Final Project: 16 Bit MIPS CPU Group (20)

Muhammad Shayan (1145894)
Zain Siddiqui (1228533)
Danial Changez (123241)

Computer Organization and Design
ENGG 3380
March 25th, 2024

Table of Contents

Problem Statement: 4

Assumptions and Constraints 4

System Overview (PART A) 5

- Program Counter 5
- Instruction Memory 5
- Control Module 6
- Sign Extend 7
- Data Memory 8

Instruction Tracing (PART A) 8

- Instruction 1: ADDI R3, R0, 5 9
- Instruction 2: ADDI R4, R0, 2 10
- Instruction 3: SLT R11, R3, R4 11
- Instruction 4: SW R3, 0(R0) 12
- Instruction 5: SW R4, 4(R0) 13
- Instruction 6: ADDI R6, R0, 4 14
- Instruction 7: LW R7, 0(R6) 15
- Instruction 8: LW R8, 0(R0) 16
- Instruction 9: ADD R9, R7, R8 17
- Instruction 10: SLT R10, R0, R1 18
- Instruction 11: SLT R10, R1, R0 19
- Instruction 12: OR R5, R10, R9 20
- Instruction 13: SUBI R10, R5, 7 21
- Instruction 14: SUB R11, R10, R7 22
- Instruction 15: SW R11, 5(R8) 23

Verification (PART A) 25

Waveform Part A (PC 1-20) 26

Waveform Part A (PC 14-32) 27

System Overview (PART B) 29

- BNE Block 29
  - Branch if Not Equal VHDL 30
- Jump Block 31

Jump VHDL 32
PC Control MUX 32
3_1 MUX VHDL 33
Instruction Tracing (PART B) 33
BNE Instruction: BNE r8, r0, 2 33
Jump Instruction: Jump 14 35
Verification (PART B) 36
Waveform Part B (PC 0 – 22) 37
Waveform Part B (PC 8 – 28) 38
Errors and Issues 39
Conclusion 40

Problem Statement:

The goal of this project is to compile all the CPU subsystems made in the previous labs into one hierarchal CPU module. The 16-bit MIPS CPU can handle many input interactions and is able to store the results into memory or a register file. This CPU will also be able to handle R, I and J instructions alongside BNE instructions based on the opcode provided. The demonstration is to be presented in the lab session, and the chart will be filled out using a testbench to stimulate the CBU's behavior.

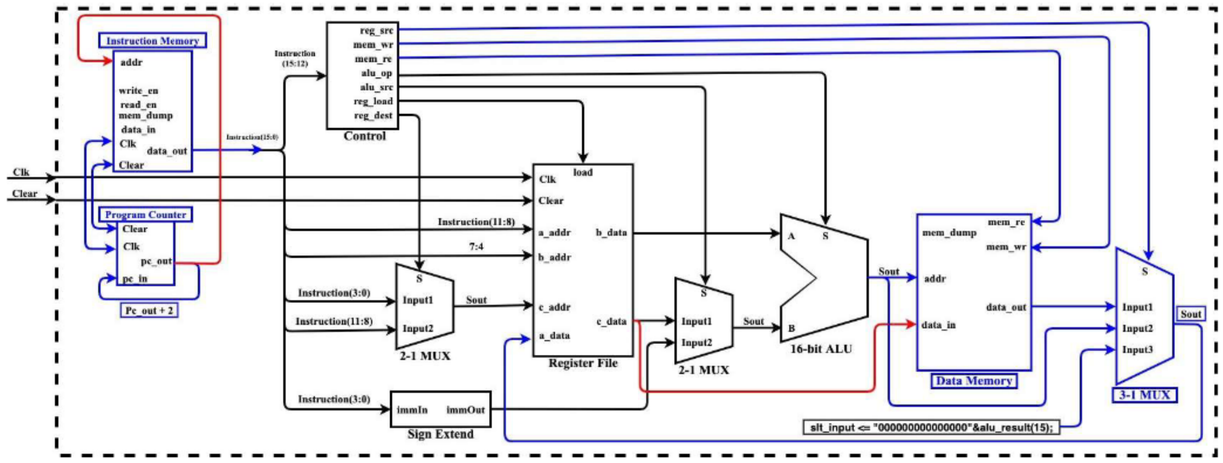


Figure 1: Part A CPU

Assumptions and Constraints

- We must ensure all components function correctly as specified.
- We need establish an initial memory paradigm, distinguishing between registers and data memory.
- We must read 16-bit instructions from "Instr.txt" (Part A) and "Instr2.txt" (Part B), load them into instruction memory, and execute them sequentially at each clock cycle.
- A constraint was that we were not given the jump and BNE block, so it required more time and effort to understand them
- We must assume that we need to use an additional 3-1 MUX for the additions of the BNE Block and the JUMP block
- Assuming we must only include one instruction for the BNE and Jump in the instruction tracing for part 2

System Overview (PART A)

Program Counter

The program counter is a component responsible for instruction sequencing. It increments its binary value by two bits after each instruction execution, allowing it to maintain the CPU's program state. On the rising edge of the clock, the PC feeds its current address to the instruction memory. This address acts as an index, prompting the instruction memory to fetch the corresponding instruction. The fetched instruction is then sent to other CPU units accordingly. The clock and clear signals are mapped directly to the system's clock and clear functionality, allowing synchronized execution.

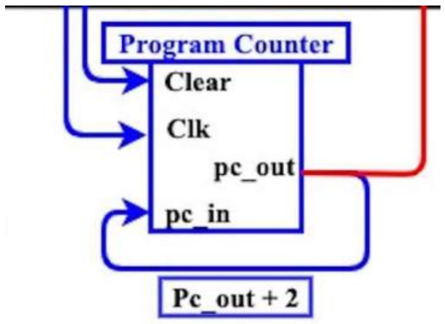


Figure 2: Program Counter

Instruction Memory

The instruction memory is a component that contains a set of 16 instructions. These dictate the control signals for multiplexors as well as enable pins in the CPU. The instruction memory's contents are pre-loaded from the file, "Instr.txt". This read-only memory ensures the instructions cannot be tampered with.

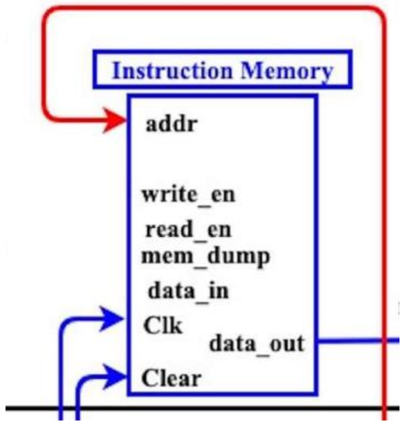


Figure 3: Instruction Memory

Control Module

The control module decodes a 4-bit opcode into its respective control signals to provide the CPU with the relevant instructions to execute its take. It determines the operations to occur on registers, data memory, and if any ALU operation will be performed.

It decided the instruction format, whether R or I, or if a SLT instruction is executed. The control signals have been summarized below in detail.

1. **reg_src (Register Source):**
   - This 2-bit signal governs the 3-to-1 multiplexer responsible for selecting data to be written back to the register file.
     - **"00"**: Data retrieved from memory is chosen.
     - **"01"**: Output from the ALU takes center stage.
     - **"10"**: A specialized signal, potentially indicating a result based on a comparison operation (e.g., "less than").

2. **mem_wr (Memory Write):**
   - This single-bit signal acts as a switch, controlling write operations to the data memory (active high signifies writing).

3. **mem_re (Memory Read):**
   - Similar to MEM_WR, this 1-bit signal flips the switch for reading data from the memory (active high signifies reading).

4. **alu_op (ALU Operation):**
   - A 4-bit signal dictating the specific operation the ALU performs, such as addition, subtraction, or bitwise operations.

5. **reg_load (Register Load):**
   - This 1-bit signal controls the writing functionality of the register file (active high enables writing).

6. **reg_dest (Register Destination):**
   - A 1-bit signal with context-dependent functionality. It defines the destination register address in some instruction formats (e.g., SW stores to memory), while being irrelevant in others (e.g., I-type instructions).

ALU_OP	Operation
0000	ADD
0001	SUB
0010	AND
0011	OR
0100	ADDI
0101	SUBI
1000	LW
1100	SW
0111	SLT

Figure 4 ALU Opcode Table

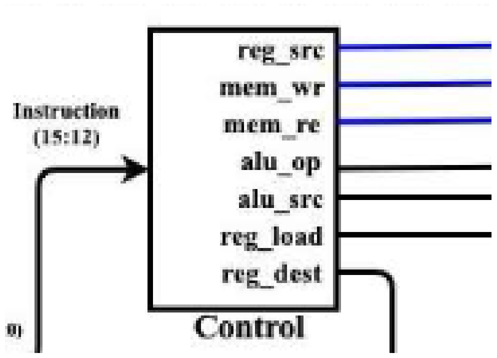


Figure 5 Control Module

Sign Extend

The sign extend block is used in I-type instruction formats, specifically for extending 4-bit unsigned immediate values to their 16-bit equivalents by sign extension. This process involves reading the most significant bits and extending it by an additional 12 bits, resulting in an output that matches the input's value while ensuring proper representation within the extended bit range.

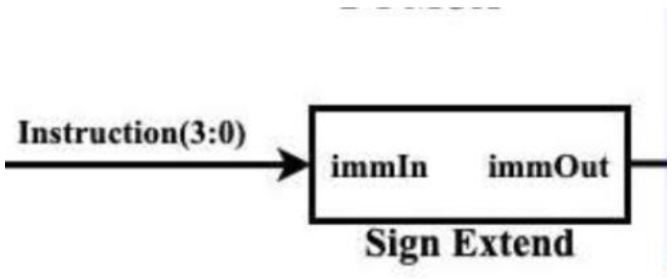


Figure 6 Sign Extend Block

Data Memory

The data memory module is composed of a 2D array with 34 rows, each containing 7 bits. Its main function is to store data and retrieve results processed by the ALU. The mem_re and mem_wr enable pins play an important role in controlling whether data is written to or read from the memory module.

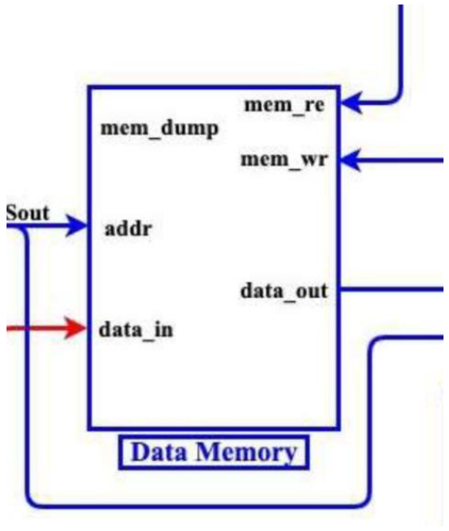


Figure 7 Data Memory Block

Instruction Tracing (PART A)

To prove we receive the correct results in our testbench, we must first understand how the CPU will process each instruction by following the data's expected path. Then we can compare with the testbench signals, and confirm they match the expected values. The general instruction path is outlined below:

1. **Fetch:** The program counter (PC) points to the instruction memory address containing the relevant instruction.

