Faculty of Engineering & Technology

Electrical & Computer Engineering Department

COMPUTER ARCHITECTURE: ENCS4370

Project2: Report

Prepared by:

Mohammad Abu Shams	1200549
Abdalrahim Thiab	1202102
Nirmeen Shaikh	1200200

Instructor: Dr. Ayman Hroub

Section: 2

Date: 20-6-2023

Birzeit

Abstract

This project focuses on designing, implementing, and verifying a multi-cycle RISC processor using Verilog. The processor architecture is based on a 16-bit instruction and word size. There will be eight 16-bit general-purpose registers from R0 through R7; R0 will be hardwired to zero, and one 16-bit program counter (PC). Moreover, it will support four instruction formats: R-type, I-type, J-type, and S-type, each with its opcode and functionality. Multi-cycle design of instruction execution allows appropriate processor resources through pipelining and reuse of functional units like the ALU and memory between dichotomous instruction phases. The processor has separate instruction and data memories, byte-addressable, using little-endian byte ordering for its improved performance and easy memory access.

Verification was done through heavy simulations with test benches that ran a wide variety of code sequences to test the correct functionality of all instructions implemented in the ISA. A detailed datapath and control unit drives the multi-cycle execution, including the generation of the right ALU signals for conditional branches and the right flag generations for the same. Instructions in the ISA capture all the basic operations: arithmetic, logical, load/store, and branches—enough to write functional programs. Team members worked collaboratively, fully understanding and implementing the processor. The detailed description includes the datapath, control signals, block diagrams of the processor, and simulation results— evidence that the processor has been correctly implemented and is functional.

Table of Contents

Abstract ..... **I**

List of Figures ..... **III**

List of Tables ..... **IV**

1. Design and Implementation **1**

1.1 PC (Program Counter) **2**

1.2 Register File **3**

1.3 Data Memory **3**

1.4 ALU **4**

1.5 Extender **4**

1.6 Instruction memory **4**

1.7 Control Unit **4**

Boolean Equation **7**

2. Testing **8**

ALU: **8**

Extender8: **8**

Extender5: **8**

Register file **9**

Mux2x1: **9**

Mux3x1: **10**

Mux4x1: **10**

Data Memory: **11**

Instruction Memory **11**

Control Unit: **11**

PC selector **12**

Adder 16 bit: **12**

3. Multi-cycle implementation **13**

4. Team Work **19**

5. Conclusion **20**

List of Figures

Figure 1: Data Path 1
Figure 2: ALU test bench 8
Figure 3: Extender8 test bench 8
Figure 4: Extender5 test bench 8
Figure 5: register file test bench 9
Figure 6: Mux 2x1 test bench 9
Figure 7: Mux 3x1 test bench 10
Figure 8: Mux 4x1 test bench 10
Figure 9: data memory test bench 11
Figure 10: instruction memory test bench 11
Figure 11: control unit test bench 11
Figure 12: PC Selector test bench 12
Figure 13: Adder test bench 12
Figure 14: Instruction fetch 13
Figure 15: Instruction decode 14
Figure 16: Execute stage 14
Figure 17: Memory stage 15
Figure 18: Instructions 15
Figure 19: the instructions in the multi-cycle data path 16
Figure 20: Monitor values and print at each time step 16
Figure 21: Path test bench 18

List of Tables

Table 1: PC Control Truth Table 2
Table 2: Main Control Truth Table 5
Table 3: ALU Control Truth Table 6
Table 4: Main Control Signals 6
Table 5: Mux 2x1 truth table 9
Table 6: Mux 3x1 Truth table 10
Table 7: Mux 4x1 truth table 10

1. Design and Implementation

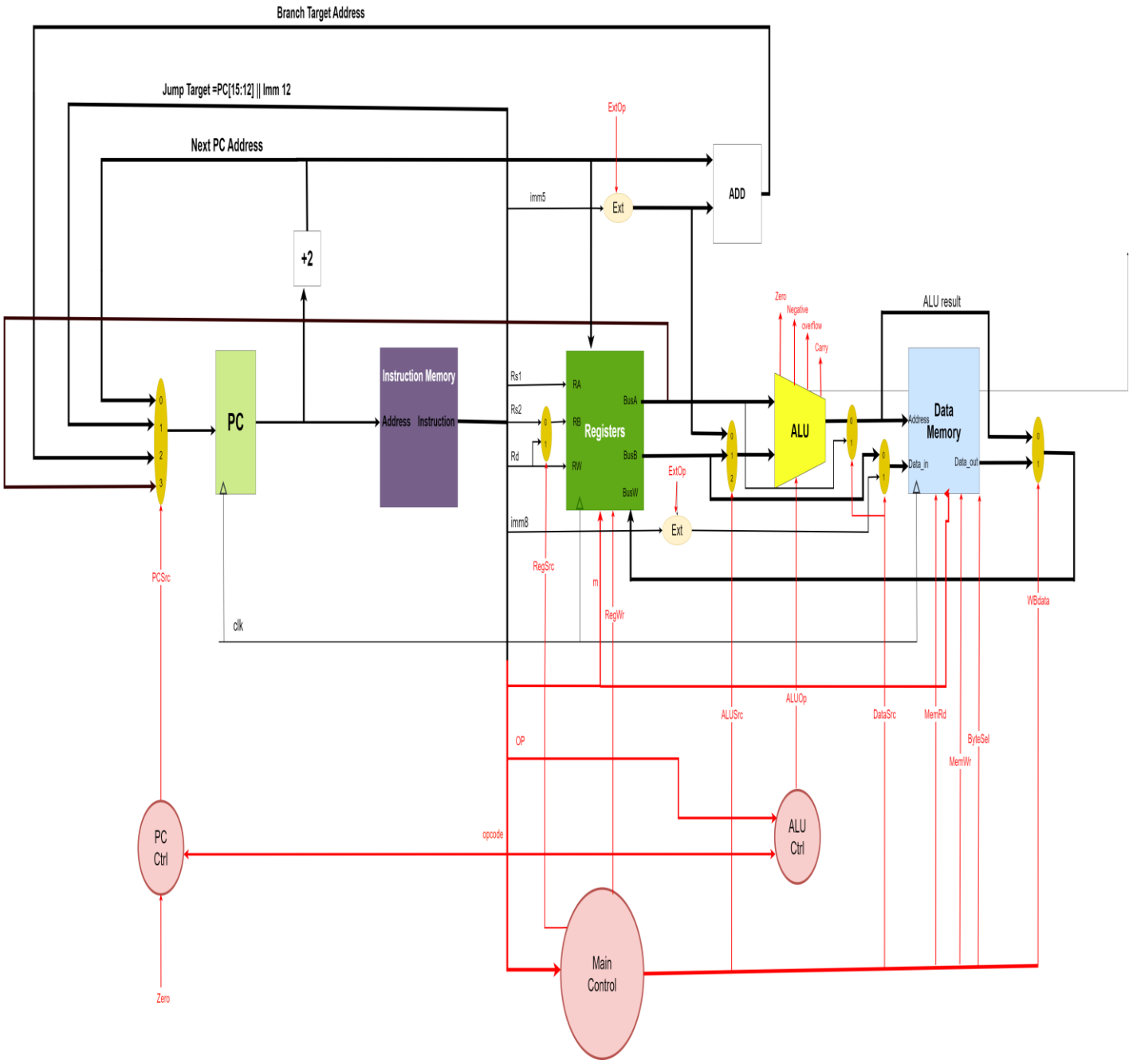


Figure 1: Data Path

1.1 PC (Program Counter)

The Program Counter (PC) in this RISC processor is essential for finding the memory address of the next instruction. It changes based on different types of operations: R-type, I-type, J-type, and S-type. Normally, the next PC value is PC + 2.

