DESIGN OF A MIPS INSTRUCTION SET SIMULATOR FOR MULTICORE PROCESSOR RESEARCH IN SYSTEMC

By

MOHAMMAD ABDUL QAYUM

Master of Science in Electrical Engineering

Oklahoma State University

Stillwater, Oklahoma

2010

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 2010

DESIGN OF A MIPS INSTRUCTION SET SIMULATOR FOR MULTICORE PROCESSOR RESEARCH IN SYSTEMC

Thesis Approved:

Dr. Louis G. Johnson

Thesis Adviser

Dr. Chris Hutchens

Dr. Reza Abdolvand

Dr. A. Gordon Emslie

Dean of the Graduate College

ACKNOWLEDGMENTS

At first, I would like to express my sincere gratitude to my advisor, Dr. Louis G. Johnson for his phenomenal guidance, continuous motivation and limitless inspiration throughout my study at Oklahoma State University. From his teaching at Computer Architecture course, I came to know about his research interest in designing an Instruction Set Simulator (ISS) in new System Level Language, SystemC. It was a wonderful experience to work in his research group and complete this thesis as a part. I would also like to thank my parents for their blessings for my educational ambitions. It is because of their dreams and sacrifices; I have been able to reach my current state. I also like to thanks to my thesis committee members, Dr. Chris Hutchens and Dr. Reza Abdolvand for their valuable comments and suggestions. I also like to thank Dr. James Stine for his Yoda Warrior Compiler from where I got the idea of using Cross compiler for testing this Instruction Set Simulator. Also, I am thankful to Dr Sohum Sohoni's effort to motivate me in reading papers in Advanced Computer Architecture course which help a lot in writing this thesis. Finally, I feel privileged to study at the Department of Electrical and Computer Engineering of Oklahoma State University which will be an invaluable experience throughout my life.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

1.1 Research interest & Literature review 3

1.2 Thesis organization 5

II. BACKGROUND 6

2.1 SystemC 6

2.2 MIPS Architecture 13

2.3 Instruction Set Simulator 14

2.4 Pipelined MIPS 17

2.5 Benchmarks 21

III. DESIGN METHODOLOGY 22

3.1 Instruction Objects 22

3.2 Fetch Unit 24

3.3 Decode Unit 25

3.4 Execute Unit 27

3.5 DMEM Unit 28

3.6 Write Back Unit 29

3.7 Multiply-Divide Unit 31

Chapter Page

3.8 Floating Point Unit 32
3.9 Control Unit 34
3.10 Branch Unit 34
3.11 Cpu Unit 34
3.12 Register, Cache and Memory 38
3.13 Multicore Processor Design 38

IV. Testing Methodology 40

4.1 Individual Instruction test 40
4.2 Benchmark Testing 41
4.2 Testing limitation and Future Testing 45

V. CONCLUSION 46

5.1 Findings and Future Work 46

REFERENCES 48

APPENDICES 50

LIST OF TABLES

Table Page

1 SystemC Architecture ……………………………………..……………………… 7

2 Contents General Purpose Register after 57 instructions……………………… 41

LIST OF FIGURES

Figure Page

1 SystemC Framework..9

2. Graphical notations for modules, interfaces, ports, channels 13

3. Design Flow Simulation 14

4. Unpipelined MIPS architecture...17

5. 5 stage pipelined MIPS architecture 18

6. Mul-div and Floating point unit in parallel with integer unit 19

7. Inter-connection among memory, cache and cpu 37

8. Top cpu module 38

9. Multicore Processor module 39

10. Test result...41

11. Hex file Generation overview 42

LIST OF ABBREVIATIONS

Abbreviation | Elaboration

1. MIPS ....**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages
2. ISA ....**I**nstruction **S**et **A**rchitecture
3. ISS..**I**nstruction **S**et **S**imulator
4. SOC..**S**ystem **O**n **C**hip
5. RTL... **R**esistor **T**ransfer **L**evel
6. HDL ....**H**ardware **D**escription **L**anguage
7. ESL ....**E**lectronic **S**ystem **L**evel
8. EDA .... **E**lectronic **D**esign **A**utomation
9. CAS...**C**ycle **A**ccurate **S**imulator
10. OSCI .... **O**pen **S**ystem**C** **I**nitiative
11. SCV ....**S**ystem**C** **V**erification
12. STL .... **S**tandard **T**emplate **L**ibraries
13. LRM...**L**anguage **R**eference Manual
14. TLM .... **T**ransaction **L**evel **M**odule
15. IP .... **I**ntellectual **P**roperty
16. DSP ....**D**igital **S**ignal **P**rocessing
17. ASIC ....**A**pplication **S**pecific **I**ntegrated **C**ircuit
18. RISC.. **R**educed **I**nstruction **S**et **C**omputer
19. CISC...**C**omplex **I**nstruction **S**et **C**omputer

20. ALU ....**A**rithmati**c** **L**ogic **U**nit

21. CPU..**C**entral **P**rocessing **U**nit

22. MUX .... **M**ultiplexer

1. GPR.. **G**eneral **P**urpose **R**esister

2. FPR .... **F**loating **P**oint **R**esister

3. IPC ....**I**nstruction **P**er **C**ycle

CHAPTER I

INTRODUCTION

As computer applications are becoming more complex, large and versatile; the advent of Complex Chip multiprocessor is ubiquitous. So, designing a complex core and other micro architectural parts of it in **R**egister **T**ransfer **L**evel (RTL) are becoming cumbersome and time consuming. So, arrival of High Level **H**ardware **D**escription **L**anguage (HDL), also called **E**lectronic **S**ystem **L**evel (ESL) Language is welcomed and necessary. This level of language abstracts long and detailed RTL descriptions and emphasis more on algorithmic problem. SystemC is an IEEE standard ESL which is now widely researched in Universities and **E**lectronic **D**esign **A**utomation (EDA) industries. SystemC also offers high productivity by providing the opportunity of co designing hardware and software for earlier verification and trade-offs. **I**nstruction **S**et **S**imulator (ISS) simulates **I**nstruction **S**et **A**rchitecture (ISA), is faster while **C**ycle **A**ccurate **S**imulator (CAS) is related to real architectural implementation. In this thesis a research based ISS designed in SystemC for MIPS architecture will be explored and testing method using benchmark will be discussed. Also, designing a cycle accurate ISS is a step forward to design a real processor. So, some future work for the implementation of this

design will be justified. This simulator is a part of a multicore computer architecture research, as SystemC is highly modular and portable, joining cores and caches will be easier and quicker than conventional programming or HDL languages.

ISS is widely used for software verification in simulated hardware and computer architectural research. Popular ISS's are SIMICS, Simplescalar, OVPSim etc. They are designed in higher level languages like C, C++ or Java and not cycle accurate. So, benchmark result on this simulators is not necessarily predicts real hardware simulation. MIPS (originally an acronym for **M**icroprocessor without **I**nterlocked **P**ipeline **S**tages) is a **R**educed **I**nstruction **S**et **C**omputer (RISC) ISA was developed by now obsolete MIPS Computer Systems (present day MIPS Technologies). MIPS is a result of Stanford University research group's work led by Dr. by John L. Hennessy in 1981. The basic concept was to increase performance by using deep pipelining. CPUs are built up from a number of dedicated sub-units such as instruction fetch unit, decoders, Arithmetic and logic unit (ALU), load/store units and so on. In a traditional non-optimized design, a particular instruction in a program sequence must be executed before next instruction is fetched. But in a pipelined architecture, successive instructions can overlap in execution.

MIPS is a very popular ISA for its simple design. It is taught almost everywhere as in Computer Architecture course as ISA. So, designing an ISS for MIPS should be comfortable for wide availability of information about its architecture. SystemC is a next generation ESL which provides outstanding opportunity for hardware and software

verification. As this ISS designed in SystemC is cycle accurate, it will provide more close hardware simulation. On the availability of high level synthesizer, this code can be easily modified for synthesisation. Simulation of multicore architecture in conventional HDL languages like Verilog or VHDL is very cumbersome and slow while in system level simulation with languages like C, C++ and Java is not good enough to provide standardization and simulate the real flow of the instructions. SystemC provides unique and unprecedented opportunity to design a complex hardware like multicore processor in more abstract level and standard way. To design a multicore system in SystemC is very suitable as it is highly modular and intuitive. This project is part of broader project where the novel micro-architectural design will be tested. So, designing a correct and efficient single core MIPS is the most important part.

1.1 Research interest & Literature review: Most of the MIPS ISA's are designed for **S**ystem **O**n **C**hip (SOC) research [1-3], where RISC type cores are integrated with other hardware modules. But there are very few instances of MIPS core designed in SystemC for Computer Architecture Research. We designed a single core for MIPS I ISA and some part of MIPS IV ISA in a modular fashion so that we can declare instances for new cores and interface with other cores and other Micro-Architectural parts to make it a full multicore Simulator for our Computer Architecture Research at Oklahoma State University.

SoC-Mobinet (System on Chip for Mobile Internet) [4] is a project of European Commission, addressing research training in microelectronics. This project has

developed a synthesizable MIPS processor in SystemC. It is also open-source and can be downloaded [5]. But it has no roadmap for multicore research.

Yon Jun et al [6] have designed a MIPS processor in SystemC which is very similar to our work. But they have used **T**ransaction **L**evel **M**odel (TLM) for passing instruction information among their modules and their design is not cycle accurate. TLM is a high-level approach in modeling digital systems where details of communication among modules are separated from the details of the implementation of functional units or of the communication architecture [7]. Transaction requests take place by calling interface functions of these channel models, which encapsulate low-level details of the information exchange. At the transaction level, the emphasis is more on the functionality of the data transfers - what data are transferred to and from what locations - and less on their actual implementation, which is, on the actual protocol used for data transfer. In our design, we have not used TLM explicitly, but we have transfer instruction objects among the modules in more like real hardware but not bitwise.

Interestingly, their testing methodology is quite similar to us, as they made a binary file created from assembly codes in PERL where we used UNIX grep command. Our work is more significant in that sense; it is cycle accurate, more complete and extensible.

MPARM [8] is a Multiprocessor simulator which uses RISC type ARM processor. Their simulator use C/C++ version of cycle accurate ISS for ARM processor but interfaces and other macro-architectural parts are designed in SystemC. SimSOC [1006] is full SOC

system simulator which uses SystemC for hardware modeling, SytemC/TLM for communications and ISS in C/C++. ISS used in SimSOC is not a conventional ISS, it uses a precompiled specialized instructions for speed up.

Most common and popular full computer System simulators are SIMICS, Simplescalar, RSIM, etc [8]. These are written in C/C++, are not cycle accurate and, are not easily modifiable. Most importantly, they do not emulate real architecture of ISA. The advantage of our ISA as it is designed HDL like language SystemC, it is close to real processor emulation and will predict performance matrices more reliable and accurate then common Computer Architecture simulators.

