Lab Assignment 09

CSL3130: Computer Organization and Architecture

Academic Year 2025–26

Name: **Anmol Yadav**

Roll No: **b23cs1004**

Department: Computer Science and Engineering

IIT Jodhpur

Contents

1. Assumptions

- The processor operates on a single-cycle datapath, where each instruction completes in one clock cycle.
- Instruction memory and data memory are separate and both are word-addressable.
- All memory addresses are aligned to 4 bytes.
- Program Counter (PC) increments by 4 after every instruction fetch, unless a branch or jump occurs.
- Control signals are combinational and derived directly from the instruction opcode.
- The processor currently supports the execution of the following instructions: `add`, `sub`, `and`, `or`, `slt`, `addi`, `lw`, `sw`, `beq`, and one jump instruction `j`.

2. Understanding of the Task

The goal of this assignment is to design, implement, and test a simplified single-cycle MIPS processor in Logisim. The design integrates various submodules such as the ALU, ALU Control, Register File, Instruction Memory, Data Memory, and Control Unit into a cohesive architecture capable of executing a subset of MIPS instructions. The processor is validated by executing MIPS assembly programs and verifying results through simulation outputs.

3. Description of Submitted .circ Files

- **main.circ:** The top-level processor design connecting all major components.
- **control.circ:** Implements the Control Unit responsible for generating control signals based on instruction opcodes.
- **alu.circ:** Contains the Arithmetic Logic Unit performing arithmetic and logic operations.
- **register_file.circ:** The Register File consists of 32 registers, featuring two read ports, one write port, and four additional ports — Read Data 1, Read Data 2, Write Data, and Clock. The RegWrite control signal manages write enable functionality.
- **instruction_memory.circ:** Implements the Instruction Memory unit, responsible for storing and fetching instructions based on the Program Counter (PC) value.
- **data_memory.circ:** Implements the Data Memory unit, used for load (`lw`) and store (`sw`) operations during program execution.

- **alu_control.circ:** Implements the ALU Control unit, which generates specific control signals for the ALU based on the instruction's `funct` field and the main control unit's ALUOp output. It ensures correct execution of R-type and arithmetic I-type instructions.

4. Non-Trivial Logic Design

4.1. Control Unit

The Control Unit uses the 6-bit opcode and 6-bit function code fields to generate all necessary control signals such as RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, Jump and ALUOp. Each signal was derived logically based on the instruction type.

