Faculty of Engineering and Technology
Electrical and Computer Engineering Department

Computer Architecture
ENCS4370

Project 2 Report

Prepared by:

Osaid Nur	1210733
Moath Wajeeh	1210125
Obayda Sarraj	1211128

Instructor: Dr. Aziz Qaroush
Section: 3
Date: 19/6/2024

Abstract

This project focuses on the design and verification of a simple multicycle RISC processor with a 16-bit instruction and word size. The processor architecture includes eight 16-bit general-purpose registers and separate instruction and data memories. It supports various instructions, including arithmetic, branch, and jump operations. The report details the architectural design, implementation, and verification processes, emphasizing the Datapath and control signals. This approach demonstrates the successful creation of a functional multicycle RISC processor.

Table of Contents

Table of Figures:

List of Tables:

1. Design Specifications

1.1 Overview

This project implements a MIPS architecture processor within the RISC architecture family. The processor supports a defined set of instructions and includes the following features:

1- Each instruction in the set is 32 bits in length.
2- The design includes 16 32-bit general-purpose registers, labeled from R0 to R15.
3- Program Counter (PC) is a 32-bit special-purpose register.
4- stack pointer (SP) is 32-bit special purpose register.
5- four distinct instruction types – R-type, I-type, J-type, and S-type.
6- The processor is designed with two separate physical memory units: one for storing instructions and the other for data.
7- Arithmetic Logic Unit is also used to decide the outcomes of certain instructions, like whether to take a branch or not.

1.2 Instruction Formats:

The instruction set has the following instructions formats:

1.2.1 R-Type:

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

Figure 1. R-type instruction format

Opcode (4 bits): specifies the operation type for the processor.
Rd (3 bits): Destination register (the result will be stored in this register)
Rs1 (3 bits): first source register
Rs2 (3 bits): second source register
Unused (3 bits)

1.2.2 I-Type:

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

Figure 2. I-type instruction format

Opcode (4 bits): specifies the operation type for the processor.
Rd (3 bits): Destination register (the result will be stored in this register)
Rs1 (3 bits): first source register
Immediate (5 bits): unsigned for logic instructions and signed otherwise
m (1 bit): this is for **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:

This type includes two instruction formats. The following is the instruction format for **Jump** and **call** instructions :

Opcode^{4}	Jump Offset^{12}

Figure 3. J-Type instruction format for jump and call instructions

Opcode (4 bits): specifies the operation type for the processor.
Jump Offset (12 bits): Specifies the offset relative to the current program counter (PC) value. the target address is calculated by concatenating the most significant 7-bit of the current PC with the 12-bit offset after multiplying offset by 2.
New PC = Current PC + 2* jump offset

This represents the instruction format for **ret**:

Opcode^{4}	Unused^{12}

Figure 4. J-Type instruction format for "ret" instruction

Opcode (4 bits): specifies the operation type for the processor.
Unused (12 bits): because in the 'ret' instruction, only the value in R7 is moved to the PC so these bits are not needed.

1.2.4 S-Type:

This format support only one instruction as following:

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

Opcode^{4}	Rs^{3}	Immediate^{8}

Figure 5. S-Type instruction format

1.3 Instruction Set:

No.	Instruction	Format	Meaning	opcode	m bit
1	AND	R-Type	Reg(Rd) = Reg(Rs1) & Reg(Rs2)	0000	
2	ADD	R-Type	Reg(Rd) = Reg(Rs1) + Reg(Rs2)	0001	
3	SUB	R-Type	Reg(Rd) = Reg(Rs1) - Reg(Rs2)	0010	
4	ADDI	I-Type	Reg(Rd) = Reg(Rs1) + Imm	0011	
5	ANDI	I-Type	Reg(Rd) = Reg(Rs1) + Imm	0100	
6	LW	I-Type	Reg(Rd) = Mem(Reg(Rs1) + Imm)	0101	
7	LBu	I-Type	Reg(Rd) = Mem(Reg(Rs1) + Imm)	0110	0
8	LBs	I-Type	Reg(Rd) = Mem(Reg(Rs1) + Imm)	0110	1
9	SW	I-Type	Mem(Reg(Rs1) + Imm) = Reg(Rd)	0111	

10	BGT	I-Type	if (Reg(Rd) > Reg(Rs1))
			Next PC = PC + sign_extended (Imm)
			else PC = PC + 2
				1000	0

11	BGTZ	I-Type	if (Reg(Rd) > Reg(0))
			Next PC = PC + sign_extended (Imm)
			else PC = PC + 2
				1000	1

12	BLT	I-Type	if (Reg(Rd) < Reg(Rs1))
			Next PC = PC + sign_extended (Imm)
			else PC = PC + 2
				1001	0

13	BLTZ	I-Type	if (Reg(Rd) < Reg(R0))
			Next PC = PC + sign_extended (Imm)
			else PC = PC + 2
				1001	1

14	BEQ	I-Type	if (Reg(Rd) == Reg(Rs1))
			Next PC = PC + sign_extended (Imm)
			else PC = PC + 2
				1010	0

15	BEQZ	I-Type	if (Reg(Rd) == Reg(R0))
			Next PC = PC + sign_extended (Imm)
			else PC = PC + 2
				1010	1

16	BNE	I-Type	if (Reg(Rd) != Reg(Rs1))
			Next PC = PC + sign_extended (Imm)
			else PC = PC + 2
				1011	0

17	BNEZ	I-Type	if (Reg(Rd) != Reg(Rs1))
			Next PC = PC + sign_extended (Imm)
			else PC = PC + 2
				1011	1

18	JMP	J-Type	Next PC = {PC[15:10], Immediate}	1100	
19	CALL	J-Type	Next PC = {PC[15:10], Immediate} PC + 4 is saved on r15	1101	
20	RET	J-Type	Next PC = r7	1110	
21	Sv	S-Type	M[rs] = imm	1111	

2. Components of Datapath

To implement the processor design, several key components are required. We will now provide a detailed description of these essential components.

