060c
/or1200_tb_top/status_if/pcreg	00000608
or1200_ctrl/if_insn	14610000
or1200_ctrl/id_insn	d981dac9
or1200_ctrl/ex_insn	e143f0e8
or1200_ctrl/wb_insn	14410000
or1200_freeze/if_freeze	1
or1200_freeze/id_freeze	1
or1200_freeze/ex_freeze	1
or1200_freeze/wb_freeze	1
/or1200_tb_top/status_if/gpr[3]	0000f7dd
/or1200_tb_top/status_if/gpr[30]	0000eeb7
/or1200_tb_top/status_if/gpr[10]	0000071d
or1200_alu/a	0000f7dd
or1200_alu/b	0000eeb7
or1200_alu/result	01efba00
or100_alu/alu_op	8
or1200_alu/shrot_op	3
or1200_alu/shifted_rotated	01efba00
or1200_ctrl/rfwb_op	1
Now	13435 ns
Cursor 1	13285 ns

Figure 5.34 Simulation results of `l.ror` on the OR1200 core.

5.6.7 Rotate Right with Immediate (l.rori) Instruction

The contents of GPR rA is rotated right by the number of bit positions specified by a 5-bit immediate value (L). The result is then placed into GPR rD.

<table>
<tr><th colspan="32">l.rori</th></tr>
<tr><td>31</td><td>.</td><td>.</td><td>.</td><td>.</td><td>26</td><td>25</td><td>.</td><td>.</td><td>.</td><td>21</td><td>20</td><td>.</td><td>.</td><td>.</td><td>16</td><td>15</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>8</td><td>7</td><td>6</td><td>5</td><td>.</td><td>.</td><td>.</td><td>.</td><td>0</td></tr>
<tr><td colspan="6">Opcode 0x2e</td><td colspan="5">D</td><td colspan="5">A</td><td colspan="8">Reserved</td><td colspan="2">Opcode 0x3</td><td colspan="6">L</td></tr>
<tr><td colspan="6">6 bits</td><td colspan="5">5 bits</td><td colspan="5">5 bits</td><td colspan="8">8 bits</td><td colspan="2">2 bits</td><td colspan="6">6 bits</td></tr>
</table>

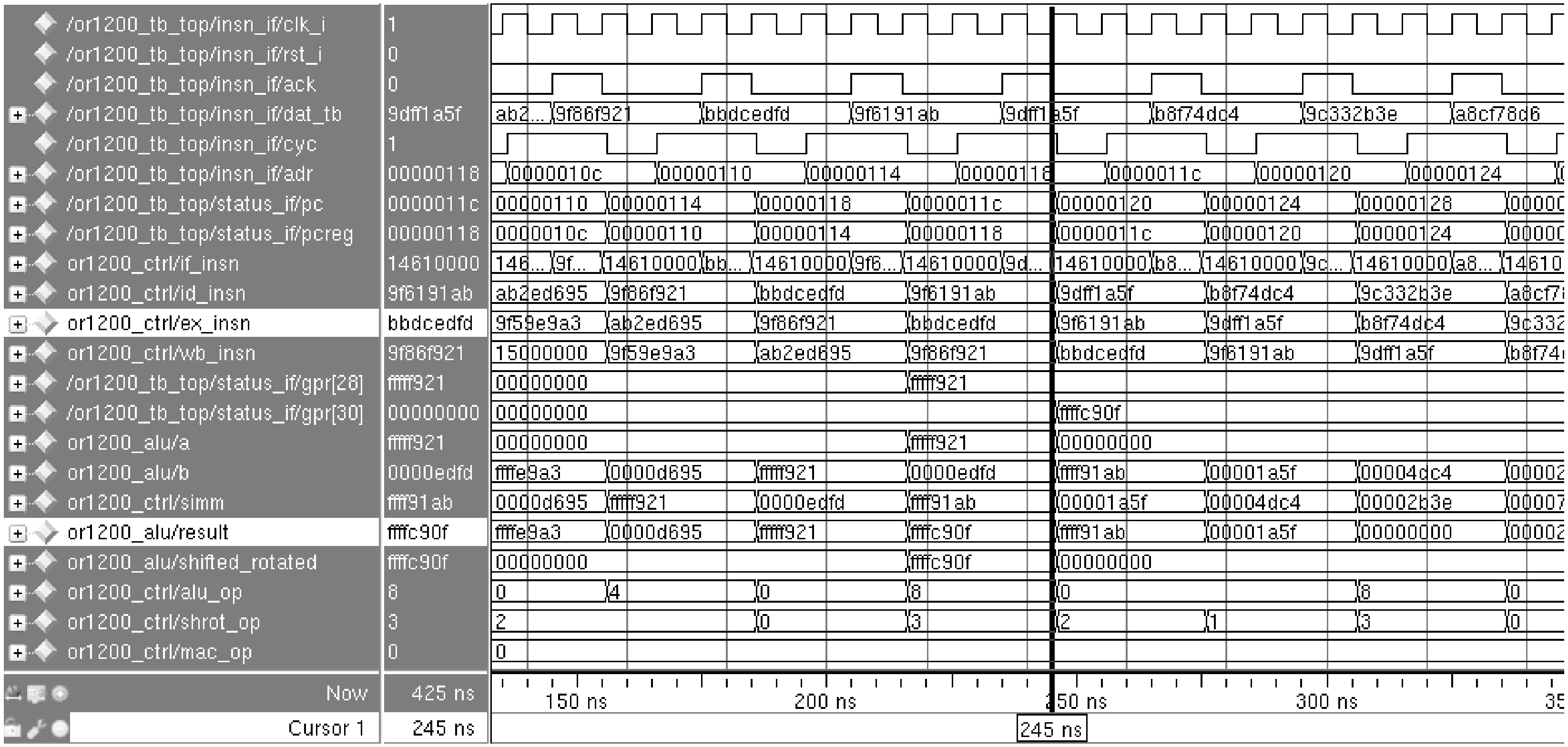
Problem Discussion

The instruction `l.rori` is not implemented in the ISS that is why it results in an illegal exception. The instruction is implemented in the OR1200 core and working correctly. A simulation of the instruction `l.rori` is shown in Figure [5.35] to show that it is correctly working in the OR1200 core. When the instruction is `l.rori`, operand ‘b’ is a 16-bit immediate value and its last 5 bits contain the value by which operand ‘a’ is rotated right. The waveform shows that the instruction `l.rori` (0xbbdc_edfd) is in the execution stage (ex_insn) at time 225 ns. The destination register is rD[30], register operand one is rA[28] (0xffff_f921) and the immediate value = 0x0000_001d (L) (last 5 bits of a 16-bit immediate value i.e., 0x0000_edfd). Operand ‘a’ (rA[28]) is rotated right by the number of bit positions specified by operand ‘b’ (L). At time 225 ns, the correct result (0xffff_c90f) is calculated and then stored into the destination register (rD[30]) at time 255 ns.

Conclusion

The instruction `l.rori` is not implemented in the golden model (ISS), therefore it is excluded from the main verification test of the OR1200 core. The instruction is working correctly in the OR1200 core as it generated a correct result. However, its correctness is not proven because it is not included in the exhaustive verification test.

Since the OR32 C/C++ compiler does not generate rotate instructions, the instruction `l.rori` is not often used. This is the reason why this error stayed unidentified before.



/or1200_tb_top/insn_if/clk_i	1
/or1200_tb_top/insn_if/rst_i	0
/or1200_tb_top/insn_if/ack	0
/or1200_tb_top/insn_if/dat_tb	9dff1a5f
/or1200_tb_top/insn_if/cyc	1
/or1200_tb_top/insn_if/adr	00000118
/or1200_tb_top/status_if/pc	0000011c
/or1200_tb_top/status_if/pcreg	00000118
or1200_ctrl/if_insn	14610000
or1200_ctrl/id_insn	9f6191ab
or1200_ctrl/ex_insn	bbdcedfd
or1200_ctrl/wb_insn	9f86f921
/or1200_tb_top/status_if/gpr[28]	fffff921
/or1200_tb_top/status_if/gpr[30]	00000000
or1200_alu/a	fffff921
or1200_alu/b	0000edfd
or1200_ctrl/simm	ffff91ab
or1200_alu/result	ffffc90f
or1200_alu/shifted_rotated	ffffc90f
or1200_ctrl/alu_op	8
or1200_ctrl/shrot_op	3
or1200_ctrl/mac_op	0
Now	425 ns
Cursor 1	245 ns

Figure 5.35 Simulation results of `l.rori` on the OR1200 core.

5.6.8 Move to/from Special Purpose Registers (l.mtspr/l.mfspr)

The content of GPR rB is moved into an SPR defined by the content of GPR rA logically ORed with an immediate value.

l.mtspr				
31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 0
Opcode 0x30	K	A	B	K
6 bits	5 bits	5 bits	5 bits	11 bits

Problem Discussion

The ISS implementation of the instruction `l.mtspr` (Appendix A.4) shows that the destination SPR is not defined according to the instruction's description. It is defined by the content of operand 'a' added to operand 'c' (immediate value) instead of logically ORed. Hence, this implementation differs from the `l.mtspr` implementation in the OR1200 core where the destination SPR is defined by the content of GPR rA logically ORed with an immediate value. The instruction "*Move From Special Purpose Register*" (`l.mfspr`) has the same implementation difference between the ISS and the OR1200 core.