1.2 Thesis organization: The thesis is organized in four different sections. First section discusses the background related to MIPS ISS design in SystemC. The second section discusses design methodology of this MIPS ISS. The third section is covered with testing methodology. The fourth section discusses contribution of this ISS as a research and future work of multicore research and synthesis of this ISS.

CHAPTER II

BACKGROUND

2.1 SystemC: SystemC™ is a language built on top of standard C++ by extending the language with the use of new class libraries. SystemC addresses the need for a system design and verification language that entails both hardware and software. SystemC is developed and maintained by Open SystemC Initiative (OSCI) and has been approved by the IEEE Standards Association as IEEE 1666-2005, the SystemC Language Reference Manual (LRM). The language is particularly applicable to model system's partitioning, to evaluate and verify the assignment of blocks to either hardware or software implementations, and to architect and measure the interactions among functional blocks. Leading intellectual property (IP), EDA, semiconductor, electronic systems, and embedded software industries currently use SystemC for architectural exploration, for the purpose to deliver high-performance hardware blocks at various levels of abstraction and to develop virtual platforms for hardware/software co-design. [9]

SystemC has similar semantic to VHDL and verilog, but it has syntactical overheads compared to these when used as a HDL. On the other hand, it offers a wider range of expression like object oriented design partitioning and template classes. Although strictly it is a C++ class library, SystemC is sometimes viewed as being a language in its own right. Source code can be compiled with the SystemC library which

includes a simulation kernel to give an executable. The performance of the OSCI open-source implementation of current SystemC is typically less optimal than commercial VHDL/Verilog simulators when used for register transfer level simulation. But research is going on optimize at the synthesis.[10]

Version 1 of SystemC had all the common hardware description language features such as structural hierarchy and connectivity, clock cycle accuracy, delta cycles, 4-state logic (0, 1, X, Z), and bus resolution functions. From version 2 onward, the aim of SystemC has moved towards communication abstraction, TLM, and virtual platform modeling. This version included abstract ports, dynamic processes, and timed event notifications.

<table>
<tr><td>User Libraries</td><td>SCV</td><td colspan="2">Other IP</td></tr>
<tr><td colspan="4">Predefined Primitive Channels like Mutexes, FIFO and Signals</td></tr>
<tr><td rowspan="2">SystemC Kernel</td><td>Threads & Methods</td><td>Channels &Interface</td><td rowspan="2">Data types: Logic, Integer, Fixed Points etc</td></tr>
<tr><td>Events, Sensitivity & Notification</td><td>Modules & Hierarchy</td></tr>
<tr><td colspan="2">C++</td><td colspan="2">STL</td></tr>
</table>

Table1 : SystemC Architecture

The table 1 shows the overall SystemC architecture. SystemC consists of C++ libraries and Standard Template Libraries (STL). SystemC has a kernel which schedules SystemC processes and threads. Processes are defined as methods in SystemC and variable are sensitive to events that means when a method is waiting on event, when the event finishes it notify the process. STL defines standard SystemC data types. On the top, SystemC has some predefined Primitive Channels like Mutexes, FIFO and Signals. The SystemC has verification libraries (SCV) for system verification. Other user libraries and IPs from other parties can be used to develop a full system.

SystemC enables design and verification at the system level, independent of any detailed hardware and software implementation, so as enabling co-verification with RTL design. This higher level of abstraction enables considerably faster, more productive architectural trade-off analysis, design, and redesign then is possible at the more detailed RTL. Furthermore, verification of system architecture and other system-level attributes is orders of magnitude faster than that at the pin-accurate, timing-accurate RT L.

Following figure describes SystemC framework. SystemC modules can be interfaced with other hardware components designed in C/C++ or HDL, can run test benches or Software written C/C++. All hardware components written in SystemC and C/C++ are compiled in standard compiler to create executable (exe) file/files. By running exe file, simulations can be run with test benches.

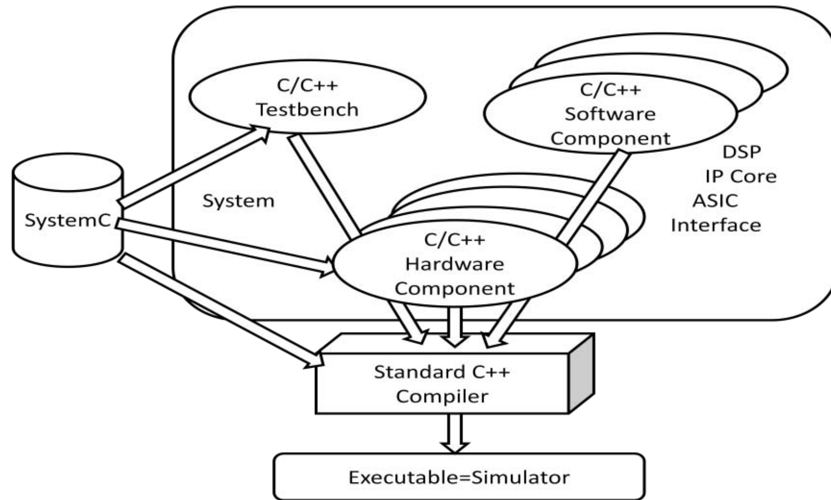

Figure 1: SystemC framework

Major SystemC components are:

Modules: A module is a C++ class which simulates a hardware or software description. SystemC defines that any module has to be derived from the existing class sc_module. SystemC modules are similar to Verilog modules or VHDL entity (.h) and architecture (.cpp) pairs as they represent the basic building block of a hierarchical system. By definition, modules communicate with other modules through channels and via ports. Typically a module will contain numerous concurrent processes used to implement their required behavior. [11] The following code illustrates the creation of the simplest of module

```
SC_MODULE(test_module) { //module
SC_CTOR(test_module) {
cout << “This is a test module” << endl;
```

```
}
};
```

Ports: A port is an integral part of a SystemC module. Ports are used by modules to communicate to or from a module with the outside module say, another module. In a simple way, we can consider a port as pin of a hardware component.

In HDLs such as VHDL or, Verilog ports are metaphorically like pins. But in SystemC ports are have substantially more general purposes and so, are designed in more sophisticated way and also, more versatile in use than its counterpart HDL languages.

A simple SystemC port declaration can be defined as:

```
sc_in <bool> test_input;
```

The port has a name *test_input* and in this particular instance, it is of input mode since we used the *sc_**in***<> port type. The last observation that we can make from this simple line of code is the use of the *bool* data type inside the <> of the *sc_in* port type. This data type refers to the kind of data that will be exchanged on that port. In other words, we are expecting to receive boolean values on the *test_input* port.

Also, there exists numerous predefined port types in SystemC such as *sc_in<Type>, sc_out<Type>, sc_inout<Type>*, etc. Most of those ports are almost

identical to their counterpart HDL equivalent of VHDL or Verilog; they have a name, a type and a mode. Usually, these kinds of ports are commonly used in RTL design in SystemC. However, SystemC ports have much more flexibility than RTL like ports; because SystemC ports not only have a name and a type which define the access mechanisms that should be used on them. Actually, the access mechanisms are just a list of allowed messages that can be used on them. If we consider the *sc_in<bool>* port of the previous example, in that case SystemC defines that by *read()* function message can be read from it. So intuitively, an *sc_out*<> port would allow one to use *write()* function to write the message. The following code demonstrates the use of an sc_in<bool> and an sc_out<bool> ports.

```
sc_in<bool> test_input;
sc_out<bool> test_output;

void simple() {
if (test_input.read() == true)

{
test_output.write(false);
}

else {
test_output.write(true);
```

```
}
}
};
```

Channels and interface: SystemC channels are written by using C++ class interface principle. In this principle abstract base classes are constructed and common interfaces for related derived classes are also defined. These abstract base classes are used to define all the access methods that a channel should have. Consequently, a channel is written as a C++ class derived from an abstract base class and it implement the access methods defined inside its abstract base parent class.

An abstract base class is written in C++ with the help of one or more pure virtual methods as part of that class. A pure virtual method is a method that is usually implemented inside a derived class from the abstract base class. The semantic of a pure C++ virtual method may look like:

```
Virtual <return_type> function_name (args>)=0;
```

For example, the following line of code uses a method called data_write that requires three input arguments and returns an integer type:

```
virtual void data_write( int address, int Bytes, sc_lv<16> *data ) = 0;
```

The keyword ‘virtual’ and the ‘=0’ are the essential parts of this declaration as they represent to the compiler that ‘data_write’ is a pure virtual method and therefore, that the class containing this declaration can never be used into an object. Once one or a number of abstract base classes (interfaces) have been formulated, channels can be implemented by simply inheriting one or more of the base classes and implementing their virtual methods. The following figure describes all the basic components and their interconnections in standard graphical notation.

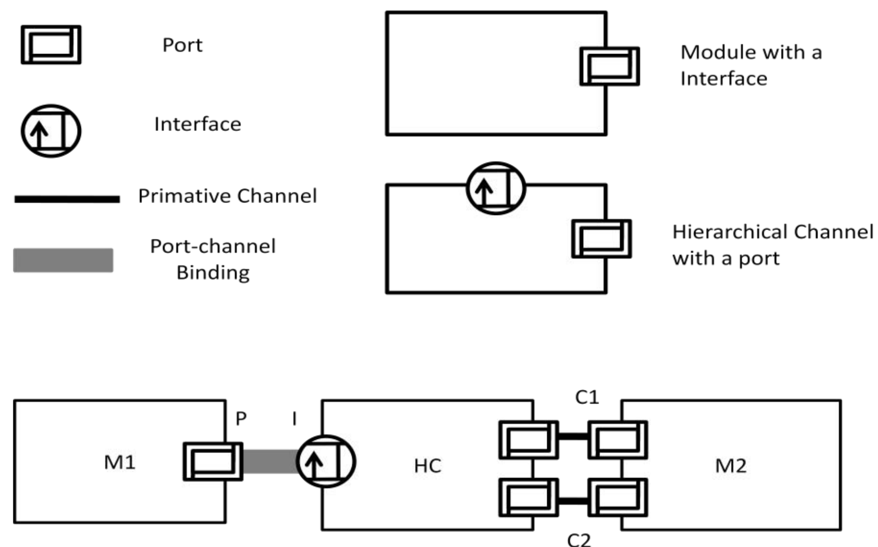

Figure 2: (a) Graphical notations for modules, interfaces, ports, channels, and port-channel binding (b) Example with two modules and a hierarchical channel [11]

Abakus Library: *OSU AbaKus* is a SystemC like kernel developed by a former PhD student Aswin Ramachandran developed in C++ to develop hardware simulator in system level. Aswin et. al. argued [12] that their kernel is more accurate, flexible to use

and in some cases increase the simulation speed and also, It is flexible to adapt to any hardware description language. The design flow for different micro-architecture simulators is illustrated in Figure 3.