2.1 Multiplexer:

A multiplexer, often abbreviated as MUX, is a combinational circuit that selects one of many input signals and forwards the selected input to a single output line. The selection of the input line is controlled by a set of selection lines.

In the Datapath, we used various MUXes. This is an example of a MUX used in the Datapath.

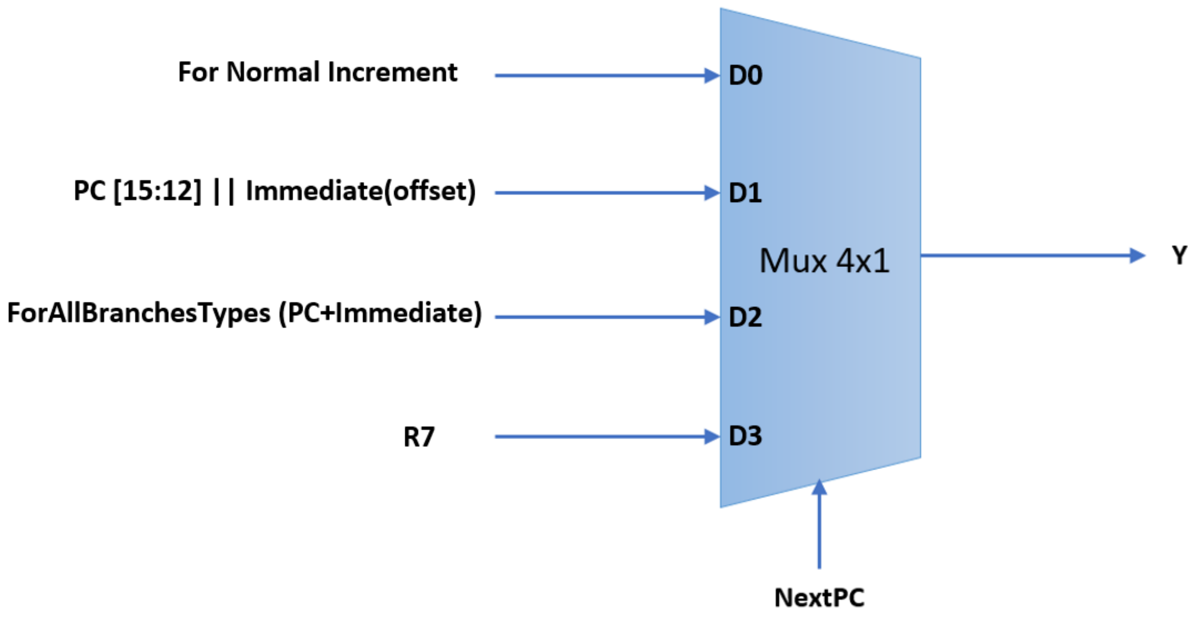

For Normal Increment
PC [15:12] || Immediate(offset)
ForAllBranchesTypes (PC+Immediate)
R7
D0
D1
D2
D3
Mux 4x1
Y
NextPC

Figure 6. Mux 4x1 used in Datapath

This MUX, located at the leftmost part of the Datapath, determines the PC value based on the instruction:

If (**NextPC = 00**) , the new PC value is PC + 2, which is used in most instructions

If (**NextPC = 01**) , The new PC value is the result of concatenating the current PC value with Immediate. This is used in jump instructions.

If (**NextPC = 10**) , The new value of PC is equal to old PC value + Immediate , This used in the branches instructions .

If (**NextPC = 11**), the new PC value is the value stored in R7, which contains the saved address after a function call. This is used in the "ret" instruction.

2.2 Instruction Memory

Instruction memory is a vital component of a computer's architecture. It stores the instructions that the processor needs to execute. In our design, the instruction memory is separate from the data memory, allowing for more efficient access and execution of instructions. This memory is read-only during program execution, ensuring the instructions remain unchanged. When the processor fetches an instruction, it uses the program counter (PC) to access the correct memory location.

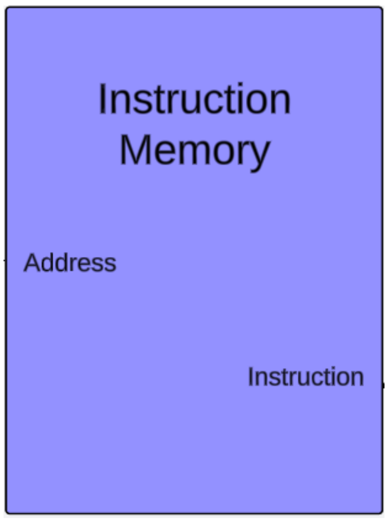


Figure 7. Instruction memory

As shown, the instruction memory has one input, the address, and one output, the 16-bit instruction stored at that address.

