CSL3130: Computer Organization and Architecture
AY 2025–26

Lab Assignment 09 — MIPS Single-Cycle (with IM/DM)

Name: Atharva Honparkhe
Roll No.: B23EE1006

This report documents the design and verification of a single-cycle MIPS processor extended with Program Counter (PC), Instruction Memory (IM), and Data Memory (DM), supporting a full fetch–decode–execute–memory–writeback cycle, including `lw`, `sw`, and `beq`.

Contents

Abstract

This work extends the Assignment 08 MIPS datapath into a complete instruction-executing system by adding a Program Counter, Instruction Memory, Data Memory, and PC update logic for sequential and branch execution. The Control Unit is expanded to generate memory-related signals and a writeback multiplexer is introduced to select between ALU results and memory data. The processor is validated on three MIPS programs that exercise R-type, I-type, memory, and branch instructions, with results confirmed via Register File and Data Memory observations.

1 Objective

Extend the Assignment 08 MIPS processor to fetch and execute a sequence of instructions from an Instruction Memory, and to access a separate Data Memory for `lw`/`sw`. The design must properly implement PC update (PC $\leftarrow$ PC + 4 or branch target) and include Control Unit support for `lw`, `sw`, and `beq` with appropriate control signals (`MemRead`, `MemWrite`, `MemtoReg`).

2 Understanding of the Task

The processor is upgraded to automatically fetch, decode, execute, and write back results over a program rather than a single hard-wired instruction. This requires introducing: (i) a PC register that steps through instructions; (ii) an Instruction Memory that outputs the instruction addressed by PC; (iii) a Data Memory for load/store; and (iv) additional datapath glue (PC+4 adder, branch target adder, a Next-PC multiplexer, and a writeback multiplexer). The Control Unit is extended to generate new signals for memory and branch operations.

3 Assumptions

1. Register \$0 is hard-wired to zero; it is never written.
2. PC + 4 is computed by a dedicated adder (not the ALU).
3. Simulation proceeds by ticking the clock until the program completes.
4. Jump instructions are not required in this assignment.
5. All registers and memory are initially zero; constants are introduced via `addi`.
6. R-type must not be the first instruction (since initial register values are zero).
7. PC is initialized to 0 before simulation begins.
8. The IM is preloaded from a hex image assembled via MARS; RAM images may include spacer zeros between instructions as required by the toolchain.

4 Architecture Overview

4.1 Core Blocks

Program Counter (PC). A 32-bit register holding the address of the current instruction. On each rising edge, PC takes the Next-PC value selected by the control/branch logic.

Instruction Memory (IM). A RAM component addressed by PC. Its 32-bit data output is broadcast to the Control Unit and the datapath.

Data Memory (DM). A RAM component addressed by the ALU result for memory operations. Write data is Register File Read Data 2 (for `sw`); read data returns to the writeback MUX (for `lw`).

4.2 PC Update Logic

1. **Sequential step**: PC + 4 is computed by an adder.

2. **Branch target**: the sign-extended 16-bit immediate is shifted left by two and added to PC + 4:

$$\text{Target} = (\text{PC} + 4) + (\text{SignExt}(\text{imm}_{15:0}) \ll 2). \tag{1}$$

3. **Next-PC select**: a 2:1 MUX chooses between PC + 4 and Target under the condition Branch $\wedge$ Zero from the ALU.

4.3 Control Unit Additions

New opcodes supported: `lw` (100011), `sw` (101011), and `beq` (000100). The Control Unit generates:

- `MemRead`: asserted for `lw`
- `MemWrite`: asserted for `sw`
- `MemtoReg`: selects DM data (1) vs. ALU result (0) for writeback
- existing signals: `RegWrite`, `RegDst`, `ALUSrc`, `Branch`, and `ALUOp[*]`

4.4 Writeback MUX

An additional 2:1 MUX selects the value written to the Register File between the ALU result (R-type, `addi`) and DM read data (`lw`).

5 Submitted Circuit Files

The design is organized into modular Logisim subcircuits:

1. Top-level system integrating PC, IM, Control, ALU, RF, and DM.
2. 32-bit Program Counter (PC) with Next-PC input.
3. Instruction Memory (IM) with hex image preload.
4. Register File (32×32, two read ports, one write port).
5. Main Control Unit (opcode → control signals).
6. ALU datapath and helper logic.
7. ALU Control (ALUOp + funct decoding).
8. Data Memory (DM) controlled by `MemRead/MemWrite`.

6 Non-Trivial Logic (Control PC Path)

6.1 Control Encodings (indicative)

For `lw`: `MemRead`=1, `MemWrite`=0, `MemtoReg`=1, `RegWrite`=1, `ALUSrc`=1. For `sw`: `MemRead`=0, `MemWrite`=1, `MemtoReg`=0, `RegWrite`=0, `ALUSrc`=1. For `beq`: `Branch`=1, `RegWrite`=0, `ALUSrc`=0, ALU performs subtraction to set Zero.

6.2 PC/Branch Timing

The branch decision uses the ALU Zero flag. The sign-extended immediate is shifted ($\ll 2$) and added to PC + 4 in a dedicated adder. The Next-PC multiplexer is driven by Branch $\wedge$ Zero.

