Pipelined Processor Design and Implementation

Monsoon 2024

Submitted
by

Krishna Negi (2110110295)
and
Purnima Pant (2110110404)

Under Supervision
of

Dr. Venkatnarayan Hariharan
Department of Electrical Engineering

SHIV NADAR
INSTITUTION OF EMINENCE DEEMED TO BE
UNIVERSITY
DELHI NCR

SCHOOL OF
ENGINEERING

Department of Electrical Engineering,
School of Engineering,
Shiv Nadar Institution of Eminence Deemed to be University,
Delhi-NCR

Abstract

The semantics of all the instructions that a processor can execute is known as the instruction set architecture (ISA). For a practical implementation of any complete ISA, the processor requires:

1. A memory to hold instructions and data.
2. A control unit to control and manage the execution of instructions, and flow of data.
3. An Arithmetic Logic Unit (ALU) to process instructions and perform arithmetic and logical operations.
4. A register to store, transfer and load data during the execution of instructions.

In a typical processor, instruction processing involves five stages: Instruction Fetch (IF), Operand Fetch (OF), Execute (EX), Memory Access (MA) and Register Write (RW).

In non-pipelined CPUs, these instruction processing stages are executed sequentially. In contrast, in a pipelined CPU, this process is optimized by overlapping instruction stages, thereby allowing the execution of multiple instructions in different stages concurrently. Pipelining, therefore, results in a significant improvement in computational efficiency when compared to a non-pipelined processor.

The objective of this project is to implement a fully functional pipelined processor that can efficiently handle instructions from SimpleRisc, the ISA from the book "Basic Computer Architecture" by Prof. Smruti Sarangi. The implementation will be carried out using Verilog.

Keywords: *ISA, Pipelined CPU, Instruction Stages, Verilog, SimpleRisc.*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The field of processor design is fundamental to computer engineering and serves as the backbone of computing systems. At the heart of every processor is the ability to execute instructions in a timely and efficient manner. As modern computing demands have increased, there has been a need to improve processor performance by making better use of available resources. One key technique employed to achieve this is pipelining.

1.1 Pipelined vs Non-Pipelined

In traditional, non-pipelined processors, instructions are executed sequentially, meaning that one instruction must fully complete before the next one can begin. This leads to underutilization of the processor's resources, as different parts of the processor remain idle while waiting for an instruction to complete. To overcome this inefficiency, pipelining is introduced, allowing multiple instructions to be processed concurrently, each in a different phase of execution. This significantly improves instruction throughput and overall system performance.

The concept of pipelining in processor design involves dividing the execution of an instruction into several stages, with each stage handling a distinct part of the instruction's execution. The goal is to have multiple instructions simultaneously progress through different stages of the pipeline, thereby utilizing all stages at once. Pipelining transforms instruction execution from a single, linear process into a parallel one, where multiple instructions are at various stages of execution in each clock cycle.

1.2 Stages in the CPU Design

In this project, we designed a pipelined CPU that follows a five-stage pipeline model, which is common in RISC (Reduced Instruction Set Computer) processors. The five stages in our pipeline are derived from the general pipeline structure as discussed in *Basic Computer Architecture* by Prof. Smruti Sarangi [1]. The five stages are:

1.2.1 Instruction Fetch (IF)

This is the first stage of the pipeline, responsible for fetching the instruction from memory. The program counter (PC) points to the address of the current instruction in memory. After fetching the instruction, the PC is incremented to point to the next instruction unless it is a branch instruction. In the case of a branch instruction, the PC points to the branch target [1].

1.2.2 Operand Fetch (OF)

In this stage, the required operands are fetched from the register file. Depending on the instruction type, the operands could either be register values or immediate constants embedded in the instruction itself. The operand fetch stage is crucial for providing the necessary data that the CPU works on in the next stage [1].

1.2.3 Execute (EX)

The execution stage is where the actual computation happens. The instruction is decoded, and the corresponding operation is performed, such as an arithmetic calculation (e.g., a logical operation, etc. This stage is critical for the processor's functionality, as it determines the result based on the operands provided by the previous stage. Branch instructions are handled by a dedicated branch unit that computes the outcome, and final target of the branch. Non branch instructions are handled by an ALU (Arithmetic Logic Unit) [1].

1.2.4 Memory Access (MA)

This stage is executed by instructions that require data to be read from or written to memory, such as load and store instructions. In these cases, the memory address is calculated during the execution stage, and in the memory access stage, data is either fetched from memory (in the case of a load instruction) or written back to memory (in the case of a store instruction). The memory unit takes control signals to determine if it is a load instruction or a store instruction. If it is neither, the memory unit is disabled [1].

1.2.5 Register Write (RW)

The final stage in the pipeline involves writing the result of the instruction back into the appropriate register in the register file. This stage is essential for updating the state of the CPU and ensuring that subsequent instructions have access to the correct data [1].

1.3 Project Objectives

The primary objective of this project is to design and implement a pipelined CPU based on SimpleRisc, a simple and efficient instruction set architecture. Later stages of the project will involve refining the pipeline to minimize stalls and implement solutions for handling hazards and control flow changes.

Chapter 2

Literature Survey

This literature survey focuses on the design and implementation of pipelined processors, drawing from the foundational knowledge provided in the chapters of Basic Computer Architecture. By examining these chapters, we will delve into concepts specific to the project. This includes SimpleRisc Instruction Set Architecture, stages of instruction processing in a pipelined processor, and the differences between Hardwired and Microprogramming design styles.

2.1 RISC and CISC:

RISC: A Reduced Instruction Set Computer (RISC) implements simple instructions that have a simple and regular structure. The number of instructions is typically a small number [1]. Examples: ARM, IBM PowerPC

CISC: A Complex Instruction Set Computer (CISC) implements complex instructions that are highly irregular, take multiple operands, and implement complex functionalities. Secondly, the number of instructions is large [1]. Examples: Intel x86, VAX

2.2 Instruction Set Architecture:

ISA: The semantics of all the instructions supported by a processor is known as the instruction set architecture (ISA). This includes the semantics of the instructions themselves, along with their operands, and interfaces with peripheral devices [1]. In other words, instruction set architecture is the way that software perceives hardware.

An ISA needs to have some necessary properties. These are:

- **Complete**: Should be able to implement all user programs. Should be able to express all programs in machine code that a user intends to write for it.
- **Concise**: Ideally shouldn't have a lot of instructions since implementing a lot of instructions will unnecessarily increase the number of transistors and increase its complexity.

- **Generic**: Should capture the common case. Implementing very rarely used computation in instructions can be skipped.
- **Simple**: Should be simple. RISC implements simple instructions that have simple and regular structure, so the operation is executed quickly but it requires the compiler to generate more instructions.

The ISA used in this project is called SimpleRisc.

2.3 SimpleRisc ISA:

SimpleRisc has 21 instructions. It has 16 registers numbered r0 – r15. The first 14 registers are general purpose registers. r14 is the stack pointer (sp) and r15 is the return address register (ra). Each register is 32 bits wide. There is a special internal register called flags assumed which is not visible to the programmer, and contains 2 fields flags.E (equal) and flags.G (greater than).

E is 1 if the result of comparison is equality and G is 1 if comparison concludes that the first operand is greater than the second one. Default values of both are 0.

Each instruction is encoded into a 32-bit value and requires 4 bytes of storage in memory.

2.3.1 Types of Instructions in SimpleRisc ISA

- **Arithmetic Instructions**: SimpleRisc has 6 arithmetic instructions: add, sub, mul, div, mod, and cmp. Their functionality of the first 5 instructions is self-explanatory.
  Example: add r1, r2, 10; mul r1, r2, r3.

  The cmp instruction is a 2-address instruction that takes two source operands. The first source operand needs to be a register, and the second one can be an immediate or a register. It compares both the operands by subtracting the second from the first. If the operands are equal, then it sets flags.E to 1. Otherwise flags.E is set to 0. If the first operand is greater than the second operand, then the result of the subtraction will be positive. In this case, the cmp instruction sets flags.GT to 1, otherwise it sets it to 0.