2.3 Data Memory

The other part of memory in our design is Data memory. It stores the data that the processor needs to read from or write to during program execution . This memory is byte-addressable, allowing each byte to have a unique address for efficient access. The processor interacts with data memory through addresses, enabling it to read or write 16-bit data values. Data memory is used for storing variables, intermediate results, and other data required by the program. In the design, it’s important to ensure that data memory operations are synchronized with the processor to maintain data integrity and performance.

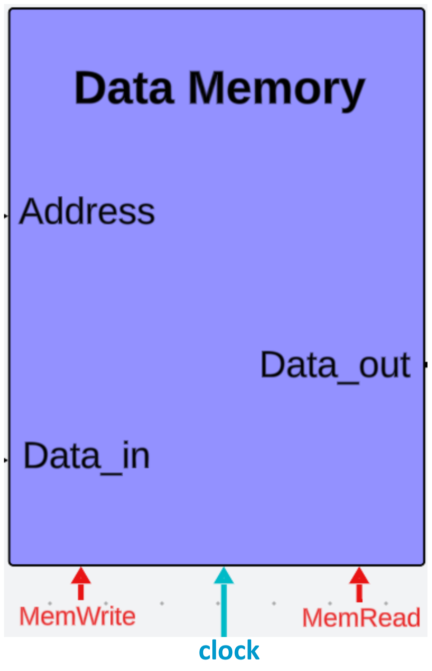


Figure 8. Data Memory

As shown in the above image , Data memory has the following inputs and outputs :

Input :

Address: This specifies the exact location in the memory where data will be stored or from where it will be retrieved. Each memory location has a unique address.

Data_in: This input carries the actual information or data that you want to store in the memory.

Clock: The clock signal synchronizes operations within the memory. It ensures that data is written to or read from the correct memory location at the appropriate time.

MemRead : enables the read operation in the memory .

MemWrite : enables the write operation in the memory .

Output:

Data_out: provides the data retrieved from the specified memory address, serving as the result of a read operation.

2.4 Register File

A register file is a critical component in computer processors designed to store temporary data for fast access. It consists of multiple registers, each functioning like a small storage unit with its unique address. These registers hold essential data required by the CPU for immediate processing tasks, including numbers, memory addresses, and control signals. Unlike dynamic RAM (DRAM) used in main memory, register files typically use static RAM (SRAM), enabling faster access speeds. In our project, we have 8 registers, each capable of storing data up to 16 bits wide.

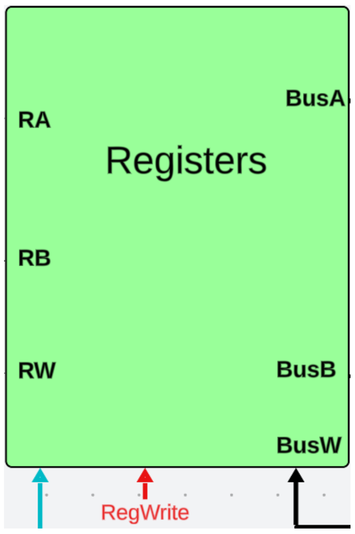


Figure 9. Register File

As shown , the Register file has the following inputs and outputs :

Input:

RA (Register Address A): Specifies the register address to read data from and send to BusA.
RB (Register Address B): Specifies the register address to read data from and send to BusB.
RW (Register Write): Specifies the register address to write data to from BusW.
RegWrite: Control signal to enable or disable writing to the register specified by RW.

Output :

BusA: carries the data read from the register specified by RA.
BusB: carries the data read from the register specified by RB.
BusW: carries data to be written into the register specified by RW if the RegWrite signal is active.

2.5 Arithmetic Logic Unit (ALU) :

The ALU, or Arithmetic Logic Unit, is a crucial part of the computer's central processing unit (CPU). It performs all arithmetic and logical operations. For example, it can add, subtract, multiply, and divide numbers. It also handles logical operations like AND, OR, and NOT. The ALU takes input data from the registers, processes it, and then sends the result back to the registers or memory. It operates under the control of the CPU's instruction set.

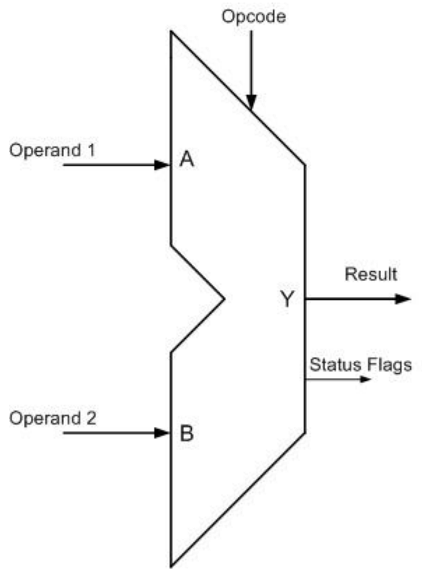


Figure 10. ALU

As shown , the ALU has the following inputs and outputs :

Inputs:

Two operands (A and B): These are 16-bit numbers.
Opcode: Determines the operation to be executed .