7 Modifications from Assignment 08

- Register File write-enable was corrected to affect only the selected destination register.
- Added `b23ee1006_pc_counter`, `b23ee1006_inst_mem`, and `b23ee1006_datamemory`, wiring them into the top-level datapath.
- Introduced a writeback MUX controlled by `MemtoReg`.
- Connected PC+4 adder, branch target adder, and Next-PC MUX per MIPS reference datapath.

8 Test Programs (`.asm`)

Below are representative programs equivalent to those used during validation. The actual hex images loaded into IM were generated in MARS; RAM images may include spacer zeros between instructions depending on the tool's addressing.

Program 1: Subtract two numbers and branch if result is zero

```
# Program 1: Subtract two numbers and branch if result is zero

    addi $t0, $zero, 15        # $t0 = 15
    addi $t1, $zero, 15        # $t1 = 15
    sw   $t0, 0($zero)         # Store 15 at memory[0]
    sw   $t1, 4($zero)         # Store 15 at memory[4]
    lw   $t2, 0($zero)         # Load value 15 from memory[0] into $t2
    lw   $t3, 4($zero)         # Load value 15 from memory[4] into $t3
    sub  $t4, $t2, $t3         # $t4 = 15 - 15 = 0
    beq  $t4, $zero, SAME      # If result == 0, branch to SAME
    addi $t5, $zero, 1         # (Skipped since $t4 == 0)
SAME: sw   $t4, 8($zero)       # Store 0 at memory[8]
END:  beq  $zero, $zero, END   # Infinite loop
```

Expected outcome: branch to `SAME`, M[8]=0.

Program 2: Add two numbers from memory and store result

```
# Program 2: Add two numbers from memory and store result

    addi $t0, $zero, 10        # $t0 = 10
    addi $t1, $zero, 20        # $t1 = 20
    sw   $t0, 0($zero)         # Store 10 at memory[0]
    sw   $t1, 4($zero)         # Store 20 at memory[4]
    lw   $t2, 0($zero)         # Load value 10 from memory[0] into $t2
    lw   $t3, 4($zero)         # Load value 20 from memory[4] into $t3
    add  $t4, $t2, $t3         # $t4 = 10 + 20 = 30
    sw   $t4, 8($zero)         # Store result (30) into memory[8]
END: beq  $zero, $zero, END    # Infinite loop to stop program
```

Expected outcome: $t0 = 0x0A,$t1=0x14, $t4 = 0x1E; M[0] = 0x0A, M[4] = 0x14, M[8] = 0x1E$.

Program 3: Bitwise AND/OR

```
# $t0=6, $t1=3 -> AND=2, OR=7; store both results
  addi $t0, $zero, 6      ; $t0 = 6
  addi $t1, $zero, 3      ; $t1 = 3
  sw   $t0, 0($zero)
  sw   $t1, 4($zero)
  lw   $t2, 0($zero)      ; $t2 = 6
  lw   $t3, 4($zero)      ; $t3 = 3
  and  $t4, $t2, $t3      ; $t4 = 2
  or   $t5, $t2, $t3      ; $t5 = 7
  sw   $t4, 8($zero)      ; M[8]  = 2
  sw   $t5, 12($zero)     ; M[12] = 7
END: beq  $zero, $zero, END   ; Infinite loop
```

Expected outcome: M[8]=2, M[12]=7; $t4=2, $t5=7.

9 Simulation Results

The following tables summarize the observed Register File and Data Memory contents after execution.

Program 1

Register	Hex	Decimal
$t0$	0x0F	15
$t1$	0x0F	15
$t4$	0x00	0

Address	Hex	Decimal
M[8]	0x00	0

Program 2

Register	Hex	Decimal
$t0$	0x0A	10
$t1$	0x14	20
$t4$	0x1E	30

Address	Hex	Decimal
M[0]	0x0A	10
M[4]	0x14	20
M[8]	0x1E	30

Program 3

Register	Hex	Decimal
$t0$	0x06	6
$t1$	0x03	3
$t4$	0x02	2
$t5$	0x07	7

Address	Hex	Decimal
M[8]	0x02	2
M[12]	0x07	7

10 Challenges and Mitigations

- **Register file resets:** values appeared to clear each cycle due to unintended global write-enable; fixed by gating WE only for the addressed destination.
- **IM/DM addressing mismatches:** resolved by auditing 32-bit buses and address alignment.
- **PC not updating:** added explicit PC+4 adder and Branch Adder; Next-PC MUX controlled by (`Branch` && `Zero`).
- **Control signal alignment:** verified `MemtoReg`/`RegWrite` timing with datapath writeback.
- **Signal tracing:** labeled tunnels and used probes to debug 32-bit interconnects.

11 Takeaways

- End-to-end view of the MIPS instruction cycle across fetch, decode, execute, memory, and writeback.
- Practical understanding of PC/branch mechanics and memory-mapped datapaths.
- Reinforced the role of control signals in orchestrating datapath movement.
- Experience in modular hardware design and systematic hardware debugging.

12 Deliverables Checklist

- Assumptions and task understanding ✓
- Description of each submitted `.circ` file ✓
- Non-trivial logic for Control and PC path ✓
- Modifications over Assignment 08 ✓
- Test programs (`.asm`) and expected effects ✓
- Simulation evidence summarised in tables (screenshots can be inserted where appropriate) ✓