Table 2.1: Arithmetic Instructions

Instructions	**Opcode**	**Semantics**
add	add reg, reg, (reg/imm)	00000
sub	sub reg, reg, (reg/imm)	00001
mul	mul reg, reg, (reg/imm)	00010
div	div reg, reg, (reg/imm)	00011
mod	mod reg, reg, (reg/imm)	00100
cmp	cmp reg, (reg/imm)	00101

- **Logical Instructions**: SimpleRisc has three logical instructions and, or, and not. and and or are 3-address instructions. They compute the bitwise AND and OR of two values respectively. The not instruction is a 2-address instruction that computes the bitwise complement of a value.
  Example: and r1, r2, r3; not r1, r2.

Table 2.2: Logical Instructions

Instructions	**Semantics**	**Opcode**
and	add reg, reg, (reg/imm)	00110
or	sub reg, reg, (reg/imm)	00111
not	mul reg, (reg/imm)	01000

- **Register Transfer Instruction - mov**: The mov instruction is a 2-address format instruction that can transfer values from one register to another, or can load a register with a constant.
  Example: mov r1, r2; mov r1, 3.

Table 2.3: Register Transfer Instruction

Instructions	**Semantics**	**Opcode**
mov	mov reg, (reg/imm)	01001

- **Shift Instructions**: SimpleRisc has three types of shift instructions: lsl (logical shift left), lsr (logical shift right), and asr (arithmetic shift right). Each of these instructions are in the 3-address format.

  The lsl instruction shifts the value in the first source register to the left. Similarly, lsr, shifts the value in the first source register to the right. The asr instruction performs an arithmetic right shift i.e. it fills up all the MSB positions with the value of the previous sign bit.
  Example: lsl r3, r1, r2; asr r3, r1, 4.

Table 2.4: Shift Instructions

Instructions	Semantics	Opcode
lsl	lsl reg, reg (reg/imm)	01010
lsr	lsr reg, reg, (reg/imm)	01011
asr	asr reg, reg, (reg/imm)	01100

- **Data transfer Instructions**: SimpleRisc has two data transfer instructions load(ld) and store(st). The load instructions loads values from memory into registers, and the store instruction saves values in registers to memory locations.
  Example: ld r1, 12[r2]

  Here, the memory address is computed as the sum of the contents of r2 and the number 12. The ld instructions accesses this memory address, fetches the stored integer and stores it in r1. The store operation does the reverse. It stores the value of r1 into the memory address (r2 + 12)

Table 2.5: Data Transfer Instructions

Instructions	Semantics	Opcode
ld	ld reg, imm[reg]	01110
st	st reg, imm[reg]	01111

- **Branch Instructions**: SimpleRisc has one unconditional branch instruction, b, which makes the program counter jump to the address corresponding to a label in the code and two two conditional branch instructions beq and bgt. The beq instruction stands for "branch if equal". This means that if any preceding cmp instruction has set the E flag, then the PC will branch to the label specified in this instruction. Otherwise, the branch is said to fail, and the processor will proceed to execute the instruction after the branch. Similarly, the bgt instruction stands for "branch if greater than". This branch instruction bases its outcome on the value of the GT flag.

Table 2.6: Branch Instructions

Instructions	Semantics	Opcode
b	b label	10010
beq	beq label	10000
bgt	bgt label	10001

- **Instructions call and ret**: The call instruction takes a single argument – the label of the first instruction of the function. It transfers control to the label and saves the return address in register ra. The ret instructions transfers the contents of register ra to the PC. It is a 0-address instruction because it does not require any operands.

Table 2.7: call and ret Instructions

Instructions	**Semantics**	**Opcode**
call	call label	10011
ret	ret	10100

- **Immediate Instruction**: The Immediate instruction is used when a constant value, or "immediate," is specified directly in the instruction rather than being stored in a register or memory. This allows quick initialization or computation without needing to retrieve the value from memory.

Table 2.8: Immediate Instruction

Instructions	**Semantics**
I (immediate bit)	0/1

- **Nop instruction**: nop is an instruction in the SimpleRisc ISA that performs no operation. It doesn't change the state of the processor or memory. It is useful for inserting delays or aligning instruction cycles.

Table 2.9: nop Instruction

Instructions	**Semantics**	**Opcode**
nop	nop	01101

2.4 Stages of Instruction Processing in a Pipelined Processor

As mentioned previously, there are primarily five stages that an instruction goes through while being processed. They are: Instruction Fetch, Operand Fetch, Execute, Memory Access and Register Write. In a pipelined processor, there is the addition of pipeline registers between each stage as shown in Figure 2.1.

Pipelining: Pipelining is the process of dividing a processor into a set of stages where the stages are ordered one after the other and simultaneously process more than one instruction by assigning an instruction to each stage. [1]

To ensure that multiple instructions proceed to the next stages simultaneously, these pipeline registers are included and are all connected with one common clock source, which read-writes data at the same time.

Example: As shown in Figure 2.1, there are 3 instructions that need to be processed. During the first clock cycle, the instruction moves to the IF unit, and at the end of it, it is transferred to the pipelined register next to it. Then, at the 2nd clock cycle, instruction 1 is then moved to the OF unit, and at the same time, instruction 2 moves to the IF unit. The same process keeps repeating until all instructions are processed.

In a non-pipelined processor, each instruction needs to complete all 5 stages before any new instruction can begin being processed. Therefore, this significantly decreases the overall time that is taken to process multiple instructions.

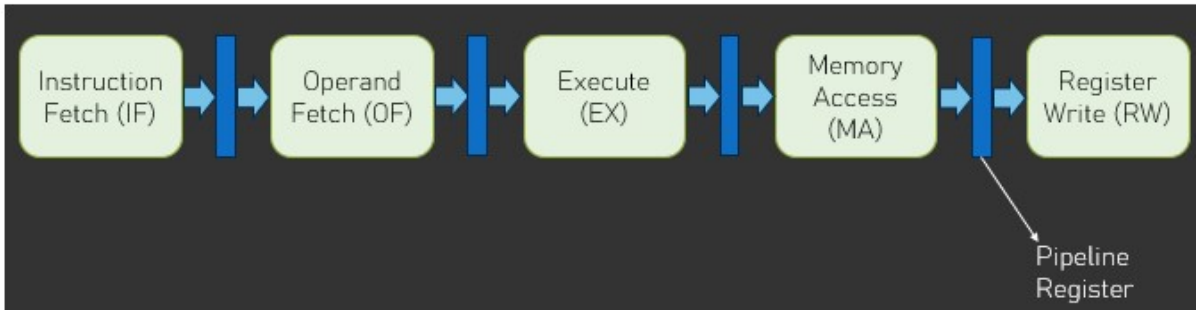

Figure 2.1: Stages of Instruction Processing

2.5 Hardwired and Microprogrammed Design Styles

There are primarily two design styles that are used to program a processor - hardwired and microprogrammed.

Hardwired: Hardwired control refers to a design style where the control signals are implemented using a fixed set of gates, flip-flops, and other digital components, making the generation of control signals fast and efficient.

Microprogrammed: Microprogrammed control uses a small, specialized memory called a microprogram memory to store microinstructions. These microinstructions define the sequence of control signals for each instruction. The control unit reads these microinstructions one by one, generating the control signals needed for executing an instruction.

The key difference between the two lies in how the control signals are generated and how the control logic is organized.

In microprogramming, the control signal is encoded in the microinstruction. It is similar to adding a layer of abstraction that the control signals correspond to. The physical gates and flip-flops are still present, but their activation is controlled through the microprogram memory rather than directly through combinational logic. Each microinstruction is decoded by simpler hardware that enables the required control signals.

Microprogramming is useful where there is a need to update an instruction or add a new instruction, making it more flexible, however this results in an increase in complexity, thus increasing the time each instruction takes to be executed.

For the scope of this project, we have used the hardwired design approach.

Chapter 3

Work Done till Midterm

In the initial phase of the project, we focused on building a strong foundation in Verilog programming by studying the book *Verilog HDL: A Guide to Digital Design and Synthesis*, authored by Samir Palnitkar. This helped us to effectively understand the syntax and semantics of Verilog.

Subsequently, we focused on understanding the principles of Computer Architecture, which provided the foundation for designing the components of our project. We successfully implemented the Instruction Fetch (IF) stage using this knowledge. For this, we referred to the book the book *Basic Computer Architecture* by Prof. Smruti Sarangi [1], as well as taking verilog concepts from the book Verilog HDL: *A Guide to Digital Design and Synthesis* [2].

