Abstract

This project consists of two parts, Implementation part and Research part.

In the Implementation part, I implemented the verification environment associated with the OpenRISC 1200 (OR1200) core and used the OR 1200 Instruction Set Simulator (ISS) as reference model to demonstrate the functionality of the design. The main contributions and achievements in this part include,

- I implemented a SystemVerilog wrapper for the OR1200 core to facilitate the interaction between the OR1200 core and the verification environment, see pages 27-29.
- I modified the OR1200 Instruction Set Simulator (ISS) and implemented a C/C++ wrapper for the ISS to achieve the on-the-fly checking in the testbench, see pages 18-19, 29-32.
- I implemented single port synchronous 32-bit ROM/RAM model with WISHBONE interfaces to provide separate instruction and data memory for the OR1200 core, see pages 33-36.
- I installed the OR1200 GNU Tool chain and compiled three C/C++ programs as test cases to simulate in the testbench , see pages 15-17
- I tested the design with three test cases and collected code coverage (line coverage, toggle coverage and path coverage) of test cases to check the functional correctness and connectivity of blocks in the design, see pages 39.

In the Research part, I focused on the hypothesis that code coverage can be used as a proxy to qualify a test suite in terms of memory hierarchy usage or power consumption, so that a test suite contains a balanced set of tests that exercise all functionality of the design at all levels of the memory hierarchy. Therefore, this part aims to confirm this assumption. The contributions and achievements include,

- I designed six pairs of examples that each pair with same functionality has different usage of the memory hierarchy and then, analyzed the difference in usage of cache both qualitatively and quantitatively, see pages 40-49.
- Based on the different cost in the memory hierarchy, we can estimate the power consumption in the design. Compared to other architecture-level power analysis tools, code coverage is free and supported by most simulation tools, see page 49, 50-51.

Acknowledgements

First of all, I would like to thank my supervisor, Kerstin Eder and my industry advisors, Ulrich Hensel and Rainer Mann. Thank them for giving me this opportunity to work on this project at GLOBALFOUNDRIES, and thanks for their guidance and patience in the progress.

I would like to thank my parents. Thank them for everything they have done for me.

I would like to thank my friends for bringing so much pleasure in my life.

My deepest love to my wife, thank her for accompanying me for so many years.

At last, I would like to thank my unborn child for bringing so much happiness to my family, even before being born.

Table of Contents

List of Figures

List of Tables

1 INTRODUCTION

This section explains the motivation, aims and objectives of the thesis. The basic structure of the thesis is also introduced in this part.

1.1 Motivation

This project is defined in collaboration with GLOBALFOUNDRIES. It is a good chance for the university to know "what the industry needs" and "what can we do for the industry". Currently, GLOBALFOUNDRIES require a reference design and associated Verification/Test Environment to be used as a demonstrator for their technology capabilities, in particular as a demonstrator for digital reference flows that are silicon validated and demonstrate technology readiness. The reference design will serve as (i) test case for reference flow development; (ii) demonstrator for GLOBALFOUNDRIES technology metrics; (iii) technology qualification block on GLOBALFOUNDRIES test chips. The OpenRISC 1200 (OR1200) was selected for this development. We will focus on the implementation of the verification environment for the design.

Also, in GLOBALFOUNDRIES' reference flow, a large amount of test suites will be needed for the functional verification and power analysis. To estimate the usage of memory hierarchy or power consumption in the RTL design, code coverage has its distinct advantages compared to other architecture-level power analysis tools. Moreover, as power dissipation has become a critical issue in modern microprocessors, an efficient approach to estimate power consumption can also help software or compiler development for power efficient. Code coverage as free metrics in most simulation tools does not require any implementation of architecture level simulator and can be used in very early stage of the flow.

1.2 Aims and Objectives

The project proposed by GLOBALFOUNDRIES includes two stages. The first one is the development of a verification environment for the reference design, and the second one is the investigation of approaches to optimize power/performance/area tradeoffs for the OR1200 design. This project is focused on the implementation of the verification environment and the investigation of the use of code coverage as a high-level proxy for power consumption to qualify test suites. The aims and objectives of the thesis consist of two aspects, Implementation and Research. These are explained blew. They are explained below.

IMPLEMENTATION

The main goal of the thesis is to implement a verification environment associated with the reference design. The verification environment is used to demonstrate basic functional correctness of the design, connectivity of blocks and timing correctness. The OR1200 Instruction Set Simulator (ISS) is used as the reference model for verification. A sanity regression test suite including reset/initialization sequences and basic functional tests is intended to be reused for gate-level simulation.

RESEARCH

The research part of the thesis focuses on the use of code coverage as a proxy to qualify the test suite in terms of the memory/cache usage or power consumption, so that the test suite can contain a balanced set of tests that exercise all functionality of the design at all levels of the memory hierarchy.

1.3 Thesis Roadmap

The thesis is structured in chapters for different emphasis. Brief introduction to these chapters is given below.

Chapter 2 briefly discusses the OpenRISC 1200 (OR1200) core. It introduces the OpenRISC 1000 architecture, Instruction and Register Set, and some important blocks in the OR1200 core. WISHBONE interfaces, as standard interfaces used for the connection between the OR1200 core and external environment is explained in this chapter. Furthermore, it also discusses the OR1200 GNU tool chain and the OpenRISC 1000 Architecture Simulator (or1ksim). This chapter provides the necessary background information for the project.

Chapter 3 briefly discusses the verification approach we adopt in the verification process. The verification plan is explained including “what to verify” and “how to verify”. A brief structure of the testbench and basic components we would implement in the next step are also described in this chapter. Moreover, it also discusses various metrics in code coverage. In the research stage, we focus on the use of code coverage.

Chapter 4 discusses the hierarchy design in the memory system. The CPU registers, as the top level of the hierarchy cost least in access time and power consumption. Cache as a small and fast memory can essentially improve the performance and reduce the power consumption. Memory accesses cost the most in performance and power consumption. Based on the different costs in memory hierarchy, we can analyze the power consumption qualitatively for the test patterns.

Chapter 5 thoroughly describes the implementation of the verification environment including the design under verification (DUV), the reference model and the memory system. The functionality and interfaces of each module are explained in detail in this chapter.

Chapter 6 first introduces the test cases used to check the functionality of the design, and then explains several examples for different usage of the cache/memory. Each example shows that two pieces of code with same functionality have different accesses to cache/memory. Furthermore, it also discusses how code coverage can be used to show the accesses both qualitatively and quantitatively.

Chapter 7 concludes the thesis and suggests the future work.

Appendix A shows six pairs of examples we design to confirm the use of code coverage.

2 OPENRISC 1200

The OpenRISC 1200 (OR1200) is one implementation of the OpenRISC 1000 architecture. It is designed as an open source, 32-bit scalar RISC processor with Harvard architecture [1]. The basic architecture of the OpenRISC 1200 includes a 5-stage pipeline, memory management units (MMU) and units for DSP operations. It focuses on performance, simplicity, and efficient power management.

The main features of OpenRISC 1200 include: (i) Users can configure major characteristics of the core; (ii) The OpenRISC 1200 can perform 300 Dhrystone 2.1 MIPS at 300 MHz [1] under normal conditions at 0.18 µm fabrication process; (iii) It has a high performance cache and memory management systems; (iv) It contains WISHBONE SoC interconnection Rev. B3 compliant interface [1].

2.1 OpenRISC 1000 Family

The OpenRISC 1000 architecture is a family of 32- and 64-bit RISC microprocessors. It implements a Harvard architecture with the 32-bit wide instruction set and operating on 32- or 64-bit data. The architecture provides WISHBONE bus interfaces for instructions and data transfer. It contains some configurable modules that can be optionally implemented. All processors which have the name with the first digit '1' belong to the OpenRISC 1000 family, as shown in Figure 2.1. The second digit normally indicates the feature in the architecture, while the last two digits define the configuration in the real applications [2].

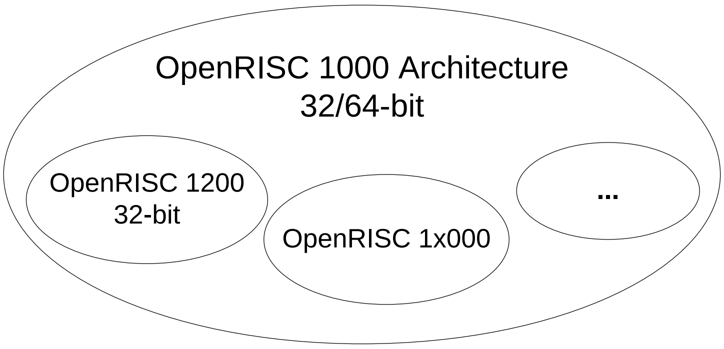


Figure 2.1OpenRISC 1000 Family

2.2 OpenRISC 1200 Architecture

The OpenRISC 1200 (OR1200) is the first RTL implementation of the OR1000 architecture. As shown in Figure 2.2, it includes a Central Processing Unit (CPU/DSP), Instruction and Data WISHONE interfaces, and optional configurations like, Instruction and Data Memory Management Units (IMMU, DMMU), Instruction and Data Caches (IC, DC), a Tick Timer, Programmable Interrupt Controller (PIC) and Power Management Unit.

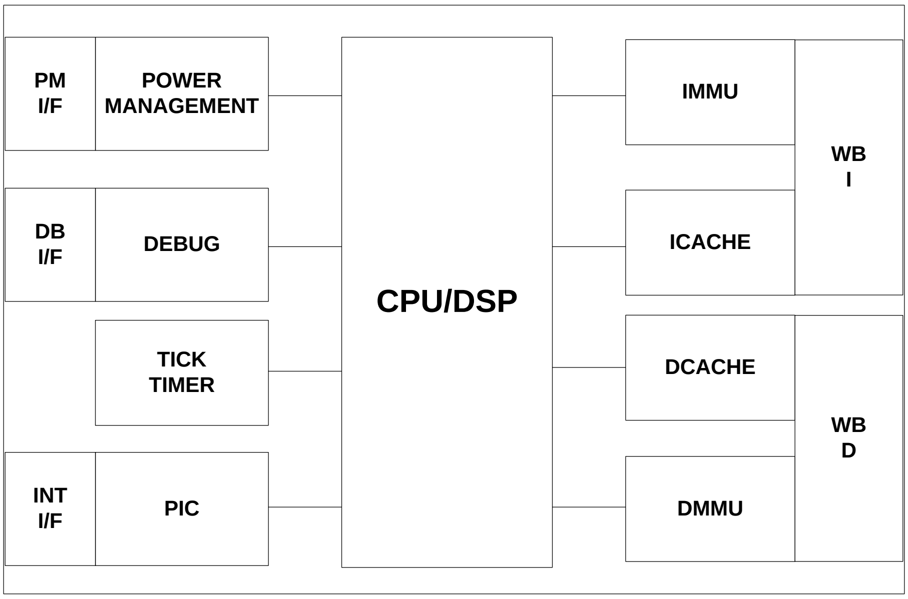


Figure 2.2 OpenRISC 1200 core

2.2.1 Instruction Set

Instructions implemented in the OR1000 architecture are 32-bit wide and 32-bit aligned in memory. The instruction set of the OR1000 architecture consists of 5 subsets. They are:

ORBIS32
OpenRISC Basic Instruction Set contains the basic 32-bit wide instructions. The instructions in this set include integer instructions, load/store instructions, basic DSP instructions, basic program flow instructions and some special instructions. Generally, these instructions operate on 8-, 16- or 32-bit data.

ORBIS64
This instruction set is also a basic instruction set in OR1000 architecture. It contains 64-bit wide instruction, including 64-bit integer instructions and 64-bit load/store instructions. Compared with ORBIS32, it can also operate on 64-bit data.

ORFPX32
OpenRISC Floating Point Extension Instruction Set contains 32-bit single precision floating point instructions. These instructions are used for the floating point operation on 32-bit data.

ORFPX64
This set contains 64-bit double precision floating point instructions and 64-bit load and store instructions.

ORVDX64

OpenRISC Vector/DSP Extension Instruction set contains the instructions for vector/DSP processing. It can operate on 8-, 16-, 32- and 64-bit data

In each instruction subset, the instructions are further grouped into Class Ⅰ and Ⅱ. Only class Ⅰ instructions are required to implement in the OR1000 architecture.

Instruction Formats

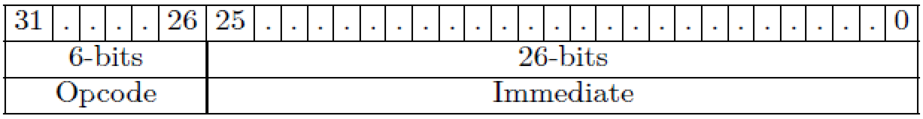
Similar to other RISC architecture, OpenRISC Instruction Set has six formats of instructions. They are the Immediate type (I), Register-to-Register type (R), Register-with-Immediate type (RI) and second Register-with-Immediate type (RI2), and types for setting the flag bit in the status register Set Flag Register-with-Immediate (RSFI) and Set Flag Register Format (RSF).

Immediate (I) type consists of a 6-bit opcode and 26-bit immediate part. The instructions in this type include the basic jump, branch instructions and some system instructions, for example, l.j, l.jal, l.bf and l.sys. Figure 2.3 shows the format of Immediate (I) type. Taking l.jal as an example, l.jal performs “jump and link”. The syntax of l.jal is,

l.jal ${Immdiate}

The function is,

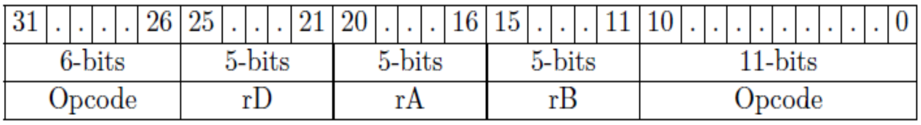
PC <– exts(Immediate << 2) + PC;
LR <– PC + 8;



31 26	25 0
6-bits	26-bits
Opcode	Immediate

Figure 2.3 Immediate Format (I)

Register-to-Register (R) type instructions include some basic arithmetic operations, like l.add, l.sub, l.xor and so on. The operands are the registers in the instruction, as shown in Figure 2.4. The second opcode is used for further distinction in the decode stage, as shown in Table 2.1.



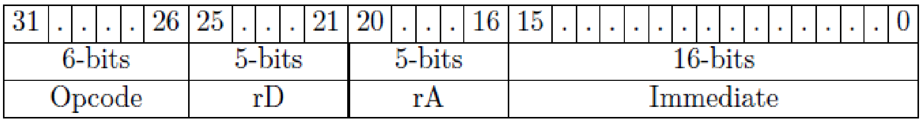
31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 0
6-bits	5-bits	5-bits	5-bits	11-bits
Opcode	rD	rA	rB	Opcode

Figure 2.4 Register-to-Register Format (R)

Opcode	Mnemonic	Opcode	Format
0x38	l.add	0x0	R
0x38	l.addc	0x1	R
0x38	l.sub	0x2	R
0x38	l.and	0x3	R
…	…	…	…

Table 2.1 Register-to-Register Format (R)

Register-with-Immediate (RI) type contains an immediate part as one operand. The instructions in this type include some load instructions, and arithmetic instructions on immediate values, for instance, l.lwz, l.lws and l.xori.



31 26	25 . . . 21	20 . . . 16	15 0
6-bits	5-bits	5-bits	16-bits
Opcode	rD	rA	Immediate

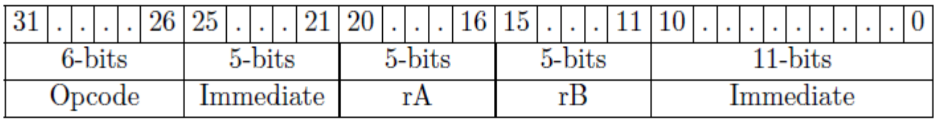
Figure 2.5 Register-with-Immediate Format (RI)

Second Register-with-Immediate (RI2) type consists two immediate parts, compared to RI type. The instructions in RI2 include the basic store instructions, l.sw and l.sb. For l.sb, the syntax is,

l.sb ${Immediate}($rA), $rB

It performs like,

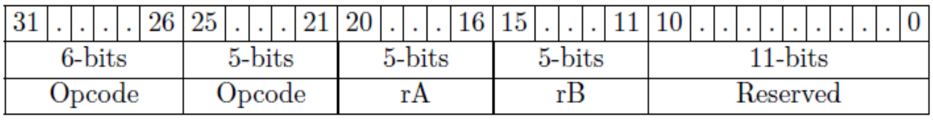
[rA + Immediate][7:0] <- rB;



31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 0
6-bits	5-bits	5-bits	5-bits	11-bits
Opcode	Immediate	rA	rB	Immediate

Figure 2.6 Second Register-with-Immediate Format (RI2)

Set Flag Register-to-Register (RSF) type sets the Flag bit in the supervision register (SR), which can be used for conditional execution. For example, l.sfeq sets the Flag bit in SR, when register A is equal to the register B. The second 5-bit opcode is used to further distinct the instructions, as shown in Table 2.2.



31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 0
6-bits	5-bits	5-bits	5-bits	11-bits
Opcode	Opcode	rA	rB	Reserved

Figure 2.7 Set Flag Register-to-Register Format (RSF)

Opcode	Mnemonic	Opcode	Format
0x39	l.sfeq	0x0	RSF
0x39	l.sfne	0x1	RSF
0x39	l.sfqtu	0x2	RSF
0x39	l.sfgeu	0x3	RSF
…	…	…	…

Table 2.2 Set Flag Register Format (RSF)

2.2.2 Register Set

The operation of the system is controlled by the registers. The register set has two subsets, General Purpose Registers and Special Purpose Registers.

General Purpose Registers

The GPRs is used to store the data during the execution. There are 32 general purpose registers implemented in the OR1000 architecture. They are either 32- or 64-bit wide depending on the implementation. Register zero, r0 is always zero as default.

Special Purpose Registers

The SPRs is used to control the modules in the OR1000 architecture. They are 32-bit wide. The essential SPRs in the OR1000 architecture include the supervision register (SR), exception program counter (EPC), exception status register (ESR) and exception effective address register (EAR). The details are listed in the following table.

Group Num	Reg Num	Reg Name	USR Mode Permission	SPR Mode Permission	Description
0	17	SR	-	R/W	Supervision Register
0	32-47	EPCR0-15	-	R/W	Exception PC Register
0	48-63	EEAR0-15	-	R/W	Exception EA Register
0	64-79	ESR0-15	-	R/W	Exception SRs

Table 2.3 OpenRISC 1000 SPRs

2.2.3 CPU/DSP

The CPU//DSP in the OR1200 core is shown in Figure 2.8. The OR1200 CPU/DSP implements the 32-bit part of the OR1000 architecture including Instruction unit, Register Set, Load/Store unit (LSU), Arithmetic and Logic Unit (ALU), Multiply Accumulate Unit (MAC), System unit and Exception unit.

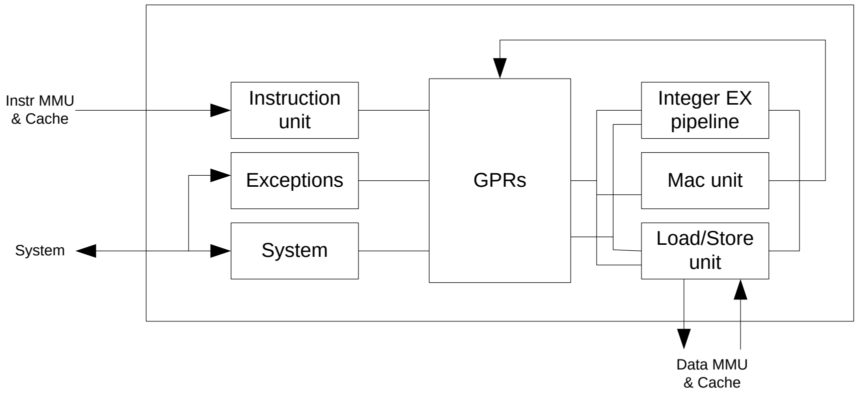


Figure 2.8 CPU/DSP

Instruction Unit

The CPU in the OR1200 core implements a 5-stage integer pipeline. The instruction unit is used to fetch instructions from the instruction cache or memory system based on the program counter (PC). The program counter (PC) is one of the most important registers. It stores the address of the next instruction to be fetched; also it may be used to calculate the target address of some relative branch instructions. After fetching the correct instructions, it will dispatch them into the appropriate execution units like LSU, ALU or MAC unit. Some special instructions like conditional/unconditional branch, jump will be executed in the instruction unit directly to save execution cycles for those instructions. As discussed in 2.1.1, Class Ⅰ of the “OpenRISC Basic Instruction Set” (ORBIS32) is mandatory to implement in the OR1000 architecture. The instruction format is explained in details in 2.1.1.

Load Store Unit (LSU)

The LSU is used to transfer data between the CPU and the memory system, which handles all load and store instructions. It is implemented as an independent block in the OR1200 core. If there is a data dependency in the pipeline, the LSU is stalled independently without affecting the master pipeline. The requests from the LSU will be forwarded to the embedded memory (if implemented), then to the data cache (if implemented). It normally takes two clock cycles to load to execute a load instruction, if it is a hit in the cache.

Arithmetic and Logic Unit (ALU)

The ALU handles all arithmetic and logic instructions. The execution of this instruction takes one clock cycle in general. Some set-flag instructions are also executed in the ALU, which set the flag bit in the supervision register after execution.

Multiply Accumulate Unit (MAC)

A fully pipelined MAC is optionally implemented in the OR1200 CPU. It is used for the multiply and divide instructions.

System Unit

The system unit connects the CPU signals with the system signals. It is used to control the read/write operations to the SPRs in the CPU.

Exception Unit

The exception unit is used to handle the exception in the OR1200 core. Each exception has a default handler address. Whenever, the exception occurs, the PC will jump to that default address pre-defined.

2.2.4 Caches and Memory system

Harvard memory architecture is implemented in the OR1200 core. Separate instruction and data caches are configurable in the architecture. The data cache (DC) is scalable from 1 KB to 8 KB, and the instruction cache (IC) can be scaled from 512 B to 8 KB. The Least-Recently-Used (LRU) replacement strategy is applied. Only the write-through operation is supported in the DC.

Two memory management units are also implemented separately for instruction and data memory. It has 8-KB page size translation-lookaside buffers (TLBs) and comprehensive page protection scheme. Both instruction/data TLBs are directed-mapped, hash based and scalable from 16 to 128 entries per each way. The default configuration is 64 entries with 8-KB page size.

2.2.5 Quick Embedded Memory (QMEM)

The QMEM is optionally implemented in the OR1200 core. The main purpose of the QMEM is to put some time critical functions into this memory and have predictable and fast access to these functions. Since the QMEM is shared by both the IC and DC, the instruction fetching may slow the performance of the data load/store operations.

2.2.6 Store Buffer (SB)

The SB is optionally implemented in the OR1200 core. It has a FIFO buffer which saves all the store operations from the CPU. The SB connects the DC with the data WISHBONE interface. Since the OR1200 implements a write-through strategy, all store instructions access to memory. The SB is essential for the performance of the

CPU. To make sure the correctness of the execution, all load operations are executed, only when the FIFO buffer is empty.

2.2.7 Power Management Unit (PM)

To optimize the power consumption, a sophisticated power management unit is implemented in the OpenRISC 1200. It is used to control the power consumption during the process. Different power modes are supported including slow and idle modes, doze mode and sleep mode [1]. In the slow and idle modes, the clock frequency reduction is controlled by the software. In the doze mode and sleep mode, the processor can be waked up by interrupts generated by some external peripherals. The power consumption reduction for each mode is shown in Table 2.4.

Power Mode	Approx. Power Consumption Reduction
Slow and Idle mode	2x – 10 x
Doze mode	100x
Sleep mode	200x
Dynamic clock gating	N/A

Table 2.4 Power Consumption Reduction

2.2.8 Debug Unit (DU)

The debug unit is optionally implemented in the OpenRISC 1200 core. It supports the basic debugging, but does not provide any advanced features like watch-points, breakpoints and program flow control registers [1] [2].

2.2.9 Tick Timer (TT)

The OpenRISC 1200 provide a tick timer which is clocked with the RISC clock. It is used by the operating systems for precise system task scheduling and time measurement [1][2]. The timer is associated with a 32-bit register. Therefore, the maximum range of the timer is 232 clock-periods, and the maximum period between the interrupts is 228 clock-periods. The tick timer also provides a maskable interrupt.

2.3 WISHBONE

The OpenRISC 1200 core is connected to external memory system through two WISHBONE interfaces, Instruction WISHBONE interface and Data WISHBONE interface. Instruction WISHBONE interface is used to fetch instructions from instruction memory, while Data WISHBONE interface is intended for the data exchange between the core and memory. The description of WISHBONE interfaces can be found in WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores [3].

2.3.1 Objectives & Features

The WISHBONE SoC Interconnection architecture is designed as a general purpose interface for semiconductor IP (Intellectual Property) cores [3]. It is intended to define a standard bus protocol for data exchange among IP cores.

The main objectives of WISHBONE interface include, (i) define a flexible methodology for structured IP cores to enforce compatibility and reusability; (ii) create a flexible and reusable interface to facilitate the use of IP cores for both core developers and end users; (3) provide a independent interface from the logic signals and functionality of various IP cores; (4) prevent over-specified design from constraining the creativity of developers or end users; (5) easy the integration process in large projects.

It is believed that all these goals have been accomplished [3]. In fact, the WISHBONE architecture, as a standard interconnection scheme behaves quite similar to the traditional computer bus. It provides a robust and flexible interconnection between IP cores which effectively reduces the integration problems. Features of WISHBONE architecture are listed in the following.

- very few logic gates are required for IP cores interconnection;
- provide various data exchange protocols including READ/WRITE, BLOCK transfer, RMW;
- support both BIG ENDIAN and LITTLE ENDIAN data ordering;
- define MASTER/SLAVE architecture in the protocol;
- support different interconnection methods including point-to-point, shared bus, crossbar switch and switched fabric interconnections.

2.3.2 Interface Specification

As shown in Figure 2.9, there are three types of signals in the WISHBONE interface. They are SYSCON Module Signals, MASTER Signals and SLAVE Signals.

Signal Description

SYSCON Module Signals

CLK_O – system clock output [CLK_O] is generated by the SYSCON module.
RST_O – reset output [RST_O] is generated by the SYSCON module.

MASTER Signals

CLK_I – system clock input.
RST_I – reset input.
DAT_I – input data from the SLAVE interface.
ACK_I – acknowledge signal from the SLAVE interface.
ERR_I – error signal indicates an abnormal cycle termination.
RTY_I – retry input presents the SLAVE interface is not ready.
DAT_O – output data to the SLAVE interface
ADR_O – output address to the SLAVE interface
CYC_O – cycle signal indicates if the current bus cycle is valid or not.
SEL_O – select signal indicates the valid bytes in data.
STB_O – strobe signal indicates a valid data transfer cycle.
WE_O – write enable output indicates the current cycle is READ/WRITE.

SLAVE Signals

CLK_I – system clock input
RST_I – reset input
DAT_I – input data from the MASTER interface.
ADR_I – input address from the MASTER interface.
CYC_I – cycle signal indicates if the current bus cycle is valid or not.
SEL_I – select output indicates the valid bytes in data.
STB_I – strobe input indicates a valid data transfer cycle.
WE_I – write enable output indicates the current cycle is READ/WRITE.
ACK_O – acknowledge signal indicates a normal termination of a bus cycle.
DAT_O – output data to the MASTER interface.
ERR_O – error output indicates an abnormal termination of a bus cycle.
RTY_O – retry output indicates the SLAVE interface is not ready.

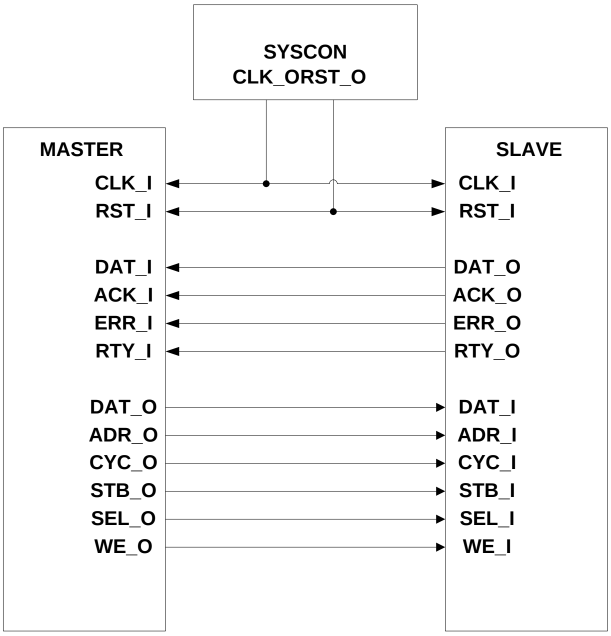


Figure 2.9 WISHBONE interface

Handshake Protocol

The bus cycles between the MASTER and SLAVE interfaces use a handshake protocol. As shown in Figure 2.10, once the MASTER interface asserts the cycle signal [CYC_O] and strobe signal [STB_O], it indicates that the current bus cycle is valid. The SLAVE interface will read in the request and assert the acknowledgement signal to indicate a termination of the bus cycle. The terminating signal will be sampled at the rising edge of the system clock.

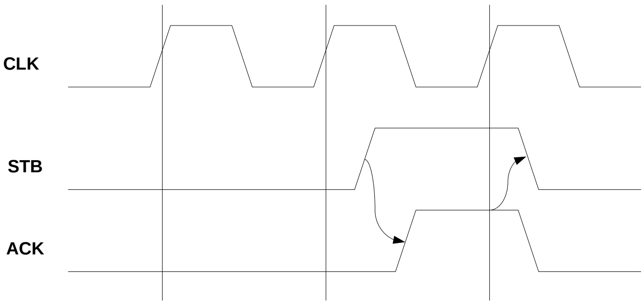


Figure 2.10 Handshake Protocol

WISHBONE READ/WRITE Bus Cycles

The WISHBONE interface has classical single READ/WRITE bus cycle, as shown in Figure 2.11 and 2.12. Once the strobe signal (STB) and cycle signal (CYC) are asserted by the MASTER, it waits for the terminating signal (ACK, ERR, or RTY). After the SLAVE asserts the terminating signal (ACK), the strobe and cycle signals in the MASTER side go low. The SLAVE can insert any number of wait cycles as long as the terminating signal is low, the MASTER will wait. The write enable signal (WE) is used to identify the current bus cycle is READ or WRITE.

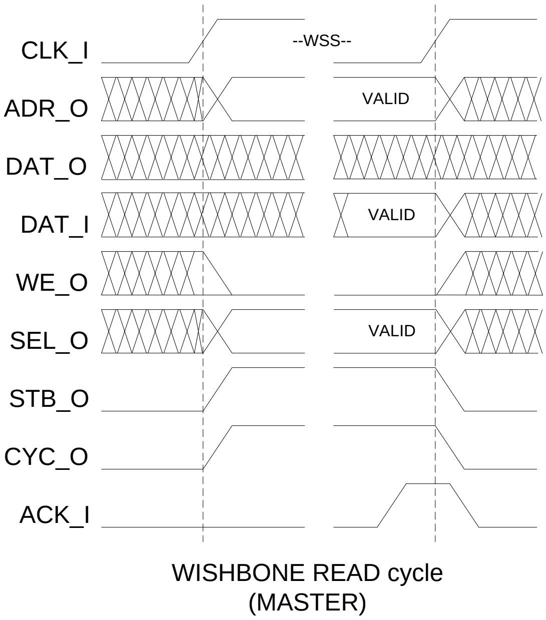


Figure 2.11 WISHBONE Read cycle

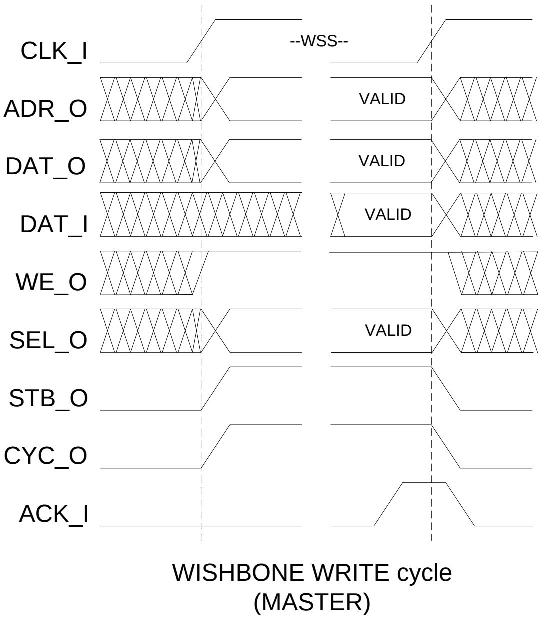


Figure 2.12 WISHBONE Write cycle

2.4 OpenRISC 1200 Tool chain

The OpenRISC 1200 architecture uses GNU tool chain for software development [4]. The OpenRISC 1200 tool chain is developed and maintained by the OpenCores organization. All source code for the tool chain can be downloaded from the OpenCores website. There are also some scripts available to build the tool chain. After a few changes in the script (MOF_ORSOC_TCHN_v5c_or32-elf.sh), it can be used to build the OpenRISC 1200 tool chain on Linux.

2.4.1 Installation

The tools to be installed 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
- or1ksim-0.4.0

More details can be found in OpenRISC 1200 GNU tool chain. [5][6][7]

Before running the script, there are some basic development tools required including

- build-essential
- make
- gcc
- g++
- flex
- bison
- patch
- texinfo
- libncurses-dev

After installing these tools, we can run the script and build the tool chain. The whole tool chain requires about two GB space. After the installation, we need to add the path to or32-elf directory in the Bash file. The steps are shown in the following.

```
vim      ~/.bashrc
export   PATH=$PATH:/…/or32-elf/bin/
```

2.4.2 Memory Initialization

After the OpenRISC 1200 tool chain is installed, we are capable of compiling and simulating applications on the OpenRISC 1200 architecture. To initialize the memory system in the OpenRISC 1200 architecture, we need to convert the high-level programs into binary data. The Intel Hex (ihex) file is a common format as memory initialization files [5].

```
:10010000180000001820000018400000186000000CF
:10011000188000018A0000018C0000018E00000BF
:1004700018600000A86380109D600000D403200075
:0C0480008441000044004800 9C2100045E
:10048C000000000000000000000000000000000060
:0400000300000100F8
:00000001FF
```

Figure 2.13 Intel Hex (iHex) file

There are three types of Intel HEX file determined by different byte orders: 8 bit, 16 bit and 32 bit. Figure 2.13 shows an example of Intel HEX file. Each line of the file contains six fields. They are start code, byte count, address, record type, data and checksum.

Start code
Start code is a single character ASCⅡ colon ':'. It indicates the beginning of each line.

Byte count
After the start code (:), the next two hex digits represent the number of bytes in the data field. For example, if the byte count is 0x10, it means there are 16 bytes of data contained in the data field.

Address
After the byte count part, the next four hex digits determine the 16-bit address of the beginning of data in the memory. This big-endian address has the limit of 64 KB.

Record type
After the address, the next two hex digits, from 0x00 to 0x05 define the type of the data field. There are six types in the data field. 0x00 means that the data field contains both data and a 16-bit address, 0x01 indicates the end of file and 0x03 means the beginning of address segment.

Data
Data field contains a sequence of data. As the byte count shows, it is represented by 2n hex digits.

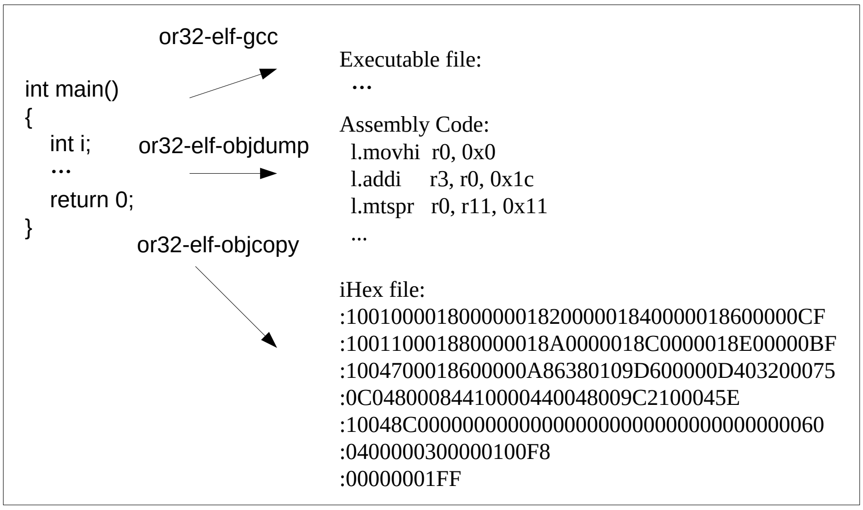
Checksum

The last two hex digits are the last least significant byte of the two's complement sum of the values in all fields except the start code (:) and the checksum itself. It is calculated by adding all hex digit pairs together. For example, if a line is 1001000018000000182000001840000018600000CF, the checksum should be,

10+01+00+00+18+00+00+00+18+20+00+00+18+40+00+00+18+60+00+00 = 131

and the two's complement of 131 is CF, which is the checksum field.

To generate the Intel HEX file for the OpenRISC 1200 architecture, we can use the GNU tool chain to compile a C/C++ program, as shown in Figure 2.14. We use a Makefile to compile the test program, a linker script file to set up the memory map of the applicant, and start-up script to initialize the architecture including clearing setting r0 to zero, invalidating each cache line and enabling instruction and data cache.



```
int main()
{
    int i;
    ...
    return 0;
}
```

Executable file:
...

Assembly Code:

```
  l.movhi  r0, 0x0
  l.addi     r3, r0, 0x1c
  l.mtspr   r0, r11, 0x11
  ...
```

iHex file:

```
:1001000018000000182000001840000018600000CF
:1001100018800000018A0000018C0000018E00000BF
:10047000186000000A86380109D600000D403200075
:0C048000844100004400480009C2100045E
:10048C0000000000000000000000000000000000060
:0400000300000100F8
:00000001FF
```

Figure 2.14 OR1200 GNU Toolchain flow

2.5 OpenRISC 1000 Architecture Simulator

The OpenRISC 1000 Architecture Simulator (ISS) is intended for the high-level simulation of the OR1000 architecture, which is capable of early code and performance analysis [7]. It is developed and maintained by the OpenCores organization. We can download freely from OpenCores website. The latest version of the simulator is or1ksim-0.4.0, which can work standalone or be used as a library [4].

2.5.1 Installation of or1ksim Library

When we install the OpenRISC 1200 GNU tool chain, the or1ksim simulator is also installed. However, it can only work standalone to emulate OpenRISC 1000 architecture. To use it as a reference model in the verification environment, we need to compile it as a library. The steps are provided below:

- Download and untar the source code in a separate directory in which to be build it:
  ```
  tar jxf or1ksim-0.4.0.tar.bz2
  mkdir or1ksim-0.4.0/install
  cd or1ksim-0.4.0/install
  ```
- Set environment variables:
  ```
  CC=/gcc-4.2.2-linux_x86_64/bin/gcc
  CXX=/gcc-4.2.2-linux_x86_64/bin/g++
  ```
- Configure the software using the configure script, specify the installation path:
  ```
  configure –prefix=/home/…/or1ksim-0.4.0/install
  ```
- Build the tool with:
  ```
  make all
  ```
- Install the tool with:
  ```
  make install
  ```
- Add the library directory and link the library
  ```
  export LD_LIBRARY_PATH=
      /home/…/or1ksim-0.4.0/install/lib:$LD_LIBRARY_PATH
  ```

2.5.2 Or1ksim Library

In the or1ksim library, there are some basic functions exported by the or1ksim simulator. The declaration can be found in the header file of the library, or1ksim.h. Some important functions are listed below.

-
  ```
  int or1ksim_init( const char *config_file,
                    const char *image_file,
                    void        *class_ptr,
                    int   (*upr) (  void *class_ptr,
                                    unsigned long int addr,
                                    unsigned char     mask[],
                                    unsigned char          rdata[],
                                    int               data_len),
                    int   (*upw) (void *class_ptr,
                                    unsigned long int addr,
                                    unsigned char          mask[],
                                    unsigned char          wdata[],
                                    int               data_len)
                    );
  ```

- int or1ksim_run(double duration)

- void or1ksim_reset_duration(double duration)

The initialization function, or1ksim_init() contains the arguments including the name of the configuration file (config_file), an executable image file (image_file), a pointer to the calling class (class_ptr) and two up-call functions, one for reads (upr); one for writes (upw). upr and upw functions are called whenever there is read/write to the external address space. Both functions will return zero on success, otherwise return non-zero.

The functions, or1ksim_run() and or1ksim_reset() can be used to set/reset the running time of the simulator.

To use the or1ksim library as a reference model in the verification environment, some modifications are needed in the library. For example, the execution of instructions should be controlled by the testbench. Moreover, after each execution, the state of the simulator should be kept in the testbench for the comparison with the state of the OR1200 core. The details will be explained in Chapter 5.

3 VERIFICATION PROCESS

In this chapter, we introduce the verification plan and analyze the approaches and techniques used to implement the verification environment.

3.1 Verification Plan

The functional verification for a sophisticated processor is always a challenging and critical task. We plan to use a gray/white-box approach with a simulation-based verification. The basic verification environment includes the design under verification (DUV) and a testbench. The design under verification is the OpenRISC 1200 core with a SystemVerilog wrapper, and the OR1200 Instruction Set Simulator can be used as a golden reference model in the testbench.

What to verify

The OpenRISC 1200 core, as a complex microprocessor has many internal states which depend on both the input signals and the previous state in the architecture. In this case, to verify all aspects of the behavior of the processor is relatively difficult. To make sure the correct execution of input instructions in the processor, we plan to verify the following aspects in the OpenRISC 1200 core.

- The Program Counter (PC).
- The instruction executed.
- All General Purpose Registers (GPRs).
- Some important Special Purpose Registers (SPRs).

How to verify

Since the OR1200 core implements a Harward architecture, we need to implement a separate instruction and data memory. The test programs should be loaded into the instruction memory before the simulation. Then, in order to verify the aspects mentioned above, we need to implement an efficient reference model to provide the correct behavior of the processor. The OR1200 Instruction Set Simulator (ISS) can be used as the reference model in the testbench. During the simulation, instructions are executed by both the DUV and the ISS independently. Then, the expected and actual results are compared in the testbench.

3.2 Testbench Design

A testbench provides the environment which the design under verification (DUV) can be connected to. It normally includes (i) a stimulus generator, (ii) a reference model, and (iii) a checker.

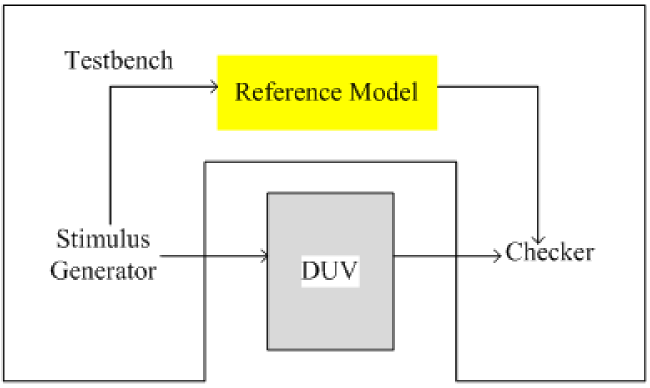


Figure 3.1 Structure of a testbench

The stimulus generator, also called driver is used to generate a sequence of test cases driven into the DUV. Since the Harvard architecture is implemented, the memory system is loaded with instructions and data before the simulation. The driver generates the clock and reset signals to boot the OR1200 core. The OR1200 will fetch instructions from the instruction memory and exchange data with the data memory based on the system clock.

Design under verification (DUV) is representing the model undergoing verification. In this project, we verify the OR 1200 core as a gray/white box. All interactions between the DUV and the verification environment will be through the standard interfaces. As described in Chapter 2, the OR1200 core has two WISHBONE interfaces for instructions and data. Moreover, we need to use another interface to keep the status of the core during the simulation. This interface connects the necessary internal signals with the testbench, which is used to check the behavior of the core.

The checker is used to automatically check and output any errors occur during the simulation. It compares the results from DUV with the expected results from the reference model. Since the OpenRISC 1200 core has a 5-stage pipeline implementation, we need some synchronization mechanism to make sure that the timing issue occurs in the hardware is handled in the verification environment.

The OR 1200 Instruction Set Simulator (ISS) is used as the reference model in the testbench. The ISS is also named or1ksim, which is an open source simulator for the OpenRISC 1000 architecture. It provides a high-level emulation for the OR 1200 core. The or1ksim simulator can run standalone or be used as a library. By declaring the up-call functions of the or1ksim simulator in the testbench, it can be used as a separate library in the verification environment.

3.3 Code Coverage Analysis

Code coverage, as the most popular functional coverage metric can be used to measure how much of the DUV in the testbench has been exercised by the test case during the simulation [10]. Code coverage analysis normally provides the information like how many lines have been executed, how many times the branch is taken, or how many signals have been toggled. The main metrics in code coverage include statement/line coverage, condition coverage, branch coverage, path coverage, toggle coverage and so on.

Statement/line Coverage

Statement coverage is the easiest type of the metrics. It indicates whether each statement/line has been executed or not. 100% statement coverage means all statements have been executed at least once. This metric can be applied directly to the source code without any processing. However, it has the shortcoming of the insensitivity to the control structures, like if-else statements or branches. For example, in the following piece of code, if a is less than b and c is larger than d, only statement 3 is executed. Statement coverage will ignore statement 1 and 2.

```
always @(posedge clk)
begin
    if ( a > b)
        statement 1;
    if ( c < d)
        statement 2;
    else
        statement 3;
end
```

Branch Coverage

Branch coverage solves the problems of statement coverage. It checks whether the Boolean expressions in the control structures have been evaluated to both true and false or not. For example, in a “if” statement, branch coverage checks whether the condition is evaluated to both true and false, no matter if there is “else” statement. For example, in the piece of code above, branch coverage will check if all cases, (a is less than b, a is larger than b, c is less than d and c is larger than d) has been tested.

Path Coverage

Since the conditional statements result in different paths in the flow, the path coverage reports whether all possible routes in the program have been executed or not. The disadvantages of this metrics include, some of the paths are impossible to be exercised and the number of paths is usually exponential to the number of branches.

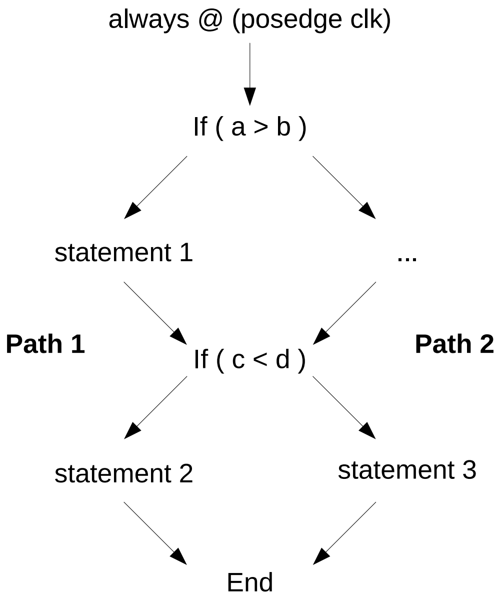


Figure 3.2 Possible paths

Toggle Coverage

The toggle coverage shows the transactions of signals in the design. It measures the changes of bits in the logic that have been toggled from 0 to 1 or from 1 to 0. It is widely used in gate level simulation, especially for the power analysis.

Code coverage analysis can be used to improve the verification process by detecting the gaps in the test cases and removing the redundant ones. Although we cannot rely on code coverage alone, it provides a cheap and effective way to qualify the test suite. And if we study further about code coverage, it also presents the characteristics of the test programs like the memory hierarchy usage, which is quite useful for the power consumption analysis [11].

4 MEMORY ACCESS COSTS

While the performance gap between processor and memory is still increasing, memory accesses also cost a large faction of power budget of the system. The memory hierarchy design not only improves the performance complying with the principle of locality, it also reduces the power consumption by minimizing the memory accesses. Programs with different memory usage often have different power consumption. For example, programs with a large amount of memory accesses usually have high power consumption, while cache hits can decrease the number of memory accesses which reduce the power consumption as well. Therefore, we can predict the power consumption of a program by observing its memory/cache usage.

4.1 Memory Hierarchy

The principle of locality says that, programs do not access all code or data uniformly. They always tend to execute a limited amount of code or data more frequently than the other [12]. Based on this principle, a memory hierarchy is designed to provide a memory system with low cost, but high speed. The hierarchy consists of several levels. The cheapest level of memory has the lowest cost and worst access time, which the fastest level is always expensive and small. Ideally, the memory hierarchy should cost as low as the cheapest level and perform as fast as the fastest level [12].

Each level in a memory hierarchy has different capacities, costs and access time. The level with smaller size normally has higher speed, which is closer to the CPU. A typical memory hierarchy is shown in Figure 4.1.

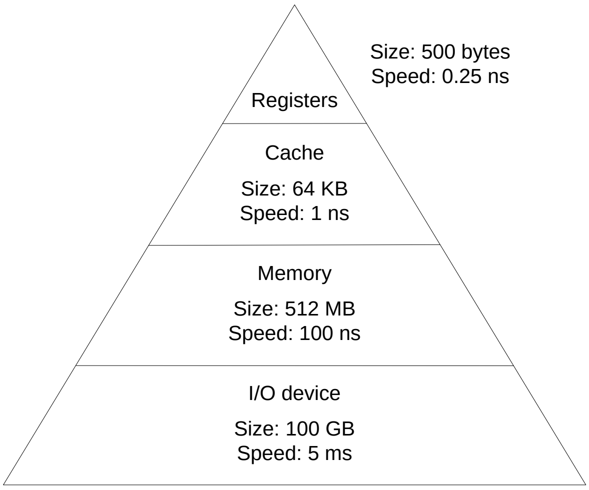


Figure 4.1 Typical Memory Hierarchy

Registers

CPU registers are at the top level of the memory hierarchy. Generally, they provide the fastest accesses to the most frequently accessed data. They can be accessed more quickly than the memory. However, the CPU only has a small amount of registers used for arithmetic and control operations. When the registers are not enough for the storage, the CPU will try to access the next level which is the small and fast cache memories that are close to the CPU.