2. **PC Update:** The PC is incremented to point to the next instruction (unless a branch or jump instruction changes the flow, but those have not been implemented in this part).

3. **Decode:** The instruction is broken down into its parts and sent to the relevant units. The opcode goes to the control unit.

4. **Control Signal Generation:** The control unit decodes the opcode and sends control signals to other units based on the instruction type.

5. **Operand Fetch:** The register file is accessed using the source register address (Rs) to retrieve the first operand (if the instruction requires one).

6. **Immediate Value Sign Extend:** If the instruction uses an immediate value, it's sign-extended to the same width as the other operand.

7. Depending on the instruction, one of the 3 following operations would occur.

- **ALU Operation:** The ALU receives the operands and performs the operation specified by the control signals (if the relevant instructions require it to).

- **Set Less Than (SLT) Operation:** The most significant bit of the result from the ALU would be concatenated with 15 zeroes.

- **Data Memory Operation:** A word is stored or loaded into the data memory.

8. **Result Write-back:** The result from the previous operation is written back to the register file at the address specified by the destination register (Rd), assuming it is not a data memory operation.

Instruction 1: ADDI R3, R0, 5

Function: Adds the contents of register 0 ($R0) to the immediate value 5 and stores the result in register 3 ($R3).

1. **Fetch**:
   - Program Counter (PC) points to the instruction memory address containing the ADDI instruction.

2. **PC Update**:
   - PC is incremented by 2 to point to the next instruction in the sequence.

3. **Decode**:
   - Instruction is broken down into its components:
     - Opcode (15:12) sent to the control module.
     - Destination register (Rd) (11:8) sent to a_addr of the register file.
     - Source register (Rs) (7:4) sent to b_addr of the register file.
     - Immediate value (4-bit) (3:0) sent to the sign extend block.

4. **Control Signal Generation**:
   - Control unit decodes the opcode and generates control signals:

- reg_src = 01 (select ALU output for register file write back).
- mem_wr = 0 (disable memory write).
- mem_re = 0 (disable memory read).
- alu_op = 00 (perform addition).
- alu_src = 1 (use immediate value for ALU input B).
- reg_load = 1 (enable register write).

5. **Operand Fetch**:
   - Contents of register 0 ($R0) are retrieved from b_addr of the register file and sent to input A of the ALU.

6. **Immediate Value Sign Extension**:
   - 4-bit immediate value (5) is sign-extended to 16 bits and sent to input B of the ALU.

7. **ALU Operation**:
   - ALU performs addition on input A (value from register 0) and input B (sign-extended immediate value).

8. **Result Write-back**:
   - 16-bit result from the ALU is written back to register 3 ($R3) in the register file.

Instruction 2: ADDI R4, R0, 2

Function: Adds the contents of register 0 ($R0) to the immediate value 2 and stores the result in register 4 ($R4).

1. **Fetch:**
   - PC points to the instruction memory address containing the ADDI instruction.

2. **PC Update:**
   - PC is incremented by 2 to point to the next instruction.

3. **Decode:**
   - Instruction is broken down into its components (opcode, Rd, Rs, and immediate value).

4. **Control Signal Generation:**
   - Control unit decodes the opcode and generates control signals:
     - reg_src = 01 (select ALU output for register file write back)
     - mem_wr = 0 (disable memory write)
     - mem_re = 0 (disable memory read)
     - alu_op = 00 (perform addition)
     - alu_src = 1 (use immediate value for ALU input B)
     - reg_load = 1 (enable register write)

5. **Operand Fetch:**
   - Contents of register 0 ($R0) are retrieved from the register file and sent to input A of the ALU.

6. **Immediate Value Sign Extension:**
   - 4-bit immediate value (2) is sign-extended to 16 bits and sent to input B of the ALU.

7. **ALU Operation:**
   - ALU performs addition on input A (value from register 0) and input B (sign-extended immediate value).

8. **Result Write-back:**
   - 16-bit result from the ALU is written back to register 4 ($R4) in the register file.

Instruction 3: SLT R11, R3, R4

Function: Compares the contents of register 3 ($R3) and register 4 ($R4) by subtracting them. Sets the least significant bit (LSB) of register 11 ($R11) to 1 if $R3 is less than $R4, and 0 otherwise.

1. **Fetch:**
   - PC points to the instruction memory address containing the SLT instruction.

2. **PC Update:**
   - PC is incremented by 2 to point to the next instruction.

3. **Decode:**
   - Instruction is broken down into its components (opcode, Rd, Rs, and Rt).

4. **Control Signal Generation:**
   - Control unit decodes the opcode and generates control signals:
     - reg_src = 10 (select subtraction result for register file write back)
     - mem_wr = 0 (disable memory write)
     - mem_re = 0 (disable memory read)
     - alu_op = 01 (perform subtraction)
     - alu_src = 0 (use register Rt value for ALU input B)
     - reg_load = 1 (enable register write)

5. **Operand Fetch:**
   - Contents of register 3 ($R3) and register 4 ($R4) are retrieved from the register file and sent to input A and B of the ALU, respectively.

6. **ALU Operation:**
   - ALU performs subtraction on input A ($R3) and input B ($R4).

7. **SLT Logic:**
   - The Most Significant Bit (MSB) of the subtraction result is extracted.

8. **Result Write-back:**
   - The MSB is concatenated with 15 zeros, and the entire 16-bit result is written back to register 11 ($R11) in the register file.

Instruction 4: SW R3, 0(R0)

Function: Stores the contents of register 3 ($R3) into the memory location calculated by adding the contents of register 0 ($R0) and an offset of 0.

1. **Fetch:**
   - PC points to the instruction memory address containing the SW instruction.

2. **PC Update:**
   - PC is incremented by 2 to point to the next instruction.

3. **Decode:**
   - Instruction is broken down into its components (opcode, Rd, Rs, and offset).

4. **Control Signal Generation:**
   - Control unit decodes the opcode and generates control signals:
     - reg_src = XX (irrelevant for this instruction)
     - mem_wr = 1 (enable memory write)
     - mem_re = 0 (disable memory read)
     - alu_op = 00 (perform addition)
     - alu_src = 1 (use immediate value for ALU input B)
     - reg_load = 0 (disable register write)

5. **Operand Fetch:**
   - Contents of register 0 ($R0) are retrieved from the register file and sent to input A of the ALU.

6. **Memory Address Calculation:**
   - The 4-bit offset (0) is sign-extended to 16 bits.
   - The ALU adds the sign-extended offset to the value from $R0, producing the memory address.

7. **Data Fetch:**
   - The value in register 3 ($R3) is retrieved from the register file.

8. **Memory Write:**
   - The value from register 3 ($R3) is written to the calculated memory address.

Instruction 5: SW R4, 4(R0)

Function: Stores the contents of register 4 ($R4) into the memory location calculated by adding the contents of register 0 ($R0) and an offset of 4.

1. **Fetch:**
   - PC points to the instruction memory address containing the SW instruction.

2. **PC Update:**
   - PC is incremented by 2 to point to the next instruction.

3. **Decode:**
   - Instruction is broken down into its components (opcode, Rd, Rs, and offset).

4. **Control Signal Generation:**

- Control unit decodes the opcode and generates control signals:
  - reg_src = XX (irrelevant for this instruction)
  - mem_wr = 1 (enable memory write)
  - mem_re = 0 (disable memory read)
  - alu_op = 00 (perform addition)
  - alu_src = 1 (use immediate value for ALU input B)
  - reg_load = 0 (disable register write)

5. **Operand Fetch:**
   - Contents of register 0 ($R0) are retrieved from the register file and sent to input A of the ALU.

6. **Memory Address Calculation:**
   - The 4-bit offset (4) is sign-extended to 16 bits.
   - The ALU adds the sign-extended offset to the value from $R0, producing the memory address.

7. **Data Fetch:**
   - The value in register 4 ($R4) is retrieved from the register file.

8. **Memory Write:**
   - The value from register 4 ($R4) is written to the calculated memory address.

Instruction 6: ADDI R6, R0, 4

Function: Adds the contents of register 0 ($R0) to the immediate value 4 and stores the result in register 6 ($R6).

1. **Fetch:**
   - PC points to the instruction memory address containing the ADDI instruction.

2. **PC Update:**
   - PC is incremented by 2 to point to the next instruction.

3. **Decode:**
   - Instruction is broken down into its components (opcode, Rd, Rs, and immediate value).

4. **Control Signal Generation:**
   - Control unit decodes the opcode and generates control signals:
     - reg_src = 01 (select ALU output for register file write back)
     - mem_wr = 0 (disable memory write)
     - mem_re = 0 (disable memory read)
     - alu_op = 00 (perform addition)
     - alu_src = 1 (use immediate value for ALU input B)
     - reg_load = 1 (enable register write)
5. **Operand Fetch:**
   - Contents of register 0 ($R0) are retrieved from the register file and sent to input A of the ALU.

6. **Immediate Value Sign Extension:**
   - The 4-bit immediate value (4) is sign-extended to 16 bits and sent to input B of the ALU.

7. **ALU Operation:**
   - ALU performs addition on input A (value from $R0) and input B (sign-extended immediate value).

8. **Result Write-back:**
   - The 16-bit result from the ALU is written back to register 6 ($R6) in the register file.

Instruction 7: LW R7, 0(R6)

Function: Loads the contents from memory location calculated by adding the contents of register 6 ($R6) and an offset of 0 and stores the result in register 7 ($R7).

1. **Fetch:**
   - PC points to the instruction memory address containing the LW instruction.

2. **PC Update:**
   - PC is incremented by 2 to point to the next instruction.

3. **Decode:**
   - Instruction is broken down into its components (opcode, Rd, Rs, and offset).

4. **Control Signal Generation:**
   - Control unit decodes the opcode and generates control signals:

        - reg_src = 00 (select data from memory for register file write back)
        - mem_wr = 0 (disable memory write)
        - mem_re = 1 (enable memory read)
        - alu_op = 00 (perform addition)
        - alu_src = 1 (use immediate value for ALU input B)
        - reg_load = 1 (enable register write)

5. **Operand Fetch:**
    - Contents of register 6 ($R6) are retrieved from the register file and sent to input A of the ALU.

6. **Memory Address Calculation:**
    - The 4-bit offset (0) is sign-extended to 16 bits.
    - The ALU adds the sign-extended offset to the value from $R6, producing the memory address.

7. **Memory Read:**
    - The data at the calculated memory address is fetched from memory.

8. **Result Write-back:**
    - The fetched data from memory is written to register 7 ($R7) in the register file.

