



**Faculty of Engineering and Technology
Computer Systems Engineering**

**COMPUTER ARCHITECTURE
ENCS4370**

Project 2 Report
Designing a multicycle RISC processor

Prepared by:

Abdel Rahman Shahan 1211753

Mahmoud Awad 1212677

Instructor:

Dr. Aziz Qaroush

Section: 1

Date: 6/18/2024

1. Design and Implementation

1.1 Detailed description of the datapath

1.1.1 Datapath Components

1. Program Counter (PC)

- **Function:** Holds the address of the next instruction to be fetched from the instruction memory.
- **Module:** pc
- **Inputs:** clk, reset, d
- **Output:** q

2. Instruction Memory (IM)

- **Function:** Stores the program's instructions.
- **Module:** InstructionMemory
- **Inputs:** addr
- **Output:** instr

3. Control Unit (CU)

- **Function:** Generates control signals based on the opcode of the current instruction.
- **Module:** ControlUnit
- **Inputs:** instruction
- **Outputs:** Various control signals like RegDst, ExtOp, ALUSrc, ALUOp, MemRd, MemWr, WBdata, RegWr, branch, ret, store, sv, LType, JR

4. Instruction Decoder (ID)

- **Function:** Decodes the instruction to extract opcode, registers, immediate values, etc.
- **Module:** InstructionDecoder
- **Inputs:** instr, func
- **Outputs:** Rd, Rs1, Rs2, Imm, SImm, JumpOffset, m

5. Register File (RF)

- **Function:** Holds the register values and provides the values of source registers to the ALU.
- **Module:** RegisterFile
- **Inputs:** clk, branch, m, Rd, Rs1, Rs2, WriteData, WriteReg, RegWrite, store
- **Outputs:** BusA, BusB

6. Extender

- **Function:** Extends immediate values to 16 bits.
- **Module:** Extender
- **Inputs:** Imm_in, ExtOp
- **Outputs:** Imm_out

7. ALU

- **Function:** Performs arithmetic and logical operations.
- **Module:** ALU
- **Inputs:** a, b, ALUOp
- **Outputs:** result, zero, Negative, carry, overflow

8. Data Memory (DM)

- **Function:** Stores data and provides read/write access to memory locations.
- **Module:** DataMemory
- **Inputs:** clk, WBdata, addr, Data_in, Rdata
- **Outputs:** Data_out

9. Load Extender

- **Function:** Extends loaded data for different types of load operations.
- **Module:** LoadExtender
- **Inputs:** loaded_data
- **Outputs:** U_Byte, S_Byte

10. PC Control

- **Function:** Controls the next value of the PC based on branch/jump conditions.
- **Module:** PcControl
- **Inputs:** Op, Zero, Negative, Carry, Overflow, branch, m
- **Outputs:** PCSrc

11. Mini PC Control

- **Function:** Handles jump and return instructions.
- **Module:** miniPcControl
- **Inputs:** Op
- **Outputs:** PCSrc

1.1.2 Datapath Assembly

1. Instruction Fetch (IF) Stage

- The pc module holds the address of the next instruction.
- The InstructionMemory module fetches the instruction at the address provided by the pc.

2. Instruction Decode (ID) Stage

- The fetched instruction is passed to the ControlUnit and InstructionDecoder.
- The ControlUnit generates control signals based on the opcode.
- The InstructionDecoder extracts the fields of the instruction (like Rd, Rs1, Rs2, Imm, etc.).

3. Register Read

- The source registers Rs1 and Rs2 are read from the RegisterFile module.

4. Immediate Extension

- The Extender module extends the immediate value if necessary.

5. Execute (EX) Stage

- The ALU performs the required operation (ADD, SUB, AND) on the operands provided.
- Operands can be register values or extended immediate values based on the control signals.

6. Memory Access (MEM) Stage

- For load/store instructions, the DataMemory module reads from or writes to memory.
- The LoadExtender module extends the data loaded from memory if necessary.

7. Write Back (WB) Stage

- The result of the ALU operation or data loaded from memory is written back to the register file.

8. PC Update

- The PcControl and miniPcControl modules determine the next value of the PC based on branch, jump, and return instructions.
- The pc module is updated with the new PC value for the next cycle.

1.2 A complete description of the control signals

1.2.1 Control Signals Description

1. **RegDst (2 bits):** Determines which register is used as the destination register.
 - **00:** Not used
 - **01:** Destination register is Rd (for R-type instructions)
 - **10:** Destination register is R7 (for CALL instruction)
2. **ExtOp:** Determines the type of immediate extension.
 - **0:** Zero extension
 - **1:** Sign extension
3. **ALUSrc:** Selects the second operand for the ALU.
 - **0:** Second operand is BusB (register)
 - **1:** Second operand is Imm_out (extended immediate value)
4. **ALUOp (3 bits):** Specifies the ALU operation to be performed.
 - **000:** AND
 - **001:** ADD
 - **010:** SUB
5. **MemRd:** Enables memory read operation.
 - **0:** Memory read disabled
 - **1:** Memory read enabled
6. **MemWr:** Enables memory write operation.
 - **0:** Memory write disabled
 - **1:** Memory write enabled
7. **WBdata (2 bits):** Selects the data to be written back to the register file.
 - **00:** ALU result
 - **01:** Data from memory
 - **10:** PC + 2 (for CALL instruction)

8. RegWr: Enables writing to the register file.

- **0:** Register write disabled
- **1:** Register write enabled

9. branch: Indicates if the instruction is a branch instruction.

- **0:** Not a branch instruction
- **1:** Branch instruction

10. store: Used in store instructions.

- **0:** Not a store instruction
- **1:** Store instruction

11. sv: Indicates if the instruction is a store value instruction.

- **0:** Not a store value instruction
- **1:** Store value instruction

12. LType: Indicates the type of load instruction.

- **0:** LBU (load byte unsigned)
- **1:** LW (load word)

13. JR: Indicates if the instruction is a jump register instruction (JMP).

- **0:** Not a jump register instruction
- **1:** Jump register instruction

14. Func: Specifies the function code for type instructions.

- **00:** R-type
- **01:** I-type
- **10:** J-type
- **11:** S-type

1.2.1 Control Signals Truth Table

Op	Instr	func	RegDst	ExtOp	ALUSrc	ALUOp	MemRd	MemWr	WBdata	RegWr	Branch	Store	Sv	LType	JR
0000	AND	00	0	X	0	000	0	0	00	1	0	0	0	X	0
0001	ADD	00	0	X	0	001	0	0	00	1	0	0	0	X	0
0010	SUB	00	0	X	0	010	0	0	00	1	0	0	0	X	0
0011	ADDI	01	0	1	1	001	0	0	00	1	0	0	0	X	0
0100	ANDI	01	0	0	1	000	0	0	00	1	0	0	0	X	0
0101	LW	01	0	1	1	001	1	0	01	1	0	0	0	1	0
0110	LBu	01	0	1	1	001	1	0	01	1	0	0	0	0	0
0110	LBs	01	0	1	1	001	1	0	01	1	0	0	0	0	0
0111	SW	01	X	1	1	001	0	1	XX	0	0	1	0	X	0
1000	BGT	01	X	1	0	010	0	0	XX	0	1	0	0	X	0
1000	BGTZ	01	X	1	0	010	0	0	XX	0	1	0	0	X	0
1001	BLT	01	X	1	0	010	0	0	XX	0	1	0	0	X	0
1001	BLTZ	01	X	1	0	010	0	0	XX	0	1	0	0	X	0
1010	BEQ	01	X	1	0	010	0	0	XX	0	1	0	0	X	0
1010	BEQZ	01	X	1	0	010	0	0	XX	0	1	0	0	X	0
1011	BNE	01	X	1	0	010	0	0	XX	0	1	0	0	X	0
1011	BNEZ	01	X	1	0	010	0	0	XX	0	1	0	0	X	0
1100	JMP	10	X	0	X	XXX	0	0	XX	0	0	0	X	X	1
1101	CALL	10	1	0	X	XXX	0	0	10	1	0	0	X	X	0
1110	RET	10	X	X	X	XXX	0	0	XX	0	0	0	X	X	1
1111	Sv	11	x	X	X	XXX	0	1	XX	0	0	0	1	X	0

1.2.3 Boolean Equations for Control Signals

1. RegDst

- **RegDst1** = CALL
- **RegDst0** = R-type (AND, ADD, SUB, ANDI, ADDI, LW, LBu, LBs)

2. ExtOp

- $\text{ExtOp} = \text{ADDI} \mid \text{ANDI} \mid \text{LW} \mid \text{SW} \mid \text{Branch}$