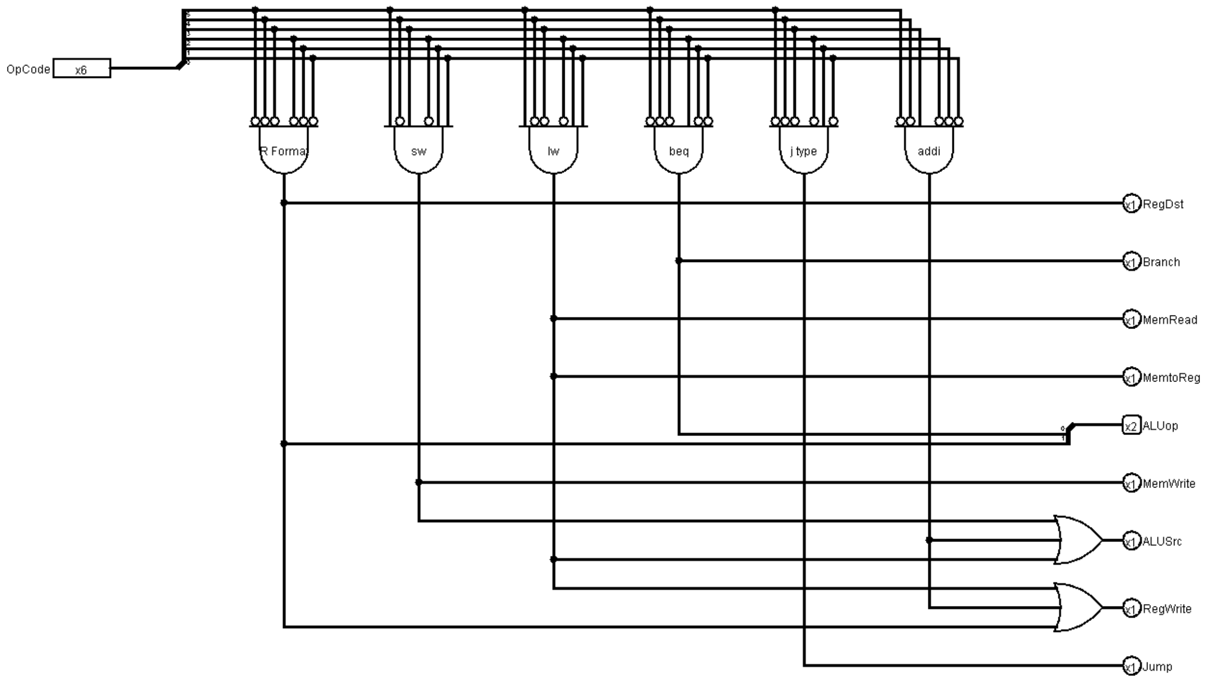


Figure 1: Circuit Implementation for Control Unit

4.2. ALU Control Unit

The ALU Control Unit takes the 2-bit `ALUOp` signal from the main Control Unit along with the 6-bit `funct` field from R-type instructions to determine the specific operation to be performed by the ALU. It generates control outputs such as `ALUControl[3:0]` that specify operations like addition, subtraction, AND, OR, and set-on-less-than (SLT). For I-type instructions, the ALU operation is determined directly from the `ALUOp` signal without using the function field.

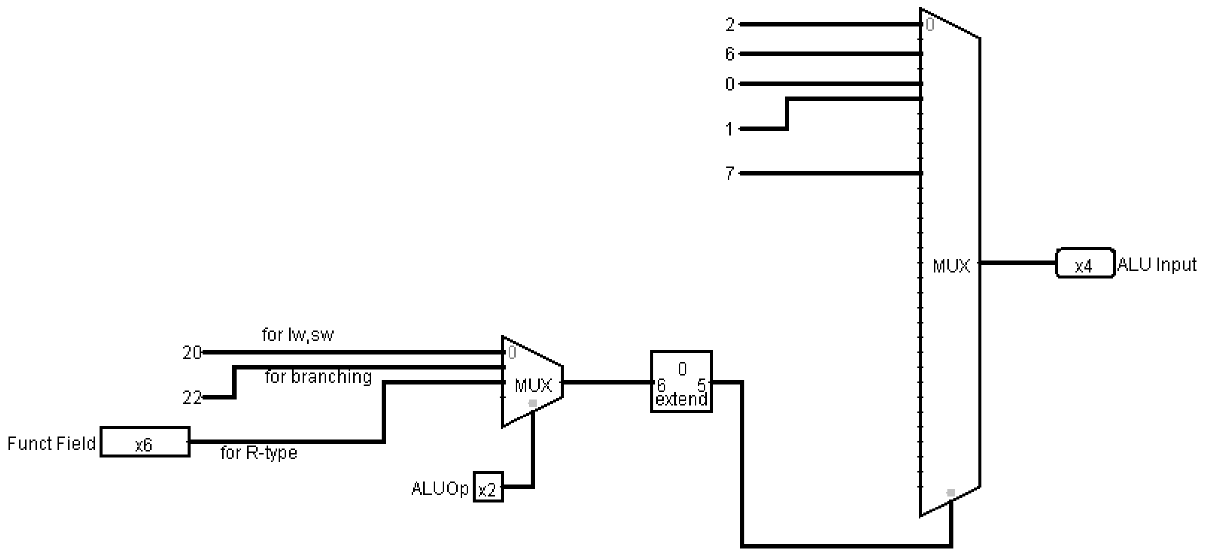


Figure 2: Circuit Implementation for ALU Control Unit

4.3. Program Counter Logic

The PC logic updates as follows:

- Default increment: PC = PC + 4
- Branch: PC = PC + 4 + (SignExtImm × 4) if the condition is met

Additional multiplexers were used to handle branching and jumping cleanly.

5. Modifications to Assignment 08 Components

- Updated Control Unit to `addi` instructions.
- Added multiplexers for ALUSrc and RegDst handling.
- Modified Control Unit logic to include signals for `lw`, `sw`, and `beq`.
- Integrated a proper PC incrementer and branch logic block.