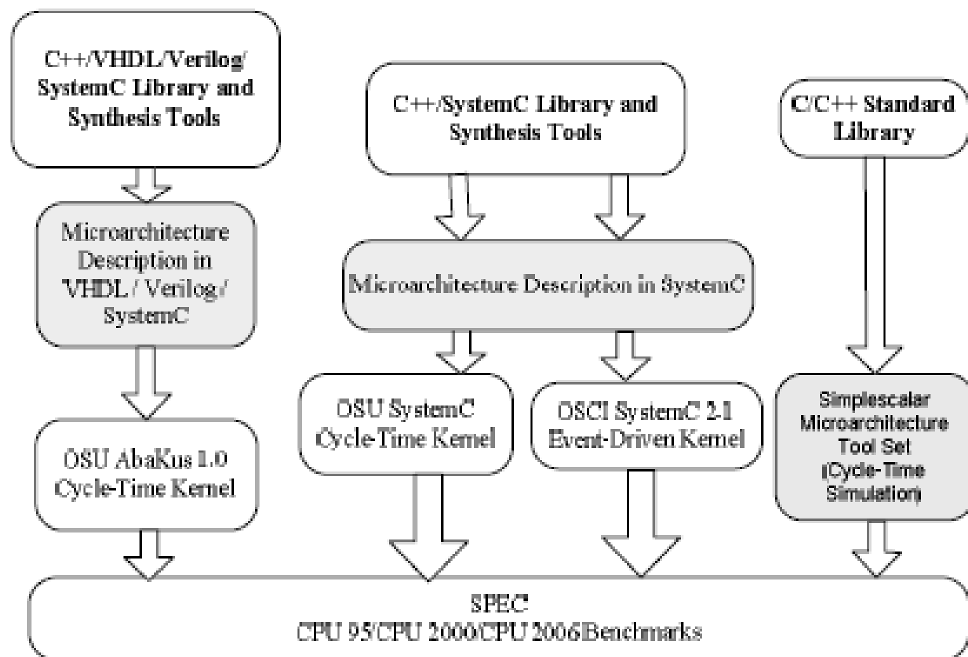

Figure 3: Design Flow Simulation

2.2 Instruction Set Simulator: ISS is a simulation model coded in a high-level programming language like C, C++ or Java, which emulates the behavior of a microprocessor by reading and decoding instructions and maintaining internal variables such as processor's registers accordingly.

ISS is widely used for following possible reasons:

- To simulate the machine code of another hardware device or entire computer for upward compatibility -a full system simulator typically includes an instruction set simulator.
- To improve the speed performance of simulations while verilog simulation is significantly slow for verification purpose. Sometimes ISS is co-simulated with other verilog micro-architectural parts.
- To collect information for predicting or analyzing performance of novel architectural design.
- To develop new software and applications earlier on future processor platform.

Widely used ISS's are SIMICS, Simplescalar, SimOS, GEMS etc. Some are full system simulator eg. Simics, SimOS which offers simulation of several types of ISA. Some are ISA specific (eg. WinMIPS64 is 64 bit mips ISA) and micro-architecture specific (eg. Dinero- a hierarchical cache simulator).

2.3 MIPS architecture: MIPS is the most common RISC type ISA. Conventional Complex Instruction Set Architecture (CISC) takes many cycles to execute an instruction due to its complex addressing modes (eg. Intel Architecture-32 or, IA-32). It is argued that such functions would perform better by sequences of simpler instructions reducing the number of slow memory accesses. In RISC design, most instructions are of uniform length and similar structure, arithmetic operations are restricted to CPU registers and only separate *load* and *store* instructions access memory. These properties enable a

better balancing of pipeline stages than before, making RISC pipelines significantly more efficient and allowing higher clock frequencies. The common features of MIPS instruction-set architecture are:

- It is usually simple load-store architecture and uses general-purpose registers.
- It has only two addressing modes, displacement and immediate, but can be formulated to other important modes from them.
- It supports 8-, 16-, 32-, and 64-bit integers, and 32- and 64-bit IEEE 754 floating-point numbers.
- It has an orthogonal set of instructions to manipulate these data types.
- It has separate comparison and branching instructions. MIPS has thirty-two 32-bit general-purpose registers (GPR), named R0, R1,… , R31. R0 always contains 0 with another value has no effect).
- It has 32 floating-point registers (FPR), which can hold either Single Precision (32-bit) or double-precision (64-bit) values.
- It 32 bit floating point status and control register which is used for floating point comparison and branching.
- It has several co-processors. In earlier version (eg. MIPS I) co-processor one is used as floating point unit (FPU). Other co-processors are for control support. Later version (eg. MIPS R3000) FPUs are used as integral part.

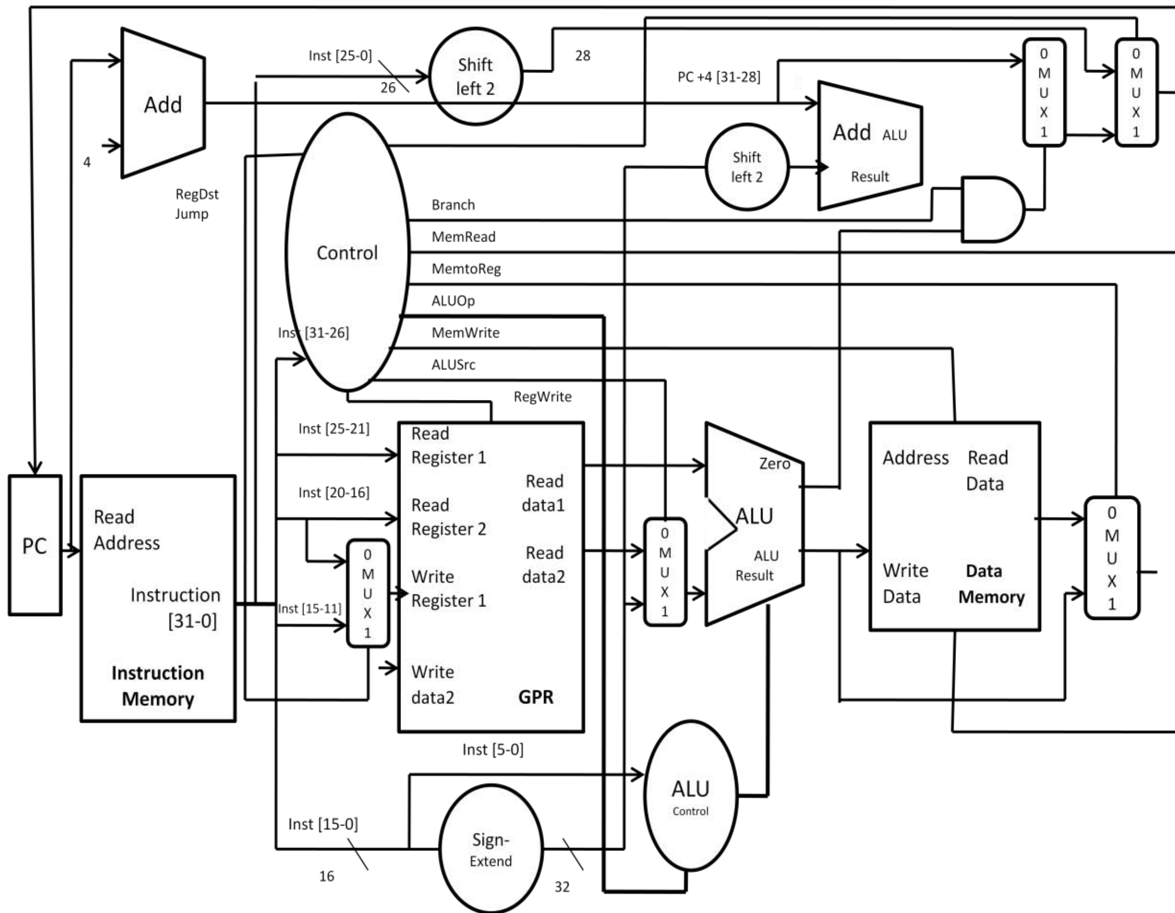

Figure 4: Un-pipelined MIPS architecture [13]

2.3 Pipeline MIPS Architecture: The problem with the unpipelined design is that each instruction must finish before another instruction can start. The hardware, for example, the ALU is only used when none of other hardware is used. The basic idea of pipelining is to utilize of the unused time of the CPU components so that more than one instruction is can be processed simultaneously; i.e other instructions can go through without waiting for the previous instruction to finish.

The following figure shows a 5 stage pipelined MIPS architecture:

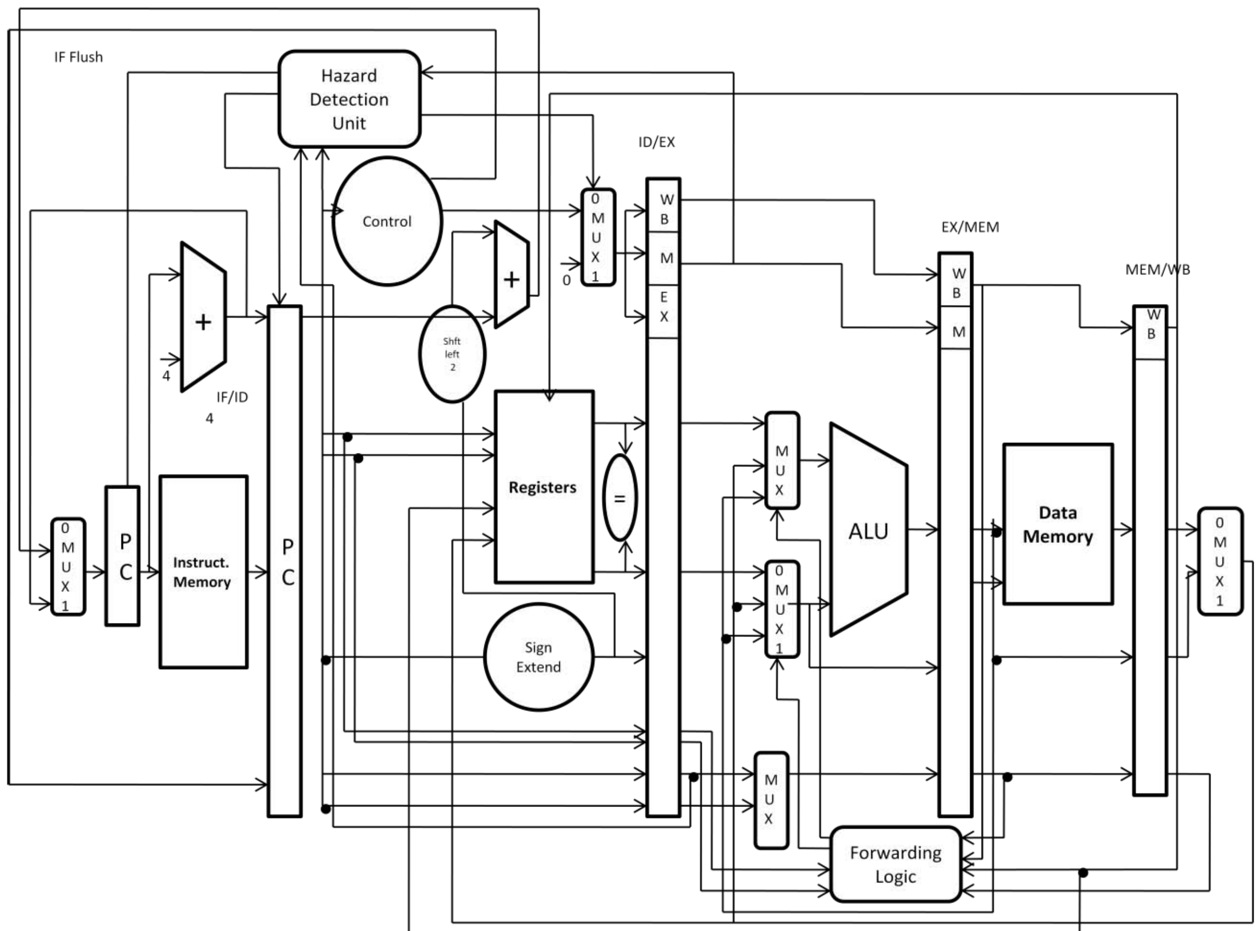

Figure: 5 stage pipelined MIPS architecture [13]

Our MIPS design has pipe stages- fetch, decode, execute, memory access and write back. Here, the description of following stages of MIPS:

- *IF/ID*: this stage controls the passing of *rs*, *rd*, and *rt* fields of the instruction with the opcode and *funct* fields, to the control other circuitry according to the instruction.

- *ID/EX*: this stage buffers control for the EX, MEM, and WB stages, while executing control for the EX stage. Control also dictates what operands will be inputs to the ALU, what ALU operation should be performed, and whether or not a branch is to be taken based on the ALU Zero output.
- *EX/MEM*: this stage buffers control for the MEM and WB stages, while executing control for the MEM stage. The control lines are executed for memory read or write, and also for data selection for memory write. This stage of control also maintains the branch control logic.
- *MEM/WB*: this stage buffers and executes control for the WB stage, and selects the value to be written into the register file.

We have separate multiply-divide (mul-div) unit and floating point unit (FPU) which work parallel with integer unit to perform multiply-divide and floating point arithmetic instructions respectively as shown in figure 6

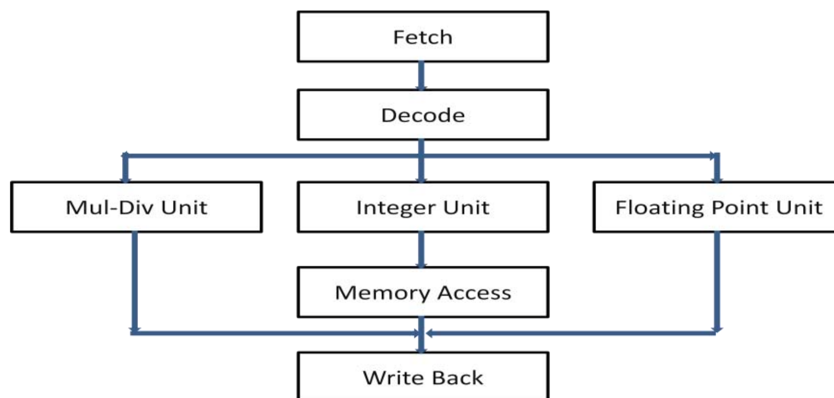

Figure 6: Mul-div and Floating point unit in parallel with integer unit

Control unit provide necessary control signals to each stage to synchronize all the functions of an instruction. There is a separate branch unit which calculate necessary branch conditions and calculate the address to jump.

There are three basic types of integer instructions:

R-type instructions refer to register type instructions. Of the three formats, the R-type is the most complex. This is the format of the R-type instruction, when it is encoded in machine code.

B31-26	B25-21	B20-16	B15-11	B10-6	B5-0
opcode	register rs	register rt	register rd	shift amount	function

I-type is short for "immediate type". The format of an I-type instruction looks like:

B31-26	B25-21	B20-16	B15-0
opcode	register rs	register rt	immediate

J-type is short for "jump type". The format of a J-type instruction looks like:

B31-26	B25-0
opcode	Target

There are two basic types of floating point instructions: **FR-type** instructions refer to floating point register type instructions.

B31-26	B25-21	B20-16	B15-11	B10-6	B5-0
opcode	register fmt	register ft	register fs	register fd	function

FI-type is short for "Floating Point immediate type". The format of an FI-type instruction looks like:

B31-26	B25-21	B20-16	B15-0
opcode	register fmt	register ft	immediate

2.4 Benchmarks: A benchmark is a standard test or a set of standard tests written in programming languages to measure the relative performance of a system. It is used in Computer Architecture research to assess the comparative performance of a hardware say, floating point performance of a CPU, or cache miss rate of a particular cache hierarchy. Benchmarks include versatile tests so that they can trial exhaustedly the different relative performance of hardware. For example, a benchmark written for testing a particular cache hierarchy will be memory intensive while a benchmark written for testing CPU speed will be computation intensive. There are other types of benchmarks called test benches which are used for validation of small hardware or software parts. Most common benchmarks in computer architecture are SPEC, SPLASH, Mediabench etc.

CHAPTER III

Design Methodology

We have used fusion of C++, SystemC version 2.0 and Abakus library for our design. As a compiler we used GCC version 4.11 and our host machine is tesla1 which is a Intel Server machine with Linux 64 bit operating system. Our top module is cpu. Inside cpu other lower modules have instances and interconnections. To test our simulator we used cpu_test.cpp file which uses cpu as Design Under Test (DUT). Each unit has a header file (.h) which defines all the ports, local channels, local modules and constructors. In cpp file all the local ports, channels and modules are initialized. All local modules connections are defined through ports and channels. And also, the main module function also declared and finally, a dump function is called for image of all the registers in that cpp file.

3.1 Instruction Objects: We designed our ISA in SystemC more like ASIC design procedures which use HDL languages like Verilog or VHDL. But we have some significant differences. We defined instructions as objects which have pointers to them. The object has several pointers to sub-object (eg. itype, r.rs, r.rt) which contains register's address, offset value, opcodes etc. The pointer to main instruction object is passes through each stage and other units e.g. control, calculation or changes are done on the sub-objects.

For instruction object we have a separate module which describes functionality of the instruction object on respective stages. For example, in the following "add" instruction r.rs pointer points to a sub-object this is the address of a GPR register, so as r.rt and r.rd. These three rnum_t function is executed in decode stage. The execute_func function does the "add" arithmetic task at the execute stage. This "add" instruction has no activity at memory access stage but has a default activity (write the result to the destination register) in the write back stage. Following is an example which shows how add instruction is implemented.

```
struct add_t : alu_r_t {
add_t() {};
virtual rnum_t get_ra(instruction* inst) { return inst->r.rs; }
virtual rnum_t get_rb(instruction* inst) { return inst->r.rt; }
virtual rnum_t get_rd(instruction* inst) { return inst->r.rd; }
virtual void execute_func(instruction *inst,
        data_t a_data,
        data_t b_data,
        data_t &data)
    { data = a_data + b_data; }
} add;
```

The Instruction implemented above is a very convenient way to add a new instruction. We can implement any new instruction of any new architecture for MIPS or even totally new architecture like PowerPC RISC type ISA in this fashion.

3.3 Fetch Unit: In the simple fetch stage icode (instruction object) is fetched according to pc_value object (PC) from the I-cache. It is initially stored IF/ID pipeline and in next clock cycle it passes to the decode unit. Following is an example of fetch unit codes

```
fetch::fetch() {}

fetch::fetch(sc_module_name name) : sc_module(name),

    //initialize ports

    pc("pc"),

    pc_next("pc_next"),

    pc_inst("pc_inst"),

    br_addr("br_addr"),

    br_cond("br_cond"),

    i_mem("i_mem")

{}

void fetch::evaluate() {

   instruction *inst = pc_inst->read();

   addr_t pc_value = pc->read();

    //read instruction

   inst->iaddr = pc_value;

   i_mem->read(pc_value, inst->icode);

   inst->decode(); //set correct fields here, hardware does it in
next stage
```

```
    //get next pc
    if ( br_cond->read() )
      pc_next->write( br_addr->read() );
    else
      pc_next->write(pc_value + sizeof(icode_t));
}

void fetch::dump(ostream &out) const {
    out << endl << name() << endl;
}
```

3.3 Decode Unit: An instruction is deciphered in the decode stage to 6 bit opcode and 6 bit funct code for the control purpose of the instruction. The Registers (rs, rt, rd, fmt, ft, fs, fd) are also read in this stage. Note that the first source register's identifier (rt/ft) in every instruction is at bit positions [25:21] and second source register's identifier (rs/fs) is at bit positions [20:16]. The destination register's identifier is either at bit positions [15:11] (for R-type) or at [20:16]. The correct destination register's identifier is identified via multiplexer controlled by the control signal RegDst [Fig. 4].

In our design, when instruction object is in decode stage, it calls a decode function. In the decode functions all the opcode, funct code and other register values are read and pointed by the some sub pointers of the instruction object. This sub

pointer and instruction object then pass to the next pipelines and units. Following is a sample code for decode function:

```
void instruction::decode() {
    icode_t opcode = icode >> 26;
    itype = dectab[opcode];
    if (itype->op == FUNCREG) {
              opcode = icode & 0x3f;
              itype = regdectab[opcode];
    }
    switch(itype->format) {
      case R:
        r.rs = (icode >> 21) & 0x1f;
        r.rt = (icode >> 16) & 0x1f;
        r.rd = (icode >> 11) & 0x1f;
        r.shamt = (icode >> 6) & 0x1f;
              break;
      case I:
        i.rs = (icode >> 21) & 0x1f;
        i.rt = (icode >> 16) & 0x1f;
        register short immh = icode & 0xffff;
        i.imm = (int) immh;  //sign extended
              break;
      case J:
```

```
                j.addr = (iaddr + 4) & 0xf0000000 | ((icode &
0x03ffffff) << 2);

                break;

     }

}
```

3.4 Execute Unit: In conventional MIPS architecture, execute stage includes generally ALU and other parts. In our case, we have three modules in parallel- integer unit (execute unit), Mul-div unit and floating point unit. In integer unit, a "evaluate" function is called which do basic arithmetic functions on the instruction object and results are given as return value which is passed to next EX/MEM pipeline. Similarly in mul-div unit and floating point unit, the evaluate function does necessary manipulation in the instructions and return a result through a sub-pointer.

In mul-div unit we have "ab-pipe" type mul-div-pipe pipeline register which simulate the arbitrary super-pipeline requirement due to the long calculation for multiply and divide. That means, multiply and divide instructions require several stages (clock cycles) to produce result. Similarly, we can simulate different number of super-pipeline requirement for different type of floating point instructions.

Normally, in RISC architecture an integer instruction takes one cycle. But the combinational logic of divide and some floating point instruction can have long delay which can be equal to several cycles. For that we have defined delay variable "divide_delay" which can simulate delay in clock time periods. Similarly, we have

different floating_point_delay for different type of floating point instructions. Following piece of code shows implementation of execute unit.

```
void execute::evaluate() {
     instruction* inst = id_ex_inst->read();
     data_t data_;
     data_t b_data_ = b_data->read();
     inst->itype->execute_func(inst, a_data->read(), b_data_,
data_);
     data->write(data_);
     st_data->write(b_data_);
     if (inst->itype->op == FUNCFP)   {
     data_t fdata_;
     d_data_t fb_data_;
     inst->itype->execute_func_f(inst, a_data->read(), fb_data_,
fdata_);
     data->write(fdata_);
     st_fdata->write(fb_data_);
     }
}
```

3.5 DMEM Unit (Memory Access Unit): In DMEM unit, register values are loaded from or store to the memory. This unit exclusively used for load/store type of instructions. Other integer instructions do nothing in this stage. Following piece of code shows an example of DMEM-

```
void d_mem::evaluate() {
    register instruction* inst = ex_mem_inst->read();
    data_t data;
    inst->itype->d_mem_func(inst, &mem, ex_mem_st_d->read(),
      (addr_t) ex_mem_reg_d->read(), data);
    mem_wb_reg_d->write(data);
    if (inst->itype->op == FUNCFP) {
    d_data_t f_data; //fp
    inst->itype->d_mem_func_f(inst, &mem, ex_mem_st_fd->read(),
(addr_t)
ex_mem_reg_d->read(), f_data);  //fp reading address should be
int
    mem_wb_reg_fd->write(f_data); //fp
    }
}
```

3.6 Write Back Unit: In write back stage result of arithmetic operation is written to register (GPR or FPR) according to destination address in the instruction (eg. rs/rd/fd). We implemented write back stage in this unit. This unit is connected to integer, mul-div and floating point unit. Though three units are connected to this unit, but our design works in sequential way. That means an instruction wait or stalled in this unit until its result come from mul-div or floating point unit. Following is a sample code for write back unit:

```
void wr_back::evaluate() {
```

```
//write register
instruction* inst = mem_wb_inst->read();
wb_src_t wb_src;
inst->itype->wr_back_func(inst, wb_src);
data_t wb_src_data;
d_data_t wb_src_fdata;
switch( wb_src ) {
  case MFGPR:
 wb_src_data = mem_wb_reg_d->read();
 break;
  case MFHI:
 wb_src_data = hi_reg_d->read();
 break;
  case MFLO:
 wb_src_data = lo_reg_d->read();
 break;
  case MFFPR:
    wb_src_fdata = f_reg_d->read();
    break;
  case FPR:
    wb_src_fdata = mem_wb_reg_fd->read();
    break;
}
reg_d->write( inst->itype->get_rd(inst), wb_src_data );
reg_fd->write( inst->itype->get_fd(inst), wb_src_fdata );
```

```
    /* maintain $r0 semantics */
    reg_d->write(REG_ZERO, 0);
}
```

3.8 Mul-div Unit: In the mul-div unit, multiply and divide instruction takes place. Actually MIPS I is 32 bit machine, but result of two 32 bit numbers is 64 bit. So, in MIPS there are two special instructions- move from low (MFLO) and move from high (MFHI) for transferring 64 bit result to two 32 bit GPR register. In our design, result is stored in two register called “HI” and “LO” in case of multiply or divide instruction. The contents of HI/LO is transferred on immediate MFLO/MFHI instruction. Instruction mul-div type instruction objects usually pass to write back stage but stalled until the result are available. The following code shows that “mul_div_func” function takes the instruction object and return HI/LO values and store to ‘hi’ and ‘lo’ register. It also shows how stalls are implemented.

```
void mul_div::evaluate() {
    //data
    instruction* ex_inst = mul_div_start_reg.qout->read();
    ex_inst->r.rs_data = a_data->read();
    ex_inst->r.rt_data = b_data->read();
    instruction* inst = mul_div_pipe.qout->read();
    data_t data_hi;
    data_t data_lo;
    inst->itype->mul_div_func(inst, data_hi, data_lo);
    hi->write(data_hi);
```

```
    lo->write(data_lo);

    //control
    op_t op = inst->itype->op;
    stall_hi_lo_chan.write( (op != MUL) && (op != DIV) );
    //div stalls
    if (ex_inst->itype->op == DIV) {
        div_delay_chan.write(1);
        cout << "starting divide delay\n";
        div_delay.write(0);
    }
}
```

3.8 Floating Point Unit: In the floating point unit, all the floating point arithmatic instructions are taken place. Actually MIPS I is 32 bit machine, but result of two 32 bit numbers is 64 bit. But as FPR contains 64 bit registers, so we do not have same problem as mul-div unit. In that case, result is stored in "f_data" and directly transferred to the write back stage. As different FP instructions have different cycle delays and multiple stages to do calculation, we have the option for pipelines and delays which can be arbitrarily defined. There are some floating point instruction like Move to Coprocessor one (MTC1), where co-processor is floating point unit and Move from Coprocessor one (MFC1) which transfer values between GPR and FPR. For this, we require some connections between integer unit and floating point unit which are implemented in the

“cpu” unit. Following piece of code shows that “float_unit_func” takes instruction object and return the result.

```
void float_unit::evaluate() {
    //data
    instruction* ex_inst = float_unit_start_reg.qout->read();
    ex_inst->fr.fs_data = fa_data->read();
    ex_inst->fr.ft_data = fb_data->read();
    instruction* inst = float_unit_pipe.qout->read();
    d_data_t data_reg;
    inst->itype->float_unit_func (inst, data_reg);
    f_reg->write(data_reg);

    //control
    op_t op = inst->itype->op;
    stall_f_reg_chan.write(op != FP);

    //div stalls
    if (ex_inst->itype->op == FP) {
        float_delay_chan.write(1);
        cout << "starting float delay\n";
        float_delay.write(0);
    }
}
```

3.9 Control Unit: The control unit dictates the flow of the instruction through stalling the pipelines when required. It is connected to all the pipeline registers, mul-div unit and floating point unit. It takes instruction object in deferent stages, look for control hazards and calculate stall logics which stall the required pipeline registers.

3.11 Branch Unit: Branch unit includes a "branch_func" which takes instruction object and caculate the branch address to be taken and branch condition- true or false. The following code shows a simple branch unit implementation.

```
void branch::evaluate() {
    instruction *inst = if_id_inst->read();
    addr_t br_addr_;
    bool br_cond_;
    inst->itype->branch_func(inst,
      reg_a_data->read(),
      reg_b_data->read(),
      br_addr_,
      br_cond_);
    br_addr->write(br_addr_);
    br_cond->write(br_cond_);
}
```

3.12 Cpu Unit: This is the top hierarchical module which includes all the lower module instances. It is like a top module in HDL language, which is declared in the test bench as

DUT. In this module all the connections and sequences are described. Following piece of code shows instances are called

```
fetch if_stage;
decode id_stage;
execute ex_stage;
d_mem mem_stage;
wr_back wb_stage;
regfile<data_t, 32, 2, 1> gpr;
control controller;
```

The remaining codes shows the basic interconnection of the major modules where each pipeline register (eg. pc, id_ex_reg_a) are connected with clock signal, stall signal from controller. Stall is required for the synchronization of the instruction flow.

```
//local module connection (data path)
    pc.clk(clk);
    pc.stall(controller.stall_if);
    id_ex_reg_a.clk(clk);
    id_ex_reg_a.stall(controller.stall_ex);
    id_ex_reg_fa.clk(clk); //fp
    id_ex_reg_fa.stall(controller.stall_ex); //fp

    id_ex_reg_b.clk(clk);
    id_ex_reg_b.stall(controller.stall_ex);
    id_ex_reg_fb.clk(clk);  //fp
    id_ex_reg_fb.stall(controller.stall_ex); //fp
```

```
ex_mem_reg_d.clk(clk);
ex_mem_reg_d.stall(controller.stall_d_mem);

ex_mem_st_d.clk(clk);
ex_mem_st_d.stall(controller.stall_d_mem);
ex_mem_st_fd.clk(clk);  //fp
ex_mem_st_fd.stall(controller.stall_d_mem);  //fp

mem_wb_reg_d.clk(clk);
mem_wb_reg_d.stall(controller.stall_wr_back);
mem_wb_reg_fd.clk(clk);   //fp
mem_wb_reg_fd.stall(controller.stall_wr_back); //fp

hi.clk(clk);
hi.stall(mul_div_unit.stall_hi_lo);
lo.clk(clk);
lo.stall(mul_div_unit.stall_hi_lo);

f_reg.clk(clk); //fp
f_reg.stall(float_point_unit.stall_f_reg); //fp

if_stage.pc(pc.qout);
if_stage.pc_next(pc.din);
if_stage.pc_inst(controller.pc_inst);
if_stage.br_addr(br_addr);
```

```
if_stage.br_cond(br_cond);
if_stage.i_mem(inst_mem);

id_stage.if_id_inst(controller.if_id_inst);
id_stage.reg_a(gpr.rd_export[0]);
id_stage.reg_b(gpr.rd_export[1]);
id_stage.id_ex_reg_a(id_reg_a);
id_stage.id_ex_reg_b(id_reg_b);

id_stage.reg_fa(fpr.rd_export[0]); //fp
id_stage.reg_fb(fpr.rd_export[1]);  //fp
id_stage.id_ex_reg_fa(id_reg_fa);  //fp
id_stage.id_ex_reg_fb(id_reg_fb);  //fp
```

Following figure shows the cpu module where instances of other modules are called and their interconnections are also shown. Due to complexity of control and branch connections to other units, it is not shown.

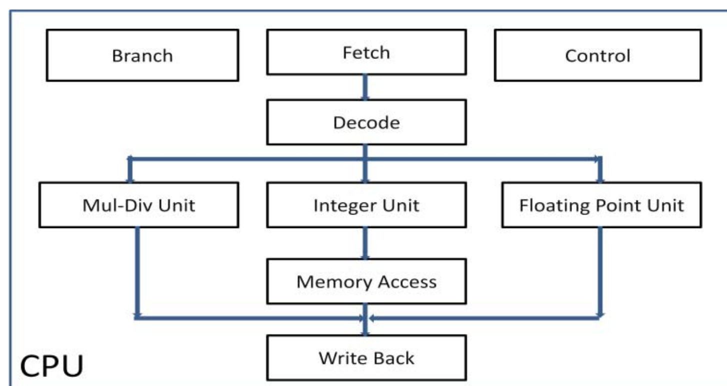

Figure 7: Top ‘cpu module’ which contains other modules

3.12 Registers, Cache and Memory: Register file, cache and memory are designed in way that it can contain 32 bit data. We use template for storing different type of data like-bool, char, int, float and double. So, we can call any register with any data size. For our GPR we used 32 bit integer and FPR 64 bit double. Our cache and memory are also scalable. Cache has all the common placement and replacement policies. Following figure shows the a simple interconnection configuration among cpu, cache, and memory.

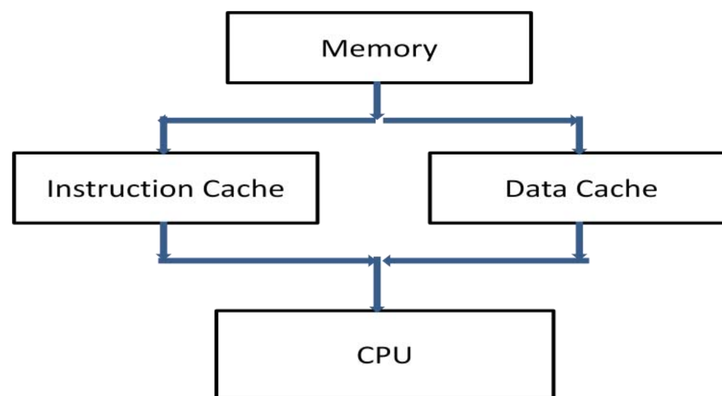

Figure 8: Inter-connection among memory, cache and cpu

3.13 Multicore Processor Design: In a multicore processor design, same cores will be instantiated within a processor where core includes cpu and cache. Cores may share higher level of caches (L2). Further, processors can be instantiated along with shared memory and system bus in the top multicore processor module. But we need to design some cache coherency and consistency protocols for multicore processor. Note that, we have not implemented this multicore model, but the goal behind this project is to

facilitate a multicore processor design. With our single core module, we can easily implement the processor part but some significant work will be required for shared cache and system bus. Following figure shows how 4 cores can be implemented to form a multicore processor.

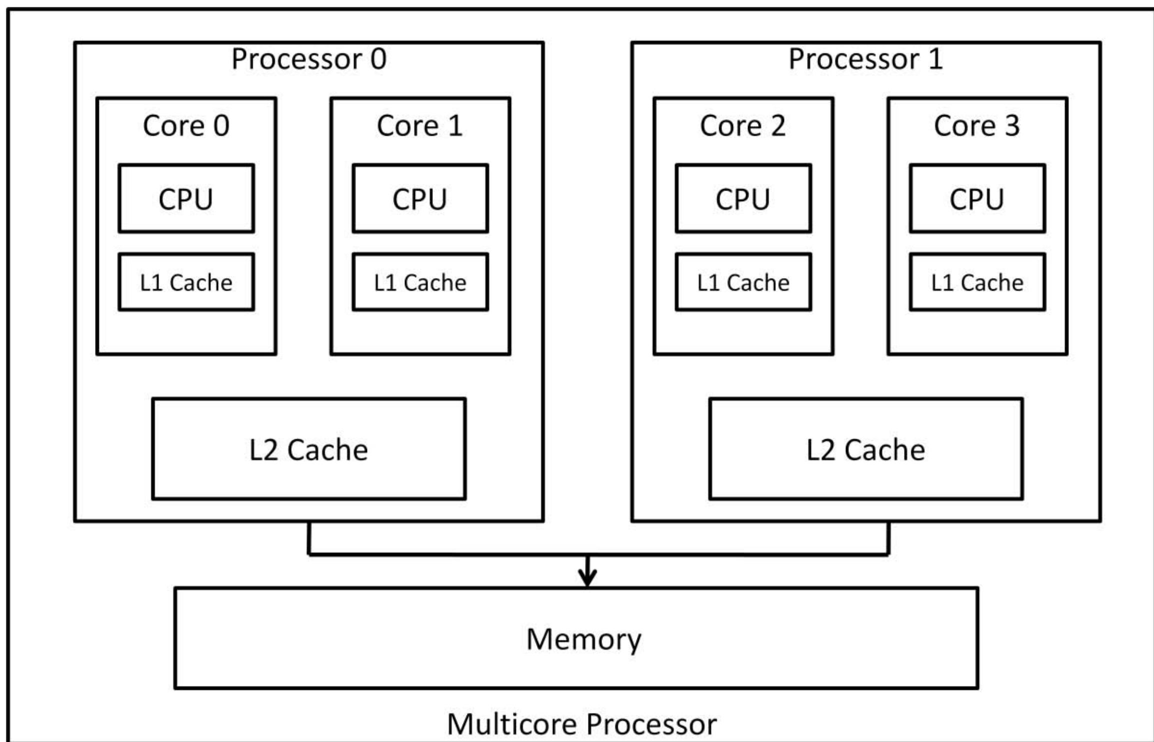

Figure 9: Multicore processor module

CHAPTER IV

Testing Methodology

Testing and debugging an ISS is a really complex task. We tested our instruction both individually and with simple benchmark written in C. We plan to run standard benchmark like SPEC 2006 but for that we need a loader. We are designing loader as part of multicore research.

4.1 Individual Instruction test: For testing individual instruction, we have written some C code to make a file which contains individual instruction in hexadecimal format. We manually calculated the GPR image for each cycle and compare with GPR image created by simulation. It gives us error if the image is different than the expected one. Following figure shows a GPR image after 57 instructions. Our ISS correctly executed 57 integer instructions and give some information as shown in figure. It took 63 cycles to execute 57 instructions. Some instructions like MUL, DIV take several cycles, so beforehand we know the the cycle numbers too.

By using “dump” function we can see current value of any register at any instance. So, we put dump function whenever we have to test a pipeline register. But checking

the GPR image in each cycle is a very convincing way to test the accuracy of an individual instruction.

0	17ffb	10004	10004	0	0	0	0
0	0	20002000	110011	40022000	22004	0	0
1c	0	0	0	20002000	20011000	40022000	22004
0	0	0	0	0	0	0	30

Table 2: Contents General Purpose Register after 57 instructions

```
In process: cpu_test.process @ 63 ns

finished sc_start

total clock cycles: 63

instruction count: 57

completed: 57

correct: 57

errors: 0

grade:100

instructions/cycle (IPC): 0.904762
```

Figure 10: Test result

4.2 Benchmark test: Here we will discuss an easy but convincing way of testing our MIPS ISS. A benchmark written in higher level language (C, C++) is compiled for MIPS

ISA. It is then to converted executable binary file (.hex) by using unix “grep” command. Upon simulation an output file (test.out) is generated with memory contents. From the memory snapshot in test.out, final memory location is checked for result of the benchmark. Following flowchart describes how executable binary file (.hex) is generated.

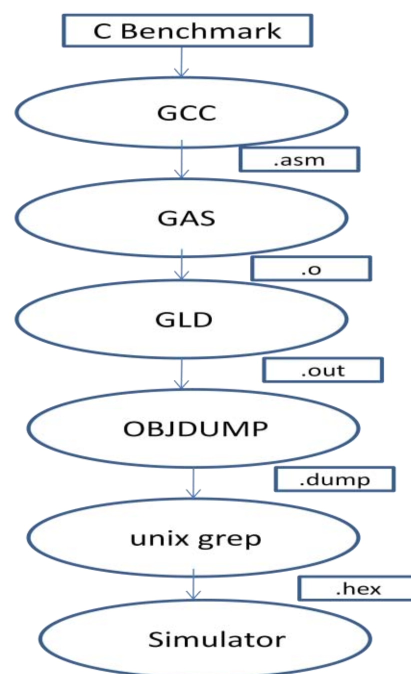

Figure 11: Hex file design flow

The trick is that a final address is provided in the code. Also, code is written in a versatile way so that compilation of it includes as more as Instructions possible, long calculation and variety of higher level language. If final result of this long exhaustive manipulation is found in the addresss defined in the code, it can be said- it is well tested. However, individual instruction is tested with one or two lines of machine code thoroughly. So, hundreds lines of machine codes generated by compiler will be well

equipped to test the correctness of the simulator. The following is a sample code used for testing where final result is stored in the address 0x160c and location of SP, FP and GP is also defined so that compiler does not use out -bound addresses.

```
//test.c
asm("addi $29, $0, 0x1000");
asm("addi $28, $0, 0x1200");
asm("addi $30, $0, 0x1400");
int main() {
        int x, y, a, b, c, d, *q;
        x=0; y=1;
        a=20;
        b=30;
        c=10;
        q=0x160c; //final result address
        d = b-a;
        while(y)
        {
        if (x<20)
```

```
 {d = d+c;
x++;}
if (x==20)
 y=0;
}
d = calculate(d);
 *q= for_loop(d);
return 0;
}
int calculate(int p)
{
p = p+3;
p = p+100;
p = p+45;
return p;
}
int for_loop (int r)
{
int i;
for (i=0; i<20; i++)
r=r+5;
```

```
return r;
}
```

4.3 Testing limitation and Future testing: We tested individual floating point instructions by looking at the image of FPR and GPR in each cycle. But for floating point, we could not test with C test benches as it requires data type to be filled into data cache. We are still developing loader which will load both instructions and data into respective caches. Then we can write test benches in C with floating point operations and can test cohesiveness of the full ISS.

We individually tested correctness of cache and memory. But the correctness of full ISS requires testing with full hierarchy of cache and memory. But the scope of thesis is limited to the full work of a single core simulator. We have designed cache, separately tested it. But it requires a loader and boot loading functions for mapping Virtual Memory concept, placement and replacement policies. Currently, the loader is in testing case. When a working loader will be available, we will integrate the cache and test with more complex benchmarks. Our final target is to run standard benchmark like SPEC 2006.

CHAPTER V

Conclusion

5.1 Findings and Future Work: As we are going towards multiple cores and complex architectural design, ESL like SystemC is an obvious choice. Designing an ISS SystemC is more real to hardware than conventional ISS as it is cycle accurate. We cannot say about speed, as we have not yet tested with any standard benchmark. The most advantage of ISS design in SystemC is that it is modular, interface-able and standardized. It can also be synthesizable with some modification. Some companies like Sysnopsis, Metor Graphics, Forte Design System have already claimed that their synthesizer can synthesize SystemC codes. With fully tested single core ISS, we can go further design multicore architectural research. As designing a multicore simulator is a long exhaustive task with limited workforce, it is not possible in a single thesis. But as multicore will be just instances of single cores and top module is required only for interconnection. But to control coherency and consistency among shared cache for multicores is also going to be big task. But single accurate MIPS ISS core is first but important step of the whole research.

The future work will be the design of a loader which can load real benchmark like SPEC. Then we can test Instruction Per Cycle (IPC), runtime and other matrices for our whole ISS. We hope that our ISS will be faster in runtime than conventional ISS as it uses

optimized and standardized SystemC and Abakus Kernel [12]. Also, runtime is the biggest bottleneck of computer architecture research. As SystemC is maturing as HDL and ESL, novel macro and micro architectural exploration will be easier than common ISS available.

.

