Faculty of Engineering and Technology
Department of Electrical and Computer Engineering

ENCS4370

Computer Architecture

Second Project

Processor Design : A Multi-Cycle Implementation

Sarah Hassouneh – 1210068

Sondos Qasarwa – 1210259

Instructor: **Dr. Aziz Qaroush**

Section (3,1)

Date: **20 June 2024**

Abstract

In our project, we aim to design and verify a simple pipelined RISC processor using Verilog. This project demonstrates how the concepts of computer architecture and organization are effectively used to make design decisions about the microprocessor components. The design of the microprocessor starts by determing the instruction set architecture and analyzing it. This leads to better understand the environment and the components needed to assemble a correct datapath. The datapath implementation can also vary and each need special handling. In this project we used a multi cycle implementation and a state machine to execute the instruction at different stages and states. This report demonstrated the design of the microprocessor from start to finish, It shows the steps taken and the verification of each instruction.

Table of Contents

Abstract .... *I*

1 Design and Implementation *1*

1.1 Processor Specifications and Overview **1**

1.2 Instruction Format & RTL operations **3**

1.2.1 Instruction Formats and Interpretations 3

1.2.1 Instruction RTL Micro operations 6

1.3 Functional Units and components **10**

1.4 Constructing the Datapath **19**

1.5 Designing the Control **22**

1.5.1 Designing the main control 24

1.5.2 Designing the PC control 28

1.5.3 Designing the ALU control: 31

2 Simulation and Testing *34*

2.1 Testing Each Instruction Type **34**

2.1.1 R-type Verification 34

2.1.2 I-type (ALU) Verification 35

2.1.3 LW Verification 36

2.1.4 LBu & LBs Verification 38

2.1.5 SW Verification 40

2.1.6 SV Verification 41

2.1.7 Branch Verification 42

2.1.8 Jump Verification 44

2.1.9 Call Verification 45

2.1.1 Return Verification 45

3 Teamwork *46*

Conclusion *47*

Table of Figures

Figure 1: Processor Specifications 1
Figure 2: instruction memory 11
Figure 3- Data memory 12
Figure 4: Register File 13
Figure 5:ALU 13
Figure 6-Decode and Multiplexing unit 16
Figure 7: Full Datapath 19
Figure 8: Full Datapath with Control Signals 21
Figure 9: State Transition Diagram of Multi- Cycle Execution 22
Figure 10- R_type test simulation result 35
Figure 11- I_type ALU test simulation result 36
Figure 12: Load instruction test waveform 38
Figure 13: LBu and LBs instruction test waveform 39
Figure 14: Store instruction test waveform 40
Figure 15: SV instruction test waveform 41
Figure 16_ Branch instruction test waveform 42
Figure 17- Branch instruction test 2 waveform 43
Figure 18- Test Branch instruction waveform (Test2) 44
Figure 19- J-type instructions waveform 45

Table of Tables

Table 1: Project Instruction Details' 6
Table 2:Instruction Decoding – Instructions that use ALU 14
Table 3: Instruction Decoding – Instructions without ALU........ 15
Table 4: Decode and Multiplexing unit Truth Table 16
Table 5: State Assignment........ 23
Table 6: Main control signals effect........ 24
Table 7- WB_Sel control signal effect. 24
Table 8- Main control truth table part1 26
Table 9- Main control truth table part 2 27
Table 10 - PC control truth table........ 29
Table 11: Opcodes and ALU opcodes for each instruction........ 32
Table 12- Registers initial values 34
Table 13- Tested R_type instructions........ 34
Table 14- Registers initial values 36
Table 15- Tested R_type instructions........ 36
Table 16- Registers initial values for Load 37
Table 17- Tested Load Instruction Memory initialization........ 37
Table 18- Tested Load Data Memory initialization 37
Table 19- Registers initial values for LBu, LBs........ 39
Table 20- Tested Load Byte Instruction Memory initialization 39
Table 21- Tested Load Byte Data Memory initialization........ 39
Table 22: Tested Store Instruction Memory initialization........ 40
Table 23: Tested S-type Instruction Memory initialization........ 41
Table 24- Registers initial values 42
Table 25- Tested branch instructions........ 42
Table 26- Tested Branch instructions (Test 2)........ 44
Table 27: Tested J-Type Instructions........ 45

1 Design and Implementation

In our project, we aim to design and verify a simple pipelined RISC processor using Verilog. The design process begins with studying the Instruction Set Architecture (ISA), understanding the types of instructions used, and specifying the processor's requirements. This understanding helps in determining the necessary functional and control units and assembling the correct data path. This report explains the steps taken to build the processor, details the components used, and describes the execution of the instructions.

1.1 Processor Specifications and Overview

As a start , the following specifications were given for the processor design:

- The instruction size and the word size is 16 bits (2 bytes)
- 8 (16-bit) general-purpose registers: from R0 to R7
- R0 is hardwired to zero. Any attempt to write it will be discarded.
- 16-bit special purpose register for the program counter (PC)
- Four instruction types (R-type, I-type, J-type, and S-type)
- Separate data and instruction memories
- Byte addressable memory
- Little endian byte ordering
- The required signals are generated from the ALU to calculate the condition branch outcome (taken/ not taken). These signals might include zero, carry, overflow, etc.

Figure 1: Processor Specifications

These specifications highly influenced the design decisions that have been taken, as will be explained in the following sections. We have decided to implement a **Multi-Cycle Processor**. In a multi-cycle implementation each instruction is broken into five main steps, namely :

1. Instruction fetch
2. Instruction decode
3. Execution
4. Memory access

5. Write Back

Each of these steps take one clock cycle , the clock cycle here will be roughly (1/5) of a cycle in a single cycle implementation. In a multi-cycle implementation each instruction takes a number of clock cycles as needed, this could vary from 2-5 cycles. This decreases the execution time and thus increase performance. The instruction fetch and decode are necessary in each of the instructions. Steps 2,3,4,5 can have different interpretations based on the instruction execution, as will be shown in the following sections.

1.2 Instruction Format & RTL operations

1.2.1 Instruction Formats and Interpretations

In our project we want to implement 21 different instructions as stated in (Table 1), the table states the opcode value and the description of each. These instructions follow 4 instructions format as follows :

1. **R-type (Register Type)**

$Opcode^4$	Rd^3	$Rs1^3$	$Rs2^3$	$Unused^3$

This format is used for register operations and is encoded as follows:

- 3-bit Rd : destination register
- 3-bit Rs1: first source register
- 3-bit Rs2: second source register
- 3-bit unused

2. **I-type (Immediate Type)**

$Opcode^4$	m^1	Rd^3	$Rs1^3$	$Immediate^5$

This format is used for operations with immediate values and for store and load operation and is encoded as follows:

- 3-bit Rd: destination register
- 3-bit Rs1: first source register
- 5-bit immediate: unsigned for logic instructions, and signed otherwise.
- 1-bit mode: this is used with load and branch instructions, such that:

For the load:

0: LBs load byte with zero extension

1: LBu load byte with sign extension

For the branch:

0: compare Rd with Rs1

1: compare Rd with R0

3. **J-type (Jump Type)**

This format is used for Jump, call and return operations. In the jump and call instructions the 12-bit immediate is concatenated with the most significant 4-bit of the current PC to produce the Jump target address, as shown :

$Opcode^{4}$	Jump $Offset^{12}$

In the return instruction however the 12-bit are unused bits.

4. **S-type (Store Type)**

$Opcode^{4}$	Rs^{3}	$Immediate^{9}$

This format is used to perform a special kind of store, where M[Rs] = Immediate, it is interpreted as:

- 3-bit Rs : source register
- 12-bit immediate: value to be stored

Group	No.	Instr	Format	Meaning	Opcode No.	Opcode Value	m
1	1	AND	R-Type	Reg(Rd) = Reg(Rs1) & Reg(Rs2)	0	0000	
	2	ADD	R-Type	Reg(Rd) = Reg(Rs1) + Reg(Rs2)	1	0001	
	3	SUB	R-Type	Reg(Rd) = Reg(Rs1) - Reg(Rs2)	2	0010	
2	4	ADDI	I-Type	Reg(Rd) = Reg(Rs1) + Imm	3	0011	
	5	ANDI	I-Type	Reg(Rd) = Reg(Rs1) + Imm	4	0100	
3	6	LW	I-Type	Reg(Rd) = Mem(Reg(Rs1) + Imm)	5	0101	
4	7	LBu	I-Type	Reg(Rd) = Mem(Reg(Rs1) + Imm)	6	0110	0
	8	LBs	I-Type	Reg(Rd) = Mem(Reg(Rs1) + Imm)	6	0110	1
5	9	SW	I-Type	Mem(Reg(Rs1) + Imm) = Reg(Rd)	7	0111	
6	10	BGT	I-Type	if (Reg(Rd) > Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	8	1000	0
	11	BGTZ	I-Type	if (Reg(Rd) > Reg(0)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	8	1000	1
	12	BLT	I-Type	if (Reg(Rd) < Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	9	1001	0
	13	BLTZ	I-Type	if (Reg(Rd) < Reg(R0)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	9	1001	1
	14	BEQ	I-Type	if (Reg(Rd) == Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	10	1010	0
	15	BEQZ	I-Type	if (Reg(Rd) == Reg(R0)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	10	1010	1
	16	BNE	I-Type	if (Reg(Rd) != Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	11	1011	0

	17	BNEZ	I-Type	if (Reg(Rd) != Reg0)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	11	1011	1
7	18	JMP	J-Type	Next PC = {PC[15:12], Imm}	12	1100	
8	19	CALL	J-Type	Next PC = {PC[15:12], Imm} PC + 2 is saved on r7	13	1101	
9	20	RET	J-Type	Next PC = r7	14	1110	
10	21	Sv	S-Type	M[Rs] = Imm	15	1111	

Table 1: Project Instruction Details'

1.2.1 Instruction RTL Micro operations

Now after we understood the meaning of each instruction, we need to write the microoperations for each. The micro-operations are written in Register transfer Level (RTL), which shows the values of the registers, and the data flow at each stage. Using a multi-cycle implementation, each instruction goes through 5 main steps:

$$IF \rightarrow ID \rightarrow EX \rightarrow M \rightarrow WB$$

The following RTL operations are written to explain what happens at each step. As specified in the specifications the word size is 2 bytes, memory is byte addressable, and PC is 16-bit. The instructions were divided in groups as denoted in (Table 1) where each group share similar micro-operations. At the start of each group, the required stages are denoted and then microoperations are stated in order. (if there is no number next to it, this means that these micro-operations can execute simultaneously at the same clock cycle with the previous micro-operation). The micro-operations are as follows:

Group 1 : R-type : $IF \to ID \to EX \to WB$

1. Fetch Instruction: $IR \leftarrow Mem[PC]$
2. Fetch Operands: $data1 \leftarrow Reg(Rs1),\ data2 \leftarrow Reg(Rs2)$
3. Execute : $ALU_{result} \leftarrow ALU_{opcode}(data1, data2)$
4. Write ALU: $Reg(Rd) \leftarrow ALU_{result}$