Cache

Cache, as a small and fast memory improves the performance of the CPU complying with both temporal and spatial locality. When the CPU finds the data in the cache, it is called a cache hit. When the data is not in the cache, it is called a cache miss, and then the CPU needs to retrieve the data from the main memory. Cache is often organized in multiple levels between the CPU and the main memory, named level-1 (L1), level-2 (L2) and so on. The capacity of a cache is defined by the total size of the cache. For example, in the OR1200 core, the default cache has the capacity of 8 KB. In the cache, the block used to transfer data is called cache lines. A cache line is the smallest unit to transfer data between the cache and the main memory. The associativity of the cache determines how the address in the main memory is mapped into the cache line. While the cache miss occurs, the CPU is stalled for several clock cycles to wait for the memory access, which is called the memory stall cycles. Then, the performance of the CPU can be represented by the execution time and the stall time, like

$$\text{CPU execution time} = (\text{CPU cycles} + \text{Memory stall cycles}) \times \text{Cycle time}$$

The cache miss penalty is defined by both the number of misses and the cost per miss in the cache. To improve the performance of the cache, we can either reduce the miss penalty or reduce the number of cache misses.

Memory

When the cache miss occurs, memory accesses are needed to load or store the data. Memory access is much slower than the cache access. The time required for the cache miss depends on both the latency and bandwidth of the memory. The memory latency determines how long it takes for the memory to responds the request from the CPU. And the memory bandwidth constraints how much data it can transfer each time. Table 4.1 shows each level in the memory hierarchy.

Level	1	2	3
Name	Registers	Cache	Memory
Typical Size	< 1KB	< 16 MB	< 16 GB
Access time (ns)	0.25-0.5	0.5-25	80-250
Bandwidth (MB/sec)	20,000-100,000	5000-10,000	1000-5000
Managed by	Compiler	Hardware	Operating System
Backed by	Cache	Memory	Disk

Table 4.1 Typical levels in Memory Hierarchy

The average memory access time is widely used to measure the performance of the memory system. It is defined as,

$$\text{Average memory access time} = \text{Hit time} + \text{Miss penalty} \times \text{Miss rate}$$

where Hit time is the time required for a cache hit, Miss penalty is the time required for a cache miss, and Miss rate represents the number of misses in the cache.

4.2 Power Consumption in Memory Hierarchy

Each level in memory hierarchy has different size, access time and power consumption. The higher level normally has smaller size, faster access time and lower power consumption. Ideally, we should use higher level as much as possible before accessing the lower level. CPU Registers, as top level in the memory hierarchy are implemented in the register file inside the processor. It has the shortest access time and lowest power consumption, but the smallest size. For example, in the OR1200 core, there are only 32 32-bit general registers.

Cache is a small and fast memory which is much closer to the CPU than the main memory. Cache accesses are much faster and cheaper than memory accesses. Therefore, in order to improve the performance and reduce power consumption, the techniques that increase cache hit rates or reduce cache miss rates are applied. The most effective technique to reduce memory accesses is the compiler optimization, which optimize the memory accesses by using some loop optimization techniques, including loop unrolling, loop interchange, loop reversal [16][17][18].

Memory accesses not only dominate the performance, they also cost a large fraction of the power budget [19]. Memory consumes dynamic power when there is either read or write operation. The dynamic power dissipated by a memory includes (i) toggling of the system clock network; (ii) decode logic for address; (iii) bit-lines in the memory array; (iv) memory cells charging or discharging; (v) registers latching in memory inputs/outputs for address/data [14][20].

5 VERIFICATION ENVIRONMENT

To demonstrate the basic functionality of the OpenRISC 1200 core, we use simulation based verification for the OR1200 core by implementing a testbench. The basic components of the testbench include the driver, the design under verification (DUV), the monitor, the golden reference model and the memory system.

5.1 Design under Verification

The design under verification (DUV) is the OR1200 core. To facilitate the interaction between the OR 1200 core and the verification environment, a SystemVerilog wrapper is implemented.

5.1.1 OR1200 Core

As shown in Figure 5.1, the OR1200 core consists of many configurable modules. Some modules are not essential for the verification environment. In this case, we only enable the necessary modules including CPU/DSP, Instruction and Data WISHBONE, and Instruction and Data Cache.

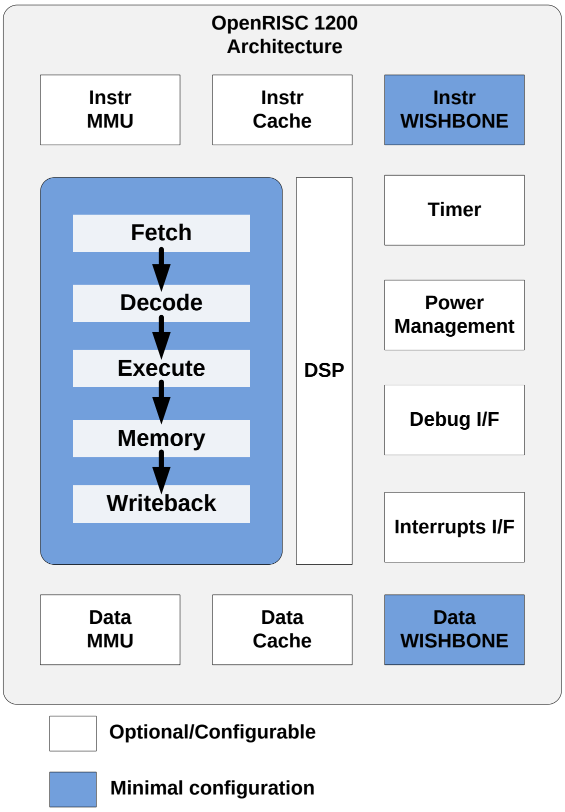


Figure 5.1 OpenRISC 1200 Architecture

We use the default configuration for the instruction and data cache, which are 1-way associate, direct-mapped 8 KB data cache and 1-way direct-mapped 8 KB instruction cache. Since the invalidation of entire cache is not supported in OR1200 architecture, we use a start-up assemble code to invalidate each cache line in the instruction and data cache. The following piece of code shows how to enable the instruction cache. The data cache can be enabled in a similar way.

```
/* Disable Instruction Cache */
    l.mfspr   r13, r0, SPR_SR
    l.addi    r11, r0, -1
    l.xori    r11, r11, SPR_SR_ICE
    l.and     r11, r13, r11
    l.mtspr   r0, r11, SPR_SR

/* Invalidate Instruction Cache */
    l.addi    r13, r0, 0
    l.addi    r11, r0, IC_SIZE
    l.mtspr   r0, r13, SPR_ICBIR
    l.sfne    r13, r11
    l.bf      -2
    l.addi    r13, r13, CL_SIZE

/* Enable Instruction Cache */
    l.mfspr   r13, r0, SPR_SR
    l.ori     r13, r13, SPR_SR_ICE
    l.mtspr   r0, r13, SPR_SR
```

5.1.2 OR1200 Wrapper

The SystemVerilog wrapper provides three interfaces for the testbench. They are Instruction Interface, Data Interface and CPU Status Interface. The instruction interface is connected to instruction WISHBONE interface of the OpenRISC 1200 core and used to fetch instructions from external instruction memory. The data interface is connected to data WISHBONE interface and responsible for all data READ/WRITE cycles. The CPU Status Interface is designed to access the internal signals of the OpenRISC 1200 core. Currently, the CPU Status Interface can access the internal signals including,

- The Program Counter (PC).
- The address and instruction executed in each stage.
- All General Purpose Registers (GPRs).
- Some important Special Purpose Registers (SPRs).

The three interfaces provide a general connection between the OR1200 core and the verification environment. All interactions between the DUV and the verification environment have to be through these interfaces. The structure of the OR1200 wrapper is shown in Figure 5.2.

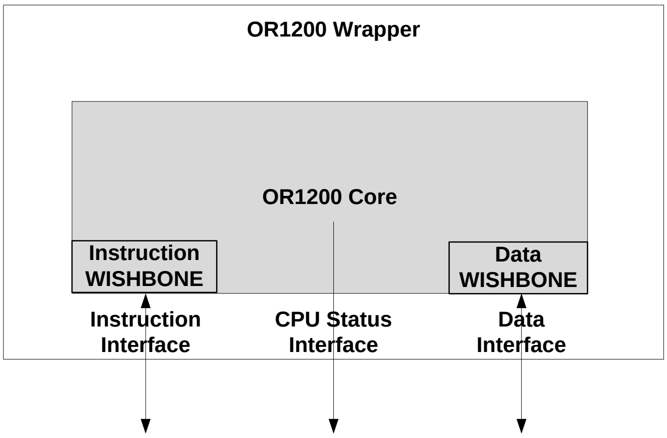


Figure 5.2 OpenRISC 1200 wrapper

5.2 Reference Model

5.2.1 Or1ksim Library

In order to use the or1ksim simulator as a reference model to check the behavior of the OR1200 core during the simulation, a C++ wrapper is implemented as an interface between the or1ksim simulator and the verification environment. Some modifications are also needed in the functions of the or1ksim library.

Modification of or1ksim Library

In the verification environment, the or1ksim simulator is initialized at the beginning of the simulation. Then, the simulator executes one instruction each time and saves the state in the testbench. After the DUV finishes the execution, the comparison will be made between the states of the reference model and the DUV. After the comparison, the simulator continues executing the next instruction. To accomplish this, some modifications need to be made in the or1ksim library.

First, we need an initialization function to initialize the or1ksim simulator. The information passed to the simulator includes the name of the configuration file, the executable image, the pointer to the calling class and three up-call functions. The configuration file contains the data to configure the simulator. The image file contains the executable program in the simulator. However, since the simulator needs to fetch instructions from the testbench, we will pass an empty image file into the simulator. The three up-call functions are intended for the instruction fetch, data exchange and the CPU status store.

Also, an execution function is necessary for the testbench to control the simulator to execute one instruction at a time. By the end of the execution, the simulator will call the status up-call function to save the current status in a structure of the testbench. The declarations of the functions are listed below.

-
  ```
  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 *cpuStatus)
                    );
  ```

  The initialization function is modified to facilitate the read/write data transfer and the access to the internal status of the or1ksim simulator.

- void or1ksim_execute();

  The function is used to make the simulator execute only one instruction. By the end of the execution, the simulator will write the CPU status to the external structure through the up-call function.

Three up-call functions, upr, upw and upCPUStatus as interfaces connect between the or1ksim simulator and the verification environment. They are implemented in the C++ wrapper.

5.2.2 Or1ksim Wrapper

To create a general interface between the or1ksim simulator and the verification environment, a C++ wrapper is implemented. The wrapper defines a global class ISS and a structure to save the status of the simulator, as described in Figure 5.3.

```
typedef struct {
    unsigned int gpr[32];
    unsigned int spr[16];
    unsigned int pc;
    unsigned int insn;
} cpu_state;

class ISS {
    public:
        ISS();
        ~ISS();
        unsigned long int readUpCall();
        void writeUpCall();
        void writeCPUStatus();
        void execute();
    private:
        cpu_state *cpuStatus;
        void copyCPUStatus;
        unsigned long int handle_endian();
} iss;
```

Figure 5.3 Class ISS and Structure cpu_state

In the constructor of the class, the or1ksim simulator is initialized by calling the or1ksim_init(). The name of the configuration file and image file, the pointer to the global class and three member functions are passed into the or1ksim simulator. Another member function in the class, execute() is used to run the simulator to execute instruction by instruction.

-
  ```
  ISS() {
        or1ksim_init("sim.cfg",
                     "empty_elf",
                     this,
                     readUpCall,
                     writeUpCall,
                     writeCPUStatus);
  }
  ```

-
  ```
  ISS::execute() {
        or1ksim_execute();
  }
  ```

The member functions, readUpCall(), writeUpCall() and writeCPUStatus() are designed to provide the implementation of the upcall functions for the or1ksim library. Each function calls a respective function in SystemVerilog to read/write data in the testbench. To call SystemVerilog functions from C/C++, the Direct Programming Interface (DPI) is necessary to be implemented. The following piece of code below shows how the up-call functions are implemented.

-
  ```
  unsigned long int ISS::readUpCall (void *instancePtr,
                                     unsigned long int addr,
                                     unsigned long int mask)
  {
      iss->rdata = (unsigned long int) sv_readData(
                                          (unsigned int) addr,
      (unsigned int) iss->handle_endian ((unsigned int) mask));
      return (unsigned long int) iss->handle_endian(iss->rdata);
  }
  ```

-
  ```
  void ISS::writeUpCall (void *instancePtr, unsigned long int addr,
                         unsigned long int mask, unsigned long int wdata)
  {
      sv_writeData((unsigned int) addr,
      (unsigned int) iss->handle_endian ((unsigned int) mask),
      (unsigned int) iss->handle_endian ((unsigned int) wdata));
  }
  ```

-
  ```
  void ISS::writeCPUStatus(void *instancePtr, void *cpuStatus)
  {
      cpu_state_up *cpuStatusPtr = (cpu_state_up *) cpuStatus;
      ISS * issPtr = (ISS *) instancePtr;
      issPtr->copyCPUStatus(cpuStatusPtr);
  }
  ```

-
  ```
  void ISS::copyCPUStatus(cpu_state_up *cpuStatusPtr)
  {
      ……
      this->sv_writeStatus(statusData);
  }
  ```

As shown in the code, the functions, sv_readData(), sv_writeData() and sv_writeStatus() are exported from the testbench. They are declared in SystemVerilog. sv_readData()/sv_writeData() reads/writes data in the memory and the mask value indicates the valid bytes in the data. The function, sv_writeStatus() saves the status of the reference model in the testbench. The declarations for these functions are described below.

-
  ```
  function longint  sv_readData (int unsigned addr, int unsigned mask);
  ```

-
  ```
  function void     sv_writeData (int unsigned addr,
                              int unsigned mask,    int unsigned wdata);
  ```

-
  ```
  function void     sv_writeStatus (int usngined status[]);
  ```

5.3 DPI Interfaces

The connection between the or1ksim simulator and the verification environment are through the up-call functions implemented by the “DPI” functions. To import C++ functions into the SystemVerilog testbench, we use the import “DPI” declaration. To export SystemVerilog functions to the simulator, we use the export “DPI” functions.

The two functions imported from the or1ksim wrapper are:

- import “DPI” function void iss_init();
- import “DPI” function void iss_execute();

The functions exported to the or1ksim wrapper are:

- export “DPI” function sv_readData;
- export “DPI” function sv_writeData;
- export “DPI” function sv_writeStatus;

Meanwhile, we need also declare these functions in the or1ksim wrapper as external functions like,

- extern “C” {

```
        void              iss_init ();
        void              iss_execute ();
        unsigned long int sv_readData ( unsigned int addr,
                                        unsigned int mask);
        void              writeData   ( unsigned int addr,
                                        unsigned int mask,
                                        unsigned int wdata);
        void              writeStatus ( unsigned int status[]);
    }
```

5.4 Memory System

The memory system used in the verification environment consists of a “Read Only Memory” (ROM) and a “Random Access Memory” (RAM).

5.4.1 ROM

A single port 32-bit ROM with synchronous Read is implemented. The ROM is connected to the instruction WIHBONE interface of the OR1200 wrapper through a SLAVE WISHBONE interface. It is initialized with instructions to be executed. The

OR 1200 core fetches instructions from the ROM one by one and executes them based on the system clock.

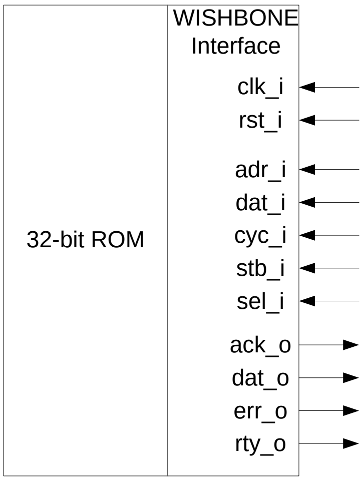


Figure 5.4 32-bit Read Only Memory (ROM)

In the Read cycle of the ROM, the Master side of the WISHBONE interface sends the READ request by asserting the strobe (stb_i) and the cycle (cyc_i) signals. The Slave WISHBONE interface responds the synchronous acknowledgement (ack_o) signal, which indicates a valid data output. The output signals, error (err_o) and retry (rty_o) signals terminate the cycle with an incorrect data output. Figure 5.5 shows the waveform of a single READ operation of ROM.

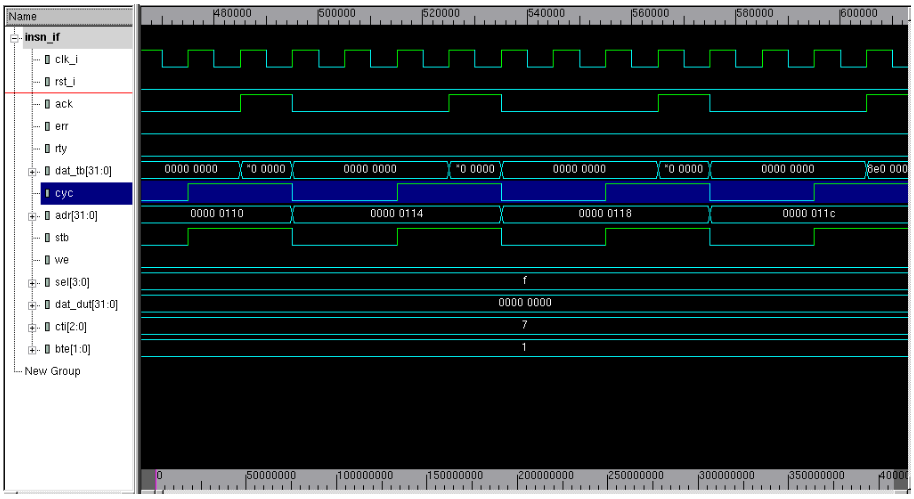


Figure 5.5 Waveform for ROM Read Cycle

5.4.2 RAM

A single port 32-bit RAM with synchronous Read/Write is implemented. It is connected to the data WISHBONE interface of the OR1200 wrapper through a Slave WISHONE interface. The RAM is used to store all the data exchange during the simulation.

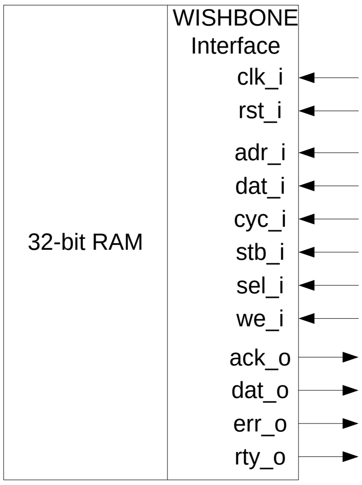


Figure 5.6 32-bit Random Access Memory (RAM)

In the READ/WRITE cycle, the Master WISHBONE interface initiates the operation by asserting the strobe (stb_i) and cycle (cyc_i) signals. Then, the RAM interface reads in the address (adr_i), select (sel_i), input data (dat_i) and write enable (we_i) signals. The select (sel_i) signal indicates the valid data bytes in the input data, and the write enable (we_i) signal shows the current cycle is READ or WRITE. The synchronous acknowledgement (ack_o) signal terminates the transfer cycle.

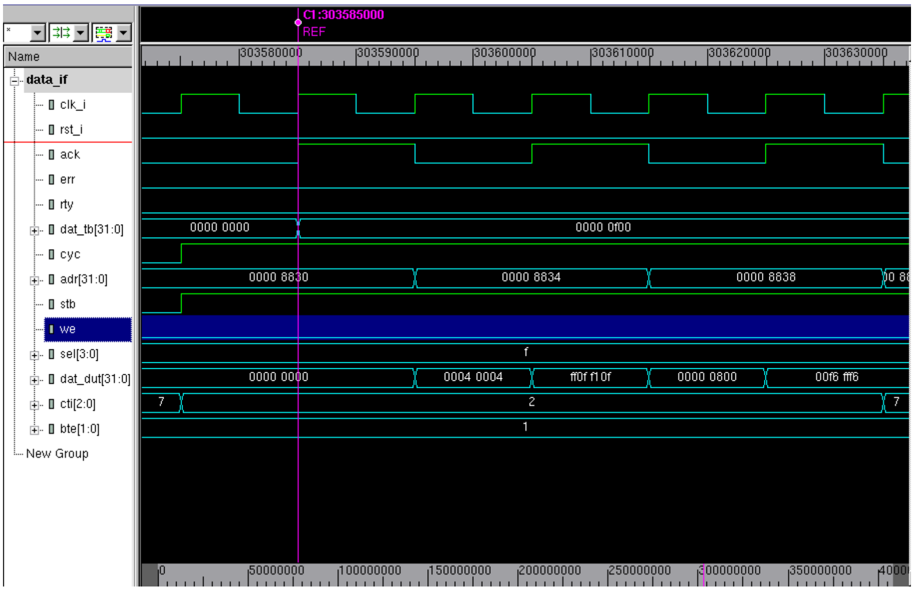


Figure 5.7 Waveform of RAM Read Cycle

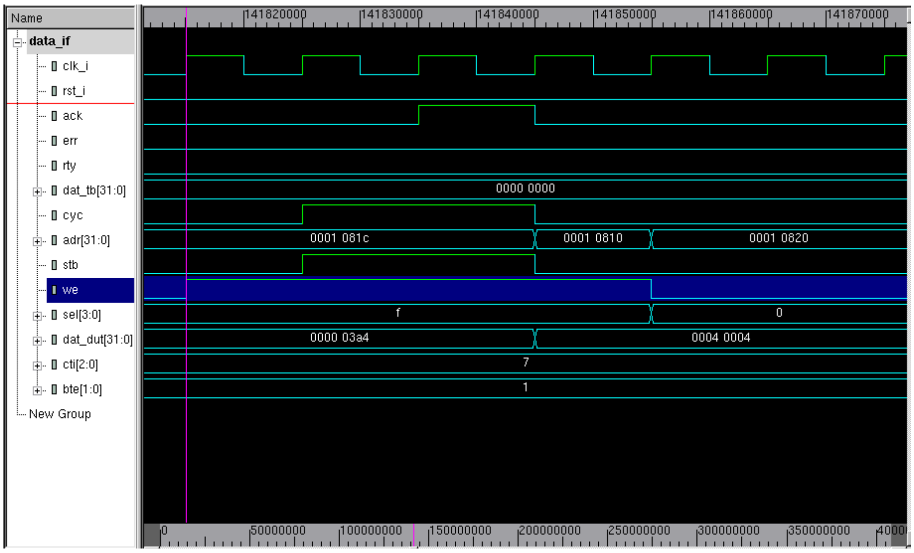


Figure 5.8 Waveform of RAM Write Cycle

5.5 Testbench

The basic structure of the OR1200 testbench consists of the Driver, the Design under Verification (DUV), the Golden Reference Model, the Monitor and the Memory System (ROM & RAM), as shown in Figure 5.9.

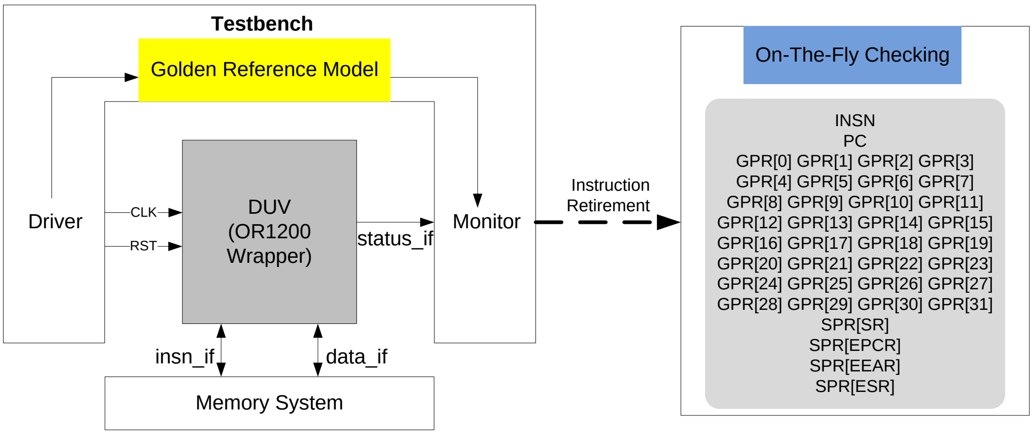
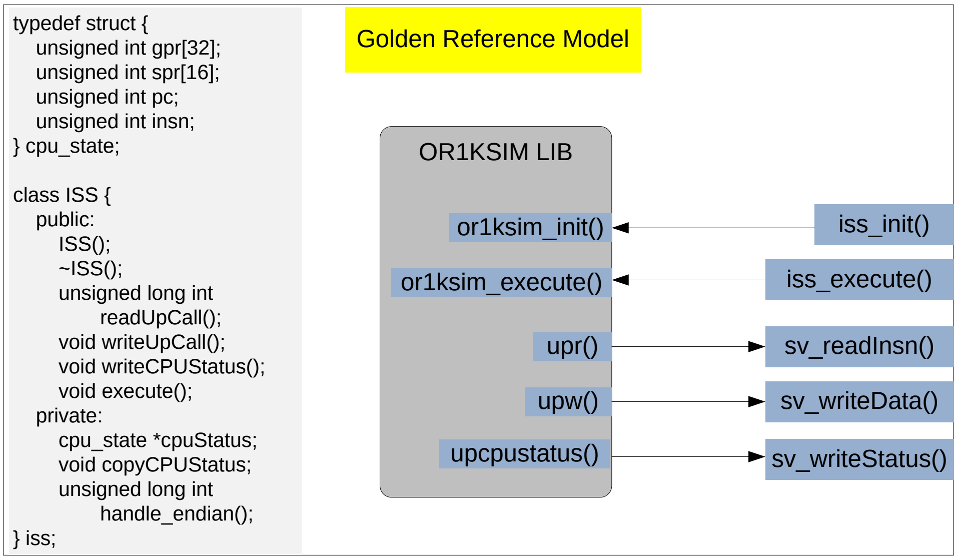

Testbench
Golden Reference Model
Driver
CLK
RST
DUV
(OR1200
Wrapper)
status_if
Monitor
insn_if
data_if
Memory System
Instruction
Retirement
On-The-Fly Checking
INSN
PC
GPR[0] GPR[1] GPR[2] GPR[3]
GPR[4] GPR[5] GPR[6] GPR[7]
GPR[8] GPR[9] GPR[10] GPR[11]
GPR[12] GPR[13] GPR[14] GPR[15]
GPR[16] GPR[17] GPR[18] GPR[19]
GPR[20] GPR[21] GPR[22] GPR[23]
GPR[24] GPR[25] GPR[26] GPR[27]
GPR[28] GPR[29] GPR[30] GPR[31]
SPR[SR]
SPR[EPCR]
SPR[EEAR]
SPR[ESR]

Figure 5.9 Testbench structure

During the simulation, the driver drives the clock (CLK) and reset (RST) signals into both the reference model and the DUV. The clock (CLK) signal provides the system clock for the whole environment. The reset (RST) signal is used to reset the whole environment when needed. When the reset (RST) signal goes low, the DUV starts to fetch instructions through instruction interface from the instruction memory. If data exchange needed, the DUV reads/writes data through data interface. At the retirement of each instruction, the DUV will trigger the On-The-Flying checking in the testbench. The state of the OR1200 core passed through the status interface will be compared with the state of the reference model.

The or1ksim simulator, as the reference model is initialized by the testbench at the beginning of the simulation. While the or1ksim simulator is running, it exchanges data with the external environment by using the up-call functions implemented in the test bench. Therefore, the or1ksim simulator can fetch instructions and read/write data. The simulator is controlled by the execution function in the library. When the execution function is called, the or1ksim simulator executes one instruction and save the state of the simulator by calling one of the up-call functions. The status of the simulator is stored in the testbench. When the instruction retires in the DUV, the state of the OR1200 core will be compared with the status stored in the testbench.

As shown in Figure 5.10, the functions, iss_init() and iss_execute() call the functions, or1ksim_init() and or1ksim_execute() in the or1ksim library respectively. The three upcall functions, upr(), upw() and upcpuStatus() are implemented by the SystemVerilog functions, sv_readData(), sv_writeData() and sv_writeStatus() respectively.



```
typedef struct {
    unsigned int gpr[32];
    unsigned int spr[16];
    unsigned int pc;
    unsigned int insn;
} cpu_state;

class ISS {
    public:
        ISS();
        ~ISS();
        unsigned long int
               readUpCall();
        void writeUpCall();
        void writeCPUStatus();
        void execute();
    private:
        cpu_state *cpuStatus;
        void copyCPUStatus;
        unsigned long int
               handle_endian();
} iss;
```

Figure 5.10 Golden Reference Model

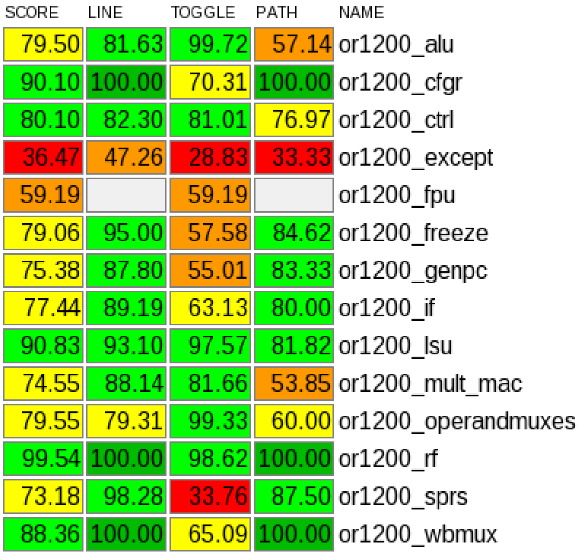
When the instruction retires in the OR1200 core, the On-The-Flying is triggered. The monitor checks the CPU status between the DUV and the Reference model including all General Purpose Registers (GPRs), important Special Purpose Registers (SPRs), the instruction the currently retires and the current Program Counter (PC). The result will be displayed if any difference comes up.

The Instruction memory is initialized with instructions from the memory initialization file generated by the OR1200 tool chain. The DUV fetches instructions from the Instruction memory through the instruction interface. During the simulation, the DUV interacts with data memory through the data interface. All data will be read or written in the data memory.

6 RESULTS AND DISCUSSION

6.1 Simulation Results

Three C programs were compiled by the OR1200 compiler and run as test cases in the testbench, as shown in Appendix. All these programs contain 8-bit, 16-bit and 32-bit data transfer, and all test cases are passed. We collected some metrics of code coverage that may be used for the functional verification and power analysis by GLOBALFOUNDRIES in the future. The metrics includes line coverage, toggle coverage and path coverage. As increasing the size of the test cases, the coverage is getting higher. Figure 6.1 shows code coverage in the CPU of the OR1200 core for one test case.



SCORE	LINE	TOGGLE	PATH	NAME
79.50	81.63	99.72	57.14	or1200_alu
90.10	100.00	70.31	100.00	or1200_cfgr
80.10	82.30	81.01	76.97	or1200_ctrl
36.47	47.26	28.83	33.33	or1200_except
59.19		59.19		or1200_fpu
79.06	95.00	57.58	84.62	or1200_freeze
75.38	87.80	55.01	83.33	or1200_genpc
77.44	89.19	63.13	80.00	or1200_if
90.83	93.10	97.57	81.82	or1200_lsu
74.55	88.14	81.66	53.85	or1200_mult_mac
79.55	79.31	99.33	60.00	or1200_operandmuxes
99.54	100.00	98.62	100.00	or1200_rf
73.18	98.28	33.76	87.50	or1200_sprs
88.36	100.00	65.09	100.00	or1200_wbmux

Figure 6.1 Code Coverage report of OR1200 CPU

As we can see in the report, there are some holes in code coverage. The reasons that these coverage holes exist include,

- Disabled modules in the OR1200 core, like Debug Unit (DU), Programmable Interrupt Controller (PIC). We can enable them later if needs.

- Test programs are compiled by the OR1200 compiler. Therefore, the instructions generated heavily depend on the implementation of the compiler. Some instructions implemented in the OR1000 architecture may not be generated or often generated. We can use assemble code to cover the holes later.

6.2 Examples for Memory/Cache Usage

Catthoor presented some simple examples of loop nesting and operation ordering [13] to illustrate the impact on the memory accesses. In this section, we collect code coverage from six examples. In each example, two pieces of code with same functionality have different usage of the cache/memory. This difference can be easily detected by code coverage. Moreover, if we enable a feature in URG, which shows the number of hits for toggled signals, we can also obtain the accurate number of cache/memory accesses.

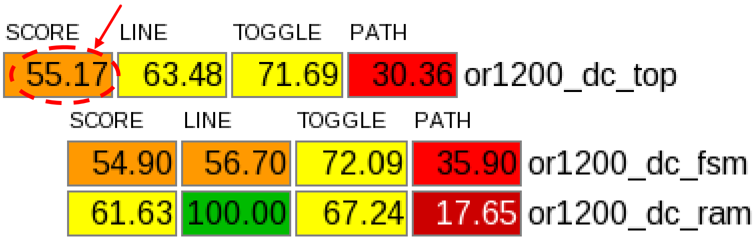
6.2.1 Example 1

```
1.1:
    For i := 1 TO N DO
        B[i] := f(A[i]);
    For i := 1 TO N DO
        C[i] := g(B[i]);
```

```
1.2:
    For i := 1 TO N DO
    Begin
        B[i] := f(A[i]);
        C[i] := g(B[i]);
    End
```

In Example 1, array B[] is assumed to be too large to be stored in (on-chip) registers. Therefore, in Example 1.1, array B[] is written into memory directly in the first loop, and then read from memory in the second one. Therefore, there are totally 4N read/write cache accesses. If we rewrite the code like what is shown in Example 1.2, the compiler will optimize the cache access by using registers to keep the intermediate values instead of using memory. Thus, array B[] needs not to be read from memory. The total number of cache accesses is reduced to 2N.