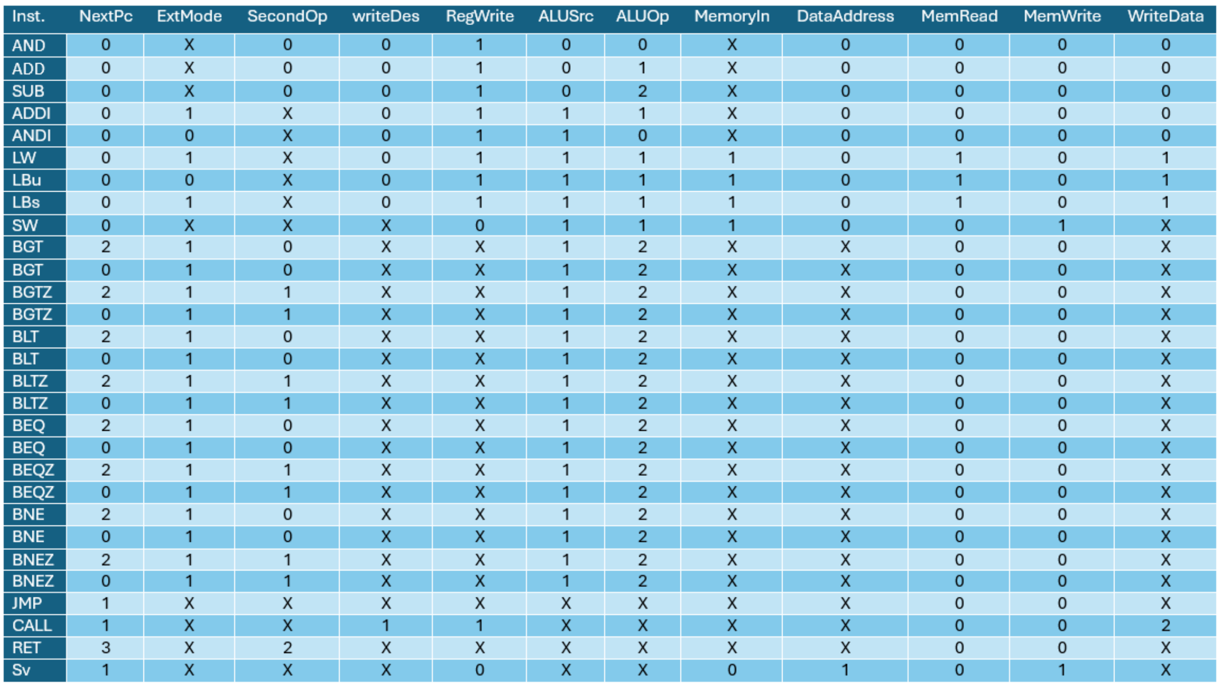
Outputs:

status flags: such as zero, carry, and overflow, which provide information about the outcome of the operation .
Result : This is where the ALU stores the outcome of the operation performed on the inputs A and B.