3.1 Instruction Fetch (IF) Unit

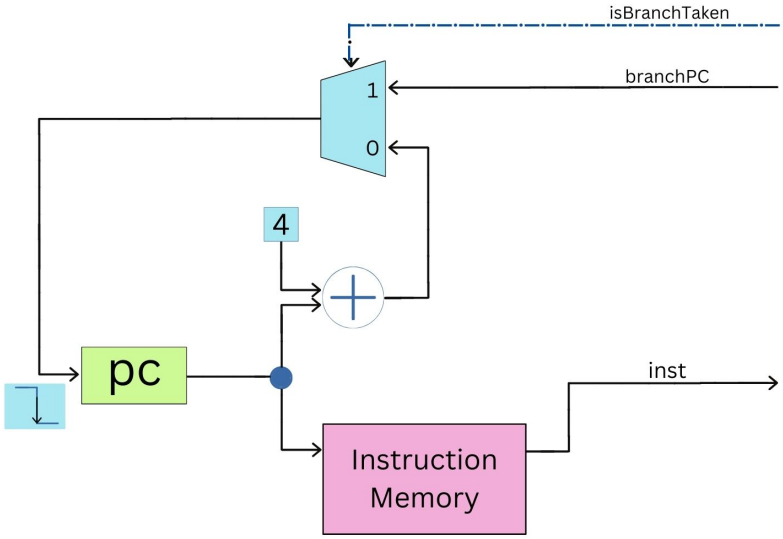

Figure 3.1: IF Stage Block Diagram

Chapter 4

Work done Post Midterm

Following the successful implementation of the Instruction Fetch (IF) stage, the subsequent stages of the pipeline were designed. The development focused on integrating functionality for Operand Fetch (OF), Execute (EX), Memory Access (MA) and Write-Back (WB) stages along with their respective pipeline stages.

Apart from the CPU stages, essential units for proper functioning of the CPU were also designed. These include an Arithmetic and Logic Unit (ALU), a Control Unit, a Register File, a Memory module and a top module where all the stages and units are instantiated ensuring proper data flow across the pipeline. To validate the design, a testbench was implemented, testing the processor's ability to handle instructions efficiently.

4.1 Pipelined Data Path

4.1.1 Block Diagram

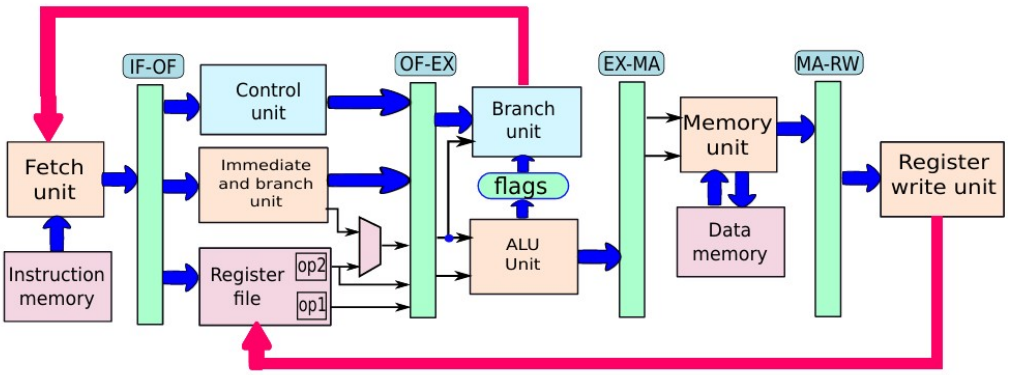

Figure 4.1: Pipelined Data Path Block Diagram

4.1.2 Instruction Fetch (IF) Unit

The IF unit is responsible for retrieving the next instruction from memory based on the current Program Counter (PC). The PC provides the starting address of a 32-bit

SimpleRisc instruction stored in instruction memory. If the instruction is not a branch, the PC is incremented by 4 to point to the next sequential instruction. In the case of a branch instruction that is taken, the branch target address, calculated in the Execute (EX) unit or Operand Fetch (OF) unit, updates the PC. A multiplexer selects between the incremented PC and the branch target (branchPC) based on the isBranchTaken control signal. The updated PC is stored back into the PC register and sent to the instruction memory to fetch the instruction contents. These contents are passed to the Operand Fetch (OF) unit for further decoding and processing.

4.1.3 Operand Fetch (OF) Unit

The OF unit decodes the 32-bit instruction fetched by the IF unit into its constituent fields: source registers (rs1 and rs2), destination register (rd), opcode, and immediate value (imm). For branch instructions, it calculates the branch target address (branchTarget) using the offset embedded in the instruction. This offset, extracted from bits [1:27], is shifted left by 2 (to align with word boundaries), sign-extended to 32 bits, and added to the PC. Similarly, immediate values are derived by extracting the 16-bit imm field, applying the appropriate modifiers, and sign-extending or zero-padding it to 32 bits, creating immx. The OF unit also retrieves the values of source registers from the register file using two read ports.

For most instructions, the first source register (rs1) is read, but for ret instructions, the return address register (ra) is selected. The second source register (rs2) is used for all instructions except store, which uses the destination register (rd). Operands (op1, op2), branch target (branchTarget), and immediate value (immx) are computed and passed to the Execute (EX) unit, along with control signals derived from the opcode.

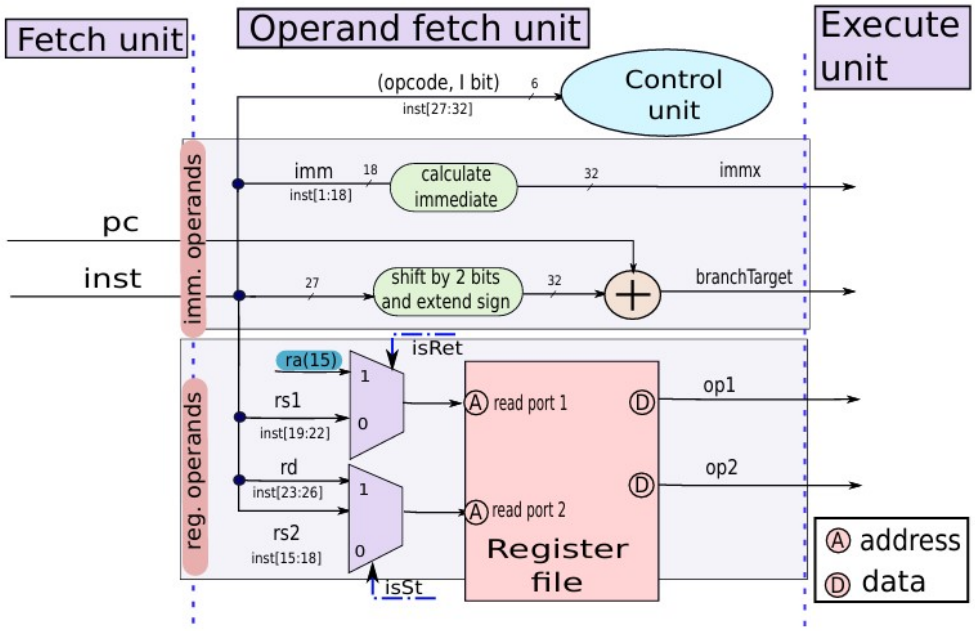

Figure 4.2: Operand Fetch Unit

4.1.4 Execute Unit

The EX unit processes the operands fetched by the OF unit and performs the operation specified in the instruction, utilizing the Arithmetic and Logic Unit (ALU) and branch decision logic. The first operand, op1, is always sourced from the register file, while the second operand, op2, is selected via a multiplexer based on the isImmediate signal. If isImmediate is high, the immediate value (immx) from the OF unit is chosen; otherwise, the second register operand (op2) is used. The ALU, controlled by signals (aluSignals) generated by the control unit, executes arithmetic, logical, comparison, or shift operations.

Branch instructions are evaluated using the branch decision logic. The branch target address (branchTarget) or return address (op1 for ret instructions) is selected via a multiplexer controlled by isRet. The branch outcome is determined by AND-ing the instruction-specific branch signal (isBeq, isBgt) with the corresponding flag (E, GT) and OR-ing these results with isUBranch for unconditional branches. If the branch is taken, the isBranchTaken signal directs the IF unit to update the Program Counter (PC) with branchPC.