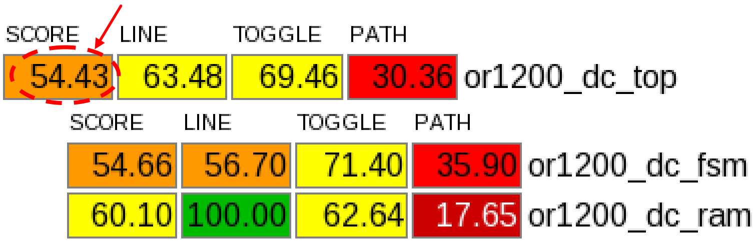
In the coverage reports of both cases, we can clearly see that, there is difference in code coverage for cache, as cycled in Figure 6.2 and 6.3.



SCORE	LINE	TOGGLE	PATH	
55.17	63.48	71.69	30.36	or1200_dc_top

SCORE	LINE	TOGGLE	PATH	
54.90	56.70	72.09	35.90	or1200_dc_fsm
61.63	100.00	67.24	17.65	or1200_dc_ram

Figure 6.2 Cache Coverage Report of Example 1.1

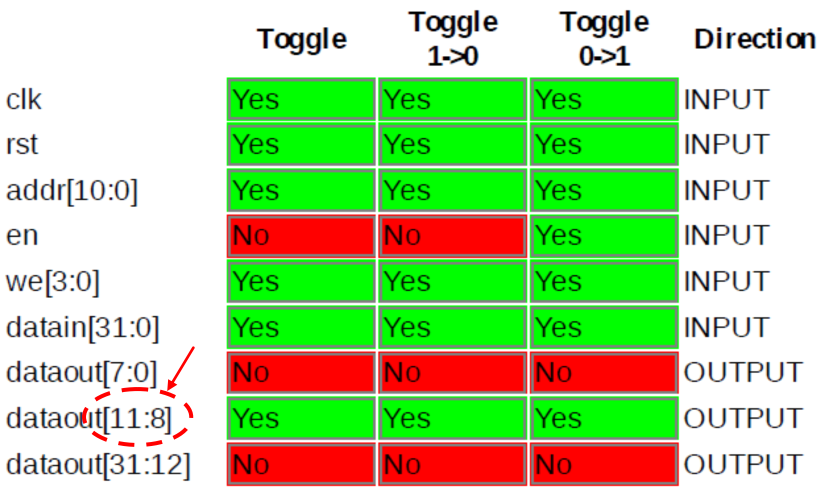


SCORE	LINE	TOGGLE	PATH	
54.43	63.48	69.46	30.36	or1200_dc_top

SCORE	LINE	TOGGLE	PATH	
54.66	56.70	71.40	35.90	or1200_dc_fsm
60.10	100.00	62.64	17.65	or1200_dc_ram

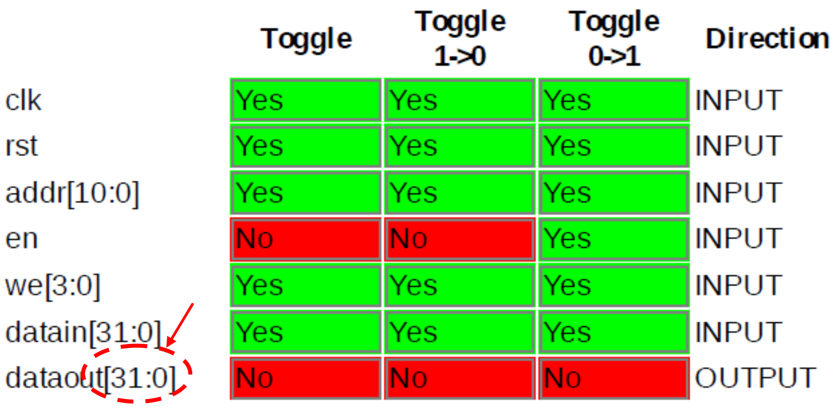
Figure 6.3 Cache Coverage Report of Example 1.2

One reason that results in this difference is that, the bits [11:8] of data in array B[] are set to one, while those bits in array A[] and array C[] are all zero. Then, if array B[] is ever read/written in cache, the bits [11:8] in the output data of cache will be toggled. As evidence in the coverage report, shown in Figure 6.4 and 6.5, the bits [11:8] of dataout signal in cache are toggle in Example 1.1, while they are not toggled in Example 1.2. Therefore, we can conclude that, in Example 1.1, the data of array B[] is read from cache and memory, while the intermediate values of array B[] are only stored in registers and never access the cache.



	Toggle	Toggle 1->0	Toggle 0->1	Direction
clk	Yes	Yes	Yes	INPUT
rst	Yes	Yes	Yes	INPUT
addr[10:0]	Yes	Yes	Yes	INPUT
en	No	No	Yes	INPUT
we[3:0]	Yes	Yes	Yes	INPUT
datain[31:0]	Yes	Yes	Yes	INPUT
dataout[7:0]	No	No	No	OUTPUT
dataout[11:8]	Yes	Yes	Yes	OUTPUT
dataout[31:12]	No	No	No	OUTPUT

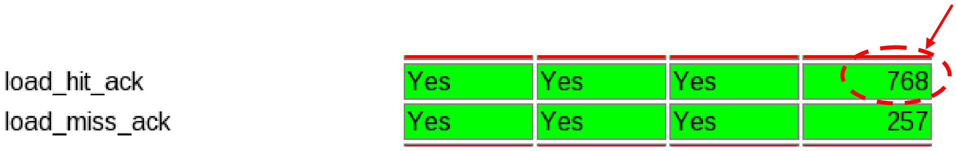
Figure 6.4 Cache RAM Coverage Report of Example 1.1



	Toggle	Toggle 1->0	Toggle 0->1	Direction
clk	Yes	Yes	Yes	INPUT
rst	Yes	Yes	Yes	INPUT
addr[10:0]	Yes	Yes	Yes	INPUT
en	No	No	Yes	INPUT
we[3:0]	Yes	Yes	Yes	INPUT
datain[31:0]	Yes	Yes	Yes	INPUT
dataout[31:0]	No	No	No	OUTPUT

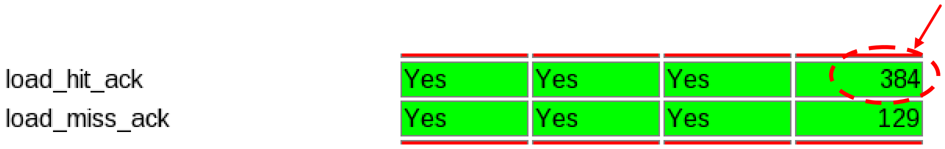
Figure 6.5 Cache RAM Coverage Report of Example 1.2

On the other hand, we can also learn the cache accesses quantitatively through code coverage. If we enable a feature in VCS, which shows the number of hits for toggled signals, it is obvious to see the different usage of the cache. In Example 1.1, as Figure 6.6 shows, the load_hit_ack signal in the cache RTL that goes high when it is a cache hit on load operation, has been hit 768 times. In Example 1.2, as Figure 6.7 shows, the number of hits of the load_hit_ack signal is 384.



load_hit_ack	Yes	Yes	Yes	768
load_miss_ack	Yes	Yes	Yes	257

Figure 6.6 Toggle coverage of load_hit_ack in Example 1.1



load_hit_ack	Yes	Yes	Yes	384
load_miss_ack	Yes	Yes	Yes	129

Figure 6.7 Toggle coverage of load_hit_ack in Example 1.2

In the actual programs, we set three integer arrays, A[], B[] and C[]. The size of each array is 512 words (1 word = 4 bytes). The data cache has the size of 8 KB with 16-byte cache line. Thus, each cache line can save four words. In Example 1.1, both array A[] and B[] have been read from the cache. The first word in each cache line will be a miss, but the following three word read will be hits. Therefore, the total number of cache hits in Example 1.1 is,

$$\frac{4 \text{ bytes} \times 512}{16 \text{ bytes}} \times 3 \text{ hits} \times 2 = 768 \text{ hits}$$

In Example 1.2, since only array A[] is read from cache, the total number of cache hit is,

$$\frac{4 \text{ bytes} \times 512}{16 \text{ bytes}} \times 3 \text{ hits} = 384 \text{ hits}$$

Both results agree with code coverage in Figure 6.6 and 6.7.

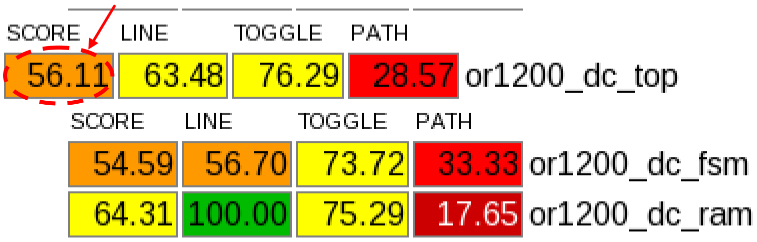
6.2.2 Example 2

```
2.1:
    For j := 1 TO M DO
        For i := 1 TO N DO
            A[i][j] := f(A[i][j]);
```

```
2.2:
    For i := 1 TO N DO
        For j := 1 TO M DO
            A[i][j] := f(A[i][j]);
```

In Example 2.1, the first piece of code has nested loops that access data in memory in non-sequential order. Since the size of the array is too large to be all kept in cache, the old data will be evicted by the new data. Therefore, there is no cache hit in Example 2.1. When we re-arrange the code by exchanging the nested loops, the access to memory becomes sequential. The data in the same cache line will be accessed in order. Since data in the same cache line is always loaded together, even the first data access is a cache miss, the following data accesses will be cache hit. Therefore, there will be cache hit in Example 2.2.

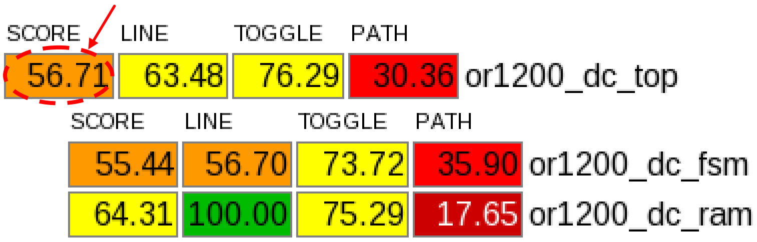
In the coverage report of both cases, there is an obvious difference in code coverage for cache, as shown in Figure 6.8 and 6.9.



SCORE	LINE	TOGGLE	PATH	
56.11	63.48	76.29	28.57	or1200_dc_top

SCORE	LINE	TOGGLE	PATH	
54.59	56.70	73.72	33.33	or1200_dc_fsm
64.31	100.00	75.29	17.65	or1200_dc_ram

Figure 6.8 Cache Coverage report of Example 2.1

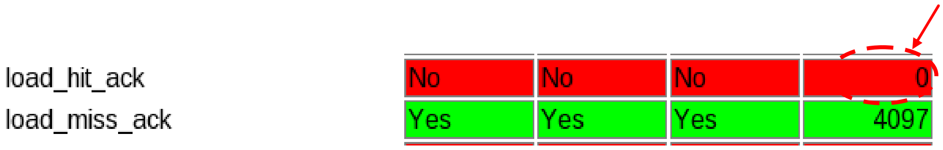


SCORE	LINE	TOGGLE	PATH	
56.71	63.48	76.29	30.36	or1200_dc_top

SCORE	LINE	TOGGLE	PATH	
55.44	56.70	73.72	35.90	or1200_dc_fsm
64.31	100.00	75.29	17.65	or1200_dc_ram

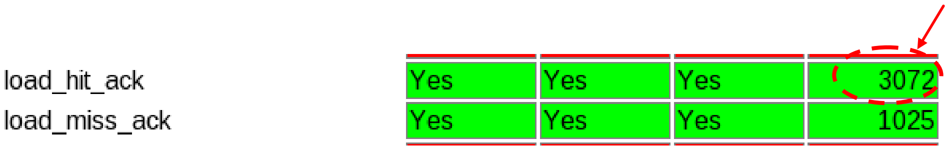
Figure 6.9 Cache Coverage report of Example 2.2

In OR1200 architecture, the state of each cache line is recorded by a Finite State Machine (dc_fsm) in the cache, since there is no cache hit in Example 2.1, the signal in dc_fsm, load_hit_ack is never toggled, while this signal is toggled in Example 2.2 because of the hit in cache. Figure 6.10 and 6.11 from the coverage report of dc_fsm shows the difference in code coverage of the load_hit_ack signal.



load_hit_ack	No	No	No	0
load_miss_ack	Yes	Yes	Yes	4097

Figure 6.10 DC_FSM Coverage Report of Example 2.1



load_hit_ack	Yes	Yes	Yes	3072
load_miss_ack	Yes	Yes	Yes	1025

Figure 6.11 DC_FSM Coverage Report of Example 2.2

For the quantitative analysis, the array is defined as, int arrayA[1024][4] in the program. Since the size of the data cache is 8 KB, arrayA[n][m] and arrayA[n+512][m] will be mapped to the same address in the cache. If we access the data cache in the sequential order, the access to the first word in the cache line will be a miss, but the following three words will all be hit. Therefore, in Example 2.2, the number of cache hits is,

$$\frac{4\text{ bytes } \times 1024 \ \times 4}{16\text{ bytes}} * 3\text{ hits} = \ 3072\text{ hits}$$

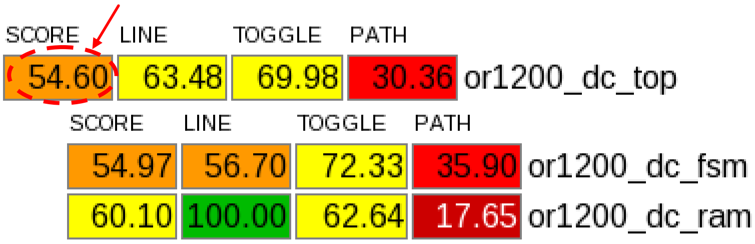
which is consistent with the result in code coverage in Figure 6.11.

6.2.3 Example 3

```
3.1:                                      3.2:
    For i := 1 TO N DO                        For i := 1 TO N DO
        C[i] := f(A[i]);                          D[i] := g(B[i]);
    For i := 1 TO N DO                        For i := 1 TO N DO
        D[i] := g(B[i]);                          C[i] := f(A[i]);
    For i := 1 TO N DO                        For i := 1 TO N DO
        A[i] := E;                                A[i] := E;
```

In Example 3, if we have limited cache size and array B[] is not needed afterward, we can rearrange the order of the code. Then, array A[] will not be evicted by array B[]. The following write to array A[] will be written to both cache and memory, since the write-through strategy is implemented.

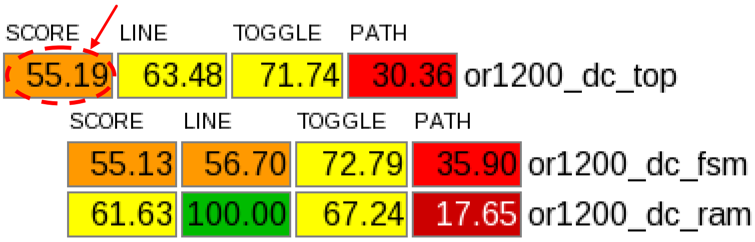
In the coverage report, the difference is obviously shown in Figure 6.12 and 6.13.



SCORE	LINE	TOGGLE	PATH	
54.60	63.48	69.98	30.36	or1200_dc_top

SCORE	LINE	TOGGLE	PATH	
54.97	56.70	72.33	35.90	or1200_dc_fsm
60.10	100.00	62.64	17.65	or1200_dc_ram

Figure 6.12 Cache Coverage of Example 3.1

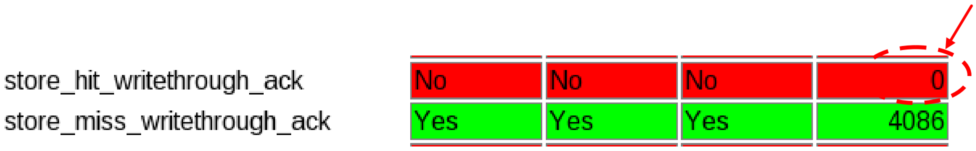


SCORE	LINE	TOGGLE	PATH	
55.19	63.48	71.74	30.36	or1200_dc_top

SCORE	LINE	TOGGLE	PATH	
55.13	56.70	72.79	35.90	or1200_dc_fsm
61.63	100.00	67.24	17.65	or1200_dc_ram

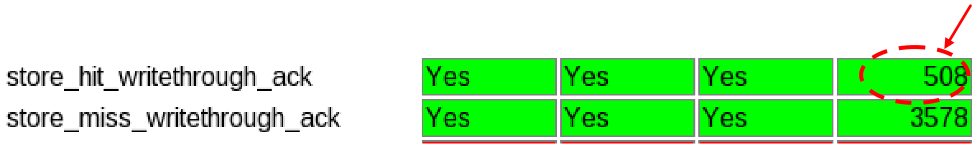
Figure 6.13 Cache Coverage of Example 3.2

The difference in the coverage report is due to the fact that, the store_hit_writethrough_ack signal in dc_fsm is not toggled in Example 3.1, which it is toggled in Example 3.2. Figure 6.14 and 6.15 describe the difference in dc_fsm, which indicates that array A[] is stored in cache in Example 3.2.



store_hit_writethrough_ack	No	No	No	0
store_miss_writethrough_ack	Yes	Yes	Yes	4086

Figure 6.14 DC_FSM Coverage Report of Example 3.1



store_hit_writethrough_ack	Yes	Yes	Yes	508
store_miss_writethrough_ack	Yes	Yes	Yes	3578

Figure 6.15 DC_FSM Coverage Report of Example 3.2

In the programs, we set two integer array A[] and array B[] with the size of 512. Moreover, the elements from 4 to 511 in array A[] are mapped to the same locations with the elements from 0 to 507 array B[]. Thus, in Example 3.2, if we store data in array A[], when array A[] is in the cache, all write operations should result in the cache hit. The total number of the store hits is,

$$\frac{4\ \text{bytes} \times 508}{16\ \text{bytes}} * 4\ \text{hits} = \ 508\ \text{hits}$$

The result is consistent with the observation from code coverage, as shown in Figure 6.15.