Instruction 8: LW R8, 0(R0)

Function: Loads the contents from memory location calculated by adding the contents of register 0 ($R0) and an offset of 0 and stores the result in register 8 ($R8).

1. **Fetch:**
    - PC points to the instruction memory address containing the LW instruction.

2. **PC Update**:
    - PC is incremented by 2 to point to the next instruction.

3. **Decode**:
    - Instruction is broken down into its components (opcode, Rd, Rs, and offset).

4. **Control Signal Generation**:
    - Control unit decodes the opcode and generates control signals:
        - reg_src = 00 (select data from memory for register file write back)
        - mem_wr = 0 (disable memory write)

        - mem_re = 1 (enable memory read)
        - alu_op = 00 (perform addition)
        - alu_src = 1 (use immediate value for ALU input B)
        - reg_load = 1 (enable register write)

5. **Operand Fetch**:
    - Contents of register 0 ($R0) are retrieved from the register file and sent to input A of the ALU.

6. **Memory Address Calculation**:
    - The 4-bit offset (0) is sign-extended to 16 bits.
    - The ALU adds the sign-extended offset to the value from $R0, producing the memory address.

7. **Memory Read**:
    - The data at the calculated memory address is fetched from memory.

8. **Result Write-back**:
    - The fetched data from memory is written to register 8 ($R8) in the register file.

Instruction 9: ADD R9, R7, R8

Function: Adds the contents of register 7 ($R7) and register 8 ($R8) and stores the result in register 9 ($R9).

1. **Fetch:**
    - PC points to the instruction memory address containing the ADD instruction.

2. **PC Update:**
    - PC is incremented by 2 to point to the next instruction.

3. **Decode:**
    - Instruction is broken down into its components (opcode, Rd, Rs, and Rt).

4. **Control Signal Generation:**
    - Control unit decodes the opcode and generates control signals:
        - reg_src = 01 (select ALU output for register file write back)
        - mem_wr = 0 (disable memory write)
        - mem_re = 0 (disable memory read)
        - alu_op = 00 (perform addition)

        - alu_src = 0 (use data from register for ALU input B)
        - reg_load = 1 (enable register write)

5. **Operand Fetch:**
    - Contents of register 7 ($R7) are retrieved from the register file and sent to input A of the ALU.
    - Contents of register 8 ($R8) are retrieved from the register file and sent to input B of the ALU.

6. **ALU Operation:**
    - ALU performs addition on input A (value from $R7) and input B (value from $R8).

7. **Result Write-back:**
    - The 16-bit result from the ALU is written back to register 9 ($R9) in the register file.

Instruction 10: SLT R10, R0, R1

Function: Compares the contents of register 0 ($R0) and register 1 ($R1) using subtraction and sets the least significant bit (LSB) of register 10 ($R10) to 1 if $R0 is less than $R1. Otherwise, the LSB is set to 0.

1. **Fetch:**
    - PC points to the instruction memory address containing the SLT instruction.

2. **PC Update:**
    - PC is incremented by 2 to point to the next instruction.

3. **Decode:**
    - Instruction is broken down into its components (opcode, Rd, Rs, and Rt).

4. **Control Signal Generation:**
    - Control unit decodes the opcode and generates control signals:
        - reg_src = 10 (select output from the SLT unit for register file write back)
        - mem_wr = 0 (disable memory write)
        - mem_re = 0 (disable memory read)
        - alu_op = 01 (perform subtraction)
        - alu_src = 1 (use data from register for ALU input B)
        - reg_load = 1 (enable register write)

5. **Operand Fetch:**
   - Contents of register 0 ($R0) are retrieved from the register file and sent to input A of the ALU.
   - Contents of register 1 ($R1) are retrieved from the register file and sent to input B of the ALU.

6. **ALU Operation:**
   - ALU performs subtraction on input A (value from $R0) and input B (value from $R1).

7. **SLT Unit:**
   - The result from the ALU is fed into the Sign Less Than (SLT) unit.
   - The SLT unit extracts the sign bit (MSB) of the subtraction result.
   - If the MSB is 1 (negative result, indicating $R0 < $R1), the output is set to 1.
   - If the MSB is 0 (positive result, indicating $R0 >= $R1), the output is set to 0.

8. **Result Concatenation:**
   - The SLT unit's output (sign bit) is concatenated with 15 zeros to form a 16-bit value.

9. **Result Write-back:**
   - The 16-bit value from the SLT unit is written back to register 10 ($R10) in the register file.

Instruction 11: SLT R10, R1, R0

Function: Compares the contents of register 1 ($R1) and register 0 ($R0) using subtraction and sets the least significant bit (LSB) of register 10 ($R10) to 1 if $R1 is less than $R0. Otherwise, the LSB is set to 0.

1. **Fetch:**
   - PC points to the instruction memory address containing the SLTI instruction.

2. **PC Update:**
   - PC is incremented by 2 to point to the next instruction.

3. **Decode:**
   - Instruction is broken down into its components (opcode, Rd, Rs, and Rt).

4. **Control Signal Generation:**

- Control unit decodes the opcode and generates control signals:
  - reg_src = 10 (select output from the SLT unit for register file write back)
  - mem_wr = 0 (disable memory write)
  - mem_re = 0 (disable memory read)
  - alu_op = 01 (perform subtraction)
  - alu_src = 1 (use data from register for ALU input B)
  - reg_load = 1 (enable register write)

5. **Operand Fetch:**
   - Contents of register 1 ($R1) are retrieved from the register file and sent to input A of the ALU.
   - Contents of register 0 ($R0) are retrieved from the register file and sent to input B of the ALU.

6. **ALU Operation:**
   - ALU performs subtraction on input A (value from $R1) and input B (value from $R0).

7. **SLT Unit (implicit in SLTI):**
   - The SLTI instruction treats register 0 (Rt) as an immediate value of 0.
   - Therefore, the subtraction effectively compares $R1 with 0.
   - The SLT unit extracts the sign bit (MSB) of the subtraction result.
   - If the MSB is 1 (negative result, indicating $R1 < 0), the output is set to 1.
   - If the MSB is 0 (positive result, indicating $R1 >= 0), the output is set to 0.

8. **Result Concatenation:**
   - The SLT unit's output (sign bit) is concatenated with 15 zeros to form a 16-bit value.

9. **Result Write-back:**
   - The 16-bit value from the SLT unit is written back to register 10 ($R10) in the register file.

Instruction 12: OR R5, R10, R9

Function: Performs a bitwise OR operation on the contents of register 10 ($R10) and register 9 ($R9), then stores the result in register 5 ($R5).

1. **Fetch:**
   - PC points to the instruction memory address containing the OR instruction.

2. **PC Update:**
   - PC is incremented by 2 to point to the next instruction.

3. **Decode:**
   - Instruction is broken down into its components (opcode, Rd, Rs, and Rt).

4. **Control Signal Generation:**
   - Control unit decodes the opcode and generates control signals:
     - reg_src = 01 (select ALU output for register file write back)
     - mem_wr = 0 (disable memory write)
     - mem_re = 0 (disable memory read)
     - alu_op = 11 (perform bitwise OR)
     - alu_src = 1 (use data from register for ALU input B)
     - reg_load = 1 (enable register write)

5. **Operand Fetch:**
   - Contents of register 10 ($R10) are retrieved from the register file and sent to input A of the ALU.
   - Contents of register 9 ($R9) are retrieved from the register file and sent to input B of the ALU.

6. **ALU Operation:**
   - ALU performs a bitwise OR operation on input A (value from $R10) and input B (value from $R9).

7. **Result Write-back:**
   - The 16-bit result from the ALU is written back to register 5 ($R5) in the register file.

Instruction 13: SUBI R10, R5, 7

Function: Subtracts the immediate value 7 from the contents of register 5 ($R5) and stores the result in register 10 ($R10).

1. **Fetch:**
   - PC points to the instruction memory address containing the ADDI instruction.

2. **PC Update:**
   - PC is incremented by 2 to point to the next instruction.

3. **Decode:**
   - Instruction is broken down into its components (opcode, Rd, Rs, and immediate).

4. **Control Signal Generation:**
   - Control unit decodes the opcode and generates control signals:
     - reg_src = 01 (select ALU output for register file write back)
     - mem_wr = 0 (disable memory write)
     - mem_re = 0 (disable memory read)
     - alu_op = 01 (perform subtraction)
     - alu_src = 1 (use immediate for ALU input B)
     - reg_load = 1 (enable register write)

5. **Operand Fetch:**
   - Contents of register 5 ($R5) are retrieved from the register file and sent to input A of the ALU.
   - Immediate value 7 is sign-extended to 16 bits and sent to input B of the ALU.

6. **ALU Operation:**
   - ALU subtracts the 16-bit immediate value from the value in input A.

7. **Result Write-back:**
   - The 16-bit result from the ALU is written back to register 10 ($R10) in the register file.

Instruction 14: SUB R11, R10, R7

Function: Subtracts the contents of register 7 ($R7) from the contents of register 10 ($R10) and stores the result in register 11 ($R11).

1. **Fetch:**
   - PC points to the instruction memory address containing the SUB instruction.

2. **PC Update:**
   - PC is incremented by 2 to point to the next instruction.

3. **Decode:**
   - Instruction is broken down into its components (opcode, Rd, Rs, and Rt).

4. **Control Signal Generation:**

- Control unit decodes the opcode and generates control signals:
    - reg_src = 01 (select ALU output for register file write back)
    - mem_wr = 0 (disable memory write)
    - mem_re = 0 (disable memory read)
    - alu_op = 01 (perform subtraction)
    - alu_src = 1 (use data from register for ALU input B)
    - reg_load = 1 (enable register write)

5. **Operand Fetch:**
    - Contents of register 10 ($R10) are retrieved from the register file and sent to input A of the ALU.
    - Contents of register 7 ($R7) are retrieved from the register file and sent to input B of the ALU.

6. **ALU Operation:**
    - ALU subtracts the value in input B (from $R7) from the value in input A (from $R10).

7. **Result Write-back:**
    - The 16-bit result from the ALU is written back to register 11 ($R11) in the register file.

Instruction 15: SW R11, 5(R8)

Function: Stores the contents of register 11 ($R11) into the memory location calculated by adding the immediate offset 5 to the contents of register 8 ($R8).

1. **Fetch:**
    - PC points to the instruction memory address containing the SW instruction.

2. **PC Update:**
    - PC is incremented by 2 to point to the next instruction.

3. **Decode:**
    - Instruction is broken down into its components (opcode, Rd, Rs, and offset).

4. **Control Signal Generation:**
    - Control unit decodes the opcode and generates control signals:
        - reg_src = XX (don't care as no register write-back)
        - mem_wr = 1 (enable memory write)

  - mem_re = 0 (disable memory read)
  - alu_op = 00 (perform addition for memory address calculation)
  - alu_src = 1 (use immediate for ALU input B)
  - reg_load = 0 (disable register write)

5. **Operand Fetch:**
   - Contents of register 8 ($R8) are retrieved from the register file and sent to input A of the ALU.
   - Immediate offset 5 is sign-extended to 16 bits and sent to input B of the ALU.
   - Contents of register 11 ($R11) are retrieved from the register file and bypassed to the memory data input.