4.1.5 Memory Access Unit

The Memory Access (MA) unit processes memory instructions by utilizing the address computed in the EX unit as aluResult. For load (ld) instructions, the address is stored in the Memory Address Register (MAR), calculated as the sum of the immediate value (immx) and the content of rs1. The MA unit fetches 4 bytes from data memory starting at the computed address, and the retrieved value (ldResult) is buffered in the Memory Data Register (MDR) before being passed to the Write-Back (RW) unit. For store (st) instructions, the source register data (op2) fetched in the OF stage is written to the MDR and subsequently stored in the memory location specified by the computed address in MAR. The control signals isLd and isSt govern whether the MA unit performs a load or store operation. If neither signal is asserted, the MA unit remains idle.

4.1.6 Register Write Unit

The RW unit finalizes instruction execution by writing results back to the register file. Depending on the instruction type, the RW unit selects the appropriate data source via a multiplexer controlled by isLd (for load instructions) and isCall (for call instructions). The sources include aluResult (for ALU instructions), ldResult (for load instructions), or PC + 4 (for return address in call instructions). The selected data is written to the destination register (rd) or the return address register (ra) using the write port of the register file, enabled by the isWb signal. Non-register-write instructions like st, cmp, and nop do not assert isWb.

4.1.7 Control Unit

The control unit is responsible for generating the 22 control signals required to orchestrate the pipeline stages, from IF to RW. These signals are determined based on the opcode (5 bits) and the immediate bit (1 bit) of the instruction, forming a 6-bit input to the

control logic. For instance, to enable the ALU for an addition operation, the opcode is checked against 00000, and if matched, the isAdd signal is asserted. Similarly, for memory instructions like ld or st, the control signals isLd or isSt are generated to activate the MA unit for reading or writing data memory.

While the control unit governs most of the pipeline, the isBranchTaken signal is generated independently by the branch unit in the EX stage. During instruction flow, after the opcode is decoded in the OF stage, the control signals are propagated to subsequent units, guiding operations in the EX (e.g., ALU operations), MA (e.g., memory read/write), and RW (e.g., write-back enable) stages. For example, in a ld instruction, the control unit activates isLd in the MA stage to fetch data from memory and isWb in the RW stage to write the data into the destination register.

4.2 Testing

4.2.1 Encoding

SimpleRisc employs a 32-bit fixed-length instruction format, divided into multiple fields to encode various instruction types. The five most significant bits (bits 31-28) specify the opcode, uniquely identifying the instruction type. Based on the number of operands and addressing mode, SimpleRisc uses three primary instruction formats: branch format, register format, and immediate format.

Branch Format: Used for 1-address instructions (b, beq, bgt, call). The opcode occupies the first 5 bits, while the remaining 27 bits encode a PC-relative offset in units of 4 bytes, allowing for efficient addressing of branch targets within a 29-bit range.

Register Format: Utilized for 3-address instructions (add, sub, mul, div, etc.) with register operands. It includes fields for the destination register (rd, bits 23–26) and two source registers (rs1, bits 19–22, and rs2, bits 15–18).

Immediate Format: Similar to the register format but used for instructions where the second operand is an immediate value. The immediate bit (I, bit 27) distinguishes between register and immediate operands. The immediate value is stored in bits 1–18, with two modifier bits (bits 17–18) specifying default, zero-extended (u), or left-shifted (h) encoding.

Special cases include 0-address instructions (ret, nop), which only require the opcode field, leaving the remaining 27 bits unused. By leveraging a consistent structure and efficient encoding techniques, SimpleRisc ensures streamlined decoding, flexible addressing, and optimized utilization of instruction fields.

4.2.2 Test Program

To be able to test our program, we have used a memory file called "Input Memory.mem" where we give our instruction based on the encoding mentioned in section 4.2.1 in hexadecimal format. Upon running the testbench, the instructions are passed on to the internal memory of the CPU so that every time there is a change in the program which needs to be implemented, it is done directly through the Input Memory file.

Test Program 1: To test the efficient working of the CPU, we first used a simple test program which uses 3 mov instructions and then multiplies and adds the moved numbers and produces the output in another register.

```
0x0   0x4cc00009   //mov r3, 9
0x4   0x4c400009   //mov r1, 9
0x8   0x4c800005   //mov r2, 5
0xc   0x68000000   //nop
0x10   0x68000000   //nop
0x14   0x68000000   //nop
0x18   0x1144c000   //mul r5, r1, r3
0x1c   0x68000000   //nop
0x20   0x68000000   //nop
0x24   0x68000000   //nop
0x28   0x00148000   //add r0, r5, r2
```

Figure 4.3: Test Program 1

Test Program 2: To further test the ability of our processor to handle loop based or branch instructions, we have used a program to find the factorial of a given number "n".

```
0x0    0x4d800005    //mov r6, 5
0x4    0x4c800001    //mov r2, 1
0x8    0x68000000    //nop
0xc    0x68000000    //nop
0x10    0x68000000    //nop
0x14    0x2c180001    //cmp r6, 1
0x18    0x68000000    //nop
0x1c    0x68000000    //nop
0x20    0x68000000    //nop
0x24    0x88000008    //bgt loop_start
0x28    0x68000000    //nop
0x2c    0x68000000    //nop
0x30    0x68000000    //nop
0x34    0x90000014    //b end
0x38    0x68000000    //nop
0x3c    0x68000000    //nop
0x40    0x68000000    //nop
```

Figure 4.4: Test Program 2

4.3 Results

Based on the test programs and testbench provided in section 4.2, upon running simulations, the following waveforms were generated. As seen in Figure 4.5, at every new clock cycle, the first 3 instructions move in the Instruction Fetch Unit of the processor. After that, as the next instruction is a nop instruction, the program now waits for 10ns per nop instruction.

These nop instructions are crucial as without it, if the program were to execute the next instruction, which is mul r5, r1, r3, a correct value will not be generated, since the registers involved in the initial instructions have not been updated in the register memory. Therefore, it needs to wait until the register values are properly updated.

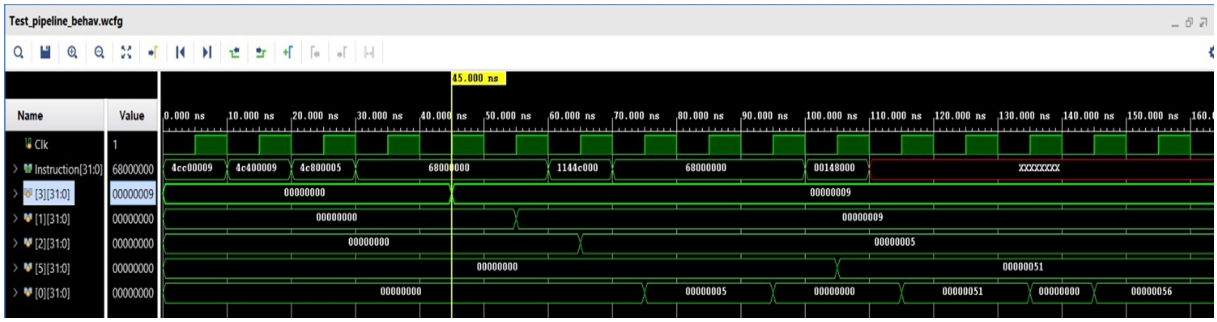

Figure 4.5: Test Program 1 Waveform

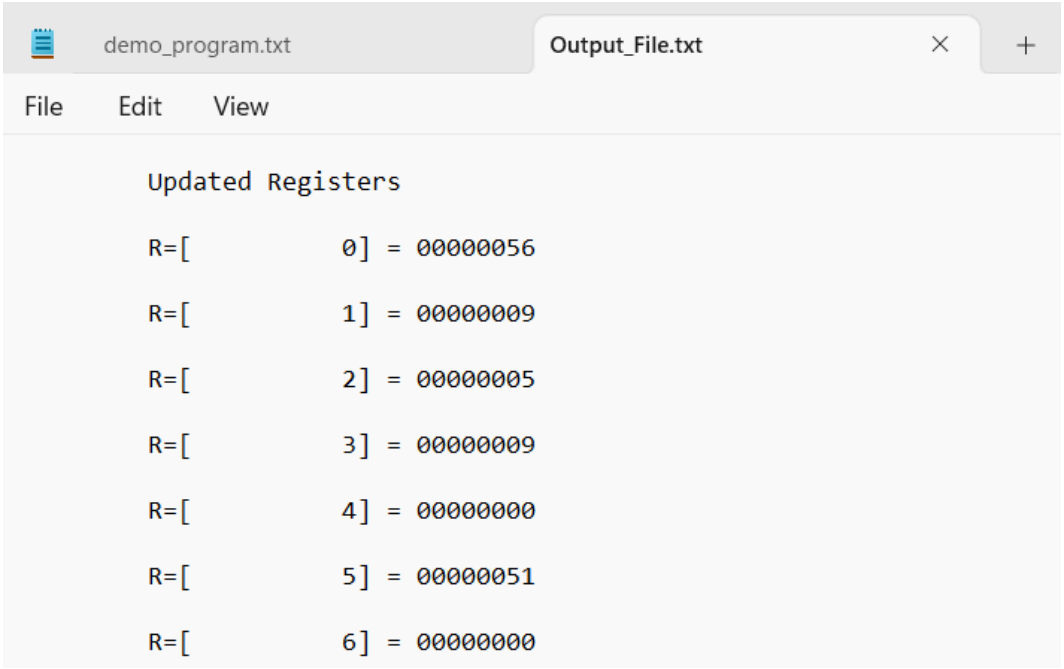

Figure 4.6: Updated Registers for Test program 1

Each instruction takes 45ns to completely go through all five stages and update the registers. The updated register values can be seen in the output file Updated Data File.txt, which is saved in the working directory of the project.