3. ALUSrc

- $\text{ALUSrc} = \text{ADDI} \mid \text{ANDI} \mid \text{LW} \mid \text{SW}$

4. ALUOp

- $\text{ALUOp}[2] = \text{SUB} \mid \text{BGT} \mid \text{BLT} \mid \text{BEQ} \mid \text{BNE}$
- $\text{ALUOp}[1] = \text{ADD} \mid \text{ADDI} \mid \text{LW} \mid \text{SW}$
- $\text{ALUOp}[0] = \text{AND} \mid \text{ANDI}$

5. MemRd

- MemRd = LW | LBU

6. MemWr

- MemWr = SW | SV

7. WBdata

- WBdata[1] = CALL
- WBdata[0] = LW | LBU

8. RegWr

- RegWr = R-type (AND, ADD, SUB) | ADDI | ANDI | LW | LBU | CALL | LBS

9. branch

- branch = BGT | BLT | BEQ | BNE

10. store

- store = SW

11. sv

- sv = SV

12. LType

- LType = LW

13. JR

- JR = JMP | RET

14. func

- Func [0] = AND | ADD | SUB
- Func [1] = ADDI | ANDI | LW | LB | SW | BGT | BLT | BEQ | BNE
- Func [2] = JMP | CALL | RET
- Func [3] = SV

1.3 The implementation details, and the design choices with justification

1.3.1 Implementation Details

1. Processor Modules:

- **ALU (Arithmetic Logic Unit):** Performs arithmetic and logic operations based on the control signals.
- **Control Unit:** Decodes instructions and generates control signals for the rest of the processor.
- **Data Memory:** Stores and retrieves data based on address and control signals.
- **Instruction Decoder:** Decodes the instruction to extract opcodes, register addresses, and immediate values.
- **Instruction Memory:** Stores the program instructions.
- **PC (Program Counter):** Holds the address of the next instruction to be executed.
- **Register File:** Holds the processor's registers and handles read and write operations.
- **PcControl:** Controls the next PC value based on branch and jump conditions.
- **Extender:** Extends immediate values to 16-bit values for arithmetic operations.
- **LoadExtender:** Extends loaded bytes to 16-bit values, either zero-extended or sign-extended.
- **MiniPcControl:** Generates PC source control signals for specific instructions like jumps and returns.

2. Pipeline Stages:

- **IF (Instruction Fetch):** Fetches the instruction from the instruction memory.
- **ID (Instruction Decode):** Decodes the fetched instruction and reads the register file.
- **EX (Execution):** Executes the operation in the ALU and calculates branch targets.
- **MEM (Memory Access):** Accesses data memory for load and store instructions.
- **WB (Write Back):** Writes the result back to the register file.

1.3.2 Design Choices and Justifications

1. Multi-Cycle Design:

- **Choice:** Implement a multi-cycle processor rather than a single-cycle or pipelined processor.
- **Justification:** Multi-cycle processors allow each instruction to take a variable number of cycles, improving overall efficiency by not forcing all instructions to take the same number of cycles. This approach also reduces hardware complexity and cost compared to single-cycle designs, which require all operations to be completed within one clock cycle.

2. ALU Design:

- **Choice:** Implement basic arithmetic and logic operations (AND, ADD, SUB) with zero, carry, overflow, and negative flags.
- **Justification:** These operations cover the essential arithmetic and logic needs for most instructions. The flags provide necessary information for branch and condition checking.
- **Control Unit Design:**
- **Choice:** Use a Control Unit that generates control signals based on the instruction opcode and function fields.
- **Justification:** This allows the processor to interpret and execute a wide variety of instructions, making it versatile. The control signals drive the operation of other components, ensuring correct instruction execution.

3. Instruction Memory:

- **Choice:** Initialize with a predefined set of instructions to test each opcode.
- **Justification:** This provides a straightforward way to verify the functionality of the processor and ensure that all instruction types are correctly implemented.

4. Register File:

- **Choice:** Use 8 registers, each 16 bits wide, with a specific hardwired zero register (R0).
- **Justification:** Eight registers provide a balance between resource usage and functionality. The hardwired zero register ensures that certain instructions always have access to a zero value, simplifying operations and reducing the need for additional control logic.

5. Data Memory:

- **Choice:** Implement a simple memory model with read and write operations controlled by signals.
- **Justification:** This allows the processor to perform load and store operations, essential for many programs. The memory model is straightforward to implement and test.

6. Extender and LoadExtender:

- **Choice:** Use separate modules for extending immediate values and loaded bytes.
- **Justification:** This separation of concerns simplifies the design of each module and ensures that each performs a specific, well-defined function.

7. PC Control:

- **Choice:** Use a dedicated PcControl module to handle the next PC value based on various conditions.
- **Justification:** Centralizing PC control logic makes it easier to manage and debug. It also allows for more flexible and complex branching and jumping logic.

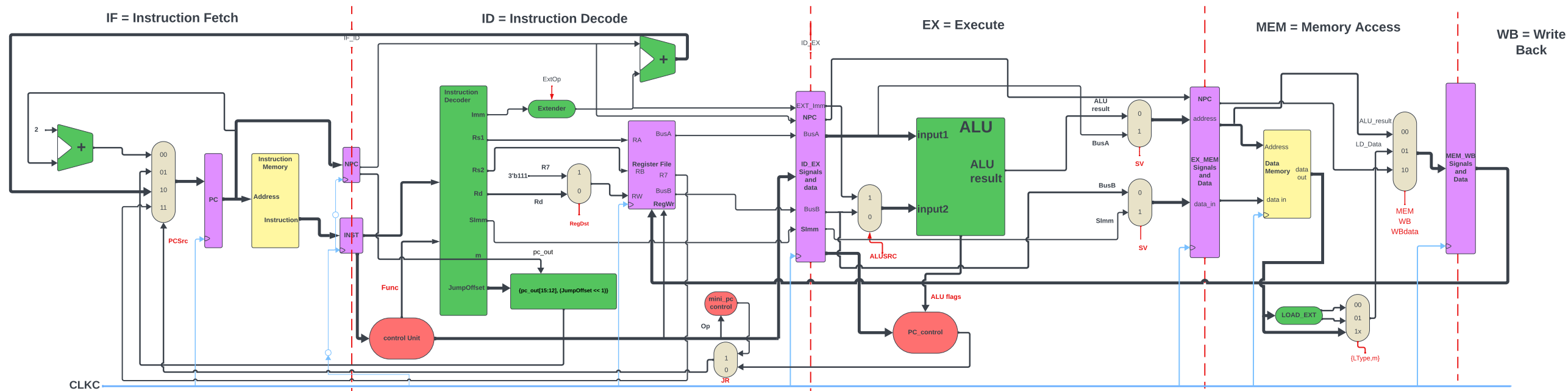
8. MiniPcControl:

- **Choice:** Handle specific jump and return instructions with a mini control module.
- **Justification:** This specialized control ensures that jumps and returns are handled efficiently and correctly, reducing the risk of errors in these critical operations.

1.4 Datapath and control path block diagrams.

1212677

1211753



1.5 A list of sources for any parts of your design that are not entirely yours.

There was no components needed from outside resources. The lecture slides are enough.

1.6 Carry out the design and implementation

While doing design and implementation we took the following aspects in mind:

- **Correctness of the individual components:** All components were tested perfectly
- **Correctness of the overall design when connecting these components together:** all components were connected then tested perfectly
- **Completeness: all instructions were implemented properly:** all instructions were tested in it's all possible cases and the results were awesome.

2. Simulation and Testing

2.1 Test1:

Sum first 4 values in memory

- **Expected output:**
 - **R [1] = 10**
 - **DataMem [8] = 10**

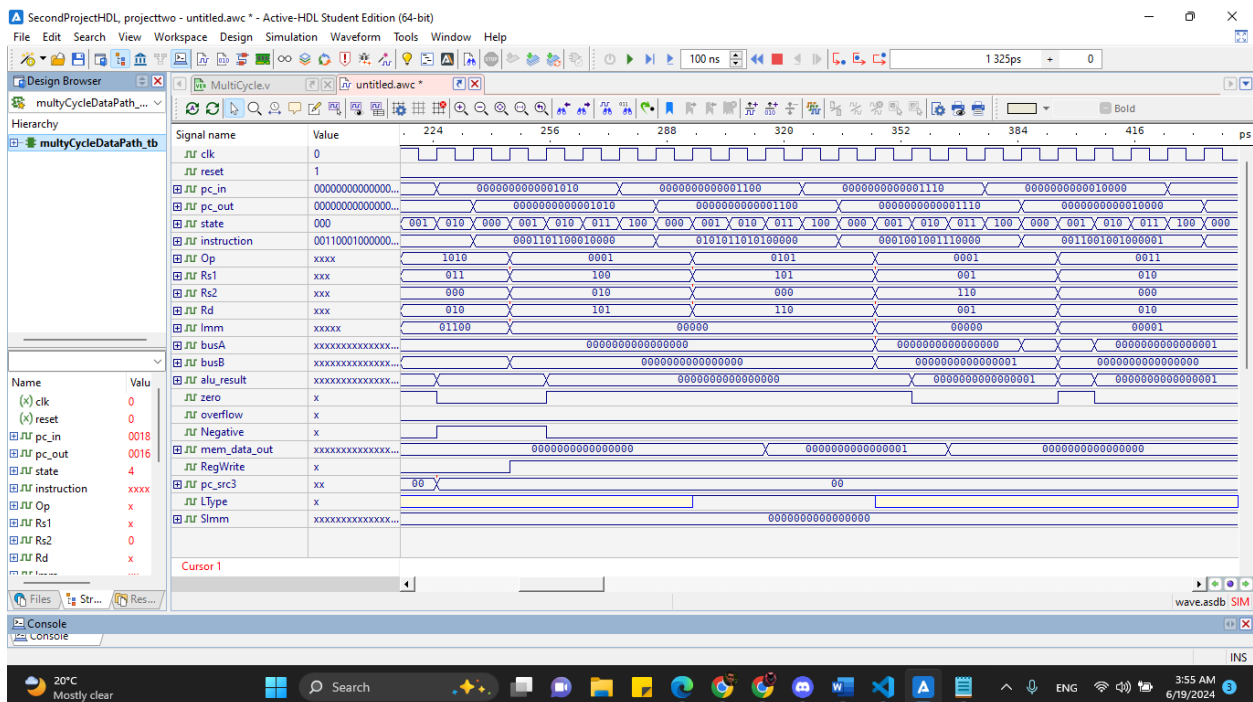
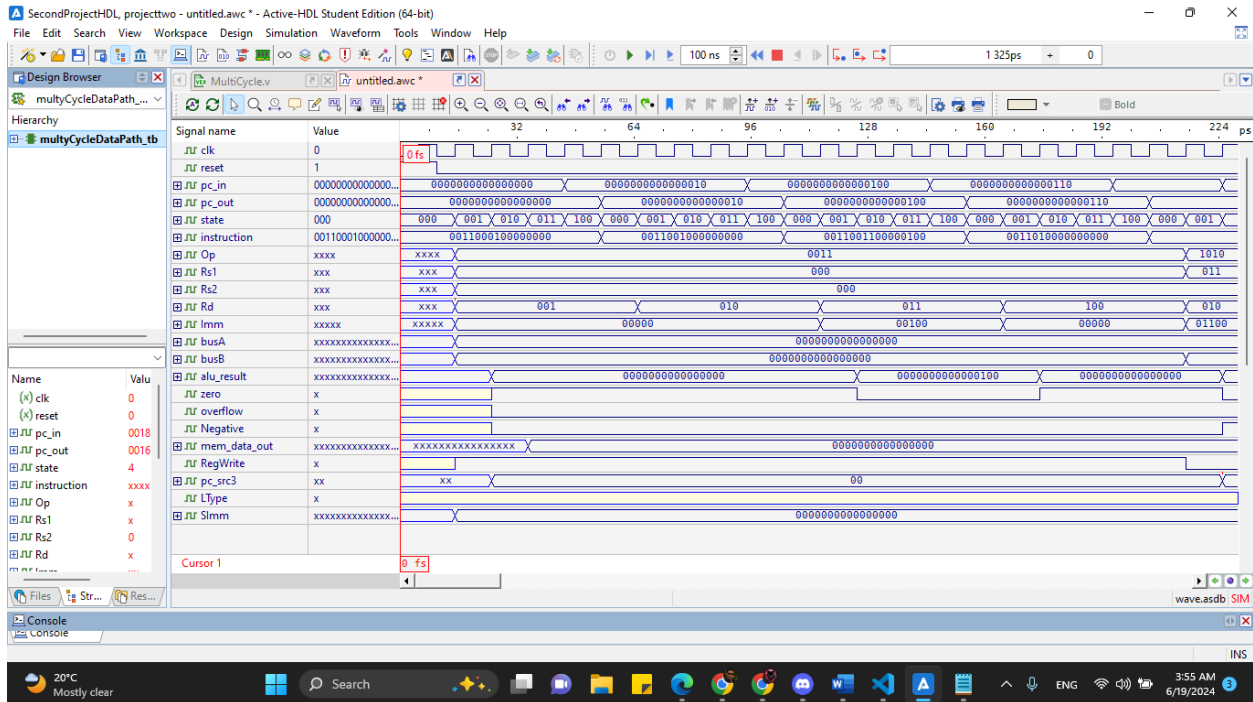
- **Instruction Memory:**

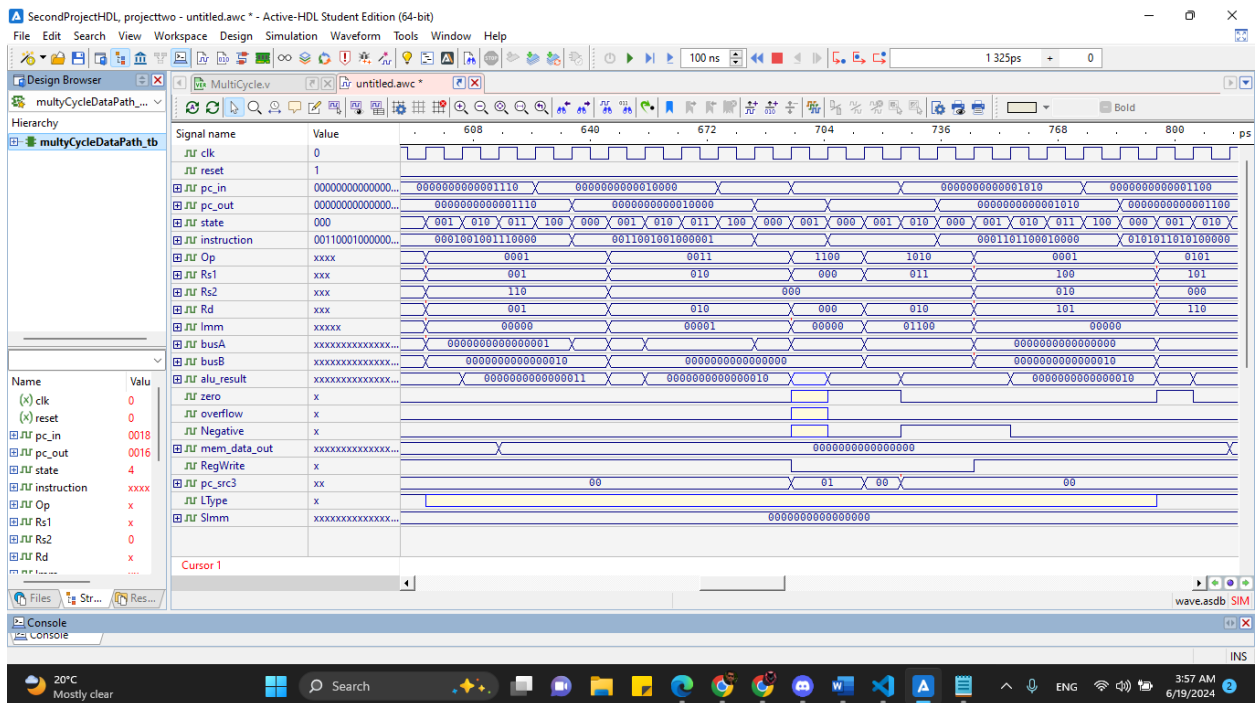
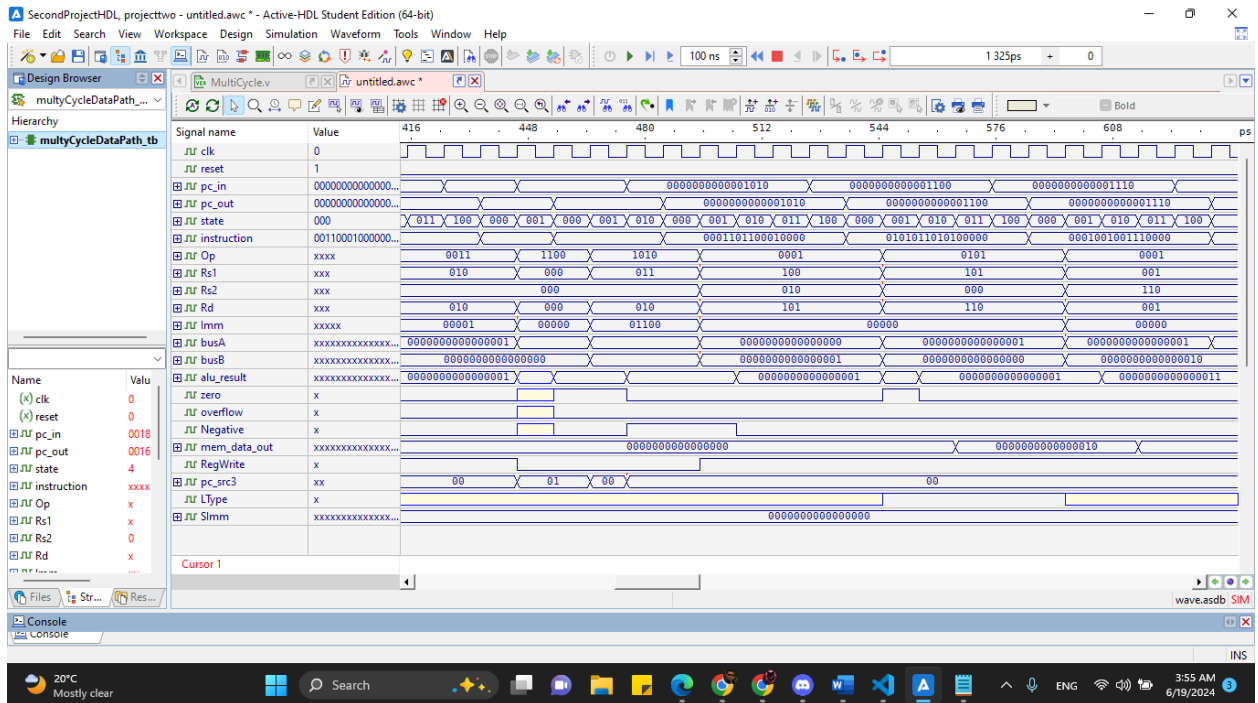
```
• // # Initialize registers
• memory[0] = 16'b0011000100000000; // ADDI R1, R0, 0      # R1 will hold the sum, initialize to 0
• memory[1] = 16'b0011001000000000; // ADDI R2, R0, 0      # R2 will be the loop counter, initialize to 0
• memory[2] = 16'b0011001100000100; // ADDI R3, R0, 4      # R3 is the loop limit (4)
• memory[3] = 16'b0011010000000000; // ADDI R4, R0, MEMORY # R4 is the base address of the memory
• memory[4] = 16'b1010001001101100; // loop BEQ R2, R3, end # If loop counter R2 equals 4, branch to end
• // # Load the value from memory
• memory[5] = 16'b0001101100010000; // ADD R5, R4, R2      # Compute the address (base + offset)
• memory[6] = 16'b0101011010100000; // # Load the word from memory into R6
• // # Add the value to the sum
• memory[7] = 16'b0001001001110000; //ADD R1, R1, R6      # R1 = R1 + R6
• // # Increment the loop counter
• memory[8] = 16'b0011001001000001; // # R2 = R2 + 1
• // # Repeat the loop
• memory[9] = 16'b1100000000000100; // J loop = (8)
• // # Store the sum in a known memory location or register
• memory[10] = 16'b0111000100001000; // # end Store the sum in memory location SUM mem[8]
```