Conclusion

The implementation of the instructions `l.mtspr` and `l.mfspr` is different from the specification in the OpenRISC1000 architectural manual [11]. However, the OR1200 core implements both instructions according to their specification in the manual. Hence, in presence of this difference, it is not possible to include these instructions in the main verification test of the OR1200 core.

5.7 The OpenRISC1200 Verification Coverage Results

5.7.1 Overview

This section presents the verification completeness (coverage) of the OR1200 core. It should be noted that all instructions and scenarios which have a problem either in the golden model (ISS) or in the OR1200 core (DUV) are not included in this verification.

5.7.2 OR1200 Functional Verification Coverage

Figure [5.36] shows the achieved verification coverage of the OR1200 core. The verification environment is discussed in the Section (4.4). This coverage is the maximum we could achieve without generating the erroneous instructions (either in the OR1200 core or in its ISS). Table [5.1] shows the OR1200 instruction set, the erroneous instructions and different instruction formats. The description about each coverage group in the coverage model (`mvc_coverage_model`) is given below.

Program Counter Coverage

The coverage group `cov_pc` shows the achieved coverage of the PC register of the OR1200 core. It also covers whether an illegal address is accessed during the period of simulation.

OR1200 Instruction Set	Erroneous Instructions	Instruction Formats
l.addi, l.addc, l.lwz, l.sw, l.j, l.jal, l.bf, l.bnf, l.rfe, l.andi, l.ori, l.nop, l.sfeq, l.sfne, l.add, l.addic, l.cmov, l.div, l.extbs, l.mac, l.mul, l.sll, l.mfspr, l.mtspr, l.and, l.csync, l.divu, l.extbz, l.exths, l.exthz, l.ff1, l.jalr, l.jr, l.lbs, l.lbz, l.lhs, l.lhz, l.lws, l.maci, l.macrc, l.movhi, l.msb, l.msync, l.muli, l.mulu, l.or, l.psync, l.ror, l.rori, l.sb, l.sfeqi, l.sfges, l.sfgesi, l.sfgeu, l.sfgeui, l.sfgts, l.sfgtsi, l.sfgtu, l.sfgtui, l.sfles, l.sflesi, l.sfleu, l.sfleui, l.sflts, l.sfltsi, l.sfltu, l.sfltui, l.sfnei, l.sh, l.slli, l.sra, l.srai, l.srl, l.srli, l.sub, l.xor, l.xori, l.fl1 [11]	l.addc, l.jr, l.jalr, l.addic, l.div, l.divu, l.extbs, l.extbz, l.exths, l.exthz, l.mfspr, l.mtspr, l.lws, l.ror, l.rori, l.muli, l.mulu, l.maci, l.macrc, l.fl1	(A) l.insn rD, rA, rB (B) l.insn rA, rB (C) l.insn rD, rA, I *Immediate* ↦ 16 bits ([15:0]) (D) l.insn rA, I *Immediate* ↦ 16 bits ([15:0]) (E) l.insn I (rA), rB *Immediate* ↦ 16 bits ([25:21]+[10:0]) (F) l.insn rD, rA, L *Immediate* ↦ 6 bits ([5:0]) (G) l.insn N *EA* ↦ 26 bits ([25:0]) (H) l.insn rD, K (16 bits ↦ [15:0]) (I) l.insn rD, rA (J) l.insn rB (K) l.insn rD (L) l.insn rB, I *Immediate* ↦ 16 bits ([25:21]+[10:0])

Table 5.1 OR1200 instruction set.

Total Executed Instructions Coverage

The coverage group **cov_dut_instruction_total** covers the total number of instructions executed on the OR1200 core. The coverage group **cov_iss_instruction_total** covers the total number of instructions executed on the ISS and it should be equal to the core's coverage. All erroneous instructions are excluded in both coverage groups.

Instruction Specific Coverage

The coverage group **cov_dut_instruction_type** provides instruction specific verification statistics. It comprises of a number of coverage points which provide a statistical details of different aspects that are required to be verified (see Section 4.4.1). There are some coverage points needed for the cross coverage namely dest_reg, src_1_reg, src_2_reg, Immed_6_bits, Immed_16_bits, Immed_16_bits_dist, Immed_26_bits, cover_carry, cover_carry_delay, cover_flag, cover_flag_delay, cover_ov_flag, cover_ov_flag_delay, stage_1_insn, stage_2_insn, stage_3_insn, stage_4_insn.

Other coverage points are individually described below.

dut_insn: This coverage point stores the histogram of all instructions executed on the OR1200 core. In total there are 78 instructions in the OR1200 instruction set. Only 58 could be included in the verification test because the rest have problems (see Table 5.1). The reports about the erroneous instructions are provided in Section (5.5) and Section (5.6).

Equation (5.1) shows the maximum achievable verification coverage for this coverage point.

$$\textit{Maximum possible coverage} = \frac{58}{78} * 100 = 74.3\% \tag{5.1}$$

Figure [5.36] shows that the maximum possible coverage is achieved.

cross_insn_X_rD_rA_rB: This coverage point provides verification coverage statistics for instructions of format (A) in Table (5.1). The coverage point shows that all tested instructions correctly write to and read from all 32 GPRs (as their register operands). This coverage point is basically a cross coverage between the following coverage points: `dut_insn`, `dest_reg`, `src_1_reg` and `src_2_reg`. There are 17 instructions of format (A) in the OR1200 instruction set while six of them are erroneous. Equation (5.2) shows the maximum achievable verification coverage for this coverage point.

$$\textit{Maximum possible coverage} = \frac{11}{17} * 100 = 64.7\% \tag{5.2}$$

Figure [5.36] shows that 93% of the maximum possible coverage is achieved. Since the test space is quite large, it takes very long simulation time to achieve 100% maximum possible coverage. One possible solution was to write a directed test and generate instructions of format (B) only. We did achieve 100% coverage from this directed test.

cross_insn_X_rA_rB: This coverage point provides verification coverage statistics for instructions of format (B) in Table (5.1). The coverage point shows that all these instructions correctly write to and read from all GPRs (as their register operands). This coverage point is basically a cross coverage between the following coverage points: `dut_insn`, `src_1_reg` and `src_2_reg`. Figure [5.36] shows that 100% coverage is achieved.

cross_insn_X_rD_rA_Immed16: This coverage point provides verification coverage statistics for instructions of format (C) in Table (5.1). The coverage point shows that all tested instructions use a valid 16 bits immediate value and correctly write to and read from all GPRs (as their register operands). This coverage point is basically a cross coverage between the following coverage points: `dut_insn`, `dest_reg`, `src_1_reg` and `Immed_16_bits`. In total there are 13 instructions of format (C) in the OR1200 instruction set while four of them are erroneous. Equation (5.3) shows the maximum achievable verification coverage for this coverage point.

$$\textit{Maximum possible coverage} = \frac{9}{13} * 100 = 69.2\% \tag{5.3}$$

Figure [5.36] shows that the maximum possible coverage is achieved.

cross_insn_X_rA_Immed16: This coverage point provides verification coverage statistics for instructions of format (D) in Table (5.1). This coverage point is a cross coverage between the following coverage points: `dut_insn`, `src_1_reg` and `Immed_16_bits`. Figure [5.36] shows that 100% coverage is achieved.

cross_insn_X_rA_rB_Immed16_dist: This coverage point provides verification coverage statistics for instructions of format (E) in Table (5.1). This coverage point is basically a cross coverage between the following coverage points: `dut_insn`, `dest_reg`, `src_1_reg` and `Immed_16_bits_dist`. There are four instructions of format (E) while one of them is erroneous. Equation (5.4) shows the maximum achievable verification coverage for this coverage point.

$$\textit{Maximum possible coverage} = \frac{3}{4} * 100 = 75\% \tag{5.4}$$

Figure [5.36] shows that the maximum possible coverage is successfully achieved.

cross_insn_X_rD_rA_Immed6: This coverage point provides verification coverage statistics for instructions of format (F) in Table (5.1). It is basically a cross coverage between the following coverage points: `dut_insn`, `dest_reg`, `src_1_reg` and `Immed_6_bits`. There are four instructions of format (F) while one of them is erroneous. Equation (5.5) shows the maximum achievable verification coverage for this coverage point.

$$\textit{Maximum possible coverage} \;=\; \frac{3}{4} * 100 \;=\; 75\% \tag{5.5}$$

Figure [5.36] shows that the maximum possible coverage is achieved.

cross_insn_X_Immed26: This coverage point provides verification coverage statistics for instructions of format (G) in Table (5.1). It is basically a cross coverage between the following coverage points: `dut_insn` and `Immed_26_bits`. Figure [5.36] shows that 100% coverage is achieved.

cross_insn_X_rD_Immed16: This coverage point provides verification coverage statistics for instructions of format (H) in Table (5.1). It is basically a cross coverage between the following coverage points: `dut_insn`, `dest_reg` and `Immed_16_bits`. Figure [5.36] shows that 100% coverage is achieved.

cross_insn_X_rD_rA: This coverage point provides verification coverage statistics for instructions of format (I) in Table (5.1). There are four instructions of format (I) and all of them are erroneous. Figure [5.36] shows that 0% coverage is achieved.