   Next PC: $PC \leftarrow PC + 2$

Group 2 : I-type (Arithmetic) : $IF \to ID \to EX \to WB$

1. Fetch Instruction: $IR \leftarrow Mem[PC]$
2. Fetch Operands: $data1 \leftarrow Reg(Rs1),\ data2 \leftarrow * \ Ext(imm_5)$
3. Execute : $ALU_{result} \leftarrow ALU_{opcode}(data1, data2)$
4. Write ALU: $Reg(Rd) \leftarrow ALU_{result}$

   Next PC: $PC \leftarrow PC + 2$

$*\ Sign\ Ext\ at\ ADDI\ , Unsigned\ Ext\ at\ ANDI$

Group 3 : I-type (Load Word): $IF \to ID \to EX \to M \to WB$

1. Fetch Instruction: $IR \leftarrow Mem[PC]$
2. Fetch Base Register: $base \leftarrow Reg(Rs1)$
3. Calculate address : $addr \leftarrow base + Sign\ Ext(imm_5)$
4. Read (Memory): $data \leftarrow Mem[addr]$
5. Write (Register) $Reg(Rd) \leftarrow data$

   Next PC: $PC \leftarrow PC + 2$

Group 4 : I-type (Load Byte): $IF \to ID \to EX \to M \to WB$

1. Fetch Instruction: $IR \leftarrow Mem[PC]$
2. Fetch Base Register: $base \leftarrow Reg(Rs1)$

3. Calculate address : $addr \leftarrow base + Sign\ Ext(imm_5)$
4. Read (Memory): $dataByte \leftarrow Mem[addr]$

   Extension: $data \leftarrow * Ext[dataByte]$
5. Write (Register) $Reg(Rd) \leftarrow data$

   Next PC: $PC \leftarrow PC + 2$

$*\ Ext : Signed\ or\ unsigned\ based\ on\ m - bit$

<u>Group 5 : I-type (Store):</u> $IF \rightarrow ID \rightarrow EX \rightarrow M$

1. Fetch Instruction: $IR \leftarrow Mem[PC]$
2. Fetch Registers: $base \leftarrow Reg(Rs1), data \leftarrow Reg(Rd),$
3. Calculate address : $addr \leftarrow base + Sign\ Ext(imm_5)$
4. Write (Memory): $Mem[addr] \leftarrow data$

   Next PC: $PC \leftarrow PC + 2$

<u>Group 6 : I-type (Branch):</u> $IF \rightarrow ID \rightarrow EX$

1. Fetch Instruction: $IR \leftarrow Mem[PC]$
2. Fetch Registers: $data1 \leftarrow Reg(Rd),\ data2 \leftarrow Reg(* Rs1/R0)$
3. Execute : $flags \leftarrow SUB(data1, data2)$

   Branch :

$$
\begin{array}{l}
if\ (flags == true) \\
\quad PC \leftarrow (PC + 2) + Sign\ Ext(imm_5) \\
else \\
\quad PC \leftarrow PC + 2
\end{array}
$$

$*\ Rs1\ or\ R0\ based\ on\ m - bit$

$*\ flags\ to\ check\ include\ zero\ and\ carry$

Group 7 : J-type (Jump): $IF \rightarrow ID$

1. Fetch Instruction: $IR \leftarrow Mem[PC]$
2. Target PC address: $target \leftarrow PC[15:12] \,||\, offset$

   Jump: $PC \leftarrow target$

Group 8 : J-type (Call): $IF \rightarrow ID \rightarrow WB$

1. Fetch Instruction: $IR \leftarrow Mem[PC]$
2. Target PC address: $target \leftarrow PC[15:12] \,||\, offset$
3. Save address: $R7 \leftarrow PC + 2$

   Jump: $PC \leftarrow target$

Group 9 : J-type (Return): $IF \rightarrow ID$

1. Fetch Instruction: $IR \leftarrow Mem[PC]$
2. Target PC address: $PC \leftarrow Reg(R7)$

Group 10 : S-type : $IF \rightarrow ID \rightarrow WB$

1. Fetch Instruction: $IR \leftarrow Mem[PC]$
2. Fetch Base Register: $base \leftarrow Reg(Rs)$

   Fetch Operands: $data \leftarrow Ext(imm_9)$
3. Write (Memory): $Mem[base] \leftarrow data$

   Next PC: $PC \leftarrow PC + 2$

These micro-operations are necessary for determining the needed structural units and control signals as will be shown in the next part.

1.3 Functional Units and components

After understanding the instruction formats, micro-operations and the instruction life cycle of each instruction, we can determine the main structural components to be used in the data path. These components has been added to prevent structural hazards where it could happen during the execution. The components are as follows:

1. **PC**

   The PC is a special purpose register necessary for program execution. It is used as a storage element to save the address of the current instruction to be executed. Based on the processors specification the PC is 16-bit. This means that it can access up to 2^{16} cells. Each cell is a byte because we have a byte addressable memory and so it can access up to 2^{16} Bytes ($2^6 . 2^{10} =$ 64 kiloBytes), or 2^{15} *words / instructions*.
   To ensure correct execution in our data path the PC is clocked, moreover, it has an enable signal(PCwrite) to enable writing at certain times.

2. **IR**

   The IR is another special purpose register that stores the current instruction. Since the instruction size is 16- bit, the IR register must also be 16-bit. The IR register is necessary to keep track of the current instruction and has a crucial rule in the decode stage where many operands and fields are brought directly from this register. Again the IR is clocked and has an enable signal (IRwrite), to enable writing at certain points.

3. **Instruction Memory:**

   Both the PC and IR work on brining the instruction from the instruction memory, saving it and then executing it. According to the processor specifications, the data memory and the instruction memory are separated. The instruction memory stores the instructions of the program. Instruction memory only provides read access because data path does not write instructions. It contains a 16-bit address as an input and one output which is a 16-bit instruction. Each instruction is stored in 2 cells in the memory because the memory is byte addressable, and the instruction length is 16. A little-endian ordering is used in this memory. To fetch the instruction form the corresponding input address and using little endian, the following statement is used:

instruction = {memory [addr + 1], memory[addr]}

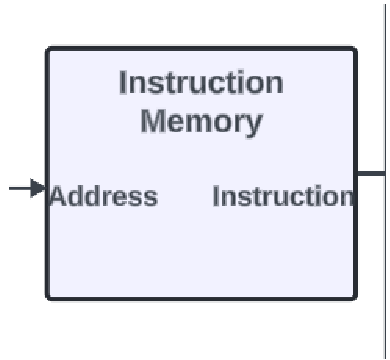

Figure 2: instruction memory

4. **Data Memory:**

We noticed that in the Memory Access stage reading data from the memory is a must. The data memory used here is used for storing data. It used with load and store instructions. This memory is byte addressable, and it uses little endian ordering. The data memory provides read access for load instructions and write access for store instructions. It has 4 inputs as shown in (Figure 3): 16-bit address, 16-bit data in, MemRd signal and MemWrite signal with one output: 16-bit data out. In Load instruction, MemRd is enabled, and the data selected by the input address is put on the data_out. In store instructions, MemWrite is enabled and the data-in is written on the memory at the address selected by the input address. Data memory also takes a clock signal to synchronize the write operation.

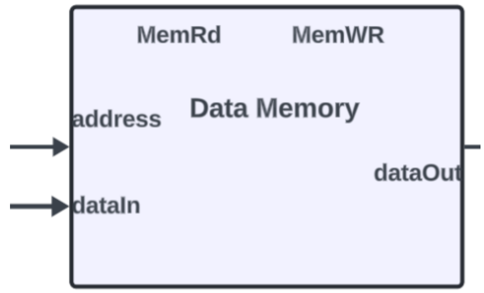

Figure 3- Data memory

5. Register file

In the micro-operations, in addition to memory and special purpose registers, the instruction made use of general-purpose registers. As specified in the Processor Specifications part. We have 8 registers with 16-bit width each from R0 to R7. Additionally this register file has been designed to have R0 hardwired to zero and any attempt to change it will be discarded. (Figure 4) shows the register file used in our design, in addition to the ports shown it also has an input clock to synchronize the data path. It has 2 reading ports and one writing port. It has 3 inputs for addresses (addr_read1, addr_read2, addr_write), each of these inputs is 3-bit width because we have 8 registers. The corresponding outputs (bus_read1, bus_read2) are the output data that corresponds to register contents at the specified addresses, it should be noted that reading from a register file is a combinational logic and needn't to be clocked. As for the writing, the 16-bit bus (bus_write) determines the data to be written at (addr_write) register, and we have an input (RegWr) which is a control signal to enable or disable writing. This writing operation is clocked and should be synchronized, such that writing only occurs at the edge of the clock if (RegWr) is enabled.

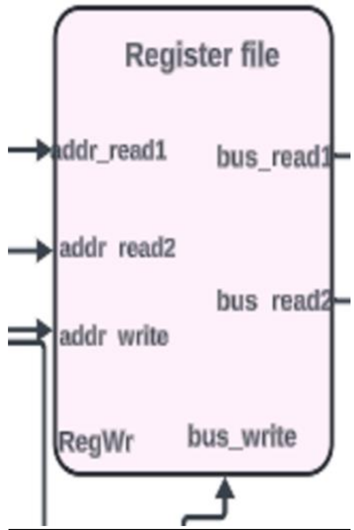

Figure 4: Register File

6. ALU

As we noticed in the micro-operation and in the description of the instruction three main logical and arithmetic operations must be performed (logical and, addition and subtraction) between two operand. The ALU has been simply designed to support these operations, in addition to (no_op) operation at stages where the ALU output is not needed, or we want to add a delay. The ALU as shown in (**Error! Reference source not found.**) has two inputs for two o perands namely A, B and output (output result) , all of these are 16-bit data. The Alu has additionally an (ALU_opcode) input that specifies the operations to be performed. It also has 4 output flags (zero, carry, negative and overflow).

Figure 5:ALU

7. <u>Decode and Multiplexing Unit :</u>

As we noticed in the instruction formats and decoding part, the interpretation of different bits is different depending on the instructions and its meaning. Additionally, for example in the I-type and R-type , the destination register Rd occupies bits from [11:9] in R-type but occupies bits from [10:8] in I-type. This leads to having many cases for the instruction format decoding. To simplify this we created Decode and Multiplexing Unit, that helps in decoding the instruction directly, this saves us time, muxes and complexity.