6. **Memory Address Calculation:**
   - ALU adds the 16-bit immediate offset to the value in input A (from $R8), resulting in the memory address for the store operation.

7. **Memory Write:**
   - The memory data input (value from $R11) is written to the calculated memory address.

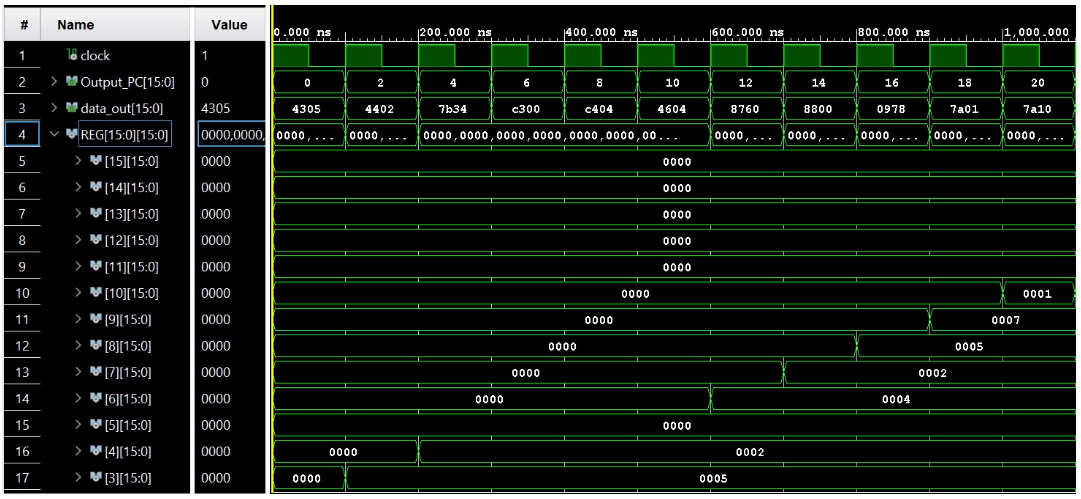
Verification (PART A)

A summary of the waveform results can be found in the table below. These results were obtained by running Instr.txt file that was provided in the lab. Please note that the Value (Rd) is not necessarily the register, it is the result of wherever the operation result is. It can be a register or in memory.

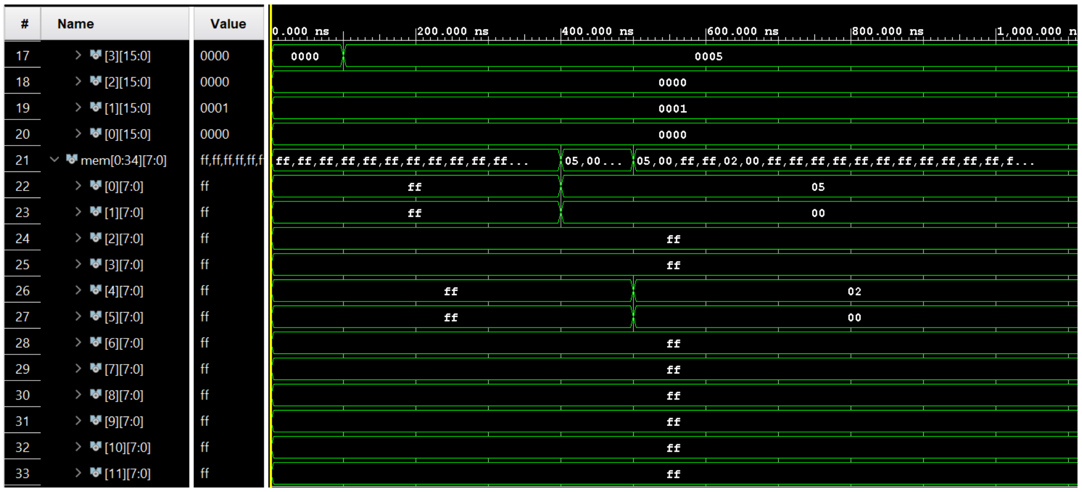
Instruction	OpCode	Rd	Rs	Rt/Imm	Rd
ADDI R3, R0, 5	0x4	0x3	0x0	0x5	0x5
ADDI R4, R0, 2	0x4	0x4	0x0	0x2	0x2
SLT R11, R3, R4	0x7	0xB	0x3	0x4	0x0
SW R3, 0(R0)	0xC	0x3	0x0	0x3	0x5
SW R4, 4(R0)	0xC	0x4	0x0	0x4	0x2
ADDI R6, R0, 4	0x4	0x6	0x0	0x4	0x4
LW R7, 0(R6)	0x8	0x7	0x6	0x0	0x2
LW R8, 0(R8)	0x8	0x8	0x8	0x0	0x5
ADD R9, R7, R8	0x0	0x9	0x7	0x8	0x7
SLT R10, R0, R1	0x7	0xA	0x0	0x1	0x1
SLT R10, R1, R0	0x7	0xA	0x1	0x0	0x0
OR R5, R10, R9	0x3	0x5	0xA	0x9	0x7
SUBI R10, R5, 7	0x5	0xA	0x5	0x7	0x0
SUB R11, R10, R7	0x1	0xB	0xA	0x7	0xFFFE
SW R11, 5(R8)	0xC	0xB	0x8	0x5	0xFFFE

Figure 8: Instruction Table of Part A

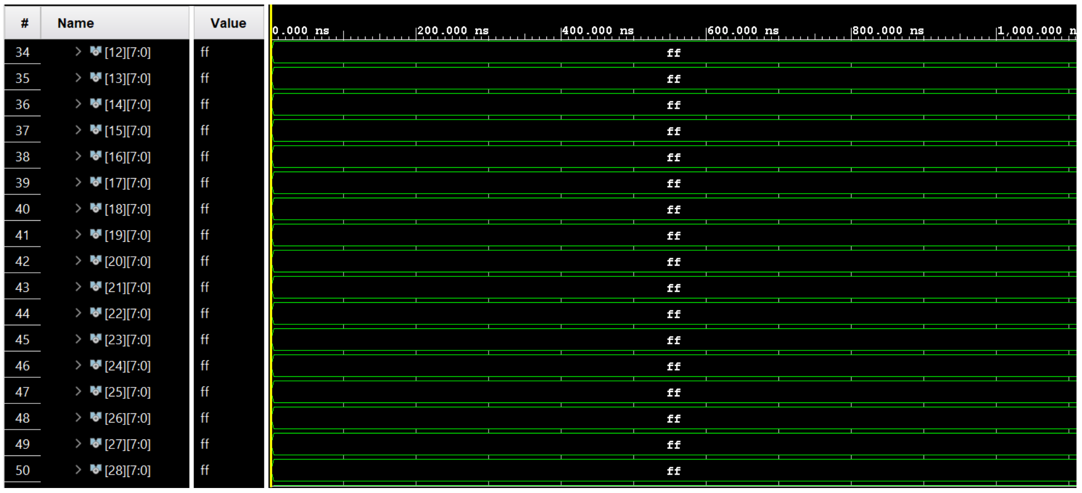
Waveform Part A: PC 0 - 20



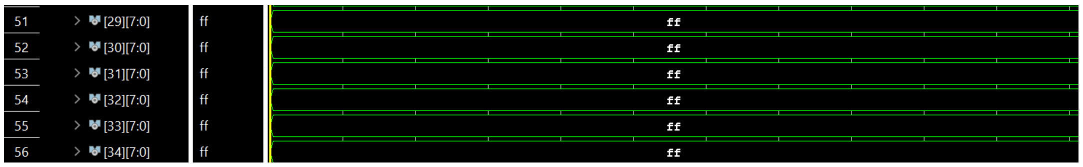
#	Name	Value	0.000 ns … 200.000 ns … 400.000 ns … 600.000 ns … 800.000 ns … 1,000.000
1	clock	1	
2	Output_PC[15:0]	0	0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20
3	data_out[15:0]	4305	4305, 4402, 7b34, c300, c404, 4604, 8760, 8800, 0978, 7a01, 7a10
4	REG[15:0][15:0]	0000,0000,	0000,..., 0000,..., 0000,0000,0000,0000,0000,0000,00..., 0000,..., 0000,..., 0000,..., 0000,..., 0000,...
5	[15][15:0]	0000	0000
6	[14][15:0]	0000	0000
7	[13][15:0]	0000	0000
8	[12][15:0]	0000	0000
9	[11][15:0]	0000	0000
10	[10][15:0]	0000	0000, 0001
11	[9][15:0]	0000	0000, 0007
12	[8][15:0]	0000	0000, 0005
13	[7][15:0]	0000	0000, 0002
14	[6][15:0]	0000	0000, 0004
15	[5][15:0]	0000	0000
16	[4][15:0]	0000	0000, 0002
17	[3][15:0]	0000	0000, 0005



#	Name	Value	0.000 ns … 200.000 ns … 400.000 ns … 600.000 ns … 800.000 ns … 1,000.000 n
17	[3][15:0]	0000	0000, 0005
18	[2][15:0]	0000	0000
19	[1][15:0]	0001	0001
20	[0][15:0]	0000	0000
21	mem[0:34][7:0]	ff,ff,ff,ff,ff,f	ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff..., 05,00..., 05,00,ff,ff,02,00,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,f...
22	[0][7:0]	ff	ff, 05
23	[1][7:0]	ff	ff, 00
24	[2][7:0]	ff	ff
25	[3][7:0]	ff	ff
26	[4][7:0]	ff	ff, 02
27	[5][7:0]	ff	ff, 00
28	[6][7:0]	ff	ff
29	[7][7:0]	ff	ff
30	[8][7:0]	ff	ff
31	[9][7:0]	ff	ff
32	[10][7:0]	ff	ff
33	[11][7:0]	ff	ff

