

**IMPLEMENTATION OF SIMPLESCALAR
PORTABLE INSTRUCTION SET
ARCHITECTURE (PISA) ON FPGA**

By

ABDUL AZIM BIN ABDULLAH

FINAL PROJECT REPORT

**Submitted to the Electrical & Electronics Engineering Programme
in Partial Fulfillment of the Requirements
for the Degree
Bachelor of Engineering (Hons)
(Electrical & Electronics Engineering)**

**Universiti Teknologi Petronas
Bandar Seri Iskandar
31750 Tronoh
Perak Darul Ridzuan**

© Copyright 2006

by

Abdul Azim bin Abdullah, 2006

CERTIFICATION OF APPROVAL

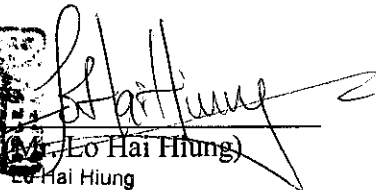
**Implementation of SimpleScalar
Portable Instruction Set Architecture (PISA) on FPGA**

by

Abdul Azim bin Abdullah

A project dissertation submitted to the
Electrical & Electronics Engineering Programme
Universiti Teknologi PETRONAS
in partial fulfilment of the requirement for the
BACHELOR OF ENGINEERING (Hons)
(ELECTRICAL & ELECTRONICS ENGINEERING)

Approved by,


Mr. Lo Hai Hiung
Lecturer
Electrical & Electronic Engineering
Universiti Teknologi PETRONAS
Bandar Seri Iskandar, 31750 Tronoh,
Perak Darul Ridzuan, Malaysia.

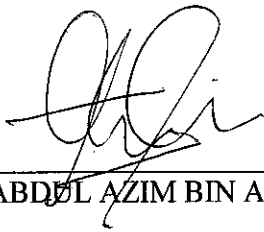
UNIVERSITI TEKNOLOGI PETRONAS

TRONOH, PERAK

DECEMBER 2006

CERTIFICATION OF ORIGINALITY

This is to certify that I am responsible for the work submitted in this project, that the original work is my own except as specified in the references and acknowledgements, and that the original work contained herein have not been undertaken or done by unspecified sources or persons.



ABDUL AZIM BIN ABDULLAH

ABSTRACT

This report describes the current progress of final year project entitled Implementation of SimpleScalar Portable Instruction Set Architecture (PISA) on FPGA. The objectives of this study are to learn computer system architecture, to sharpen skill in programming and debugging a program and to complete study in Universiti Teknologi PETRONAS.

Problem statements will explain the reasons behind of this study was conducted. Firstly, there are few microprocessors in the market currently can be reconfigurable. Secondly, there is a need to design a microprocessor which can be used freely for academic purposes. Thus, in this study, we will focus on the designing of a microprocessor that is reconfigurable, easily understood and freely available for academics purposes.

Methodology will describe way on how this project will be carried out. There are three main steps to be taken which are: 1) Studying the SimpleScalar instruction set architecture; 2) Programming and simulating by using VHDL programming language 3) Implementing the SimpleScalar architecture in VHDL and FPGA.

In the Discussion, a detail contents regarding the project will be explained. Contents included are SimpleScalar's instruction format, register and operation cycle, software and hardware used in the project, the SimpleScalar implementation in VHDL and VHDL simulation. The further details will be discussed later.

Finally, this report is concluded in the Conclusion. Recommendations describe the suggestions that can be done to the current project to improve them in the future.

ACKNOWLEDGEMENT

Alhamdulillah, after 1 year, this project has reached its end. Lots of experience and knowledge were gained throughout the period. There are several individuals, who should be praised and mentioned here. Without them, this project will not be able to be done.

I would like to express the greatest gratitude to Merciful God, Allah S.W.T for His blessings and mercy, which have helped and guided me in during this project.

My almost gratitude goes to my supervisor, Mr. Lo Hai Hiung. He given me advices, ideas, suggestions and ensured that this project will be beneficial to both parties. He supervised me since the first appointment and always kept track of my progress. I really appreciate all the hard work and the time spent, despite his bundle of workload.

I would also like to thank my Computer System Architecture lecturers, Mr. Patrick Sebastian and Dr. Yap Vooi Voon. They were very helpful in giving theoretical, guidance and hands on experience regarding computer architecture. Aside from that, a special thank you to all FYP series lecturers, who was giving me priceless advices on improving my skills in the area of researching, writings and presenting. Not to forget, thanks also to Ms. Siti Hawa who is always supportive.

Last but not least, million thanks to my parents and my fellow colleagues for the all cooperation and support. The encouragement from the people above will always be pleasant memory throughout my life.

TABLE OF CONTENTS

ABSTRACT	i
ACKNOWLEDGEMENT	ii
1. INTRODUCTION	1
1.1 Background Study	1
1.2 Problem Statements	2
1.3 Objectives	3
2. LITERATURE REVIEW	4
2.1 SimpleScalar Instruction Set Architecture	4
2.1.1 Instruction Set	4
2.1.2 Instruction Set Architecture	5
2.2 Implementation on VHDL and FPGA	6
2.2.1 VHDL	6
2.2.2 FPGA	7
3. METHODOLOGY	9
4. DISCUSSIONS	11
4.1 Instruction Format	11
4.2 Register	14
4.3 Operation Cycle	16
4.4 SimpleScalar's Operation Cycle	17
4.5 Software	19
4.5.1 Crimson Editor	19
4.5.2 C Compilers	20
4.5.3 GHDL	21
4.5.4 Altera Quartus II Web Edition	22
4.6 Hardware	25
4.6.1 DSP Development Kit Cyclone II	25

4.7	SimpleScalar in VHDL	26
4.7.1	Fetch	26
4.7.2	Decode	29
4.7.3	Control	31
4.7.4	Execute	33
4.7.5	Memory	35
4.8	VHDL Simulation	38
4.8.1	Unsigned Addition	38
4.8.2	OR Operation	39
4.8.3	Shift Right Logical	39
5.	CONCLUSION	40
6.	RECOMMENDATIONS	41
7.	REFERENCES	42

LIST OF ILLUSTRATIONS

LIST OF FIGURES

Figure 1:	FPGA Workflow	8
Figure 2:	Methodology Steps	9
Figure 3:	Instruction Format	11
Figure 4:	Instruction Set	13
Figure 5:	Register	14
Figure 6:	General Operation Cycle	16
Figure 7:	SimpleScalar's Operation Cycle	17
Figure 8:	SimpleScalar's Operation Cycle (C Language)	18
Figure 9:	Crimson Editor	19
Figure 10:	Borland C	20
Figure 11:	GHDL	21
Figure 12:	Altera Quartus II Web Edition	22
Figure 13:	Basic Design Flow	23
Figure 14:	Altera Cyclone II EP2C35 FPGA	25
Figure 15:	SimpleScalar's Operation Cycle (VHDL Implementation)	26
Figure 16:	SimpleScalar Instruction (Register & Immediate Format)	27
Figure 17:	SimpleScalar Opcodes (Register & Immediate Format)	29
Figure 18:	SimpleScalar Immediate Fields (Register Format)	30
Figure 19:	SimpleScalar Immediate Fields (Immediate Format)	30
Figure 20:	Register Selection	31
Figure 21:	Register-Memory	32
Figure 22:	32-bit Full Adder	34
Figure 23:	Read Cycle Timing Waveforms	36
Figure 24:	Write Cycle Timing Waveforms	37
Figure 25:	Unsigned Addition Operation	38
Figure 26:	OR Operation	39
Figure 27:	Shift Right Logical	39

LIST OF TABLES

Table 1:	Definitions of SimpleScalar Architecture Registers	15
Table 2:	Quartus II Web Edition Device Support	22
Table 3:	Full Adder Truth Table	33

TABLE OF APPENDICES	43
----------------------------	-----------

CHAPTER 1: INTRODUCTION

1.1. Background Study

Modern processors are incredibly complex marvels of engineering that are becoming increasing hard to evaluate. SimpleScalar tool set performs fast, flexible and accurate simulation for modern processors that implement the SimpleScalar architecture.

According to D. Burger [1], SimpleScalar simulators can emulate the Alpha, PISA, ARM, and x86 instruction sets. The tool set includes a machine definition infrastructure that permits most architectural details to be separated from simulator implementations. All of the simulators distributed with the current release of SimpleScalar can run programs from any of the above listed instruction sets. Complex instruction set emulation (e.g., x86) can be implemented with or without microcode, making the SimpleScalar tools particularly useful for modeling CISC instruction sets.

The advantages of this tool are flexibility, portability, extensibility and performance. This tool set is portable, requiring only that the GNU tools may be installed on the host system. The tool set has been used on multiple platforms such as Linux/x86, Win NT, SPARC and Solaris. The tool set is easily extensible. The instruction set is designed to support easy annotation of instructions, without requiring a retargeted compiler for incremental changes. The instruction definition method along with the ported GNU tools makes new simulators easy to write and the old ones even simpler to extend. Finally, the simulators have been aggressively tuned for performance and can run codes approaching “real” sizes in tractable amounts of time. [1]

In this project, I will design a Portable Instruction Set Architecture (PISA) microprocessor in VHSIC Hardware Description Language (VHDL) and implement it on FPGA.

The PISA instruction set is a simple MIPS-like instruction set maintained primarily for instructional use. A GNU GCC-based cross-compiler and pre-built libraries are also available for this target. The PISA target is particularly useful for computer engineering instruction as the tools can be built on a wide range of host platforms, including Linux/x86, Win2000, SPARC Solaris, and others. [1]

1.2. Problem Statements

In the current design of microprocessor, there are few microprocessors which can be reconfigurable. “Reconfigurable” term means the memory addressing and registers of the given microprocessor can be adjusted according to the author’s preferences. Currently, all microprocessor available in the market, the function units, memory addressing and registers are fixed and cannot be reconfigured. Therefore, this project is attempting to design a microprocessor which is reconfigurable.

Currently, there are a lot of microprocessors designs available today from Intel, Motorola, SPARC and others. However, not all of them are easy to be understood by students who just begin their learning in computer system. In the learning curve, to know and understand the concept of computer system is by learning from the simplest form of digital system, logic circuits until the hardest part, which is the memory system. SimpleScalar, which is based on MIPS, provides an easy and simple architecture for study. In addition, it is free for academic purposes and open source for development. In this project, the simplest microprocessors will be design.

From studies made, it is found that SimpleScalar PISA can be implemented as a microprocessor. Besides it is free for non-commercial use, it is also reconfigurable and flexible to all platforms. PISA which is like MIPS-like instruction is good architecture for study, because it is easy to understand.

1.3. Objectives

To implement SimpleScalar PISA in FPGA.

SimpleScalar tool set is used to evaluate modern processors using the SimpleScalar architecture. However, it is only available in software based. The source code must be compiled first before it can be executed. Up to date, there is no hardware based implemented for SimpleScalar PISA. Therefore, in this project, I will implement the SimpleScalar PISA in hardware called FPGA.

To design and program circuits using VHDL language.

My interest is programming and I have learnt a lot of languages such as C, C++, HTML, Visual Basic, MATLAB and PHP. I also had experienced in microcontrollers programming. However, VHDL is one of the programming languages I did not manage to learn. Therefore, this project is able to help me to gain new knowledge and experience in programming the digital circuit using VHDL language.

To apply and relate computer system architecture.

In the computer system subject, I have learnt digital logic gates, full adder system, basic computer architecture, register design and memory design. From this project, I hope I will be able to apply and relate the concept of computer system subject learnt.

CHAPTER 2: LITERATURE REVIEW

This project can be divided into two major partitions which are SimpleScalar Instruction Set Architecture and Implementation on the VHDL and FPGA.

2.1. SimpleScalar Instruction Set Architecture

2.1.1. Instruction Set

Instruction set is a collection of all operations possible in a machine's language. There are many types of instructions in a computer system, such as arithmetic instructions, data movement instructions, control or branch instructions and many more.

In arithmetic instructions, it will accept one or more operands and produce a result. Besides, it may also set a flag to indicate that the result of the operation was a negative number. In data movement instructions, it moves data within the machine and to or from input/output devices. In control or branch instructions, it affects the order in which instructions are performed, or control the flow of the executing program, much as *goto*, *for*, and function calls do in C. [2]

Every instruction must contain encodings within it to specify the following 4 things, either explicitly or implicitly:

1. Which operation to perform.
2. Where to find the operand or operands, if there are operands.
3. Where to put the results, if there is a result.
4. Where to find the next instructions.

Source: John L. Hennessy & David A. Patterson, "Computer Architecture: A Quantitative Approach" [2]

In SimpleScalar, the instruction set can be divided into 4 groups, which are:

1. Control Instruction
2. Load/Store Instruction
3. Integer Instruction
4. Floating point Instruction

Source: Doug Burger, Todd M. Austin, "The SimpleScalar Tool Set, Version 2.0" [1]

(Refer to *Appendix 1: List of SimpleScalar Instruction Set* for more details)

2.1.2. Instruction Set Architecture

Instruction set architecture is the collection of instructions and resources. It includes the instruction set, the machine's memory and all of the programmer-accessible registers in the CPU and elsewhere in the machine. [3]

The SimpleScalar architecture can be divided into parts:

- Instruction set principles.
- Memory hierarchy and register design.
- 5 stages of pipelining.
- Level 1 and level 2 cache.

Source: Doug Burger, Todd M. Austin, "The SimpleScalar Tool Set, Version 2.0" [1]

2.2. Implementation on VHDL and FPGA

2.2.1. VHDL

VHDL is an acronym of VHSIC Hardware Description Language. VHSIC is another acronym which stands for Very High Speed Integrated Circuits.

In digital design, the VHDL language is used for documentation, verification and synthesis of large digital system. It allows the system can be described in the same code to achieve these goals at one time, thus saving a lot of effort. [6]

There are three different approaches are used to describe hardware in VHDL. They are structural, data flow and behavioral methods of hardware description. In the beginning, the design behaviour is described (modeled) and verified (simulated). By using the synthesis tools, the design is able to be translated into real hardware (gates and wires). At this point, they are mapped onto a programmable logic device such a CPLD or FPGA. [6]

The VHDL standards are developed by IEEE (Institute of Electrical and Electronics Engineers). Currently, there are two standards widely used, which are VHDL'87 (STD 1076-1987) version and VHDL'93 (adopted in 1994). [6]

2.2.2. FPGA

FPGA is an acronym which stands for Field Programmable Gate Array. The term of “Field Programmable” refers to the ability to change the operation of the device, while “Gate Array” refers to the matrix of logic cell surrounded by a peripheral of I/O cells. Simply, FPGA are programmable digital logic chips which can be program to do digital function. [7]

FPGAs come in a wide variety of sizes and many different combinations of internal and external features from different manufacturers. Although they are different in many things, they have a common, which is composed of programmable logic blocks. Each of these blocks contains registers and logic elements, which are arranged in a grid and tied together using programmable interconnections. [7]

In a typical FPGA, the logic blocks that make up the bulk of the device are based on lookup tables (of perhaps four or five binary inputs) combined with one or two single-bit registers and additional logic elements such as clock enables and multiplexers. These basic structures may be replicated many thousands of times to create a large programmable hardware fabric. [7]

In more complex FPGAs these general-purpose logic blocks are combined with higher-level arithmetic and control structures, such as multipliers and counters, in support of common types of applications such as signal processing. In addition, specialized logic blocks are found at the periphery of the devices that provide programmable input and output capabilities. [7]

Figure 1 shows the general workflow when working with FPGA.

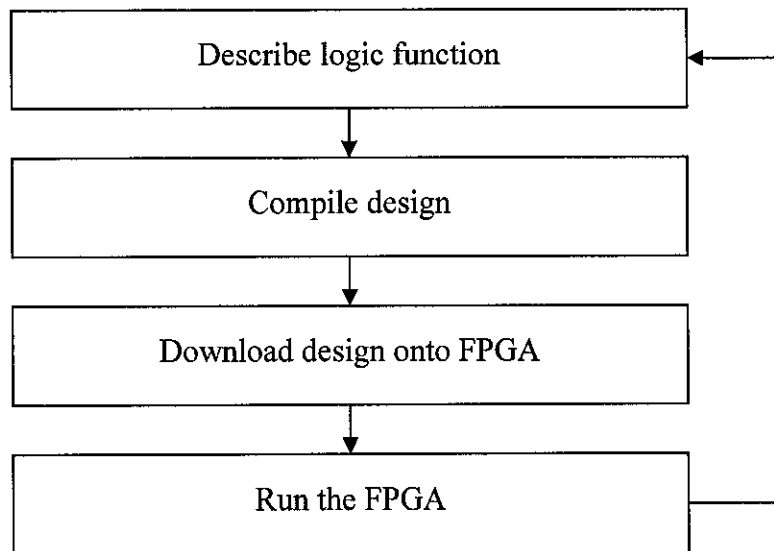


Figure 1: FPGA Workflow

First step is to describe the logic function that wants to be developed. Draw schematic or write program to describe the particular function.