- **Data Memory:**

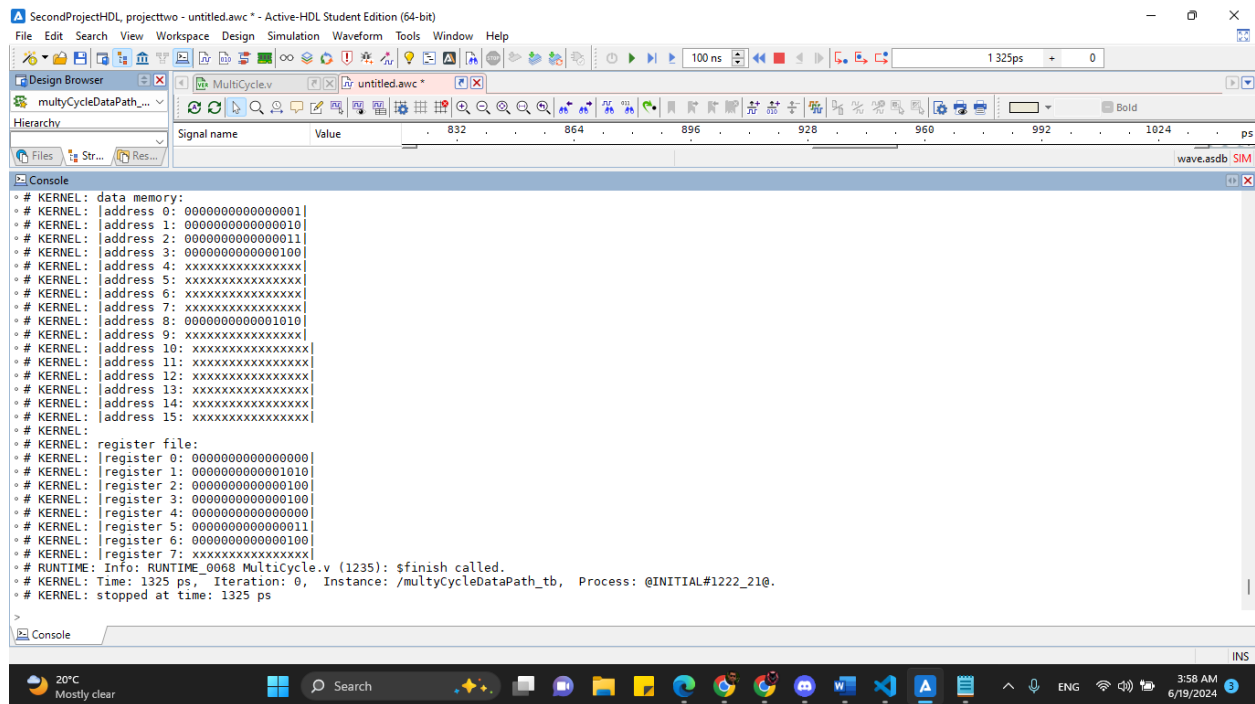
```
• memory[0] = 16'b0000000000000001;
• memory[1] = 16'b0000000000000010;
• memory[2] = 16'b0000000000000011;
• memory[3] = 16'b0000000000000100;
```

- **Simulation:**





- **Result:**



SecondProjectHDL, projecttwo - untitled.awc * - Active-HDL Student Edition (64-bit)

File Edit Search View Workspace Design Simulation Waveform Tools Window Help

Design Browser

multicycleDataPath_...

Hierarchy

Signal name Value . 832 . . 864 . . 896 . . 928 . . 960 . . 992 . . 1024 . . ps

Console

```
o # KERNEL: data memory:
o # KERNEL: |address 0: 0000000000000001|
o # KERNEL: |address 1: 0000000000000010|
o # KERNEL: |address 2: 0000000000000011|
o # KERNEL: |address 3: 0000000000000100|
o # KERNEL: |address 4: xxxxxxxxxxxxxxxxx|
o # KERNEL: |address 5: xxxxxxxxxxxxxxxxx|
o # KERNEL: |address 6: xxxxxxxxxxxxxxxxx|
o # KERNEL: |address 7: xxxxxxxxxxxxxxxxx|
o # KERNEL: |address 8: 0000000000001010|
o # KERNEL: |address 9: xxxxxxxxxxxxxxxxx|
o # KERNEL: |address 10: xxxxxxxxxxxxxxxxx|
o # KERNEL: |address 11: xxxxxxxxxxxxxxxxx|
o # KERNEL: |address 12: xxxxxxxxxxxxxxxxx|
o # KERNEL: |address 13: xxxxxxxxxxxxxxxxx|
o # KERNEL: |address 14: xxxxxxxxxxxxxxxxx|
o # KERNEL: |address 15: xxxxxxxxxxxxxxxxx|
o # KERNEL:
o # KERNEL: register file:
o # KERNEL: |register 0: 0000000000000000|
o # KERNEL: |register 1: 0000000000001010|
o # KERNEL: |register 2: 0000000000000100|
o # KERNEL: |register 3: 0000000000000100|
o # KERNEL: |register 4: 0000000000000000|
o # KERNEL: |register 5: 0000000000000011|
o # KERNEL: |register 6: 0000000000000100|
o # KERNEL: |register 7: xxxxxxxxxxxxxxxxx|
o # RUNTIME: Info: RUNTIME_0068 MultiCycle.v (1235): $finish called.
o # KERNEL: Time: 1325 ps, Iteration: 0, Instance: /multicycleDataPath_tb, Process: @INITIAL#1222_21@.
o # KERNEL: stopped at time: 1325 ps
```

20°C Mostly clear 3:58 AM 6/19/2024

➤ **Passed Test**

- **Covers:** ADDI, BEQ, ADD, LW, JMP, SW

2.2 Test2:

Sum the array values in the memory. Then uses much operations and check values after operations.