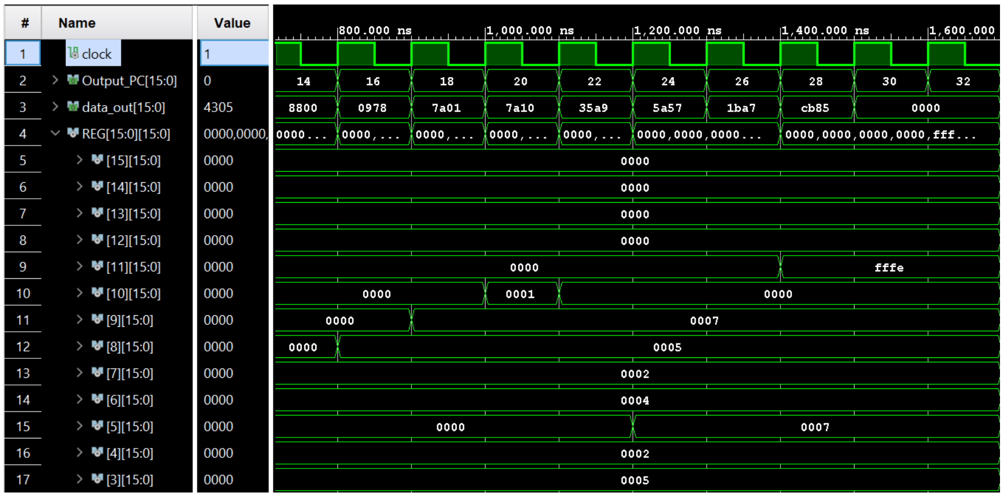


#	Name	Value	0.000 ns … 200.000 ns … 400.000 ns … 600.000 ns … 800.000 ns … 1,000.000 n
34	[12][7:0]	ff	ff
35	[13][7:0]	ff	ff
36	[14][7:0]	ff	ff
37	[15][7:0]	ff	ff
38	[16][7:0]	ff	ff
39	[17][7:0]	ff	ff
40	[18][7:0]	ff	ff
41	[19][7:0]	ff	ff
42	[20][7:0]	ff	ff
43	[21][7:0]	ff	ff
44	[22][7:0]	ff	ff
45	[23][7:0]	ff	ff
46	[24][7:0]	ff	ff
47	[25][7:0]	ff	ff
48	[26][7:0]	ff	ff
49	[27][7:0]	ff	ff
50	[28][7:0]	ff	ff



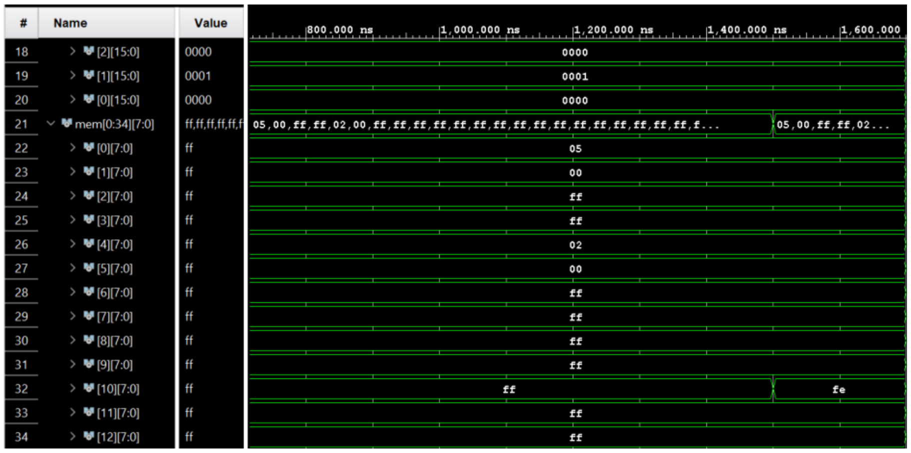
51	> [29][7:0]	ff	ff
52	> [30][7:0]	ff	ff
53	> [31][7:0]	ff	ff
54	> [32][7:0]	ff	ff
55	> [33][7:0]	ff	ff
56	> [34][7:0]	ff	ff

Waveform Part A: PC 14 – 32



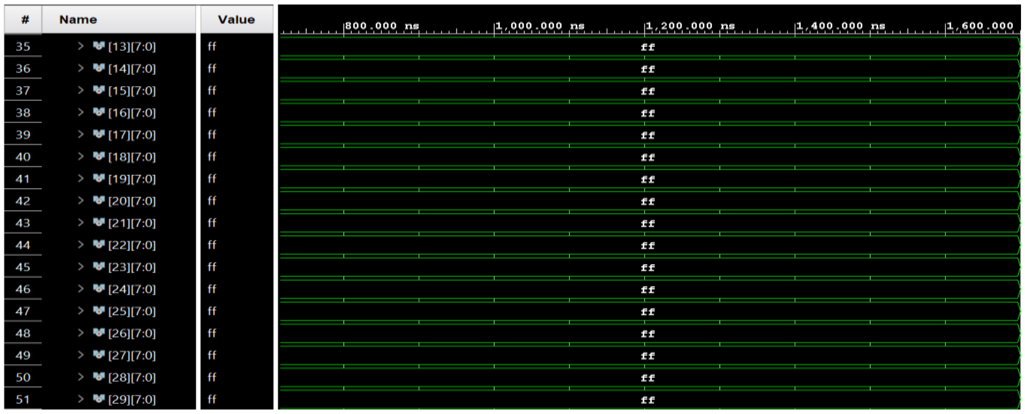
800,000 ns 1,000,000 ns 1,200,000 ns 1,400,000 ns 1,600,000

#	Name	Value	
1	clock	1	
2	> Output_PC[15:0]	0	14, 16, 18, 20, 22, 24, 26, 28, 30, 32
3	> data_out[15:0]	4305	8800, 0978, 7a01, 7a10, 35a9, 5a57, 1ba7, cb85, 0000
4	⌄ REG[15:0][15:0]	0000,0000,	0000..., 0000,..., 0000,..., 0000,..., 0000,..., 0000,0000,0000..., 0000,0000,0000,0000,fff...
5	> [15][15:0]	0000	0000
6	> [14][15:0]	0000	0000
7	> [13][15:0]	0000	0000
8	> [12][15:0]	0000	0000
9	> [11][15:0]	0000	0000, fffe
10	> [10][15:0]	0000	0000, 0001, 0000
11	> [9][15:0]	0000	0000, 0007
12	> [8][15:0]	0000	0000, 0005
13	> [7][15:0]	0000	0002
14	> [6][15:0]	0000	0004
15	> [5][15:0]	0000	0000, 0007
16	> [4][15:0]	0000	0002
17	> [3][15:0]	0000	0005



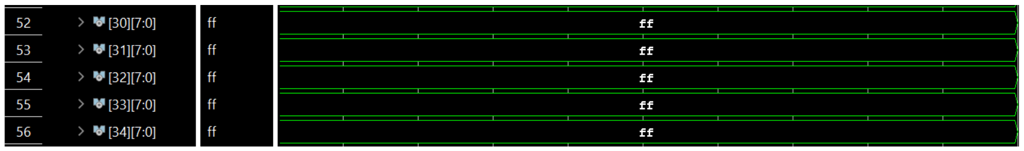
800,000 ns 1,000,000 ns 1,200,000 ns 1,400,000 ns 1,600,000

#	Name	Value	
18	> [2][15:0]	0000	0000
19	> [1][15:0]	0001	0001
20	> [0][15:0]	0000	0000
21	⌄ mem[0:34][7:0]	ff,ff,ff,ff,ff,f	05,00,ff,ff,02,00,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,f..., 05,00,ff,ff,02...
22	> [0][7:0]	ff	05
23	> [1][7:0]	ff	00
24	> [2][7:0]	ff	ff
25	> [3][7:0]	ff	ff
26	> [4][7:0]	ff	02
27	> [5][7:0]	ff	00
28	> [6][7:0]	ff	ff
29	> [7][7:0]	ff	ff
30	> [8][7:0]	ff	ff
31	> [9][7:0]	ff	ff
32	> [10][7:0]	ff	ff, fe
33	> [11][7:0]	ff	ff
34	> [12][7:0]	ff	ff



800,000 ns 1,000,000 ns 1,200,000 ns 1,400,000 ns 1,600,000

#	Name	Value	
35	> [13][7:0]	ff	ff
36	> [14][7:0]	ff	ff
37	> [15][7:0]	ff	ff
38	> [16][7:0]	ff	ff
39	> [17][7:0]	ff	ff
40	> [18][7:0]	ff	ff
41	> [19][7:0]	ff	ff
42	> [20][7:0]	ff	ff
43	> [21][7:0]	ff	ff
44	> [22][7:0]	ff	ff
45	> [23][7:0]	ff	ff
46	> [24][7:0]	ff	ff
47	> [25][7:0]	ff	ff
48	> [26][7:0]	ff	ff
49	> [27][7:0]	ff	ff
50	> [28][7:0]	ff	ff
51	> [29][7:0]	ff	ff



```
52  > [30][7:0]  ff   ff
53  > [31][7:0]  ff   ff
54  > [32][7:0]  ff   ff
55  > [33][7:0]  ff   ff
56  > [34][7:0]  ff   ff
```

System Overview (PART B)

After completing the 16-bit CPU in Part A, the architecture required to support jump and BNE instructions must be built using a couple of extra modules, an extra control signal, and existing hardware. After some research and going through course notes we added the remaining block diagrams to the top module diagram and implemented these modules in the final iteration of our CPU module. The blocks that were created in this section were the BNE, JUMP, and the PC Mux blocks.

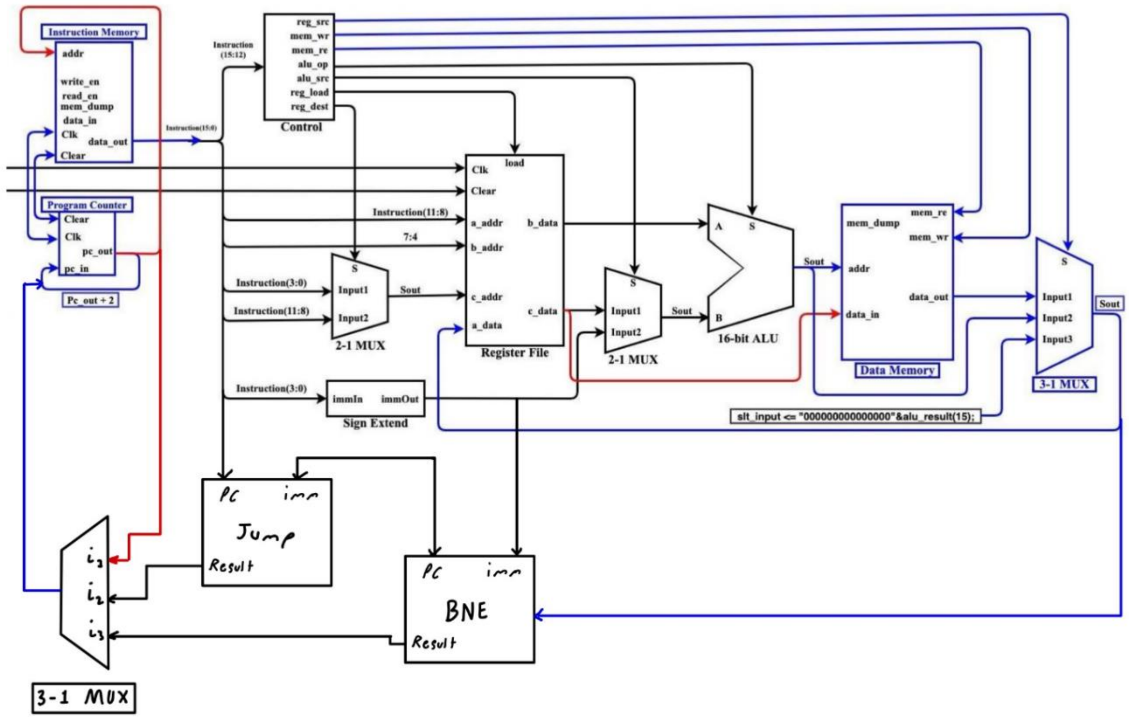


Figure 9 Part-B CPU

BNE Block

The Branch if Not Equal block in the CPU handles conditional branching based on the 16-bit result from the ALU. When the ALU_Result is zero, the BNE block increments the program counter by 2, indicating the program should continue sequentially. However, if the ALU_Result is nonzero, the block adds the 4-bit immediate offset to the PC and then increments by 2, directing the program to a different memory location to continue execution. This mechanism allows for decision-making in programs based on whether a specific condition is met or not, making the program flow better in response to outcome given by the computer.

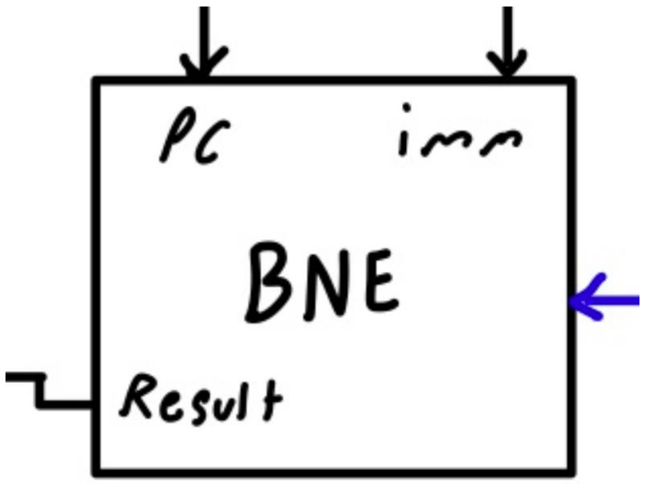


Figure 10 Branch if Not Equal

Branch if Not Equal VHDL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity BNE is
  port(
    BNE_Imm:        in STD_LOGIC_VECTOR(3 downto 0);
    BNE_ALU_Result:   in STD_LOGIC_VECTOR(15 downto 0);
    BNE_PC_IN:       in STD_LOGIC_VECTOR(15 downto 0);
    BNE_PC_Branched: out STD_LOGIC_VECTOR(15 downto 0));
end BNE;
architecture Behavioral of BNE is

begin
  stm: process(BNE_ALU_result,BNE_PC_IN,BNE_Imm)
   begin
    if(BNE_ALU_Result = x"0000") then
      BNE_PC_Branched <= BNE_PC_IN;
```

```
        else
            BNE_PC_Branched <= ("000000000000" & BNE_Imm) + BNE_PC_IN;
        end if;
    end process;
end Behavioral;
```

Jump Block

The jump block is a part of the CPU that performs a specific function, it takes in a 12-bit immediate value from the instruction and the current program counter value. The purpose is to apply an offset to the program counter to jump to a new instruction address. To achieve this, the Jump block combines the 12-bit immediate value with the four most significant bits of the program counter. By concatenating these values, it forms a new program counter address, allowing the CPU to branch to the desired instruction location in memory,

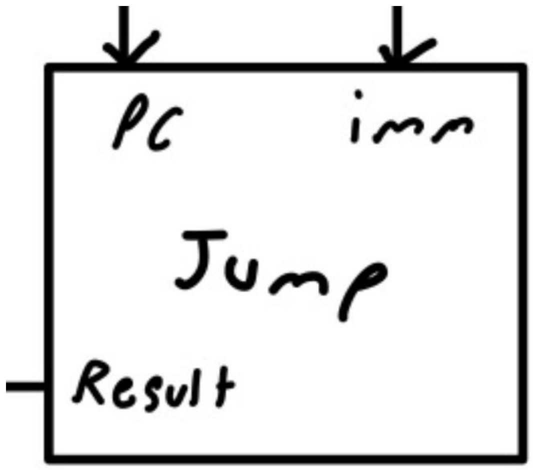


Figure 11 Jump Block

Jump VHDL

```
entity Jump is
  port(
  JUMP_PC_IN  : in  STD_LOGIC_VECTOR(15 downto 0);
  JUMP_Imm_IN : in  STD_LOGIC_VECTOR(11 downto 0);
  JUMP_PC_OUT : out STD_LOGIC_VECTOR(15 downto 0));
end Jump;

architecture Behavioral of Jump is
signal pc_minus2 : std_logic_vector(15 downto 0);
begin
  pc_minus2 <= JUMP_PC_IN - x"0002";
  JUMP_PC_OUT <= (pc_minus2(15 downto 12) & JUMP_Imm_IN(11 downto 0));
end Behavioral;
```

PC Control MUX

The PC control mux will take in each of the possible PC increments, and a 2-bit control signal from the control unit to determine the increment of the PC. I1 is the incremented program counter for all instructions from Part A, i2 is the program counter output from the jump block, and i3 is the program counter from the BNE block. The opcode input from the instruction will determine which output the PC will see on the next rising edge of the clock.

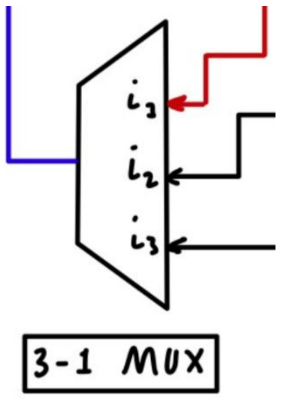


Figure 12 PC CTRL MUX

3_1 MUX VHDL