This unit is used in decode stage. In this stage the opcode is determined then operands are fetched, and the destination register also is determined. The table below illustrates how the opcode is used to determine the destination address and the source operand addresses for instructions that use the ALU:

Instruction	destination register (addr_write in reg file)	operand 1 (in Alu) (addr_read1 in reg file)	operand 2 (in Alu) (addr_read2 in reg file)	Comments
R - type	Rd = IR [11:9]	Rs1 = IR [8:6]	Rs2 = IR [5:3]	
ALU_I	Rd = IR [10:8]	Rs1 = IR [7:5]	Imm [4:0]	
LW / LBu / LBs	Rd = IR [10:8]	Rs1 = IR [7:5]	Imm [4:0]	
SW	-	Rs1 = IR [7:5]	Imm [4:0]	Data in : Rd = IR [10:8]
BGT, BLT, BEQ, BNE	-	Rd = IR [10:8]	Rs1 = IR [7:5]	Imm = IR[4:0] for calculating branch target address
BGTZ, BLTZ, BEQZ, BNEZ	-	Rd = IR [10:8]	R0	Imm = IR[4:0] for calculating branch target address

Table 2:Instruction Decoding – Instructions that use ALU

Other instructions that do not use ALU, the data is extracted at the decode stage as shown in the table:

Instruction	Data
J	Offset = IR [11:0] for calculating jump address
Call	(addr_write = Dst reg) = R7 Imm = IR [11:0] for calculating function address
Ret	Read Reg = R7 from Reg file Addr_read1 = 111
Sv	Data in = imm = IR [8:0] Mem_address = Rs So addr_read1= IR[11:9]

Table 3: Instruction Decoding – Instructions without ALU

As shown in the previous 2 tables there is different options for addr_read1, addr_read2 and addr_write in the register file, instead of adding 3 multiplexers we add a new component to decode instruction and it is called decode _and_multiplexing_unit. It takes the instruction as input, and produces the opcode, m bit, address_read1 , address_read2, address_write for the register file, where :

Address_read1: Address of the first register to read from the register file.

Address_read2: Address of the second register to read form the register file.

Address_write: Address of the destination register which is to be written on the register file.

The unit also produces the I_type_imm, J_type_imm and S_type_imm. Because the immediate size varies depending on the type of instruction, it is divided into three types:

I_type_imm: immediate in I type instructions (5 bits)

J_type_imm: immediate used in Jump and Call instructions. (12 bits)

S_type_imm: immediate used in S_type instructions. (9 bits)

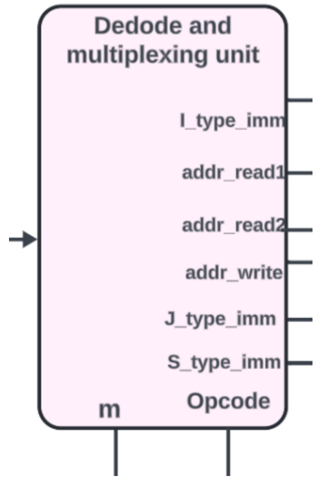

Figure 6-Decode and Multiplexing unit

The following is the decode and multiplexing unit component's truth table:

Opcode Instruction[15:12]	Address_write	Address_read1	Address_read2	I_type imm	S_type_imm	J_type_imm
R-type	IR [11:9]	IR [8:6]	IR [5:3]	X	X	X
Alu-I	IR [10:8]	IR [7:5]	X	IR [4:0]	X	X
LW / LBu / LBs	IR [10:8]	IR [7:5]	X	IR [4:0]	X	X
SW	X	IR [7:5]	IR [10:8]	IR [4:0]	X	X
BGT, BLT, BEQ, BNE	X	IR [10:8]	IR [7:5]	IR[4:0]	X	X
BGTZ, BLTZ, BEQZ, BNEZ	X	IR [10:8]	R0 = 000	IR[4:0]	X	X
J	X	X	X	X	X	IR[11:0]
Call	R7 = 111	X	X	X	X	IR[11:0]
Ret	-	R7	-	-	-	-
Sv	X	IR [11:9]	X	X	IR[8:0]	X

Table 4: Decode and Multiplexing unit Truth Table

8. Extenders

Following the instruction format and the explanation of the decode and multiplexing unit the immediate in the instructions has different sizes. For that extenders are needed to extend each immediate from (5,9) to 16 bits. The extension can be signed or unsigned and that can be determined by a control signal.
Additionally, when loading a byte with (LBu) or (LBs) instruction, an extender is needed to extend the byte whether that is signed or unsigned before saving it int a register.

9. Muxes

Following our understanding of the instructions micro-operations, we can notice that some components have different values written based on the instruction. The selection of these components is determined by a mux, where the selection lines are control signals of the control units. We have solved part of this in the decode and multiplexing unit, but some others needs additional muxes. These can be summarized as :

- The next instruction to be written at the PC can have 4 options: Pc + 2, Branch target address, jump/call target address or R7 in Return instruction.
- The second operand of the ALU can have two options: the extended I immediate or (bus_read_2). (bus_read_2) depends on addr_read2 that is also chosen by the decode and mux unit.
- The data memory address input can have two options: bus_Read1 (Rs in S_type) or the Alu result (in Load and Store).
- The data memory data_in input can have two options: (bus_read_2) or the extended S immediate.
- The Write back data to the register file (bus_write input) can have 4 options: Alu result, Byte extender output (in LBu and LBs) , Memory data out or PC+2 (in call instruction)

The configuration of each mux will be shown in the data path construction part.

10. Buffers between stages

Because we are developing a multi cycle implementation, each of the stages is done separately, however because data is needed to pursue the next clock cycle storage elements must be used and synchronized to the clock. In addition to using IR and PC, we used 4 additional buffers to save the outcomes:

- Register A saves the result of (bus_read1) from the decode stage to be used as the ALU first operand in the execution stage.
- Register B saves the result of (bus_read2) from the decode stage to be used as the ALU second operand in the execution stage.
- Register Immediate saves the result of (extended I immediate) from the decode stage to be used as the ALU second operand in the execution stage.
- Register ALU_result saves the result of the ALU from the execution stage to be used in the Memory or Write Back stage.

1.4 Constructing the Datapath

Following our understanding of each of the components and how they work we constructed the data path as follows (Figure 7):

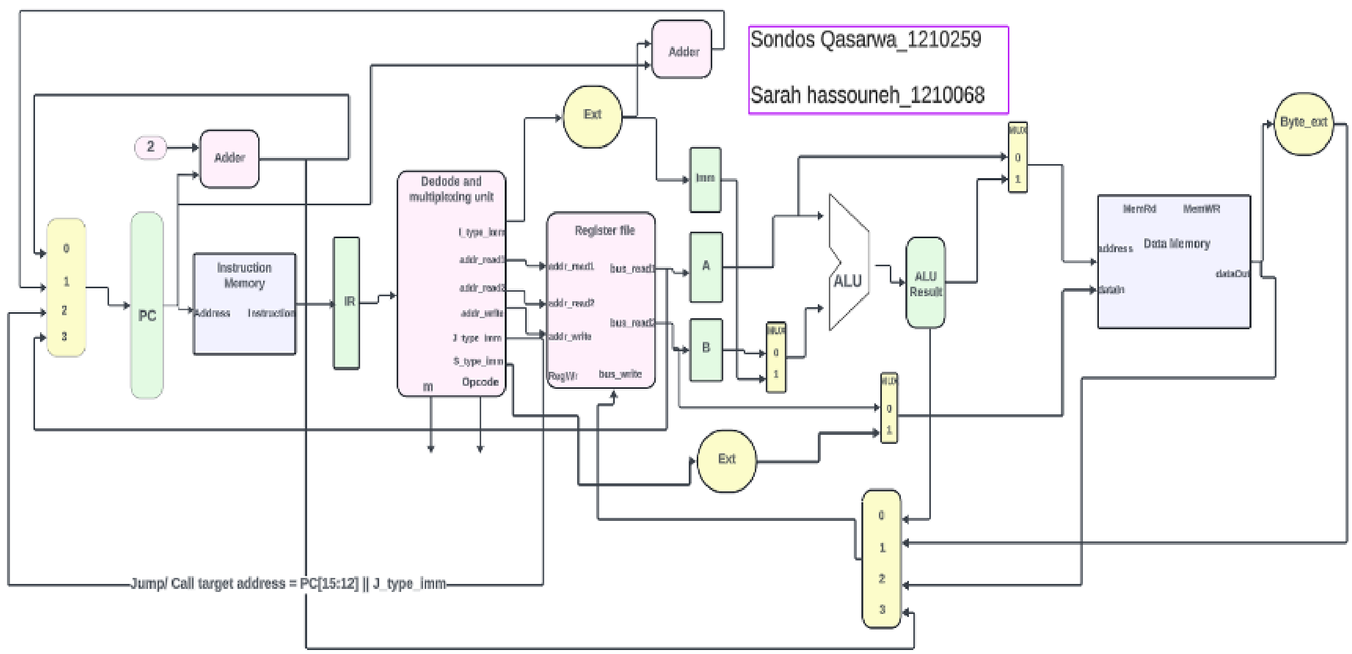

Figure 7: Full Datapath

We start at the PC in the instruction fetch where the address of the next instruction is saved. The PC is connected with (+2) adder to find next PC and another adder to find branch target address when branching. In the decode stage the corresponding instruction goes into IR and then directly to the decode unit (combinational logic) to extract the needed fields. Following that the instruction goes to Execute, Memory or Write Back stage based on its opcode.

In the Execute stage the needed operands are ready by the end of the decode stage. The ALU computes the result and sets the flags. In the Memory Access the address is ready either directly form instruction(as is SV) or computed from ALU. The data_in is also ready as bus_read2 or S_type immediate and then memory data_out is ready. In the WB stage the bus_write data is being selected by the Mux after the data is ready to be written.

The mux configurations are as follows:

PC Multiplexer:

0 :Next PC = PC + 2

1: Next PC = Branch target address

2: Next PC = Jump/Call target address

3: Next PC = bus_read1 (R7 register content in Ret instruction)

ALU Second Operand Multiplexer:

0: Operand2 = bus_read_2

1 :Operand2 = Ext (I_type_imm)

Memory address multiplexer:

0: address = bus_Read1 (Rs in S_type)

1: address = Alu result (in load and Store)

Memory data_in multiplexer:

0 :data in = bus_Read2 (Rd in SW)

1: data_in = Ext(S_type_imm) in S_type instruction

Write Back multiplexer:

0: bus_write = Alu result

1: bus_write = Byte Extender output (in LBu, LBs)

2: bus_write = Memory data out

3: bus_write = Next PC (PC+2) (in call instruction)

Each of these mux have selection lines that are determined by the control units. So in addition to the structural components, three control units must be added to the data path : PC control, Main Control and ALU control. The selection lines and signals are connected as shown in (Figure 8). The execution and data flow at each stage is determined by the values of these signals.
The PC control has two output signals : PCwrite connected with the PC and acts as an enable for writing on the PC, and PCsrc as the mux selection and. The main control is responsible for

determining the current state of execution and has many outputs signal. The (IRwrite) is an enable signal, that is enabled only when fetching a new instruction to be decoded. (ALU src) is the selection line for the ALU Second Operand Multiplexer, similarly (Mem_addr_sel) is the selection line for the address Multiplexer, (Data_in_sel) is the selection line for the data_in Multiplexer and the (WBsel) is the selection line for the Write Back multiplexer. The Ext signal is connected to the extender to determine signed or unsigned extension, likewise the (Byte_ext_signal) is connected with Byte_ext to determine type of extension. The (MemRd and MemWr) are used to control the data memory and enable read and write operations.

The ALU control has one main output (ALU_opcode), this is a main input in the ALU unit, and it determines the operation done in the ALU at any stage.