6. MIPS Test Programs

6.1. MIPS Program 1: Addition and Memory Test

The following MIPS assembly program adds two numbers, stores the result in memory, and loads it back into a register.

```
.text

main:
    addi $t0, $zero, 5       # $t0 = 5
    addi $t1, $zero, 10      # $t1 = 10
    add  $t2, $t0, $t1       # $t2 = $t0 + $t1 = 15
    sw   $t2, 0($zero)       # Store $t2 (15) at memory[0]
    lw   $t5, 0($zero)       # Load memory[0] into $t5
halt:
    beq $zero, $zero, halt   # Infinite loop (halt)
```

Listing 1: Addition and Memory Test Program

Generated Hex File:

```
20080005
2009000a
01095020
ac0a0000
8c0d0000
1000ffff
```

6.2. MIPS Program 2: Multi-Load and Arithmetic Operation

This program demonstrates the use of `lw`, `sw`, and arithmetic instructions to perform computations on multiple values stored in memory. The program stores three numbers in memory, loads them into registers, performs additions, and stores the final result back into memory.

```
.text
.globl main

main:
    # Store 10 at M[100] (0x64)
    addi $t0, $zero, 10
    sw $t0, 100($zero)

    # Store 20 at M[104] (0x68)
    addi $t0, $zero, 20
    sw $t0, 104($zero)

    # Store 30 at M[108] (0x6C)
    addi $t0, $zero, 30
    sw $t0, 108($zero)

    # Load A (10) into $s0
    lw $s0, 100($zero)

    # Load B (20) into $s1
    lw $s1, 104($zero)

    # Load C (30) into $s2
    lw $s2, 108($zero)

    # Wait for load instructions to complete (NOPs)
    sll $zero, $zero, 0   # nop
    sll $zero, $zero, 0   # nop
    sll $zero, $zero, 0   # nop

    # $t1 = A + B (10 + 20)
    add $t1, $s0, $s1

    # $s3 = $t1 + C (30 + 30)
    add $s3, $t1, $s2       # $s3 should now be 60

    # Store 60 (0x3C) at M[112] (0x70)
    sw $s3, 112($zero)

# --- Infinite Halt ---
halt:
    beq $zero, $zero, halt   # Infinite loop (halt)
```

Listing 2: Multi-Load and Arithmetic Operation Program

Generated Hex File:

```
2008000a
ac080064
20080014
ac080068
2008001e
ac08006c
8c100064
8c110068
8c12006c
00000000
00000000
00000000
02114820
01329820
ac130070
1000ffff
```

6.3. Program 3: Arithmetic and Branch Integration Test

Objective: Test arithmetic and logical operations combined with conditional branching, verifying correct data flow between registers and memory.

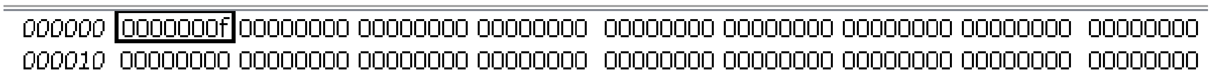
```
.text
.globl main

main:
    # Initialize registers
    addi $t0, $zero, 15     # $t0 = 15
    addi $t1, $zero, 10     # $t1 = 10
    addi $t2, $zero, 0      # $t2 = 0 (accumulator)

    # Compare and accumulate
    slt  $t3, $t1, $t0      # $t3 = 1 since 10 < 15
    beq  $t3, $zero, skip   # Skip addition if false (not taken)
    add  $t2, $t0, $t1      # $t2 = 15 + 10 = 25

skip:
    sub  $t4, $t2, $t1      # $t4 = 25 - 10 = 15
    and  $t5, $t4, $t0      # $t5 = 15 AND 15 = 15
    or   $t6, $t4, $t1      # $t6 = 15 OR 10 = 15

    # Store and reload results
    sw   $t2, 200($zero)    # Store 25 at memory[200]
    sw   $t5, 204($zero)    # Store 15 at memory[204]
    lw   $s0, 200($zero)    # Load memory[200]     $s0 (25)
    lw   $s1, 204($zero)    # Load memory[204]     $s1 (15)

# --- Infinite Halt ---
halt:
    beq $zero, $zero, halt  # Infinite loop (halt)
```

Listing 3: Arithmetic and Branch Integration Program

7. Simulation Results

7.1. Program 1 Results

Expected Outcome: Register $t2$ and memory[0] contain 15; $t5$ also contains 15.