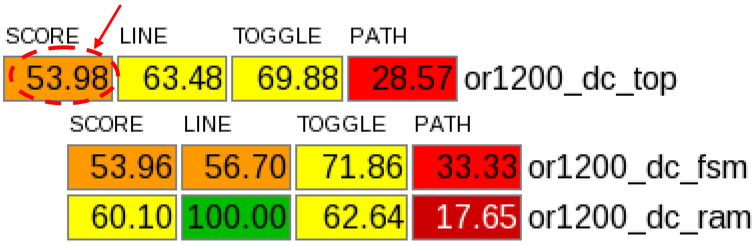
6.2.4 Example 4

```
4.1:
    For i := 1 TO N DO
        C[i] := f(A[i]);
    For i := 1 TO N DO
        D[i] := g(B[i]);
    For i := 1 TO N DO
        C[i] := h(A[i]);
```

```
4.2:
    For i := 1 TO N DO
        D[i] := g(B[i]);
    For i := 1 TO N DO
        C[i] := f(A[i]);
    For i := 1 TO N DO
        C[i] := h(A[i]);
```

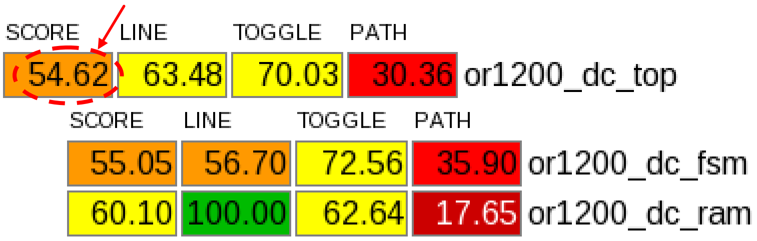
Similar to Example 3, array A[] and array B[] are mapped to the same location in cache. If array B[] is read in the middle of the operation of array A[], array A[] will be evicted from cache, which results in the cache miss when reading array A[] again. If we rearrange the order, array A[] will kept in cache, which can reduce the cost of memory access for array A[].

Figure 6.16 and 6.17 shows the difference of code coverage for both cases. And Figure 6.18 and 6.19 describe the difference in the load_hit_ack signal in dc_fsm coverage report.



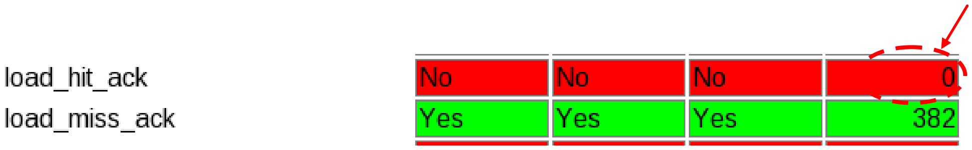
SCORE	LINE	TOGGLE	PATH	
53.98	63.48	69.88	28.57	or1200_dc_top
SCORE	LINE	TOGGLE	PATH	
53.96	56.70	71.86	33.33	or1200_dc_fsm
60.10	100.00	62.64	17.65	or1200_dc_ram

Figure 6.16 Cache Coverage Report of Example 4.1



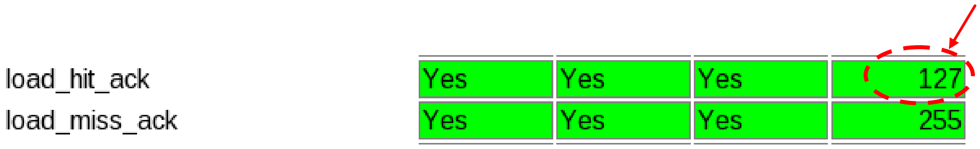
SCORE	LINE	TOGGLE	PATH	
54.62	63.48	70.03	30.36	or1200_dc_top
SCORE	LINE	TOGGLE	PATH	
55.05	56.70	72.56	35.90	or1200_dc_fsm
60.10	100.00	62.64	17.65	or1200_dc_ram

Figure 6.17 Cache Coverage Report of Example 4.2



load_hit_ack	No	No	No	0
load_miss_ack	Yes	Yes	Yes	382

Figure 6.18 DC_FSM Coverage Report of Example 4.1



load_hit_ack	Yes	Yes	Yes	127
load_miss_ack	Yes	Yes	Yes	255

Figure 6.19 DC_FSM Coverage Report of Example 4.2

In the programs, we set two integer array A[] and array B[] with the size of 512. The elements from 4 to 511 of array A[] are mapped to the same locations in the cache with the elements from 0 to 507 of array B[]. To avoid any interference, we only read the first word in the cache line. Therefore, in Example 4.1, there is no hit in the cache. In Example 4.2, since array A[] is in the cache when there is read operations to array A[], all read operations will be hit in the cache. The total number of hits is,

$$\frac{4\text{ bytes} \times 508}{16\text{ bytes}} * 1\text{ hits} = \ 127\text{ hits}$$

This result is presented accurately by code coverage, as shown in Figure 6.19.

6.2.5 Example 5

5.1:

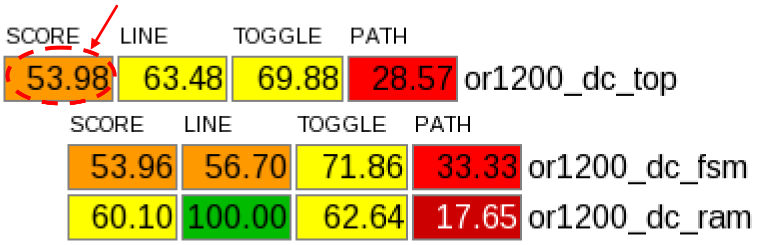
```
For i := 1 TO N DO
    C[i] := f(A[i]);
For i := 1 TO N DO
    C[i] := g(B[i]);
For i := 1 TO N DO
    A[i+1] := D;
```

5.2:

```
For i := 1 TO N DO
    C[i] := g(B[i]);
For i := 1 TO N DO
    C[i] := f(A[i]);
For i := 1 TO N DO
    A[i+1] := D;
```

In Example 5, if we have write operations on the data in the same cache line, we can reorder the code to improve the spatial locality. The cache always loads the whole cache line together. If we can maximize the use of the data in the same cache line before the data is evicted, we can have less memory accesses.

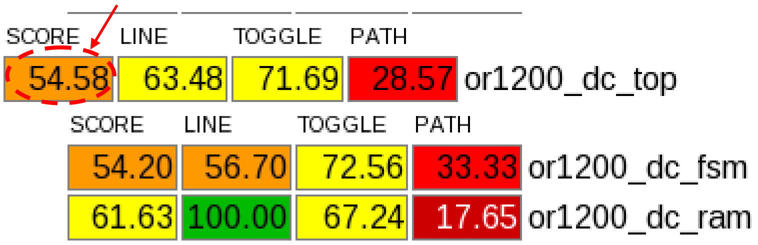
Figure 6.20 and 6.21 show the difference of code coverage report. And Figure 6.21 proves that, after reordering the code, there will be cache hit in Example 5.2.



SCORE	LINE	TOGGLE	PATH	
53.98	63.48	69.88	28.57	or1200_dc_top

SCORE	LINE	TOGGLE	PATH	
53.96	56.70	71.86	33.33	or1200_dc_fsm
60.10	100.00	62.64	17.65	or1200_dc_ram

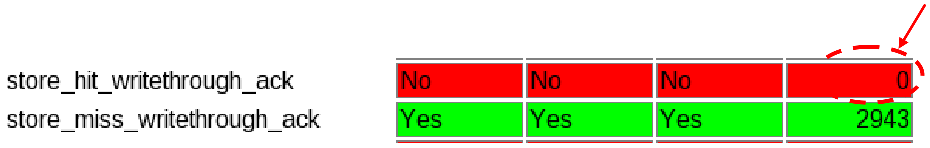
Figure 6.20 Cache Coverage Report of Example 5.1



SCORE	LINE	TOGGLE	PATH	
54.58	63.48	71.69	28.57	or1200_dc_top

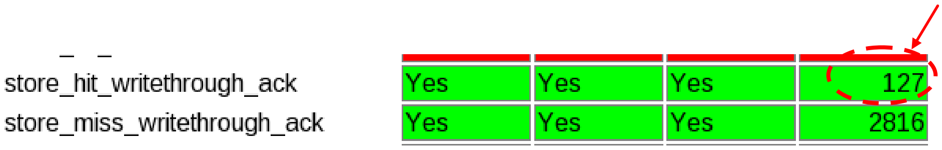
SCORE	LINE	TOGGLE	PATH	
54.20	56.70	72.56	33.33	or1200_dc_fsm
61.63	100.00	67.24	17.65	or1200_dc_ram

Figure 6.21 Cache Coverage Report of Example 5.2



store_hit_writethrough_ack	No	No	No	0
store_miss_writethrough_ack	Yes	Yes	Yes	2943

Figure 6.22 DC_FSM Coverage Report of Example 5.1



store_hit_writethrough_ack	Yes	Yes	Yes	127
store_miss_writethrough_ack	Yes	Yes	Yes	2816

Figure 6.23 DC_FSM Coverage Report of Example 5.2

In this example, we also use two integer array A[] and array B[] with the size of 512. The elements from 4 to 511 of array A[] are mapped to the same locations in the cache with the elements from 0 to 507 of array B[]. First, we read the first word in each cache line. Since the whole cache line will be loaded into cache, if we write to the second word in the cache line, it will be a cache hit. The number of store cache hits should be,

$$\frac{4\text{ bytes } \times 508}{16\text{ bytes}} * 1\text{ hits} = \ 127\text{ hits}$$

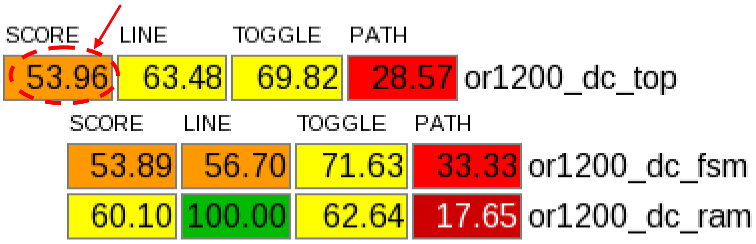
As Figure 6.23 shows, in Example 5.2, the store_hit_writethrough_ack signal has been hit 127 times, which is consistent with the result we calculated.

6.2.6 Example 6

```
6.1:
    For i := 1 TO N DO
        C[i] := f(A[i]);
    For i := 1 TO N DO
        C[i] := g(B[i]);
    For i := 1 TO N DO
        D[i] := h(A[i+1]);
```

```
6.2:
    For i := 1 TO N DO
        C[i] := g(B[i]);
    For i := 1 TO N DO
        C[i] := f(A[i]);
    For i := 1 TO N DO
        D[i] := h(A[i+1]);
```

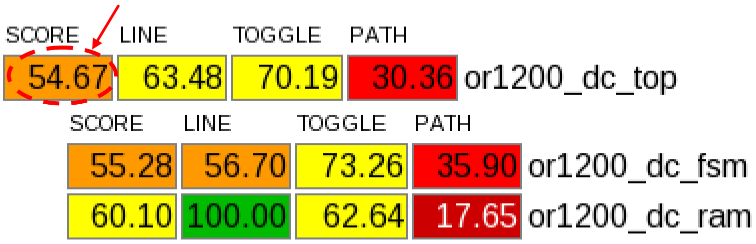
Similar to Example 5, if we have read accesses to the data in the same cache line, the use of the cache will also be different, as shown in Figure 6.24 and 6.25.



SCORE	LINE	TOGGLE	PATH	
53.96	63.48	69.82	28.57	or1200_dc_top

SCORE	LINE	TOGGLE	PATH	
53.89	56.70	71.63	33.33	or1200_dc_fsm
60.10	100.00	62.64	17.65	or1200_dc_ram

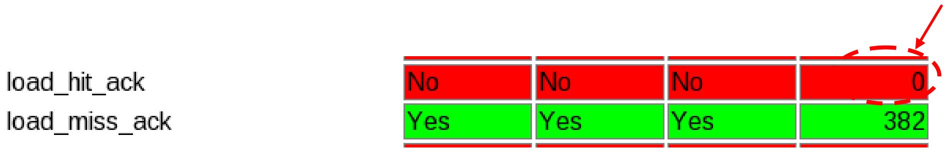
Figure 6.24 Cache Coverage Report of Example 6.1



SCORE	LINE	TOGGLE	PATH	
54.67	63.48	70.19	30.36	or1200_dc_top

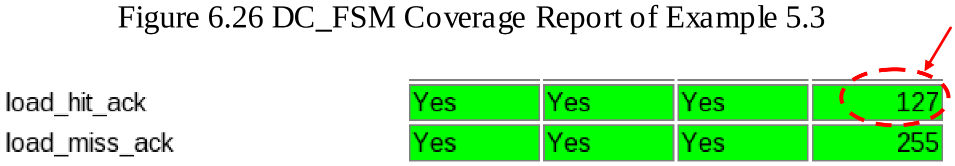
SCORE	LINE	TOGGLE	PATH	
55.28	56.70	73.26	35.90	or1200_dc_fsm
60.10	100.00	62.64	17.65	or1200_dc_ram

Figure 6.25 Cache Coverage Report of Example 6.2



load_hit_ack	No	No	No	0
load_miss_ack	Yes	Yes	Yes	382

Figure 6.26 DC_FSM Coverage Report of Example 5.3



load_hit_ack	Yes	Yes	Yes	127
load_miss_ack	Yes	Yes	Yes	255

Figure 6.27 DC_FSM Coverage Report of Example 5.4

In Example 5.2, if we read the second word in each cache line, when array A[] resides in the cache, each read operation will be a hit in the cache. The number of load hits is,

$$\frac{4\text{ bytes }\times 508}{16\text{ bytes}} * 1\text{ hits} = \ 127\text{ hits}$$

which is proven by code coverage, as shown in Figure 6.27.

6.3 Discussion

According to the results in this chapter, code coverage can be used to present the usage of cache/memory both qualitatively and quantitatively. Generally, higher code coverage in cache means more cache accesses. We can distinguish if there is any hit or miss in the cache by checking the specific signals in code coverage. Moreover, we can also learn more about the accesses to the cache by observing the number of hits of the toggled signals. The number of accesses to the cache can be calculated accurately from code coverage, as shown in the examples. The accesses to memory can also be obtained from the cache hits or cache misses.

Furthermore, according to the study in Chapter 4, based on the difference access cost for each level in memory hierarchy, we can estimate the power consumption of a test case by the usage of memory hierarchy. If a test case has more memory accesses, we can assume it costs more power during simulation.

Therefore, we conclude that, code coverage can be used as a cheap and efficient way to analyze the characteristics of programs, for example, the usage of memory hierarchy or power consumption. Moreover, code coverage can help us understand what the program uses in memory hierarchy, and hence help us estimate the power consumption involved with that usage. If we want to have a test suite that has a variety of power profiles, then we can use code coverage to estimate how good this test suite is with respect to power consumption.

7 CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

The thesis consists of two major parts. The first part is the implementation of the verification environment for the OR1200 core. And the second part is the research about how code coverage represents the usage of the cache and memory and how it can be used as a cheap way to identify the characteristics of the test cases.

IMPLEMENTATION

A simulation-based verification environment is implemented based on the testbench. The testbench is composed of a Driver, a Design under Verification (DUV), a Golden Reference Model, a Monitor and the memory system. The Driver generates the clock and reset signals to both the DUV and the Reference Model. The DUV is the OR1200 core with a SystemVerilog wrapper. The SystemVerilog wrapper provides three interfaces between the OR1200 core and the verification environment. The three interfaces include two WISHBONE interfaces for instruction fetch and data exchange and one interface to access the internal signals inside the OR1200 core. All interactions between the OR1200 core and the external environment are provided by these interfaces. The Golden Reference Model is implemented using the OpenRISC 1000 Architecture Simulator (or1ksim), also called Instruction Set Simulator (ISS) with a C/C++ wrapper. The OR1200 ISS is installed as a static library, which is manipulated by the functions exported from the or1ksim library. The up-call functions in the or1ksim library are implemented in SystemVerilog through the Direct Programming Interface (DPI). During the simulation, the On-The-Flying checking compares the current states between the DUV and the Reference Model. The checking includes all General Purpose Registers (GPRs), some important Special Purpose Registers, the Program Counter (PC) and the instruction executed. The test cases are compiled and converted to the memory initialization file by the OR1200 GNU tool chain.