cross_insn_X_rB: This coverage point provides verification coverage statistics for instructions of format (J) in Table (5.1). There are two instructions of format (J) and both are erroneous. Figure [5.36] shows that 0% coverage is achieved.

cross_insn_X_rD: This coverage point provides verification coverage statistics for instructions of format (K) in Table (5.1). There is only one instruction of format (K) and it is erroneous. Figure [5.36] shows that 0% coverage is achieved.

cross_insn_X_rB_Immed16_dist: This coverage point provides the verification coverage statistics for the instructions of format (L) in Table (5.1). There is only one instruction of format (L) and it is erroneous. Figure [5.36] shows that 0% coverage is achieved.

cross_insn_X_carry_carrydelay: This coverage point stores the histogram of all instructions that can drive the carry flag (CY). It verifies that all these instructions correctly set and reset the carry flag. There are ten such instructions while six of them are erroneous. It is basically a cross coverage between the following coverage points: `dut_insn`, `cover_carry` and `cover_carry_delay`. Equation (5.6) shows the maximum achievable verification coverage for this coverage point.

$$\textit{Maximum possible coverage} \;=\; \frac{4}{10} * 100 \;=\; 40\% \tag{5.6}$$

Figure [5.36] shows that the maximum possible coverage is achieved.

cross_insn_X_flag_flagdelay: This coverage point stores the histogram of all instructions that can drive the branch flag (F). It verifies that all these instructions correctly set and reset this flag. It is basically a cross coverage between the following coverage points: `dut_insn`, `cover_flag` and `cover_flag_delay`. Figure [5.36] shows that 100% coverage is achieved.

cross_insn_X_ovflag_ovflagdelay: This coverage point stores the histogram of all instructions that can drive the overflow flag (OV). It verifies that all these instructions correctly set and reset this flag. It is basically a cross coverage between the following coverage points: `dut_insn`, `cover_ov_flag` and `cover_ov_flag_delay`. There are ten such instructions while six of them are erroneous. It means that the total number of combinations is 40. The combinations for one instruction are given below.

1. l.insn, ov_low, ov_low
2. l.insn, ov_low, ov_high
3. l.insn, ov_high, ov_low
4. l.insn, ov_high, ov_high

Since four instructions are working correctly, 16 combinations can be covered at maximum. However, as discussed in Subsection (5.5.9), the overflow flag is not correctly implemented in the OR1200 core. Therefore, only four combinations can be covered (OV flag stays low). Equation (5.7) shows the maximum achievable verification coverage for this coverage point.

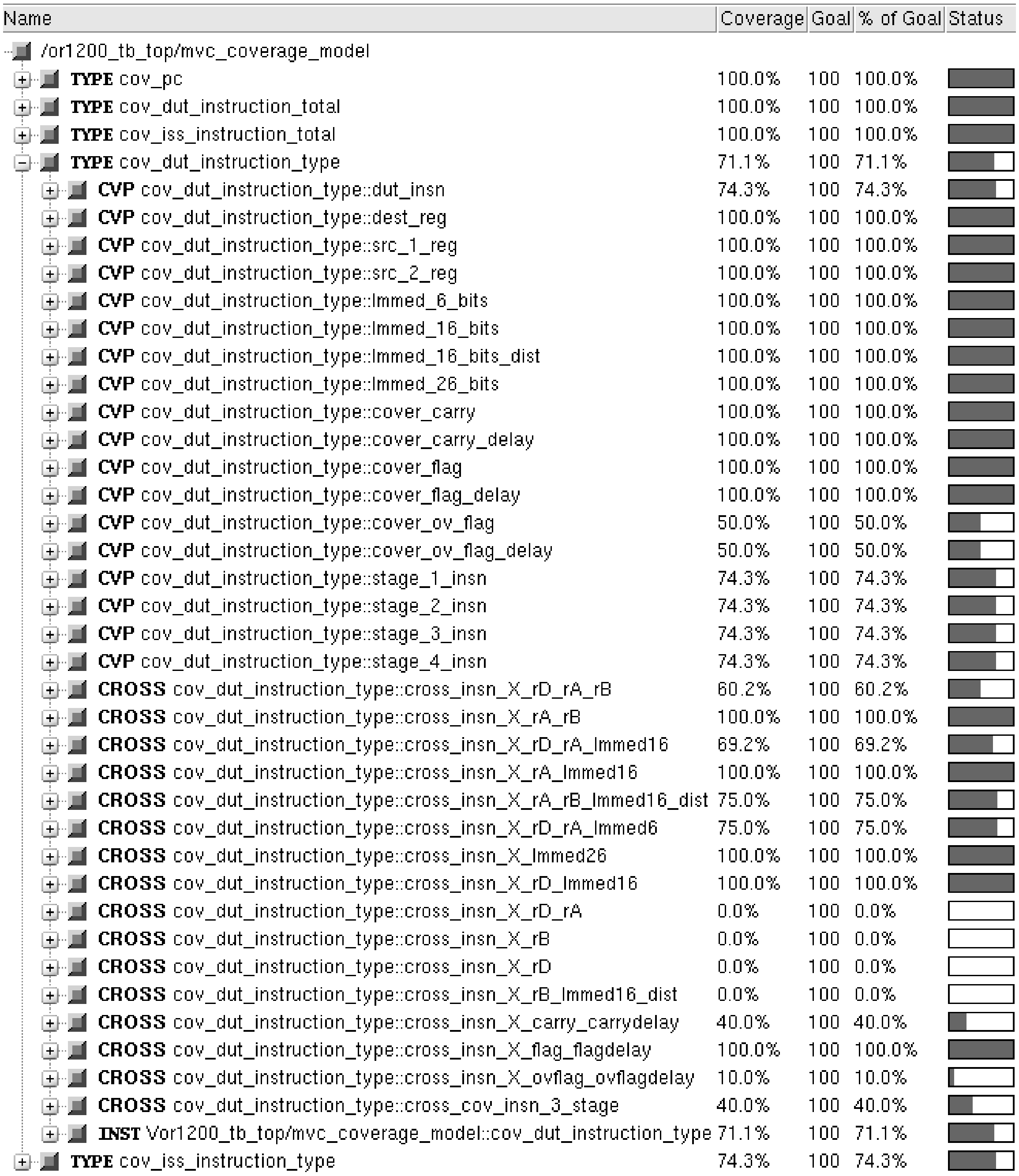
$$Maximum\ possible\ coverage \ = \ \frac{4}{40} * 100 \ = \ 10\% \tag{5.7}$$

Figure [5.36] cross verifies the calculated verification coverage.

cross_cov_insn_3_stage: This coverage stores the histogram of three instructions in the OR1200 pipeline (in contiguous stages) to account the dependencies between instructions. It is basically a cross coverage between the following coverage points: `stage_1_insn`, `stage_2_insn` and `stage_3_insn`. As there are total 78 instructions in total, the overall number of combinations to cover is 474552. Since only 58 instructions are working correctly, the total number of correctly working combinations is 195112. Moreover, we have to restrict the generation of jump/branch instructions in the delay slot. This constraint is relevant only for the verification of the OR1200 core. The reason behind this restriction is to stop the generation of a sequence of instructions with two consecutive jump/branch instructions followed by another instruction which can generate an exception. If the delay slot instruction generates an exception in the ISS, the EPCR is set to PC − 4. However, if the jump/branch instruction of this delay slot itself is executing in the delay slot of another jump/branch instruction, the value of EPCR is incorrect and nondeterministic. This restriction excludes 1856 combinations as there are four working jump/branch instructions having a delay slot. Equation (5.8) shows the maximum achievable verification coverage for this coverage point.

$$Maximum\ possible\ coverage \ = \ \frac{(195112 - 1856)}{474552} * 100 \ = \ 40.7\% \tag{5.8}$$

Figure [5.36] shows that 98.2% of the maximum possible coverage is achieved. Since the test space is quite large, it takes very long simulation time to achieve 100% maximum possible coverage.