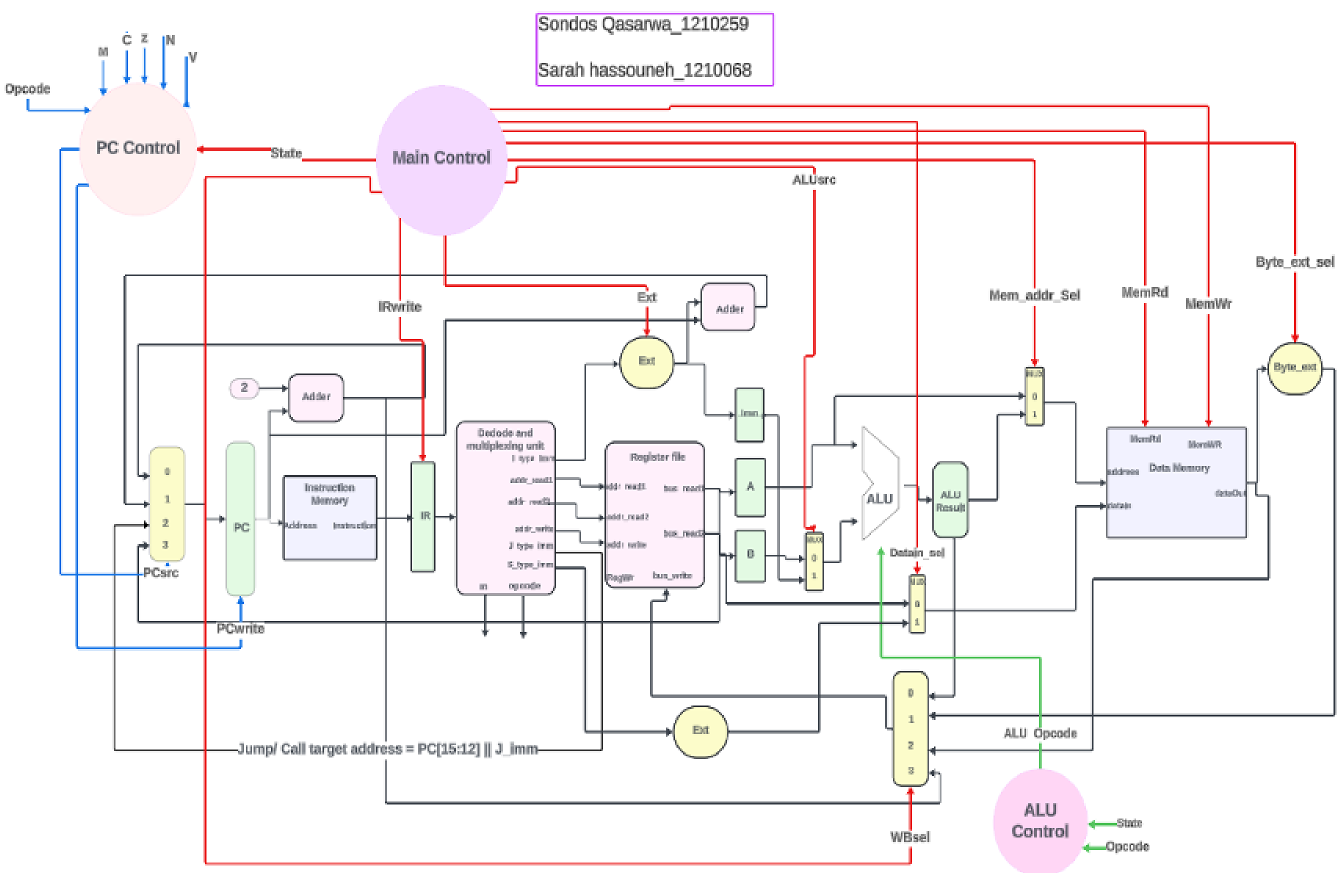

Figure 8: Full Datapath with Control Signals

The derivation of each control signal and states will be explained in the next section.

1.5 Designing the Control

In a multi-cycle implementation each instruction takes a number of clock cycles based on the necessary stages it needs to go through. This means that the next stage for each instruction will be different, depending on both the current state and the opcode. This is why we design the multi cycle control unit as a finite state machine (Mealy state machine). A state is uniquely defined by the control signals produced by the control units at that specific time, these control signals define what is happening at the state.

The state diagram for the state machine we designed is shown in (Figure 9), each state has the control units that are set during that stage. If some control units are not shown this means that they are set by default to 0 and ALU_opcode default is no_op. The state assignment for each of the states is shown in (Table 5).

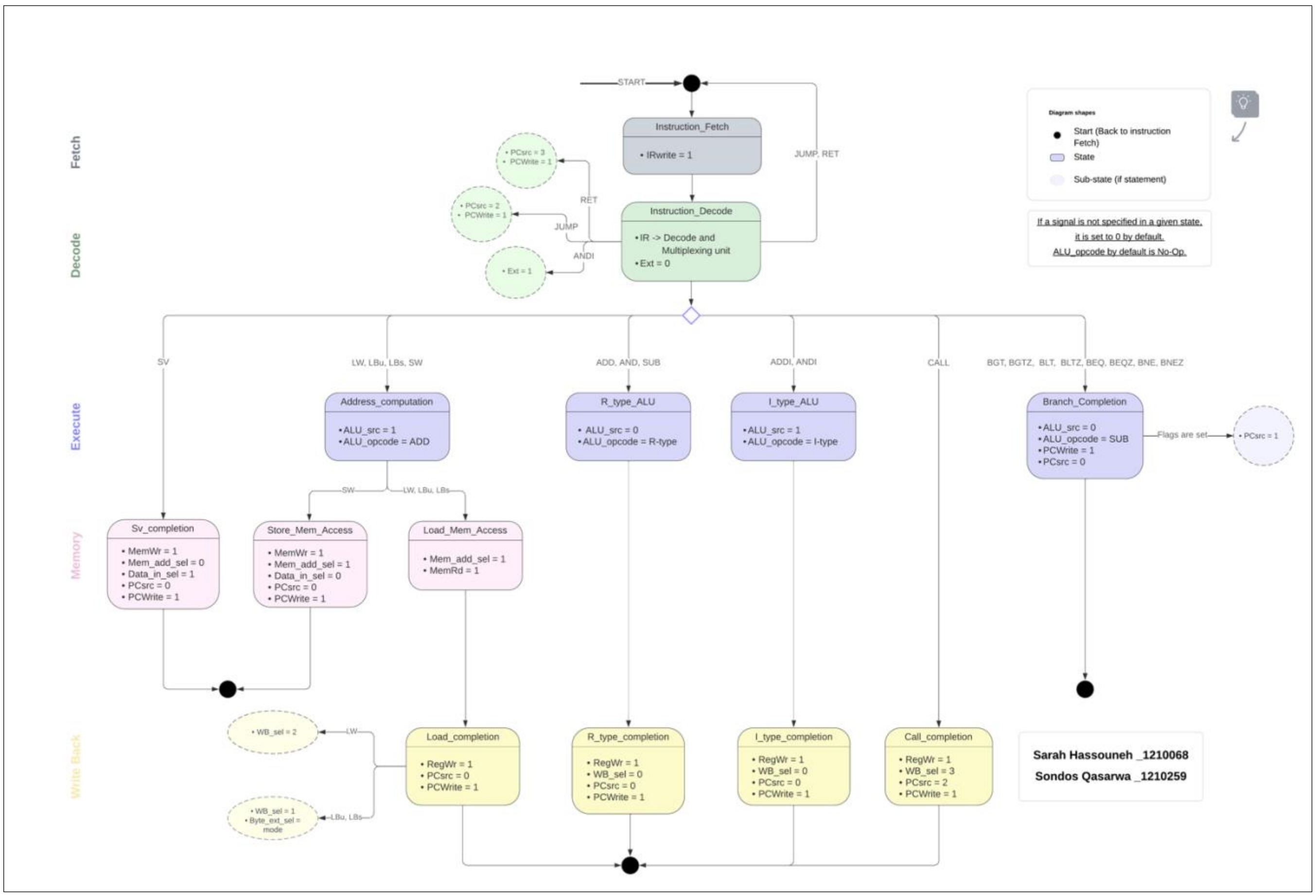

Figure 9: State Transition Diagram of Multi- Cycle Execution

To better understand the diagram, if we take (ADDI) instruction for example, it starts at the fetch state, then it goes to the decode stage. These two states are shared among all instructions. After the decode the (ADDI) goes to the I_type_ALU state in this state (ALUsrc) signal is set to 1 and ALU_opcode to the I_type operation, which in this case is ADD. After this execute stage, the instruction goes to the I_type_completion state (WriteBack stage), where the value of ALU result is written on a register and the PC is updated to the next PC. This operation went into 4 states or stages this means it needs 4 clock cycles to execute. On the contrary, Sv takes 3 clock cycles as it foes from fetch to decode to Sv_Completion state, which is a memory access stage and it skips the execute stage.

To implement that in our data path and code it easily and modularly, we divided the control signals to PC, ALU and Main Control. The main control was designed as a state machine and given the task of determining the next state. This state is an input to the ALU control and PC control units which outputs the necessary signals based on that state. This input state can be shown in Figure 8.
The following sections breaks down these signals and their derivation.

State	State number	Abbreviation
Instruction_fetch	0	IF
Instruction_decode	1	ID
R_type_ALU	2	R_ALU
R_type_completion	3	RC
I_type_ALU	4	I_ALU
I_type_completion	5	IC
Address_computation	6	AC
Load_Mem_Access	7	L_Mem
Store_Mem_Access	8	S_Mem
Load_completion	9	LC
Branch_Completion	10	BC
Call_completion	11	CC
Sv_completion	12	SC

Table 5: State Assignment

1.5.1 Designing the main control

In the main control unit, we start by studying the output control signals and their effect of the datapath. The table below shows the one-bit signals and the effect of each signal when its value is 0 or 1:

Signal	Effect when 0	Effect when 1
IRwrite	IR value does not change	The output of instruction memory is written on IR
Ext	I_type_imm is signed extended.	I_type_imm is unsigned extended.
Data_in_Sel	Data_in = BusRead2 form register file (Rd is SW instruction)	Data_in = S_type_imm in S_type instruction.
ALU_src	Alu src = Reg B	Alu src = Immediate
Mem_add_sel	Data memory addr = Reg A In Sv instruction	Data memory addr = ALU-result (in SW instruction)
MemRd	No effect	Data is put on data_out
MemWr	No effect	Data in is written into memory
RegWr	No effect	Write on register file
Byte_ext_sel	Least 8 bit of data out are signed extended (in LBs)	Least 8 bit of data out are unsigned extended (in LBu)

Table 6: Main control signals effect

WB_sel is another control signal produced by the main control, but it is 2-bit. Its effect is as follows:

Signal	Effect when 0	Effect when 1	Effect when 2	Effect when 3
WB_sel	Write back data is ALU result (in R-type and I_type ALU)	Write back data is byte_ext result (in LBu & LBs)	Write back data is memory data_out (in LW)	Write back data is PC + 2 (in CALL instruction)

Table 7- WB_Sel control signal effect.

Main control unit is a finite state machine. It takes a clock signal and the opcode and the mode bit as inputs and it determines the value of 10 main control signals and the next state based on the inputs and the current state.

The following table shows the main control truth table:

Note: this table shows 5 output signals. And the next table will show the truth table for the remaining outputs.

Inputs			Outputs					
Current state	**Opcode**	**m**	**Next State**	**IRWrite**	**Ext**	**Data_in_ sel**	**ALU_ src**	**Mem_add_ sel**
IF	**X**	**X**	ID	1	0	X	X	X
ID	**AND ADD**	**X**	R _ALU	0	0	X	X	X
ID	**ADDI**	**X**	I_ ALU	0	0	X	X	X
ID	**ANDI**	**X**	I_ALU	0	1	X	X	X
ID	**LW/LB**	**X**	AC	0	0	X	X	X
ID	**SW**	**X**	AC	0	0	X	X	X
ID	**Call**	**X**	CC	0	X	X	X	X
ID	**Sv**	**X**	SC	0	X	X	X	X
ID	**branch**	**X**	BC	0	0	X	X	X
R_ALU	**R-type**	**X**	RC	0	X	X	0	X
RC	**R_type**	**X**	IF	0	X	X	X	X
I_ALU	**I_type**	**X**	IC	0	X	X	1	X
IC	**I_type**	**X**	IF	0	X	X	X	X
BC	**branch**	**X**	IF	0	X	X	1	X
AC	**LW/LB**	**X**	L_Mem	0	X	X	1	X
AC	**SW**	**X**	S_Mem	0	X	X	1	X
L_Mem	**X**	**X**	LC	0	X	X	X	1
LC	**LW**	**X**	IF	0	X	X	X	X
LC	**LB**	**0**	IF	0	X	X	X	X
LC	**LB**	**1**	IF	0	X	X	X	X
S_Mem	**SW**	**X**	IF	0	X	0	X	1

BC	**X**	**X**	IF	0	X	X	X	X
CC	**Call**	**X**	IF	0	X	X	X	X
SC	**Sv**	**X**	IF	0	X	1	X	0

Table 8- Main control truth table part1

Inputs			Outputs					
Current state	**Opcode**	**m**	**Next State**	**RegWr**	**WBsel**	**Mem Rd**	**Mem Wr**	**Byte_Ext_sel**
IF	**X**	**X**	ID	0	X	0	0	X
ID	**AND ADD**	**X**	R _ALU	0	X	0	0	X
ID	**ADDI**	**X**	I_ ALU	0	X	0	0	X
ID	**ANDI**	**X**	I_ALU	0	X	0	0	X
ID	**LW/LB**	**X**	AC	0	X	0	0	X
ID	**SW**	**X**	AC	0	X	0	0	X
ID	**Call**	**X**	CC	0	X	0	0	X
ID	**Sv**	**X**	SC	0	X	0	0	X
ID	**branch**	**X**	BC	0	X	0	0	X
R_ALU	**R-type**	**X**	RC	0	X	0	0	X
RC	**R_type**	**X**	IF	1	00	0	0	X
I_ALU	**I_type**	**X**	IC	0	X	0	0	X
IC	**I_type**	**X**	IF	1	00	0	0	X
BC	**branch**	**X**	IF	0	X	0	0	X
AC	**LW/LB**	**X**	L_Mem	0	X	0	0	X
AC	**SW**	**X**	S_Mem	0	X	0	0	X
L_Mem	**X**	**X**	LC	0	X	1	0	X
LC	**LW**	**X**	IF	1	10	0	0	X
LC	**LB**	**0**	IF	1	01	0	0	1
LC	**LB**	**1**	IF	1	01	0	0	0
S_Mem	**SW**	**X**	IF	0	X	0	1	X
BC	**X**	**X**	IF	0	X	0	0	X
CC	**Call**	**X**	IF	1	11	0	0	X

SC	**Sv**	**X**	IF	0	X	0	1	X

Table 9- Main control truth table part 2

The logical expressions that are derived for each signal:

```
IRwrite = IF

Ext = IF.ANDI

Data_in_sel = SC

ALU_src = I_ALU + AC + BC

Mem_add_Sel = L_Mem + S_Mem

RegWr = RC + IC + CC + LC

WB_sel [0] = LC.LB + CC

WB_sel [1] = LC.LW + CC

MemRd = L_Mem

MemWr = S_Mem + SC

Byte_ext_Sel = LC.LBu (opcode = 0110 and m = 0)
```

1.5.2 Designing the PC control

PC control unit takes the opcode, mode bit, Alu flags and state as inputs and determine the value of PC control signals: PCwrite (one bit) and PCsrc (2-bits).

PCwrite is one bit signal. When it is enabled a new value is written on the PC register. It is set to 1 at the last stage in each instruction. For example, in the branch instructions, PCwrite is zero in the instruction fetch and instruction decode stages. It is set to one in the branch completion state in which the instruction finishes execution and the flags are updated. so the PCsrc can be determined (if the branch is taken or not) and the PC is updated to the right value in the next cycle.

PCsrc signal determines which address will be written on the pc as follows:

0 :Next PC = PC + 2

1: Next PC = Branch target address

2: Next PC = Jump/Call target address

3: Next PC = bus_read1 (R7 register content in Ret instruction)

The following table shows the PC control unit truth table.

Inputs						Outputs	
State	**Opcode**	**m**	**Zero**	**negative**	**overflow**	PCwrite	PCsrc
Instruction Decode	**Jmp**	**X**	**X**	**X**	**X**	1	10
Instruction decode	**RET**	**X**	**X**	**X**	**X**	1	11
Call completion	**Call**	**X**	**X**	**X**	**X**	1	10
Branch completion	**BEQ/BEQZ 1010**	**X**	**0**	**X**	**X**	1	00
Branch completion	**BEQ/BEQZ 1010**	**X**	**1**	**X**	**X**	1	01
Branch completion	**BGT/BGTZ 1000**	**X**	**0**	**0**	**0**	1	01
Branch completion	**BGT/BGTZ 1000**	**X**	**0**	**1**	**1**	1	01

Branch completion	**BGT/BGTZ 1000**	**X**	**0**	**0**	**1**	1	00
Branch completion	**BGT/BGTZ 1000**	**X**	**0**	**1**	**0**	1	00
Branch completion	**BGT/BGTZ 1000**	**X**	**1**	**X**	**X**	1	00
Branch completion	**BLT/BLTZ 1001**	**X**	**X**	**0**	**0**	1	00
Branch completion	**BLT/BLTZ 1001**	**X**	**X**	**1**	**1**	1	00
Branch completion	**BLT/BLTZ 1001**	**X**	**X**	**1**	**0**	1	01
Branch completion	**BLT/BLTZ 1001**	**X**	**X**	**0**	**1**	1	01
Branch completion	**BNE/BNEZ 1011**	**X**	**1**	**X**	**X**	1	00
Branch completion	**BEQ/BEQZ 1011**	**X**	**0**	**X**	**X**	1	01
R_type_completion	**X**	**X**	**X**	**X**	**X**	1	00
I_type_completion	**X**	**X**	**X**	**X**	**X**	1	00
Load_completion	**LW**	**X**	**X**	**X**	**X**	1	00
Store_mem_Access	**SW**	**X**	**X**	**X**	**X**	1	00
Sv_completion	**SV**	**X**	**X**	**X**	**X**	1	00

Table 10 - PC control truth table

The Logical expressions that were derived are as follows:

```
PCwrite = (Instruction_decode.(JMP+RET)) + Call_completion +
Branch_completion + I_type_Completion + R_type_completion +
load_completion + store_Mem_Acess + Sv_Completion


PCsrc[0] = instruction_decode.RET + Branch_completion.(BEQ.Z +
(BGT.Z’.(N⊕V)’) + (BLT.(N⊕V))+ BNE.Z’)
```

```
PCsrc[1] = instruction_decode.(JMP  + RET) + call_completion.
```

1.5.3 Designing the ALU control:

The ALU control unit has 2 inputs; the state and the opcode and 1 output the ALU opcode which determines the operation in the ALU unit. Before designing the ALU control we must take a look at the opcodes and operations needed in each instruction.

Opcodes for the ALU

The table below shows the necessary operation to be performed in each instruction in the execute stage. As we can notice we have 3 distinct ALU operations so we only need 2 bits (AND,ADD,SUB). The binary assignment is as follows:

AND_Op = 00

ADD_Op = 01

SUB_Op = 10

- An extra opcode was added for no-operation :

  No_Op = 11

No.	Instr	Opcode Value	ALU Operation	ALU Opcode (decimal)	ALU Opcode Value	State
1	**AND**	**0000**	AND_Op	0	00	R_type_ALU
2	**ADD**	**0001**	ADD_Op	1	01	R_type_ALU
3	**SUB**	**0010**	SUB_Op	2	10	R_type_ALU
4	**ADDI**	**0011**	ADD_Op	1	01	I_type_ALU
5	**ANDI**	**0100**	AND_Op	0	00	I_type_ALU
6	**LW**	**0101**	ADD_Op	1	01	Address_Computation
7	**LBu**	**0110**	ADD_Op	1	01	Address_Computation
8	**LBs**	**0110**	ADD_Op	1	01	Address_Computation
9	**SW**	**0111**	ADD_Op	1	01	Address_Computation
10	**BGT**	**1000**	SUB_Op	2	10	Branch_Completion
11	**BGTZ**	**1000**	SUB_Op	2	10	Branch_Completion

12	**BLT**	**1001**	SUB_Op	2	10	Branch_Completion
13	**BLTZ**	**1001**	SUB_Op	2	10	Branch_Completion
14	**BEQ**	**1010**	SUB_Op	2	10	Branch_Completion
15	**BEQZ**	**1010**	SUB_Op	2	10	Branch_Completion
16	**BNE**	**1011**	SUB_Op	2	10	Branch_Completion
17	**BNEZ**	**1011**	SUB_Op	2	10	Branch_Completion
18	**JMP**	**1100**	-	-	-	-
19	**CALL**	**1101**	-	-	-	-
20	**RET**	**1110**	-	-	-	-
21	**Sv**	**1111**	-	-	-	-

Table 11: Opcodes and ALU opcodes for each instruction

*instructions 18-21 do not go into execute stage and thus do not use ALU.

ALU control signal

Now we want to derive a Boolean expression for the ALU_opcode. We need to take into consideration both the opcode and the state of execution. We can write it behaviorally as follows:

```
If (  State != R_type_ALU || State != I_type_ALU ||
      State != Address_Computation || State != Branch_Completion)
            ALU_Opcode = No_Op
else if (State == Address_Computation)
            ALU_Opcode = ADD_Op
else If (Opcode == Branch_Completion)
            ALU_Opcode = SUB_Op
else : // R_type_ALU or I_type_ALU
      If (Opcode == SUB)
            ALU_Opcode = SUB_Op
```

```
        else If (Opcode == AND || Opcode == ANDI)
              ALU_Opcode = AND_Op
        else If (Opcode == ADD || Opcode == ADDI)
              ALU_Opcode = ADD_Op
```

This can also be expressed as Boolean expression for each of the bits of the ALU opcode as follows:

```
ALU_Opcode[0] = ADD. R_type_ALU + ADDI. I_type_ALU +Address_Computation +
Instruction_Fetch + Instruction_decode + Call_completion + I_type_Completion
+ R_type_completion + load_completion + Load_Mem_Acess + store_Mem_Acess +
Sv_Completion
```

```
ALU_Opcode[1] = SUB. R_type_ALU + Branch_completion +
Instruction_Fetch + Instruction_decode + Call_completion + I_type_Completion
+ R_type_completion + load_completion + Load_Mem_Acess + store_Mem_Acess +
Sv_Completion
```

It is important to note here that ALU_opcode is only set in the execution state, so if for example we have ANDI instruction the opcode will be No_op in all other stages, in instruction fetch, decode and write back.

2 Simulation and Testing

In our processor we managed to design and test all the 21 instructions. All the 21 instructions work properly and as expected. The explanation and verification of each will be shown in this part. This part takes each instruction and explain the changes at each cycle, additionally it shows how the program runs as a whole with consecutive instructions written and present in the instruction memory.

2.1 Testing Each Instruction Type

2.1.1 R-type Verification

To test R-type instructions the registers are initialized as shown in the following table.

Register	Initial value
R1	1
R2	2
R3	0
R4	7
R5	0
R6	6

Table 12- Registers initial values

the Instruction memory is initialized with the following instructions:

Memory address	Instruction in hexadecimal	Instruction in Assembly	Expected ALU_Result	Expected Update on reg file
{memory[1], memory[0]}	1688	ADD R3,R2,R1	3	R3 = 3
{memory[3], memory[2]}	2B18	SUB R5,R4,R3	4	R5 = 4
{memory[5], memory[4]}	0D28	AND R6,R4,R5	4	R6 = 4

Table 13- Tested R_type instructions

Note: Initial PC = 0 and clock period = 10 ns.

The simulation result is shown in the following waveform:

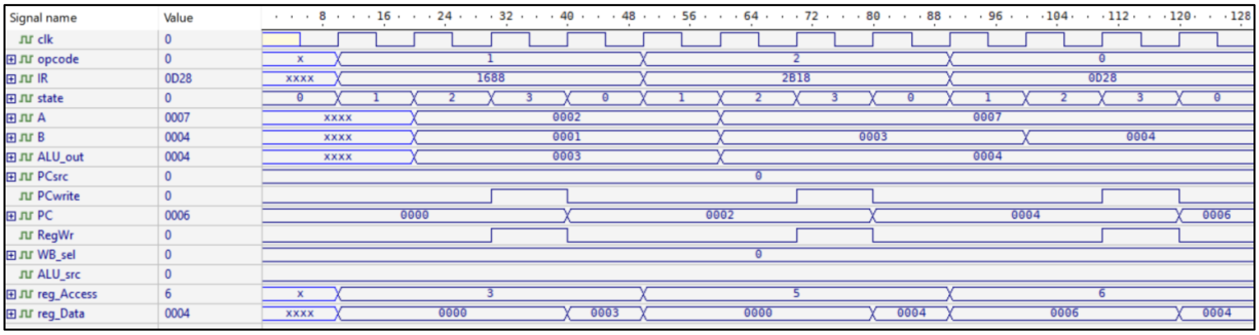

Figure 10- R_type test simulation result

The waveform illustrates the execution of three R type instructions. Each instruction takes 4 cycles and processed through 4 different states. The sequential states corresponds to fetch, decode, R_type_ALU(execute) and R_type competion (write back). The reg_Data signal shows a specified register value selected by the register access signal. This signal is used to show the register value after the write back stage.

In the execution cycle of the first instruction, the ALU result was 3 as expected. RegWr signal was enabled In the completion state (write back stage) and the register is updated in the next clock edge as shown in reg_Data signal in the waveform. Also, the PC is updated to PC + 2 in this state so a new instruction is fetched in the next state. The second and third instructions are executed in the same way and the ALU_result, Reg_Data values were as expected and that indicates the correctness of the processor design.

2.1.2 I-type (ALU) Verification

To test I-type ALU instructions the registers are initialized as shown in the following table.

Register	Initial value
R2	2
R3	0
R4	7
R5	-2
R6	2

Table 14- Registers initial values

The following table shows the tested I-type ALU instructions in the instruction memory.

Memory address	Instruction in hexadecimal	Instruction in Assembly	Expected ALU_Result	Expected Update on reg file
{memory[1], memory[0]}	34A9	ADDI R4,R5,9	7	R4 = 7
{memory[3], memory[2]}	4688	ANDI R6,R4,8	0	R6 = 0

Table 15- Tested R_type instructions

Note: Initial PC = 0 and clock period = 10 ns.

The simulation result is shown in the following waveform:

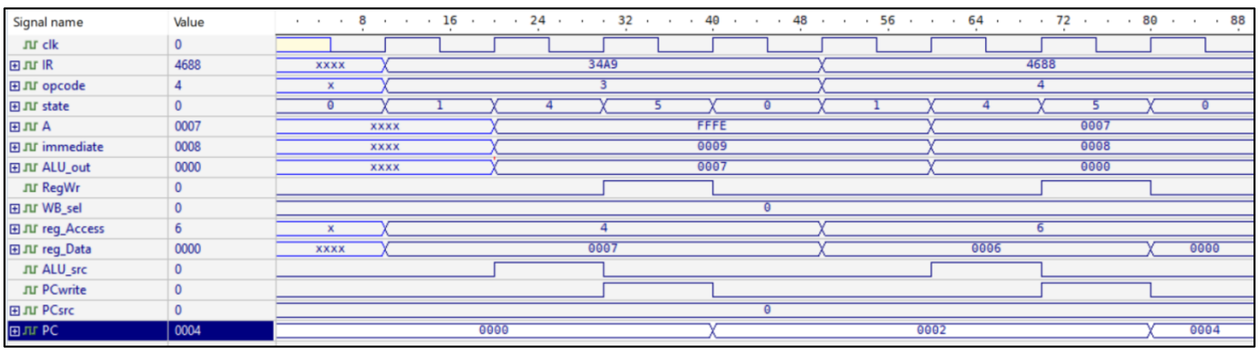

Figure 11- I_type ALU test simulation result

The waveform illustrates the execution of the I-type ALU instructions (ADDI and ANDI). Each instruction takes 4 cycles and processed through 4 different states. The sequential states (0, 1, 4, 5) corresponds to fetch, decode, I_type_ALU and I_type_completion (write back). In both instructions, the result is computed in the third cycle (state 4 = I_type_ALU) and it matches the expected result. ALUsrc is set to 1 to choose the immediate as the second ALU operand. At state 5 (I_type_completion), WB_sel is set to 1 and the RegWr is enabled so the ALU result is written on the register file at the next clock edge. In the same state ,PCsrc is 0 and PCwrite is 1 so the PC is updated to PC + 2 at the end of each instruction.

2.1.3 LW Verification

To test Load word instructions the registers are initialized as shown in the following table.

Register	Initial value – in hexadecimal
R1	0001
R2	0002
R3	000F
R4	32F5
R5	0005
R6	0006
R7	0007

Table 16- Registers initial values for Load

the Instruction memory is initialized with the following instructions:

Memory address	Instruction in hexadecimal	Instruction in Assembly
{memory[1], memory[0]}	554A	LW R5 ,R2, 10 (R5 = Mem[R2 + 10])
{memory[3], memory[2]}	2AD0	SUB R5,R3,R2

Table 17- Tested Load Instruction Memory initialization

the data memory is initialized with the following data:

Memory address	data in hexadecimal
{memory[13], memory[12]}	000A

Table 18- Tested Load Data Memory initialization

The simulation result is shown below:

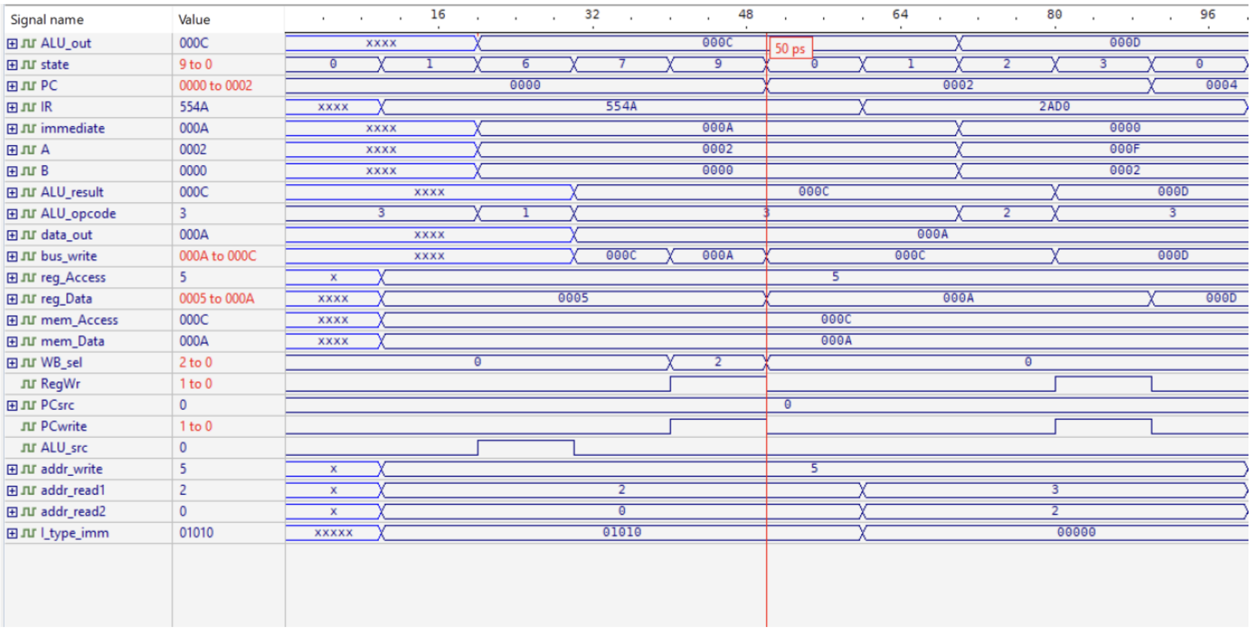

Figure 12: Load instruction test waveform

The initial simulation shows the PC set at 0, the clock cycle is 10ps, reg_Data shows the value of register 5 and mem_Data shows the value at {memory[13], memory[12]}, IR shows the current instruction after being fetched . The Load instructions starts at 0 and finishes execution at 50ps. We can notice the states of excution, state 0 for instruction fetch then state 1 for instruction decode. At instruction decode addr_write is set for 5, because writing will be on reg 5 , addr_read1 is 2 for reg 2 and I_type_immediate is shown to be 5 bits in binary with value of 10. At state 6 which is address computation, ALUsrc is set to 1 to choose the immediate as second operand, the ALU opcode in this stage is set to 1 which is the addition opcode. At state 7, the instruction is in load memory access, where mem_add_selection signal is set to 1, to choose address computed by ALU, which is shown in the ALU result to be (000C) or 12 in decimal, mem_read signal is also set to 1 to enable reading. In this state the value of the data_out read from memory is now (000A), which is the value at {memory[13], memory[12]}. At the last stage, state 9 (Load completion), RegWr is enabled, and we can notice the value of R5 changes by the end of this stage to (000A). At this stage PCwrite is enabled and PCsrc is chosen 0, by the end of this stage PC is updated to PC+2.

2.1.4 **<u>LBu & LBs Verification</u>**

To test Load word instructions the registers are initialized as shown in the following table.

Register	Initial value – in hexadecimal
R3	000F
R4	32F5

Table 19- Registers initial values for LBu, LBs

the Instruction memory is initialized with the following instructions:

Memory address	Instruction in hexadecimal	Instruction in Assembly
{memory[1], memory[0]}	634A	LBu R3 ,R2, 10
{memory[3], memory[2]}	6C4A	LBs R3 ,R2, 10

Table 20- Tested Load Byte Instruction Memory initialization

the data memory is initialized with the following data:

Memory address	data in hexadecimal
{memory[12], memory[13]}	F590

Table 21- Tested Load Byte Data Memory initialization

The load byte signed and unsigned goes into the same stages as the load, but differs in some signals. The simulation result is shown as follows:

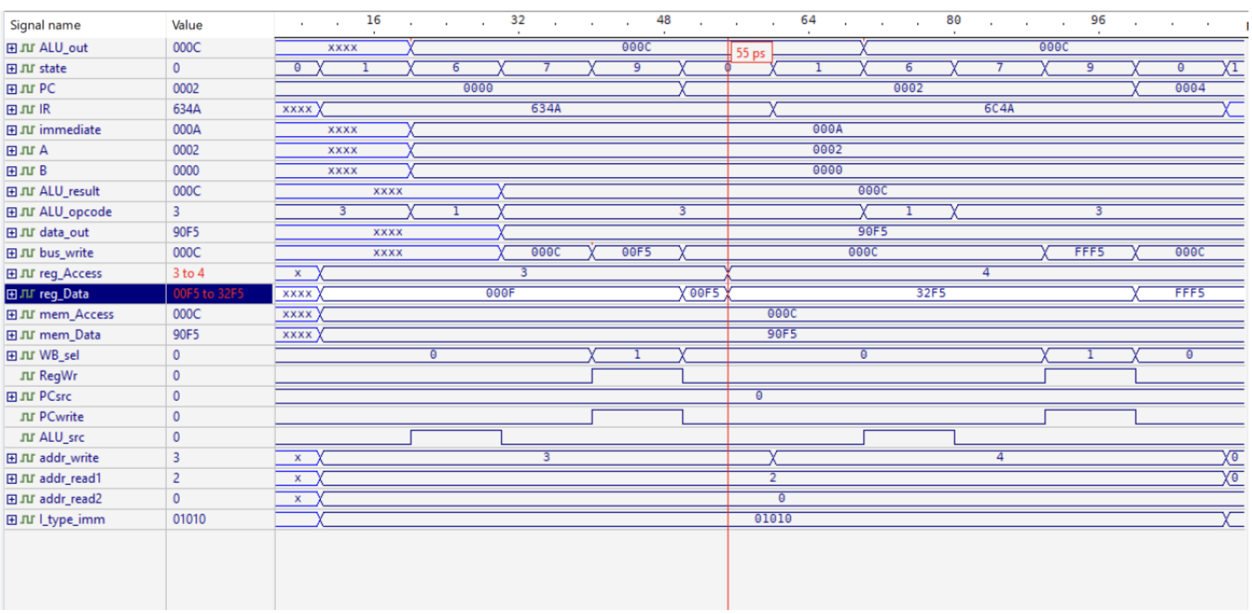

Figure 13: LBu and LBs instruction test waveform

For both the load byte signed and unsigned the WB selection is set to 1, which is the output of the byte extender. In the first instruction which is load unsigned, the Byte_ext_selection is set to 1 for unsigned and in the next instruction it is set to 0 for signed. The write back data is shown on bus write at stage 9 , where in the unsigned the data is (00F5) and in the signed it is (FFF5).. Reg_access gives number of register whose value is show at reg_Data. The resister have been assigned new values by the end of stage 9.

2.1.5 SW Verification

For verification of store word, same values have been used as the Load configuration initialization for data and register. Instruction memory is as follows:

Memory address	Instruction in hexadecimal	Instruction in Assembly
{memory[1], memory[0]}	744A	SW R4 ,R2, 10

Table 22: Tested Store Instruction Memory initialization

The result of the simulation is as follows:

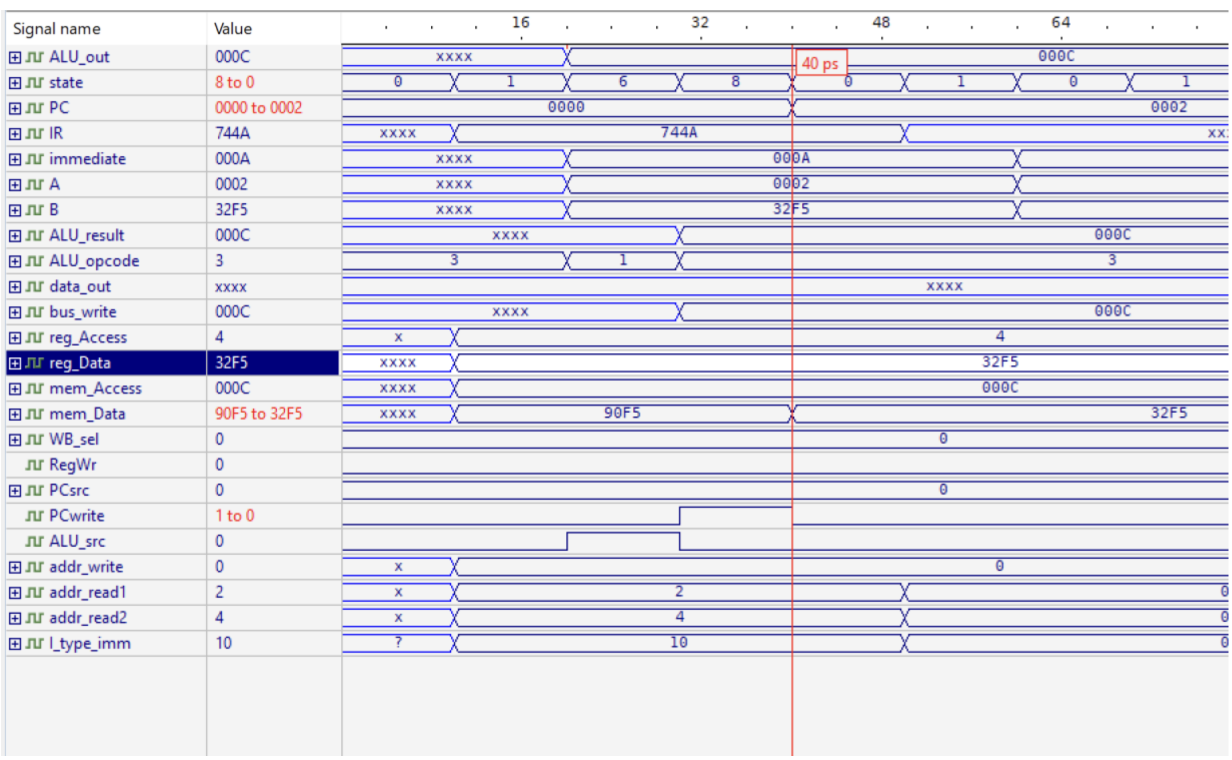

Figure 14: Store instruction test waveform

This shows that the store stage goes into 4 states and thus takes 4 clock cycles. After the first 2 stages which are the fetch and decode, addr_read1 is set to 2 to read the reg 2 and addr_read2 to 4 to get value to be saved at the memory location. At stage 6 the instruction same as the load goes into address computation stage and finally it goes to stage 8, which is Store_Mem_Access. At this stage MemWr is set to 1 and data_in_sel is set to 0 to choose the value of bus_read2 to be written at the computed memory. At the end of this stage, the memory cell 12,13 is changed to 32F5 which is the value of reg4, this change is shown at mem_Data.

2.1.6 SV Verification

The SV instruction is also a store instruction similar to the store word. To verify it the registers has been initialized same as load.

Instruction memory is as follows:

Memory address	Instruction in hexadecimal	Instruction in Assembly
{memory[1], memory[0]}	FC32	SV R6 ,50 (immediate value 50 is in decimal)

Table 23: Tested S-type Instruction Memory initialization

The simulation result is as follows:

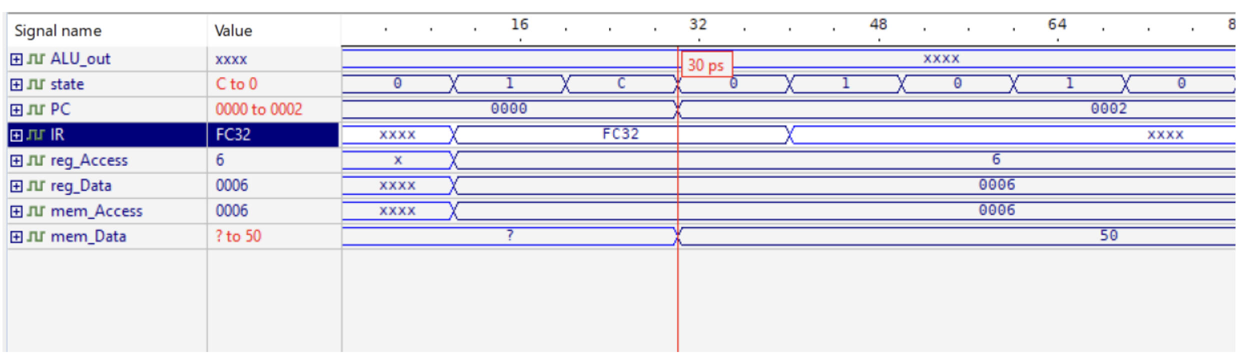

Figure 15: SV instruction test waveform

This instruction goes into 3 stages: Fetch, Decode and Memory (Sv_completion).

The register value at R6 is 6 so the value of the immediate will be stored at data memory {memory[7], memory[6]}. The immediate value that will be sored is 50 as decoded in the instruction, by the end of the 3rd stage the value of memory location {memory[7], memory[6]}, is set to 50 as shown in mem_Data.

2.1.7 Branch Verification

To test Branch instructions the registers are initialized as shown in the following table.

Register	Initial value
R1	1
R2	2
R3	0
R4	7
R5	2

Table 24- Registers initial values

the Instruction memory is initialized with the following instructions:

Memory address	Instruction in hexadecimal	Instruction in Assembly
{memory[1], memory[0]}	8D08	BGTZ R5, 8 (if R5 > 0 : PC <- PC +8)
{memory[3], memory[2]}	2B18	SUB R5,R4,R3
{memory[9], memory[8]}	1688	ADD R3,R2,R1

Table 25- Tested branch instructions

Note: Initial PC = 0 and clock period = 10 ns.