The same can be observed for test program 2 in Figure 4.7 and Figure 4.8.

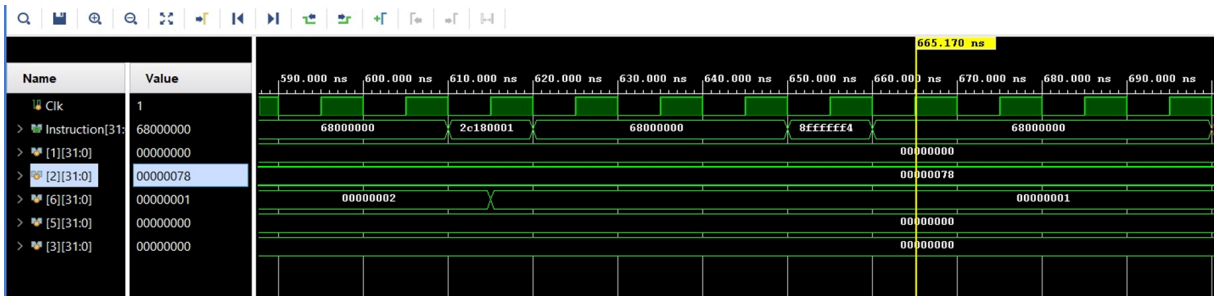

Figure 4.7: Test Program 2 Waveform

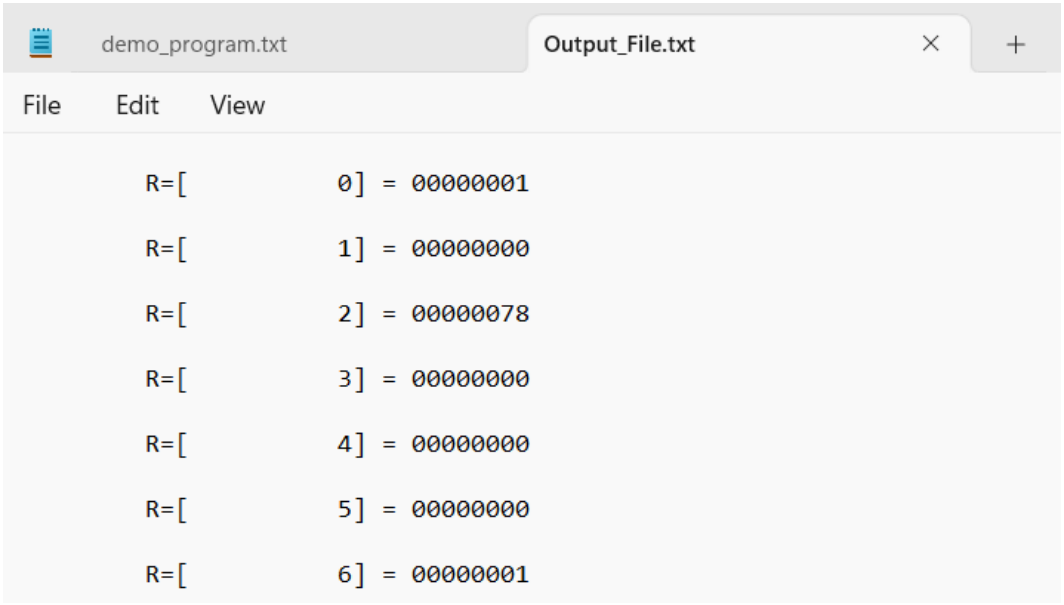

Figure 4.8: Updated Registers for Test program 2

The Verilog codes for all stages and units used in this project can be accessed through this link: *https://github.com/llama-lore/SimpleRisc-pipelined-processor*

Chapter 5

Future Work

1. There are several areas for improvement and extension to increase the functionality, performance, and real-world applications of this project.

2. Currently, the design does not include mechanisms for detecting or resolving hazards. Future work can focus on implementing and refining these mechanisms to resolve control or data hazards.

3. This design only works for the simulation of the processor. Transitioning it into a synthesisable HDL implementation is a crucial step for hardware realization. This would enable deployment on FPGAs, allowing physical validation.

4. Developing an assembler for the processor instruction set would significantly streamline workflow. This tool would automate the conversion of assembly code into machine code, reducing errors and enabling more efficient software testing.

5. The design can further be enhanced to support floating point arithmetic.

Bibliography

[1] S. R. Sarangi, *Basic Computer Architecture*. White Falcon Publishing, 2021.

[2] S. Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis*. SunSoft Press, 1996.

Chapter 6

Appendix

6.1 Verilog Codes

6.1.1 IF Unit

```
`timescale 1ns / 1ps

module IF(
    input Clk,
    input [31:0] Branch_PC,
    input IsBranchTaken,
    output [31:0] Instruction,
    output [31:0] PC_Current
);

    integer x = 0;
    reg [31:0] Instruction;
    reg [31:0] PC_Current, PC_Temp, PC;
    wire [31:0] Branch_PC;
    wire Clk;
    wire IsBranchTaken;

    initial
    begin
        PC = 32'd0;
    end

    always @(negedge Clk)
    begin
        if (x == 0)
        begin
            PC_Temp = PC;
            PC_Current = PC;
        end
```

```
        for (x = 0; x < 4; x = x + 1)
        begin
            Instruction[8 * x +: 8] = PL.M.Memory[PC_Temp];
            PC_Temp = PC_Temp + 1;
        end

        if (IsBranchTaken == 1)
        begin
            PC_Current = Branch_PC;
        end
        else
        begin
            PC_Current = PC_Temp - 4;
        end
    end

    always @(IsBranchTaken)
    begin
        if (IsBranchTaken == 1)
        begin
            PC_Temp = Branch_PC;
        end
    end

endmodule
```

6.1.2 Pipeline Register IF-OF

```
`timescale 1ns / 1ps

module IF_OF (
    input Clk
);

    reg [31:0] Instruction, PC_Current;
    wire Clk;


    always @(negedge Clk)
        begin
            #9.99
            PC_Current  <= PL.PC_Current;
            Instruction <= PL.Instruction;
        end
```

endmodule

6.1.3 OF Unit

```
`timescale 1ns / 1ps

module OF(
    input IsRet,
    input IsSt,
    input [31:0] Instruction,
    input [31:0] PC_Current,
    output [31:0] op1,
    output [31:0] op2,
    output [4:0] Opcode,
    output [3:0] Rd,
    output [31:0] Branch_Target,
    output I,
    output [31:0] Immd

);

    wire [31:0] Instruction, PC_Current;
    wire IsRet, IsSt;
    reg [3:0] Rd;
    reg signed [26:0] Branch_Temp;
    reg [4:0] Opcode;
    reg I;
    reg signed [15:0] Imm_Temp;
    reg [31:0] Immd, Branch_Target, op1, op2;;

    always @(*)
    begin
        I = Instruction[26];
        Branch_Temp = Instruction[26:0];
        Rd = Instruction[25:22];
        Imm_Temp = Instruction[15:0];
        Opcode = Instruction[31:27];

        if (I == 1)
        begin
            if (Instruction[17:16] == 2'b00)
            begin
                Immd = Imm_Temp;
            end
            else if (Instruction[17:16] == 2'b10)
            begin
```

```
                Immd = 32'd0;
                Immd = Immd + Instruction[15:0];
                Immd = Immd << 16;
            end
            else if (Instruction[17:16] == 2'b01)
            begin
                Immd = 32'd0;
                Immd = Immd + Instruction[15:0];
            end
        end

        Branch_Temp   = Branch_Temp << 2;
        Branch_Target = Branch_Temp;
        Branch_Target = Branch_Target + PC_Current;

        if (IsSt == 1)
        begin
            PL.a2 = Instruction[25:22];
        end
        else
        begin
            PL.a2 = Instruction[17:14];
        end

        if (IsRet == 1)
        begin
            PL.a1 = 4'b1111;
        end
        else
        begin
            PL.a1 = Instruction[21:18];
        end

        op1 = PL.d1;
        op2 = PL.d2;
    end

endmodule
```