Name	Coverage	Goal	% of Goal	Status
/or1200_tb_top/mvc_coverage_model				
TYPE cov_pc	100.0%	100	100.0%	
TYPE cov_dut_instruction_total	100.0%	100	100.0%	
TYPE cov_iss_instruction_total	100.0%	100	100.0%	
TYPE cov_dut_instruction_type	71.1%	100	71.1%	
CVP cov_dut_instruction_type::dut_insn	74.3%	100	74.3%	
CVP cov_dut_instruction_type::dest_reg	100.0%	100	100.0%	
CVP cov_dut_instruction_type::src_1_reg	100.0%	100	100.0%	
CVP cov_dut_instruction_type::src_2_reg	100.0%	100	100.0%	
CVP cov_dut_instruction_type::Immed_6_bits	100.0%	100	100.0%	
CVP cov_dut_instruction_type::Immed_16_bits	100.0%	100	100.0%	
CVP cov_dut_instruction_type::Immed_16_bits_dist	100.0%	100	100.0%	
CVP cov_dut_instruction_type::Immed_26_bits	100.0%	100	100.0%	
CVP cov_dut_instruction_type::cover_carry	100.0%	100	100.0%	
CVP cov_dut_instruction_type::cover_carry_delay	100.0%	100	100.0%	
CVP cov_dut_instruction_type::cover_flag	100.0%	100	100.0%	
CVP cov_dut_instruction_type::cover_flag_delay	100.0%	100	100.0%	
CVP cov_dut_instruction_type::cover_ov_flag	50.0%	100	50.0%	
CVP cov_dut_instruction_type::cover_ov_flag_delay	50.0%	100	50.0%	
CVP cov_dut_instruction_type::stage_1_insn	74.3%	100	74.3%	
CVP cov_dut_instruction_type::stage_2_insn	74.3%	100	74.3%	
CVP cov_dut_instruction_type::stage_3_insn	74.3%	100	74.3%	
CVP cov_dut_instruction_type::stage_4_insn	74.3%	100	74.3%	
CROSS cov_dut_instruction_type::cross_insn_X_rD_rA_rB	60.2%	100	60.2%	
CROSS cov_dut_instruction_type::cross_insn_X_rA_rB	100.0%	100	100.0%	
CROSS cov_dut_instruction_type::cross_insn_X_rD_rA_Immed16	69.2%	100	69.2%	
CROSS cov_dut_instruction_type::cross_insn_X_rA_Immed16	100.0%	100	100.0%	
CROSS cov_dut_instruction_type::cross_insn_X_rA_rB_Immed16_dist	75.0%	100	75.0%	
CROSS cov_dut_instruction_type::cross_insn_X_rD_rA_Immed6	75.0%	100	75.0%	
CROSS cov_dut_instruction_type::cross_insn_X_Immed26	100.0%	100	100.0%	
CROSS cov_dut_instruction_type::cross_insn_X_rD_Immed16	100.0%	100	100.0%	
CROSS cov_dut_instruction_type::cross_insn_X_rD_rA	0.0%	100	0.0%	
CROSS cov_dut_instruction_type::cross_insn_X_rB	0.0%	100	0.0%	
CROSS cov_dut_instruction_type::cross_insn_X_rD	0.0%	100	0.0%	
CROSS cov_dut_instruction_type::cross_insn_X_rB_Immed16_dist	0.0%	100	0.0%	
CROSS cov_dut_instruction_type::cross_insn_X_carry_carrydelay	40.0%	100	40.0%	
CROSS cov_dut_instruction_type::cross_insn_X_flag_flagdelay	100.0%	100	100.0%	
CROSS cov_dut_instruction_type::cross_insn_X_ovflag_ovflagdelay	10.0%	100	10.0%	
CROSS cov_dut_instruction_type::cross_cov_insn_3_stage	40.0%	100	40.0%	
INST Vor1200_tb_top/mvc_coverage_model::cov_dut_instruction_type	71.1%	100	71.1%	
TYPE cov_iss_instruction_type	74.3%	100	74.3%	

Figure 5.36 Functional verification coverage of the OR1200 core.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This thesis is divided into two major parts: the first part is the implementation of a CPU Subsystem and the second part is the functional verification of this Subsystem.

The CPU Subsystem is used in an advance control architecture for multimode transceivers. It operates as a central control unit of the architecture. Its foremost function is to configure the transceiver and its interface for a particular communication standard. The Subsystem is comprised of an open source OR1200 core, a triple-layer Sub-bus system, a memory subsystem and several interfaces. All components comply with the Wishbone interconnection standard. The OR1200 GNU toolchain is used to generate the memory initialization file for the OR1200 core and also for an early code analysis. The simulation-based verification of the CPU Subsystem includes the coverage-driven constrained random verification of the OR1200 core, the Sub-bus system and the memory subsystem. For the verification of the OR1200 core, a golden model is implemented using the OR1200 ISS with a SystemC wrapper around to incorporate the verification environment. Moreover, OVM is used to implement a configurable and reusable verification environment. This thesis includes the co-simulation of the programming languages VHDL, Verilog, C, C++ (SystemC), DPI and SystemVerilog.

The verification results of the Sub-bus system and the memory system show that both subsystems are implemented correctly. Furthermore, the simulation of the CPU Subsystem demonstrates that it provides the maximum possible throughput for most of the OR1200 instructions i.e., three clock cycles for single cycle instructions.

The verification results of the OR1200 core describe that the core has some malfunctions including (i) erroneous instructions, (ii) unimplemented instructions, (iii) design errors and (iv) discrepancies between the specification and its implementation. Moreover, the OR1200 ISS (golden model) also has some implementation errors and unimplemented instructions. This significantly restricts the achievable verification coverage of the OR1200 core. There are 78 instructions in the OR1200 instruction set but only 58 instructions could be included in the verification test because 20 instructions are erroneous or unimplemented (either in the OR1200 core or in its ISS). Hence, the single instruction verification coverage is restricted to 74.3%. The cross coverage of three contiguous instructions to observe the dependencies between the instructions is restricted to 40.7%.

6.2 Future Work

There are several improvements which could be done for a more qualitative verification of the CPU Subsystem. Some possible improvements are listed below.

1. A possible improvement could be to perform the functional verification of the OR1200 core after the rectification of all found malfunctions in the core and in its ISS.

2. Another possible improvement could be to perform the formal verification of the OR1200 core, the Sub-bus system and the memory system.

3. It could be possible to perform the verification of the peripherals of the OR1200 processor such as the debug unit, the power management unit and the interrupt controller.

Furthermore, it could be possible to enhance the throughput of the CPU Subsystem by maximizing the throughput of Wishbone interfaces of the OR1200 core.

Appendix A

Appendices

A.1 Software development

A.1.1 Test application program

```
/*************Test application program (main.c)*************/

#include "Test.h"
#define SIM
#ifdef SIM
void exit_sim (void){
  __asm__ __volatile__ ("l.add r3,r0,%0\n\t"
                         "l.nop %1": : "r" (1), "K" (0x0001));
}
#endif
int mem[] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
static int b = 6;
int main() {
    int i = 12;
    if (i % 5 == 0){
    mem[b] = mem[b] + 5;
    }
    i += 2;
    if (i % 2 == 0){
        i = Mult(10);
    }
#ifdef SIM
  exit_sim();
#endif
    return 0;
}
```

```
/*--------------------Test.h--------------------*/
#ifndef TEST
#define TEST
    int Mult(int a);
#endif

/*--------------------Test.c--------------------*/
#include "Test.h"
```

```
int Mult(int a){
    if(a <= 1) return 1;
    a = a * Mult(a - 1);
    return a;
}
```