```
begin
        Sout <=         Input1 when S="00" else
                                Input2 when S="01" else
                                Input3 when S="10" else
                                Input3 when S="11";
end Behavioral;
```

Instruction Tracing (PART B)

Only one BNE and Jump instruction were traced from the Instr2.txt instruction list to reduce repetition. This is mentioned in the assumptions section. A summary of the rest can be found in the results table.

BNE Instruction: BNE r8, r0, 2

BNE subtracts the contents of register 5 ($R8), and the contents of register 0 ($R0) and will offset the program counter by 4 (2 immediate plus 2 incremented counter) if the branches are not equal and will offset the program counter by 2 (incremented counter) if they are equal. The instruction looks like

Function: The BNE instruction performs a conditional branch based on the comparison of register contents.

1. **Fetch:**
   - The program counter (PC) points to the instruction memory address containing the BNE instruction.

2. **PC Update:**
   - The PC is incremented by 2 to point to the next instruction in the sequence.

3. **Decode:**
   - The instruction is broken down into its components:
     - Opcode (15:12) is sent to the control module.
     - Destination register (Rd) (11:8) is sent to a_addr of the register file.
     - Source register (Rs) (7:4) is sent to b_addr of the register file.
     - Immediate value (4-bit) (3:0) is sent to the sign extend block.

4. **Control Signal Generation:**

- The control unit decodes the opcode and generates control signals:
    - REG_SRC = 01 (select ALU output for register file write back).
    - MEM_WR = 0 (disable memory write).
    - MEM_RE = 0 (disable memory read).
    - ALU_OP = 01 (subtract inputs).
    - ALU_SRC = 00 (use register contents for ALU input B).
    - REG_LOAD = 0 (disable register write).

5. **Operand Fetch:**
    - Contents of register 5 ($R8) are retrieved from b_addr of the register file and sent to input A of the ALU.
    - Contents of register 0 ($R0) are retrieved from a_addr of the register file and sent to input B of the ALU.

6. **ALU Operation:**
    - ALU subtracts the contents of register 5 ($R8) and register 0 ($R0).

7. **Branch Decision:**
    - If ALU_Result is zero, output PC + offset (4: 2 immediate + 2 incremented counter) to the program counter.
    - If ALU_Result is not zero, output incremented program counter to the program counter.

8. **Control Unit Output:**
    - MUX select line directs PC + Offset + 2 to the program counter for the next instruction.

Jump Instruction: Jump 14

The jump instruction will offset the program counter by a 12-bit immediate. In the case of this lab, it will concatenate the 4 MSB’s of the program count (always 0 in the lab) with the 12-bit immediate so that the jumped address will simply just be the 12-bit immediate. The instruction looks like

Function: The Jump instruction redirects program execution to a specified memory address.

1. **Fetch:**
   - The program counter (PC) points to the instruction memory address containing the Jump instruction.

2. **PC Update:**
   - The non-incremented PC is fed into the PC_IN path for the Jump block.

3. **Decode:**
   - The instruction is decomposed into its components:
     - Opcode (15:12) is sent to the control module.
     - Offset (11:0) (Imm = 0xE) is sent to the Jump Immediate of the Jump Block.

4. **Control Signal Generation:**
   - The control unit decodes the opcode and generates control signals:
     - REG_SRC = “XX” (Do not care).
     - MEM_WR = ‘0’ (Disable memory write).
     - MEM_RE = ‘0’ (Disable memory read).
     - PC_MUX = “01” (Output Jump branch into PC).
     - ALU_OP = “XX” (Do not care).
     - ALU_SRC = ‘X’ (Do not care).
     - REG_LOAD = ‘0’ (Disable register write).

5. **Jump Immediate:**
   - The 12-bit immediate (Offset) is sent to the 12-bit immediate of the Jump Block.

6. **Control Unit Output:**
   - MUX select line directs PC (15:12) || Jump (11:0) to the program counter for the next instruction.

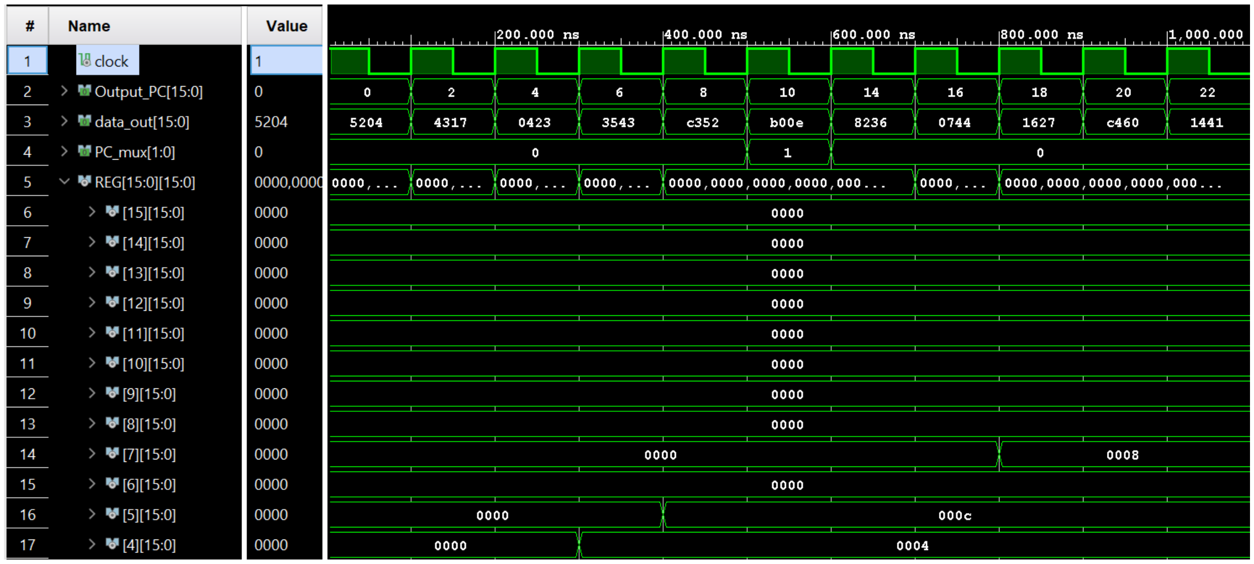
Verification (PART B)

A summary of the waveform results can be found in Figure 12. Note that this is the results of Instr2.txt. Please note that the value Rd does not necessarily mean the value of register rd. It is the value of where the destination of the operation is. It can be a register, memory, or program counter.

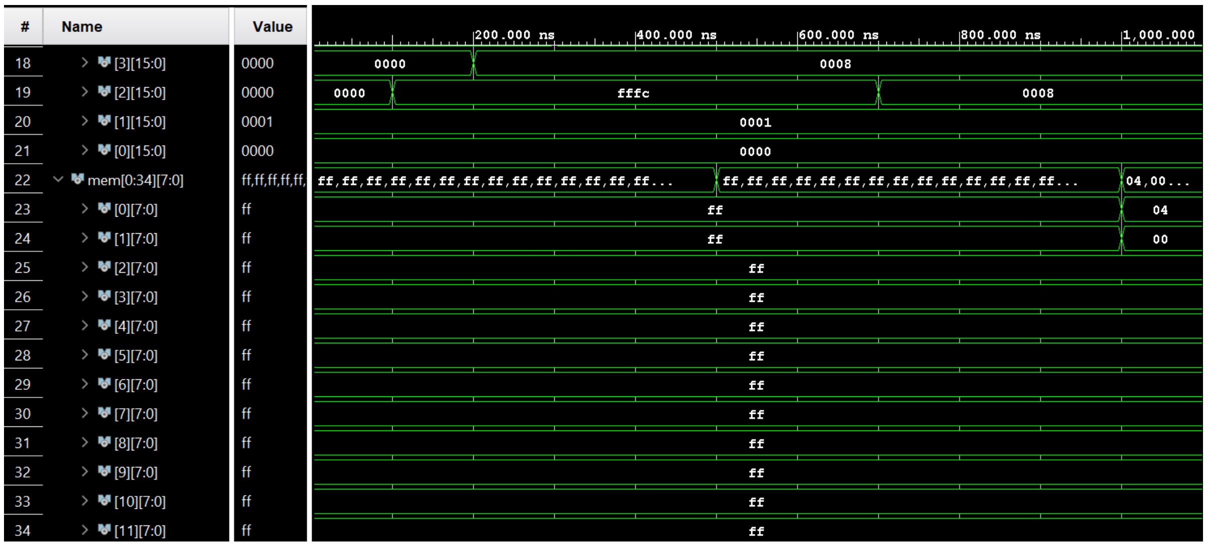
Instruction	OpCode	Rd	Rs	Rt	Rd
SUBI R2, R0, 4	0x5	0x2	0x0	0x4	0xFFFC
ADDI R3, R1, 7	0x4	0x3	0x1	0x7	0x8
ADD R4, R2, R3	0x0	0x4	0x2	0x3	0x4
OR R5, R4, R3	0x3	0x5	0x4	0x3	0xC
SW R3, 2(R5)	0xC	0x3	0x5	0x2	0x8
JMP 14	0xB	0x0	0x0	0xE	0xE
LW R2, 6(R3)	0x8	0x2	0x3	0x6	0x8
ADD R7, R4, R4	0x0	0x7	0x4	0x4	0x8
SUB R6, R2, R7	0x1	0x6	0x2	0x7	0x0
SW R4, 0(R6)	0xC	0x4	0x6	0x0	0x4
SUB R4, R4, R1	0x1	0x4	0x4	0x1	0x3
SLT R8, R4, R1	0x7	0x8	0x4	0x1	0x0
BNE R8, R0, 2	0x9	0x8	0x0	0x2	0x1C
JMP 16	0xB	0x0	0x2	0xF	0x10
AND R5, R7, R1	0x2	0x5	0x7	0x1	0x0

Figure 13: Instruction Table for Part B

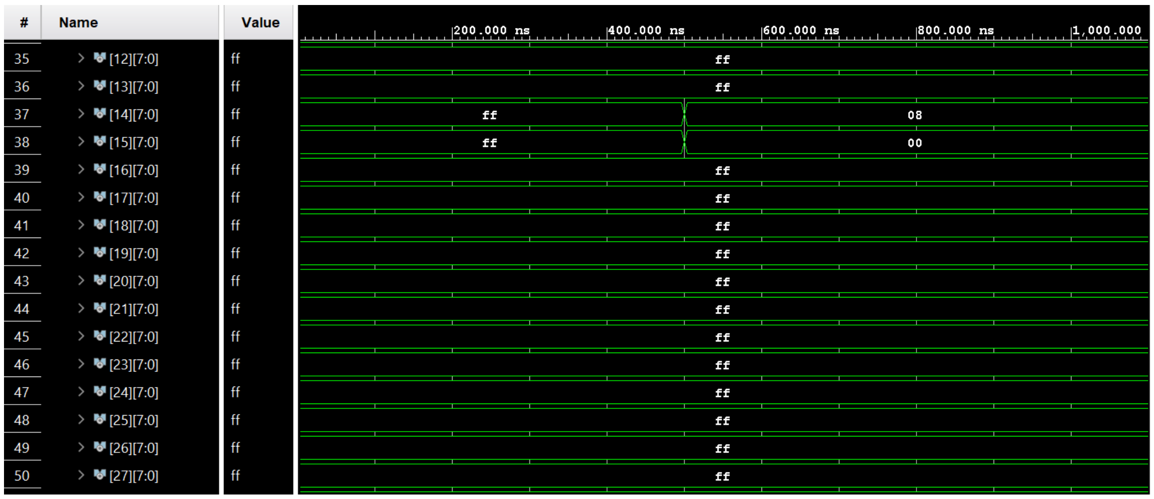
Waveform Part B: PC 0 – 22



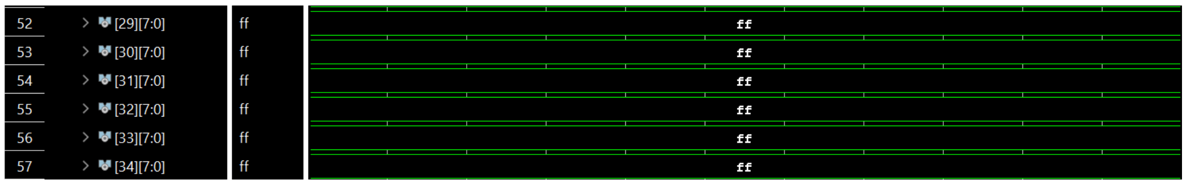
```
#   Name               Value       200.000 ns | 400.000 ns | 600.000 ns | 800.000 ns | 1,000.000
1   clock              1
2   Output_PC[15:0]    0           0 | 2 | 4 | 6 | 8 | 10 | 14 | 16 | 18 | 20 | 22
3   data_out[15:0]     5204        5204 | 4317 | 0423 | 3543 | c352 | b00e | 8236 | 0744 | 1627 | c460 | 1441
4   PC_mux[1:0]        0           0 | 1 | 0
5   REG[15:0][15:0]    0000,0000   0000,... | 0000,... | 0000,... | 0000,... | 0000,0000,0000,0000,000... | 0000,... | 0000,0000,0000,0000,000...
6   [15][15:0]         0000        0000
7   [14][15:0]         0000        0000
8   [13][15:0]         0000        0000
9   [12][15:0]         0000        0000
10  [11][15:0]         0000        0000
11  [10][15:0]         0000        0000
12  [9][15:0]          0000        0000
13  [8][15:0]          0000        0000
14  [7][15:0]          0000        0000 | 0008
15  [6][15:0]          0000        0000
16  [5][15:0]          0000        0000 | 000c
17  [4][15:0]          0000        0000 | 0004
```



```
#   Name               Value       200.000 ns | 400.000 ns | 600.000 ns | 800.000 ns | 1,000.000
18  [3][15:0]          0000        0000 | 0008
19  [2][15:0]          0000        0000 | fffc | 0008
20  [1][15:0]          0001        0001
21  [0][15:0]          0000        0000
22  mem[0:34][7:0]     ff,ff,ff,ff,ff,  ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff... | ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff... | 04,00...
23  [0][7:0]           ff          ff | 04
24  [1][7:0]           ff          ff | 00
25  [2][7:0]           ff          ff
26  [3][7:0]           ff          ff
27  [4][7:0]           ff          ff
28  [5][7:0]           ff          ff
29  [6][7:0]           ff          ff
30  [7][7:0]           ff          ff
31  [8][7:0]           ff          ff
32  [9][7:0]           ff          ff
33  [10][7:0]          ff          ff
34  [11][7:0]          ff          ff
```

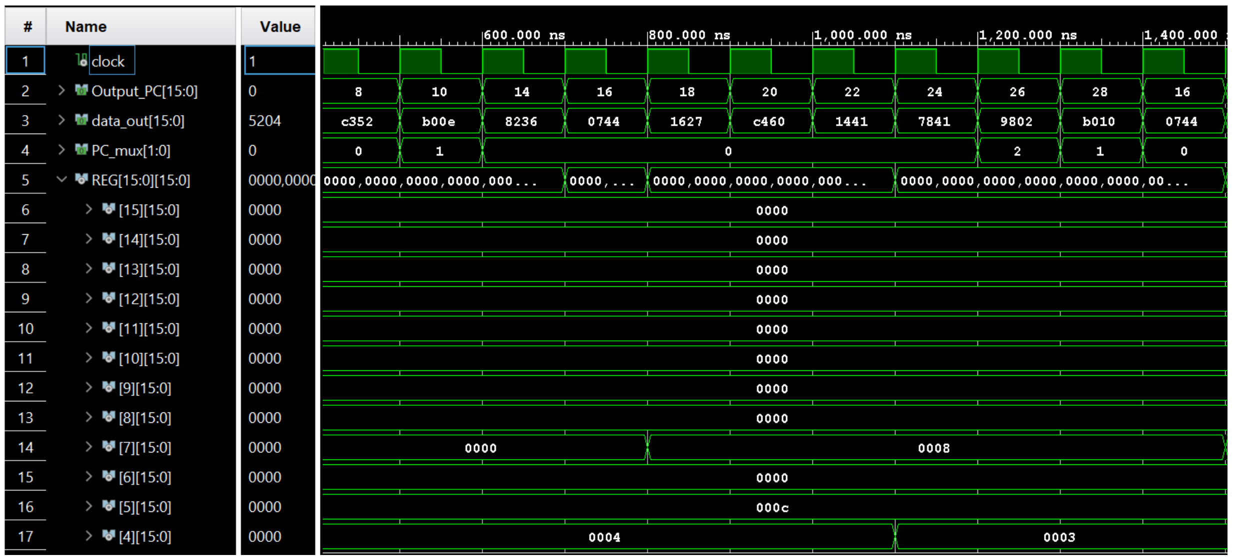


```
#   Name               Value       200.000 ns | 400.000 ns | 600.000 ns | 800.000 ns | 1,000.000
35  [12][7:0]          ff          ff
36  [13][7:0]          ff          ff
37  [14][7:0]          ff          ff | 08
38  [15][7:0]          ff          ff | 00
39  [16][7:0]          ff          ff
40  [17][7:0]          ff          ff
41  [18][7:0]          ff          ff
42  [19][7:0]          ff          ff
43  [20][7:0]          ff          ff
44  [21][7:0]          ff          ff
45  [22][7:0]          ff          ff
46  [23][7:0]          ff          ff
47  [24][7:0]          ff          ff
48  [25][7:0]          ff          ff
49  [26][7:0]          ff          ff
50  [27][7:0]          ff          ff
```

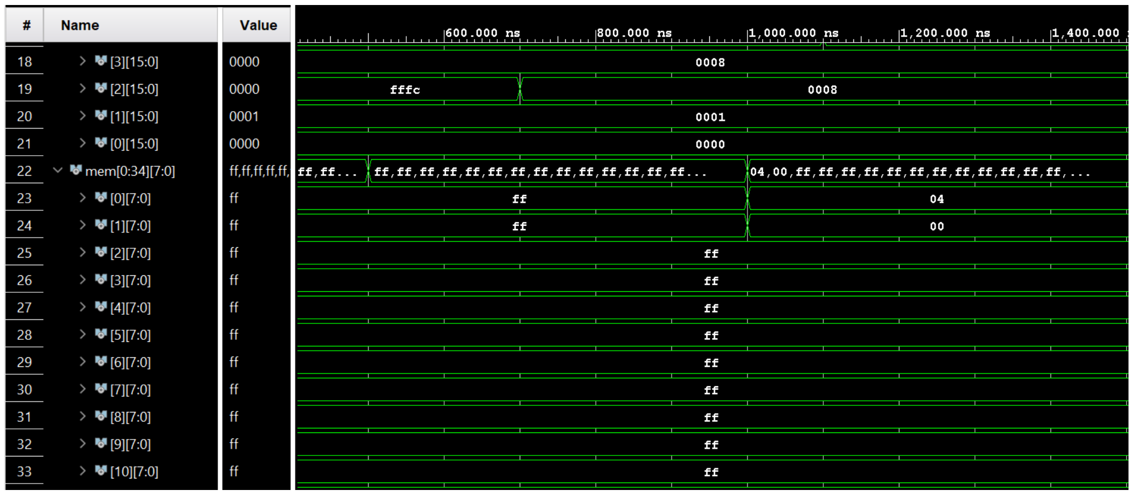


```
52  > [29][7:0]  ff   ff
53  > [30][7:0]  ff   ff
54  > [31][7:0]  ff   ff
55  > [32][7:0]  ff   ff
56  > [33][7:0]  ff   ff
57  > [34][7:0]  ff   ff
```

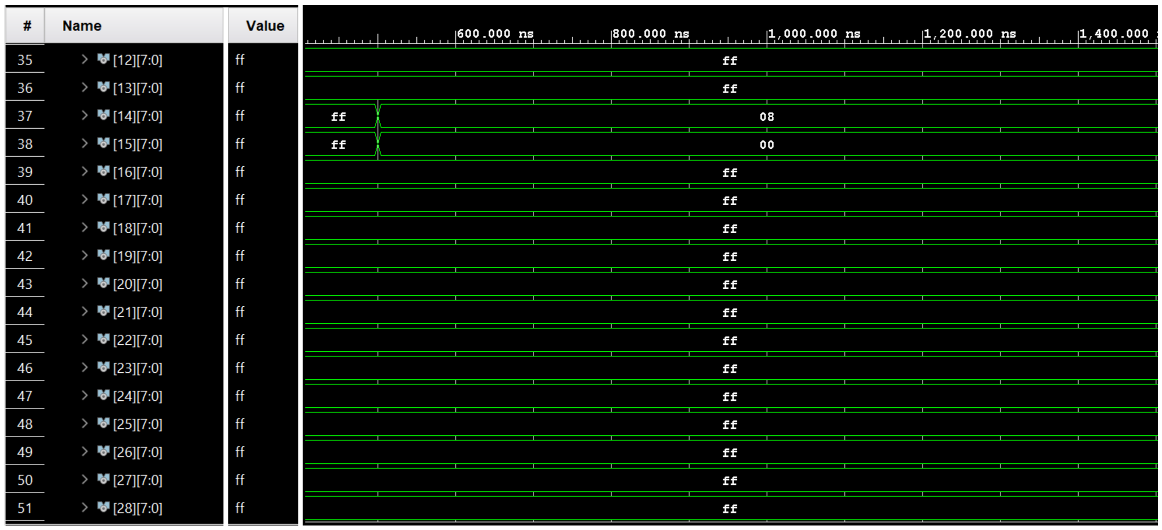
Waveform Part B: PC 8 – 28



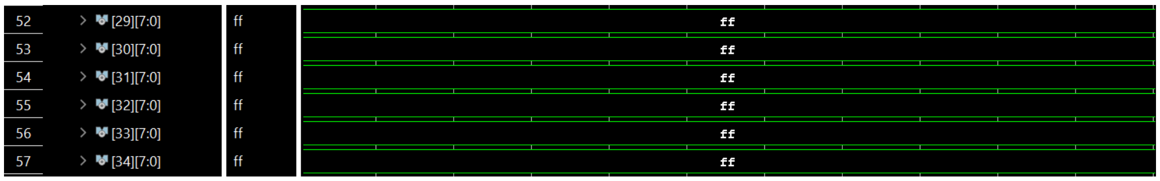
```
#   Name                Value      |600.000 ns      |800.000 ns      |1,000.000 ns    |1,200.000 ns    |1,400.000
1   clock               1
2   > Output_PC[15:0]   0          8     10    14    16    18    20    22    24    26    28    16
3   > data_out[15:0]    5204       c352  b00e  8236  0744  1627  c460  1441  7841  9802  b010  0744
4   > PC_mux[1:0]       0          0     1     0                                   2     1     0
5   v REG[15:0][15:0]   0000,0000  0000,0000,0000,0000,000...  0000,...  0000,0000,0000,0000,000...  0000,0000,0000,0000,0000,0000,00...
6     > [15][15:0]      0000       0000
7     > [14][15:0]      0000       0000
8     > [13][15:0]      0000       0000
9     > [12][15:0]      0000       0000
10    > [11][15:0]      0000       0000
11    > [10][15:0]      0000       0000
12    > [9][15:0]       0000       0000
13    > [8][15:0]       0000       0000
14    > [7][15:0]       0000       0000  0008
15    > [6][15:0]       0000       0000
16    > [5][15:0]       0000       000c
17    > [4][15:0]       0000       0004  0003
```



```
#   Name                Value          |600.000 ns    |800.000 ns    |1,000.000 ns    |1,200.000 ns    |1,400.000
18    > [3][15:0]       0000           0008
19    > [2][15:0]       0000           fffc  0008
20    > [1][15:0]       0001           0001
21    > [0][15:0]       0000           0000
22  v mem[0:34][7:0]    ff,ff,ff,ff,ff,  ff,ff...  ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff...  04,00,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,...
23    > [0][7:0]        ff             ff    04
24    > [1][7:0]        ff             ff    00
25    > [2][7:0]        ff             ff
26    > [3][7:0]        ff             ff
27    > [4][7:0]        ff             ff
28    > [5][7:0]        ff             ff
29    > [6][7:0]        ff             ff
30    > [7][7:0]        ff             ff
31    > [8][7:0]        ff             ff
32    > [9][7:0]        ff             ff
33    > [10][7:0]       ff             ff
```



#	Name	Value	600.000 ns … 800.000 ns … 1,000.000 ns … 1,200.000 ns … 1,400.000
35	[12][7:0]	ff	ff
36	[13][7:0]	ff	ff
37	[14][7:0]	ff	ff → 08
38	[15][7:0]	ff	ff → 00
39	[16][7:0]	ff	ff
40	[17][7:0]	ff	ff
41	[18][7:0]	ff	ff
42	[19][7:0]	ff	ff
43	[20][7:0]	ff	ff
44	[21][7:0]	ff	ff
45	[22][7:0]	ff	ff
46	[23][7:0]	ff	ff
47	[24][7:0]	ff	ff
48	[25][7:0]	ff	ff
49	[26][7:0]	ff	ff
50	[27][7:0]	ff	ff
51	[28][7:0]	ff	ff



#	Name	Value	
52	[29][7:0]	ff	ff
53	[30][7:0]	ff	ff
54	[31][7:0]	ff	ff
55	[32][7:0]	ff	ff
56	[33][7:0]	ff	ff
57	[34][7:0]	ff	ff

Errors and Issues

During the building process of our CPU, we ran into a couple of major and minor issues, some which we were not able to figure out, and some that we were. The first error was during the creation of the data memory. When running the file, we would get the correct values in our registers and thought we had completed the first section of the lab, but during demonstration when we went through the memory, we were off by 2 for each location in memory. It was a struggle trying to figure out where this issue was occurring. We ran the implemented design and said one of the blocks was not built properly. After this issue was fixed, we were able to move onto part 2. The second error was in creating the Jump block for our CPU. When first creating the jump function, we had the understanding that the Jump function would add the jump offset to the program counter. Therefore, when implementing it in our code it was working. However, after checking the values, it seemed odd that the jump 14 instruction was the second final instruction in the file while skipping over most of the instructions. BNE instruction before jump would bypass the jump instruction if registers 2 and 8 were not equal, which did not happen after running the code for the first time. This caused a lot of confusion, however after looking in and understanding the instructions better, the issue was resolved. Most errors we face are assumed to be due to errors in the system, as simply restating the program fixed them. We also run into some minor syntax errors, and missed placed signals that were easily fixed.

Conclusion

The goal of this project was to compile all the subsystems made in previous labs into a hierarchal CPU module. Part A entailed the implementation of the CPU to support functions which contained ALU operations such as lw, sw and slt. In part B the CPU module was extended to support bne and j instructions. Instr.txt was used to test part A and instr2.txt was used to test part B. In summary the lab was a success since we were able to achieve the results on the test bench as expected from all the functions. The outputs matched expectations, and the demonstration went reasonable apart from some technical and logical errors.