Then, compile the design. The logic function designed is compiled by using the software provided from FPGA vendor (e.g.: Xilinx ISE, Altera Quartus, Active VHDL and etc). This will create a binary file that can be downloaded into the FPGA.

The next step is to download the design onto FPGA. Connect cable from the computer to the FPGA and download the binary file created to the FPGA.

Finally, run the FPGA. If successfully, the FPGA will behave according to the logic function. If not, repeat the steps again to re-develop.

Source: fpga4fun.com, What are FPGAs? [20]

CHAPTER 3: METHODOLOGY

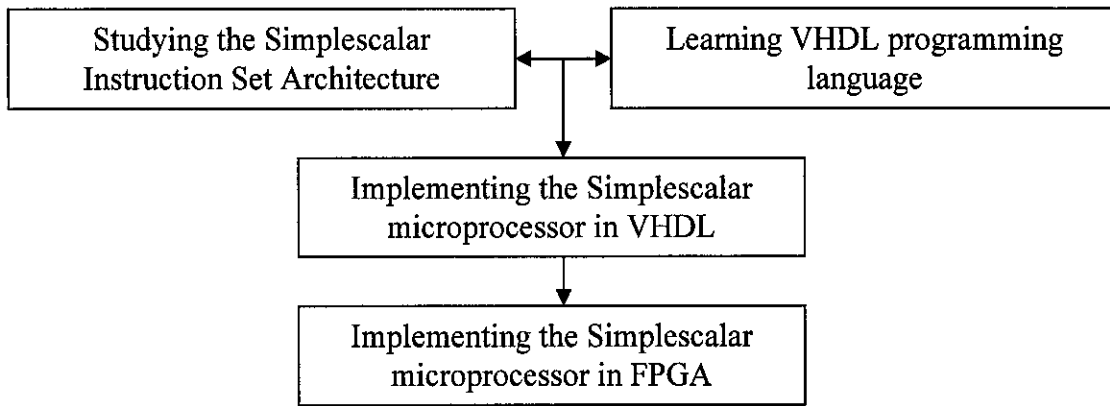


Figure 2: Methodology Steps

Figure 2 above shows the steps I will be taking during implementation of this project. The first part is to study the SimpleScalar Instruction Set Architecture. This involves understanding the source code given, what instruction sets are to be used, how to set the memory addressing and registers and many more. Besides that, I also will have to simulate the microprocessors by using the tools given in order to help me to understand how it works.

Parallel with the SimpleScalar architecture studies, I will have to learn the VHDL programming language. This requires understanding of the digital system design concepts, writing the source codes and doing some programming exercises given in the books. The software I will be using in VHDL programming is Altera Quartus II software.

Then, I will have to implement the SimpleScalar microprocessor in VHDL. This step requires me to convert from the source code given and implement it by using VHDL programming languages I have learnt. This step requires a lot of programming and debugging the program.

Final step of this project is to implement the SimpleScalar microprocessor which has been designed by using VHDL on the FPGA. This step requires a lot of programming, debugging the program and troubleshooting the hardware.

The schedule of this project during Final Year Project I and II can be referred to Planning Schedule on Appendix 2: Planning Schedule.

CHAPTER 4: DISCUSSIONS

4.1. Instruction Format

The format of an instruction is usually depicted by a rectangular box symbolizing the bits of the instruction, as they appear in memory words or in a control register. The bits are divided into groups or parts called fields. Each field is assigned a specific item, such as the operation code, a constant value, or a register file address. The various fields specify different functions for the instruction and when shown together, constitute instruction format. [9]

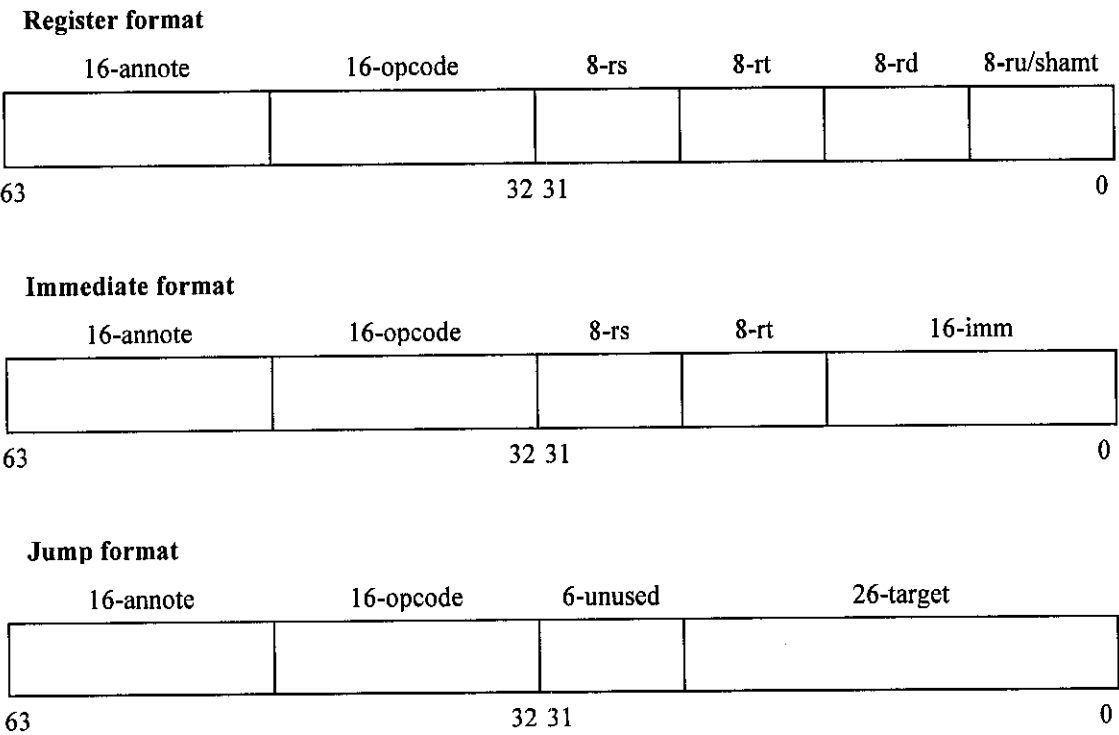


Figure 3: Instruction Format

Source: Doug Burger, Todd M. Austin, "The SimpleScalar Tool Set, Version 2.0" [1]

The three instruction formats for the SimpleScalar are illustrated in the Figure 3. SimpleScalar architecture is derived from MIPS-IV instruction set architecture. Therefore, it has same instruction set as MIPS-IV. All instructions are 64 bits in length. The instructions can be divided into three formats: *register*, *immediate* and *jump*. [1]

The register format is used for computational instructions. The immediate format supports the inclusion of a 16-bit constant. The jump format support specification of 24-bit jump targets. The register fields are all 8 bits, to support extension of the architected registers to 256 integers and floating point register. Each instruction format has a fixed-location, 16-bit opcode field that facilitates fast instruction encodings. [1]

8 bits, $2^8 = 256$ integers from 00000000 to 11111111

The bits are divided into groups or parts called fields. Each field is assigned a specific item, such as operation code, a constant value or a register file address.

The operation code of an instruction, often shortened to “opcode”, is a group of bits in the instruction format. This determines which operations to be conducted by the processor. The operation of instruction is differentiate by using opcode. For example, the opcode for ADD instruction is 0x40 or 01000000 while the opcode for SUB instruction is 0x44 or 01000100. In SimpleScalar, the opcode is in hexadecimal. However, the opcodes of all instructions are 8 bits. The instruction format for the opcode is 16 bits. Therefore, the remaining 8 bits must be filled by using either zero fill or sign extension. In this architecture, zero-fill is specified for the operand. [1]

Constant value is the immediate value available in the instruction. In SimpleScalar, the value supported for the immediate value is from 0 to 65536. [1]

For full instructions, please refer to Appendix 3: SimpleScalar Instructions for.

File *pisa.def* defines all aspects of the SimpleScalar instruction set architecture. Each instruction set in the architecture has a DEFINST macro call. Here, shows example on how the instructions are organized and defined in the source code:

```

#define ADD_IMPL
{
    if (OVER(GPR(RS), GPR(RT)))
        DECLARE_FAULT(md_fault_overflow);

    SET_GPR(RD, GPR(RS) + GPR(RT));
}
DEFINST(ADD,
        "add",
        IntALU,
        DGPR(RD), DNA,
        0x40,
        "d,s,t",
        F_ICOMP,
        DGPR(RS), DGPR(RT), DNA)

```

← semantics

← opcode

← instruction flags

Figure 4: Instruction Set

Source: Todd M. Austin, “SimpleScalar Hacker’s Guide” [8]

Figure 4 shows on how the instruction set is defined in the *pisa.def*. The instruction is ADD arithmetic operation. The operation will involve:

1. Reading values from general purpose register of RS and general purpose register of RT.
2. Doing the operation, adding between general purpose register of RS and general purpose register of RT.
3. Writing (Storing) the results in the general purpose register of RD.

The opcode of this instruction is 0x40 in hexadecimal or 01000000 in binary. Different operation will use different opcode. Since the instruction is arithmetic operation between integers, therefore the functional unit requirement is IntALU. This operation also has helper function which is available to assist in the construction of instruction expression. OVER(GPR(RS), GPR(RT)) function is an overflow checking. This will check whether the results of the operation given have overflow or not. If overflow has occurred, a function DECLARE_FAULT(md_fault_overflow) will be called. [8]

4.2. Register

This module implements the SimpleScalar architected register state, which includes integer and floating point registers and miscellaneous registers. The architected register state is as follows:

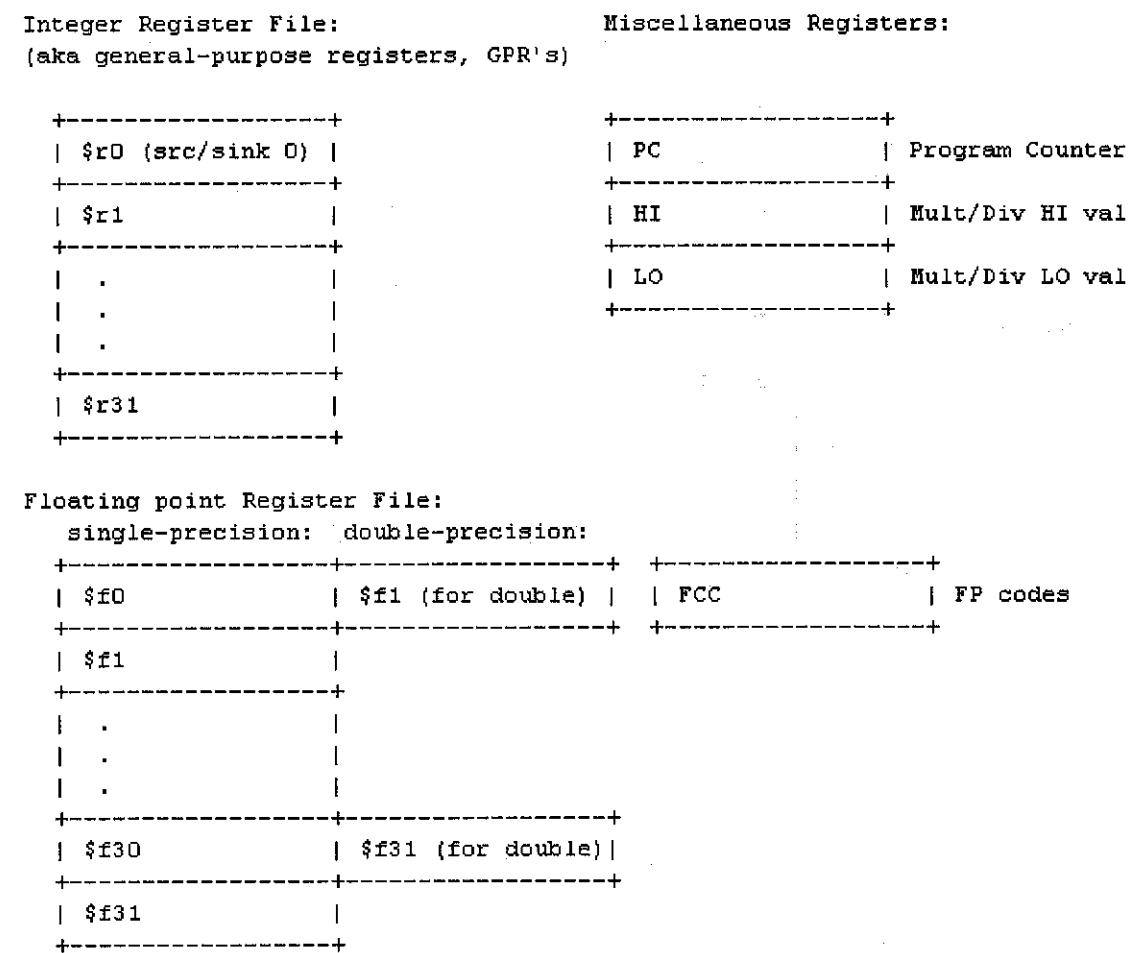


Figure 5: Register

Source: SimpleScalar Source Code (regs.h) [10]

The floating point register file can be viewed as either 32 single-precision (32-bit IEEE format) floating point values \$f0 to \$f31, or as 16 double-precision (64-bit IEEE format) floating point values \$f0 to \$f31. [10]

Table below shows the definitions of SimpleScalar architecture register.

Hardware Name	Software Name	Description
\$0	\$zero	zero-valued source/sink
\$1	\$at	reserved by assembler
\$2-\$3	\$v0-\$v1	fn return result regs
\$4-\$7	\$a0-\$a3	fn argument value regs
\$8-\$15	\$t0-\$t7	temp regs, caller saved
\$16-\$23	\$s0-\$s7	saved regs, callee saved
\$24-\$25	\$t8-\$t9	temp regs, caller saved
\$26-\$27	\$k0-\$k1	reserved by OS
\$28	\$gp	global pointer
\$29	\$sp	stack pointer
\$30	\$s8	saved regs, caller saved
\$31	\$ra	return address reg
\$hi	\$hi	high result register
\$lo	\$lo	low result register
\$f0-\$f31	\$f0-\$f31	floating point registers
\$fcc	\$fcc	floating point condition code

Table 1: Definitions of SimpleScalar architecture registers

Source: Doug Burger, Todd M. Austin, “The SimpleScalar Tool Set, Version 2.0” [1]

These registers defined in SimpleScalar architecture with their hardware name, software name and description. Take note, the registers used by the SimpleScalar is the same with MIPS IV ISA. [1]

4.3. Operation Cycle

The basic operation cycle of a computer is controlled by a control unit that puts into the following steps:

Step 1: Fetch the instruction from memory into a control register

Step 2: Decode the instruction

Step 3: Locate the operands used by the instruction

Step 4: Fetch operands from memory (if necessary)

Step 5: Execute the operation in processor register

Step 6: Store the results in the proper locations

Step 7: Repeat Step 1 with next instruction

Figure 6: General Operation Cycle

Source: M. Morris Mano, Charles R. Kime, "Logic and Computer Design Fundamentals" [9]

There is a register in the computer called the Program Counter (PC) that keeps track of the instructions in the program stored in the memory. The PC holds the address of the instruction to be executed next and is incremented by one each time a word is read from the program in memory. The decoding done in the Step 2 determines the operation to be performed and the addressing mode of the instruction. The operands in Step 3 are located from the addressing mode and the address field of the instruction. The computer executes the instruction, storing the results and returns to Step 1 to fetch the next instruction in sequences. [9]

4.4. SimpleScalar’s Operation Cycle

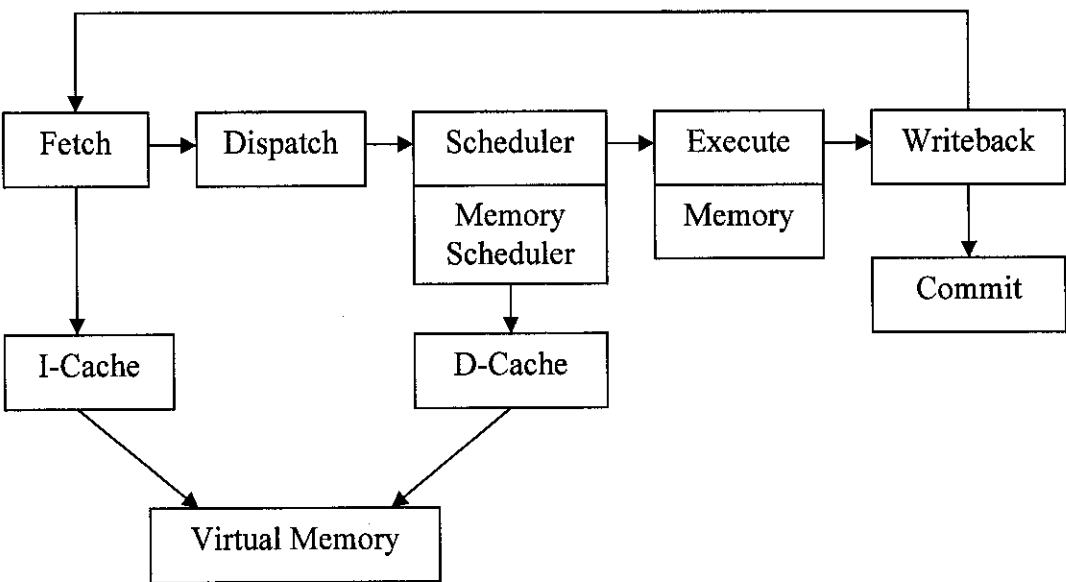


Figure 7: SimpleScalar’s Operation Cycle

Source: Doug Burger, Todd M. Austin, “The SimpleScalar Tool Set, Version 2.0” [1]

Figure 7 shows the operation cycle of SimpleScalar processors. The concept of SimpleScalar’s operation cycle has similar to the general operation cycle we have discussed before. The only different is the term used in Dispatch process, Scheduler process and Writeback process. However, their purpose is the same as the general operation cycle. There are 6 cycles of SimpleScalar processors, which are Fetch, Dispatch, Execute, Writeback and Commit. Each cycle will be discussed later.

```
ruu_init()
for (;;) {
    ruu_commit();
    ruu_writeback();
    lsq_refresh();
    ruu_issue();
    ruu_dispatch();
    ruu_fetch();
}
```




Figure 8: SimpleScalar’s Operation Cycle (C Language)

Source: SimpleScalar Source Code (sim-outorder.c) [10]

Figure 8 shows the C language implementation of the SimpleScalar’s operation cycle. In *sim-outorder.c*, this operation cycle is implemented as pipelining. It is implemented reversely from Commit to Fetch. According to Doug Burger, this will eliminate this/next state synchronization and relaxation problems. [1]

4.5. Software

For this project to be completed and successful, I have used software for development. There are editors, compilers, synthesizers and simulators software. The development platform of this project will be under Windows XP operating system.

4.5.1. Crimson Editor

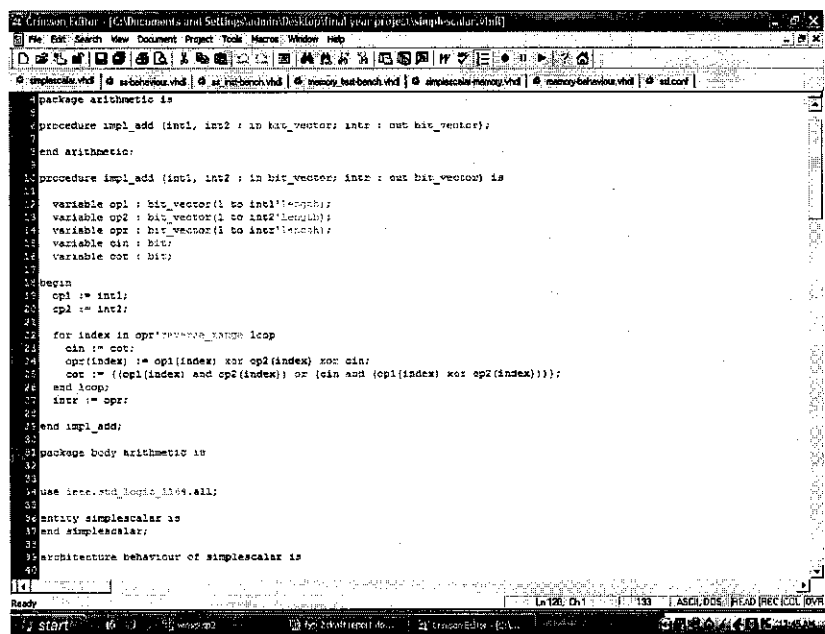


Figure 9: Crimson Editor

Crimson Editor is a professional source code editor for Windows platform. It can be downloaded free from the Internet at <http://www.crimsoneditor.com/>.

This software supports many programming languages such as HTML, C/C++, Perl, Java and even VHDL. One features of this editor is it enables syntax highlighting of all programming languages and can be extend for other programming languages. [11]

4.5.2. C Compilers

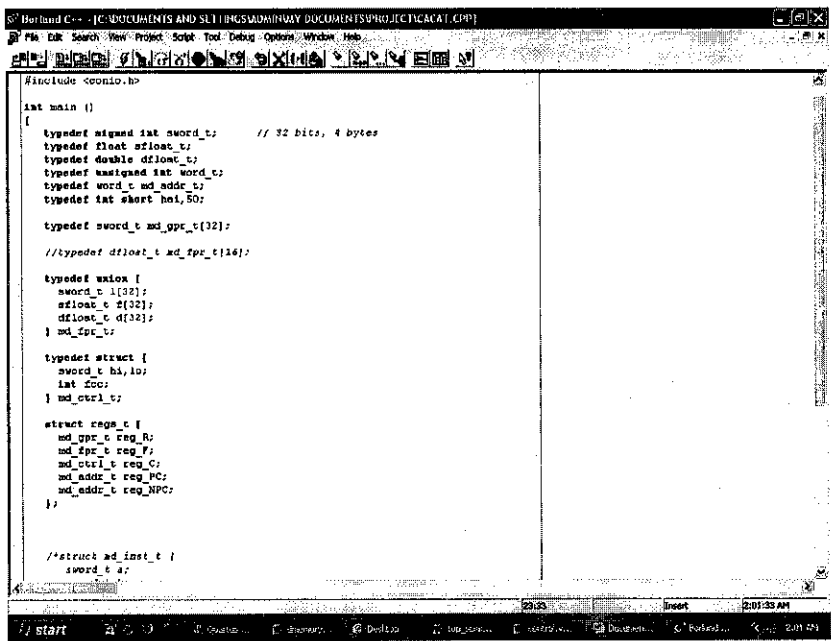


Figure 10: Borland C

I have used two C compilers, which are Borland C and Microsoft Visual Studio. Both programs can be used to edit, view and compile a C source codes. However, I will not be using this program to compile the SimpleScalar source codes. Rather than, these programs are used to check and test the SimpleScalar source codes. These involving checks the size of an arrays, the syntax used and variables used in the SimpleScalar source code.

4.5.3. GHDL

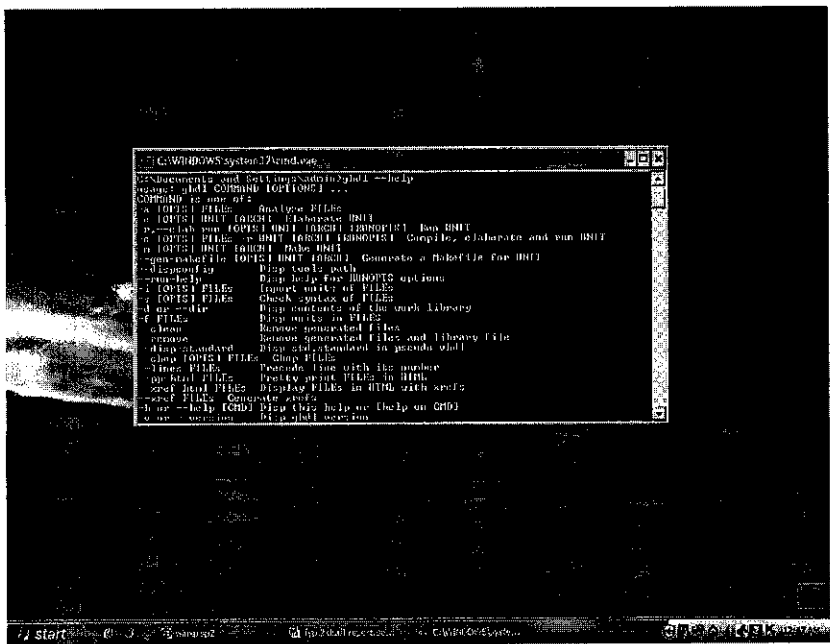


Figure 11: GHDL

GHDL is one of software I had used in this final year project. GHDL is a VHDL simulator, using the GCC technology and implementing the VHDL language according to the VHDL 1987 (IEEE 1076-1987) and VHDL 1993 (IEEE 1076-1993) standards. With GHDL, the program and designed written in VHDL can be compiled into executable files. With the binary files created from compilation, the design can be simulated. [12]

GHDL is an open source project and is free under GNU General Public License. Under the GNU license, this software can be redistributed and modified. It is free from restriction and license issues that arise with commercial simulators. Currently, there are two processors which are successfully compiled and run by using GHDL. There are DLX processors and LEON1 SPARC processors. [12]

However, it has disadvantage over the commercial simulator software. The design created does not be able to synthesis. It cannot translate the design into netlist and not be able to transfer onto FPGA. [12]

4.5.4. Altera Quartus II Web Edition

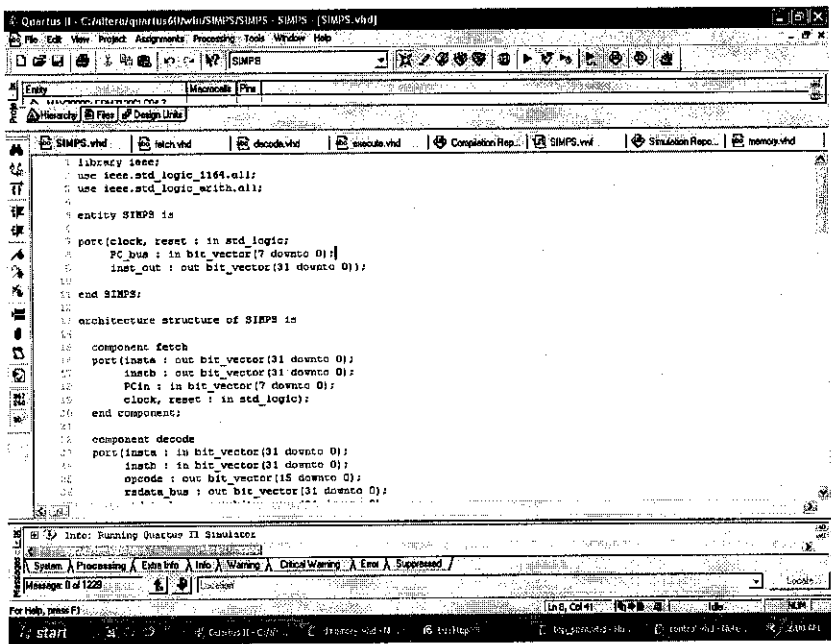


Figure 12: Altera Quartus II Web Edition

The main software for VHDL development will be Quartus II Web Edition. This software can be obtained free from Altera site, <http://www.altera.com/> . A license is required and it can be enquired freely at the particular website. This software supports Cyclone II of device family, which is the hardware that I will be using for FPGA. [13]

Device Family	Device Supported
MAX II MAX 3000A MAX 7000AE MAX 7000B MAX 7000S Cyclone II Cyclone FLEX 10K® FLEX® 10KA ACEX® FLEX 6000	All devices

Table 2: Quartus II Web Edition Device Support

Source: Altera, Quartus II Web Edition Software [13]

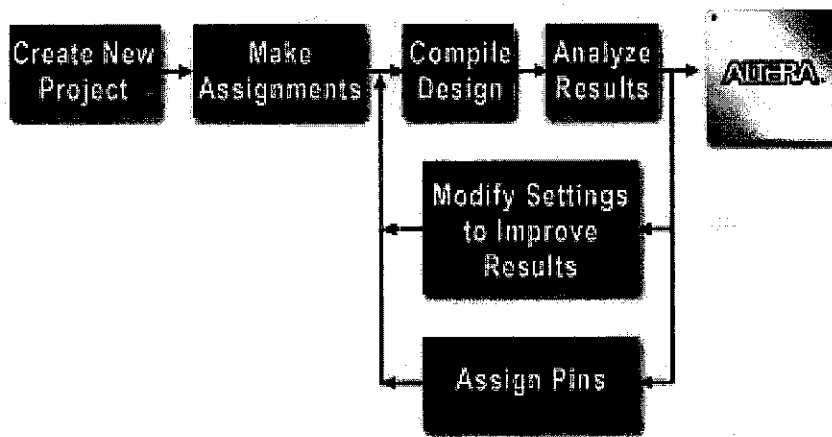


Figure 13: Basic Design Flow

Source: Altera, Quartus II Software Basic Design Flow [13]

Figure 13 shows the basic design flow for the Quartus II software. The users can set up project and compile the design by using these steps. Altera defines 6 stages of developing the VHDL. [13]

The first stage is creating new project. At this stage involves declaration of entity or component, design files and libraries used in the project, and the device family and package used by the project. Next is making assignments. This stage requires specifying global maximum operating frequency requirements (fMAX), paths should not be reported in timing analysis reports and others. [13]

The next step will be compile design and analyzed the results. Before the project can be simulated and implemented, the project must be verified first. Here, each syntax of entity, component and architecture developed are checked. After compiling the design, a report summary of compiled results will be shown automatically. This report shows all the place-and-route results details and it is linked to many other software features. [13]

At the same time, if the results are not satisfied and we want to improve the current results, it can be changed by using assignment settings assignment editors) or by changing timing requirements in the Timing Wizard. Then, the design is compiled again and the results are analyzed. By default, the software will automatically assign pins to the top-level I/O signals. It also can be done by manual using the Assignment Editor. [13]

4.6. Hardware

4.6.1. DSP Development Kit Cyclone II

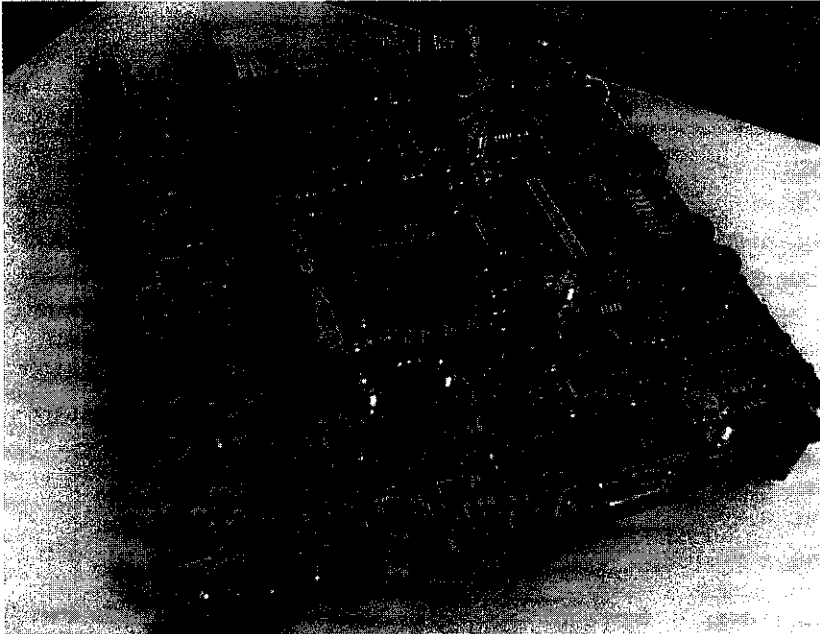


Figure 14: Altera Cyclone II EP2C35 FPGA

Figure 14 shows the hardware that I will be using to implement the SimpleScalar processor on it. The hardware is Cyclone II EP2C35 FPGA. An overview is summarized:

- Logic Elements: 33,216
- M4K RAM Blocks (4 kbits + 512 Parity Bits): 105
- Total RAM Bits: 483,840
- Embedded 18x18 Multipliers: 35
- PLLs: 4
- Maximum User I/O Pins: 475
- Differential Channels: 205

Source: Cyclone II FPGA Family Overview [13]

4.7. SimpleScalar in VHDL

In this section, I will describe the VHDL implementation of SimpleScalar processors. In the implementation, the processors are divided into five cycles, which are Fetch, Decode, Execute and Memory.

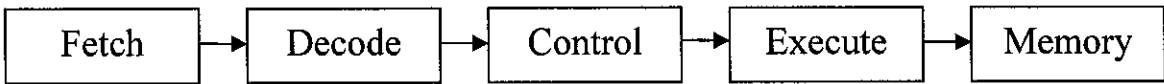


Figure 15: SimpleScalar’s Operation Cycle (VHDL Implementation)

4.7.1. Fetch

In the Fetch cycle, the stored instructions are bring out from the memory and send to the bus line. Then, these instructions will be decoded in the Decode cycle.

```
constant mem0 : bit_vector(31 downto 0) := B"00000000000000000000000000000000"; -- no operation
constant mem1 : bit_vector(31 downto 0) := B"00000000000000000000000000000000";
constant mem2 : bit_vector(31 downto 0) := B"00000000000000000000000001010000"; -- load opcodes
constant mem3 : bit_vector(31 downto 0) := B"00000001000000010000000000000111";

..
..

constant mem20: bit_vector(31 downto 0) := B"00000000000000000000000001010101"; -- shift left
constant mem21: bit_vector(31 downto 0) := B"000000010000000100000101000000001";
constant mem22: bit_vector(31 downto 0) := B"00000000000000000000000001010111"; -- shift right
constant mem23: bit_vector(31 downto 0) := B"000000010000000100000101000000001";
```

Source: *fetch.vhd* [Appendix IV]

From the source code above, mem0 to mem23 represents the stored instructions. Each instruction is 32 bit width. For ease of simplification, all the instructions are stored at the specific memory location. Instruction 00000000000000000000000000000000 is stored at mem0, another instruction 00000001000000010000000000000111 is stored at mem3 and consequently.

The SimpleScalar instructions can be divided into two sections as defined in the C languages, which are SimpleScalar opcodes and SimpleScalar unsigned immediate fields. Each section is an unsigned word data type and has 32 bit width.

```
typedef struct {
    word_t a;          /* simplescalar opcode (must be unsigned) */
    word_t b;          /* simplescalar unsigned immediate fields */
} md_inst_t;
```

Source: SimpleScalar Source Code (pisa.h) [10]

Register & Immediate format

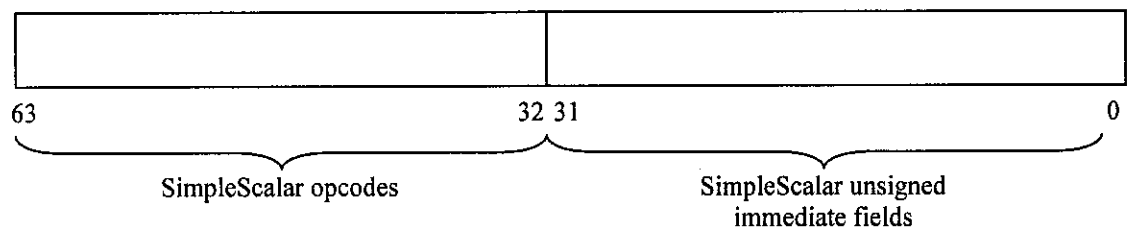


Figure 16: SimpleScalar Instruction (Register & Immediate Format)

Source: SimpleScalar Tools Set [1]

SimpleScalar unsigned immediate fields are 32-bit from bit 0 till bit 31 of SimpleScalar instructions and SimpleScalar opcodes are also 32-bit from bit 32 till bit 63. The instructions are fetched from the memory accordingly to the program counter by using function MD_FETCH_INST. (See the source code below)

```
#define MD_FETCH_INST(INST, MEM, PC)
{ inst.a = MEM_READ_WORD(mem, (PC));
  inst.b = MEM_READ_WORD(mem, (PC) + sizeof(word_t)); }

MD_FETCH_INST(inst, mem, regs.regs_NPC);
```

Source: SimpleScalar Source Code (pisa.h, sim-fast.c) [10]

```

process(PC)
begin
  case PC is
    when X"00" =>
      insta <= mem0;
      instb <= mem1;
      ...
      ...
    when X"2C" =>
      insta <= mem22;
      instb <= mem23;

    when others =>
      insta <= null;
      instb <= null;
    end case;
  end process;

```

Source: fetch.vhd [Appendix IV]

Source code above shows on how the instructions are fetched using VHDL language. When the PC is X"00", the instruction at mem0 will be sent to insta (as SimpleScalar opcode) and another instruction at mem1 will be sent to instb (as SimpleScalar immediate fields).

```

entity fetch is
  port(insta : out bit_vector(31 downto 0); -- inst.a
        instb : out bit_vector(31 downto 0); -- inst.b
        PCin : in bit_vector(7 downto 0);   -- PC
        clock, reset : in std_logic);
end fetch;

```

Source: fetch.vhd [Appendix IV]

In *fetch.vhd*, there are three inputs, which are the PC (program counter), clock and reset and two outputs, which are insta and instb. insta is the SimpleScalar opcodes and instb is the SimpleScalar immediate fields. These outputs will be the inputs during Decode cycle.

(Refer to *Appendix IV: fetch.vhd* for the source code)

4.7.2. Decode

In Decode cycle, the fetched instructions will be translated into specific fields, which are OP (opcode), RS (register source #1), RT (register source #2), RD (register destination) and IMM (immediate value). The implementation of Decode cycle in C language can be seen as follows:

```
/* returns the opcode field value of SimpleScalar instruction INST */
#define MD_OPFIELD(INST)      (INST.a & 0xff)
#define MD_SET_OPCODE(OP, INST)  ((OP) = ((INST).a & 0xff))

/* integer register specifiers */
#undef RS      /* defined in /usr/include/sys/syscall.h on HP/UX boxes */
#define RS      (inst.b >> 24)      /* reg source #1 */
#define RT      ((inst.b >> 16) & 0xff) /* reg source #2 */
#define RD      ((inst.b >> 8) & 0xff)  /* reg dest */
```

Source: SimpleScalar Source Code (pisa.h) [10]

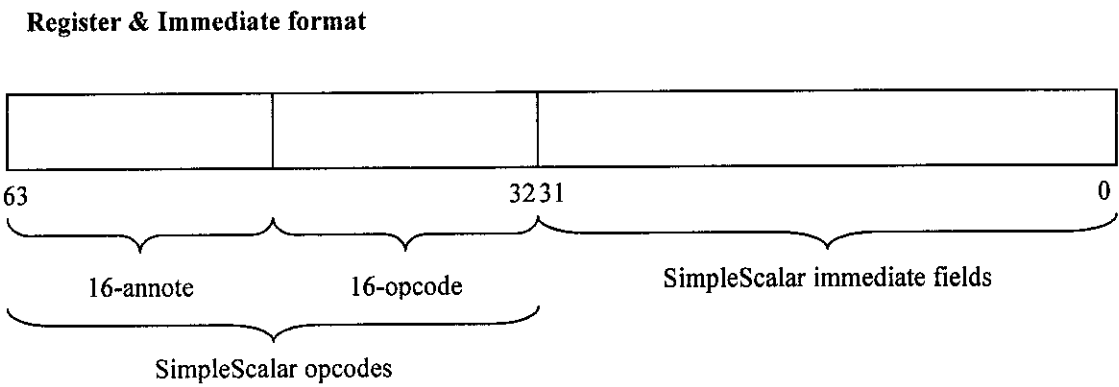


Figure 17: SimpleScalar Opcodes (Register & Immediate Format)

Source: SimpleScalar Tools Set [1]

Figure 17 shows the SimpleScalar opcodes for Register and Immediate format. In the SimpleScalar opcodes, there will be two fields, which are annotate and opcode. In the annotate field, the SimpleScalar allows new instructions to be added or implemented into the current instruction set. The length of this field is 16 bit. In the opcode field, the SimpleScalar operation codes are defined. In other words, any operations of the instructions to be executing will be depending to this field. For example, in SimpleScalar,

the addition between two registers will happen when the opcode is 0x40. If the instruction fetched having the opcode of 0x40 in this field, the addition will be executed. Otherwise, another operation will be executed depending on the opcodes.

Register format

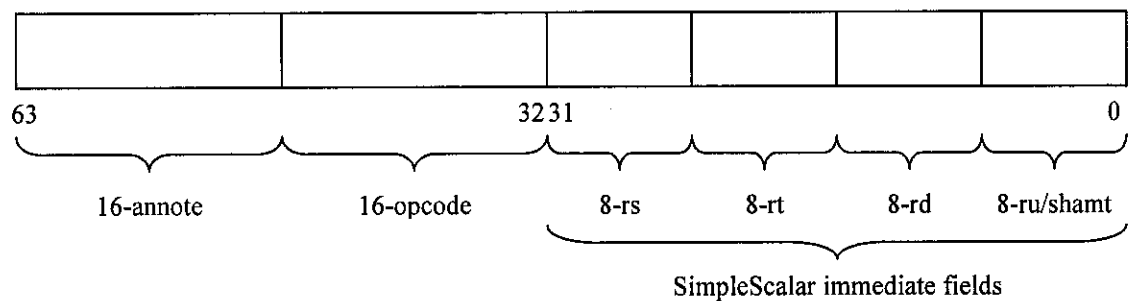


Figure 18: SimpleScalar Immediate Fields (Register Format)

Immediate format

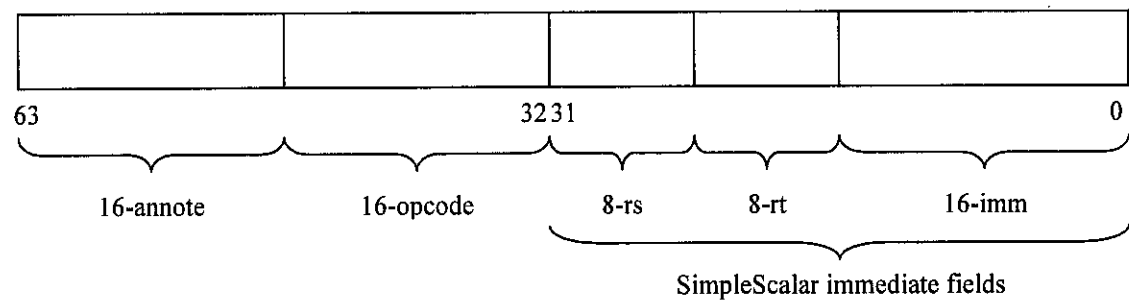


Figure 19: SimpleScalar Immediate Fields (Immediate Format)

Figure 18 and 19 shows the SimpleScalar Immediate Fields. For Register format, there are 4 fields and for Immediate format, there are only 3 fields. In Register format, there are 8 bit register source #1 (RS), 8 bit register source #2 (RT), 8 bit register destination (RD) and 8 bit register shift arithmetic (RU/SHAMT). In Immediate format, there are 8 bit register source (RS), 16 bit immediate value (IMM) and 8 bit register destination (RT).

(Refer to *Appendix IV: decode.vhd* for the source code)

4.7.3. Control

The purpose of Control cycle is to control the movement of data between the register and the memory.

```
ra_bus : out bit_vector(31 downto 0);  
wa_bus : out bit_vector(31 downto 0);  
reg_wrt : out std_logic;  
reg_dst : out std_logic;
```

Source: control.vhd [Appendix IV]

For register design, there are two outputs, which are reg_wrt and reg_dst. reg_wrt is required to control the writing process onto the register while reg_dst is required to select which destination the register will be writing onto during the writeback.

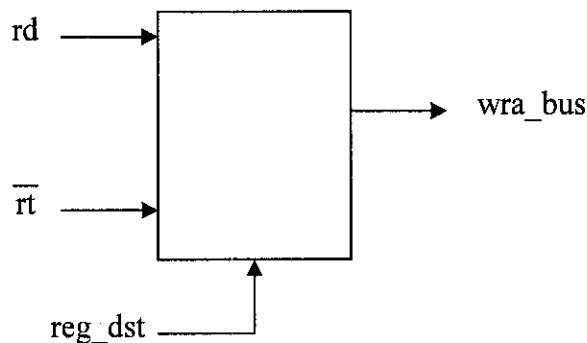


Figure 20: Register Selection

```
wra_bus <= rt when reg_dst='1' else rs;
```

Source: decode.vhd[Appendix IV]

Figure 20 shows the register selection. The purpose is to select which registers, either RS register or RT register during writeback.

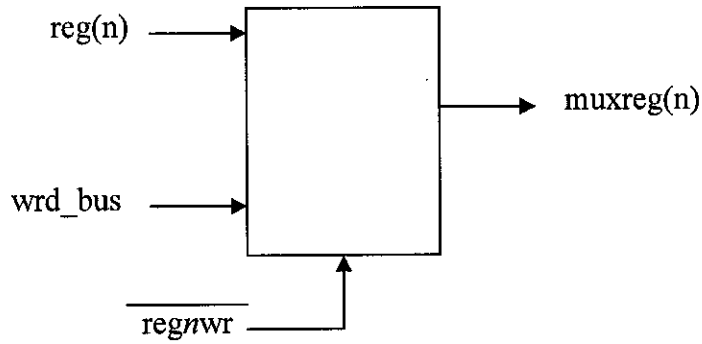


Figure 21: Register-Memory

```
reg0wr <= '1' when ((wra_bus = "00000000") and (reg_wrt='1')) else '0';
reg1wr <= '1' when ((wra_bus = "00000001") and (reg_wrt='1')) else '0';
reg2wr <= '1' when ((wra_bus = "00000010") and (reg_wrt='1')) else '0';

muxreg(0) <= reg(0) when reg0wr='0' else wrd_bus;
muxreg(1) <= reg(1) when reg1wr='0' else wrd_bus;
muxreg(2) <= reg(2) when reg2wr='0' else wrd_bus;
```

Source: decode.vhd[Appendix IV]

This design will select the register output between the intermediate register and the memory. Immediate register is the current register during operation and labeled as reg(n), where n is between 0 to 31.

(Refer to *Appendix IV: control.vhd* for the source code)

4.7.4. Execute

In Execute cycle, the operation of an instruction will be carry out. The selection of which operations will be executing is done by opcode field. In this section, only integer instructions are implemented. Integer instruction

In arithmetic, the operation involves unsigned addition without overflow checking, unsigned subtraction without overflow checking and unsigned multiplication without overflow checking. In logical, the operation involves AND-operation, OR-operation, XOR-operation and NOR-operation. Other operations are shift arithmetic left and shift arithmetic right. Here, an unsigned addition will be explained.

Full adder is a combinational circuit that performs the arithmetic addition of three inputs and produces two outputs. Two of the inputs are two bits to be added while another input is the carry bit from previous adder (if any). Three inputs are denoted by A, B and C_{in} . Two outputs are needed and denoted by S and C_{out} . [9]

The truth table for full adder:

Inputs			Outputs	
A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 3: Full Adder Truth Table

Source: M. Morris Mano, Charles R. Kime, “Logic and Computer Design Fundamentals” [9]

The simplified sum-of-product functions of two outputs are:

$$S = \bar{A} \bar{B} C_{in} + \bar{A} B \bar{C}_{in} + A \bar{B} \bar{C}_{in} + ABC$$

$$C = AB + BC + AC$$

Source: M. Morris Mano, Charles R. Kime, "Logic and Computer Design Fundamentals" [9]

This implementation requires seven AND gates and 2 OR gates. However, the operation can be simplified into the simplest form which is can be expressed as:

$$S = (A \oplus B) \oplus C_{in}$$

$$C = AB + C_{in} (A \oplus B)$$

Source: M. Morris Mano, Charles R. Kime, "Logic and Computer Design Fundamentals" [9]

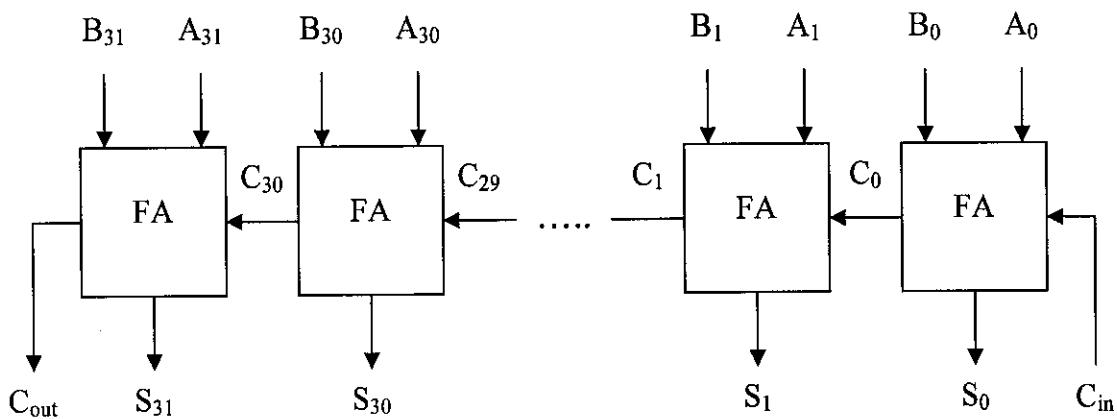


Figure 22: 32 bit Full Adder

Figure 22 shows the visual aid of 32-bit full adder implemented.

```
result(index) := op1(index) xor op2(index) xor carry;
carry := (op1(index) and op2(index)) or (carry and (op1(index) xor op2(index)));
```

Source: *execute.vhd* [Appendix IV]

(Refer to *Appendix IV: execute.vhd* for the source code)

4.7.5. Memory

In Memory cycle, the operations which memory always performs are writing and reading. Writing is when the data is transfer into the memory to be stored. Reading is when the data stored is retrieved out from the memory.

rd_bus stands for read data from the memory, ra_bus is the read address from bus line, wd_bus write data to the memory and wa_bus is the write address to the bus line. rd_bus acts as output while ra_bus, wd_bus and wa_bus acts as inputs to the *memory.vhd*. mem_wrt, mem_red, mem_reg are the inputs from the Control cycle.

```
rd_bus : out bit_vector(31 downto 0);
ra_bus : in bit_vector(31 downto 0);
wd_bus : in bit_vector(31 downto 0);
wa_bus : in bit_vector(31 downto 0);
mem_wrt : in std_logic;
mem_red : in std_logic;
mem_reg : in std_logic;
clock, reset : in std_logic;
```

Source: memory.vhd[Appendix IV]

In this project, the implementation of SimpleScalar memory is not successfully. By part, the data is managed to be read from the memory and store into the given memory location. However, during the Execute cycle, the data is unable to retrieve back.