3. Control Path

3.1 Control Unit truth table

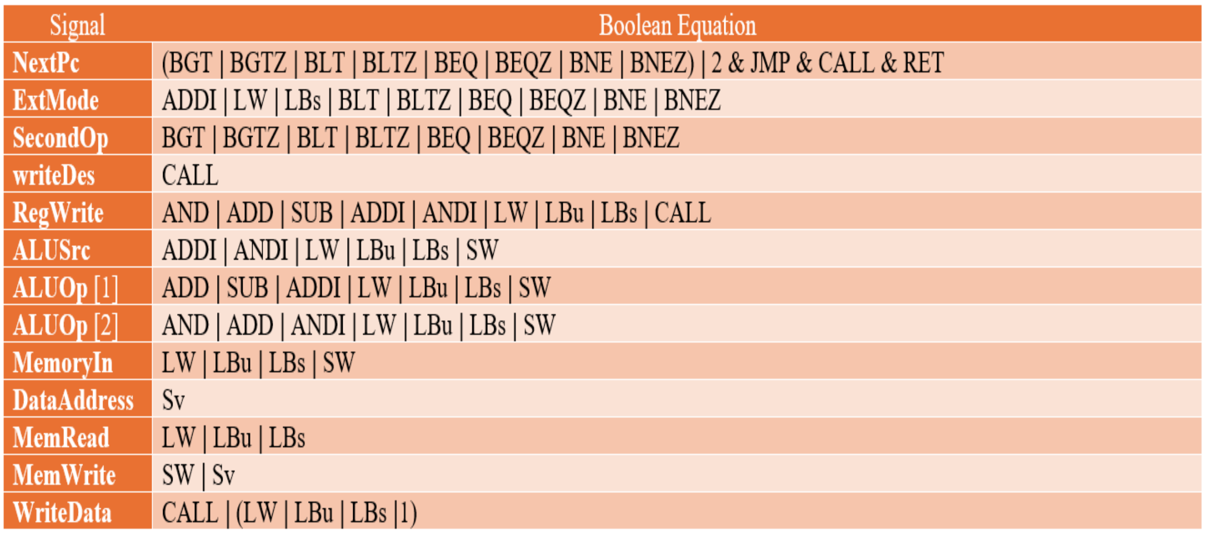
1 Control Signal Truth Table Table



Inst.	NextPc	ExtMode	SecondOp	writeDes	RegWrite	ALUSrc	ALUOp	MemoryIn	DataAddress	MemRead	MemWrite	WriteData
AND	0	X	0	0	1	0	0	X	0	0	0	0
ADD	0	X	0	0	1	0	1	X	0	0	0	0
SUB	0	X	0	0	1	0	2	X	0	0	0	0
ADDI	0	1	X	0	1	1	1	X	0	0	0	0
ANDI	0	0	X	0	1	1	0	X	0	0	0	0
LW	0	1	X	0	1	1	1	1	0	1	0	1
LBu	0	0	X	0	1	1	1	1	0	1	0	1
LBs	0	1	X	0	1	1	1	1	0	1	0	1
SW	0	X	X	X	0	1	1	1	0	0	1	X
BGT	2	1	0	X	X	1	2	X	X	0	0	X
BGT	0	1	0	X	X	1	2	X	X	0	0	X
BGTZ	2	1	1	X	X	1	2	X	X	0	0	X
BGTZ	0	1	1	X	X	1	2	X	X	0	0	X
BLT	2	1	0	X	X	1	2	X	X	0	0	X
BLT	0	1	0	X	X	1	2	X	X	0	0	X
BLTZ	2	1	1	X	X	1	2	X	X	0	0	X
BLTZ	0	1	1	X	X	1	2	X	X	0	0	X
BEQ	2	1	0	X	X	1	2	X	X	0	0	X
BEQ	0	1	0	X	X	1	2	X	X	0	0	X
BEQZ	2	1	1	X	X	1	2	X	X	0	0	X
BEQZ	0	1	1	X	X	1	2	X	X	0	0	X
BNE	2	1	0	X	X	1	2	X	X	0	0	X
BNE	0	1	0	X	X	1	2	X	X	0	0	X
BNEZ	2	1	1	X	X	1	2	X	X	0	0	X
BNEZ	0	1	1	X	X	1	2	X	X	0	0	X
JMP	1	X	X	X	X	X	X	X	X	0	0	X
CALL	1	X	X	1	1	X	X	X	X	0	0	2
RET	3	X	2	X	X	X	X	X	X	0	0	X
Sv	1	X	X	X	0	X	X	0	1	0	1	X

3.2 Logic equation of the control signals

2 Boolean Equation Table Table



Signal	Boolean Equation
NextPc	(BGT | BGTZ | BLT | BLTZ | BEQ | BEQZ | BNE | BNEZ) | 2 & JMP & CALL & RET
ExtMode	ADDI | LW | LBs | BLT | BLTZ | BEQ | BEQZ | BNE | BNEZ
SecondOp	BGT | BGTZ | BLT | BLTZ | BEQ | BEQZ | BNE | BNEZ
writeDes	CALL
RegWrite	AND | ADD | SUB | ADDI | ANDI | LW | LBu | LBs | CALL
ALUSrc	ADDI | ANDI | LW | LBu | LBs | SW
ALUOp [1]	ADD | SUB | ADDI | LW | LBu | LBs | SW
ALUOp [2]	AND | ADD | ANDI | LW | LBu | LBs | SW
MemoryIn	LW | LBu | LBs | SW
DataAddress	Sv
MemRead	LW | LBu | LBs
MemWrite	SW | Sv
WriteData	CALL | (LW | LBu | LBs |1)

4. State Diagram

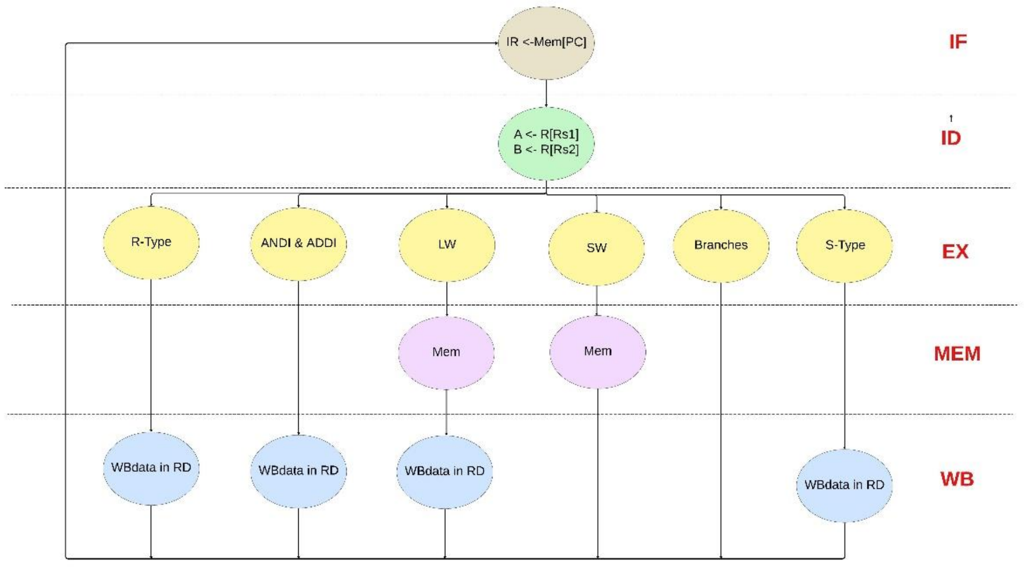
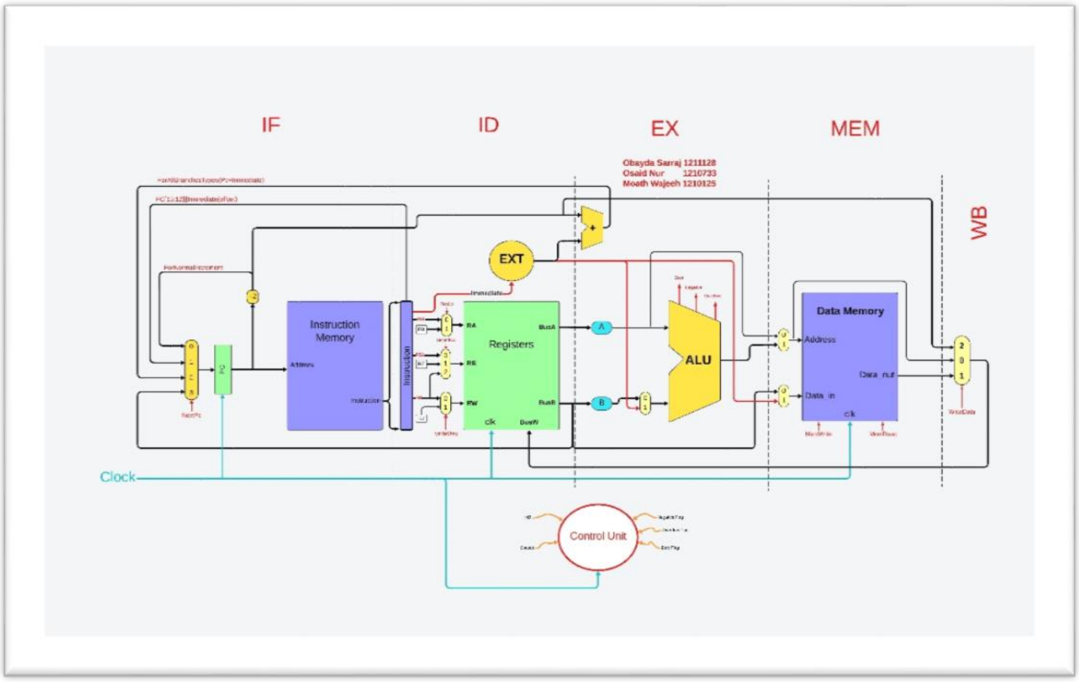
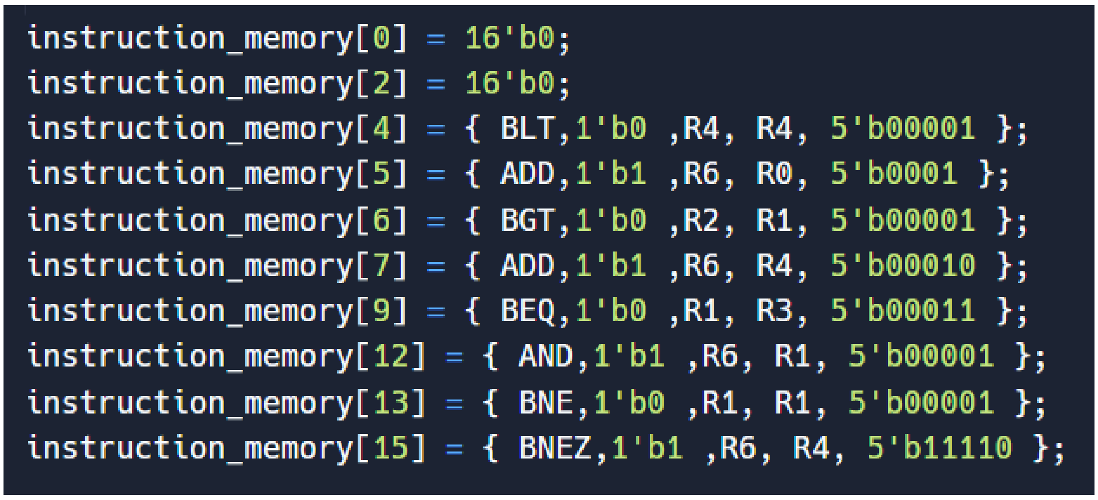


Figure 11: State Diagram

5. Project Data Path



6. Simulation and Testing results

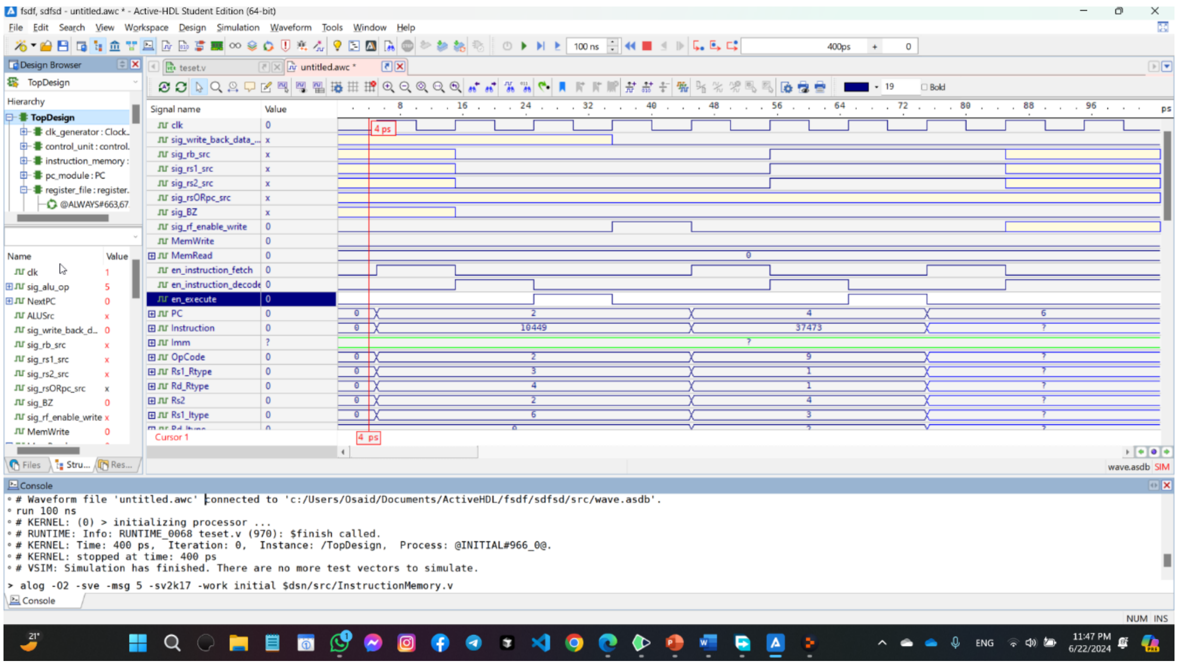


```
instruction_memory[0] = 16'b0;
instruction_memory[2] = 16'b0;
instruction_memory[4] = { BLT,1'b0 ,R4, R4, 5'b00001 };
instruction_memory[5] = { ADD,1'b1 ,R6, R0, 5'b0001 };
instruction_memory[6] = { BGT,1'b0 ,R2, R1, 5'b00001 };
instruction_memory[7] = { ADD,1'b1 ,R6, R4, 5'b00010 };
instruction_memory[9] = { BEQ,1'b0 ,R1, R3, 5'b00011 };
instruction_memory[12] = { AND,1'b1 ,R6, R1, 5'b00001 };
instruction_memory[13] = { BNE,1'b0 ,R1, R1, 5'b00001 };
instruction_memory[15] = { BNEZ,1'b1 ,R6, R4, 5'b11110 };
```

Figure 12: Instructions set

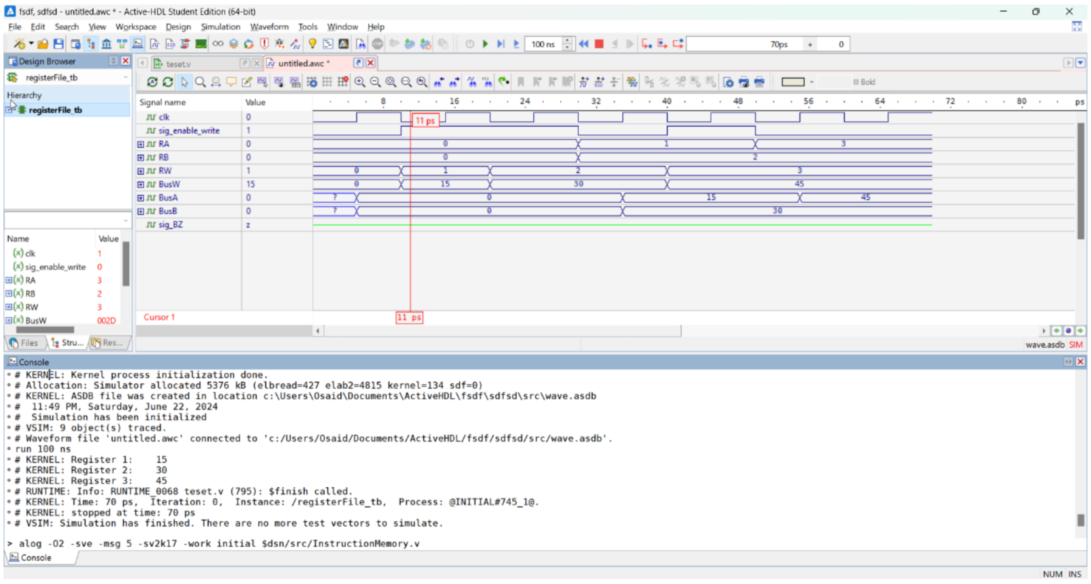
These instructions that we have tested our program with.

Also, we have attached some testbenches for Data memory and register file, we tested them also.



```
# Waveform file 'untitled.awc' connected to 'c:/Users/Osaid/Documents/ActiveHDL/fsdf/sdfsd/src/wave.asdb'.
run 100 ns
# KERNEL: (0) > initializing processor ...
# RUNTIME: Info: RUNTIME_0068 teset.v (970): $finish called.
# KERNEL: Time: 400 ps,  Iteration: 0,  Instance: /TopDesign,  Process: @INITIAL#966_0@.
# KERNEL: stopped at time: 400 ps
# VSIM: Simulation has finished. There are no more test vectors to simulate.
> alog -O2 -sve -msg 5 -sv2k17 -work initial $dsn/src/InstructionMemory.v
```