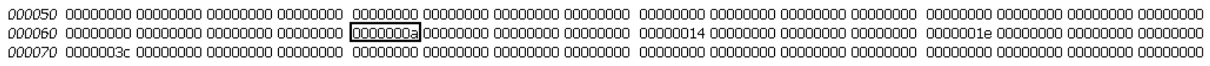


```
000000 0000000f 00000000 00000000 00000000  00000000 00000000 00000000 00000000  00000000
000010 00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000  00000000
```

Figure 3: Data Memory after executing add-store-load program

7.2. Program 2 Results

Expected Outcome: Memory at 100 store 10, at 104 store 20, at 108 store 30 and after addition memory at address 112 stores 60.



```
000050 00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000
000060 00000000 00000000 00000000 00000000  0000000a 00000000 00000000 00000000  00000014 00000000 00000000 00000000  0000001e 00000000 00000000 00000000
000070 0000003c 00000000 00000000 00000000  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000
```

Figure 4: Data Memory after executing program 2

7.3. Program 3 Results

This program performs multiple arithmetic and logical operations while testing branch execution. Since `$t1 < $t0`, the branch is not taken, and the addition executes. After execution: `$t2 = 25`, `$t4 = 15`, `$t5 = 15`, and `$t6 = 15`. The results are stored in memory locations 200 and 204, which are later loaded into `$s0` and `$s1`.

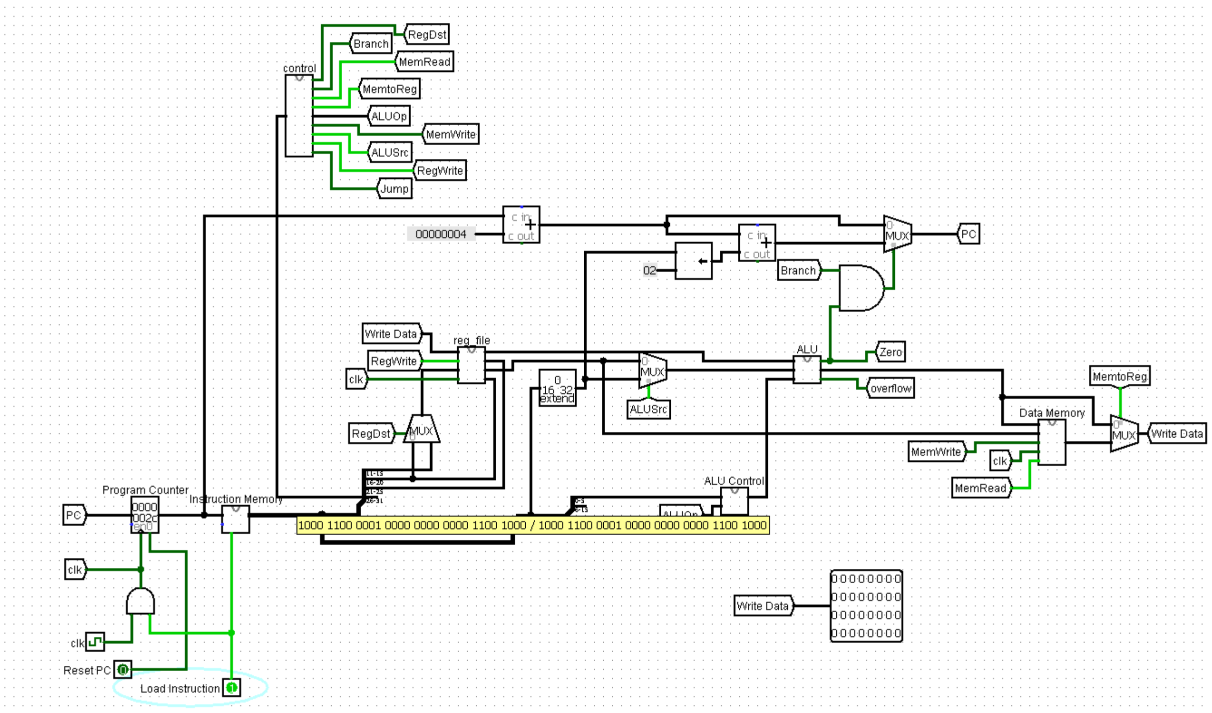


Figure 5: Intermediate image of circuit while executing program 3

8. Challenges Faced

- Implementation of the Program Counter.
- Debugging complex control signal dependencies.
- Managing multiple subcircuits and hierarchical design in Logisim.
- Ensuring synchronization between memory reads/writes.

9. Takeaways

- Developed a deep understanding of MIPS datapath design.

- Learned modular design principles using hierarchical circuits.
- Improved debugging and simulation skills using Logisim.

10. References

- Course lecture slides, CSL3130: Computer Organization and Architecture, IIT Jodhpur.
- ChatGPT (OpenAI), used for report formatting.
- Patterson, D. A., & Hennessy, J. L. (2017). *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers.