Faculty of Engineering and Technology

Electrical and Computer Engineering Department

Computer Architecture

ENCS4370

Project #2

Implementation and Validation of a Multi-Cycle RISC Processor Architecture Using Verilog

---

Prepared by :

Malak Moqbel 1210608

Omar Hussain 1212739

Instructor: Aziz Qaroush

Section: 3

Date: 20/06/2024

Abstract

This project focuses on designing and verifying a simple multicycle RISC processor using Verilog. The processor operates with 16-bit instructions and includes 8 general-purpose registers, a program counter, and separate memories for instructions and data. It supports four types of instructions: R-type, I-type, J-type, and S-type. By utilizing a multicycle execution process, the processor handles each step (fetch, decode, execute, memory access, write back) separately, optimizing resource use and improving overall efficiency. The design aims to balance performance and complexity, providing insights into the internal workings of a RISC processor.

The project involves a comprehensive design and implementation phase, including detailed descriptions of the processor's datapath, control signals, and integration of components to ensure proper operation. Verification is carried out through simulation and testing, using a testbench and multiple code sequences to confirm the processor's functionality. The report presents the design process, implementation details, and verification results, demonstrating the processor's capability to execute a variety of operations from basic arithmetic to complex decision-making. The inclusion of Verilog code further illustrates the processor's functionality and efficiency, highlighting the effectiveness of the multicycle approach in RISC processor design.

Table of Contents

Table of figures

Table of tables

1. Design Requirements

1.1 Motivation

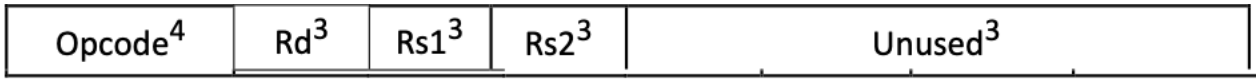
This endeavor is heavily influenced by the MIPS architecture, particularly the MIPS32, and is situated within the broader category of RISC (Reduced Instruction Set Computer) architecture. Our primary aim is to devise a comprehensive set of instructions tailored for a multicycle processor. The salient attributes of this instruction set include:

1. Each instruction is 16 bits long.
2. The design includes 8 general-purpose 16-bit registers, labeled R0 to R7.
3. R0 is hardwired to zero and cannot be altered.
4. The program counter (PC) is a 16-bit special-purpose register.
5. The processor supports four distinct instruction types: R-type, I-type, J-type, and S-type.
6. Separate physical memory units are used for instructions and data.
7. The Arithmetic Logic Unit (ALU) is utilized to determine the outcomes of specific instructions, such as deciding whether to take a branch.

1.2 Instruction Formats

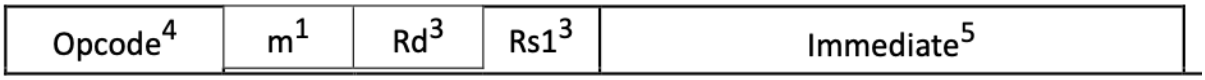
As previously indicated, this Instruction Set Architecture (ISA) encompasses four instruction types: R-type, I-type, J-type, and S-type. These instruction types share a common opcode field, which specifies the exact operation to be performed by the instruction.

1.2.1 R-type (Register Type)



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

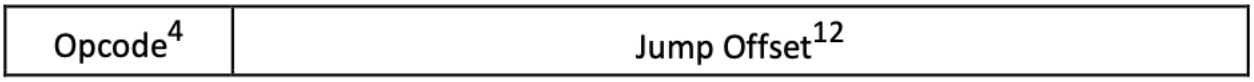
1.2.2 I-type (Immediate Type)



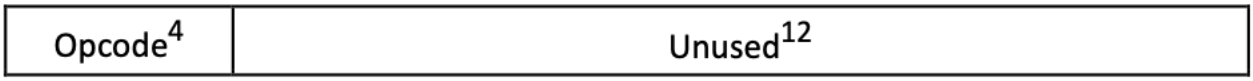
3-bit Rd: destination register
3-bit Rs1: first source register
5-bit immediate: unsigned for logic instructions and signed otherwise.
1-bit mode: this is used with load and branch instructions, such that:
For the load:
0: LBs load byte with zero extension
1: LBu load byte with sign extension
For the branch:
0: compare Rd with Rs1
1: compare Rd with R0

1.2.3 J-type (Jump Type)

This instruction type includes specific formats, each identified by a unique opcode: **jmp L** is an unconditional jump to the target label L, allowing the program to continue execution from that point, and **call F** calls the function labeled F, saving the return address in register r7 for proper return after function execution. The target address for these instructions is calculated by concatenating the most significant 7 bits of the current program counter (PC) with a 12-bit offset, which is multiplied by 2 (effectively shifted left by one bit), enabling efficient access to word-aligned memory locations.

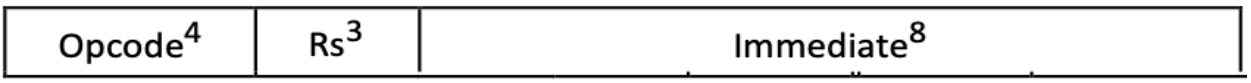


ret : return from a function, the next PC will be the value stored in r7.



1.2.1 S-type (Store)

This format ensures that the store operation is executed efficiently by directly calculating the memory address and storing the immediate value in one step



Sv rs, imm : `M[rs] = imm`

1.3 Instruction's Encoding

Table 1 Instruction's Encoding

R-Type			
AND	Reg(Rd) = Reg(Rs1) & Reg(Rs2)	0000	
ADD	Reg(Rd) = Reg(Rs1) + Reg(Rs2)	0001	
SUB	Reg(Rd) = Reg(Rs1) - Reg(Rs2)	0010	
I-Type			
ADDI	Reg(Rd) = Reg(Rs1) + Imm	0011	
ANDI	Reg(Rd) = Reg(Rs1) + Imm	0100	
LW	Reg(Rd) = Mem(Reg(Rs1) + Imm)	0101	
LBu	Reg(Rd) = Mem(Reg(Rs1) + Imm)	01001	0
LBs	Reg(Rd) = Mem(Reg(Rs1) + Imm)	0110	1
SW	Mem(Reg(Rs1) + Imm) = Reg(Rd)	0111	
BGT	if (Reg(Rd) > Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1000	0
BGTZ	if (Reg(Rd) > Reg(0)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1000	1
BLT	if (Reg(Rd) < Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1001	0
BLTZ	if (Reg(Rd) < Reg(R0)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1001	1
BEQ	if (Reg(Rd) == Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1010	0
BEQZ	if (Reg(Rd) == Reg(R0)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1010	1
BNE	if (Reg(Rd) != Reg(Rs1)) Next PC = PC + sign_extended (Imm)	1011	0
BNEZ	if (Reg(Rd) != Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1011	1
J-type			
JMP	Next PC = {PC[15:10], Immediate}	1100	
CALL	Next PC = {PC[15:10], Immediate} PC + 4 is saved on r15	1101	
RET	Next PC = r7	1110	
S-type			
sv	M[rs] = imm	1111	

2. Datapath Components

2.1 Multiplexer

The 4×1 multiplexer, or 4-to-1 multiplexer, has four input lines and one output line. The output is selected from one of the four inputs based on two selection lines, determined by the formula \(\log_2(4) = 2 \).

Multiplexers are important in computer data systems because they act as selectors, choosing one signal from multiple inputs based on a control signal. This makes the system more efficient. In our datapath design, we use multiplexers multiple times. Specifically, we used both 2×1 and 4×1 multiplexers in various parts of the design. The 4×1 multiplexer shown in Figure 1 is one of the multiplexers we used to choose the next PC address.

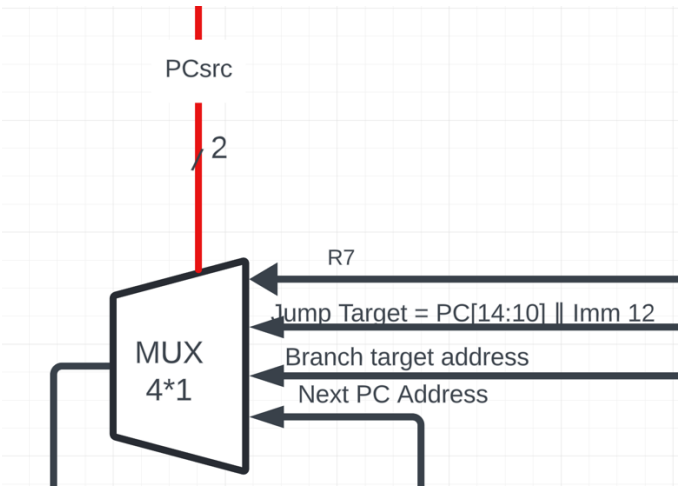


Figure 1 Multiplexer used in the datapath.

1. **PCsrc = 00**: The multiplexer selects the "Next PC Address" input which is PC+2.
2. **PCsrc = 01**: The multiplexer selects the "Branch target address" input.
3. **PCsrc = 10**: The multiplexer selects the "Jump Target = PC[14:10] || Imm 12" input. This input is used when a jump instruction is executed, and the processor needs to jump to an address formed by concatenating part of the program counter (PC) and an immediate value.
4. **PCsrc = 11**: The multiplexer selects the "R7" input. This might be used for RET instructions where the target address is stored in register R7.

Each of these selections enables the processor to determine the next address to fetch an instruction from, based on the control signals and the specific operation being executed.

2.2 ALU

The ALU (Arithmetic Logic Unit) in the diagram receives two data inputs and performs operations determined by the ALUOP control signal. The result of the operation is a 16-bit output, and the ALU also updates two status flags: N (Negative) and Z (Zero). These flags indicate if the result is negative or zero, respectively. The ALU is crucial in a processor, as it executes arithmetic and logical operations and provides status flags for conditional instructions.

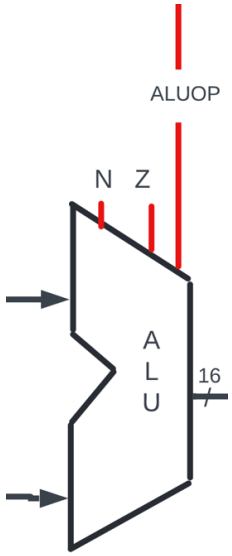


Figure 2 ALU symbol

This ALU module takes two 16-bit operands and performs an operation based on the 2-bit ALUop control signal, which can specify AND, ADD, or SUBTRACT operations. The result is stored in the 16-bit Output register. The ALU also updates two status flags: Zflag, which is set if the result is zero, and Nflag, which is set if the result is negative (indicated by the most significant bit being 1). The operation is enabled by the EN signal and synchronized with the clock signal, ensuring operations are performed on the rising edge of the clock.

2.3 Instruction Memory

The instruction Memory module in Verilog models an instruction memory unit with a 16-bit address input and a 16-bit instruction output, synchronized by a clock signal. It contains an internal memory array (`mem`) with 256 16-bit registers. On each positive clock edge, it fetches the instruction from the memory location specified by the address input and assigns it to the instruction output. The memory is pre-initialized with a set of example instructions, including basic operations and formatted instructions. This setup allows the module to provide instructions to the processor based on the current address, enabling program execution.

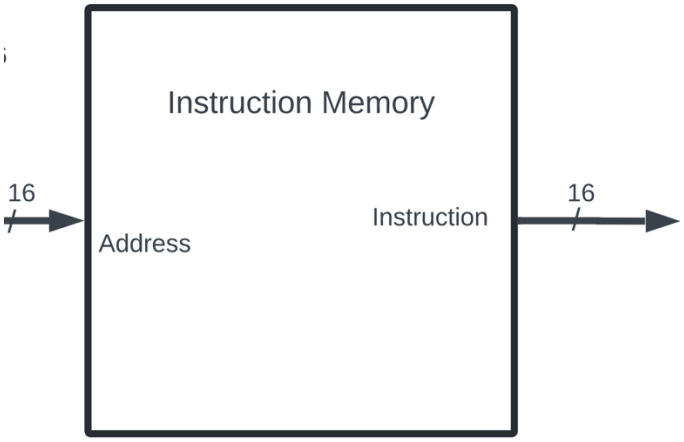


Figure 3 Instruction memory

2.4 Data Memory

The data Memory module in Verilog models a data memory unit with read and write capabilities, controlled by various input signals. The module takes inputs including a clock signal, memory read (memRead) and write (memWrite) control signals, and an enable (EN) signal. It has a 16-bit output (dataOut). The memory is represented by a 256-element array of 16-bit registers. On a positive clock edge, if enabled (EN), the module performs read or write operations based on the memRead and `memWrite` signals. When memRead is high, the module retrieves the data from the specified address and outputs it through dataOut. When memWrite is high, the module stores the input data at the specified address. The initial block sets up some test values in memory for validation. This module is designed to interface with a processor, providing data storage and retrieval based on specified addresses and control signals.

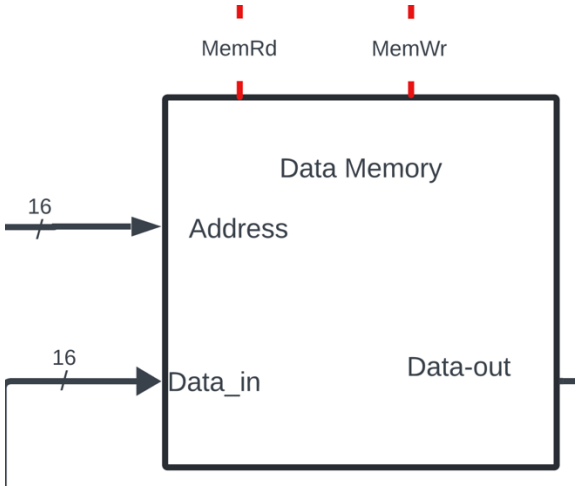


Figure 4 Data Memory

2.5 Register File

The reg_file module in Verilog represents a register file with eight 16-bit registers. It takes inputs including a clock signal (clk), a write enable signal (RegWrite), register addresses (RA, RB, RW), and a data input (BusW). The outputs are two data buses (BusA, BusB) and a specific register output (R7_out). On each positive clock edge, if enabled (EN), the module reads the values from the registers specified by RA and RB into BusA and BusB, respectively, and outputs the value of register 7 (R7_out). If RegWrite is high, it writes the value from BusW into the register specified by RW. The initial block sets predefined values in the registers for testing purposes.

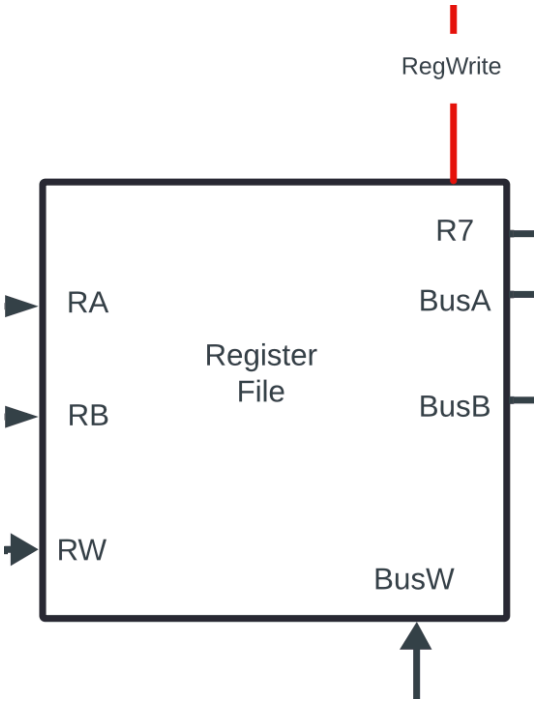


Figure 5 Register File

3. Control Unit

3.1 Signals

The control signals in a datapath design are critical for managing the flow of data and the execution of instructions within the processor. These control signals are managed by the control unit based on the current instruction's opcode and are crucial for directing the appropriate operations in each clock cycle, ensuring the correct execution of instructions and proper data flow within the processor.

Table 2 Signals effect

Signal Name	Value	Effect
PCsrc	00	The Next Pc address (pc+2)is written to the PC
	01	The Branch target address is written to the PC
	10	The jump target address is written to the PC
	00	R7 value is written to the PC
RegSrc1	0	Register input (RA) = Rs1
	1	Register input (RA) = R0
RegSrc2	0	Register input (RB) = Rd
	1	Register input (RB) = Rs2
Regdes	0	Register input (RW) = R7
	1	Register input (RW) = Rd
ExtOp	00	Zero extend for the Sv type immediate
	01	Unsigned extend for the 5bit immediate
	10	Check the ExtM signal
	11	signed extend for the 5bit immediate
ExtM	0	load byte with zero extension
	1	load byte with sign extension
ALUsrc	0	The Second AlU operand is the extended immediate
	1	The Second Alu operand is BusB
ALUOP	00	The ALU performs an AND operation
	01	The ALU performs an ADD operation
	10	The ALU performs an SUB operation

WriteBack	00	BusW<= Data memory (Data_out)
	01	BusW<= Alu result
	10	BusW<= Next PC (PC +2)

Address_src	0	Data Memory Address <= BusA
	1	Data Memory Address <= Alu result
Data_src	0	Data_in <= extended immediate
	1	Data_in <= BusB
MemRd	0	Memory read operation is disabled.
	1	Memory read operation is enabled.
MemWr	0	Memory write operation is disabled.
	1	Memory write operation is enabled.

3.2 State Diagram

The state diagram shows the sequence of operations for a processor, starting with fetching an instruction from memory and incrementing the program counter (PC). It then decodes the instruction by loading values from registers. Depending on the instruction type, the processor either performs register operations (R-type), adds immediate values (ADDI, ANDI), loads or stores data (LW, SW), or handles branch (BGT, BEQ) and jump (JMP, CALL, RET) instructions. Branch instructions conditionally update the PC, while jump and call instructions set the PC to specific addresses. Each instruction cycle ends by fetching the next instruction, continuing the process.

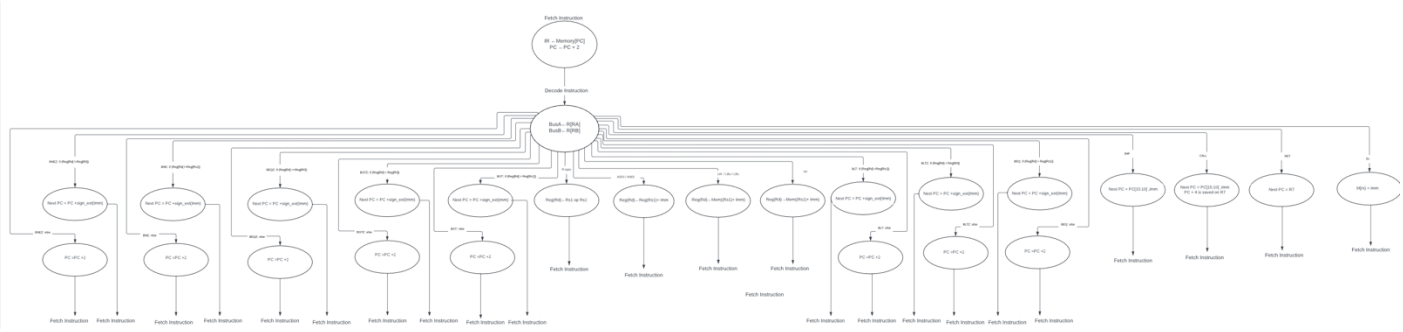


Figure 6 State Diagram

3.3 Truth Table

3.3.1 R-type Truth Table

Table 3 R-type truth table

Opcode	Pc_src	Reg_src1	Reg_src2	Regdes	EXtOp	ExtM	ALUsrc	RegWrite	ALUOP	Address_src	Data_src	MemRd	MemWr	WB
AND	00	0	1	1	X	X	0	1	00	X	x	x	X	01
ADD	00	0	1	1	X	X	0	1	01	X	X	X	X	01
SUB	00	0	1	1	X	x	0	1	10	x	X	X	x	01

3.3.2 I-type Truth Table

Table 4 I-type truth table

Opcode	Pc_src	Reg_src1	Reg_src2	Regdes	EXtOp	ExtM	ALUsrc	RegWrite	ALUOP	Address_src	Data_src	MemRd	MemWr	WB
ANDI	00	0	X	1	01	X	1	1	00	X	x	x	X	01
ADDI	00	0	X	1	01	X	1	1	01	X	X	X	X	01
LW	00	0	X	1	11	x	1	1	01	0	X	1	x	00
LBs	00	0	X	1	10	1	1	1	01	0	X	1	X	00
LBu	00	0	X	1	10	0	1	1	01	0	X	1	X	00
SW	00	0	0	X	XX	X	1	X	01	0	0	0	1	XX
BGT	01	0	0	X	11	0	0	X	10	X	X	X	X	XX
BGTZ	01	1	0	X	11	1	0	X	10	X	X	X	X	XX
BLT	01	0	0	X	11	0	0	X	10	X	X	X	X	XX
BLTZ	01	1	0	X	11	1	0	X	10	X	X	X	X	XX
BEQ	01	0	0	X	11	0	0	X	10	X	X	X	X	XX
BEQZ	01	1	0	X	11	1	0	X	10	X	X	X	X	XX
BNE	01	0	0	X	11	0	0	X	10	X	X	X	X	XX
BNEZ	01	1	0	X	11	1	0	x	10	X	x	X	X	XX

3.3.3 J-type Truth Table

Table 5 j-type truth table

Opcode	Pc src	Reg src1	Reg src2	Reg des	EXt Op	Ext M	Reg Write	ALU OP	Address _src	Data_src	Mem Rd	Mem Wr	WB
JMP	10	X	X	X	XX	X	X	XX	X	X	X	X	XX
CALL	10	X	X	0	XX	X	1	XX	X	X	X	X	10
RET	11	X	X	X	XX	X	X	XX	X	X	X	X	XX

3.3.4 S-type Truth Table

Table 6 S-type truth table

Opcode	Pc_src	Reg_src1	Reg_src2	Regdes	Ext Op	Ext M	RegWrite	ALUOP	Address _src	Data_ src	Mem Rd	Mem Wr	WB
sv	00	1	X	X	00	X	X	XX	1	1	0	1	X

3.4 Boolean equations

Pc_src0 = JMP + RET

Pc_src1= BGT + BGTZ + BLT+ BLTZ + BEQ + BEQZ + BNE + BNEZ + RET

Reg_src1 = BGTZ + BLTZ + BEQZ+ BNEZ + sv

Reg_src2 = AND + ADD + SUB

Regdes = AND + ADD +SUB+ ANDI+ ADDI+ LW +LBs +LBu

EXTOP0= BGT + BGTZ + BLT+ BLTZ + BEQ + BEQZ + BNE + BNEZ +LBs +LBu + LW

EXTOP1= ANDI +ADDI+ LW + BGT + BGTZ + BLT+ BLTZ + BEQ + BEQZ + BNE + BNEZ

EXTM= LBs +BGTZ+ BLTZ+ BEQZ+ BNEZ

ALUsrc = ANDI +ADDI+ LW+ LBs +LBu + SW

RegWrite = AND +ADD+ SUB+ ANDI+ ADDI+ LW +LBs +LBu +CALL

ALUOP0=SUB+ BGT + BGTZ + BLT+ BLTZ + BEQ + BEQZ + BNE + BNEZ

ALUOP1 = ADD+ADDI + LW+ LBs +LBu + SW

Address_src = sv

Data_src= sv

MemRd= LW +LBs + LBu

MemWr= SW +sv

WR0= CALL

WR1= AND +ADD+ SUB+ ANDI+ ADDI

3.5 Constant File

Table 7 Constant File

PC Source selection	
pcDefault	2'b00
pcSgnImm	2'b01
pcImm	2'b10
R7PC	2'b11
ALU Operations	
ALU_AND	2'b00
ALU_ADD	2'b01
ALU_SUB	2'b10
Write Back Selection	
WB_Default	2'b00
WB_One	2'b01
WB_Two	2'b10
Enable and Disable Signals	
LOW	1'b0
HIGH	1'b1
Register selection for specific operations	
R0	
R7	
Opcode	
AND	4'b0000
AND	4'b0001
SUB	4'b0010
ADDI	4'b0011
ANDI	4'b0100
LW	4'b0101
LBu	4'b0110
LBs	4'b0110
SW	4'b0111
BGT	4'b1000
BGTZ	4'b1000
BLT	4'b1001
BLTZ	4'b1001
BEQ	4'b1010,
BEQZ	4'b1010
BNE	4'b1011
BNEZ	4'b1011
JMP	4'b1100
CALL	4'b1101
RET	4'b1110
Sv	4'b1111
Extender Operations	
Zero	2'b00
One	2'b01
Two	2'b10

3.6 Datapath Stages

3.6.1 Initial Stage

The Initialization stage (STG_INIT) serves as the starting point for the control unit, ensuring that the system is properly set up to begin instruction processing. During this stage, the Instruction Fetch stage (IF_Stage) is disabled by setting it to LOW, and the control unit prepares to transition to the next stage by setting nextStage to the Instruction Fetch stage (nextStage <= STG_FTCH).

3.6.2 Instruction Fetch Stage

In the Instruction Fetch stage (STG_FTCH), the control unit fetches the next instruction from memory. This stage begins by disabling all next stages (ID_Stage , E_Stage , Mem_Stage , WRB_Stage are set to LOW) and deactivating control signals (sigENW, sigMemW, sigMemR are set to LOW). It then enables the Instruction Fetch stage (IF_Stage = HIGH). Based on the OpCode and condition flags (ZFlag, NFlag), the control unit determines the source for the Program Counter (PCSrc),for branch instructions (BGT, BLT, BEQ, BNE), it sets PCSrc to pcSgnImm if conditions are met, for jump instructions (JMP, CALL), it sets PCSrc to pcImm , for return instructions (RET), it sets PCSrc to R7, otherwise, it defaults to pcDefault. The next stage is set to Instruction Decode (nextStage <= STG_DCDE).

3.6.3 Instruction Decode Stage

In the Instruction Decode stage (STG_DCDE), the control unit decodes the fetched instruction and prepares the necessary signals for execution. It begins by disabling the Instruction Fetch stage (IF_Stage = LOW) and enabling the Instruction Decode stage (ID_Stage = HIGH). Based on the OpCode, it sets up various control signals: for CALL, it sets DstReg to LOW, WB to 2'b10, and sigENW to HIGH, then transitions back to the Instruction Fetch stage (nextStage <=STG_FTCH); for JMP and RET, it also transitions back to STG_FTCH; otherwise, it transitions to the Execute stage (nextStage <= STG_EXEC). It configures the extender and register source signals (ExtOp, ExtM, RegSrc1, RegSrc2, DstReg, ALUSrc) based on specific OpCode values.

3.6.4 Execute Stage

The Execute stage (STG_EXEC) is where the actual computation takes place. It starts by disabling the Instruction Decode stage (ID_Stage = LOW) and deactivating sigENW (set to LOW). The Execute stage is then enabled (E_Stage = HIGH). The next stage is determined by the OpCode: for load and store instructions (LW, SW, LBs, LBu), it transitions to the Memory stage (nextStage <= STG_MEM); for arithmetic and logical instructions (AND, ADD, SUB, ANDI, ADDI), it transitions to the Write Back stage (nextStage <= STG_WRB); otherwise, it returns to the Instruction Fetch stage (nextStage <= STG_FTCH). The ALUOp signal is set based on the Opcode: for AND and ANDI, it sets ALUOp to 2'b00 (ALU_AND); for ADD, ADDI, LW, SW, LBs, LBu, it sets ALUOpto 2'b01 (ALU_ADD); otherwise, it sets ALUOp to 2'b10 (ALU_SUB).

3.6.5 Memory Stage

The Memory stage (STG_MEM) handles memory access operations. It begins by disabling the Execute stage (E_Stage = LOW) and the Instruction Decode stage (ID_Stage = LOW). The Memory stage is then enabled (Mem_Stage = HIGH). The next stage is determined by the OpCode: for load instructions (LW, LBs, LBu), it transitions to the Write Back stage (nextStage <= STG_WRB); otherwise, it returns to the Instruction Fetch stage (nextStage <= STG_FTCH). The sigMemW signal is set to HIGH for store instructions (SW, Sv) and LOW otherwise. The sigMemR signal is set to HIGH for load instructions (LW, LBs, LBu) and LOW otherwise.
The Address_src and Data_src signals are configured based on whether the operation is a store (Sv) or another memory access.

3.6.6 Write Back Stage

In the Write Back stage (STG_WRB), the control unit completes the instruction execution by writing results back to the register file. It starts by disabling memory access signals (sigMemW = LOW , sigMemR = LOW) and the Execute stage (E_Stage = LOW). The Write Back stage is enabled (WRB_Stage = HIGH). The next stage is the Instruction Fetch stage (nextStage <= STG_FTCH). The sigENWsignal is set to HIGH to enable writing to the register file. The WB signal is configured based on the Opcode: for arithmetic and logical instructions (AND, ADD, SUB, ANDI, ADDI), it sets WB to 2'b01; for load instructions (LW, LBs, LBu), it sets WB to 2'b00; otherwise, it sets WB to 2'b10.

4. Full Datapath

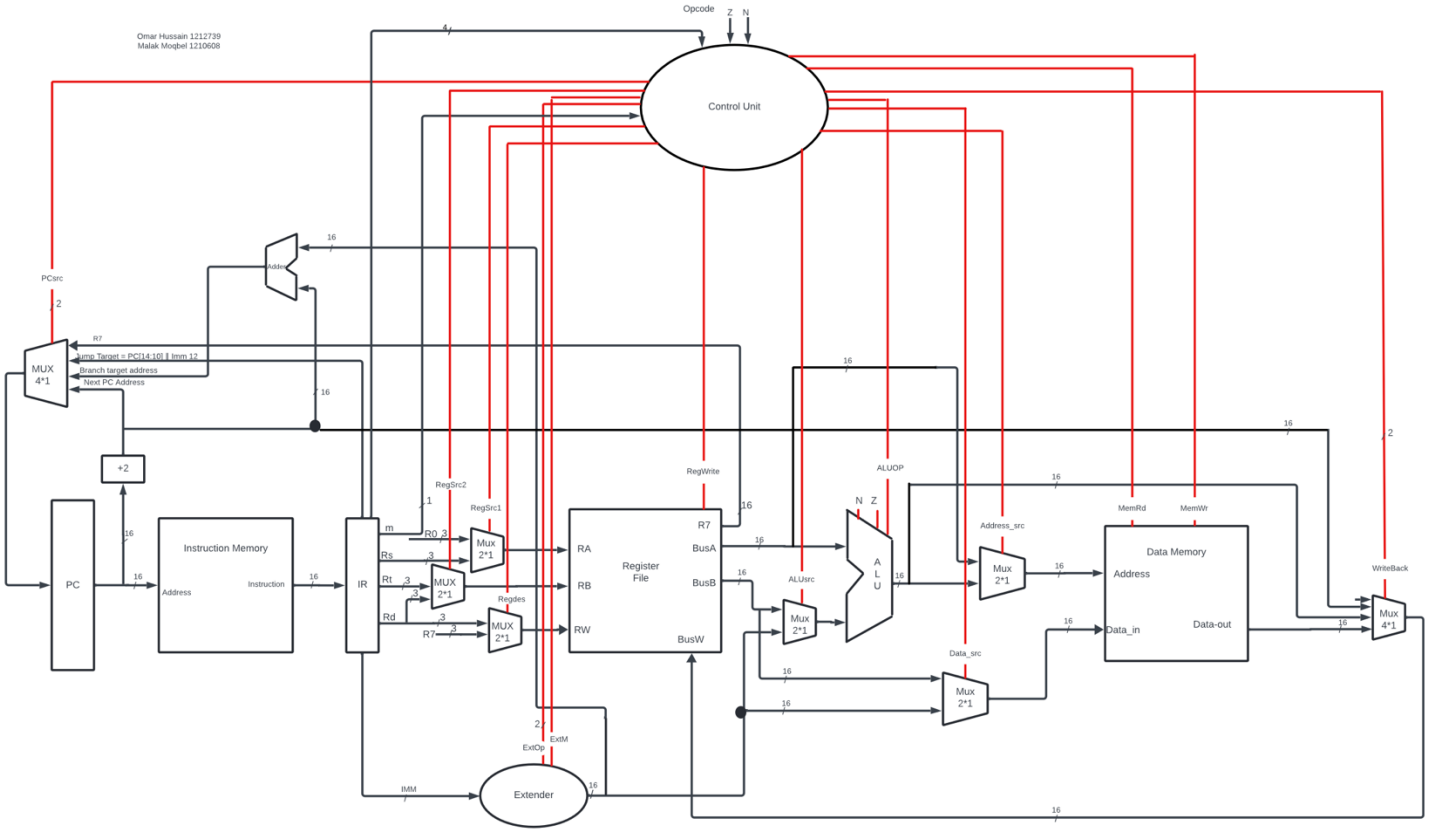


Figure 7 Datapath

5. Simulation and Testing

The test program used for this simulation consists of a sequence of instructions designed to verify the correct operation of the processor's components, including the ALU, register file, data memory, and control unit.

5.1 Test Program 1

The initial memory values are set up using the provided code snippets.

```
initial begin
    // Store initial values in the memory for testing purposess
    {memory[1],memory[0]} = 16'h1210;
    {memory[3],memory[2]} = 16'h1110;
    {memory[5],memory[4]} = 16'h1014;

end
```

Figure 8 Initial Memory Test case 1

Instruction 1: `{mem[1], mem[0]} = {4'b0101, 1'b0, 3'b011, 3'b100, 5'b00011};`
(LW R3, R4, 3)
Loads the word from memory address R4 + 3 into register R3.
Expected Output: R3 should hold the value from memory address R4 + 3.

The following image shows the initialization of the instruction memory with the test program:

```
{mem[1], mem[0]} = {4'b0101, 1'b0, 3'b011, 3'b100, 5'b00011};
```

Figure 9 Instruction Test case 1

The following image shows the waveform diagram for the test program:

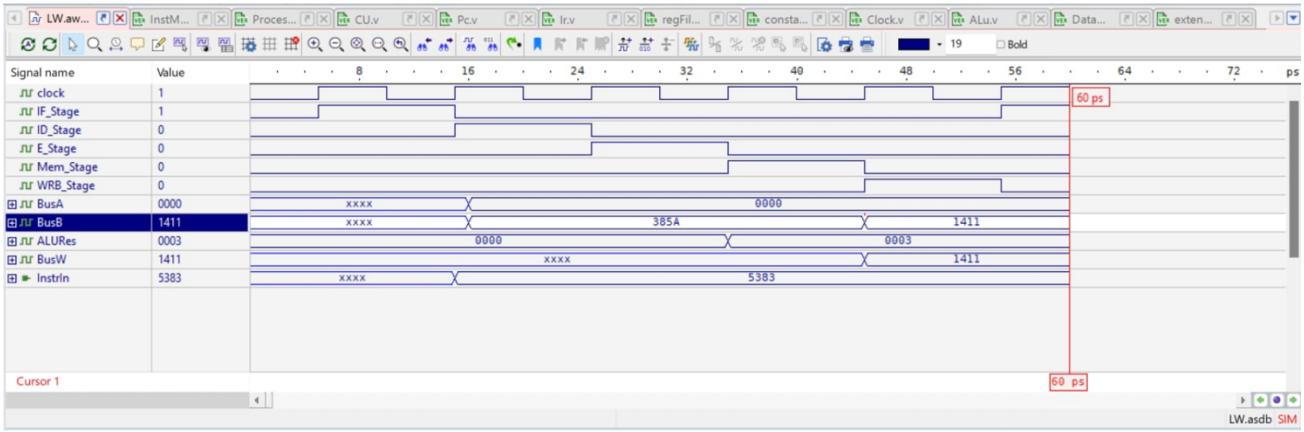


Figure 10 Waveform Load instruction

The value 1411 loaded into R3 confirms that the LW instruction executed correctly, fetching the data from the expected memory location.

The simulation results indicate that the processor correctly executes the test program, with proper handling of instruction fetch, decode, execution, memory access, and write-back stages. The observed values in the waveforms match the expected results, demonstrating the correct functionality of the designed RISC processor.

5.2 Test Program 2

The memory was initialized with the following values for testing purposes:

```
{mem[1], mem[0]}  = {4'b0001, 3'b111, 3'b011, 3'b010, 3'b000};
```

Figure 11 Instruction Test case 2

(AND R7, R3, R2)

Adds the values in registers R3 and R2, and stores the result in register R7.

Expected Output: R7 should hold the sum of the values in R3 and R2

The register file was initialized with the following values:

```
initial begin
    regArray[0] = 16'h0000; // R0 is zero
    regArray[1] = 16'h0000;
    regArray[2] = 16'h5;
    regArray[3] = 16'h385A;
    regArray[4] = 16'h0000;
    regArray[5] = 16'h0001;
    regArray[6] = 16'h2738;
    regArray[7] = 16'h0F5A;
end
```

Figure 12 Initial register file

The following image shows the initialization of the register file with the test program:

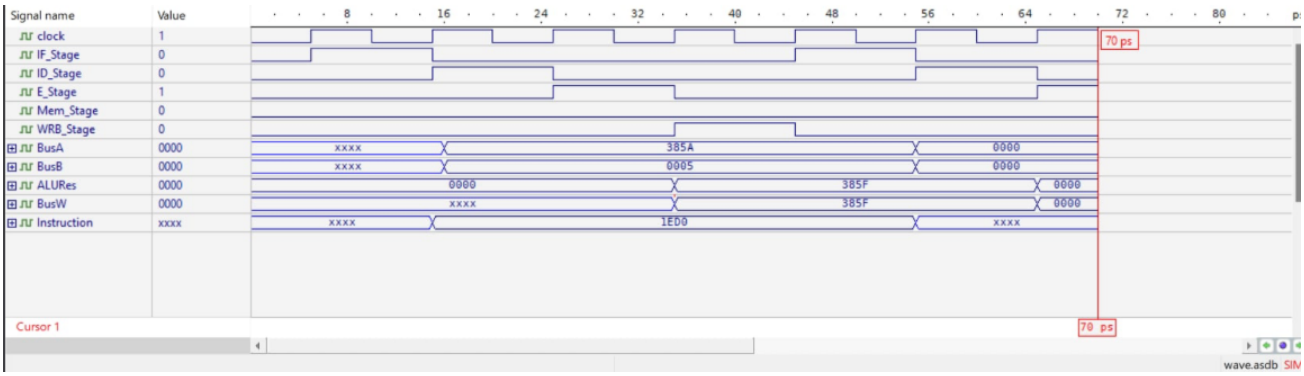


Figure 13 Waveform ADD instruction

The ADD instruction executed correctly, adding the values from R3 and R2 and storing the result in R7

5.3 Test Program 3

The memory was initialized with the following values for testing purposes:

```
{mem[1], mem[0]} = {4'b0100,1'b0, 3'b101, 3'b110, 5'b01010};
```

Figure 14 Instruction Test case 3

Description: Load a value from memory into a register.

Expected Output: The value from memory at the address computed as the sum of register contents and an immediate value should be loaded into the destination register.

The register file was initialized with the following values:

```
initial begin
    regArray[0] = 16'h0000; // R0 is zero
    regArray[1] = 16'h0000;
    regArray[2] = 16'h5;
    regArray[3] = 16'h385A;
    regArray[4] = 16'h0000;
    regArray[5] = 16'h0001;
    regArray[6] = 16'h2738;
    regArray[7] = 16'h0F5A;
end
```

Figure 15 Initial register file testcase

The following image shows the waveform diagram for the test program:

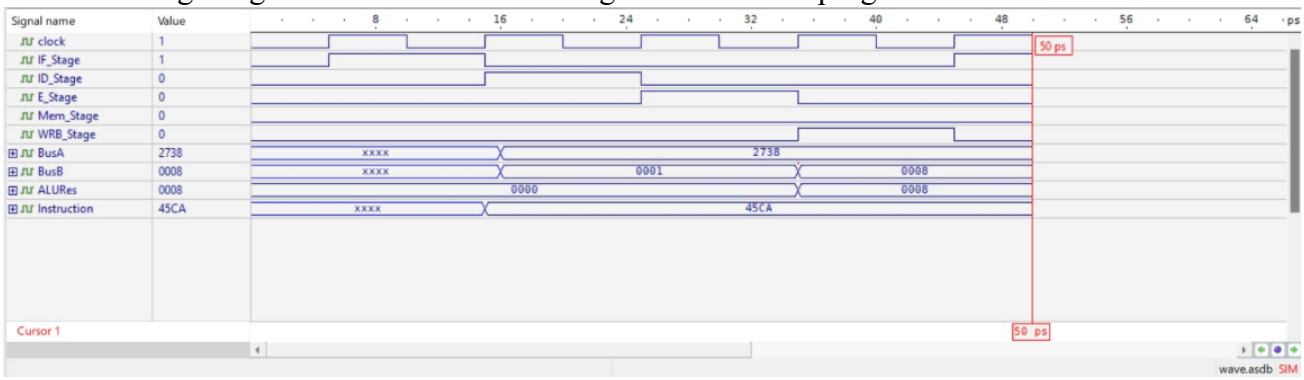


Figure 16 Waveform LW instruction

The load instruction executed correctly, completing in 4 stages as expected. The value from the memory cell was loaded into the destination register as intended. Specifically, the value from memory address R6 + 0xA was correctly loaded into register R5.

5.4 Test Program 4

The memory was initialized with the following values for testing purposes:

```
{mem[1],  mem[0]} = {4'b0110, 1'b0, 3'b011, 3'b100, 5'b00010};
```

Figure 17 Instruction Test case 4

(LBu R3, R4, 2)

Loads a byte from the memory address R4 + 2 into register R3 with zero extension.
Expected Output: R3 should hold the zero-extended byte from memory address R4 + 2.

The memory was initialized with the following values for testing purposes:

```
initial begin
    // Store initial values in the memory for testing purposess
    {memory[1],memory[0]} = 16'h1210;
    {memory[3],memory[2]} = 16'h1110;
    {memory[5],memory[4]} = 16'h1014;

end
```

Figure 18 Initial Memory Test case 4

The following image shows the waveform diagram for the test program:

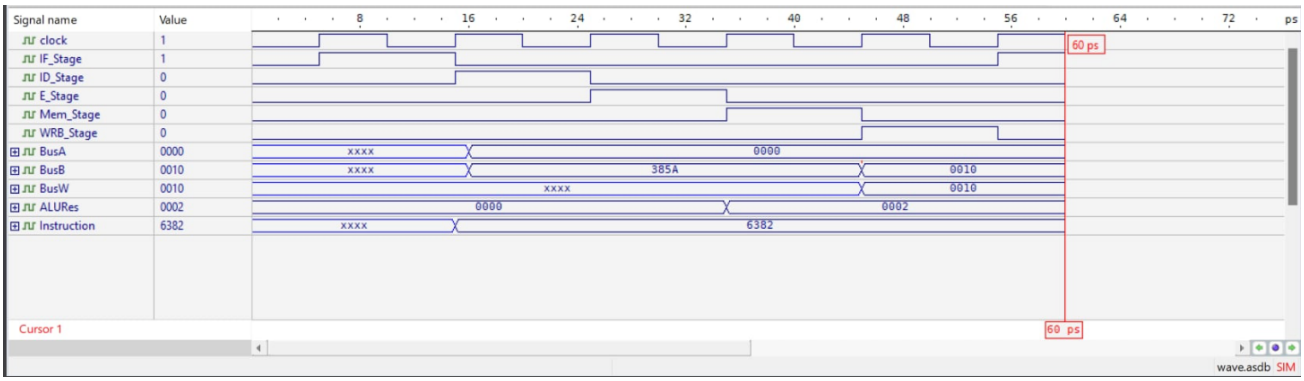


Figure 19 Waveform LBu instruction

The LBu instruction executed correctly, loading the byte from memory address R4 + 2 into R3 with zero extension.

5.5 Test Program 5

The memory was initialized with the following values for testing purposes:

```
{mem[1],  mem[0]}  = {4'b0111, 1'b0, 3'b011, 3'b100, 5'b00011};
```

Figure 20 Instruction Test case 5

(SW R3, R4, 3)

Stores the value in register R3 into the mem ory address R4 + 3.
Expected Output: The memory at address R4 + 3 should hold the value from register R3.

The register file was initialized with the following values:

```
initial begin
    regArray[0] = 16'h0000; // R0 is zero
    regArray[1] = 16'h0000;
    regArray[2] = 16'h5;
    regArray[3] = 16'h385A;
    regArray[4] = 16'h0000;
    regArray[5] = 16'h0001;
    regArray[6] = 16'h2738;
    regArray[7] = 16'h0F5A;
end
```

Figure 21 Register file test case 7

The following image shows the waveform diagram for the test program:

Figure 22 waveform SW instruction

The SW instruction executed correctly, storing the value 385A from R3 into the memory address R4 + 3. The operation completed in four stages as expected, ensuring that the value was stored correctly in the desired memory cell.

5.6 Test Program 6

The memory was initialized with the following values for testing purposes:

```
{mem[1],  mem[0]}= {4'b1111, 3'b010,1'b0 ,8'b11111111};
```

Figure 23 Instruction Test case 6

(SV R2, Imm)
Stores the immediate value 0xFF in the special register SV[2].
Expected Output: The special register SV[2] should hold the value 0xFF.

The register file was initialized with the following values:

```
initial begin
    regArray[0] = 16'h0000; // R0 is zero
    regArray[1] = 16'h0000;
    regArray[2] = 16'h5;
    regArray[3] = 16'h385A;
    regArray[4] = 16'h0000;
    regArray[5] = 16'h0001;
    regArray[6] = 16'h2738;
    regArray[7] = 16'h0F5A;
end
```

Figure 24 register file testcase 6

The following image shows the waveform diagram for the test program:

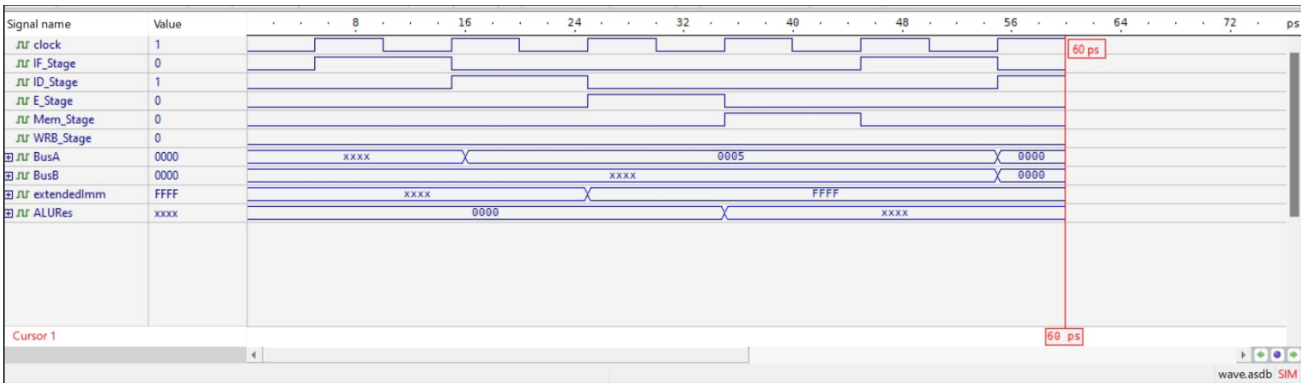


Figure 25 waveform sv instruction

The SV instruction executed correctly, storing the immediate value 0xFF in the special register SV[2].

5.7 Test Program 7

The memory was initialized with the following values for testing purposes:

```
{mem[1], mem[0]} = {4'b1100, 12'b000000000100};
```

Figure 26 Instruction Test case 7

(JUMP 4)
Description: Jumps to the address specified by the immediate value 4.
Expected Output: The next instruction to be fetched should be at the address PC + Jump Offset.

The following image shows the waveform diagram for the test program:

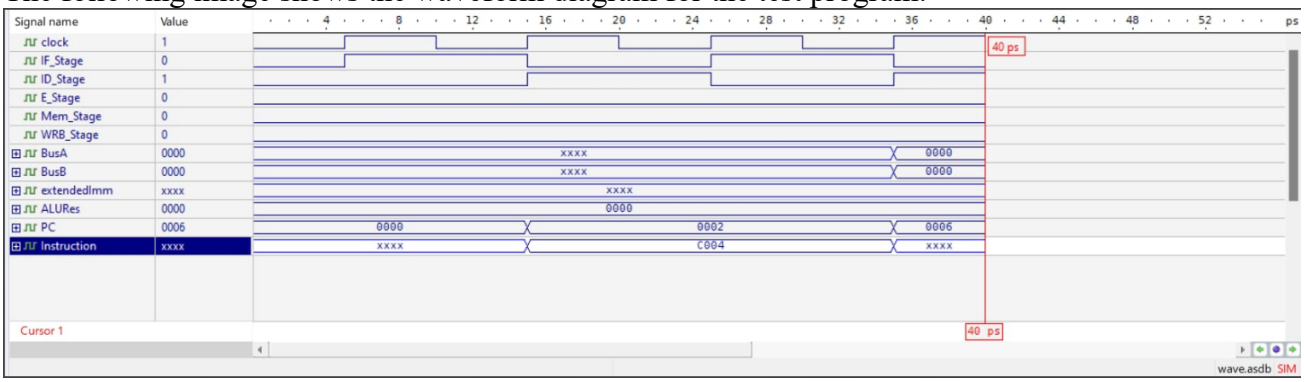


Figure 27 waveform JUMP instruction

The JUMP instruction executed correctly, completing in 2 stages as expected. After finishing, the next address is the jump target address. The jump offset after concatenation is 4, and the instruction fetched is the jump offset plus the current PC, as expected.

5.8 Test Program 8

The memory was initialized with the following values for testing purposes:

```
{mem[1], mem[0]} = {4'b1101, 12'b000000000100};
```

Figure 28 Instruction Test case 8

(CALL 4)

Description: Calls a subroutine at the address specified by the immediate value 4.

Expected Output: The next instruction to be fetched should be at the address PC + Jump Offset, and the current PC should be saved to R7 for return.