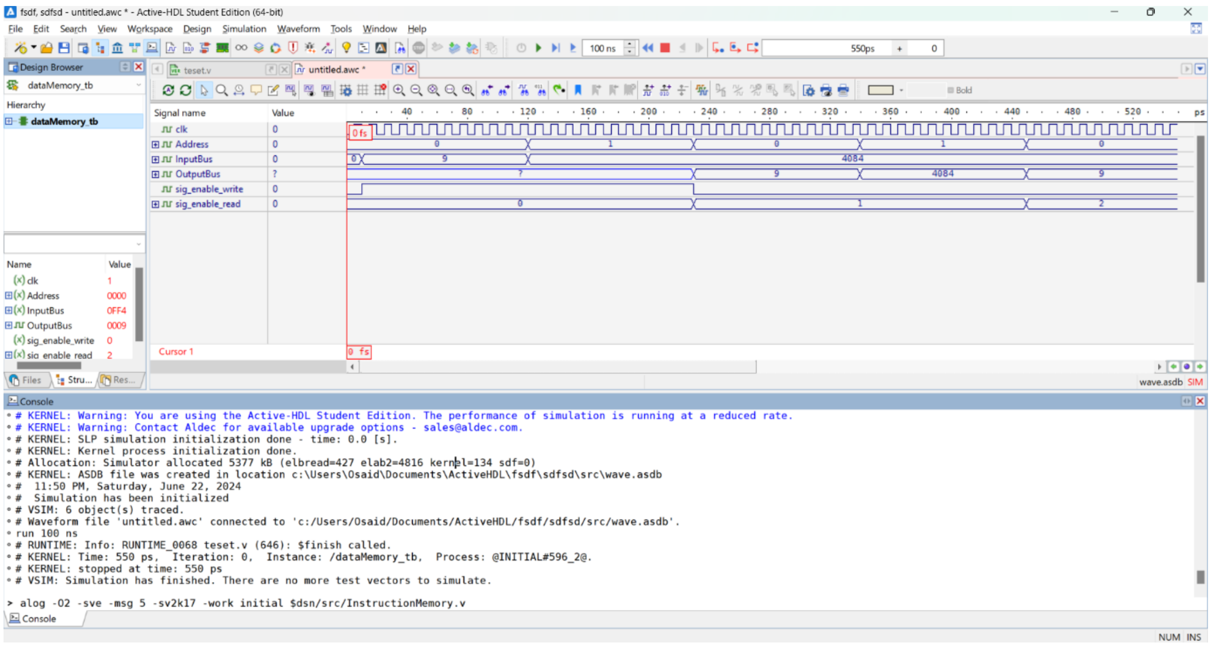
Figure 13: wave form



```
# KERNEL: Kernel process initialization done.
# Allocation: Simulator allocated 5376 kB (elbread=427 elab2=4815 kernel=134 sdf=0)
# KERNEL: ASDB file was created in location c:\Users\Osaid\Documents\ActiveHDL\fsdf\sdfsd\src\wave.asdb
#  11:49 PM, Saturday, June 22, 2024
#  Simulation has been initialized
# VSIM: 9 object(s) traced.
# Waveform file 'untitled.awc' connected to 'c:/Users/Osaid/Documents/ActiveHDL/fsdf/sdfsd/src/wave.asdb'.
run 100 ns
# KERNEL: Register 1:    15
# KERNEL: Register 2:    30
# KERNEL: Register 3:    45
# RUNTIME: Info: RUNTIME_0068 teset.v (795): $finish called.
# KERNEL: Time: 70 ps,  Iteration: 0,  Instance: /registerFile_tb,  Process: @INITIAL#745_1@.
# KERNEL: stopped at time: 70 ps
# VSIM: Simulation has finished. There are no more test vectors to simulate.

> alog -O2 -sve -msg 5 -sv2k17 -work initial $dsn/src/InstructionMemory.v
```

Figure 14: Another testbench waveform



```
# KERNEL: Warning: You are using the Active-HDL Student Edition. The performance of simulation is running at a reduced rate.
# KERNEL: Warning: Contact Aldec for available upgrade options - sales@aldec.com.
# KERNEL: SLP simulation initialization done - time: 0.0 [s].
# KERNEL: Kernel process initialization done.
# Allocation: Simulator allocated 5377 kB (elbread=427 elab2=4816 kernel=134 sdf=0)
# KERNEL: ASDB file was created in location c:\Users\Osaid\Documents\ActiveHDL\fsdf\sdfsd\src\wave.asdb
#  11:50 PM, Saturday, June 22, 2024
#  Simulation has been initialized
# VSIM: 6 object(s) traced.
# Waveform file 'untitled.awc' connected to 'c:/Users/Osaid/Documents/ActiveHDL/fsdf/sdfsd/src/wave.asdb'.
run 100 ns
# RUNTIME: Info: RUNTIME_0068 teset.v (646): $finish called.
# KERNEL: Time: 550 ps,  Iteration: 0,  Instance: /dataMemory_tb,  Process: @INITIAL#596_2@.
# KERNEL: stopped at time: 550 ps
# VSIM: Simulation has finished. There are no more test vectors to simulate.

> alog -O2 -sve -msg 5 -sv2k17 -work initial $dsn/src/InstructionMemory.v
```

Figure 15: another waveform

Conclusion:

Building and testing a multi-cycle RISC processor using Verilog has taught us a lot about how computers are built and how digital logic works. Using a multi-cycle model helped us use resources efficiently and make the control system simpler, while still handling a wide range of operations like moving data, jumping to different parts of code, and storing information. We tested the processor thoroughly to make sure it worked correctly with the instructions we wanted it to follow. This project showed how we can apply concepts from digital logic and computer organization in a Verilog-based environment, giving us a solid base for exploring more about processor design and hardware description languages. It also helped us understand the key parts and considerations needed to develop computer processors.