A.1.2 Disassembly file of the test program

```
/*************** Test.lst ****************/

output:     file format elf32-or32

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .bss          00000050  f0000000  f0000000  00002000  2**0
                  ALLOC
  1 .text         000002dc  00000100  00000100  00000100  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  2 .data         00000040  000003dc  000003dc  000003dc  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  3 .rdata        00000010  0000041c  0000041c  0000041c  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  4 .stab         0000030c  00000000  00000000  0000042c  2**2
                  CONTENTS, RELOC, READONLY, DEBUGGING
  5 .stabstr      000004f5  00000000  00000000  00000738  2**0
                  CONTENTS, READONLY, DEBUGGING
  6 .comment      00000024  00000000  00000000  00000c2d  2**0
                  CONTENTS, READONLY

Disassembly of section .text:

00000100 <_stext>:
 100:   18 20 f0 00     l.movhi r1,0xf000
 104:   a8 21 04 50     l.ori r1,r1,0x450

00000108 <_mem_data_copy>:
 108:   18 60 f0 00     l.movhi r3,0xf000
 10c:   a8 63 00 00     l.ori r3,r3,0x0
 110:   18 80 00 00     l.movhi r4,0x0
 114:   a8 84 03 dc     l.ori r4,r4,0x3dc
 118:   18 a0 00 00     l.movhi r5,0x0
 11c:   a8 a5 04 1c     l.ori r5,r5,0x41c
 120:   e0 a5 20 02     l.sub r5,r5,r4
 124:   bc 05 00 00     l.sfeqi r5,0x0
 128:   10 00 00 0a     l.bf 150 <_jump_main>
 12c:   15 00 00 00     l.nop 0x0

00000130 <_mem_data_loop>:
 130:   84 c4 00 00     l.lwz r6,0x0(r4)
 134:   d4 03 30 00     l.sw 0x0(r3),r6
 138:   9c 63 00 04     l.addi r3,r3,0x4
 13c:   9c 84 00 04     l.addi r4,r4,0x4
 140:   9c a5 ff fc     l.addi r5,r5,0xfffffffc
 144:   bd 45 00 00     l.sfgtsi r5,0x0
 148:   13 ff ff fa     l.bf 130 <__stack+0xffffce0>
 14c:   15 00 00 00     l.nop 0x0

00000150 <_jump_main>:
 150:   18 40 00 00     l.movhi r2,0x0
 154:   a8 42 02 30     l.ori r2,r2,0x230
 158:   44 00 10 00     l.jr r2
 15c:   15 00 00 00     l.nop 0x0
```

```
00000160 <_Mult>:
#include "Test.h"

int Mult(int a){
 160:    9c 21 ff d4      l.addi r1,r1,0xffffffd4
 164:    d4 01 10 04      l.sw 0x4(r1),r2
 168:    9c 41 00 2c      l.addi r2,r1,0x2c
 16c:    d4 01 48 00      l.sw 0x0(r1),r9
 170:    d7 e2 1f fc      l.sw 0xfffffffc(r2),r3

   if(a <= 1) return 1;
 174:    84 62 ff fc      l.lwz r3,0xfffffffc(r2)
 178:    d7 e2 1f e8      l.sw 0xffffffe8(r2),r3
 17c:    84 82 ff e8      l.lwz r4,0xffffffe8(r2)
 180:    bd 44 00 01      l.sfgtsi r4,0x1
 184:    10 00 00 06      l.bf 19c <_Mult+0x3c>
 188:    15 00 00 00      l.nop 0x0
 18c:    9c 60 00 01      l.addi r3,r0,0x1
 190:    d7 e2 1f f0      l.sw 0xfffffff0(r2),r3
 194:    00 00 00 15      l.j 1e8 <_Mult+0x88>
 198:    15 00 00 00      l.nop 0x0

   a = a * Mult(a - 1);
 19c:    84 82 ff fc    l.lwz r4,0xfffffffc(r2)
 1a0:    d7 e2 27 e4     l.sw 0xffffffe4(r2),r4
 1a4:    84 62 ff e4     l.lwz r3,0xffffffe4(r2)
 1a8:    9c 63 ff ff     l.addi r3,r3,0xffffffff
 1ac:    d7 e2 1f f4     l.sw 0xfffffff4(r2),r3
 1b0:    84 62 ff f4     l.lwz r3,0xfffffff4(r2)
 1b4:    07 ff ff eb     l.jal 160 <__stack+0xffffd10>
 1b8:    15 00 00 00     l.nop 0x0
 1bc:    d7 e2 5f f8     l.sw 0xfffffff8(r2),r11
 1c0:    84 82 ff fc     l.lwz r4,0xfffffffc(r2)
 1c4:    d7 e2 27 e0     l.sw 0xffffffe0(r2),r4
 1c8:    84 62 ff e0     l.lwz r3,0xffffffe0(r2)
 1cc:    84 82 ff f8     l.lwz r4,0xfffffff8(r2)
 1d0:    e0 63 23 06     l.mul r3,r3,r4
 1d4:    d7 e2 1f dc     l.sw 0xffffffdc(r2),r3
 1d8:    84 62 ff dc     l.lwz r3,0xffffffdc(r2)
 1dc:    d7 e2 1f fc     l.sw 0xfffffffc(r2),r3
    return a;
 1e0:    84 82 ff fc     l.lwz r4,0xfffffffc(r2)
 1e4:    d7 e2 27 f0     l.sw 0xfffffff0(r2),r4
 1e8:    84 62 ff f0     l.lwz r3,0xfffffff0(r2)
 1ec:    d7 e2 1f ec     l.sw 0xffffffec(r2),r3

}
 1f0:    85 62 ff ec     l.lwz r11,0xffffffec(r2)
 1f4:    85 21 00 00     l.lwz r9,0x0(r1)
 1f8:    84 41 00 04     l.lwz r2,0x4(r1)
 1fc:    44 00 48 00     l.jr r9
 200:    9c 21 00 2c     l.addi r1,r1,0x2c

00000204 <_exit_sim>:
#include "Test.h"
#define SIM
#ifdef SIM
void exit_sim (void){
 204:    9c 21 ff f8     l.addi r1,r1,0xfffffff8
 208:    d4 01 10 00     l.sw 0x0(r1),r2
 20c:    9c 41 00 08     l.addi r2,r1,0x8
   __asm__ __volatile__ ("l.add r3,r0,%0\n\t"
 210:    9c 60 00 01     l.addi r3,r0,0x1
 214:    d7 e2 1f fc     l.sw 0xfffffffc(r2),r3
 218:   84 62 ff fc    l.lwz r3,0xfffffffc(r2)
 21c:   e0 60 18 00    l.add r3,r0,r3
 220:   15 00 00 01    l.nop 0x1
 224:   84 41 00 00    l.lwz r2,0x0(r1)
 228:   44 00 48 00    l.jr r9
 22c:   9c 21 00 08    l.addi r1,r1,0x8

00000230 <_main>:
}
#endif
int mem[] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
static int b = 6;

int main() {
 230:   9c 21 ff 8c    l.addi r1,r1,0xffffff8c
 234:   d4 01 10 04    l.sw 0x4(r1),r2
 238:   9c 41 00 74    l.addi r2,r1,0x74
 23c:   d4 01 48 00    l.sw 0x0(r1),r9
    int i = 12;
 240:   9c 60 00 0c    l.addi r3,r0,0xc
 244:   d7 e2 1f d0    l.sw 0xffffffd0(r2),r3
 248:   84 82 ff d0    l.lwz r4,0xffffffd0(r2)
 24c:   d7 e2 27 fc    l.sw 0xfffffffc(r2),r4
    if (i % 5 == 0){
 250:   84 62 ff fc    l.lwz r3,0xfffffffc(r2)
 254:   d7 e2 1f cc    l.sw 0xffffffcc(r2),r3
 258:   9c 80 00 05    l.addi r4,r0,0x5
 25c:   d7 e2 27 c4    l.sw 0xffffffc4(r2),r4
 260:   84 62 ff cc    l.lwz r3,0xffffffcc(r2)
 264:   84 82 ff c4    l.lwz r4,0xffffffc4(r2)
 268:   e0 63 23 09    l.div r3,r3,r4
 26c:   d7 e2 1f c8    l.sw 0xffffffc8(r2),r3
 270:   84 62 ff c8    l.lwz r3,0xffffffc8(r2)
 274:   d7 e2 1f c0    l.sw 0xffffffc0(r2),r3
 278:   84 82 ff c0    l.lwz r4,0xffffffc0(r2)
 27c:   b8 84 00 02    l.slli r4,r4,0x2
 280:   d7 e2 27 bc    l.sw 0xffffffbc(r2),r4
 284:   84 62 ff bc    l.lwz r3,0xffffffbc(r2)
 288:   84 82 ff c8    l.lwz r4,0xffffffc8(r2)
 28c:   e0 63 20 00    l.add r3,r3,r4
 290:   d7 e2 1f bc    l.sw 0xffffffbc(r2),r3
 294:   84 62 ff cc    l.lwz r3,0xffffffcc(r2)
 298:   84 82 ff bc    l.lwz r4,0xffffffbc(r2)
 29c:   e0 63 20 02    l.sub r3,r3,r4
 2a0:   d7 e2 1f d8    l.sw 0xffffffd8(r2),r3
 2a4:   84 62 ff d8    l.lwz r3,0xffffffd8(r2)
 2a8:   bc 23 00 00    l.sfnei r3,0x0
 2ac:   10 00 00 2b    l.bf 358 <_main+0x128>
 2b0:   15 00 00 00    l.nop 0x0
    mem[b] = mem[b] + 5;
 2b4:   18 80 00 00    l.movhi r4,0x0
 2b8:   a8 84 04 18    l.ori r4,r4,0x418
 2bc:   d7 e2 27 b8    l.sw 0xffffffb8(r2),r4
 2c0:   84 62 ff b8    l.lwz r3,0xffffffb8(r2)
 2c4:   84 63 00 00    l.lwz r3,0x0(r3)
 2c8:   d7 e2 1f dc    l.sw 0xffffffdc(r2),r3
 2cc:   18 80 00 00    l.movhi r4,0x0
 2d0:   a8 84 04 18    l.ori r4,r4,0x418
 2d4:   d7 e2 27 b4    l.sw 0xffffffb4(r2),r4
 2d8:   84 62 ff b4    l.lwz r3,0xffffffb4(r2)
 2dc:   84 63 00 00    l.lwz r3,0x0(r3)
 2e0:   d7 e2 1f e0    l.sw 0xffffffe0(r2),r3
 2e4:   18 80 00 00    l.movhi r4,0x0
 2e8:   a8 84 03 dc    l.ori r4,r4,0x3dc
 2ec:   d7 e2 27 b0    l.sw 0xffffffb0(r2),r4
 2f0:   84 62 ff e0    l.lwz r3,0xffffffe0(r2)
 2f4:   b8 63 00 02    l.slli r3,r3,0x2
 2f8:   d7 e2 1f ac    l.sw 0xffffffac(r2),r3
 2fc:   84 82 ff ac    l.lwz r4,0xffffffac(r2)
 300:   84 62 ff b0    l.lwz r3,0xffffffb0(r2)
 304:   e0 84 18 00    l.add r4,r4,r3
 308:   d7 e2 27 a8    l.sw 0xffffffa8(r2),r4
 30c:   84 82 ff a8    l.lwz r4,0xffffffa8(r2)
 310:   84 84 00 00    l.lwz r4,0x0(r4)
 314:   d7 e2 27 e4    l.sw 0xffffffe4(r2),r4
 318:   84 62 ff e4    l.lwz r3,0xffffffe4(r2)
 31c:   9c 63 00 05    l.addi r3,r3,0x5
 320:   d7 e2 1f e8    l.sw 0xffffffe8(r2),r3
 324:   18 80 00 00    l.movhi r4,0x0
 328:   a8 84 03 dc    l.ori r4,r4,0x3dc
 32c:   d7 e2 27 a4    l.sw 0xffffffa4(r2),r4
 330:   84 62 ff dc    l.lwz r3,0xffffffdc(r2)
 334:   b8 63 00 02    l.slli r3,r3,0x2
 338:   d7 e2 1f a0    l.sw 0xffffffa0(r2),r3
 33c:   84 82 ff a0    l.lwz r4,0xffffffa0(r2)
 340:   84 62 ff a4    l.lwz r3,0xffffffa4(r2)
 344:   e0 84 18 00    l.add r4,r4,r3
 348:   d7 e2 27 9c    l.sw 0xffffff9c(r2),r4
 34c:   84 62 ff e8    l.lwz r3,0xffffffe8(r2)
 350:   84 82 ff 9c    l.lwz r4,0xffffff9c(r2)
 354:   d4 04 18 00    l.sw 0x0(r4),r3
    }
    i += 2;
 358:   84 82 ff fc    l.lwz r4,0xfffffffc(r2)
 35c:   d7 e2 27 98    l.sw 0xffffff98(r2),r4
 360:   84 62 ff 98    l.lwz r3,0xffffff98(r2)
 364:   9c 63 00 02    l.addi r3,r3,0x2
 368:   d7 e2 1f 94    l.sw 0xffffff94(r2),r3
 36c:   84 82 ff 94    l.lwz r4,0xffffff94(r2)
 370:   d7 e2 27 fc    l.sw 0xfffffffc(r2),r4
    if (i % 2 == 0){
 374:   84 62 ff fc    l.lwz r3,0xfffffffc(r2)
 378:   d7 e2 1f ec    l.sw 0xffffffec(r2),r3
 37c:   84 82 ff ec    l.lwz r4,0xffffffec(r2)
 380:   a4 84 00 01    l.andi r4,r4,0x1
 384:   d7 e2 27 f0    l.sw 0xfffffff0(r2),r4
 388:   84 62 ff f0    l.lwz r3,0xfffffff0(r2)
 38c:   bc 23 00 00    l.sfnei r3,0x0
 390:   10 00 00 08    l.bf 3b0 <_main+0x180>
 394:   15 00 00 00    l.nop 0x0
        i = Mult(10);
 398:   9c 60 00 0a    l.addi r3,r0,0xa
 39c:   07 ff ff 71    l.jal 160 <__stack+0xffffd10>
 3a0:   15 00 00 00    l.nop 0x0
 3a4:   d7 e2 5f f4    l.sw 0xfffffff4(r2),r11
 3a8:   84 82 ff f4    l.lwz r4,0xfffffff4(r2)
 3ac:   d7 e2 27 fc    l.sw 0xfffffffc(r2),r4
    }
#ifdef SIM
  exit_sim();
 3b0:   07 ff ff 95    l.jal 204 <__stack+0xffffdb4>
 3b4:   15 00 00 00    l.nop 0x0
#endif
    return 0;
 3b8:   9c 60 00 00    l.addi r3,r0,0x0
 3bc:   d7 e2 1f f8    l.sw 0xfffffff8(r2),r3
 3c0:   84 82 ff f8    l.lwz r4,0xfffffff8(r2)
 3c4:   d7 e2 27 d4    l.sw 0xffffffd4(r2),r4
}
 3c8:   85 62 ff d4    l.lwz r11,0xffffffd4(r2)
 3cc:    85 21 00 00     l.lwz r9,0x0(r1)
 3d0:    84 41 00 04     l.lwz r2,0x4(r1)
 3d4:    44 00 48 00     l.jr r9
 3d8:    9c 21 00 74     l.addi r1,r1,0x74
```

A.1.3 Linker Script

```
/***************Linker Script to set-up the Memory-map***************/
/*---------startup.ld---------*/
/*
 * Setup the memory map of the Code.
 * stack grows down from high memory.
 * --------------------------------------------------------------------
 * The .text    section - contains instructions
 * The .data    section - contains static initialized data
 * The .rdata   section - contains static constant data
 * The .bss     section - contains uninitialized data
 * The .ctor    section - contains addresses of global constructors
 * The .dtor    section - contains addresses of global destructors
 * The .stabs   section - part of the debug symbol table
 * The .stabstr section - part of the debug symbol table
 * --------------------------------------------------------------------
 * The memory map look like this:
 * +--------------------+ <- Start of ROM
 * |Interrupt Table     |
 * +--------------------+ <- 0x100
 * | .text              |
 * |       _stext       |
 * |       *.text       |
 * |       _etext       |
 * +--------------------+ <- initialized data goes here
 * | .data              |
 * |       _sdata       |
 * |       *.data       |
 * |       _sdata       |
 * +--------------------+ <- the ctor and dtor lists are for
 * | .rdata             |    C++ support (if requied)
 * |       *.rdata      |
 * |                    |
 * +--------------------+ <- Start of RAM
 * |                    |    start of bss, cleared by crt0
 * | .bss               |    start of heap
 * |       __bss_start  |
 * |       _end         |
 * +--------------------+
 * .                    .
 * .                    .
 * .                    .
 * |       __stack      |
 * +--------------------+ <- top of stack
 */

STACKSIZE = 0x100;
OFFSET = 0x0;

/*The next line in the script gives a value to the linker symbol __stack.*/

PROVIDE (__stack = ADDR(.bss) + SIZEOF(.bss) + STACKSIZE + OFFSET);
PROVIDE (__copy_start = _copy_start);
PROVIDE (__copy_end = _copy_end);
PROVIDE (__copy_adr = _copy_adr);
```

```
MEMORY
{
    rom (rx)  : ORIGIN = 0x00000000, LENGTH = 0x000f0000
    ram  (rwx): ORIGIN = 0xf0000000, LENGTH = 0x000f0000
}

SECTIONS
{
    .text 0x100 :
    {
        _stext = .;
        *(.text)
        _etext = .;
    } > rom
/*
    All initialized data sections go in the RAM. Don’t forget to
    write additional code to intialize these sections. Better
    still, include explicit initialization code in your
    application program.
*/
    .data : {
        _copy_start = .;
        _sdata = .;
        *(.data)
        _edata = .;
    } > rom

    .rdata :
    {
        *(.rdata)
    _copy_end = .;
/*
        Include any C++ global constructors. If you’re not using
        C++ then you can delete this code.
*/
        __CTOR_LIST__ = .;
        LONG((__CTOR_END__ - __CTOR_LIST__) / 4 - 2)
        *(.ctors)
        LONG(0)
        __CTOR_END__ = .;
/*
        Include any C++ global destructors. You can delete
        this too.
*/
        __DTOR_LIST__ = .;
        LONG((__DTOR_END__ - __DTOR_LIST__) / 4 - 2)
        *(.dtors)
        LONG(0)
        __DTOR_END__ = .;
    } > rom
/*
    The .bss sections, which are uninitialized in the source and
    set to zero at run time, are located in RAM after the .data
    sections. Beyond the end of the .bss sections are the
    heap and stack.
*/
    .bss (NOLOAD):
    {
        _copy_adr = .;
        . = ( SIZEOF (.data) + SIZEOF (.rdata) );
        __bss_start = .;
        *(.bss)
        *(COMMON)
        end = ALIGN(0x2);
        _end = ALIGN(0x2);
    } > ram
/*
    Debug symbol tables are not loaded, but do need to get
    linked with the rest of the program.
*/
    .stab 0 (NOLOAD) :
    {
        [ .stab ]
    }

    .stabstr 0 (NOLOAD) :
    {
        [ .stabstr ]
    }
}
```

A.1.4 Startup Script

```
/*--------Startup Script to include explicit initialization code--------*/
/*--------startup.S--------*/

.extern __stack
.extern __copy_start
.extern __copy_end
.extern __copy_adr


/* Core jumps here at start and reset */
_stext:
  /* Stack initialization */
  l.movhi r1,hi(__stack)
  l.ori   r1,r1,lo(__stack)

_mem_data_copy:
  /* Copy the initialated data and static variables from Rom to Ram */
  l.movhi r3,hi(__copy_adr)
  l.ori   r3,r3,lo(__copy_adr)
  l.movhi r4,hi(__copy_start)
  l.ori   r4,r4,lo(__copy_start)
  l.movhi r5,hi(__copy_end)
  l.ori   r5,r5,lo(__copy_end)
  l.sub   r5,r5,r4
  l.sfeqi r5,0
  l.bf    _jump_main
  l.nop

_mem_data_loop:
    l.lwz   r6,0(r4)
  l.sw    0(r3),r6
  l.addi  r3,r3,4
  l.addi  r4,r4,4
  l.addi  r5,r5,-4
  l.sfgtsi r5,0
  l.bf     _mem_data_loop
  l.nop

/* Jump to Main */
_jump_main:
  l.movhi r2,hi(_main)
  l.ori   r2,r2,lo(_main)
  l.jr    r2
  l.nop
```

A.1.5 A Sample Makefile

```
/******Makefile to compile the application programs with the OR1200 GNU
Toolchain******/
######### Names #############
EXECUTABLE = output
MAIN_FILE = main.c
LINKER_SCRIPT_NAME = startup.ld
SIM_CONFIG_FILE = cpu.cfg

S_FILES = $(wildcard *.S)
C_FILES = $(filter-out $(MAIN_FILE),$(wildcard *.c))
O_FILES = $(S_FILES:%.S=%.o) $(C_FILES:%.c=%.o)

DELETE = $(EXECUTABLE) $(EXECUTABLE).lst $(EXECUTABLE).ihx $(wildcard *.o)

####### GCC, LD, OBJDUMP #######
#export PATH="$PATH:/opt/or32-elf/bin"

CC = or32-elf-gcc
LD = or32-elf-ld
OD = or32-elf-objdump
OC = or32-elf-objcopy
SIZE = or32-elf-size
SIM = or32-elf-sim

######### FLAGS #############
DEBUG_ON_OFF     = -gstabs3
OPTIMIZE_LEVEL  =
#-O1 -O2 -O3 -Os -finline-functions
FLAGS           = -nostartfiles -mhard-div -mhard-mul -I. -W -Wall
CFLAGS          =  $(DEBUG_ON_OFF) $(OPTIMIZE_LEVEL) $(FLAGS)

.PHONY : all clean

all : $(EXECUTABLE)

$(EXECUTABLE) : $(O_FILES)
#### Create executable ####
  $(CC) $(CFLAGS) -T $(LINKER_SCRIPT_NAME) $^ -o $@ $(MAIN_FILE)
  $(OD) -S -h $@ > $@.lst
  $(OC) -O ihex $@ $@.ihx
  $(SIZE) $@ -A --radix=16

.c.o:
#### Create Object-files of c ####
  $(CC) $(CFLAGS) -c $^

.S.o:
#### Create Object-files of assembler ####
  $(CC) $(CFLAGS) -c $^

sim :
  $(SIM) -f $(SIM_CONFIG_FILE) $(EXECUTABLE)

######## Remove Files #######
clean :
  rm -rf $(DELETE)
```

A.2 Functional Verification of the OR1200 core

A.2.1 Empty ELF file

```
BITS 32
                org     0x08048000
  ehdr:                                                 ; Elf32_Ehdr
                db      0x7F, "ELF", 1, 1, 1, 0         ;   e_ident
        times 8 db      0
                dw      2                               ;   e_type
                dw      3                               ;   e_machine
                dd      1                               ;   e_version
                dd      _start                          ;   e_entry
                dd      phdr - $$                       ;   e_phoff
                dd      0                               ;   e_shoff
                dd      0                               ;   e_flags
                dw      ehdrsize                        ;   e_ehsize
                dw      phdrsize                        ;   e_phentsize
                dw      1                               ;   e_phnum
                dw      0                               ;   e_shentsize
                dw      0                               ;   e_shnum
                dw      0                               ;   e_shstrndx
    ehdrsize      equ     $ - ehdr
  phdr:                                                 ; Elf32_Phdr
                dd      1                               ;   p_type
                dd      0                               ;   p_offset
                dd      $$                              ;   p_vaddr
                dd      $$                              ;   p_paddr
                dd      filesize                        ;   p_filesz
                dd      filesize                        ;   p_memsz
                dd      5                               ;   p_flags
                dd      0x1000                          ;   p_align
    phdrsize    equ     $ - phdr
    _start:
    filesize    equ     $ - $$
```

A.2.2 Configuration File for the Or1ksim Library

```
 /* ---------------------------------------------------------------------------
 *
 * This file is part of OpenRISC 1000 Architectural Simulator.  It contains
 * the configuration suitable for running the simple SoC.
 *
 * For explanation of the different fields, see the default simulation
 * configuration file supplied with the Or1ksim (sim.cfg).
 *
 * The "generic" section is an extension to the Or1ksim to support modeling of
 * external peripherals.
 *
 * $Id$
 *
 */

section generic
  enabled         =          1
  baseaddr        = 0x00000000
  size            = 0x7FFFFFFF
  byte_enabled    =          1
  hw_enabled      =          1
  word_enabled     =          1
  name             = "Gen_dev1"
end

section generic
  enabled          =          1
  baseaddr         = 0x80000000
  size             = 0x80000000
  byte_enabled     =          1
  hw_enabled       =          1
  word_enabled     =          1
  name             = "Gen_dev2"
end

/*To cover addr    = 0xFFFFFFFF   */
/*
section generic
  enabled          =          1
  baseaddr         = 0xFFFFFFFF
  size             = 0x00000001
  byte_enabled     =          1
  hw_enabled       =          1
  word_enabled     =          1
  name             = "Gen_dev3"
end
*/
section sim
  verbose          =          0
  debug            =          0
  profile          =          0
  history          =          0
  clkcycle         =         10ns
end

section cpu
  ver              =     0x1200
  rev              =     0x0001
  superscalar      =          0
  hazards          =          0
  dependstats      =          0
  sbuf_len         =          0
end

  /* Disabled Sections. The first two need all their additional fields due
     to a bug in Or1ksim */

section ic
  enabled          =   0
  nsets            = 512
  nways            =   1
  blocksize        =  16
  hitdelay         =  20
  missdelay        =  20
end

section dc
  enabled          =   0
  nsets            = 512
  nways            =   1
  blocksize        =  16
  load_hitdelay    =  20
  load_missdelay   =  20
  store_hitdelay   =  20
  store_missdelay  =  20
end
```

```
section immu
  enabled = 0
end

section dmmu
  enabled = 0
end

section VAPI
  enabled = 0
end

section dma
  enabled = 0
end

section pm
  enabled = 0
end

section bpb
  enabled = 0
end

section debug
  enabled = 0
end

section uart
  enabled = 0
end

section ethernet
  enabled = 0
end

section gpio
  enabled = 0
end

section ata
  enabled = 0
end

section vga
  enabled = 0
end

section fb
  enabled = 0
end

section kbd
  enabled = 0
end

section mc
  enabled = 0
end
```

A.2.3 Modifications in the ISS

Following list of modifications we made in the ISS (Or1ksim) to use it as a golden model for the simulation-based verification of the OR1200 core.

```
/****simple.cfg****/
  /*a.  Generic peripheral mapping for complete 32-bit address space (Appendix (A.2.2)).*/

/****Or1ksim.h****/
  /*a.  Addition of a function pointer for third upcall i.e., upcpustatus, in the
      or1ksim_init() declaration.*/
  int or1ksim_init( const char *config_file,
                    const char *image_file,
                    void *class_ptr,
                    unsigned long int (*upr)( void *class_ptr, unsigned long int addr,
                                                          unsigned long int mask),
                    void (*upw)( void *class_ptr, unsigned long int addr, unsigned long
                                                int mask, unsigned long int wdata),
                    void (*upcpustatus)( void *class_ptr, void *cpu_statusPtr));

/****libtoplevel.c****/
  /*a.  In the defination of or1ksim_init().
      i.  Add function pointer argument for third upcall i.e.,  upcpustatus().
          ...void (*upcpustatus)( void *class_ptr, void *cpu_statusPtr));
      ii. Assign upcall pointer to a function pointer which can be accessed within the ISS
          .*/
          config.ext.write_up_cpustatus = upcpustatus

/****sim-config.h****/
  /*a.  Add a function pointer write_up_cpustatus() in data structure for configuration
      data*/
  struct config {
            struct{ /* External linkage for SystemC */
                void *class_ptr;
                unsigned long int (*read_up) (void *class_ptr,
                          unsigned long int addr, unsigned long int mask);
                void (*write_up) (void *class_ptr, unsigned long int addr,
                          unsigned long int mask, unsigned long int wdata);
                void (*write_up_cpustatus) (void *class_ptr, void *cpu_statusPtr);
            } ext;

  /*b.  Addition of a data structure to hold a void function pointer for writing cpu state
       when upcall*/
  struct ext_access_cpu_status {
          void  (*write_cpustatus_up) (void *); };
  extern struct ext_access_cpu_status cpustatus_up;

/****sim-config.c****/
  a.  struct ext_access_cpu_status cpustatus_up;
  b.  config.ext.write_up_cpustatus = NULL;

/****generic.c****/
  /*a.  Generic write status upcall routine.*/
  static void ext_write_cpustatus (void *cpu_statusPtr){
                config.ext.write_up_cpustatus (config.ext.class_ptr, cpu_statusPtr);
  } /* ext_callback() */

  /*b.  In generic_sec_start ().*/
  cpustatus_up.write_cpustatus_up = ext_write_cpustatus;

/****execute.c****/
  /*a.  Call write_cpustatus_up() upcall after every instruction execution to write the
      ISS state up.*/
  cpustatus_up.write_cpustatus_up(&cpu_state);
```