- **Expected output:**
 - DataMem [5] = 75
 - R [2] = -1
 - R [6] = -1

- **Instruction Memory:**

```
• // load the base address of the array into $R1
• memory[0] = 16'b0011000100000000; // addi $R1, $R0, array = 0
•
• // initialize sum to 0
• memory[1] = 16'b0011001000000000; //addi $R2, $R0, 0
• memory[2] = 16'b0111001000000101; //sw $R2, 5($R0)
•
• // loop through the array and sum the elements
• memory[3] = 16'b0011001100000101; // addi $R3, $R0, 5 # length of the array
• memory[4] = 16'b0011010000000000; //addi $R4, $R0, 0 # loop counter
•
• //loop:
• memory[5] = 16'b1010010001101100; //loop: beq $R4, $R3, end_loop(mem[11]) # if counter equals array length, end loop
• memory[6] = 16'b0101010100100000; // lw $R5, 0($R1) # load current array element into $R5
• memory[7] = 16'b0001010010101000; // add $R2, $R2, $R5 # add element to sum
• memory[8] = 16'b0011000100100001; // addi $R1, $R1, 1 # move to the next array element
• memory[9] = 16'b0011010010000001; // addi $R4, $R4, 1 # increment loop counter
• memory[10] = 16'b1100000000000101; // jMP loop # jump back to the beginning of the loop
• memory[11] = 16'b0111001000000101; //sw $R2, 5 # Store the final sum
•
• memory[12] = 16'b0011010000001011; //addi $R4, $R0, 11 # First argument
• memory[13] = 16'b0011010100001110; //addi $R5, $R0, 14 # Second argument
• memory[14] = 16'b1101000000010011; //call logical_operations # Call the function
• memory[15] = 16'b0; // since you asked to add 4 to the return address when using call an instruction is skipped here so no matter what
it holds it does not affect the code. if the added value is 2 no skip needed
• memory[16] = 16'b0011010100010100; //addi $R5, $R0, 20 # Second argument
```

```

• // store the result of the logical operations
• memory[17] = 16'b0011011001000000; //addi, $R6, $R2,0
• memory[18] = 16'b1100001000000000; // here the jum must jump to an address that does not exist to simulate system call of exit
•
•
• // function to perform logical operations and comparisons
• memory[19] = 16'b0000110100101000; //and $R6, $R4, $R5 Perform AND operation //logical_operations:
• memory[20] = 16'b0011011011000101; //addi $R6, $R6, 5 # Add immediate to result
• memory[21] = 16'b0011000110011111; //addi $R1, $R4, -1 # Subtract immediate from argument
• memory[22] = 16'b0100000100111101; // andi $R1, $R1, 29 # AND immediate with result
•
•
• // # Perform some comparisons
• memory[23] = 16'b1000010010101000 ; //bgt $R4, $R5, greater # If $R4 > $R5, branch to greater
• memory[24] = 16'b1001010010101010; // blt $R4, $R5, lesser # If $R4 < $R5, branch to lesser
• memory[25] = 16'b1010010010101100; // beq $R4, $R5, equal # If $R4 == $R5, branch to equal
• memory[26] = 16'b1011010010101110; // bne $R4, $R5, notequal # If $R4 != $R5, branch to notequal
• memory[27] = 16'b0011001000000001; // greater: addi $R2, $R0, 1 # Set return value to 1
• memory[28] = 16'b1100000000100000; // j end_func
• memory[29] = 16'b0011001000011111; //lesser : addi $R2, $R0, -1 # Set return value to -1
• memory[30] = 16'b1100000000100000; // j end_func
• memory[31] = 16'b0011001000000000; // equal: ddi $R2, $R0, 0 # Set return value to 0
• memory[32] = 16'b1100000000100000; // j end_func
• // call the function to perform logical operations

```

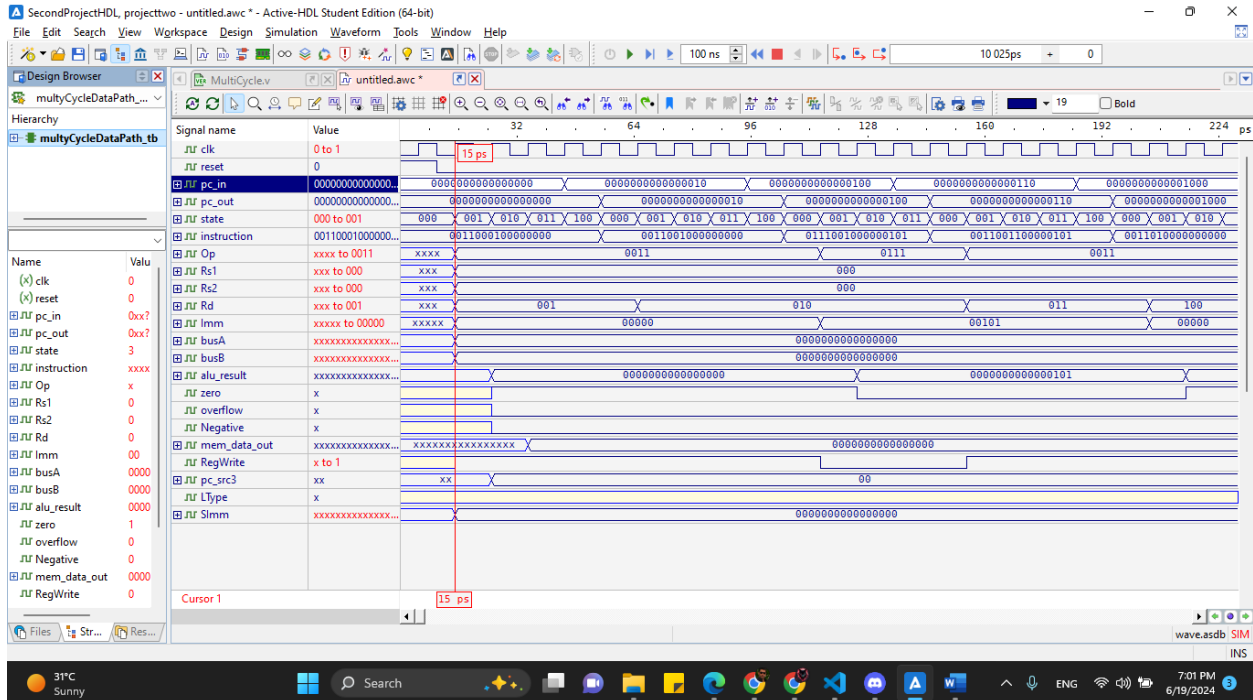
• Data Memory:

```

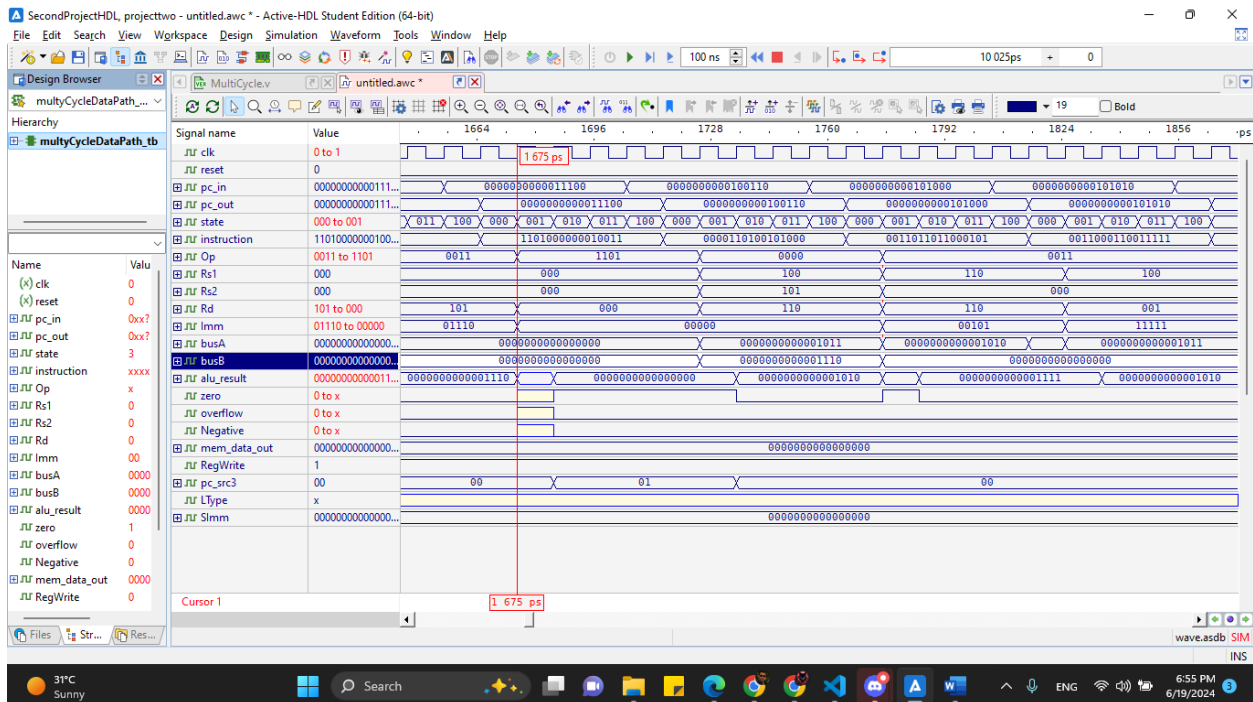
• // array values
• memory[0] = 16'b0000000000000101; // 5 array @ 0
• memory[1] = 16'b0000000000001010; // 10
• memory[2] = 16'b0000000000001111; // 15
• memory[3] = 16'b0000000000010100; // 20
• memory[4] = 16'b000000000011001; // 25
•
•
• // other values
• memory[5] = 16'b0000000000000000; // sum @ 5
• memory[6] = 16'b0000000000000000; // temp @ 6

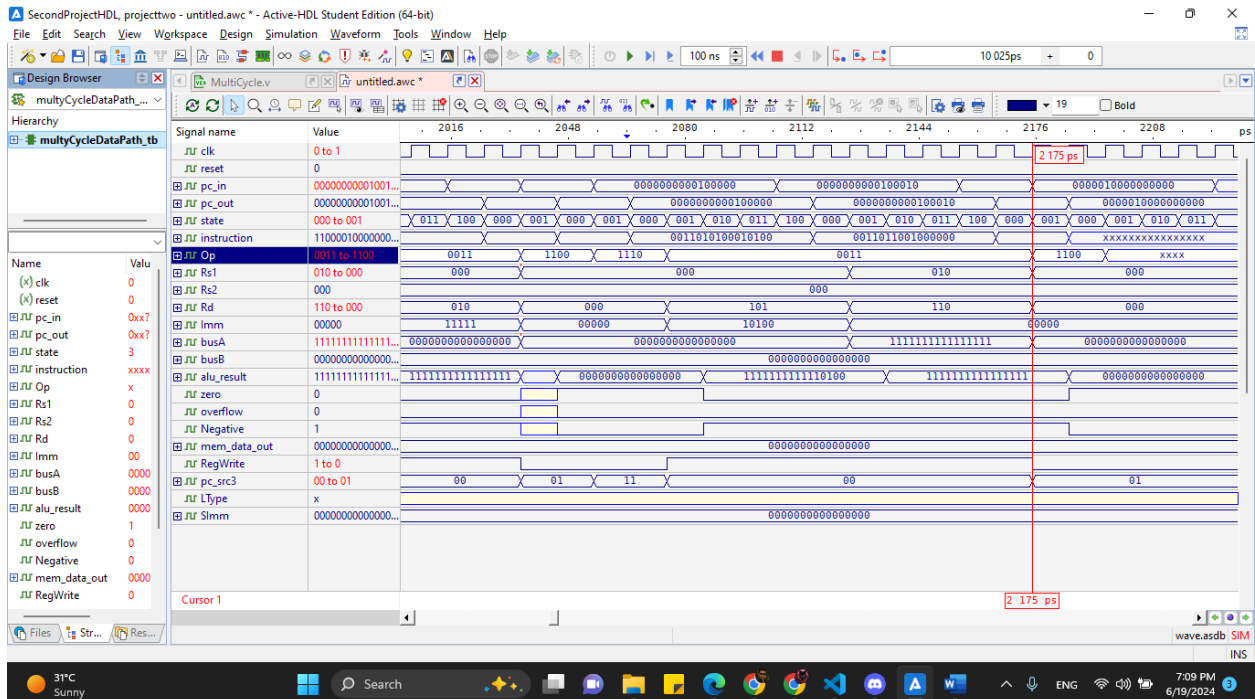
```