The design was tested by three C/C++ test programs (in the source code file) and passed them all. Some code coverage metrics are collected including line coverage, toggle coverage and path coverage, which may be used for gate-level simulation in the future.

RESEARCH

In the second part of the thesis, we focus on how code coverage can be used as a proxy to qualify a test suite in terms of memory hierarchy usage or power consumption, so that the test suite contains a balanced set of tests that exercise all functionality of the design at all levels of the memory hierarchy. Then, we designed six examples to prove this, as described in Chapter 6. In each example, code coverage clearly presents the usage of the cache through some specific toggling signals. Moreover, when the feature in URG that counts the number of hits for toggled signals is enabled, code coverage can show the actual number of accesses to the cache. Although we have not measured the accurate power consumption in the architecture, according to other related work in power consumption of the memory hierarchy, we can estimate the power consumption qualitatively based on the usage of the memory hierarchy confidently. The quantitative estimation in power consumption can be done in the future work.

Compared to other architecture-level power analysis tools [24], code coverage is a much cheaper approach to analyze the power consumption. It does not require any implementation of high-level power analysis simulator, and code coverage analysis is free and supported by most simulation tools. Therefore, once we have the RTL design, we can estimate the power consumption of the design by analyzing code coverage. It is also useful for validating memory hierarchy efficient software patterns. These could be used to optimize software development. Note that RTL simulation is limited in capacity such that complete programs in operating systems cannot be validated in an RTL simulation environment.

7.2 Future Work

First of all, we can use more powerful test cases to improve code coverage. For example, test cases we use now are generated by the OR1200 compiler. Thus, the test cases are heavily dependent on the compiler. We can use assemble code to test more corner cases. Moreover, some units in the OR1200 core are not enabled, and some ports are connected to default values now. In the future, we can use some approaches to test the interrupt and exception service in the design. Also, we can add another interface to the OR1200 core to test those unused ports.

Furthermore, we can measure the power consumption in the design, and then conclude the accuracy that code coverage can estimate power consumption. Also, if we have a power consumption profile for each block, we can accurately estimate the power consumption based on code coverage in each block.

The future work can also focus on the OR1200 architecture. Currently, the critical path in the OR1200 design is through memory. To demonstrate GLOBALFOUNDRIES' technology metrics and eliminate the influence of custom blocks, we can redesign the architecture accordingly.

APPENDIX A

```
Example 1.1:
#define MAXARRAY    512
int i = 0;
int arrayA[MAXARRAY];
int arrayB[MAXARRAY];
int arrayC[MAXARRAY];

int main(void) {
        for (i=0; i<MAXARRAY; i++)
                arrayA[i] = 0;
        for (i=0; i<MAXARRAY; i++)
                arrayB[i] = arrayA[i] + 0xf00;
        for (i=0; i<MAXARRAY; i++)
                arrayC[i] = arrayB[i] - (0xf00-0xf);
        return 0;
}
```

```
Example 1.2:
#define MAXARRAY    512
int i = 0;
int arrayA[MAXARRAY];
int arrayB[MAXARRAY];
int arrayC[MAXARRAY];

int main(void) {
        for (i=0; i<MAXARRAY; i++)
                arrayA[i] = 0;

        for (i=0; i<MAXARRAY; i++)
        {
                arrayB[i] = arrayA[i] + 0xf00;
                arrayC[i] = arrayB[i] - (0xf00-0xf);

        }
        return 0;
}
```

```
Example 2.1:
int arrayA[1024][4];

int main(void) {
        int m, n;
        for (m=0; m<1024; m++)
        {
                for (n=0; n<4; n++)
                        arrayA[m][n] = m+n;
        }

        for (n=0; n<4; n++)
        {
                for (m=0; m<1024; m++)
                        arrayA[m][n] = arrayA[m][n]
                        + 0xf0;
        }
        return 0;
}
```

```
Example 2.2:
int arrayA[1024][4];

int main(void) {
        int m, n;
        for (m=0; m<1024; m++)
        {
                for (n=0; n<4; n++)
                        arrayA[m][n] = m+n;
        }

        for (m=0; m<1024; m++)
        {
                for (n=0; n<4; n++)
                        arrayA[m][n] = arrayA[m][n]
                        + 0xf0;
        }
        return 0;
}
```

Example 3.1:

```
#define MAXARRAY 512
int i;
int arrayA[MAXARRAY];
int arrayB[MAXARRAY];
int arrayC[MAXARRAY];
int arrayD[MAXARRAY];
int arrayE[MAXARRAY];

int main(void) {
        for (i=0; i<MAXARRAY; i++)
                arrayA[i] = 0;
        for (i=0; i<MAXARRAY; i++)
                arrayB[i] = 0;
        for (i=0; i<MAXARRAY; i++)
                arrayC[i] = 0;
        for (i=0; i<MAXARRAY; i++)
                arrayD[i] = 0;
        for (i=0; i<MAXARRAY; i++)
                arrayE[i] = 0;
        for (i=4; i<MAXARRAY; i++)
                arrayD[i] = arrayC[i];
        for (i=0; i<MAXARRAY-4; i++)
                arrayD[i] = arrayA[i];
        for (i=4; i<MAXARRAY; i++)
                arrayC[i] = 0xf;
        return 0;
}
```

Example 3.2:

```
#define MAXARRAY 512
int i;
int arrayA[MAXARRAY];
int arrayB[MAXARRAY];
int arrayC[MAXARRAY];
int arrayD[MAXARRAY];
int arrayE[MAXARRAY];

int main(void) {
        for (i=0; i<MAXARRAY; i++)
                arrayA[i] = 0;
        for (i=0; i<MAXARRAY; i++)
                arrayB[i] = 0;
        for (i=0; i<MAXARRAY; i++)
                arrayC[i] = 0;
        for (i=0; i<MAXARRAY; i++)
                arrayD[i] = 0;
        for (i=0; i<MAXARRAY; i++)
                arrayE[i] = 0;
        for (i=4; i<MAXARRAY; i++)
                arrayD[i] = arrayC[i];
        for (i=4; i<MAXARRAY; i++)
                arrayC[i] = 0xf;
        for (i=0; i<MAXARRAY-4; i++)
                arrayD[i] = arrayA[i];
        return 0;
}
```

Example 4.1:

```
#define MAXARRAY 512
int i;
int arrayA[MAXARRAY];
int arrayB[MAXARRAY];
int arrayC[MAXARRAY];
int arrayD[MAXARRAY];
int arrayE[MAXARRAY];

int main(void) {
    for (i=0; i<MAXARRAY; i++)
        arrayA[i] = 0;
    for (i=0; i<MAXARRAY; i++)
        arrayB[i] = 0;
    for (i=0; i<MAXARRAY; i++)
        arrayC[i] = 0;
    for (i=0; i<MAXARRAY; i++)
        arrayD[i] = 0;
    for (i=0; i<MAXARRAY; i++)
        arrayE[i] = 0;
    for (i=4; i<MAXARRAY; i=i+4)
        arrayD[i] = arrayC[i];
    for (i=0; i<MAXARRAY-4; i=i+4)
        arrayE[i] = arrayA[i];
    for (i=4; i<MAXARRAY; i=i+4)
        arrayB[i] = arrayC[i];
    return 0;
}
```

Example 4.2:

```
#define MAXARRAY 512
int i;
int arrayA[MAXARRAY];
int arrayB[MAXARRAY];
int arrayC[MAXARRAY];
int arrayD[MAXARRAY];
int arrayE[MAXARRAY];

int main(void) {
    for (i=0; i<MAXARRAY; i++)
        arrayA[i] = 0;
    for (i=0; i<MAXARRAY; i++)
        arrayB[i] = 0;
    for (i=0; i<MAXARRAY; i++)
        arrayC[i] = 0;
    for (i=0; i<MAXARRAY; i++)
        arrayD[i] = 0;
    for (i=0; i<MAXARRAY; i++)
        arrayE[i] = 0;
    for (i=4; i<MAXARRAY; i=i+4)
        arrayD[i] = arrayC[i];
    for (i=4; i<MAXARRAY; i=i+4)
        arrayB[i] = arrayC[i];
    for (i=0; i<MAXARRAY-4; i=i+4)
        arrayE[i] = arrayA[i];
    return 0;
}
```

Example 5.1:

```
#define MAXARRAY 512
int i;
int arrayA[MAXARRAY];
int arrayB[MAXARRAY];
int arrayC[MAXARRAY];
int arrayD[MAXARRAY];
int arrayE[MAXARRAY];

int main(void) {
        for (i=0; i<MAXARRAY; i++)
                arrayA[i] = 0;
        for (i=0; i<MAXARRAY; i++)
                arrayB[i] = 0;
        for (i=0; i<MAXARRAY; i++)
                arrayC[i] = 0;
        for (i=0; i<MAXARRAY; i++)
                arrayD[i] = 0;
        for (i=0; i<MAXARRAY; i++)
                arrayE[i] = 0;
        for (i=4; i<MAXARRAY; i=i+4)
                arrayD[i] = arrayC[i];
        for (i=0; i<MAXARRAY-4; i=i+4)
                arrayE[i] = arrayA[i];
        for (i=4; i<MAXARRAY; i=i+4)
                arrayC[i+1] = 0xf00;
        return 0;
}
```

Example 5.2:

```
#define MAXARRAY 512
int i;
int arrayA[MAXARRAY];
int arrayB[MAXARRAY];
int arrayC[MAXARRAY];
int arrayD[MAXARRAY];
int arrayE[MAXARRAY];

int main(void) {
        for (i=0; i<MAXARRAY; i++)
                arrayA[i] = 0;
        for (i=0; i<MAXARRAY; i++)
                arrayB[i] = 0;
        for (i=0; i<MAXARRAY; i++)
                arrayC[i] = 0;
        for (i=0; i<MAXARRAY; i++)
                arrayD[i] = 0;
        for (i=0; i<MAXARRAY; i++)
                arrayE[i] = 0;
        for (i=4; i<MAXARRAY; i=i+4)
                arrayD[i] = arrayC[i];
        for (i=4; i<MAXARRAY; i=i+4)
                arrayC[i+1] = 0xf00;
        for (i=0; i<MAXARRAY-4; i=i+4)
                arrayE[i] = arrayA[i];
        return 0;
}
```

```
Example 6.1:
#define MAXARRAY 512
int i;
int arrayA[MAXARRAY];
int arrayB[MAXARRAY];
int arrayC[MAXARRAY];
int arrayD[MAXARRAY];
int arrayE[MAXARRAY];

int main(void) {
        for (i=0; i<MAXARRAY; i++)
                arrayA[i] = 0;
        for (i=0; i<MAXARRAY; i++)
                arrayB[i] = 0;
        for (i=0; i<MAXARRAY; i++)
                arrayC[i] = 0;
        for (i=0; i<MAXARRAY; i++)
                arrayD[i] = 0;
        for (i=0; i<MAXARRAY; i++)
                arrayE[i] = 0;
        for (i=4; i<MAXARRAY; i=i+4)
                arrayD[i] = arrayC[i];
        for (i=0; i<MAXARRAY-4; i=i+4)
                arrayE[i] = arrayA[i];
        for (i=4; i<MAXARRAY; i=i+4)
                arrayB[i] = arrayC[i+1];
        return 0;
}
```

```
Example 6.2:
#define MAXARRAY 512
int i;
int arrayA[MAXARRAY];
int arrayB[MAXARRAY];
int arrayC[MAXARRAY];
int arrayD[MAXARRAY];
int arrayE[MAXARRAY];

int main(void) {
        for (i=0; i<MAXARRAY; i++)
                arrayA[i] = 0;
        for (i=0; i<MAXARRAY; i++)
                arrayB[i] = 0;
        for (i=0; i<MAXARRAY; i++)
                arrayC[i] = 0;
        for (i=0; i<MAXARRAY; i++)
                arrayD[i] = 0;
        for (i=0; i<MAXARRAY; i++)
                arrayE[i] = 0;
        for (i=4; i<MAXARRAY; i=i+4)
                arrayD[i] = arrayC[i];
        for (i=4; i<MAXARRAY; i=i+4)
                arrayB[i] = arrayC[i+1];
        for (i=0; i<MAXARRAY-4; i=i+4)
                arrayE[i] = arrayA[i];
        return 0;
}
```

BIBLIOGRAPHY

[1] Lampret, D. *OpenRISC 1200 IP Core Specification*, OpenCores.org, Rev. 0.11, January 2011.

[2] Lampret, D. *OpenRISC 1000 Architecture Manual*, OpenCores.org, January 2004.

[3] OpenCores Organization, *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*, Revision: B3, Released: September 7, 2002.

[4] Bennett, J. *Or1ksim User Guide*, Embecosm Limited, Issue 1 for Or1ksim 0.4.0, 2009. [Online]. Available: http://www.opencores.org/openrisc,or1ksim

[5] M.E.J.B. Bennett, J., Rich D'Addio, *OR1200 GNU Toolchain*, OpenCores. [Online]. Available: http://www.opencores.org/openrisc,gnu_toolchain

[6] Bennett, J., *The OpenCores OpenRISC 1000 Simulator and Toolchain*, Embecosm Limited, November 2008. [Online]. Available: http://www.embecosm.com/appnotes/ean2/html/index/html

[7] Embecosm, *The Or1ksim Simulator*, Embecosm. [Online]. Available: http://www.embecosm.com/appnotes/ean2/html/ch03s07.html

[8] Bennett, J. *Building a Loosely Timed SoC Model with OSCI TLM 2.0, A Case Study Using an Open Source ISS and Linux 2.6 Kernel*, Embecosm Limited, Application Note 1. Issue 2, May 2010.

[9] Embesosm Application Note2. *The OpenCores OpenRISC 1000 Simulator and Tool chain: Installation Guide*, Embecosm Limited, June 2008.

[10] Wang, T. and Tan, C. G., *Practical Code Coverage for Verilog*, Int'l Verilog HDL Conference, Mar. 1995.

[11] Najm, F. N., *A Survey of Power Estimation Techniques in VLSI Circuits*, IEEE Trans. on VLSI Systems, VOL. 2, NO.4, pp. 446-445, Dec. 1994.

[12] Manegold, S., Boncz, P. A., and Kersen, M. L., *Generic Database Cost Models for Hierarchy Memory System*, Technical Report INS-R0203, CWI, Amsterdam, The Netherlands, March 2002.

[13] Catthoor, F., Franssen, F., Wuytack, S., Nachtergaele, L., and DeMan, H., *Global communication and memory optimizing transformations for low power signal processing systems*, In Proceedings, IEEE Workshop on VLSI Signal Processing,

178-187. 1994.

[14] Mathur, A., *Memory Power Reduction in SoC Designs Using PowerProMG. Design And Reuse S.A.*, May 2009.

[15] DeGreef, E., Catthoor, F., and DeMan, H., *Memory organization for video algorithms on programmable signal processors*, In Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors, 552-557, 1995.

[16] Davidson, J. W. and Jinturkar, S., *Memory access coalescing: A technique for eliminating redundant memory access*, ACM SIGPLAN Notices, 29(6), 1994.

[17] Catthoor F., Danckaert K., Wyytack. S. and Dutt., N., *Code transformations for data transfer and storage exploration preprocessing in multimedia processors*, IEEE Journal on Design and Test of Computers, 18(3):70-82, May-June 2001.

[18] Catthoor, F., Wuytack, S., Greef, E. D. Balasa, F., Nachetergaele, L. and Vandecappelle, A., *Custom Memory Management Methodology: Exploration of Memory Organization for Embedded Multimedia System Design, Kluwer Academic Publishers*, Boston/Dordrecht/London, 1998.

[19] Gonzales, R. and Horowitz, M., *Energy dissipation in general-purpose microprocessor*, IEEE Journal of Solid-state Circuit, SC-31(9):1277-1283, September 1996.

[20] Venky, R., *Minimizing Power Consumption in RTL Designs Using Sequential Clock Gating and Low-Power Synthesis*, EDA Design Line, May 2008.

[21] Stickley, J. and Warmke, D., *Detailed DPI Example Code Fragments*, [Online] Available: http://www.systemverilog.org, May 2003.

[22] Synopsys, *Unified Coverage Reporting User Guide*, Version X-2005.12, December 2005.

[23] Roy, K. and Johnson, M. C., *Software Design For Low Power*, In Low Power Design in Deep Submicron Electronics – Proceedings of the NATO Advanced Study Institute, Lucca Italy, August 1996, Kluwer Academic Publishers, Dordrecht, the Netherlands.

[24] Brooks, D., Tiwari, V. and Martonosi, M., *Wattch: A Framework for Architectural-level Power Analysis and Otimizations*, ACM SIGARCH Computer Architecture News, 2000.