For branch instructions like BEQ, the PC updates to PC + sign_extended(imm) * 2 if the condition is true. This allows the program to jump to different parts conditionally. A multiplexer, controlled by the PCSrc signal, helps select the correct PC value based on the current operation.

In jump instructions like JMP, the PC changes to a new address made by combining the upper bits of the current PC with a 12-bit offset (imm12). For JAL (Jump and Link), it updates the PC similarly and also saves PC + 2 in register R7 as the return address.

This design uses a multiplexer with the PCSrc control signal to choose the right next PC address, ensuring smooth and accurate execution of instructions across different scenarios.

Table 1: PC Control Truth Table

PC Control Truth Table			
OP	**Type**	**Zero flag**	**PCsrc**
All-ops	R-Type	X	0 (PC+2)
All-ops	S-Type	X	0 (PC+2)
RET	J-Type	X	3 (PC = Reg[r7])
Else ops		X	1 (Jump Target)
1000 to 1011	I-Type	0	0 (PC+2)
1000 to 1011		1	2 (PC + Imm)
Else ops		X	0 (PC+2)

Note: Zero flag is set when compare Reg(Rd) & Reg(R0).

1000 to 1011: BGT, BGTZ, BLT, BLTZ, BEQ, BEQZ, BNE and, BNEZ.

1.2 Register File

The register file in this RISC processor is very important for quickly storing and accessing data during program execution. It uses three 3-bit inputs to choose which registers to read from or write to. These inputs are called Rs1, Rs2/Rd, and Rd.

Rs1 always selects a register to read from. Rs2/Rd can either select another register to read from or a register to write to, depending on what the instruction needs.

The third input, Rd, is only used to select the register where data will be written after an operation. The data to be written comes through the BusW input, which is a 16-bit data bus. The register file works with a clock signal and a RegWr signal. The RegWr signal allows writing when it is set to high.

The outputs from the register file are provided through two buses: BusA and BusB. BusA reads data from the register chosen by Rs1. BusB reads data from the register chosen by Rs2 or Rd. This setup allows other parts of the processor to use these data values for calculations and other operations.

Writing to the register file only happens if the RegWr signal is active. When RegWr is active, data from BusW is written to the register specified by Rd. The register R0 is special: it is always set to zero and cannot be written to. This ensures it always provides a zero value when accessed. This design helps the processor work efficiently and without errors.

1.3 Data Memory

The data memory in your RISC-V processor is an important storage unit that holds data at various memory locations. It takes inputs like MemRd (Memory Read) and MemWr (Memory Write) signals, a 16-bit memory address, and a 16-bit data input, and it outputs the data stored at the given address.

When the MemRd signal is active, the data memory reads and provides the data from the specified address. When the MemWr signal is active, it writes the incoming data to the specified memory address on the clock's rising edge, ensuring timely updates as the program runs.

1.4 ALU

The ALU (Arithmetic Logic Unit) in your RISC processor is responsible for executing arithmetic and logical operations. It receives two 16-bit operands and an ALUOp code that determines the specific operation to be performed. The unit outputs the operation result, a zero flag, a carry flag, and an overflow flag, supporting comprehensive status feedback.

For operational specifics, ALUOp codes such as 0000 and 0001 denote AND and ADD operations respectively. The codes 0010 and 0011 represent SUBTRACTION and SET ON LESS THAN operations, aligning with typical RISC operations. The processor further handles shifts and rotates through codes 0100 and 0101, for logical left and logical right shifts, enhancing the ALU's versatility in handling various data manipulations.

1.5 Extender

The Extender unit in your RISC processor is crucial for converting immediate values to a 16-bit format suitable for the ALU. It adjusts the immediate values based on their sign bit (the leftmost bit), ensuring that positive and negative values are correctly interpreted during computations.

1.6 Instruction memory

The Instruction Memory in your RISC processor is important for storing all executable instructions. It uses the output of the Program Counter to get the address of the next instruction. Then, it provides key information for the next steps in the datapath, such as the addresses of source and destination registers, the immediate value, the instruction type, and a flag that indicates the end of the program.

1.7 Control Unit

The Control Unit in your RISC processor manages the operations of the datapath. It decodes instructions and creates control signals based on each instruction type and function code. These signals ensure that parts like the ALU, register file, and memory units work correctly. The

Control Unit synchronizes everything, guiding the datapath through each step of instruction execution.

Table 2: Main Control Truth Table

Main Control Truth Table										
OP(input)		RegSrc	RegWr	ExtOp	ALUSrc	MemRd	MemWr	WBdata	DataSrc	ByteSel
R-Type	AND	0	1	X	1	0	0	0	0	X
	ADD	0	1	X	1	0	0	0	0	X
	SUB	0	1	X	1	0	0	0	0	X
I-Type	ADDI	X	1	1	0	0	0	0	0	X
	ANDI	X	1	1	0	0	0	0	0	X
	LW	X	1	1	0	1	0	1	0	X
	LB(u/s)	0	1	X:mC	0	1	0	1	0	1
	SW	X	0	1	0	0	1	0	0	X
	BGT/Z	1	0	1	1	0	0	X	0	X
	BLT/Z	1	0	1	1	0	0	X	0	X
	BEQ/Z	1	0	1	1	0	0	X	0	X
	BNE/Z	1	0	1	1	0	0	X	0	X
J-Type	JMP	X	0	X	X	0	0	X	X	X
	CALL	0	0	1	1	0	0	1	0	1
	RET	X	X	X	X	X	X	X	X	X
S-Type	SV	0	0	1	1	0	1	0	1	X

ExtOp -> 0: unsign // 1: sign

RegSrc -> 0: Rs2 // 1: Rd

In I-Type Main control don't care if we have Z or not in the OP name, but when we have Z it will be handled by instruction logic inside the Registers.

X:mC -> ExtOp is 0 or 1 handled by instruction logic depends on m value.