- **Simulation:**



- ❖ **Call:**





- **Result:**

```

# KERNEL:
# KERNEL: data memory:
# KERNEL: |address 0: 000000000000101|
# KERNEL: |address 1: 0000000000001010|
# KERNEL: |address 2: 0000000000001111|
# KERNEL: |address 3: 0000000000010100|
# KERNEL: |address 4: 000000000011001|
# KERNEL: |address 5: 0000000001001011|
# KERNEL: |address 6: 0000000000000000|
# KERNEL: |address 7: xxxxxxxxxxxxxxxxx|
# KERNEL: |address 8: xxxxxxxxxxxxxxxxx|
# KERNEL: |address 9: xxxxxxxxxxxxxxxxx|
# KERNEL: |address 10: xxxxxxxxxxxxxxxxx|
# KERNEL: |address 11: xxxxxxxxxxxxxxxxx|
# KERNEL: |address 12: xxxxxxxxxxxxxxxxx|
# KERNEL: |address 13: xxxxxxxxxxxxxxxxx|
# KERNEL: |address 14: xxxxxxxxxxxxxxxxx|
# KERNEL: |address 15: xxxxxxxxxxxxxxxxx|
# KERNEL:
# KERNEL: register file:
# KERNEL: |register 0: 0000000000000000|
# KERNEL: |register 1: 0000000000001000|
# KERNEL: |register 2: 1111111111111111|
# KERNEL: |register 3: 0000000000000101|
# KERNEL: |register 4: 0000000000001011|
# KERNEL: |register 5: 1111111111010100|
# KERNEL: |register 6: 1111111111111111|
# KERNEL: |register 7: 0000000000100000|
# RUNTIME: Info: RUNTIME_0068 Multicycle.v (1279): $finish called.
# KERNEL: Time: 10025 ps, Iteration: 0, Instance: /multicycleDataPath_tb, Process: @INITIAL#1266_21@.
> # KERNEL: |register 2: 1111111111111111| # KERNEL: |register 2: 1111111111111111|

```

➤ **Passed Test**

- **Covers:** ADDI, ADD, LW, JMP, SW, CALL, BEQ, BGT, BLT, BNE, RET, ANDI, AND

2.3 Test3:

Load signed and unsigned values subtract them compare the result with zero and store value according to the competition result.

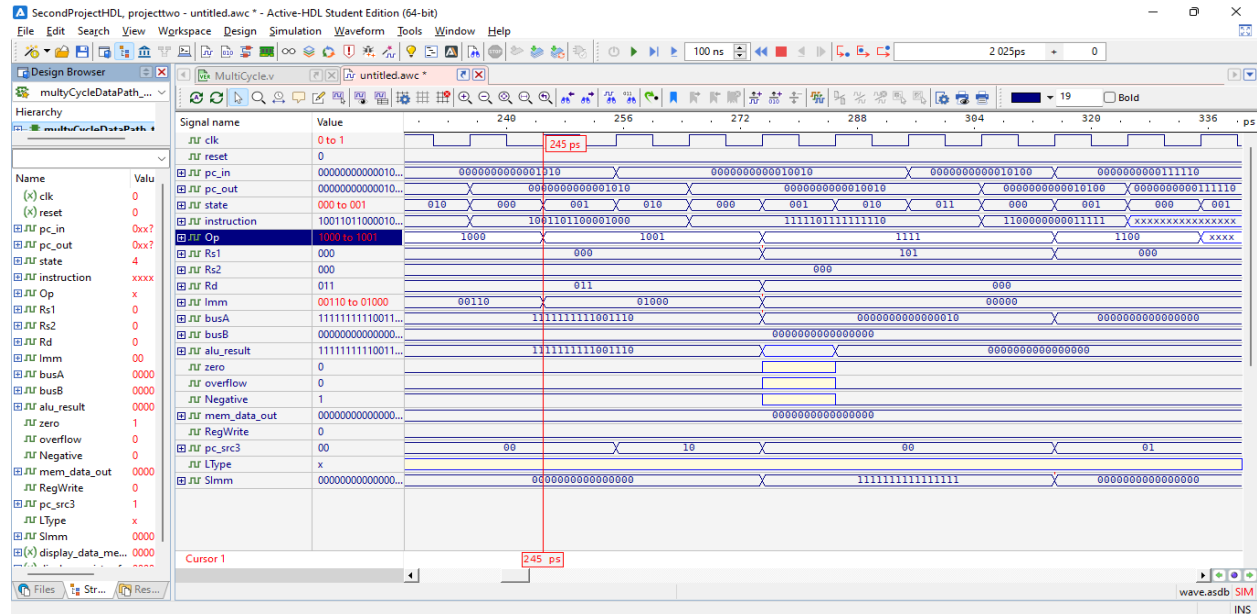
- **Expected output:**
 - **Greater:**
DataMem [2] = 1
 - **Less:**
DataMem [2] = -1
 - **Equal:**
DataMem [2] = 0
- **Instruction Memory:**

```
• // load data
• memory[0] = 16'b0110000100000000; // LBU $R1, 0($R0) load unsigned byte 100
• memory[1] = 16'b0110101000000001; // LBS $R2, 1($R0) load signed byte -50
•
• memory[2] = 16'b0011010100000010; // ADDI $R5,$R0,2
•
• // subtract data
• memory[3] = 16'b0010011001010000; // SUB $R3,$R1,$R2
•
• // check result
• memory[4] = 16'b1000101100000110; // BGTZ $R3,7
• memory[5] = 16'b1001101100001000; // BLTZ $R3,9
• memory[6] = 16'b1010101100001010; // BEZ $R3,11
•
• // greater
• memory[7] = 16'b111110100000010; // SV $R5 ,1
• memory[8] = 16'b110000000011111; // JMP
•
• // less
• memory[9] = 16'b111110111111110; // SV $R5 ,-1
• memory[10] = 16'b110000000011111; // JMP
•
• // equal
• memory[11] = 16'b111110100000000; // SV $R5 ,0
```

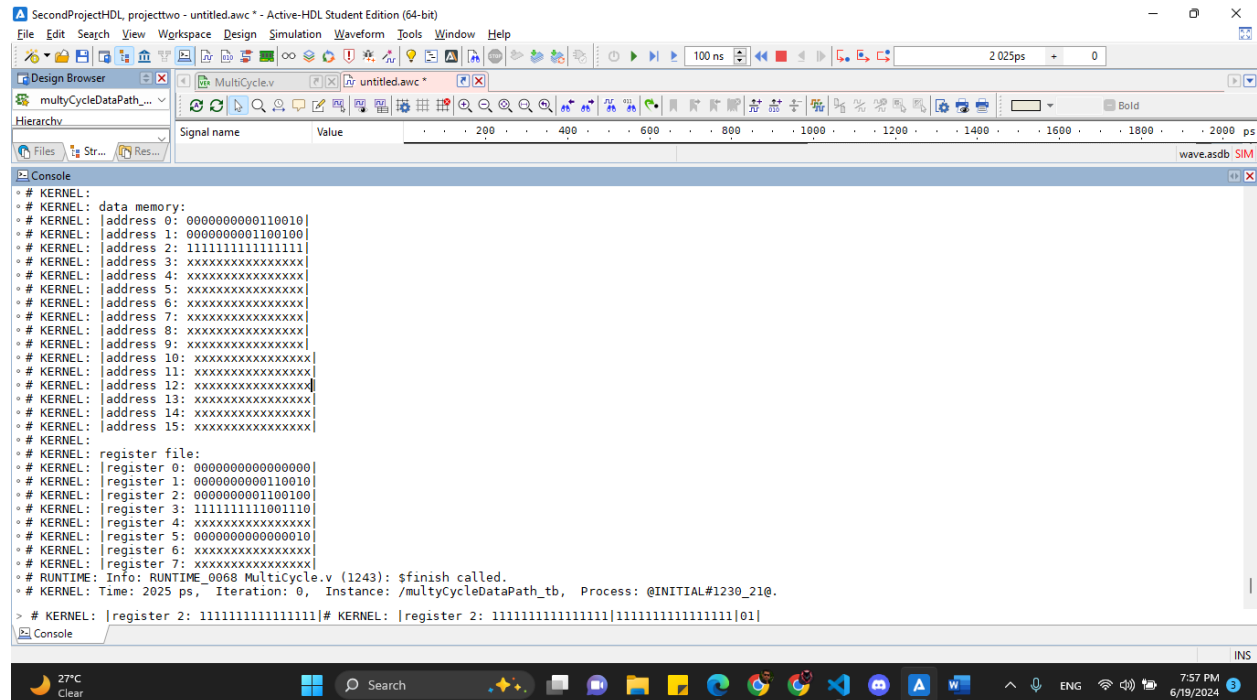
• Data Memory attempt1: (LESS)

```
// less
memory[0] = 16'b000000000110010; // 50
memory[1] = 16'b000000000110010; // 100
```

• Simulation:



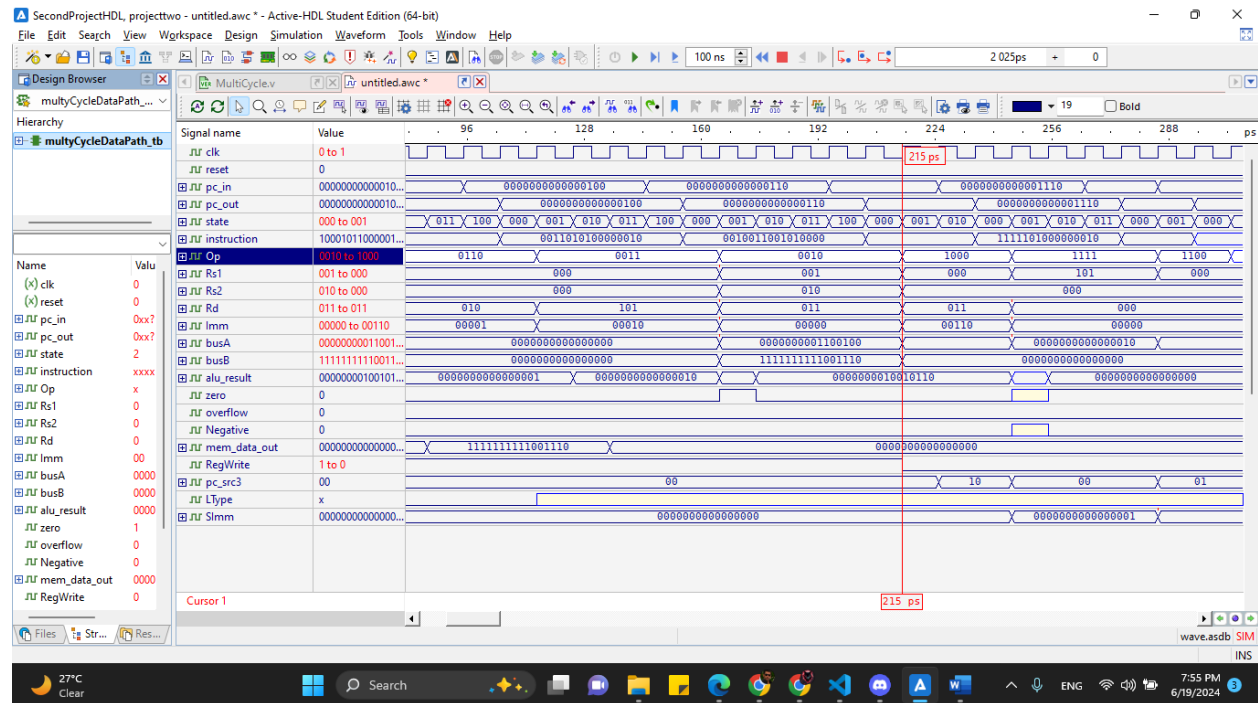
• Result:



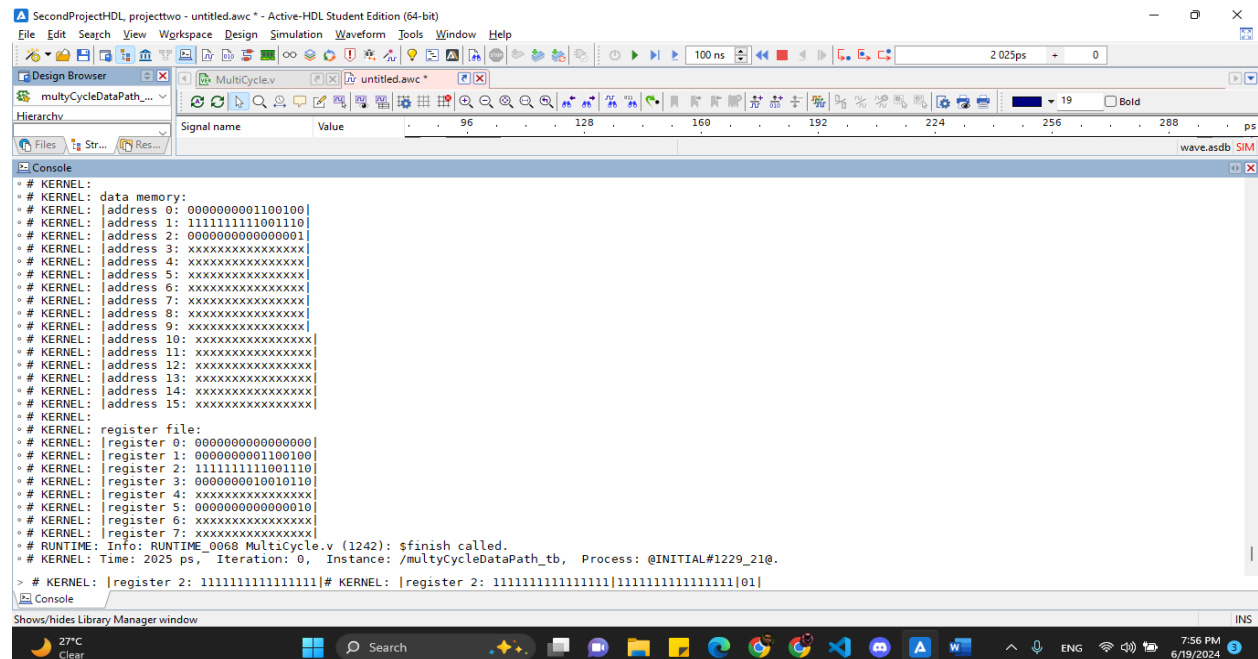
- **Data Memory attempt2: (Greater)**

```
// greater
memory[0] = 16'b0000000001100100; // 100
memory[1] = 16'b1111111111001110; // - 50
```

- **Simulation:**



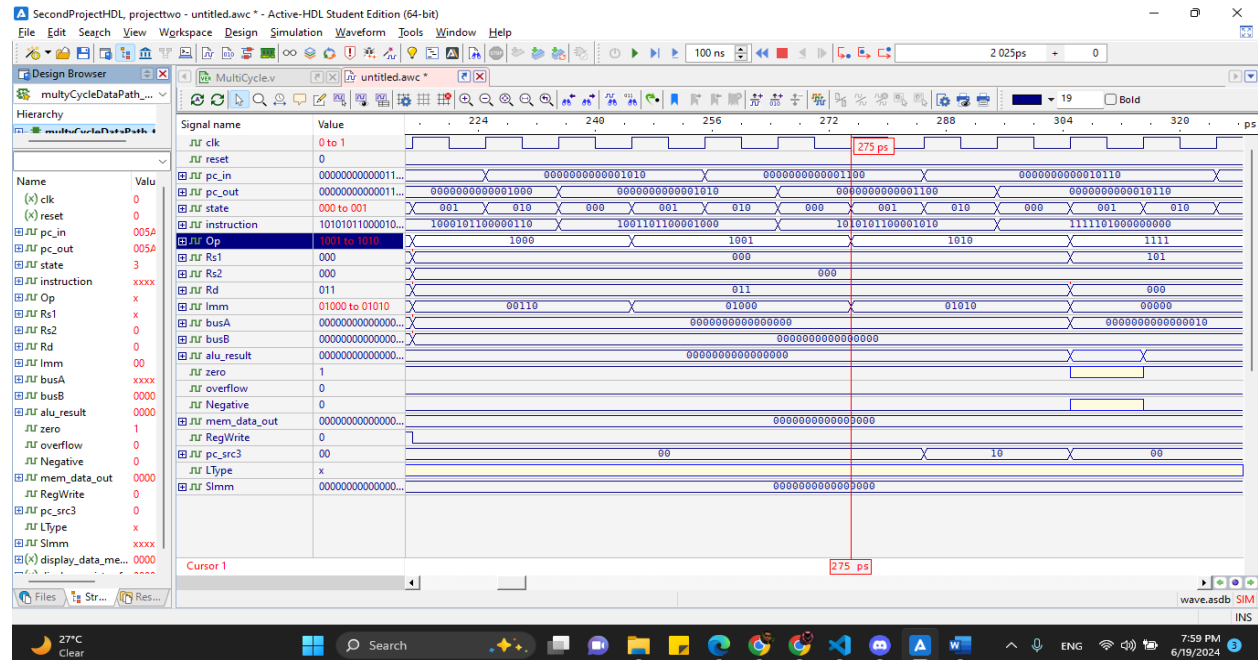
- **Result:**



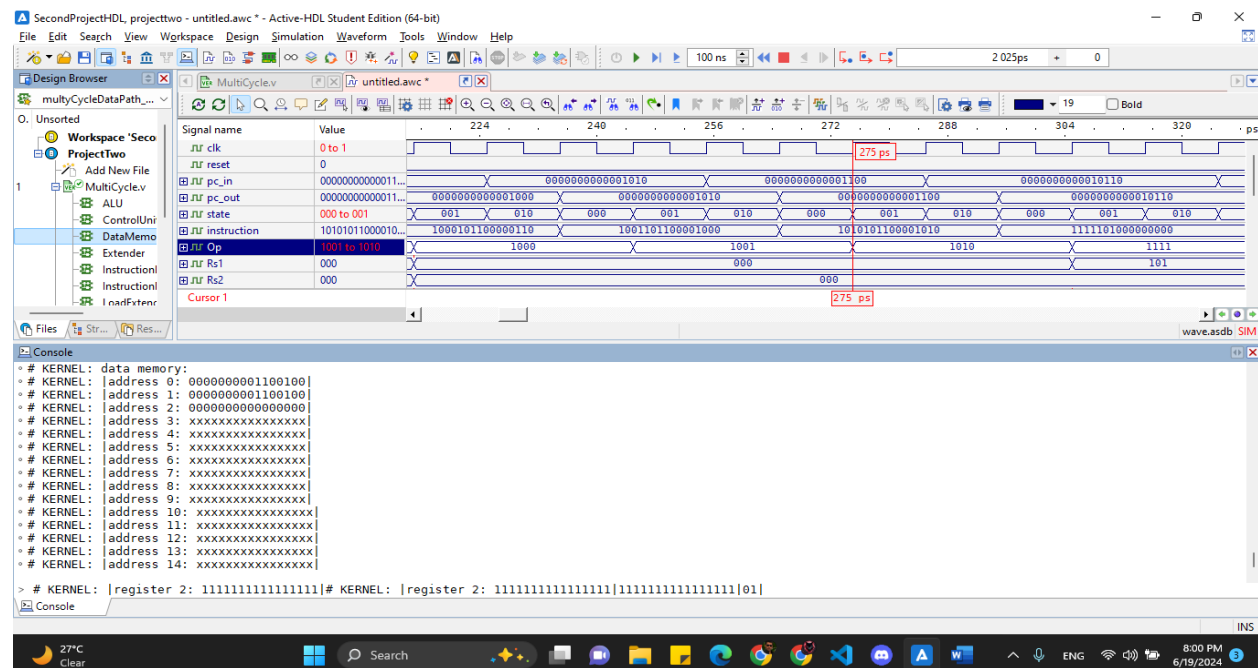
- **Data Memory attempt2: (Equal)**

```
// equal
memory[0] = 16'b000000001100100; // 100
memory[1] = 16'b000000001100100; // 100
```

- **Simulation:**



- **Result:**



➤ **Passed Test**

- **Covers:** LBU, LBS, ADDI, SUB, JMP, SV, BEZ, BGTZ, BLTZ

3. Team Work:

3.1 Data path design

We used a shared web service (lucid chart) to collaborate in designing and deciding needed parts and components. so, we worked completely together. also, if a change was needed on the data path design, we were to make sure of that change together.

3.2 Components implementations using Verilog

we distributed creating components between us like the following:

Abdelrahman Shahren:

- Instruction Memory
- Instruction Decoder
- PC
- PC Control
- Mini PC Control

Mahmoud Awad:

- ALU
- Data Memory
- Extender
- Load Extender
- Register File

for the control unit we worked together since it needs more attention. and 2 minds is better than 1 in this type of components. After each one of us finished his components, we reviewed each of our components together to make sure everything is right.

multicycle Datapath implementation using Verilog

In the implementation of the Datapath we worked together to make sure every connection and every wire is in its place.

3.3 Multicycle Datapath test bench implementation using Verilog

here also we made the test bench together.

3.4 Writing assembly code for the given Isa

we came with the ideas and worked into translating the idea into machine code together so we were able to bring 3 ideas (**that covers all the instructions**) to machine code so they could be executed in our processor. So, we collaborated into creating instruction memory values and data memory values for these 3 ideas together.

3.5 Testing our codes

We ran our processor with the corresponding test values and checked expected stored values and the results were amazing everything went as expected. we used share screen on discord to work together all the time.

Mahmoud Awad tested with ActiveHDL software to get the wave results

Abdelrahman Shahan tested with VS code for decoding the output and signals

3.6 report writing

we worked together to write a good and clear report. we have used real time Microsoft office Word online to share the same document and write on it at the same time

So, we have both worked as needed to finish this task. we have used many real time applications.

- **Discord:** For communicating and screen sharing and files sharing.
- **LucidChart:** For design the Datapath and its assembly.
- **VSCode:** To write the shared codes together.
- **MicrosoftWord:** To write a clear and understandable report.

(احنا التنين اشتغلنا بحق الله)