A.3 ISS implementation of instructions `l.jalr` and `l.jr`

```
/********execgen.c*******/

L.JR:
  case 0x11:
    /* Not unique: real mask fffffffffc000000 and current mask fc000000 differ - do final
        check */
    if((insn & 0xfc000000) == 0x44000000) {
      /* Instruction: l.jr */
      {
        uorreg_t a;
        /* Number of operands: 1 */
        a = (insn >> 11) & 0x1f;
        #define SET_PARAM0(val) cpu_state.reg[a] = val
        #define PARAM0 cpu_state.reg[a]
        {           /* "l_jr" */
          cpu_state.pc_delay = PARAM0;
          next_delay_insn = 1;
          if (config.sim.profile)
            fprintf (runtime.sim.fprof, "-%08llX %"PRIxADDR"\n", runtime.sim.cycles,
                     cpu_state.pc_delay);
        }
        #undef SET_PARAM
        #undef PARAM0

        if (do_stats) {
          current->insn_index = 104;    /* "l.jr" */
          analysis(current);
        }
      }
    } else {
      /* Invalid insn */
      {
        l_invalid ();

        if (do_stats) {
          current->insn_index = -1;    /* "???" */
          analysis(current);
        }
      }
    }
    break;

L.JALR:

  case 0x12:
    /* Not unique: real mask fffffffffc000000 and current mask fc000000 differ - do final
        check */
    if((insn & 0xfc000000) == 0x48000000) {
      /* Instruction: l.jalr */
      {
        uorreg_t a;
        /* Number of operands: 1 */
        a = (insn >> 11) & 0x1f;
        #define SET_PARAM0(val) cpu_state.reg[a] = val
        #define PARAM0 cpu_state.reg[a]
        {           /* "l_jalr" */
          cpu_state.pc_delay = PARAM0;
          setsim_reg(LINK_REGNO, cpu_state.pc + 8);
          next_delay_insn = 1;
        }
        #undef SET_PARAM
        #undef PARAM0

        if (do_stats) {
          current->insn_index = 105;   /* "l.jalr" */
          analysis(current);
        }
      }
    } else {
      /* Invalid insn */
      {
        l_invalid ();

        if (do_stats) {
          current->insn_index = -1;   /* "???" */
          analysis(current);
        }
      }
    }
    break;
```

A.4 ISS implementation of instruction `l.mtspr`

```
/**********execgen.c**********/

 case 0x30:
    /* Not unique: real mask fffffffffc000000 and current mask fc000000 differ - do final
        check */
    if((insn & 0xfc000000) == 0xc0000000) {
      /* Instruction: l.mtspr */
      {
        uorreg_t a, b, c;
        /* Number of operands: 3 */
        a = (insn >> 16) & 0x1f;
        #define SET_PARAM0(val) cpu_state.reg[a] = val
        #define PARAM0 cpu_state.reg[a]
        b = (insn >> 11) & 0x1f;
        #define PARAM1 cpu_state.reg[b]
        c = (insn >> 0) & 0x7ff;
        c |= ((insn >> 21) & 0x1f) << 11;
        #define PARAM2 c
        {           /* "l_mtspr" */
          uint16_t regno = PARAM0 + PARAM2;
          uorreg_t value = PARAM1;

          if (cpu_state.sprs[SPR_SR] & SPR_SR_SM)
            mtspr(regno, value);
          else {
            PRINTF("WARNING: trying to write SPR while SR[SUPV] is cleared.\n");
            sim_done();
          }
        }
        #undef SET_PARAM
        #undef PARAM0
        #undef PARAM1
        #undef PARAM2

        if (do_stats) {
          current->insn_index = 139;    /* "l.mtspr" */
          analysis(current);
        }
      }
    } else {
      /* Invalid insn */
      {
        l_invalid ();

        if (do_stats) {
          current->insn_index = -1;    /* "???" */
          analysis(current);
        }
      }
    }
    break;
```

Bibliography

[1] M. C. M. Eftimakis, "High-speed serial fully digital interface between wlan rf and bb chips," June 2005. [Online]. Available: http://www.eurasip.org/Proceedings/Ext/IST05/papers/515.pdf

[2] J. v. d. L. A. v. R. A. Vidojkovic, V Tang, *Adaptive Multi-Standard RF Front-Ends*. Springer, 2008.

[3] M. C. Daniel Mattsson, "Evaluation of synthesizable cpu cores," Master's thesis, Chalmer Univeristy od Technology, 2004. [Online]. Available: http://www.gaisler.com/doc/Evaluation_of_synthesizable_CPU_cores.pdf

[4] R. Herveille, *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*, OpenCores Organization, September 2002. [Online]. Available: http://www.opencores.org/downloads/wbspec_b3.pdf

[5] OpenCores, *User's Manual Wishbone Builder*, OpenCores Organization. [Online]. Available: http://www.opencores.org/project,wb_builder,downloads

[6] A. E. Olle Seger, Per Karlström, *Laboratory manual for TSEA02*, Department of Electrical Engineering Linköping University, Sweden, October 2007. [Online]. Available: https://mail.cs.pub.ro/~laurentiu.duca/openrisc/labkomp.pdf

[7] J. H. Bahn, "Overview of network on chip." [Online]. Available: http://gram.eng.uci.edu/comp.arch/lab/NoCOverview.htm

[8] B. T. William DALLY, *Principles and Practices of Interconnection Networks*. morgan kaufmann publishers, 2003. [Online]. Available: http://cva.stanford.edu/books/ppin/

[9] D. Lampret, *OpenRISC 1200 IP Core Specification*, rev 0.2 ed., OpenCores, 6 september 2001. [Online]. Available: http://www.da.isy.liu.se/courses/tsea44/OpenRISC/or1200_spec.pdf

[10] *OpenRISC 1200 RISC/DSP core*, OpenCore.org. [Online]. Available: http://docs.huihoo.com/openrisc/openrisc1200-overview.pdf

[11] D. Lampart, *OpenRISC 1000 Architecture Manual*, OpenCores, April 5 2006. [Online]. Available: http://www.opencores.org/openrisc,architecture

[12] OpenCores, “Openrisc1200 rtl source code.”

[13] S. G. Kapil Anand, “Designing of customized signal processor,” Master’s thesis, IIT, May 2007. [Online]. Available: http://www.cse.iitd.ernet.in/esproject/homepage/release/or1200vector/pdfs/BTP_THESIS.pdf

[14] O. Seger, “A soft somputer.” [Online]. Available: http://www.da.isy.liu.se/courses/tsea44/coursemtrl_08/4-Computer-Ny.pdf

[15] M. E. J. B. Jeremy Bennett, Rich D’Addio, *OR1200 GNU Toolchain*, Opencores. [Online]. Available: http://www.opencores.org/openrisc,gnu_toolchain

[16] J. Bennett, *The OpenCores OpenRISC 1000 Simulator and Tool Chain*, Embecosm Limited, November 2008. [Online]. Available: http://www.embecosm.com/appnotes/ean2/html/index.html

[17] ——, *Or1ksim User Guide*, Embecosm, 2009. [Online]. Available: http://www.opencores.org/openrisc,or1ksim

[18] Embecosm, *The Or1ksim Simulator*, Embecosm. [Online]. Available: http://www.embecosm.com/appnotes/ean2/html/ch03s07.html

[19] M. Pfaff, *Advanced Methods of Verification*, 2008.

[20] S. Imam, *Step-by-Step Functional Verification with SystemVerilog and OVM*. Hansen Brown Publishing Company, 2008.

[21] D. T. Kropf, *Introduction to Formal Hardware Verification*. Springer, 1999.

[22] Cadence, *Open Verification Methodology User Guide*, version 2.0 ed., Cadence Design System, September 2008. [Online]. Available: http://www.ovmworld.org/

[23] J. Bennett, *Building a Loosly Timed SoC Model with OSCI TLM2.0*, 1st ed., EMBECOSM, June 2008, 1. [Online]. Available: http://www.embecosm.com/appnotes/ean1/sysc_tlm2_simple_or1k.pdf

[24] SystemC, *SystemC User Guide*, version 2.0 ed., SystemC.org, 2002. [Online]. Available: http://www.cse.iitd.ernet.in/~panda/SYSTEMC/LangDocs/UserGuide20.pdf

[25] Accellera, *SystemVerilog 3.1a Language Reference Manual*, Accellera, 2003. [Online]. Available: http://www.vhdl.org/sv/SystemVerilog_3.1a.pdf

[26] Gorlak, “A whirlwind tutorial on creating really teensy elf executables for linux.” [Online]. Available: http://www.muppetlabs.com/~breadbox/software/tiny/teensy.html

[27] M. Graphics, *ModelSim User’s Manual*, Mentor Graphics, May 2008. [Online]. Available: http://www.actel.com/documents/modelsim_ug.pdf

Appendix B

List of Acronyms

DSP	Digital Signal Processor
SoC	System on Chip
IHex	Intel Hexadecimal File format
ROM	Read Only Memory
RAM	Random Access Memory
OR1200	OpenRISC1200
ISS	Instruction Set Simulator
OVM	Open Verification Methodology
DPI	Direct Programming Interface
DUV	Design Under Verification
CDV	Coverage Driven Verification
IVC	Interface Verification Component
MVC	Module Verification Component
SVC	System Verification Component
OVC	Open Verification Component
VE	Verification Environment
BFM	Bus Functional Model
TLM	Transaction Level Modeling
PC	Program Counter

SR	Supervision Register
ESR	Exception Supervision Register
EPCR	Exception Program Counter Register
EEAR	Exception Effective Address Register
EA	Effective Address
CPU	Central Processing Unit
ALU	Arithmetic and Logic Unit
LSU	Load Store Unit
MAC	Multiply Accumulate Unit
IC	Instruction Cache
DC	Data Cache
IM	Instruction Memory
DM	Data Memory
IMMU	Instruction Memory Management Unit
DMMU	Data Memory Management Unit
RTL	Register Transfer Level
ORBIS32	OpenRISC Basic Instruction Set
RFE	Return From Exception
GPR	General Purpose Register
IWB	Instruction Wishbone Interface
DWB	Data Wishbone Interface
IF	Instruction Fetch
ID	Instruction Decode
EX	Instruction Execute
WB	Write Back
RF	Register File
NOP	No Operation
TLM	Transaction Level Modeling