6.1.4 Pipeline Register OF-EX

```
`timescale 1ns / 1ps

module OF_EX (
    input Clk
);
```

```
    reg IsSt, IsLd, IsBeq, IsBgt, IsRet, IsImmediate, IsWb,
       IsUBranch, IsCall;
    reg [4:0] AluSignal;
    reg [31:0] Instruction, PC_Current, Branch_Target, op2, op1,
        Immd;
    wire Clk;

    always @(negedge Clk)
    begin
        #9.99
        Instruction <= PL.R1.Instruction;
        IsBeq = PL.IsBeq;
        IsRet = PL.IsRet;
        IsBgt = PL.IsBgt;
        IsImmediate = PL.IsImmediate;
        IsLd = PL.IsLd;
        IsWb = PL.IsWb;
        IsUBranch = PL.IsUBranch;
        AluSignal = PL.AluSignal;
        IsCall = PL.IsCall;
        PC_Current <= PL.R1.PC_Current;
        IsSt = PL.IsSt;
        op1 = PL.op1;
        op2 = PL.op2;
        Branch_Target  = PL.Branch_Target;
        Immd = PL.Immd;
    end

endmodule
```

6.1.5 EX Unit

```
`timescale 1ns / 1ps

module EX(
    input [4:0] AluSignal,
    input [31:0] op1,
    input [31:0] op2,
    input IsImmediate,
    input IsUBranch,
    input IsBeq,
    input IsRet,
    input IsBgt,
    input [31:0] Immd,
    input [31:0] Branch_Target,
```

```
    output [31:0] Branch_PC,
    output IsBranchTaken,
    output [31:0] Alu_Result1
);

    wire [31:0] op1, op2, Immd, Branch_Target;
    wire [4:0] AluSignal;
    wire IsRet, IsBeq, IsBgt, IsUBranch, IsImmediate;
    reg IsBranchTaken;
    reg y1, y2;
    reg [31:0] Alu_Result1, Branch_PC;

    always @(*)
    begin
        if (IsBgt == 1 && PL.Flags[1] == 1)
        begin
            y2 = 1;
        end
        else
        begin
            y2 = 0;
        end

        if (IsBeq == 1 && PL.Flags[0] == 1)
        begin
            y1 = 1;
        end
        else
        begin
            y1 = 0;
        end

        if (IsRet == 1)
        begin
            Branch_PC = op1;
        end
        else
        begin
            Branch_PC = Branch_Target;
        end

        if (IsUBranch == 1 || y1 == 1 || y2 == 1)
        begin
            IsBranchTaken = 1;
        end
        else
```

```
        begin
            IsBranchTaken = 0;
        end

        PL.A = op1;
        if (IsImmediate == 0)
        begin
            PL.B = op2;
        end
        else if (IsImmediate == 1)
        begin
            PL.B = Immd;
        end

        Alu_Result1 = PL.ALU.Alu_Result;
    end

endmodule
```

6.1.6 Pipeline Register EX-MA

```
`timescale 1ns / 1ps

module EX_MA (
    input Clk
);

    reg [31:0] Instruction, PC_Current, Branch_Target, op2,
       Alu_Result1;
    reg [4:0] AluSignal;
    reg IsSt, IsLd, IsBeq, IsBgt, IsRet, IsImmediate, IsWb,
       IsUBranch, IsCall;
    wire Clk;

    always @(negedge Clk)
    begin
        #9.99
        Instruction <= PL.R2.Instruction;
        Branch_Target <= PL.R2.Branch_Target;
        PC_Current <= PL.R2.PC_Current;
        op2 <= PL.R2.op2;
        IsRet <= PL.R2.IsRet;
        IsBeq <= PL.R2.IsBeq;
        IsLd <= PL.R2.IsLd;
        IsSt <= PL.R2.IsSt;
        IsBgt <= PL.R2.IsBgt;
```

```
        IsUBranch <= PL.R2.IsUBranch;
        IsImmediate <= PL.R2.IsImmediate;
        AluSignal <= PL.R2.AluSignal;
        IsWb <= PL.R2.IsWb;
        IsCall <= PL.R2.IsCall;
        Alu_Result1 <= PL.Alu_Result1;
    end

endmodule
```

6.1.7 MA Unit

```
`timescale 1ns / 1ps

module MA (
    input [31:0] Alu_Result1,
    input [31:0] op2,
    input IsSt,
    input IsLd,
    output [31:0] Ld_Result
);

    reg [31:0] Ld_Result;
    wire [31:0] op2, Alu_Result1;
    wire IsLd, IsSt;

    always @(*)
    begin
        if (IsSt == 1)
        begin
            PL.rw = 1;
            PL.a_m2 = Alu_Result1;
            PL.d_m3 = op2;
        end

        else if (IsLd == 1)
        begin
            PL.a_m1 = Alu_Result1;
            Ld_Result = PL.d_m1;
            PL.rw = 0;
        end
    end

endmodule
```

6.1.8 Pipeline Register MA-WB

```
`timescale 1ns / 1ps

module MA_WB (
    input Clk
);

    wire Clk;
    reg [31:0] Instruction, PC_Current;
    reg [31:0] Branch_Target;
    reg [31:0] Alu_Result1, Ld_Result;
    reg [4:0] AluSignal;
    reg [3:0] Rd;
    reg IsSt, IsLd, IsBeq, IsBgt, IsRet, IsImmediate, IsWb,
       IsUBranch, IsCall;

    always @(negedge Clk)
    begin
        #9.99;
        Instruction <= PL.R3.Instruction;
        Alu_Result1 <= PL.R3.Alu_Result1;
        Branch_Target <= PL.R3.Branch_Target;
        PC_Current <= PL.R3.PC_Current;
        IsBgt <= PL.R3.IsBgt;
        IsImmediate <= PL.R3.IsImmediate;
        IsCall <= PL.R3.IsCall;
        IsSt <= PL.R3.IsSt;
        IsUBranch <= PL.R3.IsUBranch;
        IsLd <= PL.R3.IsLd;
        IsRet <= PL.R3.IsRet;
        IsBeq <= PL.R3.IsBeq;
        AluSignal <= PL.R3.AluSignal;
        IsWb <= PL.R3.IsWb;
        Ld_Result = PL.Ld_Result;
        Rd <= PL.R3.Instruction[25:22];
    end

endmodule
```

6.1.9 WB Unit

```
`timescale 1ns / 1ps

module WB (
    input IsLd,
```

```
    input IsCall,
    input IsWb,
    input [31:0] Ld_Result,
    input [31:0] PC_Current,
    input [3:0] Rd,
    input [31:0] Alu_Result1
);

    wire [31:0] Ld_Result, PC_Current;
    wire [31:0] Alu_Result1;
    wire IsWb, IsCall, IsLd;
    wire [3:0] Rd;

    always @(*)
    begin
        if (IsWb == 1)
        begin
            if (IsCall == 0)
            begin
                PL.a3 = Rd;
            end
            else if (IsCall == 1)
            begin
                PL.a3 = 4'b1111;
            end

            if (IsLd == 0 && IsCall == 0)
            begin
                PL.d3 = Alu_Result1;
            end
            else if (IsLd == 0 && IsCall == 1)
            begin
                PL.d3 = PC_Current + 4;
            end
            else if (IsLd == 1 && IsCall == 0)
            begin
                PL.d3 = Ld_Result;
            end
        end
    end

endmodule
```