REFERENCES

[1] David Chih-Wei Chang, I-Tao Liao, Jenq-Kuen Lee, Wen-Feng Chen, Shau-Yin Tseng, and Chein-Wei Jen, "Applying ESL in A Dual-Core SoC Platform Designing, Computer Design," *Proceedings. International Conference*, IEEE, 17-20, pp. 335 – 342. September 2000.

[2] Chien-Chang Wang, "A Dual RISC Core SoC Platform," The Master Thesis of Department of electrical engineering, National Cheng-Kung University, Taiwan, June 2004.

[3] Yu-Liang Chou, "A Superscalar Dual Core Architecture for ARM9 ISA," The Master Thesis of Department of Electrical Engineering, National Sun Yat-Sen University, Taiwan, July 2005.

[4] J. Nurmi, J. Madsen, E. Ofner, J. Isoaho and H. Tenhunen: The SoC-Mobinet Model in System-on-Chip Education, Proc. IEEE Int. Conf. on Microelectronic Systems Education (MSE'05), 2005, pp. 71-72.

[5] "A MIPS R2000 Core" [Online]. Available: http://www2.imm.dtu.dk/SoC-Mobinet/elements/index.htm [Accessed: Dec. 3, 2003].

[6] Yon Jun Shin and Sofiène Tahar et al. A SystemC Transaction Level Model for the MIPS R3000 Processor, 4th International Conference: Sciences of Electronic, Technologies of Information and Telecommunications March 25-29, 2007 – TUNISIA

[9] Open SystemC Initiative. *The SystemC Library*, Website, 2006. www.systemc.org/

[10] Open SystemC Initiative, "Forte Design Systems Announces SystemC Synthesis Success with Fujitsu Microelectronics Europe," [Online]. Available: http://www.systemc.org/news/pr/view?item_key=5cc7fb8a687cba91873e783919a8c949531e055f. [Accessed: July. 7, 2010].

[11] T. Gro¨ tker, S. Liao, G. Martin, and S. Swan, "System Design with SystemC." Kluwer Academic, 2002.

[12] Ramachandran, A.; Johnson, L.G.; "Advanced microarchitecture simulator for design, verification and synthesis." Circuits and Systems, 2007. MWSCAS 2007.

[13] Hennessy and D.A. Patterson, "Computer Organization and design" Morgan Kaufmann Publishers, 4th Edition, 2009

APPPENDICES

Appendix A: Instruction description

This table lists all MIPS instructions with their opcode, assembler format, and semantics.

A.1 Control instructions:

J: Jump to absolute address.
Opcode: 0x01
Format: J target
Semantics: PC = nPC; nPC = (PC & 0xf0000000) | (target << 2)

JAL: Jump to absolute address and link.
Opcode: 0x03
Format: JAL target
$31 = PC + 8 (or nPC + 4); PC = nPC;
nPC = (PC & 0xf0000000) | (target << 2)

JR: Jump to register address.
Opcode: 0x00
Funccode: 0x08
Format: JR rs
Semantics: PC = nPC; nPC = $s;

JALR: Jump to register address and link.
Opcode: 0x00
Funccode: 0x09
Format: JALR rs
Semantics: $31 = PC + 8 (or nPC + 4); PC = nPC; nPC = $s;

BEQ: Branch if equal.
Opcode: 0x04
Format: BEQ rs,rt,offset
Semantics: if $s == $t advance_pc (offset << 2)); else advance_pc (4);

BNE: Branch if not equal.
Opcode: 0x05
Format: BEQ rs,rt,offset
Semantics: if $s != $t advance_pc (offset << 2)); else advance_pc (4);

BLEZ: Branch if less than or equal to zero.
Opcode: 0x06
Format: BLEZ rs,offset
Semantics: if $s <= 0 advance_pc (offset <<
2)); else advance_pc (4);

BGTZ: Branch if greater than zero.
Opcode: 0x07
Format: BGTZ rs,offsetSemantics: if $s > 0 advance_pc (offset << 2)); else advance_pc (4); bgtz $s, offset;

BLTZ: Branch if less than zero.
Opcode: 0x01
Format: BLTZ rs,offset
Semantics: if $s < 0 advance_pc (offset << 2)); else advance_pc (4);

BGEZ: Branch if greater than or equal to zero.
Opcode: 0x01
Format: BGEZ rs,offset
Semantics: if $s >= 0 advance_pc (offset << 2)); else advance_pc (4);

A.2 Load/store instructions

LW: Load word, indexed addressing.
Opcode: 0x23

Format: lw $t, offset($s)
Semantics: $t = MEM[$s + offset];
advance_pc (4);

LHW: Load half word, indexed addressing.
Opcode: 0x23
Format: lw $t, offset($s)
Semantics: $t = MEM[$s + offset];
advance_pc (4);
LB: Load half word, indexed addressing.
Opcode: 0x23
Format: lw $t, offset($s)
Semantics: $t = MEM[$s + offset];
advance_pc (4);

LUI: Load half word, indexed addressing.
Opcode: 0x23
Format: lw $t, offset($s)
Semantics: $t = MEM[$s + offset];
advance_pc (4);

LWC1: Load half word, indexed addressing.
Opcode: 0x23
Format: lw $t, offset($s)
Semantics: $t = MEM[$s + offset];
advance_pc (4);

LDC1: Load half word, indexed addressing.
Opcode: 0x23
Format: lw $t, offset($s)
Semantics: $t = MEM[$s + offset];
advance_pc (4);

SW: Load half word, indexed addressing.
Opcode: 0x23
Format: lw $t, offset($s)
Semantics: $t = MEM[$s + offset];
advance_pc (4);

SHW: Load half word, indexed addressing.
Opcode: 0x23
Format: lw $t, offset($s)
Semantics: $t = MEM[$s + offset];
advance_pc (4);

SB: Load half word, indexed addressing.
Opcode: 0x23
Format: lw $t, offset($s)
Semantics: $t = MEM[$s + offset];
advance_pc (4);

SWC1: Load half word, indexed addressing.
Opcode: 0x23
Format: lw $t, offset($s)
Semantics: $t = MEM[$s + offset];
advance_pc (4);

SDC1: Load half word, indexed addressing.
Opcode: 0x23
Format: lw $t, offset($s)
Semantics: $t = MEM[$s + offset];
advance_pc (4);

A.3 Integer instructions

ADD: Add signed (with overflow check).
Opcode: 0x00
Funccode:0x20
Format: ADD rd,rs,rt
Semantics: $d = $s + $t; advance_pc (4);

ADDI: Add immediate signed (with overflow check).
Opcode: 0x08
Format: ADDI rd,rs,rt
Semantics: addi $t, $s, imm;

ADDU: Add unsigned (no overflow check).
Opcode: 0x00
Funccode:0x21
Format: ADDU rd,rs,rt
Semantics: $d = $s + $t; advance_pc (4);

ADDIU: Add immediate unsigned (no overflow check).x
Opcode: 0x09
Format: ADDIU rd,rs,rt
Semantics: $t = $s + imm; advance_pc (4);

SUB: Subtract signed (with underflow check).
Opcode: 0x00
Funccode:0x22
Format: SUB rd,rs,rt
Semantics: $d = $s - $t; advance_pc (4);

SUBU: Subtract unsigned (without underflow check).
Opcode: 0x00
Funccode:0x23
Format: SUBU rd,rs,rt
Semantics: $d = $s - $t; advance_pc (4);

MULT: Multiply signed.
Opcode: 0x00
Funccode: 0x18
Format: MULT rs,rt
Semantics: $LO = $s * $t; advance_pc (4);

MULTU: Multiply unsigned.
Opcode: 0x00
Funccode: 0x19
Format: MULTU rs,rt;
Semantics: $LO = $s * $t; advance_pc (4);

MUL_DIV: Divide signed.
Opcode: 0x00
Funccode: 0x1A
Format: DIV rs,rt
Semantics: $LO = $s / $t; $HI = $s % $t; advance_pc (4);

MFHI: Move from HI register.
Opcode: 0x00

Funccode: 0x10
Format: MFHI rd
Semantics: $d = $HI; advance_pc (4);

MFLO: Move from LO register.
Opcode: 0x00
Funccode: 0x12
Format: MFLO rd
Semantics: 4d = $LO; advance_pc (4);

AND: Logical AND.
Opcode: 0x00
Funccode :0x14
Format: AND rd,rs,rt
Semantics: $d = $s & $t; advance_pc (4);

ANDI: Logical AND immediate.
Opcode: 0x12
Format: ANDI rd,rt,imm
Semantics: $d = $s & imm; advance_pc (4);

OR: Logical OR.
Opcode: 0x00
Funccode: 0x15
Format: OR rd,rs,rt
Semantics: $d = $s | $t; advance_pc (4);
ORI: Logical OR immediate.
Opcode: 0x0d
Format: ORI rd,rt,imm
Semantics: $s | imm;

advance_pc (4);

XOR: Logical XOR.
Opcode: 0x00
Funccode: 0x16
Format: XOR rd,rs,rt
Semantics: $d = $s^$t;
advance_pc (4);

XORI: Logical XOR immediate.
Opcode: 0x0e
Format: ORI rd,rt,imm
Semantics: : $d = $s^imm;
advance_pc (4);

SLL: Shift left logical.
Opcode: 0x00
Funccode:0x00
Format: SLL rd,rt,shamt
Semantics: $d = $t << h; advance_pc (4);

SLLV: Shift left logical variable.
Opcode: 0x00
Funccode: 0x04
Format: SLLV rd,rt,rs
Semantics: $d = $t << $s; advance_pc (4);

SRL: Shift right logical.
Opcode: 0x00
Funccode: 0x02
Format: SRL rd,rt,shamt
Semantics: $d = $t >> h; advance_pc (4);

SRLV: Shift right logical variable.
Opcode: 0x00
Funccode: 0x06
Format: SRLV rd,rt,rs
Semantics$d = $t >> $s; advance_pc (4);

SRA: Shift right arithmetic.
Opcode: 0x00
Funccode: 0x03
Format: SRA rd,rt,shamt
Semantics: $d = $t >> h; advance_pc (4);
SLT: Set register if less than.
Opcode: 0x00
Funccode: 0x1a
Format: SLT rd,rs,rt
Semantics: if $s < $t $d = 1; advance_pc (4); else $d = 0; advance_pc (4);

SLTI: Set register if less than immediate.
Opcode: 0x0a
Format: SLTI rd,rs,imm
Semantics: if $s < imm $t = 1;
advance_pc (4); else $t = 0; advance_pc (4);

SLTU: Set register if less than unsigned.
Opcode: 0x00
Funccode: 0x1b
Format: SLTU rd,rs,rt
Semantics: if $s < $t $d = 1; advance_pc (4); else $d = 0; advance_pc (4);

SLTIU: Set register if less than unsigned immediate.
Opcode: 0x0b
Format: SLTIU rd,rs,imm
Semantics: if $s < imm $t = 1;
advance_pc (4); else $t = 0; advance_pc (4);

A.3 Floating Point Instruction

BCLT: Branch on FP condition true.
Opcode: 0x11
Fmtcode: 0x08
Format: BCLT fmt, offset
Semantics: if (FPcond)advance_pc (offset << 2)); else advance_pc (4);

BCLF: Branch on FP condition true.
Opcode: 0x11
Fmtcode: 0x08
Format: BCLF fmt, offset
Semantics: if (!FPcond) advance_pc (offset << 2)); else advance_pc (4);

C.EQ.S: FP Compare Single Precision.
Opcode: 0x11
Fmtcode: 0x10
Funccode: 0x32
Format: C.EQ.S fs, ft
Semantics: FPcond=(F[fs]==F[ft])? 1 : 0
advance_pc (4);

C.EQ.D: FP Compare Double Precision.
Opcode: 0x11
Fmtcode: 0x11
Funccode: 0x32
Format: C.EQ.D fs, ft
Semantics: FPcond=({F[fs], F[fs+1]}==F{[ft]), F[ft+1]}? 1 : 0
advance_pc (4);

C.LT.S: FP Compare Single Precision.
Opcode: 0x11
Fmtcode: 0x10
Funccode: 0x3c
Format: C.LT.S fs, ft
Semantics: FPcond=(F[fs]<F[ft])? 1 : 0
advance_pc (4);

C.LT.D: FP Compare Double Precision.
Opcode: 0x11
Fmtcode: 0x11
Funccode: 0x3c
Format: C.LT.D fs, ft
Semantics: FPcond=({F[fs], F[fs+1]}<F{[ft]), F[ft+1]}? 1 : 0
advance_pc (4);

C.LE.S: FP Compare Single Precision.
Opcode: 0x11
Fmtcode: 0x10
Funccode: 0x3e
Format: C.LE.S fs, ft
Semantics: FPcond=(F[fs]<=F[ft])? 1 : 0
advance_pc (4);

C.LE.D: FP Compare Double Precision.
Opcode: 0x11
Fmtcode: 0x11
Funccode: 0x3c
Format: C.LE.D fs, ft
Semantics: FPcond=({F[fs], F[fs+1]}<=F{[ft]), F[ft+1]}? 1 : 0
advance_pc (4);

ADD.S: Add single precision FP numbers
Opcode: 0x11
Fmtcode: 0x10
Funccode:0x00
Format: ADD fd,fs,ft
Semantics: F[fd] = F[fs] + F[t];
advance_pc (4);

ADD.D: Add double precision FP numbers
Opcode: 0x11
Fmtcode: 0x11
Funccode:0x00
Format: ADD.D fd,fs,ft
Semantics: {F[fd], F[fd+1]} = {F[fs], F[fs+1]} + {F[ft], F[ft+1]};
advance_pc (4);

SUB.S: Subtract single precision FP numbers
Opcode: 0x11
Fmtcode: 0x10
Funccode:0x01
Format: SUB.S fd,fs,ft
Semantics: F[fd] = F[fs] - F[t];
advance_pc (4);

SUB.D: Subtract double precision FP numbers
Opcode: 0x11
Fmtcode: 0x11
Funccode:0x00
Format: SUB.D fd,fs,ft
Semantics: {F[fd], F[fd+1]} = {F[fs], F[fs+1]} - {F[ft], F[ft+1]}; advance_pc (4);

MUL.S: Multiply single precision FP numbers
Opcode: 0x11
Fmtcode: 0x10
Funccode:0x02
Format: MUL.S fd,fs,ft
Semantics: F[fd] = F[fs] * F[t]; advance_pc (4);

MUL.D: Multiply double precision FP numbers
Opcode: 0x11
Fmtcode: 0x11
Funccode:0x00
Format: MUL.D fd,fs,ft
Semantics: {F[fd], F[fd+1]} = {F[fs], F[fs+1]} * {F[ft], F[ft+1]}; advance_pc (4);

DIV.S: Divide single precision FP numbers
Opcode: 0x11
Fmtcode: 0x10
Funccode:0x01
Format: DIV.S fd,fs,ft
Semantics: F[fd] = F[fs] / F[t]; advance_pc (4);

DIV.D: Divide double precision FP numbers
Opcode: 0x11
Fmtcode: 0x11
Funccode:0x00
Format: DIV.D fd,fs,ft
Semantics: {F[fd], F[fd+1]} = {F[fs], F[fs+1]}
/{F[ft], F[ft+1]}; advance_pc (4);

CVT.S.W: Convert Integer to single precison FP number
Opcode: 0x11
Fmtcode: 0x14
Funccode:0x20
Format: CVT.S.W fd, fs
Semantics: fd = convert_and_round(fs) advance_pc (4);

CVT.D.W: Convert Integer to double precison FP number
Opcode: 0x11
Fmtcode: 0x14
Funccode:0x20
Format: CVT.S.W fd, fs
Semantics: fd = convert_and_round(fs) advance_pc (4);

CVT.W.S: Convert single precision FP number to integer
Opcode: 0x11
Fmtcode: 0x14
Funccode:0x20
Format: CVT.S.W fd, fs
Semantics: fd = convert_and_round(fs) advance_pc (4);

CVT.D.S: Convert single to double precision FP number
Opcode: 0x11
Fmtcode: 0x14

Funccode:0x20
Format: CVT.S.W fd, fs
Semantics: fd
= convert_and_round(fs) advance_pc
(4);

CVT.S.D: Convert single precision FP
number to integer
Opcode: 0x11
Fmtcode: 0x14
Funccode:0x20
Format: CVT.S.W fd, fs
Semantics: fd
= convert_and_round(fs) advance_pc
(4);

CVT.S.D: Convert single to double
precision FP number
Opcode: 0x11
Fmtcode: 0x14
Funccode:0x20
Format: CVT.S.W fd, fs
Semantics: fd
= convert_and_round(fs) advance_pc
(4);

A.3 Other instructions

Syscall: Call OS routine
Opcode: 0x00
Funccode: 0x0c
Format: syscall
Semantics: advance_pc (4);
NOOP: No operation
Opcode: 0x00
Funccode: 0x00
Format: No operation

Appendix B: A Makefile sample for creating simulator executable

```
SHELL = /usr/bin/tcsh -f
PUBLIC = /x/lgjohn/public
COPY_DIR = /x/lgjohn/public/src/cpumips

#DEBUG = -ggdb
DEBUG =

C_FILES = cpu_test.cpp cpu.cpp fetch.cpp decode.cpp execute.cpp
d_mem.cpp wr_back.cpp control.cpp forwd_mux.cpp branch.cpp mul_div.cpp
buffer.cpp float_unit.cpp

H_FILES = cpu.h fetch.h decode.h execute.h d_mem.h wr_back.h\
        control.h forwd_mux.h branch.h mul_div.h\
        buffer.h float_unit.h

O_FILES = cpu_test.o cpu.o fetch.o decode.o execute.o d_mem.o wr_back.o
control.o forwd_mux.o branch.o mul_div.o float_unit.o

A_FILES = $(HOME)/lib/libmips.a $(HOME)/lib/libabakus.a\
          $(HOME)/lib/libsystemc.a $(HOME)/lib/libmem.a
BINARY = cpu
LIBS    = -L$(HOME)/lib -L$(PUBLIC)/lib -lsystemc -labakus -lmips -lmem
-lm

IFLAGS  = -I$(HOME)/include -I$(PUBLIC)/include
#CFLAGS  = -Wall -DSC_INCLUDE_FX -O3 $(DEBUG)
CFLAGS  = -Wall -O3 $(DEBUG)
CC      = g++
```

```
$(BINARY):      $(O_FILES) $(A_FILES)
      $(CC) $(CFLAGS) $(IFLAGS) -o $(BINARY) $(O_FILES) $(LIBS)
.cpp.o:
      $(CC) $(CFLAGS) $(IFLAGS) -c $<

$(O_FILES):     $(H_FILES)

install:
      cd memory; make
      cd memory; make install
      cd instruction; make
      cd instruction; make install
      cd regfile; make install
      make

copy:
      if ! -d $(COPY_DIR) mkdir $(COPY_DIR)
      cp Makefile $(COPY_DIR)
      cp prog.hex $(COPY_DIR)
      cp $(C_FILES) $(COPY_DIR)
      cp $(H_FILES) $(COPY_DIR)
      cd instruction; make copy
      cd memory; make copy
      cd regfile; make copy
```

Appendix C: A Makefile sample for creating instruction hex file

```
# Tell where various comiler exists
#export PATH=$PATH:/home/maqayum/gnu-mips-installer/install/bin
CC = /home/maqayum/gnu-mips-installer/install/bin/mips-elf-gcc
AS = /home/maqayum/gnu-mips-installer/install/bin/mips-elf-as
LD = /home/maqayum/gnu-mips-installer/install/bin/mips-elf-ld
DUMP = /home/maqayum/gnu-mips-installer/install/bin/mips-elf-
objdump

# Where the source directories are
BLD = ../build
SRC = ./Desktop
INCLUDEFILES =

# Build all of the main programs in the src folder
all: test.hex

# This line prevents make from automatically deleting these files
as temporary
.PRECIOUS: %.dat %.dump %.out %.o %.s %.asm
%.asm: %.c
	$(CC) $(CFLAGS) -S -c $< -o $@

%.o: %.asm
	$(AS) -c $< -o $@
```

```
%.dump: %.o
	$(DUMP) -d --disassemble-zeroes $< > $@

%.hex: %.dump
	cat $< | grep --only-matching "^ *[0-9a-fA-F]\+:[^0-9a-fA-
F]*[0-9a-fA-F ]\+" | tr -d " " | grep --only-matching "[0-9a-fA-
F]\{8\}" > $@


clean:

	rm *.o *.dump
```

Appendix D: A Test code in hexadecimal format

```
2001fffc

20020008

221820

ac430100

8c420100

10400003

8000002

20217fff

3e00008

c000007

2010001c

2000009

3c0a1000

214a1000

3c0b0011

216b0011

14b0018

6012

6810

aa040

154a82a

16a0000a

cb040

db840
```

```
196a82a

12a00002

22ed0001

800001d

22ed0000

22cc0000

228a0000

8000013

1aa001a

a812

0

0

0

c

0
```

VITA

Mohammad Abdul Qayum

Candidate for the Degree of

Master of Science

Thesis: DESIGN OF A MIPS INSTRUCTION SET SIMULATOR FOR MULTICORE PROCESSOR RESEARCH IN SYSTEMC

Major Field: Electrical Engineering

Biographical:

Personal Data: Born in Brahmanbaria, Bangladesh on January 29, 1982

Education:

Bachelors: Earned Bachelor of Science in Electrical and Electronic Engineering at Bangladesh University of Engineering and Technology, Dhaka on July 2005.

Masters: Completed the requirements for the Master of Science in Electrical Engineering at Oklahoma State University, Stillwater, Oklahoma in July, 2010.