Table 3: ALU Control Truth Table

ALU Control Truth Table			
Type	OP	ALUOp	3-bit Coding
R-Type	AND	AND	000
	ADD	ADD	001
	SUB	SUB	010
I-Type	ADDI	ADD	001
	ANDI	AND	000
	LW	ADD	001
	LB(u/s)	ADD	001
	SW	ADD	001
	BGT/Z	SUB	010
	BLT/Z	SUB	010
	BEQ/Z	SUB	010
	BNE/Z	SUB	010
J-Type	JMP	X	X
	CALL	X	X
	RET	X	X
S-Type	Sv	X	X

Table 4: Main Control Signals

Signal	Effect when '0'	Effect when '1'
RegWr	No register is written.	Destination register (Rd) is written with the data on BusW.
ExtOp	14-bit immediate is zero-extended	14-bit immediate is sign-extended.
ALUSrc	Second ALU operand is the value of the extended 14-bit immediate.	Second ALU operand is the value of register (Rs2 or Rd) that appears on BusB.
MemRd	Data memory is NOT read.	Data memory is read Data_out ← Memory[address].
MemWr	Data Memory is NOT written.	Data memory is written Memory[address] ← Data_in.
WBdata	BusW = ALU result	BusW = Data_out from Memory
DataSrc	Data in = ext(immediate 8)	Data in = BusB
ByteSel	Load one word (2 bytes) from the data memory	Load one byte from the data memory

Boolean Equation

RegW = (AND + ADD + SUB + ADDI + LW + LBu/LBs)

MemRd = (LW + LBu/LBs)

MemWr = (SW + SV)

ALUSrc = (ADDI)

ALUOp00 = (AND)

ALUOp01 = (ADD + ADDI)

ALUOp10 = (SUB)

ALUOp11 = (BEQ)

PCSrc00 = (default when none of the below are true)

PCSrc01 = (JMP + CALL)

PCSrc10 = (BEQ when zero)

PCSrc11 = (RET)

ByteSel = (LBu/LBs)

ExtOp = (LBs)

WBdata = (LW + LBu/LBs)

Push = (CALL)

Pop = (RET)

DataSrc = (SV)

2. Testing

Testing is done to make sure our data path works correctly, as shown in the following figures.

ALU:

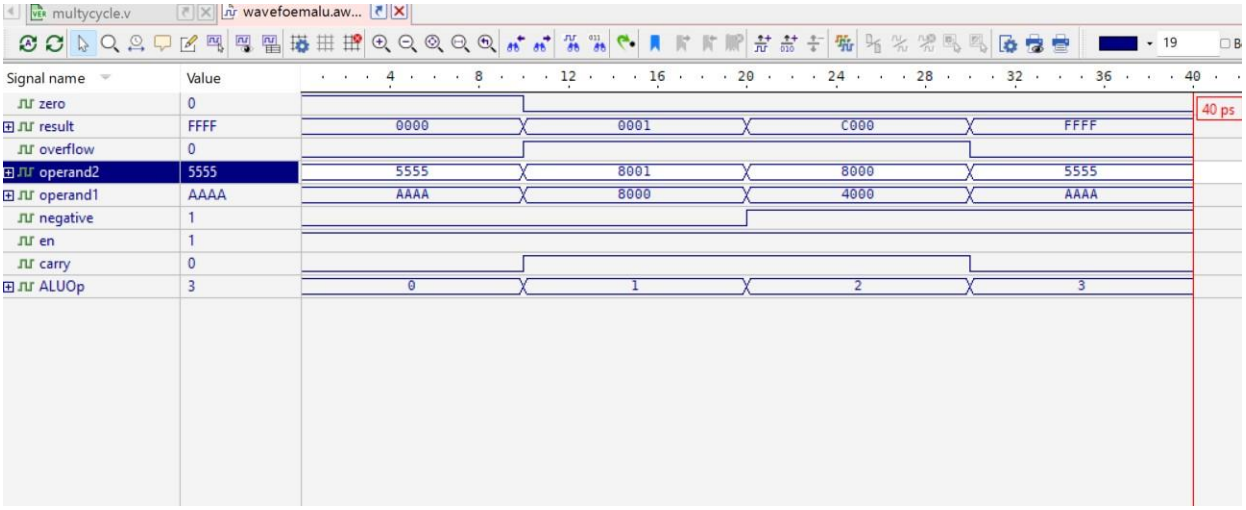


Figure 2: ALU test bench

Extender8:

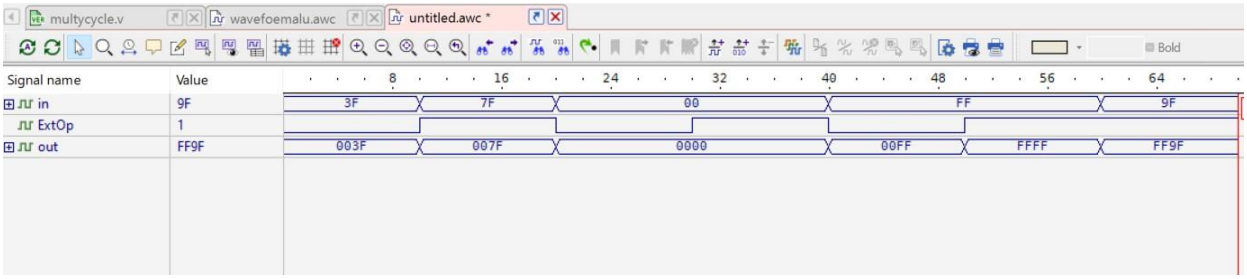


Figure 3: Extender8 test bench

Extender5:

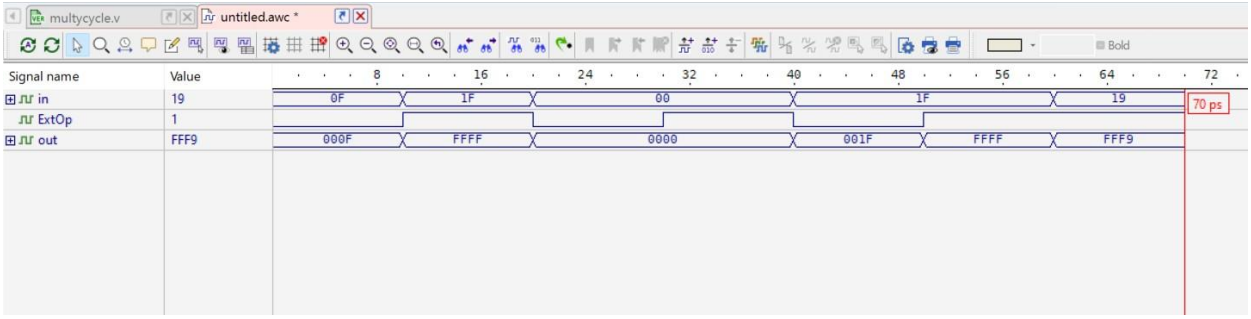


Figure 4: Extender5 test bench

Register file:

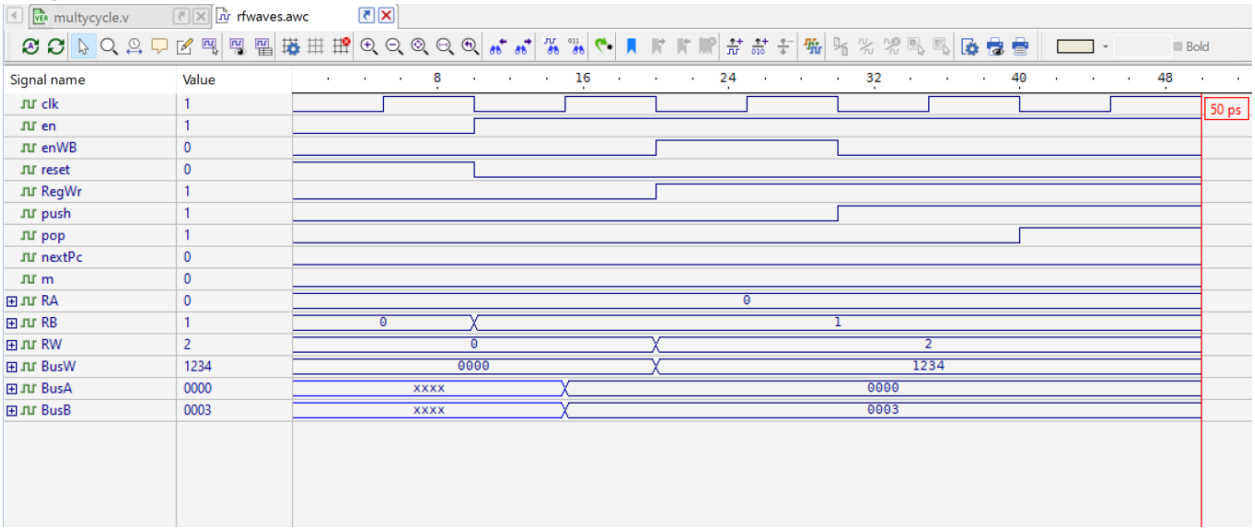


Figure 5: register file test bench

Mux2x1:

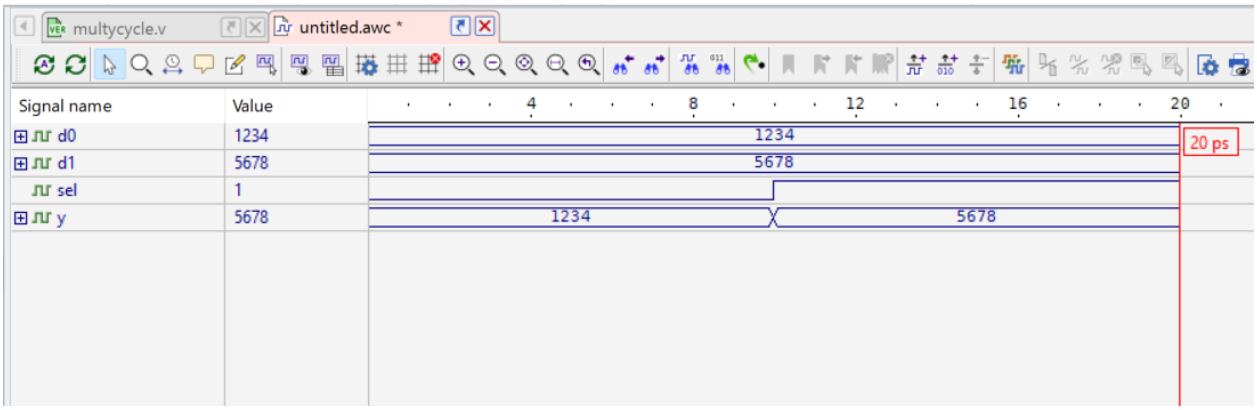
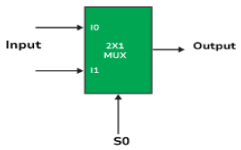


Figure 6: Mux 2x1 test bench

Table 5: Mux 2x1 truth table

2:1 Multiplexer



Truth Table

S_0	I_0	I_1	Y
0	0	X	0
0	1	X	1
1	X	0	0
1	X	1	1

Mux3x1:

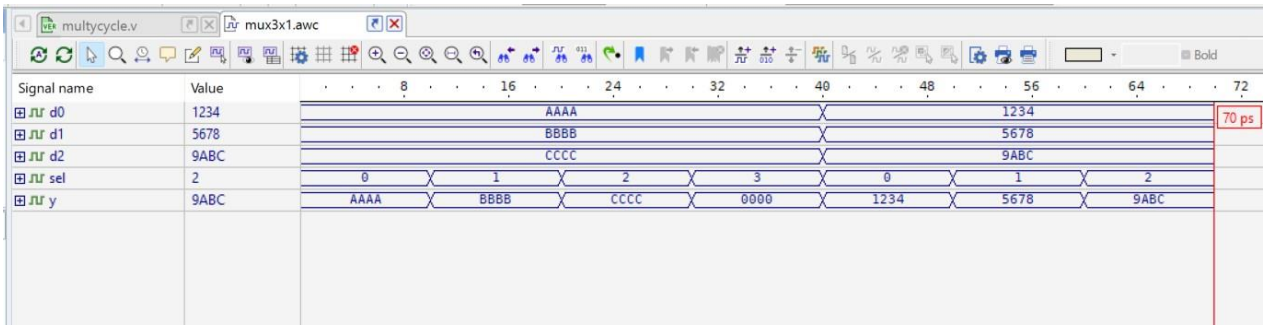


Figure 7: Mux 3x1 test bench

Table 6: Mux 3x1 Truth table

S1	S0	A	B	C	Y
0	0	X	X	X	A
0	1	X	X	X	B
1	0	X	X	X	C
1	1	X	X	X	-

Mux4x1:

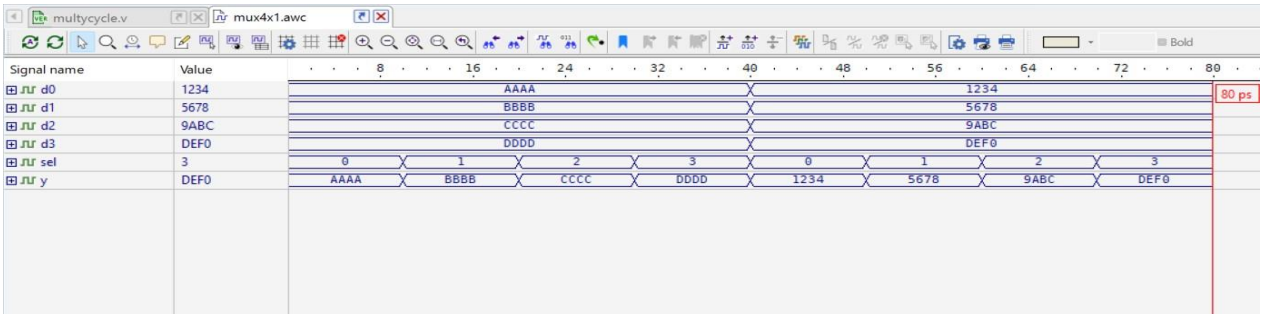
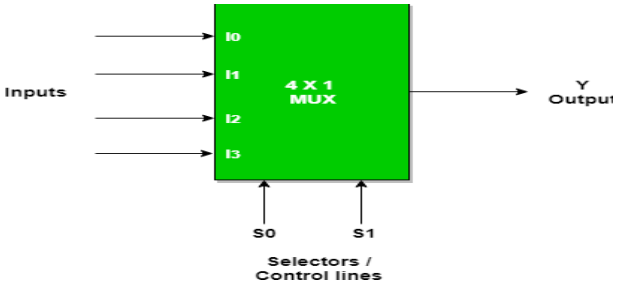


Figure 8: Mux 4x1 test bench

Table 7: Mux 4x1 truth table



Truth Table

S0	S1	Y
0	0	I0
0	1	I1
1	0	I2
1	1	I3

So, final equation,
Y = S0'.S1'.I0 + S0'.S1.I1 + S0.S1'.I2 + S0.S1.I3

Data Memory:

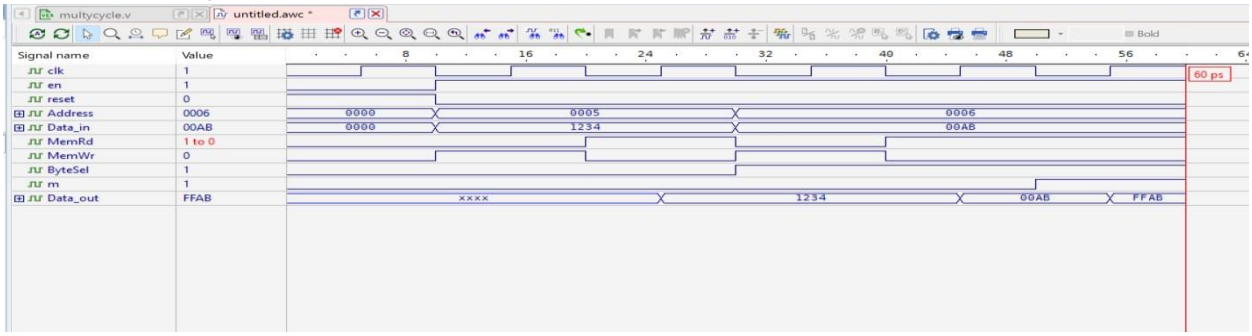


Figure 9: data memory test bench

Instruction Memory:

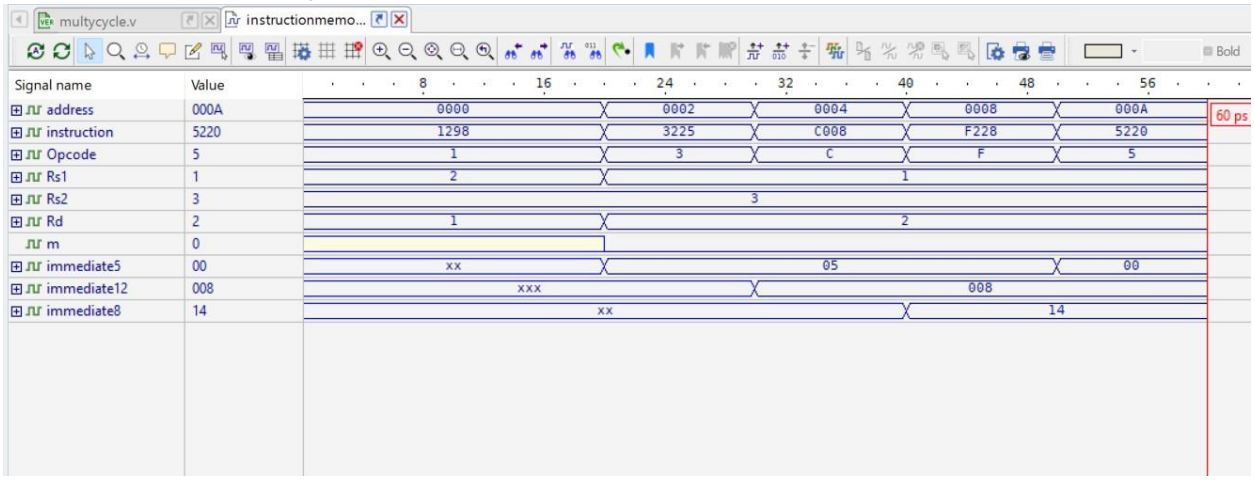


Figure 10: instruction memory test bench

Control Unit:

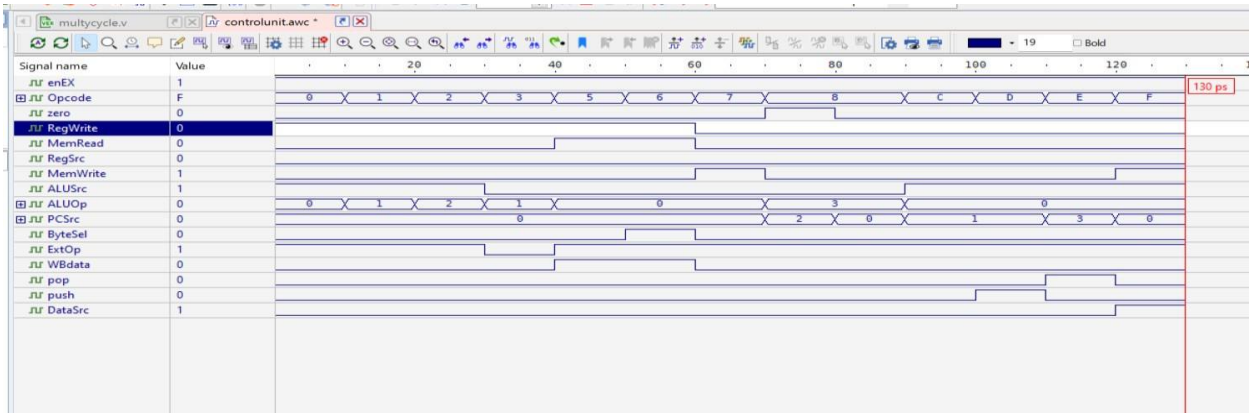


Figure 11: control unit test bench

PC selector:

Figure 12: PC Selector test bench

Adder 16 bit:

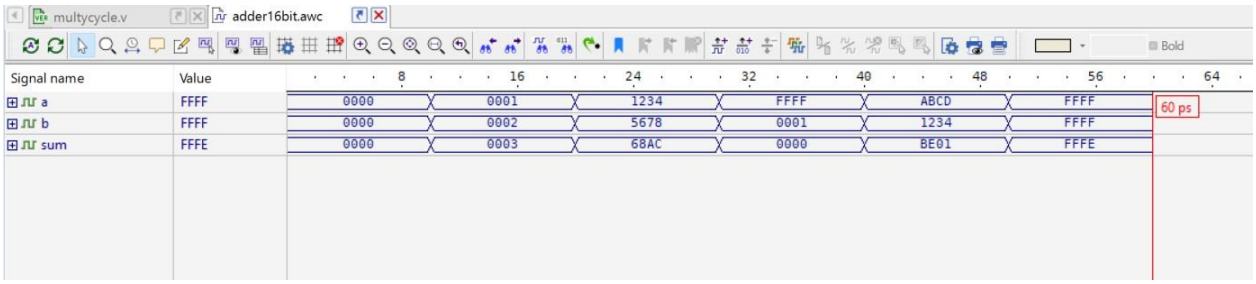


Figure 13: Adder test bench

3. Multi-cycle implementation

In the multi-cycle implementation, we break down the stages of the single-cycle processor into five parts: IF (Instruction Fetch), where we grab the instruction from memory using the Program Counter (PC); ID (Instruction Decode), where we access the register file and interpret the instruction using the control unit; EX (Execution), where the ALU performs the operation specified by the instruction; M (Memory), where data memory operations like reading or writing occur; and W (Write Back), where the result of the operation is written back to the register file. Each of these stages needs to be completed within a single clock cycle.

The figure below shows the Instruction fetch.

```
1484    ////////////////////////////////////////////////////////////////////////////////////////////
1485    //IF
1486
1487    Mux4to1 pcreg(.d0(NextPC),.d1(JTA),.d2(BTA),.d3(BusA),.sel(PCSrc),.y(PC_out));//mux before PC
1488
1489    always @(posedge clk) begin
1490        if(enIF == 1)    begin
1491            PC = PC_out;
1492        end
1493
1494    end
1495
1496    //PC_selector PC1(.PC_in(PC_out),.output1(PC));
1497
1498
1499    InstructionMemory  im(.address(PC), .instruction(instruction), .Opcode(opcode),
1500                         .Rs1(Rs1), .Rs2(Rs2), .Rd(Rd), .m(m),
1501                         .immediate5(immediate5), .immediate8(immediate8), .immediate12(immediate12));
```

Figure 14: Instruction fetch

The figure below shows the Instruction decode.

```
1503    ////////////////////////////////////////////////////////////////////////////////////////////
1504    // ID
1505
1506    // increment the pc after fetch the instruction
1507        always @(posedge clk) begin
1508            if (enID == 1)begin
1509                NextPC = PC + 2;
1510            end
1511        end
1512
1513    Mux2to1  src(.d0(Rs2),.d1(Rd),.sel(RegSrc),.y(RB)); //mux before REgister File
1514
1515
1516
1517    // control unit
1518    ControlUnit  control(
1519    .enEX(enEX),
1520    .Opcode(opcode),
1521    .zero(zero),
1522    .RegSrc(RegSrc),
1523    .RegWrite(RegWrite),
1524    .ExtOp(ExtOp),
1525    .ALUSrc(ALUSrc),
1526    .MemRead(MemRead),
1527    .MemWrite(MemWrite),
1528    .WBdata(WBdata),
1529    .PCSrc(PCSrc),
1530    .ALUOp(ALUOp),
1531    .pop(pop),
1532    .push(push),
1533    .ByteSel(ByteSel),
1534    .DataSrc(DataSrc)
1535
1536    );
1537
```

```
1542        // register file
1543        Register_File  RF (
1544             .clk(clk),
1545             .enWB(enWB),
1546             .en(enID),
1547             .reset(reset),
1548             .RegWr(RegWrite),
1549             .RA(Rs1),
1550             .RB(RB),
1551             .RW(Rd),
1552             .BusW(BusW),
1553             .BusA(BusA),
1554             .BusB(BusB),
1555             .push(push),
1556             .pop(pop),
1557             .nextPc(NextPC),
1558             .m(m)
1559        );
1560
1561
1562
1563
1564        Extender5  ex(immediate5,ExtOp,Ext5);//extender5
1565
1566        Extender8  ex2(immediate8,ExtOp,Ext8);//extender8
1567
1568
1569        adder_16bit add(.a(NextPC),.b(Ext5),.sum(BTA));
1570
1571        // make JTA euqal to PC[15:12] || immediate12
1572        always @(posedge clk) begin
1573             if (enID == 1)begin
1574                  JTA = {PC[15:12],immediate12};
1575             end
1576        end
```

Figure 15: Instruction decode

The figure below shows the Execute stage.

```
1580    ////////////////////////////////////////////////////////////////////////////////////////////////
1581
1582    // EX
1583
1584    Mux2to1  mu(Ext5, BusB, ALUSrc, operand2); // mux before ALU
1585
1586
1587    ALU  uutt (
1588        .en(enEX),
1589        .operand1(BusA),
1590        .operand2(operand2),
1591        .ALUOp(ALUOp),
1592        .result(result),
1593        .zero(zero),
1594        .negative(negative),
1595        .carry(carry),
1596        .overflow(overflow)
1597    );
1598
```

Figure 16: Execute stage

The figure below shows the Memory stage.

```
    ////////////////////////////////////////////////////////////////////////////////////////////
    //wire datamemory_address;
    Mux2to1  dataSrc(.d0(BusB),.d1(Ext8),.sel(DataSrc),.y(DataIn)); //mux before Data Memory
    Mux2to1  dataSrcAddress(.d0(result),.d1(BusA),.sel(DataSrc),.y(datamemory_address)); //mux before Data memory address

    // MEM
    data_memory DM (
        .en(enMEM),
        .clk(clk),
        .reset(reset),
        .Address(datamemory_address),
        .Data_in(DataIn),
        .MemRd(MemRead),
        .MemWr(MemWrite),
        .Data_out(Data_out),
        .ByteSel(ByteSel),
        .m(m)
    );

    Mux2to1 n(.d0(result),.d1(Data_out),.sel(WBdata),.y(BusW)); //last mux


endmodule
```

Figure 17: Memory stage

The figure below shows the instructions that will be sent to the data path, which are stored in the instruction memory.

```
 // R-Type Instruction: ADD R1, R2, R3
// Opcode = 0001, Rd = R1, Rs1 = R2, Rs2 = R3, Unused = 000
instruction_memory[0] = 8'h98; // Lower byte (Rs2 = 011, Unused = 000)
instruction_memory[1] = 8'h12; // Higher byte (Opcode = 0001, Rd = 001, Rs1 = 010)


  // I-Type Instruction: ADDI R2, R1, #5 (Immediate addition)
// Opcode = 0011, m = 0, Rd = R2, Rs1 = R1, Immediate = 5
instruction_memory[2] = 8'h25; // Lower byte (Immediate = 5)
instruction_memory[3] = 8'h32; // Higher Byte


 // J-Type Instruction: JMP 8 (Unconditional Jump)
// Opcode = 1100, Immediate12 = 8
instruction_memory[4] = 8'h08; // Lower byte (Immediate12 low part)
instruction_memory[5] = 8'hC0; // Higher byte (Opcode = 1100)


// S-Type Instruction: SV R1, 20 (Store Immediate)
// Opcode = 1111, Rs1 = R1, Immediate8 = 20, unused = 0
instruction_memory[8] = 8'h28; // Lower byte (Immediate8 = 20)
instruction_memory[9] = 8'hF2; // Higher byte (Opcode = 1111, Rs1 = 001)

// Load the address of R1 into R2
// LW R2, (R1)0 (Load word from address in R1 to R2)
// Opcode = 0101, m = 0, Rd = R2, Rs1 = R1, Immediate5 = 0
instruction_memory[10] = 8'h20; // Lower byte (Rs1 = 001, Immediate5 = 00000)
instruction_memory[11] = 8'h52; // Higher byte (Opcode = 0101, Rd = 010)
```

Figure 18: Instructions

The figures below show how the instructions mentioned are executed in the multi-cycle data path.

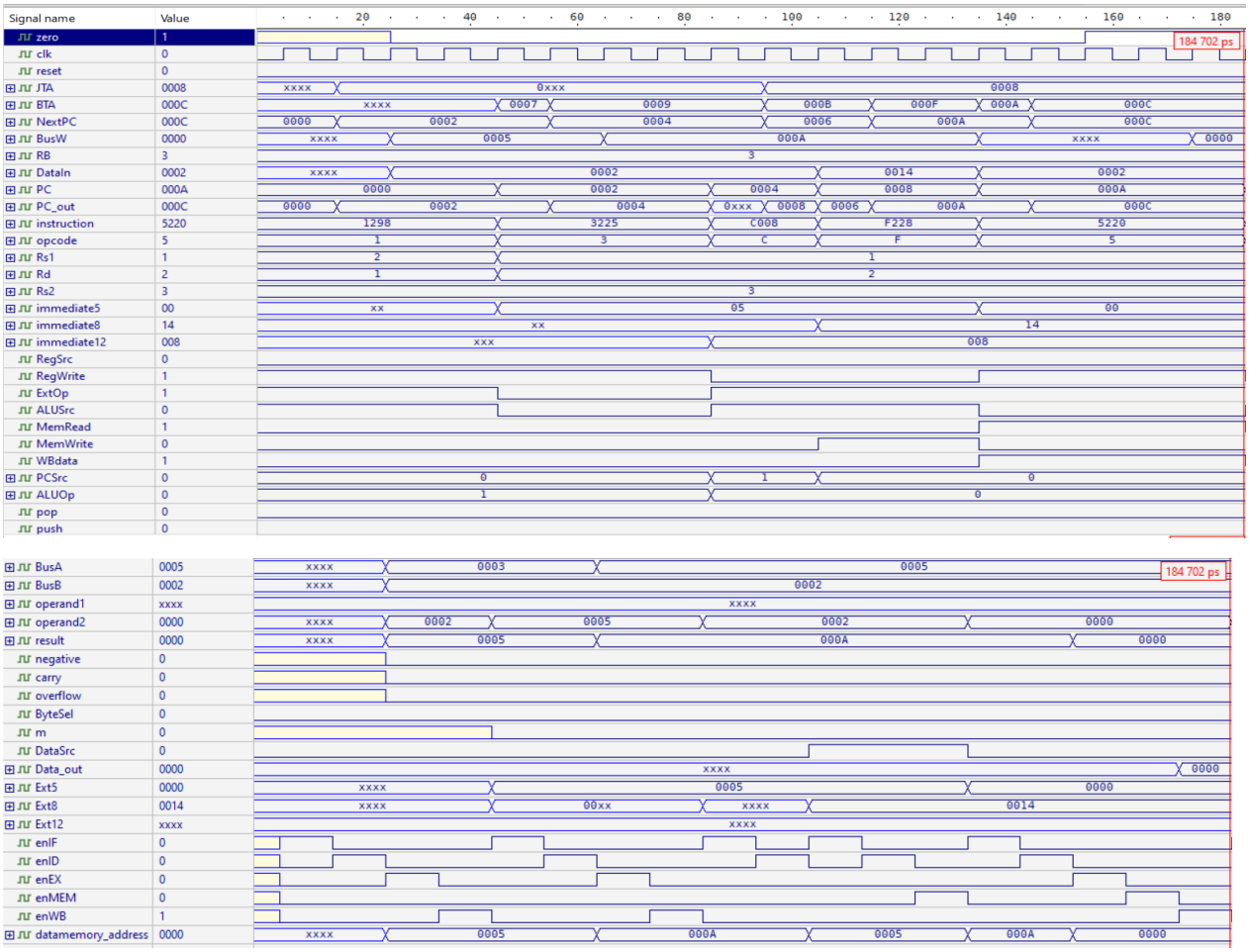


Figure 19: the instructions in the multi-cycle data path

```
run 100 ns
# KERNEL: Time = 0 | reset: 0 | opcode: 1 | m: x | AluSrc: 1 | MemR: 0 | MemW: 0 | RegWr: 1 | RegDes: 0 | WrB: 0 | ExtOp: 1 | R0: x | R7: x | AluOp: 01 | PcSrc: 00 | instType: 0 |
 state: 0000000000000000
# KERNEL: Time = 15000 | reset: 0 | opcode: 1 | m: x | AluSrc: 1 | MemR: 0 | MemW: 0 | RegWr: 1 | RegDes: 0 | WrB: 0 | ExtOp: 1 | R0: x | R7: x | AluOp: 01 | PcSrc: 00 | instType: 0 |
 state: 0000000000000010
# KERNEL: Time = 25000 | reset: 0 | opcode: 1 | m: x | AluSrc: 1 | MemR: 0 | MemW: 0 | RegWr: 1 | RegDes: 0 | WrB: 0 | ExtOp: 1 | R0: 1 | R7: 0 | AluOp: 01 | PcSrc: 00 | instType: 0 |
 state: 0000000000000010
# KERNEL: Time = 45000 | reset: 0 | opcode: 3 | m: 0 | AluSrc: 0 | MemR: 0 | MemW: 0 | RegWr: 1 | RegDes: 0 | WrB: 0 | ExtOp: 0 | R0: 1 | R7: 0 | AluOp: 01 | PcSrc: 00 | instType: 0 |
 state: 0000000000000010
# KERNEL: Time = 55000 | reset: 0 | opcode: 3 | m: 0 | AluSrc: 0 | MemR: 0 | MemW: 0 | RegWr: 1 | RegDes: 0 | WrB: 0 | ExtOp: 0 | R0: 1 | R7: 0 | AluOp: 01 | PcSrc: 00 | instType: 0 |
 state: 0000000000000100
# KERNEL: Time = 85000 | reset: 0 | opcode: c | m: 0 | AluSrc: 1 | MemR: 0 | MemW: 0 | RegWr: 0 | RegDes: 0 | WrB: 0 | ExtOp: 1 | R0: 1 | R7: 0 | AluOp: 00 | PcSrc: 01 | instType: 0 |
 state: 0000000000000100
# KERNEL: Time = 95000 | reset: 0 | opcode: c | m: 0 | AluSrc: 1 | MemR: 0 | MemW: 0 | RegWr: 0 | RegDes: 0 | WrB: 0 | ExtOp: 1 | R0: 1 | R7: 0 | AluOp: 00 | PcSrc: 01 | instType: 0 |
 state: 0000000000000110
# KERNEL: stopped at time: 100 ns
run 100 ns
# KERNEL: Time = 105000 | reset: 0 | opcode: f | m: 0 | AluSrc: 1 | MemR: 0 | MemW: 1 | RegWr: 0 | RegDes: 0 | WrB: 0 | ExtOp: 1 | R0: 1 | R7: 0 | AluOp: 00 | PcSrc: 00 | instType: 1 |
 state: 0000000000000110
# KERNEL: Time = 115000 | reset: 0 | opcode: f | m: 0 | AluSrc: 1 | MemR: 0 | MemW: 1 | RegWr: 0 | RegDes: 0 | WrB: 0 | ExtOp: 1 | R0: 1 | R7: 0 | AluOp: 00 | PcSrc: 00 | instType: 1 |
 state: 0000000000001010
# KERNEL: Time = 135000 | reset: 0 | opcode: 5 | m: 0 | AluSrc: 0 | MemR: 1 | MemW: 0 | RegWr: 1 | RegDes: 0 | WrB: 1 | ExtOp: 1 | R0: 1 | R7: 0 | AluOp: 00 | PcSrc: 00 | instType: 0 |
 state: 0000000000001010
# KERNEL: Time = 145000 | reset: 0 | opcode: 5 | m: 0 | AluSrc: 0 | MemR: 1 | MemW: 0 | RegWr: 1 | RegDes: 0 | WrB: 1 | ExtOp: 1 | R0: 1 | R7: 0 | AluOp: 00 | PcSrc: 00 | instType: 0 |
 state: 0000000000001100
# KERNEL: Time = 185000 | reset: 0 | opcode: x | m: 0 | AluSrc: 1 | MemR: 0 | MemW: 0 | RegWr: 0 | RegDes: 0 | WrB: 0 | ExtOp: 1 | R0: 1 | R7: 0 | AluOp: 00 | PcSrc: 00 | instType: 0 |
 state: 0000000000001100
# KERNEL: Time = 195000 | reset: 0 | opcode: x | m: 0 | AluSrc: 1 | MemR: 0 | MemW: 0 | RegWr: 0 | RegDes: 0 | WrB: 0 | ExtOp: 1 | R0: 1 | R7: 0 | AluOp: 00 | PcSrc: 00 | instType: 0 |
 state: 0000000000001110
# KERNEL: stopped at time: 200 ns

>
Console
```

Figure 20: Monitor values and print at each time step

A testbench was used to test the path, as the below figure show.

```
`timescale 1ns / 1ps
module multiCycle_tb;
    wire zero;
    reg clk, reset;


    wire [15:0] JTA;
    wire [15:0] BTA;
    wire [15:0] NextPC;
    wire [15:0]BusW;
    wire [2:0] RB;
    wire [15:0] DataIn;
    wire [15:0] PC,PC_out,instruction;
    wire [3:0] opcode;
    wire [2:0] Rs1;
    wire [2:0] Rd;
    wire [2:0] Rs2;
    wire [4:0] immediate5;
    wire [7:0] immediate8;
    wire [11:0] immediate12;
    wire RegSrc;
    wire RegWrite;
    wire ExtOp;
    wire ALUSrc;
    wire MemRead;
    wire MemWrite;
    wire WBdata;
    wire [1:0] PCSrc;
    wire [1:0] ALUOp;
    wire pop;
    wire push;
    wire [15:0]BusA;
    wire [15:0]BusB;

    reg [15:0] operand1;
    wire [15:0] operand2;
    wire [15:0] result;
    wire negative;
    wire carry;
    wire overflow;
```

```
    wire ByteSel;
    wire m;
    wire DataSrc;
    wire [15:0] Data_out;
    wire [15:0] Ext5, Ext8, Ext12;
    wire enIF, enID, enEX, enMEM, enWB;
    wire [15:0] datamemory_address;


    // Instantiate the Unit Under Test (UUT)
    CPU uut (
        .clk(clk),
        .reset(reset),
        .zero(zero),
        .PC(PC),
        .PC_out(PC_out),
        .instruction(instruction),
        .opcode(opcode),
        .Rs1(Rs1),
        .Rs2(Rs2),
        .Rd(Rd),
        .immediate5(immediate5),
        .immediate8(immediate8),
        .immediate12(immediate12),
        .JTA(JTA),
        .BTA(BTA),
        .NextPC(NextPC),
        .BusA(BusA),
        .BusB(BusB),
        .result(result),
        .Data_out(Data_out),
        .RegWrite(RegWrite),
        .ExtOp(ExtOp),
        .ALUSrc(ALUSrc),
        .MemRead(MemRead),
        .MemWrite(MemWrite),
        .WBdata(WBdata),
        .PCSrc(PCSrc),
        .ALUOp(ALUOp),
```

```
1712            .ByteSel(ByteSel),
1713            .m(m),
1714            .BusW(BusW),
1715            .RB(RB),
1716            .DataIn(DataIn),
1717            .RegSrc(RegSrc),
1718            .pop(pop),
1719            .push(push),
1720            .operand1(operand1),
1721            .operand2(operand2),
1722            .negative(negative),
1723            .carry(carry),
1724            .overflow(overflow),
1725            .DataSrc(DataSrc),
1726            .Ext5(Ext5),
1727            .Ext8(Ext8),
1728            .Ext12(Ext12),
1729            .enIF(enIF),
1730            .enID(enID),
1731            .enEX(enEX),
1732            .enMEM(enMEM),
1733            .enWB(enWB),
1734            .datamemory_address(datamemory_address)
1735
1736        );
1737
1738
1739
1740        initial begin
1741            // Initialize inputs
1742            clk = 1'b0;
1743            reset = 1'b0;
1744        //   NextPC=32'h0;
1745            //BTA = 32'h0;
1746            //JTA=32'h1;
1747            // Generate clock
1748            forever begin
1749              #5 clk = ~clk;
1750            end
```

```
1751
1752    #400 $finish;
1753    end
1754
1755
1756
1757
1758    initial begin
1759        $monitor("Time = %t, PC = %h, Zero = %b, Result = %h", $time, uut.PC, zero, result);
1760    end
1761
1762  endmodule
1763
```

Figure 21: Path test bench

4. Team Work

Three of us worked on building the data path together, and then each of us focused on implementing a specific part. Nirmeen worked on the data memory and collecting the CPU components, and Mohammad handled the ALU and worked on the register file, and Abdalrahim took care of the instruction memory and the Program Counter (PC). When it came to simulating the system, all of us participated by testing a number of instructions on the data path to ensure it worked correctly.

5. Conclusion

In conclusion of our multi-cycle RISC processor design and implementation project, the applicability and efficiency of the multi-cycle instruction execution approach were vividly brought out. Division of the instructions into a number of stages made our processor certain to utilize resources to the full while executing correctly upon all types of instructions, arithmetic, logic, load/store, and branch operations. Separate instruction and data memories, as well as byte-addressable and little-endian memory configurations, improved processor performance and operational simplicity.

The functionality of the processor was checked by running extensive simulations and rigorous testing on the implemented instructions. Teamwork helped to understand the architecture and all its components, thereby helping in developing a strong and well-documented design. Not only does the completion of this project indication reflect the effectiveness of multi-cycle processors, but it also outlines the serious significance of teamwork plus rigorous design verification for a reliable digital system.