6.1.10 Memory Unit

```
`timescale 1ns / 1ps
```

```
module Memory(
    input rw,
    input [31:0] a_m1,
    input [31:0] a_m2,
    input [31:0] d_m3,
    output [31:0] d_m1,
    output [31:0] d_m2
);

    integer x;
    reg [31:0] z1, z2;
    reg [7:0] Memory [16'h1000:0];
    reg [31:0] d_m1, d_m2;
    wire rw;
    wire [31:0] a_m1, a_m2, a_m3;

    always @(a_m1)
    begin
        z1 = a_m1;
        for (x = 0; x < 4; x = x + 1)
        begin
            d_m1[8 * x +: 8] = Memory[z1];
            z1 = z1 + 1;
        end
    end

    always @(a_m2)
    begin
        z2 = a_m2;
        if (rw == 1)
        begin
            for (x = 0; x < 4; x = x + 1)
            begin
                Memory[z2] <= d_m3[8 * x +: 8];
                z2 = z2 + 1;
            end
        end
        else if (rw == 0)
        begin
            for (x = 0; x < 4; x = x + 1)
            begin
                d_m2[8 * x +: 8] = Memory[z2];
                z2 = z2 + 1;
            end
        end
```

```
    end

endmodule
```

6.1.11 Control Unit

```
`timescale 1ns / 1ps

module Control (
    input I,
    input [4:0] Opcode,
    output [4:0] AluSignal,
    output IsImmediate,
    output IsRet,
    output IsBgt,
    output IsLd,
    output IsSt,
    output IsUBranch,
    output IsBeq,
    output IsCall,
    output IsWb
);

    reg [4:0] AluSignal;
    reg IsSt, IsLd, IsBeq, IsRet, IsImmediate, IsWb, IsUBranch,
       IsCall, IsBgt;
    wire I;
    wire [4:0] Opcode;

    always @(*)
    begin
        if (I == 0)
            IsImmediate = 0;
        else
            IsImmediate = 1;

        case (Opcode)
            5'b00000:
                begin
                    AluSignal = Opcode;
                    IsWb = 1;
                    IsSt = 0;
                    IsLd = 0;
                    IsRet = 0;
                    IsUBranch = 0;
                    IsCall = 0;
```

```
            IsBeq = 0;
            IsBgt = 0;
        end

    5'b00001:
        begin
            if (I == 1 && PL.Immd[31])
                AluSignal = 5'b00000;
            else
                AluSignal = Opcode;

            IsWb = 1;
            IsSt = 0;
            IsLd = 0;
            IsRet = 0;
            IsUBranch = 0;
            IsCall = 0;
            IsBeq = 0;
            IsBgt = 0;
        end

    5'b00010:
        begin
            AluSignal = Opcode;
            IsWb = 1;
            IsSt = 0;
            IsLd = 0;
            IsRet = 0;
            IsUBranch = 0;
            IsCall = 0;
            IsBeq = 0;
            IsBgt = 0;
        end

    5'b00011:
        begin
            AluSignal = Opcode;
            IsWb = 1;
            IsSt = 0;
            IsLd = 0;
            IsRet = 0;
            IsUBranch = 0;
            IsCall = 0;
            IsBeq = 0;
            IsBgt = 0;
        end
```

```
5'b00100:
    begin
        AluSignal = Opcode;
        IsWb = 1;
        IsSt = 0;
        IsLd = 0;
        IsRet = 0;
        IsUBranch = 0;
        IsCall = 0;
        IsBeq = 0;
        IsBgt = 0;
    end

5'b00101:
    begin
        AluSignal = Opcode;
        IsWb = 0;
        IsSt = 0;
        IsLd = 0;
        IsRet = 0;
        IsUBranch = 0;
        IsCall = 0;
        IsBeq = 0;
        IsBgt = 0;
    end

5'b00110:
    begin
        AluSignal = Opcode;
        IsWb = 1;
        IsSt = 0;
        IsLd = 0;
        IsRet = 0;
        IsUBranch = 0;
        IsCall = 0;
        IsBeq = 0;
        IsBgt = 0;
    end

5'b00111:
    begin
        AluSignal = Opcode;
        IsWb = 1;
        IsSt = 0;
        IsLd = 0;
```

```
            IsRet = 0;
            IsUBranch = 0;
            IsCall = 0;
            IsBeq = 0;
            IsBgt = 0;
        end

    5'b01000:
        begin
            AluSignal = Opcode;
            IsWb = 1;
            IsSt = 0;
            IsLd = 0;
            IsRet = 0;
            IsUBranch = 0;
            IsCall = 0;
            IsBeq = 0;
            IsBgt = 0;
        end

    5'b01001:
        begin
            AluSignal = Opcode;
            IsWb = 1;
            IsSt = 0;
            IsLd = 0;
            IsRet = 0;
            IsUBranch = 0;
            IsCall = 0;
            IsBeq = 0;
            IsBgt = 0;
        end

    5'b01010:
        begin
            AluSignal = Opcode;
            IsWb = 1;
            IsSt = 0;
            IsLd = 0;
            IsRet = 0;
            IsUBranch = 0;
            IsCall = 0;
            IsBeq = 0;
            IsBgt = 0;
        end
```

```
5'b01011:
    begin
        AluSignal = Opcode;
        IsWb = 1;
        IsSt = 0;
        IsLd = 0;
        IsRet = 0;
        IsUBranch = 0;
        IsCall = 0;
        IsBeq = 0;
        IsBgt = 0;
    end

5'b01100:
    begin
        AluSignal = Opcode;
        IsWb = 1;
        IsSt = 0;
        IsLd = 0;
        IsRet = 0;
        IsUBranch = 0;
        IsCall = 0;
        IsBeq = 0;
        IsBgt = 0;
    end

5'b01101:
    begin
        AluSignal = Opcode;
        IsWb = 1;
        IsSt = 0;
        IsLd = 0;
        IsRet = 0;
        IsUBranch = 0;
        IsCall = 0;
        IsBeq = 0;
        IsBgt = 0;
    end

5'b01110:
    begin
        AluSignal = 5'b00000;
        IsLd = 1;
        IsWb = 1;
        IsSt = 0;
        IsRet = 0;
```

```
            IsUBranch = 0;
            IsCall = 0;
            IsBeq = 0;
            IsBgt = 0;
        end

    5'b01111:
        begin
            AluSignal = 5'b00000;
            IsSt = 1;
            IsWb = 0;
            IsLd = 0;
            IsRet = 0;
            IsUBranch = 0;
            IsCall = 0;
            IsBeq = 0;
            IsBgt = 0;
        end

    5'b10000:
        begin
            AluSignal = 5'b01101;
            IsBeq = 1;
            IsWb = 0;
            IsSt = 0;
            IsLd = 0;
            IsRet = 0;
            IsUBranch = 0;
            IsCall = 0;
        end

    5'b10001:
        begin
            AluSignal = 5'b01101;
            IsBgt = 1;
            IsWb = 0;
            IsSt = 0;
            IsLd = 0;
            IsRet = 0;
            IsUBranch = 0;
            IsCall = 0;
        end

    5'b10010:
        begin
            AluSignal = 5'b01101;
```

```
                    IsUBranch = 1;
                    IsWb = 0;
                    IsSt = 0;
                    IsLd = 0;
                    IsRet = 0;
                    IsCall = 0;
                end

            5'b10011:
                begin
                    AluSignal = 5'b01101;
                    IsUBranch = 1;
                    IsCall = 1;
                    IsWb = 1;
                    IsSt = 0;
                    IsLd = 0;
                    IsRet = 0;
                end

            5'b10100:
                begin
                    AluSignal = 5'b01101;
                    IsUBranch = 1;
                    IsRet = 1;
                    IsWb = 0;
                    IsSt = 0;
                    IsLd = 0;
                    IsCall = 0;
                end

        endcase
    end

endmodule
```

6.1.12 ALU Unit

```
`timescale 1ns / 1ps

module ALU(
    input [31:0] A,
    input [31:0] B,
    input [4:0] AluSignal,
    output [31:0] Alu_Result,
    output [1:0] Flags
);
```

```
wire signed [31:0] A, B;
wire signed [31:0] A_signed;
wire [4:0] AluSignal;
reg [31:0] Alu_Result;
reg [1:0] Flags;

assign A_signed = A;

always @(*)
begin
    case (AluSignal)
        5'b00000:
            Alu_Result = A + B;
        5'b00001:
            Alu_Result = A - B;
        5'b00010:
            Alu_Result = A * B;
        5'b00011:
            Alu_Result = A / B;
        5'b00100:
            Alu_Result = A % B;
        5'b00101:
        begin
            if (A > B)
            begin
                Flags[1] = 1'b1;
                Flags[0] = 1'b0;
            end
            else if (A == B)
            begin
                Flags[1] = 1'b0;
                Flags[0] = 1'b1;
            end
            else
            begin
                Flags[1] = 1'b0;
                Flags[0] = 1'b0;
            end
        end

        5'b00110:
            Alu_Result = A & B;
        5'b00111:
            Alu_Result = A | B;
        5'b01000:
```

```
                Alu_Result = ~A;
            5'b01001:
                Alu_Result = B;
            5'b01010:
                Alu_Result = A << B;
            5'b01011:
                Alu_Result = A >> B;
            5'b01100:
            begin
                Alu_Result = A_signed >>> B; // asr (signed)
            end
            5'b01101:
            begin

            end
            default:
            begin
                Alu_Result = 32'd0;
                Flags[1] = 1'b0;
                Flags[0] = 1'b0;
            end
        endcase
    end

endmodule
```

6.1.13 Register File

```
`timescale 1ns / 1ps

module Register_File(
    input Reset,
    input Clk,
    input IsWb,
    input [3:0] a1,
    input [3:0] a2,
    input [3:0] a3,
    input [31:0] d3,
    output [31:0] d1,
    output [31:0] d2
);
    wire Reset, Clk;
    wire IsWb;
    wire [3:0] a1, a2, a3;
    wire [31:0] d3;
    reg [31:0] Regs [15:0];
```

```
    reg [31:0] d1, d2;
    reg [15:0] x;

    always @(Reset)
    begin
        if (Reset == 1)
        begin
            for (x = 0; x < 16; x = x + 1)
            begin
                if (x != 14)
                    Regs[x] = 32'b0;
                else
                    Regs[x] = 32'h1000;
            end
        end
    end

    always @(posedge Clk)
    begin
        if (Reset != 1 && IsWb == 1)
            Regs[a3] <= d3;
    end

    always @(Clk or a1 or a2)
    begin
        if (Reset != 1 && Clk == 0)
        begin
            d1 = Regs[a1];
            d2 = Regs[a2];
        end
    end

endmodule
```

6.1.14 Pipeline

```
`timescale 1ns / 1ps

`include "IF.v"
`include "IF_OF.v"
`include "OF.v"
`include "OF_EX.v"
`include "EX.v"
`include "EX_MA.v"
`include "MA.v"
`include "MA_WB.v"
```

```
`include "WB.v"
`include "Control.v"
`include "Register_File.v"
`include "Memory.v"
`include "ALU.v"


module Pipeline (
    input Clk
);

    integer x1, x2, x3;
    wire Clk;
    wire [31:0] Instruction, Immd, op1, op2, Branch_Target;
    wire [31:0] Alu_Result, Alu_Result1, Ld_Result, d_m1, d_m2,
       d1, d2;
    wire IsLd, IsSt, IsBeq, IsBgt, IsRet, IsImmediate, IsWb,
       IsUBranch, IsCall, I, IsBranchTaken;
    wire [31:0] Branch_PC, PC_Current;
    wire [4:0] AluSignal, Opcode;
    wire [3:0] Rd;
    wire [1:0] Flags;
    reg [31:0] A, B, a_m1, a_m2, d_m3, d3, PC;
    reg [3:0] a1, a2, a3;
    reg rw, Reset;


    IF IF(
        .Clk(Clk),
        .Instruction(Instruction),
        .IsBranchTaken(IsBranchTaken),
        .PC_Current(PC_Current),
        .Branch_PC(Branch_PC)
    );

    IF_OF R1(
        .Clk(Clk)
    );

    OF OF(
        .Instruction(R1.Instruction),
        .Branch_Target(Branch_Target),
        .PC_Current(R1.PC_Current),
        .I(I),
        .op1(op1),
        .op2(op2),
```

```
        .Immd(Immd),
        .Opcode(Opcode),
        .IsRet(IsRet),
        .IsSt(IsSt),
        .Rd(Rd)
);

OF_EX_R2(
        .Clk(Clk)
);

EX EX(
        .Branch_Target(R2.Branch_Target),
        .Branch_PC(Branch_PC),
        .Immd(R2.Immd),
        .Alu_Result1(Alu_Result1),
        .op1(R2.op1),
        .op2(R2.op2),
        .AluSignal(R2.AluSignal),
        .IsRet(R2.IsRet),
        .IsBeq(R2.IsBeq),
        .IsBgt(R2.IsBgt),
        .IsUBranch(R2.IsUBranch),
        .IsImmediate(R2.IsImmediate),
        .IsBranchTaken(IsBranchTaken)
);

EX_MA_R3(
        .Clk(Clk)
);

MA MA(
        .Alu_Result1(R3.Alu_Result1),
        .op2(R3.op2),
        .IsLd(R3.IsLd),
        .IsSt(R3.IsSt),
        .Ld_Result(Ld_Result)
);

MA_WB_R4(
        .Clk(Clk)
);

WB WB(
        .PC_Current(R4.PC_Current),
        .Alu_Result1(R4.Alu_Result1),
```

```
        .IsWb(R4.IsWb),
        .IsCall(R4.IsCall),
        .IsLd(R4.IsLd),
        .Rd(R4.Rd),
        .Ld_Result(R4.Ld_Result)

    );

    Control C(
        .Opcode(Opcode),
        .I(I),
        .IsSt(IsSt),
        .IsLd(IsLd),
        .IsBeq(IsBeq),
        .IsBgt(IsBgt),
        .IsRet(IsRet),
        .IsImmediate(IsImmediate),
        .IsWb(IsWb),
        .IsUBranch(IsUBranch),
        .IsCall(IsCall),
        .AluSignal(AluSignal)
    );

    Register_File RF(
        .Clk(Clk),
        .d1(d1),
        .d2(d2),
        .a1(a1),
        .a2(a2),
        .a3(a3),
        .d3(d3),
        .Reset(Reset),
        .IsWb(IsWb)
    );

    Memory M(
        .a_m1(a_m1),
        .a_m2(a_m2),
        .d_m1(d_m1),
        .d_m2(d_m2),
        .d_m3(d_m3),
        .rw(rw)
    );

    ALU ALU(
        .A(A),
```

```
        .B(B),
        .Alu_Result(Alu_Result),
        .AluSignal(AluSignal),
        .Flags(Flags)
    );

endmodule
```

6.1.15 Testbench

```
`timescale 1ns / 1ps
`define td 1000

module Test_Pipeline();

    integer x1, x2, x3;
    integer File, Filex, temp, c = 0;
    reg [64:0] Buff[32'h1000:0];
    reg Clk;

    Pipeline PL (.Clk(Clk));

    initial
    begin
        Clk = 0;
        x2 = 1;
        x3 = 0;
        PL.Reset = 1;

        Filex = $fopen("Memory.mem", "r");
        temp = $fgetc(Filex);
        while (!$feof(Filex))
        begin
            if (temp == "\n")
                c = c + 1;
            temp = $fgetc(Filex);
        end

        $readmemh("Memory.mem", Buff, 0, c * 2 - 1);

        for (x1 = 0; x1 < 32'h1000; x1 = x1 + 1)
        begin
            if (x3 == 4)
            begin
                x3 = 0;
                x2 = x2 + 2;
```

```
            end
            PL.M.Memory[x1] = Buff[x2][8 * x3 +: 8];
            x3 = x3 + 1;
        end

        #1 PL.Reset = 0;

        #`td
        $finish;
    end

    initial
    begin
        #`td
        File = $fopen("C:\\Users\\pants\\simplerisc-project\\
            Output_File.txt", "w");
        $fdisplay(File, "\tUpdated Registers\t\n");

        for (x1 = 0; x1 < 16; x1 = x1 + 1)
        begin
            $fdisplay(File, "\tR=[%d] ==%h\n", x1, PL.RF.Regs[x1
                ]);
        end

        $fdisplay(File, "\n\tUpdated Memory\t\n");

        for (x1 = 0; x1 < 32'h1000; x1 = x1 + 1)
        begin
            $fdisplay(File, "\tA=%h D=%h \n", x1, PL.M.Memory[x1
                ]);
        end

        $fclose(File);
    end

    always
    begin
        #5 Clk = ~Clk;
    end

endmodule
```