The following image shows the waveform diagram for the test program:

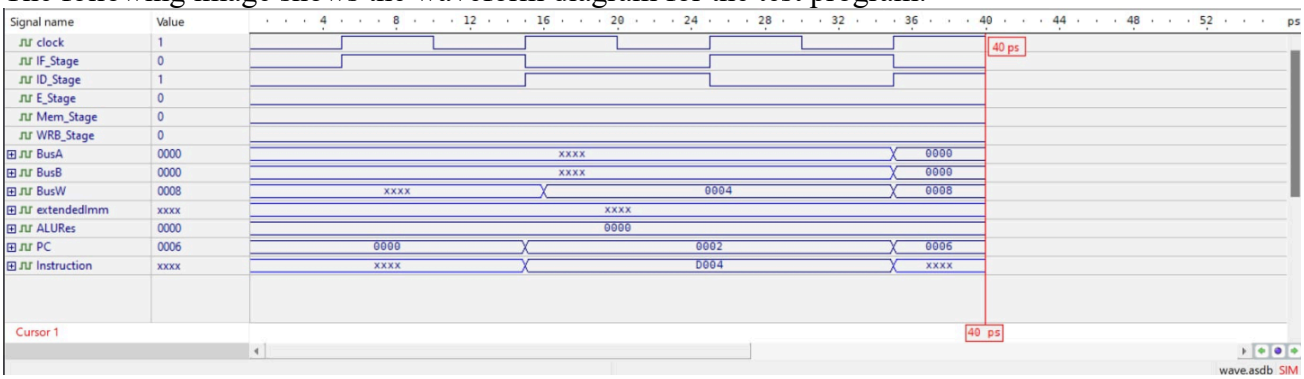


Figure 29 waveform CALL instruction

The CALL instruction executed correctly, completing in 2 stages as expected. After finishing, the next address is the jump target address, and the current PC is saved to R7. The jump offset after concatenation is 4, and the instruction fetched is the jump offset plus the current PC, as expected.

5.9 Test Program 9

The memory was initialized with the following values for testing purposes:

```
{mem[1],  mem[0]} = {4'b1000, 1'b0, 3'b010, 3'b001, 5'b00010};
```

Figure 30 Instruction Test case 9

(BGT R2, R1, 2)

Description: Branch if the value in register R2 is greater than the value in register R1, with a branch offset of 2.

Expected Output: If the condition is met, the next instruction to be fetched should be at the address PC + Branch Offset.

The following image shows the waveform diagram for the test program:

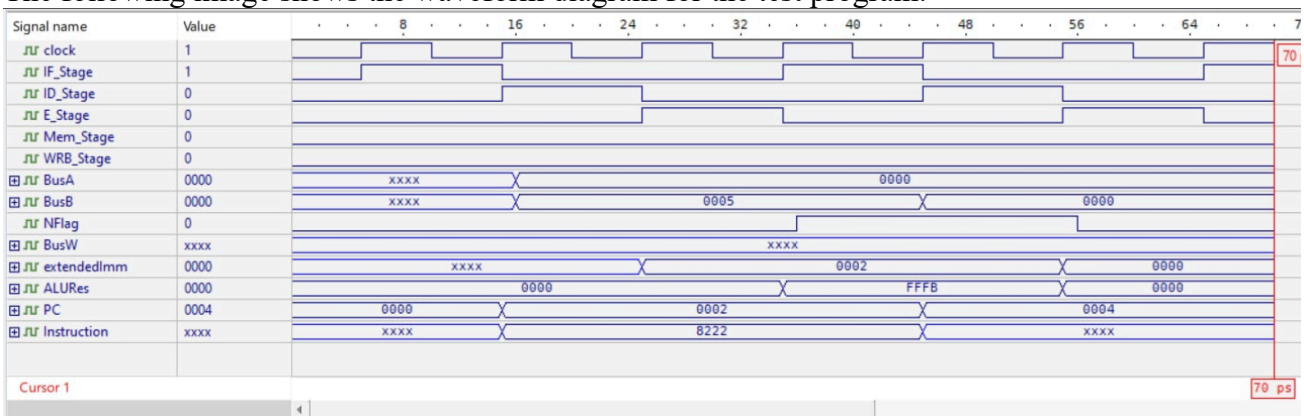


Figure 31 waveform BGT instruction

The BGT instruction executed correctly, completing in 2 stages as expected. After finishing, the next address is the branch target address since the condition was met (R2 > R1) , And the negative flag is set to one. The branch offset after concatenation is 2 as expected (PC+2+Brach Target address), and the instruction fetched is the branch offset plus the current PC, as expected.

5.10 Test Program 10

The memory was initialized with the following values for testing purposes:

```
{mem[1], mem[0]} = {4'b1110, 12'b000000000010}  ;
```

Figure 32 Instruction Test case 10

(RET)

Description: Return from subroutine by setting the PC to the value stored in register R7.

Expected Output: The next instruction to be fetched should be at the address stored in R7.

The register file was initialized with the following values:

```
// Initial values for registers for testing purposes
initial begin
    regArray[0] = 16'h0000; // R0 is zero
    regArray[1] = 16'h0000;
    regArray[2] = 16'h5;
    regArray[3] = 16'h385A;
    regArray[4] = 16'h0000;
    regArray[5] = 16'h0001;
    regArray[6] = 16'h2738;
    regArray[7] = 16'h000a;
end
```

Figure 33 register file test case 10

The following image shows the waveform diagram for the test program:

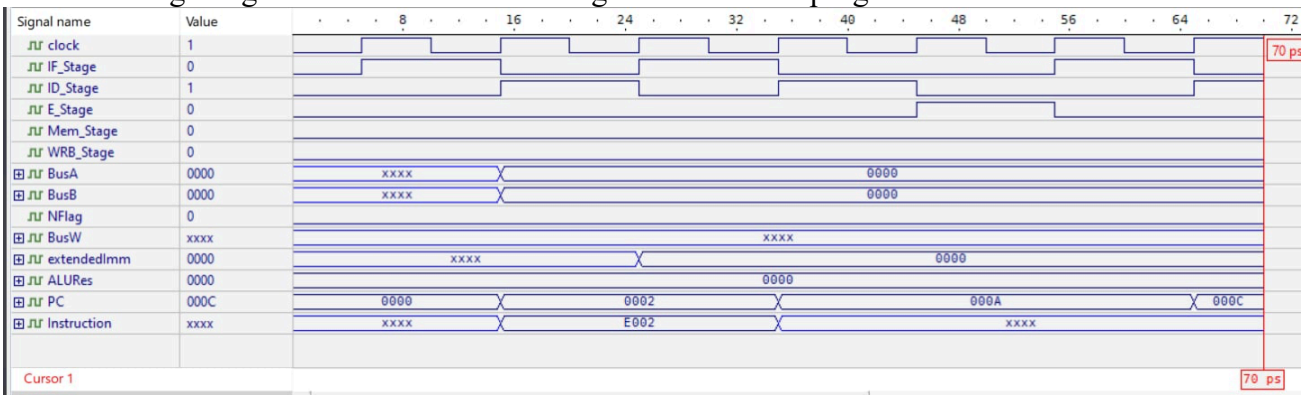


Figure 34 Waveform RET instruction

The RET instruction executed correctly, completing in 2 stages as expected. After finishing, the next address is the value stored in R7, which is 0x000A. The next instruction to be fetched corresponds to this address, as expected.