Given example, a data of 32 bit of X"00001010" is stored at memory location addressing X"00000010". During execution, the data X"00001010" is unable to be retrieve. A further work can be done to investigate this error.

In this section, the information provided the general how the instructions are read from and store into the memory.

Read

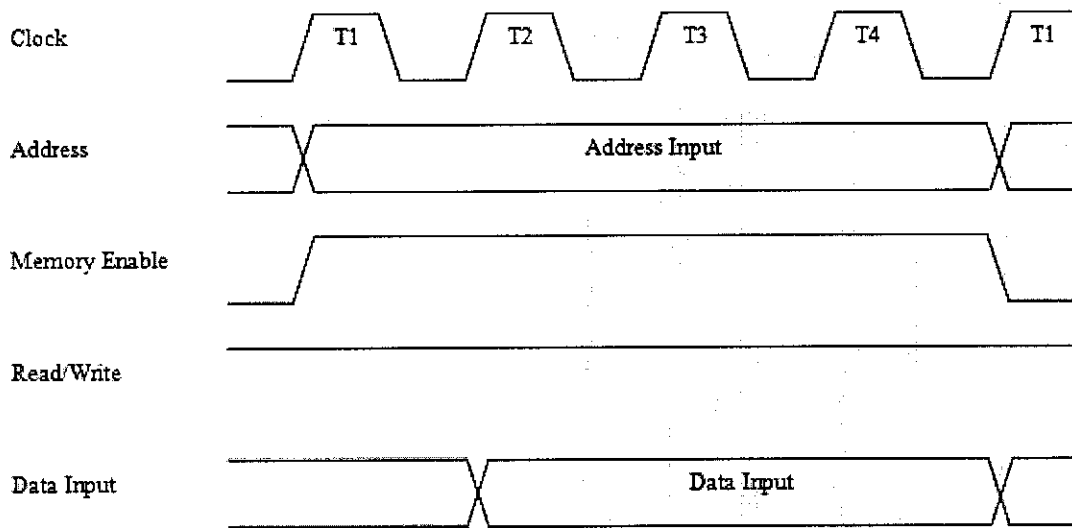


Figure 23: Read Cycle Timing Waveforms

Source: M. Morris Mano, Charles R. Kime, "Logic and Computer Design Fundamentals" [9]

Figure 23 shows the read cycle timing waveforms of general memory design. Steps taken for read operation:

1. Apply the binary address of the desired word into address lines.
2. Active the Read input.

Source: M. Morris Mano, Charles R. Kime, "Logic and Computer Design Fundamentals" [9]

Write

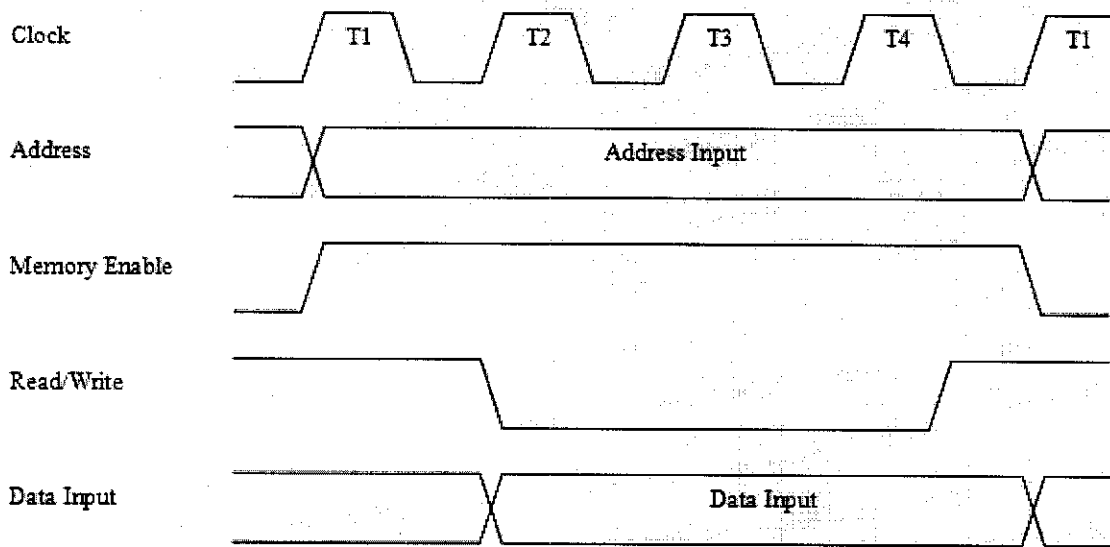


Figure 24: Write Cycle Timing Waveform

Source: M. Morris Mano, Charles R. Kime, "Logic and Computer Design Fundamentals" [9]

Figure 24 shows the write cycle timing waveforms of general memory design. The steps that must be taken for a write operation:

1. Apply the binary address of the desired word into address lines.
2. Apply the data bits that must be stored in memory to the data input lines.
3. Active the Write input.

Source: M. Morris Mano, Charles R. Kime, "Logic and Computer Design Fundamentals" [9]

(Refer to *Appendix IV: memory.vhd* for the source code)

4.8. VHDL Simulation

From the implementation in VHDL, only integer instructions were implemented. They are unsigned addition, unsigned subtraction, unsigned multiplication, AND-operation, OR-operation, XOR-operation, NOR-operation, shift left logical and shift right logical. In the VHDL simulation, assumptions have been made:

- 1. Only functional are tested.
- 2. The RS register stored value of 0x00001010 and RT register stored value of 0x0000100F.

4.8.1. Unsigned Addition

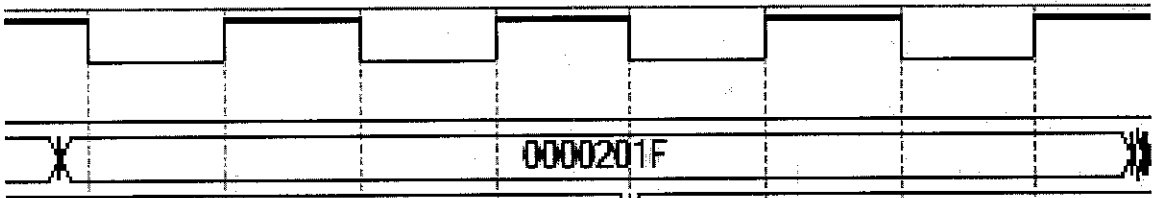


Figure 25: Unsigned Addition Operation

Figure 25 show the result of an addition operation. The operation does not require overflow checking. The addition operation in hexadecimal and binary form:

Hexadecimal	Binary
0x00001010	00000000000000000000000000000000
+ 0x0000100F	+ 00000000000000000000000000000000
<u>0x0000201F</u>	<u>00000000000000000000000000000000</u>

The result of 0x0000201F will be stored at register RD.

4.8.2. OR Operation

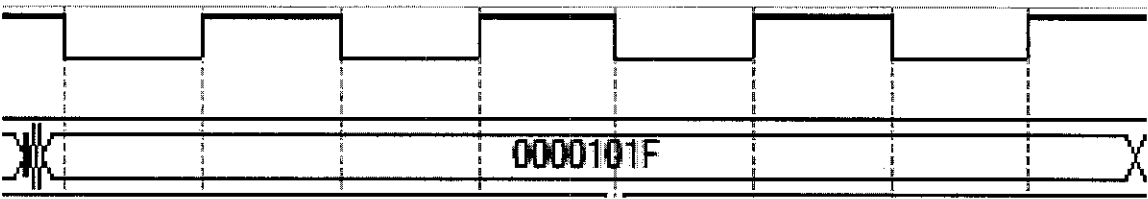


Figure 26: OR Operation

Figure 26 show the result of an OR operation. The OR operation:

Hexadecimal	Binary
0x00001010	00000000000000000000000000000000
0x0000100F	00000000000000000000000000000000
<u>0x0000101F</u>	<u>00000000000000000000000000000000</u>

The result of 0x0000101F will be stored at register RD.

4.8.3. Shift Right Logical

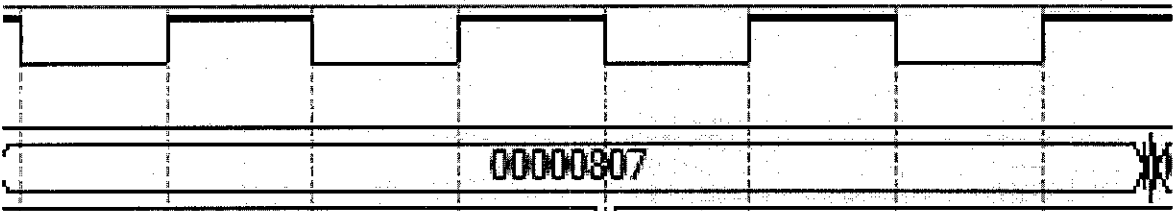


Figure 27: Shift Right Logical

Figure 27 show the shift right logical operation. The operation:

0x0000100F >> 1 00000000000000000000000000000000 >> 1
0x00000807 00000000000000000000000000000000

The result of 0x00000807 will be stored at register RD.

CHAPTER 5: CONCLUSION

From the project that will be done, I hope I will be able to fulfill objectives as described above. This project is well-done and is only able to be functional in VHDL simulation. However, this project is not yet completed within timeframe given. There were several reasons contributing to this cause. They were development progress and synthesizable problem.

Most of the development in this project spent on studying the C source code of SimpleScalar and the VHDL programming. A lot of exercises and examples done in C and VHDL programming before started the project. The lack of source codes available in the Internet makes the project has to be started from scratch. Thus, it takes longer than expected.

Another reason contributes to the project is the VHDL implemented are not synthesizable. During the project, I found a DLX source code, which is similar to MIPS architecture. I had developed the SimpleScalar architecture on it. However, when I tried to compile the source code, it was not synthesizable. Before downloading onto FPGA, it requires the source code to be synthesized first. When it comes to this, the project schedule is delayed.

Only integer instructions were implemented. They are unsigned addition, unsigned subtraction, unsigned multiplication, AND-operation, OR-operation, XOR-operation, NOR-operation, shift left logical and shift right logical.

In this project, the implementation on FPGA was unsuccessful. The code developed is able to be downloaded on the FPGA. However, when I tried to run the FPGA, the board does not working as expected.

CHAPTER 6: RECOMMENDATIONS

Redesign the Control Module

In this project, I have implemented the Control module which is between the Decode and Execute modules. The purpose of this module is to control the data movement between register and memory. However, in this project, this module is not working perfectly. Therefore, for future works, I recommend to redesign the Control module.

Implement Other Instructions

In this project, only integer instructions were implemented. Others instructions such as control instructions, load and store instructions, and floating point instructions are not implemented yet. In the future, I recommend implementing other types of instructions.

Program on FPGA

In this project, the program on FPGA was unsuccessful. The current source code is divided into 5 architectures for ease of use. Each module has own purposes as described earlier. In the future, to program on the FPGA, I recommend to test and program each module separately. This might be able to help them to troubleshoot the source code and solve the problem.

CHAPTER 7: REFERENCES

- [1] Doug Burger, Todd M. Austin, “The SimpleScalar Tool Set, Version 2.0”
- [2] John L. Hennessy & David A. Patterson, “Computer Architecture: A Quantitative Approach”, 2003, Third Edition, Morgan Kaufmann Publishers.
- [3] Vincent P. Heuring & Harry F. Jordan, “Computer Systems Design and Architecture”, 2004, Second Edition, Pearson Prentice Hall.
- [4] SimpleScalar LLC, <http://www.simplescalar.com/>
- [5] SimpleScalar Tools Home Page, <http://www.cs.wisc.edu/~mscalar/simplescalar.html>
- [6] VHDL Tutorial, <http://www.gmvhdl.com/introduc.htm/>
- [7] FPGA as Computing Platform,
<http://www.informit.com/articles/article.asp?p=382614&rl=1>
- [8] Todd M. Austin, “SimpleScalar Hacker’s Guide”
- [9] M. Morris Mano, Charles R. Kime, “Logic and Computer Design Fundamentals”, 2004, Third Edition, Pearson Prentice Hall.
- [10] SimpleScalar Source Code
- [11] Homepage of Crimson Editor, <http://www.crimsoneditor.com/>
- [12] GHDL home page, <http://ghdl.free.fr/>
- [13] Altera, <http://www.altera.com/>
- [14] Cyclone II FPGA Overview,
<http://www.altera.com/products/devices/cyclone2/overview/cy2-overview.html>
- [15] Charles Price, “MIPS Instruction Set”, Revision 3.2, September 2005
- [16] Todd Austin, Eric Larson, Dan Ernst, “SimpleScalar: An Infrastructure for Computer System Modeling”, February 2002, IEEE Journal
- [17] Douglas L. Perry, “VHDL Programming by Example”, 2002, Fourth Edition, McGraw Hill
- [18] H.M. Deitel, P.J. Deitel, “C How To Program”, 2001, Third Edition, Prentice Hall
- [19] The Hamburg VHDL Archive, <http://tams-www.informatik.uni-hamburg.de/vhdl/>
- [20] fpga4fun.com, What are FPGAs?, <http://www.fpga4fun.com>

TABLE OF APPENDICIES

Appendix 1

List of SimpleScalar Instruction Set

Control	j – jump jal – jump and link jr – jump register jalr – jump and link register beq – branch == 0 bne – branch != 0	blez – branch <= 0 bgtz – branch > 0 bltz – branch < 0 bgez – branch >= 0 bct – branch FCC TRUE bcf – branch FCC FALSE
Load/Store	lb – load byte lbu – load byte unsigned lh – load half (short) lhu – load half unsigned lw – load word dlw – load double word ls – load single-precision FP	ld – load double-precision FP sb – store byte sbu – store byte unsigned sw – store word dsw – store double word ss – store single-precision FP sd – store double-precision FP
Integer Arithmetic	add – integer add addu – integer add unsigned sub – integer subtract subu – integer subtract unsigned mult – integer multiply multu – integer multiply unsigned div – integer divide divu – integer divide unsigned and – logical AND	or – logical OR xor – logical XOR nor – logical NOR sll – shift left logical srl – shift right logical sra – shift right arithmetic slt – shift less than sltu – shift less than unsigned
Floating Point Arithmetic	add.s – single-precision (SP) add add.d – double-precision (DP) add sub.s – SP subtract sub.d – DP subtract mult.s – SP multiply mult.d – DP multiply div.s – SP divide div.d – DP divide abs.s – SP absolute value	abs.d – DP absolute value neg.s – SP negation neg.d – DP negation sqrt.s – SP square root sqrt.d – DP square root cvt – int, single, double conversion c.s – SP compare c.d – DP compare

Appendix 2

Planning Schedule

Final Year Project I

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1 FYP Titles														
FYP Briefing														
Selection of FYP Titles														
2 FYP Submissions														
Log Book														
Preliminary Report														
Progress Report														
Interim Report First Draft														
Interim Report Final Draft														
Oral Presentation														
Interim Report Final														
3 Meetings														
Supervisor														
4 Project Development														
Installing the Programs Required														
Learning the Programs Required														
Learning the VHDL														
Programming the VHDL														
Understanding the Source Code														
Source Code -> VHDL														

Final Year Project II

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1 FYP Submissions														
Log Book														
Progress Report														
Interim Report First Draft														
Interim Report Final Draft														
Oral Presentation														
Interim Report Final														
2 Meetings														
Supervisor														
3 Project Development														
Understanding the Source Code														
Programming the VHDL														

Appendix 3

SimpleScalar Instructions

Control Instruction

Instructions	Opcode	Mnemonic	Format
Jump to absolute address	0x01	J	J target
Jump to absolute address and link	0x02	JAL	JAL target
Jump to register address	0x03	JR	JR rs
Jump to register address and link	0x04	JALR	JALR rs
Branch if equal	0x05	BEQ	BEQ rs, rt, offset
Branch if not equal	0x06	BNE	BEQ rs, rt, offset
Branch if less than or equal to zero	0x07	BLEZ	BLEZ rs, offset
Branch if greater than zero	0x08	BGTZ	BGTZ rs, offset
Branch if less than zero	0x09	BLTZ	BLTZ rs, offset
Branch if greater than or equal to zero	0x0a	BGEZ	BGEZ rs, offset
Branch on floating point compare false	0x0b	BC1F	BC1F offset
Branch on floating point compare true	0x0c	BC1T	BC1T offset

Load / Store Instruction

Instructions	Opcode	Mnemonic	Format
Load byte signed, displaced addressing	0x20	LB	LB rt, offset(rs) inc_dec
Load byte signed, indexed addressing	0xc0	LB	LB rt, (rs+rd) inc_dec
Load byte unsigned, displaced addressing	0x22	LBU	LBU rt, offset(rs) inc_dec
Load byte unsigned, indexed addressing	0xc1	LBU	LBU rt, (rs+rd) inc_dec
Load half signed, displaced addressing	0x24	LH	LH rt, offset(rs) inc_dec
Load half signed, indexed addressing	0xc2	LH	LH rt, (rs+rd) inc_dec
Load half unsigned, displaced addressing	0x26	LHU	LHU rt, offset(rs) inc_dec
Load half unsigned, indexed addressing	0xc3	LHU	LHU rt, (rs+rd) inc_dec
Load word, displaced addressing	0x28	LW	LW rt, offset(rs) inc_dec
Load word, indexed addressing	0xc4	LW	LW rt, (rs+rd) inc_dec
Double load word, displaced addressing	0x29	DLW	DLW rt, offset(rs) inc_dec
Double load word, indexed addressing	0xc5	DLW	DLW rt, (rs+rd) inc_dec
Load word into floating point register file, displaced addressing	0x2a	L.S	L.S ft, offset(rs) inc_dec
Load word into floating point register file, indexed addressing	0xc5	L.S	L.S ft, (rs+rd) inc_dec

Integer Instructions

Instructions	Opcode	Mnemonic	Format
Add signed (with overflow check)	0x40	ADD	ADD rd, rs, rt
Add immediate signed (with overflow check)	0x41	ADDI	ADDI rd, rs, rt
Add unsigned (no overflow check)	0x42	ADDU	ADDU rd, rs, rt
Add immediate unsigned (no overflow check)	0x43	ADDIU	ADDIU rd, rs, rt
Subtract signed (with underflow check)	0x44	SUB	SUB rd, rs, rt
Subtract unsigned (no underflow check)	0x45	SUBU	SUBU rd, rs, rt
Multiply signed	0x46	MULT	MULT rs, rt
Multiply unsigned	0x47	MULTU	MULTU rs, rt
Divide signed	0x48	DIV	DIV rs, rt
Divide unsigned	0x49	DIVU	DIVU rs, rt
Move from HI register	0x4a	MFHI	MFHI rd
Move to HI register	0x4b	MTHI	MTHI rs
Move from LO register	0x4c	MFLO	MFLO rd
Move to LO register	0x4d	MTLO	MTLO rs
Logical AND	0x4e	AND	AND rd, rs, rt
Logical AND immediate	0x4f	ANDI	ANDI rd, rt, imm
Logical OR	0x50	OR	OR rd, rs, rt
Logical OR immediate	0x51	ORI	ORI rd, rt, imm
Logical XOR	0x52	XOR	XOR rd, rs, rt
Logical XORI	0x53	XORI	XORI rd, rt, uimm
Logical NOR	0x54	NOR	NOR rd, rs, rt
Shift left logical	0x55	SLL	SLL rd, rt, shamt
Shift left logical variable	0x56	SLLV	SLLV rd, rt, rs
Shift right logical	0x57	SRL	SRL rd, rt, shamt
Shift right logical variable	0x58	SRLV	SRLV rd, rt, rs
Shift right arithmetic	0x59	SRA	SRA rd, rt, shamt
Shift right arithmetic variable	0x59	SRAV	SRAV rd, rt, rs
Set register if less than	0x5b	SLT	SLT rd, rs, rt
Set register if less than immediate	0x5c	SLTI	SLTI rd, rs, imm
Set register if less than unsigned	0x5d	SLTU	SLTU rd, rs, rt
Set register if less than unsigned immediate	0x5d	SLTIU	SLTIU rd, rs, imm

Floating Point Instruction

Instructions	Opcode	Mnemonic	Format
Add floating point, single precision	0x70	ADD.S	ADD.S fd, fs, ft
Add floating point, double precision	0x71	ADD.D	ADD.D fd, fs, ft
Subtract floating point, single precision	0x72	SUB.S	SUB.S fd, fs, ft
Subtract floating point, double precision	0x73	SUB.D	SUB.D fd, fs, ft
Multiply floating point, single precision	0x74	MUL.S	MUL.S fd, fs, ft
Multiply floating point, double precision	0x75	MUL.D	MUL.D fd, fs, ft
Divide floating point, single precision	0x76	DIV.S	DIV.S fd, fs, ft
Divide floating point, double precision	0x77	DIV.D	DIV.D fd, fs, ft
Absolute value, single precision	0x78	ABS.S	ABS.S fd, fs
Absolute value, double precision	0x79	ABS.D	ABS.D fd, fs
Move floating point value, single precision	0x7a	MOV.S	MOV.S fd, fs
Move floating point value, double precision	0x7b	MOV.D	MOV.D fd, fs
Negate floating point value, single precision	0x7c	NEG.S	NEG.S fd, fs
Negate floating point value, double precision	0x7d	NEG.D	NEG.D fd, fs
Convert double precision to single precision	0x80	CVT.S.D	CVT.S.D fd, fs
Convert integer to single precision	0x81	CVT.S.W	CVT.S.W fd, fs
Convert single precision to double precision	0x82	CVT.D.S	CVT.D.S fd, fs
Convert integer to double precision	0x83	CVT.D.W	CVT.D.W fd, fs
Convert single precision to integer	0x84	CVT.W.S	CVT.W.S fd, fs
Convert double precision to integer	0x85	CVT.W.D	CVT.W.D fd, fs
Test if equal, single precision	0x90	C.EQ.S	C.EQ.S fs, ft
Test if equal, double precision	0x91	C.EQ.D	C.EQ.D fs, ft
Test if less than, single precision	0x92	C.LT.S	C.LT.S fs, ft
Test if less than, double precision	0x93	C.LT.D	C.LT.D fs, ft
Test if less than or equal, single precision	0x94	C.LE.S	C.LE.S fs, ft
Test if less than or equal, double precision	0x95	C.LE.D	C.LE.D fs, ft
Square root, single precision	0x96	SQRT.S	SQRT.S fd, fs
Square root, double precision	0x97	SQRT.D	SQRT.D fd, fs

Appendix 4

VHDL Source Code

(SIMPS.vhd)


```
1  -- Abdul Azim bin Abdullah
2  -- Universiti Teknologi PETRONAS
3  -- SIMPS.vhd
4
5  library ieee;
6  use ieee.std_logic_1164.all;
7  use ieee.std_logic_arith.all;
8
9
10 -- SIMPS entity
11
12 entity SIMPS is
13     port(clock, reset : in std_logic;
14           PC_bus : in bit_vector(7 downto 0);
15           inst_out : out bit_vector(31 downto 0));
16 end SIMPS;
17
18
19 -- SIMPS architecture
20
21 architecture structure of SIMPS is
22
23     component fetch
24     port(insta : out bit_vector(31 downto 0);
25           instb : out bit_vector(31 downto 0);
26           PCin : in bit_vector(7 downto 0);
27           clock, reset : in std_logic);
28     end component;
29
30     component decode
31     port(insta : in bit_vector(31 downto 0);
32           instb : in bit_vector(31 downto 0);
33           opcode : out bit_vector(15 downto 0);
34           rsdata_bus : out bit_vector(31 downto 0);
35           rtdata_bus : out bit_vector(31 downto 0);
36           rddata_bus : out bit_vector(31 downto 0);
37           extend : out bit_vector(31 downto 0);
38           wrd_bus : in bit_vector(31 downto 0);
39           reg_wrt : in std_logic;
40           reg_dst : in std_logic;
41           clock, reset : in std_logic);
42     end component;
43
44     component control
45     port(PCin : in bit_vector(7 downto 0);
46           ra_bus : out bit_vector(31 downto 0);
47           wa_bus : out bit_vector(31 downto 0);
48           reg_wrt : out std_logic;
49           reg_dst : out std_logic;
50           mem_wrt : out std_logic;
51           mem_red : out std_logic;
52           mem_reg : out std_logic);
53     end component;
54
55     component execute
56     port(opcode : in bit_vector(15 downto 0);
57           extend : in bit_vector(31 downto 0);
```

```

58     rsdata_bus : in bit_vector(31 downto 0);
59     rtdata_bus : in bit_vector(31 downto 0);
60     rddata_bus : in bit_vector(31 downto 0);
61     data_bus : out bit_vector(31 downto 0);
62     clock : in std_logic);
63 end component;
64
65 component memory
66 port(rd_bus : out bit_vector(31 downto 0);
67      wd_bus : in bit_vector(31 downto 0);
68      ra_bus : in bit_vector(31 downto 0);
69      wa_bus : in bit_vector(31 downto 0);
70      mem_wrt : in std_logic;
71      mem_red : in std_logic;
72      mem_reg : in std_logic;
73      clock, reset : in std_logic);
74 end component;
75
76 signal pc_in : bit_vector(7 downto 0);
77
78 signal insta_bus : bit_vector(31 downto 0);
79 signal instb_bus : bit_vector(31 downto 0);
80
81 signal opcode : bit_vector(15 downto 0);
82 signal rsd_bus : bit_vector(31 downto 0);
83 signal rtd_bus : bit_vector(31 downto 0);
84 signal rdd_bus : bit_vector(31 downto 0);
85
86 signal wrd_bus : bit_vector(31 downto 0);
87
88 signal reg_wrt : std_logic;
89 signal reg_dst : std_logic;
90
91 signal mem_wrt : std_logic;
92 signal mem_red : std_logic;
93 signal mem_reg : std_logic;
94
95 signal ra_bus : bit_vector(31 downto 0);
96 signal wa_bus : bit_vector(31 downto 0);
97
98 signal dat_bus : bit_vector(31 downto 0);
99
100 signal extend : bit_vector(31 downto 0);
101
102 begin
103
104     pc_in <= PC_bus;
105     inst_out <= dat_bus;
106
107     FE : fetch
108
109     port map(insta => insta_bus,
110             instb => instb_bus,
111             PCin => pc_in,
112             clock => clock,
113             reset => reset);
114

```

```
115 DE : decode
116
117 port map(insta => insta_bus,
118         instb => instb_bus,
119         opcode => opcode,
120         rsdata_bus => rsd_bus,
121         rtdata_bus => rtd_bus,
122         rddata_bus => rdd_bus,
123         extend => extend,
124         wrd_bus => wrd_bus,
125         reg_wrt => reg_wrt,
126         reg_dst => reg_dst,
127         clock => clock,
128         reset => reset);
129
130 CT : control
131
132 port map (PCin => pc_in,
133         ra_bus => ra_bus,
134         wa_bus => wa_bus,
135         reg_wrt => reg_wrt,
136         reg_dst => reg_dst,
137         mem_wrt => mem_wrt,
138         mem_red => mem_red,
139         mem_reg => mem_reg);
140
141 EX : execute
142
143 port map (opcode => opcode,
144         extend => extend,
145         rsdata_bus => rsd_bus,
146         rtdata_bus => rtd_bus,
147         rddata_bus => rdd_bus,
148         data_bus => dat_bus,
149         clock => clock);
150
151 ME : memory
152
153 port map(rd_bus => wrd_bus,
154         wd_bus => dat_bus,
155         ra_bus => ra_bus,
156         wa_bus => wa_bus,
157         mem_wrt => mem_wrt,
158         mem_red => mem_red,
159         mem_reg => mem_reg,
160         clock => clock,
161         reset => reset);
162
163 end structure;
```

(fetch.vhd)

```

1  -- Abdul Azim bin Abdullah
2  -- Universiti Teknologi PETRONAS
3  -- fetch.vhd
4
5  library ieee;
6  use ieee.std_logic_1164.all;
7  use ieee.std_logic_arith.all;
8
9
10 -- fetch entity
11
12 entity fetch is
13     port(insta : out bit_vector(31 downto 0); -- inst.a
14           instb : out bit_vector(31 downto 0); -- inst.b
15           PCin : in bit_vector(7 downto 0);    -- PC
16           clock, reset : in std_logic);
17 end fetch;
18
19
20 -- fetch architecture
21
22 architecture behaviour of fetch is
23
24     signal PC : bit_vector(7 downto 0);
25
26     constant mem0 : bit_vector(31 downto 0) := B"000000000000000000000000
00000000"; -- no operation
27     constant mem1 : bit_vector(31 downto 0) := B"000000000000000000000000
00000000";
28     constant mem2 : bit_vector(31 downto 0) := B"000000000000000000000000
01010000"; -- load opcodes ( load $30,7 )
29     constant mem3 : bit_vector(31 downto 0) := B"000000010000000100000000
00000111";
30     constant mem4 : bit_vector(31 downto 0) := B"000000000000000000000000
01010000"; -- load opcodes ( load $20,8 )
31     constant mem5 : bit_vector(31 downto 0) := B"000000010000000100000000
00001000";
32
33     constant mem6 : bit_vector(31 downto 0) := B"000000000000000000000000
01000010"; -- add unsigned
34     constant mem7 : bit_vector(31 downto 0) := B"0000000100000001000001010
00000000";
35     constant mem8 : bit_vector(31 downto 0) := B"000000000000000000000000
01000101"; -- subtract unsigned
36     constant mem9 : bit_vector(31 downto 0) := B"0000000100000001000001010
00000000";
37
38     constant mem10 : bit_vector(31 downto 0) := B"000000000000000000000000
01001110"; -- and
39     constant mem11 : bit_vector(31 downto 0) := B"0000000100000001000001010
00000000";
40     constant mem12 : bit_vector(31 downto 0) := B"000000000000000000000000
01010000"; -- or
41     constant mem13 : bit_vector(31 downto 0) := B"0000000100000001000001010
00000000";
42     constant mem14 : bit_vector(31 downto 0) := B"000000000000000000000000
01010010"; -- xor

```

```

43  constant mem15: bit_vector(31 downto 0) := B"0000000010000001000001010
00000000";
44  constant mem16: bit_vector(31 downto 0) := B"0000000000000000000000000
01010100"; -- nor
45  constant mem17: bit_vector(31 downto 0) := B"0000000010000001000001010
00000000";
46
47  constant mem18: bit_vector(31 downto 0) := B"0000000000000000000000000
01000111"; -- multiply unsigned
48  constant mem19: bit_vector(31 downto 0) := B"0000000010000001000001010
00000000";
49
50  constant mem20: bit_vector(31 downto 0) := B"0000000000000000000000000
01010101"; -- shift left logical
51  constant mem21: bit_vector(31 downto 0) := B"0000000010000001000001010
00000001";
52  constant mem22: bit_vector(31 downto 0) := B"0000000000000000000000000
01010111"; -- shift right logical
53  constant mem23: bit_vector(31 downto 0) := B"0000000010000001000001010
00000001";
54
55  begin
56
57  process
58  begin
59      wait until (clock'event) and (clock='1');
60      if reset='1' then
61          PC <= X"00";
62      else PC <= PCin;
63      end if;
64  end process;
65
66  process(PC)
67  begin
68      case PC is
69          when X"00" =>
70              insta <= mem0;
71              instb <= mem1;
72          when X"04" =>
73              insta <= mem2;
74              instb <= mem3;
75          when X"08" =>
76              insta <= mem4;
77              instb <= mem5;
78          when X"0C" =>
79              insta <= mem6;
80              instb <= mem7;
81          when X"10" =>
82              insta <= mem8;
83              instb <= mem9;
84          when X"14" =>
85              insta <= mem10;
86              instb <= mem11;
87          when X"18" =>
88              insta <= mem12;
89              instb <= mem13;
90          when X"1C" =>

```

```
91     insta <= mem14;
92     instb <= mem15;
93     when X"20" =>
94         insta <= mem16;
95         instb <= mem17;
96     when X"24" =>
97         insta <= mem18;
98         instb <= mem19;
99     when X"28" =>
100         insta <= mem20;
101         instb <= mem21;
102     when X"2C" =>
103         insta <= mem22;
104         instb <= mem23;
105     when others =>
106         insta <= null;
107         instb <= null;
108     end case;
109 end process;
110
111 end behaviour;
```

(decode.vhd)


```
1  -- Abdul Azim bin Abdullah
2  -- Universiti Teknologi PETRONAS
3  -- decode.vhd
4
5  library ieee;
6  use ieee.std_logic_1164.all;
7  use ieee.std_logic_arith.all;
8
9
10 -- decode entity
11
12 entity decode is
13     port (insta : in bit_vector(31 downto 0);           -- inst.a
14           instb : in bit_vector(31 downto 0);           -- inst.b
15           opcode : out bit_vector(15 downto 0);         -- OP
16           rsdata_bus : out bit_vector(31 downto 0);
17           rtdata_bus : out bit_vector(31 downto 0);
18           rddata_bus : out bit_vector(31 downto 0);
19           extend : out bit_vector(31 downto 0);
20           wrd_bus : in bit_vector(31 downto 0);
21           reg_wrt : in std_logic;
22           reg_dst : in std_logic;
23           clock, reset : in std_logic);
24 end decode;
25
26
27 -- decode architecture
28
29 architecture behaviour of decode is
30
31     type reg_array is array (0 to 31) of bit_vector(31 downto 0);
32
33     signal annote : bit_vector(15 downto 0);
34
35     signal rs : bit_vector(7 downto 0);
36     signal rt : bit_vector(7 downto 0);
37     signal rd : bit_vector(7 downto 0);
38     signal ru : bit_vector(7 downto 0);
39     signal imm_v : bit_vector(15 downto 0);
40
41     signal reg : reg_array;
42     signal ireg : reg_array;
43     signal muxreg : reg_array;
44
45     signal reg0wr, reg1wr, reg2wr : std_logic;
46     signal wra_bus : bit_vector(7 downto 0);
47
48 begin
49
50     annote <= insta(31 downto 16);
51     opcode <= insta(15 downto 0);
52
53     rs <= instb(31 downto 24);
54     rt <= instb(23 downto 16);
55     rd <= instb(15 downto 8);
56     ru <= instb(7 downto 0);
57
```

```

58  imm_v <= instb(15 downto 0);
59
60  with rs select
61  rsdata_bus <= reg(0) when "00000000",
62                  reg(1) when "00000001",
63                  reg(2) when "00000010",
64                  reg(3) when "00000011",
65                  reg(4) when "00000100",
66                  reg(5) when "00000101",
67                  reg(6) when "00000110",
68                  reg(7) when "00000111",
69                  reg(8) when "00001000",
70                  reg(9) when "00001001",
71                  reg(10) when "00001010",
72                  reg(11) when "00001011",
73                  reg(12) when "00001100",
74                  reg(13) when "00001101",
75                  reg(14) when "00001110",
76                  reg(15) when "00001111",
77                  reg(16) when "00010000",
78                  reg(17) when "00010001",
79                  reg(18) when "00010010",
80                  reg(19) when "00010011",
81                  reg(20) when "00010100",
82                  reg(21) when "00010101",
83                  reg(22) when "00010110",
84                  reg(23) when "00010111",
85                  reg(24) when "00011000",
86                  reg(25) when "00011001",
87                  reg(26) when "00011010",
88                  reg(27) when "00011011",
89                  reg(28) when "00011100",
90                  reg(29) when "00011101",
91                  reg(30) when "00011110",
92                  reg(31) when "00011111",
93                  X"FFFFFFF" when others;
94
95  with rt select
96  rtdata_bus <= reg(0) when "00000000",
97                  reg(1) when "00000001",
98                  reg(2) when "00000010",
99                  reg(3) when "00000011",
100                 reg(4) when "00000100",
101                 reg(5) when "00000101",
102                 reg(6) when "00000110",
103                 reg(7) when "00000111",
104                 reg(8) when "00001000",
105                 reg(9) when "00001001",
106                 reg(10) when "00001010",
107                 reg(12) when "00001100",
108                 reg(13) when "00001101",
109                 reg(14) when "00001110",
110                 reg(15) when "00001111",
111                 reg(16) when "00010000",
112                 reg(17) when "00010001",
113                 reg(18) when "00010010",
114                 reg(19) when "00010011",

```

```

115         reg(20) when "00010100",
116         reg(21) when "00010101",
117         reg(22) when "00010110",
118         reg(23) when "00010111",
119         reg(24) when "00011000",
120         reg(25) when "00011001",
121         reg(26) when "00011010",
122         reg(27) when "00011011",
123         reg(28) when "00011100",
124         reg(29) when "00011101",
125         reg(30) when "00011110",
126         reg(31) when "00011111",
127         X"FFFFFFFF" when others;
128
129 with rd select
130 rddata_bus <= reg(0) when "00000000",
131               reg(1) when "00000001",
132               reg(2) when "00000010",
133               reg(3) when "00000011",
134               reg(4) when "00000100",
135               reg(5) when "00000101",
136               reg(6) when "00000110",
137               reg(7) when "00000111",
138               reg(8) when "00001000",
139               reg(9) when "00001001",
140               reg(10) when "00001010",
141               reg(11) when "00001011",
142               reg(12) when "00001100",
143               reg(13) when "00001101",
144               reg(14) when "00001110",
145               reg(15) when "00001111",
146               reg(16) when "00010000",
147               reg(17) when "00010001",
148               reg(18) when "00010010",
149               reg(19) when "00010011",
150               reg(20) when "00010100",
151               reg(21) when "00010101",
152               reg(22) when "00010110",
153               reg(23) when "00010111",
154               reg(24) when "00011000",
155               reg(25) when "00011001",
156               reg(26) when "00011010",
157               reg(27) when "00011011",
158               reg(28) when "00011100",
159               reg(29) when "00011101",
160               reg(30) when "00011110",
161               reg(31) when "00011111",
162               X"FFFFFFFF" when others;
163
164 -- initial value
165
166 ireg(0) <= X"00000000";
167 ireg(1) <= X"00000000";
168 ireg(2) <= X"00000000";
169 ireg(3) <= X"00000000";
170 ireg(4) <= X"00000000";
171 ireg(5) <= X"00000000";

```

```

172  ireg(6) <= X"00000000";
173  ireg(7) <= X"00000000";
174  ireg(8) <= X"00000000";
175  ireg(9) <= X"00000000";
176  ireg(10) <= X"00000000";
177  ireg(11) <= X"00000000";
178  ireg(12) <= X"00000000";
179  ireg(13) <= X"00000000";
180  ireg(14) <= X"00000000";
181  ireg(15) <= X"00000000";
182  ireg(16) <= X"00000000";
183  ireg(17) <= X"00000000";
184  ireg(18) <= X"00000000";
185  ireg(19) <= X"00000000";
186  ireg(20) <= X"00000000";
187  ireg(21) <= X"00000000";
188  ireg(22) <= X"00000000";
189  ireg(23) <= X"00000000";
190  ireg(24) <= X"00000000";
191  ireg(25) <= X"00000000";
192  ireg(26) <= X"00000000";
193  ireg(27) <= X"00000000";
194  ireg(28) <= X"00000000";
195  ireg(29) <= X"00000000";
196  ireg(30) <= X"00000000";
197  ireg(31) <= X"00000000";
198
199  wra_bus <= rt when reg_dst='1' else rs;
200
201  reg0wr <= '1' when ((wra_bus = "00000000") and (reg_wrt='1')) else '0';
202  reg1wr <= '1' when ((wra_bus = "00000001") and (reg_wrt='1')) else '0';
203  reg2wr <= '1' when ((wra_bus = "00000010") and (reg_wrt='1')) else '0';
204
205  muxreg(0) <= X"00000000" when reg0wr='0' else wrd_bus;
206  muxreg(1) <= X"00001010" when reg1wr='0' else wrd_bus;
207  muxreg(2) <= X"0000100F" when reg2wr='0' else wrd_bus;
208
209  extend(15 downto 0) <= imm_v;
210  extend(31 downto 16) <= X"FFFF" when imm_v(15)='1' else X"0000";
211
212  process
213  begin
214      wait until (clock'event) and (clock='1');
215      if reset='1' then
216          reg(0) <= ireg(0);
217          reg(1) <= ireg(1);
218          reg(2) <= ireg(2);
219          reg(3) <= ireg(3);
220          reg(4) <= ireg(4);
221          reg(5) <= ireg(5);
222          reg(6) <= ireg(6);
223          reg(7) <= ireg(7);
224          reg(8) <= ireg(8);
225          reg(9) <= ireg(9);

```

```
226     reg(10) <= ireg(10);
227     reg(11) <= ireg(11);
228     reg(12) <= ireg(12);
229     reg(13) <= ireg(13);
230     reg(14) <= ireg(14);
231     reg(15) <= ireg(15);
232     reg(16) <= ireg(16);
233     reg(17) <= ireg(17);
234     reg(18) <= ireg(18);
235     reg(19) <= ireg(19);
236     reg(20) <= ireg(20);
237     reg(21) <= ireg(21);
238     reg(22) <= ireg(22);
239     reg(23) <= ireg(23);
240     reg(24) <= ireg(24);
241     reg(25) <= ireg(25);
242     reg(26) <= ireg(26);
243     reg(27) <= ireg(27);
244     reg(28) <= ireg(28);
245     reg(29) <= ireg(29);
246     reg(30) <= ireg(30);
247     reg(31) <= ireg(31);
248   else
249     reg(0) <= muxreg(0);
250     reg(1) <= muxreg(1);
251     reg(2) <= muxreg(2);
252     reg(3) <= muxreg(3);
253     reg(4) <= muxreg(4);
254     reg(5) <= muxreg(5);
255     reg(6) <= muxreg(6);
256     reg(7) <= muxreg(7);
257     reg(8) <= muxreg(8);
258     reg(9) <= muxreg(9);
259     reg(10) <= muxreg(10);
260     reg(11) <= muxreg(11);
261     reg(12) <= muxreg(12);
262     reg(13) <= muxreg(13);
263     reg(14) <= muxreg(14);
264     reg(15) <= muxreg(15);
265     reg(16) <= muxreg(16);
266     reg(17) <= muxreg(17);
267     reg(18) <= muxreg(18);
268     reg(19) <= muxreg(19);
269     reg(20) <= muxreg(20);
270     reg(21) <= muxreg(21);
271     reg(22) <= muxreg(22);
272     reg(23) <= muxreg(23);
273     reg(24) <= muxreg(24);
274     reg(25) <= muxreg(25);
275     reg(26) <= muxreg(26);
276     reg(27) <= muxreg(27);
277     reg(28) <= muxreg(28);
278     reg(29) <= muxreg(29);
279     reg(30) <= muxreg(30);
280     reg(31) <= muxreg(31);
281   end if;
282 end process;
```

```
283  
284 end behaviour;
```

(control.vhd)

```
1 -- Abdul Azim bin Abdullah
2 -- Universiti Teknologi PETRONAS
3 -- control.vhd
4
5 library ieee;
6 use ieee.std_logic_1164.all;
7 use ieee.std_logic_arith.all;
8
9
10 -- control entity
11
12 entity control is
13     port(PCin : in bit_vector(7 downto 0);
14           ra_bus : out bit_vector(31 downto 0);
15           wa_bus : out bit_vector(31 downto 0);
16           reg_wrt : out std_logic;
17           reg_dst : out std_logic;
18           mem_wrt : out std_logic;
19           mem_red : out std_logic;
20           mem_reg : out std_logic);
21 end control;
22
23
24 -- control architecture
25
26 architecture behaviour of control is
27
28 begin
29
30     process (PCin)
31     begin
32         case PCin is
33             when X"04" =>
34                 reg_wrt <= '0';
35                 reg_dst <= '0';
36                 mem_wrt <= '0';
37                 mem_red <= '1';
38                 mem_reg <= '0';
39                 ra_bus <= X"00000001";
40             when X"08" =>
41                 reg_wrt <= '0';
42                 reg_dst <= '1';
43                 mem_wrt <= '0';
44                 mem_red <= '1';
45                 mem_reg <= '0';
46                 ra_bus <= X"00000010";
47             when X"0C" =>
48                 reg_wrt <= '0';
49                 reg_dst <= '1';
50                 mem_wrt <= '0';
51                 mem_red <= '0';
52                 mem_reg <= '0';
53                 ra_bus <= X"00000010";
54             when X"10" =>
55                 reg_wrt <= '0';
56                 reg_dst <= '1';
57                 mem_wrt <= '0';
```



```
58     mem_red <= '0';
59     mem_reg <= '0';
60     ra_bus <= X"00000010";
61     when X"14" =>
62         reg_wrt <= '0';
63         reg_dst <= '1';
64         mem_wrt <= '0';
65         mem_red <= '0';
66         mem_reg <= '0';
67         ra_bus <= X"00000010";
68     when X"18" =>
69         reg_wrt <= '0';
70         reg_dst <= '1';
71         mem_wrt <= '0';
72         mem_red <= '0';
73         mem_reg <= '0';
74         ra_bus <= X"00000010";
75     when X"1C" =>
76         reg_wrt <= '0';
77         reg_dst <= '1';
78         mem_wrt <= '0';
79         mem_red <= '0';
80         mem_reg <= '0';
81         ra_bus <= X"00000010";
82     when X"20" =>
83         reg_wrt <= '0';
84         reg_dst <= '1';
85         mem_wrt <= '0';
86         mem_red <= '0';
87         mem_reg <= '0';
88         ra_bus <= X"00000010";
89     when X"24" =>
90         reg_wrt <= '0';
91         reg_dst <= '1';
92         mem_wrt <= '0';
93         mem_red <= '0';
94         mem_reg <= '0';
95         ra_bus <= X"00000010";
96     when X"28" =>
97         reg_wrt <= '0';
98         reg_dst <= '1';
99         mem_wrt <= '0';
100        mem_red <= '0';
101        mem_reg <= '0';
102        ra_bus <= X"00000010";
103    when X"2C" =>
104        reg_wrt <= '0';
105        reg_dst <= '1';
106        mem_wrt <= '0';
107        mem_red <= '0';
108        mem_reg <= '0';
109        ra_bus <= X"00000010";
110
111    when others =>
112        reg_wrt <= '1';
113        reg_dst <= '1';
114        mem_wrt <= '0';
```

```
115     mem_red <= '1';
116     mem_reg <= '0';
117     ra_bus <= X"00000000";
118     end case;
119 end process;
120
121 end behaviour;
```

(execute.vhd)

```
1 -- Abdul Azim bin Abdullah
2 -- Universiti Teknologi PETRONAS
3 -- execute.vhd
4
5
6 -- bv_arithmetic package
7
8 package bv_arithmetic is
9
10 function "+" (bv1, bv2 : in bit_vector) return bit_vector;
11
12 function "-" (bv1, bv2 : in bit_vector) return bit_vector;
13
14 function "-" (bv : in bit_vector) return bit_vector;
15
16 function "*" (bv1, bv2 : in bit_vector) return bit_vector;
17
18 procedure bv_multu (bv1, bv2 : in bit_vector;
19                    bv_result : out bit_vector;
20                    overflow : out boolean);
21
22 procedure bv_addu (bv1, bv2 : in bit_vector;
23                  bv_result : out bit_vector;
24                  overflow : out boolean);
25
26 procedure bv_add (bv1, bv2 : in bit_vector;
27                  bv_result : out bit_vector;
28                  overflow : out boolean);
29
30 procedure bv_addu (bv1, bv2 : in bit_vector;
31                  bv_result : out bit_vector);
32
33 procedure bv_sub (bv1, bv2 : in bit_vector;
34                  bv_result : out bit_vector;
35                  overflow : out boolean);
36
37 procedure bv_subu (bv1, bv2 : in bit_vector;
38                  bv_result : out bit_vector);
39
40 procedure bv_and (bv1, bv2 : in bit_vector;
41                  bv_result : out bit_vector);
42
43 procedure bv_or (bv1, bv2 : in bit_vector;
44                 bv_result : out bit_vector);
45
46 procedure bv_xor (bv1, bv2 : in bit_vector;
47                  bv_result : out bit_vector);
48
49 procedure bv_nor (bv1, bv2 : in bit_vector;
50                  bv_result : out bit_vector);
51
52 function bv_sll (bv : in bit_vector;
53                 shift_count : in natural) return bit_vector;
54
55 function bv_srl (bv : in bit_vector;
56                 shift_count : in natural) return bit_vector;
57
```

```

58 end bv_arithmetic;
59
60
61 -- bv_arithmetic package body
62
63 package body bv_arithmetic is
64
65     function "+" (bv1, bv2 : in bit_vector) return bit_vector is
66         variable op1      : bit_vector(1 to bv1'length);
67         variable op2      : bit_vector(1 to bv2'length);
68         variable result   : bit_vector(1 to bv1'length);
69         variable carry_in : bit;
70         variable carry_out : bit := '0';
71     begin
72         op1 := bv1;
73         op2 := bv2;
74         for index in result'reverse_range loop
75             carry_in := carry_out;
76             result(index) := op1(index) xor op2(index) xor carry_in;
77             carry_out := (op1(index) and op2(index)) or (carry_in and
78 (op1(index) xor op2(index)));
79         end loop;
80         return result;
81     end "+";
82
83     function "-" (bv1, bv2 : in bit_vector) return bit_vector is
84         variable op1      : bit_vector(1 to bv1'length);
85         variable op2      : bit_vector(1 to bv2'length);
86         variable result   : bit_vector(1 to bv1'length);
87         variable carry_in : bit;
88         variable carry_out : bit := '1';
89     begin
90         op1 := bv1;
91         op2 := bv2;
92         for index in result'reverse_range loop
93             carry_in := carry_out;
94             result(index) := op1(index) xor (not op2(index)) xor carry_in;
95             carry_out := (op1(index) and (not op2(index))) or (carry_i
96 n and (op1(index) xor (not op2(index))));
97         end loop;
98         return result;
99     end "-";
100
101     function "-" (bv : in bit_vector) return bit_vector is
102         constant zero      : bit_vector(bv'range) := (others => '0');
103     begin
104         return zero - bv;
105     end "-";
106
107     function "*" (bv1, bv2 : in bit_vector) return bit_vector is
108         variable negative_result : boolean;
109         variable op1             : bit_vector(bv1'range) := bv1;
110         variable op2             : bit_vector(bv2'range) := bv2;
111         variable result          : bit_vector(bv1'range);
112     begin
113         negative_result := (op1(op1'left) = '1') xor (op2(op2'left) = '1');
114         if (op1(op1'left) = '1') then

```

```

113     op1 := - bv1;
114 end if;
115 if (op2(op2'left) = '1') then
116     op2 := - bv2;
117 end if;
118 bv_multu(op1, op2, result);
119 if (negative_result) then
120     result := - result;
121 end if;
122 return result;
123 end "*";
124
125 procedure bv_multu (bv1, bv2 : in bit_vector;
126                     bv_result : out bit_vector;
127                     overflow : out boolean) is
128     constant bv_length      : natural := bv1'length;
129     constant accum_length   : natural := bv_length * 2;
130     constant zero           : bit_vector(accum_length-1 downto bv_
length) := (others => '0');
131     variable accum          : bit_vector(accum_length-1 downto 0);
132     variable addu_overflow  : boolean;
133     variable carry         : bit;
134 begin
135     accum(bv_length-1 downto 0) := bv1;
136     accum(accum_length-1 downto bv_length) := zero;
137     for count in 1 to bv_length loop
138         if (accum(0) = '1') then
139             bv_addu( accum(accum_length-1 downto bv_length), bv2,
140                     accum(accum_length-1 downto bv_length), addu_overflow)
;
141             carry := bit'val(boolean'pos(addu_overflow));
142         else
143             carry := '0';
144         end if;
145         accum := carry & accum(accum_length-1 downto 1);
146     end loop;
147     bv_result := accum(bv_length-1 downto 0);
148     overflow := accum(accum_length-1 downto bv_length) /= zero;
149 end bv_multu;
150
151 procedure bv_addu (bv1, bv2 : in bit_vector;
152                   bv_result : out bit_vector;
153                   overflow : out boolean) is
154     variable op1      : bit_vector(1 to bv1'length);
155     variable op2      : bit_vector(1 to bv2'length);
156     variable result   : bit_vector(1 to bv_result'length);
157     variable carry    : bit := '0';
158 begin
159     op1 := bv1;
160     op2 := bv2;
161     for index in result'reverse_range loop
162         result(index) := op1(index) xor op2(index) xor carry;
163         carry := (op1(index) and op2(index)) or (carry and (op1(i
ndex) xor op2(index)));
164     end loop;
165     bv_result := result;
166     overflow := carry = '1';

```

```

167 end bv_addu;
168
169 procedure bv_add (bv1, bv2 : in bit_vector;
170                  bv_result : out bit_vector;
171                  overflow : out boolean) is
172     variable op1      : bit_vector(1 to bv1'length);
173     variable op2      : bit_vector(1 to bv2'length);
174     variable result   : bit_vector(1 to bv_result'length);
175     variable carry_in : bit;
176     variable carry_out : bit := '0';
177 begin
178     op1 := bv1;
179     op2 := bv2;
180     for index in result'reverse_range loop
181         carry_in := carry_out;
182         result(index) := op1(index) xor op2(index) xor carry_in;
183         carry_out := (op1(index) and op2(index)) or (carry_in and (op
184         1(index) xor op2(index)));
185     end loop;
186     bv_result := result;
187     overflow := carry_out /= carry_in;
188     --overflow := true;
189 end bv_add;
190
191 procedure bv_addu (bv1, bv2 : in bit_vector;
192                   bv_result : out bit_vector) is
193     variable op1 : bit_vector(1 to bv1'length);
194     variable op2 : bit_vector(1 to bv2'length);
195     variable result : bit_vector(1 to bv_result'length);
196     variable carry : bit := '0';
197 begin
198     op1 := bv1;
199     op2 := bv2;
200     for index in result'reverse_range loop
201         result(index) := op1(index) xor op2(index) xor carry;
202         carry := (op1(index) and op2(index)) or (carry and (op1(index) xo
203         r op2(index)));
204     end loop;
205     bv_result := result;
206 end bv_addu;
207
208 procedure bv_sub (bv1, bv2 : in bit_vector;
209                  bv_result : out bit_vector;
210                  overflow : out boolean) is
211     variable op1 : bit_vector(1 to bv1'length);
212     variable op2 : bit_vector(1 to bv2'length);
213     variable result : bit_vector(1 to bv_result'length);
214     variable carry_in : bit;
215     variable carry_out : bit := '1';
216 begin
217     op1 := bv1;
218     op2 := bv2;
219     for index in result'reverse_range loop
220         carry_in := carry_out;
221         result(index) := op1(index) xor (not op2(index)) xor carry_in;
222         carry_out := (op1(index) and (not op2(index))) or (carry_in and (
223         op1(index) xor (not op2(index))));

```

```
221     end loop;
222     bv_result := result;
223     overflow := carry_out /= carry_in;
224 end bv_sub;
225
226 procedure bv_subu (bv1, bv2 : in bit_vector;
227                   bv_result : out bit_vector) is
228     variable op1 : bit_vector(1 to bv1'length);
229     variable op2 : bit_vector(1 to bv2'length);
230     variable result : bit_vector(1 to bv_result'length);
231     variable borrow : bit := '0';
232 begin
233     op1 := bv1;
234     op2 := bv2;
235     for index in result'reverse_range loop
236         result(index) := op1(index) xor op2(index) xor borrow;
237         borrow := (not op1(index) and op2(index)) or (borrow and not (op1
(index) xor op2(index)));
238     end loop;
239     bv_result := result;
240 end bv_subu;
241
242 procedure bv_and (bv1, bv2 : in bit_vector;
243                  bv_result : out bit_vector) is
244     variable op1 : bit_vector(1 to bv1'length);
245     variable op2 : bit_vector(1 to bv2'length);
246     variable result : bit_vector(1 to bv_result'length);
247 begin
248     op1 := bv1;
249     op2 := bv2;
250     for index in result'reverse_range loop
251         result(index) := op1(index) and op2(index);
252     end loop;
253     bv_result := result;
254 end bv_and;
255
256 procedure bv_or (bv1, bv2 : in bit_vector;
257                 bv_result : out bit_vector) is
258     variable op1 : bit_vector(1 to bv1'length);
259     variable op2 : bit_vector(1 to bv2'length);
260     variable result : bit_vector(1 to bv_result'length);
261 begin
262     op1 := bv1;
263     op2 := bv2;
264     for index in result'reverse_range loop
265         result(index) := op1(index) or op2(index);
266     end loop;
267     bv_result := result;
268 end bv_or;
269
270 procedure bv_xor (bv1, bv2 : in bit_vector;
271                  bv_result : out bit_vector) is
272     variable op1 : bit_vector(1 to bv1'length);
273     variable op2 : bit_vector(1 to bv2'length);
274     variable result : bit_vector(1 to bv_result'length);
275 begin
276     op1 := bv1;
```



```

277     op2 := bv2;
278     for index in result'reverse_range loop
279         result(index) := op1(index) xor op2(index);
280     end loop;
281     bv_result := result;
282 end bv_xor;
283
284 procedure bv_nor (bv1, bv2 : in bit_vector;
285                  bv_result : out bit_vector) is
286     variable op1 : bit_vector(1 to bv1'length);
287     variable op2 : bit_vector(1 to bv2'length);
288     variable result : bit_vector(1 to bv_result'length);
289 begin
290     op1 := bv1;
291     op2 := bv2;
292     for index in result'reverse_range loop
293         result(index) := not (op1(index) or op2(index));
294     end loop;
295     bv_result := result;
296 end bv_nor;
297
298 function bv_sll (bv : in bit_vector;
299                shift_count : in natural) return bit_vector is
300     constant bv_length : natural := bv'length;
301     constant actual_shift_count : natural := shift_count mod bv_length;
302     variable bv_norm : bit_vector(1 to bv_length);
303     variable result : bit_vector(1 to bv_length) := (others => '0');
304 begin
305     bv_norm := bv;
306     result(1 to bv_length - actual_shift_count) := bv_norm(actual_shift
307 _count + 1 to bv_length);
308     return result;
309 end bv_sll;
310
311 function bv_srl (bv : in bit_vector;
312                shift_count : in natural) return bit_vector is
313     constant bv_length : natural := bv'length;
314     constant actual_shift_count : natural := shift_count mod bv_length;
315     variable bv_norm : bit_vector(1 to bv_length);
316     variable result : bit_vector(1 to bv_length) := (others => '0');
317 begin
318     bv_norm := bv;
319     result(actual_shift_count + 1 to bv_length) := bv_norm(1 to bv_leng
320 th - actual_shift_count);
321     return result;
322 end bv_srl;
323 end bv_arithmetic;
324
325 library ieee;
326 use ieee.std_logic_1164.all;
327 use ieee.std_logic_arith.all;
328 use work.bv_arithmetic.all;
329
330 -- execute entity
331

```

```

332 entity execute is
333     port(opcode : in bit_vector(15 downto 0);      -- OP
334           extend : in bit_vector(31 downto 0);
335           rsdata_bus : in bit_vector(31 downto 0);  -- rs data bus
336           rtdata_bus : in bit_vector(31 downto 0);  -- rt data bus
337           rddata_bus : in bit_vector(31 downto 0);  -- rd data bus
338           data_bus : out bit_vector(31 downto 0);
339           clock : in std_logic);
340 end execute;
341
342
343 -- execute architecture
344
345 architecture behaviour of execute is
346
347     signal op_impl : bit_vector(15 downto 0);
348
349     constant op_nop : bit_vector(15 downto 0) := X"0000";      -- NOP
350
351     constant op_lw : bit_vector(15 downto 0) := X"0028";      --
352
353     -- arithmetic operations
354     constant op_add : bit_vector(15 downto 0) := X"0040";      -- ADD r
d, rs, rt // add signed (with overflow check) // rs + rt -> rd
355     constant op_addu : bit_vector(15 downto 0) := X"0042";      -- ADDU
rd, rs, rt // add unsigned (without overflow check) // rs + rt -> rd
356     constant op_sub : bit_vector(15 downto 0) := X"0044";      -- SUB r
d, rs, rt // sub signed (with underflow check) // rs - rt -> rd
357     constant op_subu : bit_vector(15 downto 0) := X"0045";      -- SUBU
rd, rs, rt // sub unsigned (without underflow check) // rs - rt -> rd
358
359     constant op_multu : bit_vector(15 downto 0) := X"0047";      -- MULTU
rd, rs, rt
360
361     -- logical operations
362     constant op_and : bit_vector(15 downto 0) := X"004E";      -- AND r
d, rs, rt // rs & rt -> rd
363     constant op_or : bit_vector(15 downto 0) := X"0050";      -- OR rd
, rs, rt // rs | rt -> rd
364     constant op_xor : bit_vector(15 downto 0) := X"0052";      -- XOR r
d, rs, rt // rs ^ rt -> rd
365     constant op_nor : bit_vector(15 downto 0) := X"0054";      -- NOR r
d, rs, rt // ~(rs | rt) -> rd
366
367     constant op_sll : bit_vector(15 downto 0) := X"0055";      -- SLL r
d, rt, shamt // rt << shamt -> rd
368     constant op_srl : bit_vector(15 downto 0) := X"0057";      -- SRL r
d, rt, shamt // rt >> shamt -> rd
369
370 begin
371
372     op_impl <= opcode;
373
374     process
375
376     variable rs_data : bit_vector(31 downto 0);
377     variable rt_data : bit_vector(31 downto 0);

```

```
378 variable rd_data : bit_vector(31 downto 0);
379
380 begin
381   wait until (clock'event) and (clock='1');
382
383   rs_data := rsdata_bus;
384   rt_data := rtdata_bus;
385   rd_data := rddata_bus;
386
387   case op_impl is
388
389   when op_nop =>
390     data_bus <= X"00000000";
391
392   when op_lw =>
393     data_bus <= extend;
394
395   when op_add =>
396     bv_add(rs_data,rt_data,rd_data);
397     data_bus <= rd_data;
398
399   when op_addu =>
400     bv_addu(rs_data,rt_data,rd_data);
401     data_bus <= rd_data;
402
403   when op_sub =>
404     bv_sub(rs_data,rt_data,rd_data);
405     data_bus <= rd_data;
406
407   when op_subu =>
408     bv_subu(rs_data,rt_data,rd_data);
409     data_bus <= rd_data;
410
411   when op_multu =>
412     bv_multu(rs_data,rt_data,rd_data);
413     data_bus <= rd_data;
414
415   when op_and =>
416     bv_and(rs_data,rt_data,rd_data);
417     data_bus <= rd_data;
418
419   when op_or =>
420     bv_or(rs_data,rt_data,rd_data);
421     data_bus <= rd_data;
422
423   when op_xor =>
424     bv_xor(rs_data,rt_data,rd_data);
425     data_bus <= rd_data;
426
427   when op_nor =>
428     bv_nor(rs_data,rt_data,rd_data);
429     data_bus <= rd_data;
430
431   when op_sll =>
432     data_bus <= bv_sll(rt_data, 1);
433
434   when op_srl =>
```

```
435     data_bus <= bv_srl(rt_data, 1);
436
437     when others =>
438         data_bus <= null;
439
440     end case;
441 end process;
442
443 end behaviour;
```

(memory.vhd)

```

1  -- Abdul Azim bin Abdullah
2  -- Universiti Teknologi PETRONAS
3  -- memory.vhd
4
5  library ieee;
6  use ieee.std_logic_1164.all;
7  use ieee.std_logic_arith.all;
8
9  entity memory is
10     port(rd_bus : out bit_vector(31 downto 0);
11           wd_bus : in bit_vector(31 downto 0);
12           ra_bus : in bit_vector(31 downto 0);
13           wa_bus : in bit_vector(31 downto 0);
14           mem_wrt : in std_logic;
15           mem_red : in std_logic;
16           mem_reg : in std_logic;
17           clock, reset : in std_logic);
18 end memory;
19
20 architecture behaviour of memory is
21
22     signal muxout : bit_vector(31 downto 0);
23     signal address : bit_vector(31 downto 0);
24
25     signal mem0, mem1, mem2 : bit_vector(31 downto 0);
26     signal imem0, imem1, imem2 : bit_vector(31 downto 0);
27     signal muxmem0, muxmem1, muxmem2 : bit_vector(31 downto 0);
28
29     signal mem0wrt, mem1wrt, mem2wrt : std_logic;
30
31 begin
32
33     imem0 <= X"00000000A";
34     imem1 <= X"00000000B";
35     imem2 <= X"00000000C";
36
37     address <= ra_bus;
38     muxout <= mem0 when address=X"00000000" else
39             mem1 when address=X"00000001" else
40             mem2 when address=X"00000010";
41
42     rd_bus <= muxout when mem_red='1' else
43             X"FFFFFFFF";
44
45     --mem0wrt <= '1' when ((mem_wrt='1') and (wa_bus=X"00000000")) else '
46     0';
47     --mem1wrt <= '1' when ((mem_wrt='1') and (wa_bus=X"00000001")) else '
48     0';
49     --mem2wrt <= '1' when ((mem_wrt='1') and (wa_bus=X"00000010")) else '
50     0';
51
52     muxmem0 <= wd_bus when mem_wrt='1' else mem0;
53     muxmem1 <= wd_bus when mem_wrt='1' else X"00000000";
54     muxmem2 <= wd_bus when mem_wrt='1' else X"00000000";
55
56     process
57     begin

```

```
55     wait until (clock'event) and (clock='1');
56     if (reset='1') then
57         mem0 <= imem0;
58         mem1 <= imem1;
59         mem2 <= imem2;
60     else
61         mem0 <= muxmem0;
62         mem1 <= muxmem1;
63         mem2 <= muxmem2;
64     end if;
65 end process;
66
67 end behaviour;
```