The simulation result is shown in the following waveform:

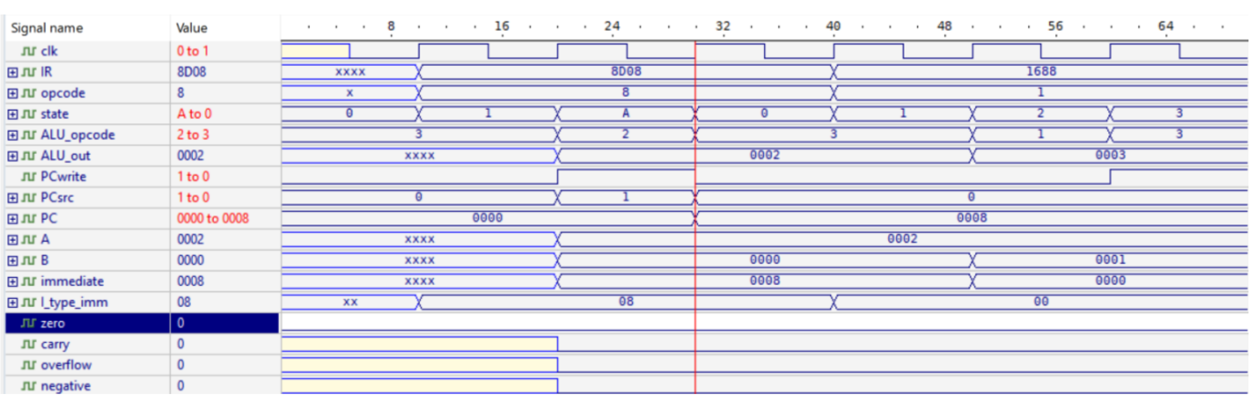

Figure 16_ Branch instruction test waveform

The initial value of PC equals 0 so the first executed instruction is BGTZ R5, 8. As shown in the previous figure this instruction takes 3 cycles in the data path and processed through three different stages. The sequential states (0, 1, A) corresponds to fetch, decode, and branch completion (ALU stage). In the third cycle, the instruction is in the ALU stage. The ALU_opcode is 2 so the operation is SUB. The flags are set by the ALU and the condition (zero == 0 && negative == overflow) is satisfied . Therefore, R5 is greater than R0 and the PCsrc value changed to 1 (Branch target address) and the PC changed to (PC + 8).

The instruction finished execution at 30 ns. And the next fetched instruction was ADD R3,R2,R1.

The same instructions are executed but the value of R5 is changed to -1 and the simulation is repeated. The result is shown in the following waveform.

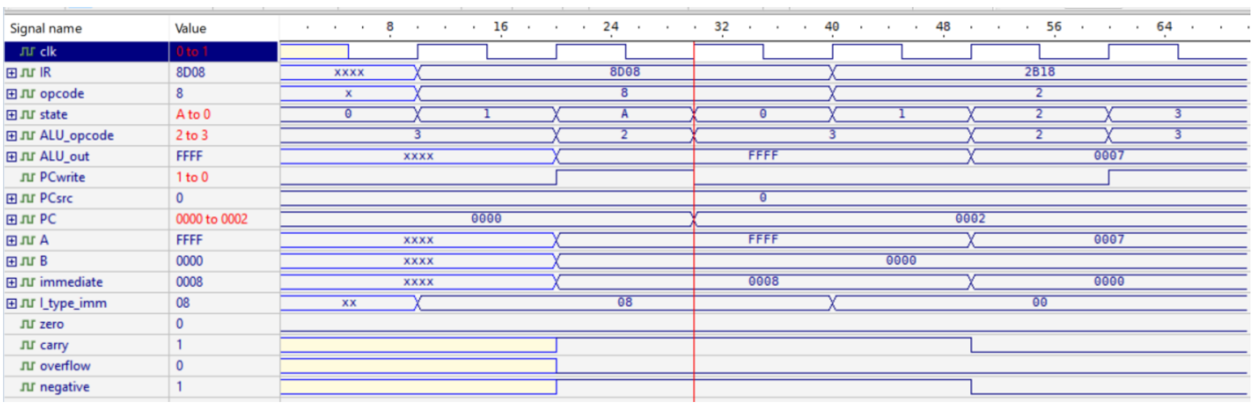

Figure 17- Branch instruction test 2 waveform

After changing R5 to -1, the branch condition is not satisfied (onerflow != Negative). So R5 is not greater than R0. In this case the PCsrc is 0 and the next PC is PC+2 so the second instruction in this waveform is SUB R5,R4,R3.

Another Branch instruction test:

the Instruction memory is initialized with the following instructions:

Memory address	Instruction in hexadecimal	Instruction in Assembly
{memory[1], memory[0]}	2B18	SUB R5,R4,R3
{memory[5], memory[4]}	A35C	BEQ R2,R3,-4 If (R2==R3) PC <= PC-4

Table 26- Tested Branch instructions (Test 2)

The initial values of Registers is shown in the following table:

Register	Initial value
R1	1
R2	2
R3	2
R4	7

Note: in this simulation the initial value of PC was 4.

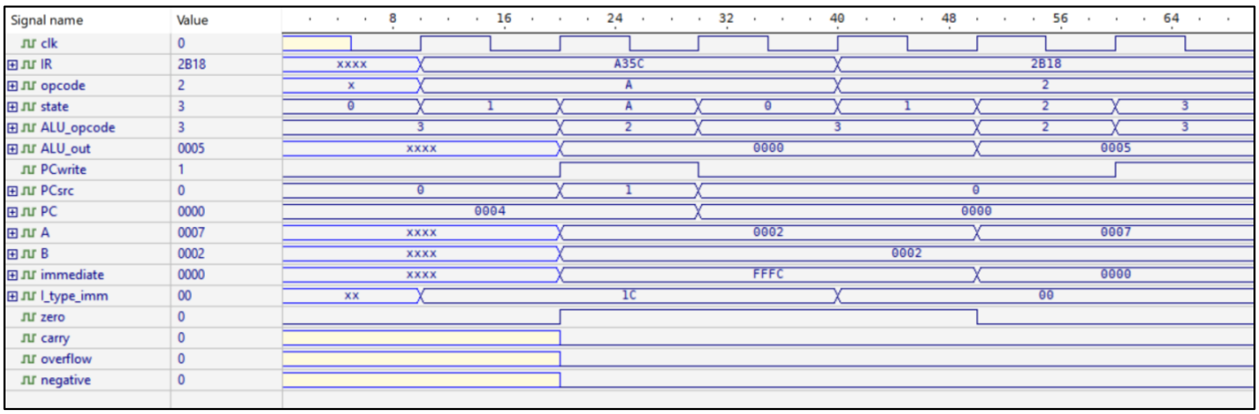

Figure 18- Test Branch instruction waveform (Test2)

The waveform illustrates that the initial value of the PC is 4. So the first processed instruction is BEQ R2,R3,-4. In the third cycle, the instruction is in the ALU stage. The ALU_opcode is 2 so the operation is SUB. The flags are set by the ALU and the condition (zero == 1) satisfied . Therefore, R3 equals R2 and the PCsrc value changed to 1 (Branch target address) and the PC changed to (PC – 4 = 0).The instruction finished execution at 30 ns. And the next fetched instruction was SUB R5,R4,R3.

2.1.8 Jump Verification

In the next 3 sections we will verify the 3 J-type instructions. For that memory is initialized with these 3 instructions:

Memory address	Instruction in hexadecimal	Instruction in Assembly
{memory[1], memory[0]}	C01E	JMP 30

{memory[31], memory[30]}	D028	Call 40
{memory[41], memory[40]}	E000	Ret

Table 27: Tested J-Type Instructions

The simulation result is as follows (note that reg7 data is initialized as 7, shown in reg_Data and PC is initialized at 0):

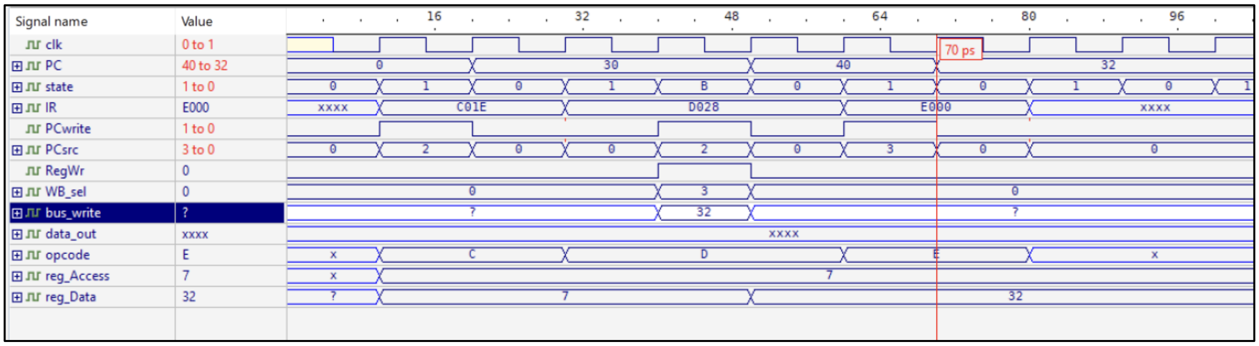

Figure 19- J-type instructions waveform

As for the J-type instruction it goes into two stages the fetch and decode. By the end of state 1, the PC value is changed to the concatenation of PC and the J-immediate, in this case it is 30 and PC is set to 30. The PC write is set to 1 and source is set to 2, to write the jump target address at PC.

2.1.9 Call Verification

Continuing in the previous section of J-verification, the call instruction goes to instruction fetch, decode and write back (Call_completion state). At the call completion satat, the value of PC+2 should be written, RegWr signal is enabled and WB_sel is chosen as 3 to write output of the adder (PC+2) at reg7. By the end of this reg7 value is set to 32 as shown in red_Data.

2.1.1 Return Verification

Continuing in the previous section of J-verification, the return instruction goes into two states instruction fetch, decode. During instruction decode PCwrite is enabled PCsrc is set to 3, this means that the new value of the PC will be the value stored at reg7, which is connected from bus_read1. After the return statement, the PC becomes 32, which means the excution of instructions continues from where it left.

3 Teamwork

This work is proudly done by both team members, each teammate contributed significantly in all parts of making this processor. In the design phase, Sarah started by breaking the instruction format and writing the stages and micro-operations of each operation, after discussing it and determining the needed components Sodos drew the data path to be implemented. Sarah later drew the state transition diagram. For control units design and signals needed, Sondos designed Main and PC control and Sarah designed ALU control.

In the coding part, Sarah wrote code for ALU, ALU control, muxes, adders, extenders, register file and the datapath connection of all units. Sondos wrote decoding and multiplexing unit, instruction memory, data memory, Main and PC control.

In the simulation and testing part. Sondos tested R-type, I-type immediate and Branch instructions. Sarah tested Load, Load Byte, Store, Sv and J-type operations. Necessary adjustments were made as we were testing.

As for writing the report we both made joint efforts to write about the parts we worked and tested on.

Conclusion

In conclusion, our project successfully designed and verified a simple pipelined RISC processor using Verilog. Through the steps taken, we have showcased the practical application of computer architecture and organization principles in making design decisions regarding microprocessor components. Starting with determining the instruction set architecture and conducting a thorough analysis, we gained a deeper understanding of the necessary environment and components required to construct an accurate datapath.

Our approach involved implementing a multi-cycle datapath where instruction goes into Fetch, Decode, Excute, Memory and WriteBack stages as necessary. We used a state machine to execute instructions across different stages and states. Throughout the project, we also documented each step of the design process and conducted verification of every instruction, explaining the control signals, storage elements and states of excution.

This comprehensive report shows the entire design journey, from the initial design phase to the final verification stage, demonstrating the successful realization of a functional multi-cycle RISC processor.