MIPS Processor (Without Memory) — Design & Simulation

Course: Digital Design and Computer Organization

Report

Submitted by:
B23EE1006

Date of submission: October 27, 2025

Contents

1. Introduction

This report documents the design and simulation of a simplified MIPS processor without instruction memory (IM) and data memory (DM). The processor supports a subset of MIPS instructions:

- R-type: `add`, `sub`, `slt`, `and`, `or`
- I-type: immediate arithmetic/logical counterparts (e.g., `addi`), and `beq`

Key design constraints from the assignment:

- Instructions are supplied via a special register named `$im0`.
- Branch address for `beq` is written to a special register `$im1`.
- There is no program counter; assume current instruction address is `0x00000000`. Branch target is computed as `PC+4 + (sign_ext(imm) « 2)` and stored in `$im1`.
- All data storage is in the register file; there is no external Data Memory.

2. Assumptions

The following assumptions were made during design and documentation:

1. Instruction format and opcodes follow the standard MIPS encoding unless noted otherwise.
2. The register file has 32 registers (R0–R31). Register R0 is hardwired to zero.
3. The special registers `$im0` and `$im1` are mapped inside the register file (for example using reserved register numbers, say R30 = `$im0` and R31 = `$im1`) OR implemented as separate dedicated registers connected to the register file ports; document which approach you used in the design files.
4. The ALU outputs a 32-bit result and a single-bit Zero flag that is high when result==0.
5. For branch offset calculation, shift-left-by-2 is applied to the sign-extended immediate as per MIPS semantics.
6. The design uses a global clock (for register file write timing). All combinational elements are assumed to be zero-delay for simulation snapshots; sequential storage is clocked.
7. No hazards (since only single instruction execution at a time). No pipeline implemented.

3. Understanding of the Task

The goal is to integrate the previously created components (ALU, ALU Control, Register File, Control Unit) into a cohesive datapath that can execute a single MIPS instruction stored at `$im0`. The processor must:

- Decode the instruction (using the control unit).
- Read the required registers from the register file.
- Use ALU (and ALU control) to compute results or compare operands (for `beq`).
- Write results back to the register file at the correct destination (using RegDst and RegWrite).
- Compute branch target address when `beq` and write that address to `$im1`.

4. Files Submitted

Below list the circuit files included in the submission. Replace the placeholder names with the actual filenames in your submitted zip. Each filename is expected to be prefixed by your roll number as required (e.g., `B23EE1006_ALU.circ`).

Filename	Description
`B23EE1006_CompleteProcessor.circ`	Top-level processor integrating all components. Instruction input provided via `$im0`, and branch target address written to `$im1`. Handles overall datapath connections among ALU, Control Unit, and Register File.
`RegisterFile.circ`	Register file with two read ports and one write port, controlled by a single global clock. Supports `RegWrite` signal for data write and includes reserved registers `$im0` (instruction) and `$im1` (branch address).
`B23EE1006_ALU.circ`	32-bit Arithmetic Logic Unit supporting operations: add, sub, slt, and, or. Outputs both 32-bit result and a Zero flag (set when the result is zero).
`ALUCtrlUnit.circ`	ALU Control Unit that interprets the function field (for R-type) and ALUOp signal from the Control Unit to generate 4-bit control signals for the ALU.
`ControlUnit.circ`	Main Control Unit decoding Opcode from instruction to generate signals such as `RegDst`, `ALUSrc`, `RegWrite`, `Branch`, and `ALUOp`. Governs control flow for R-type, I-type, and branch instructions.

Note: If any of the above files don't exist in your submission, replace or remove the corresponding entries accordingly.

5. Component Details and Non-trivial Logic

This section provides detailed explanations of the main components implemented in the MIPS Processor design. Each component performs a specific role in executing MIPS instructions, and together they form the complete single-cycle processor.

5.1. B23EE1006_CompleteProcessor.circ

This is the top-level circuit that connects all submodules such as the Control Unit, Register File, ALU, and ALU Control. It acts as the main datapath through which the instruction flows from decoding to execution.

- **Purpose:** Integrates all processor components into a single working circuit.
- **Instruction Input:** The instruction is loaded into the special register `$im0`, which acts like a mini instruction memory.
- **Branch Handling:** When a branch (`beq`) is executed, the computed branch address is stored in `$im1`.

- **Data Flow:**
  1. The Control Unit decodes the instruction and sets control signals.
  2. The Register File provides source operands to the ALU.
  3. The ALU performs the operation and sends the result back to the Register File (if required).
- **Key Logic:** Proper synchronization between modules ensures correct execution for each instruction type (R-type, I-type, and Branch).

In simple words: This file brings all smaller circuits together — it is the "brain and body" of your processor where every component interacts in one single clock cycle.

—

5.2. RegisterFile.circ

The Register File stores all the temporary and result data used by the processor during execution.

- **Structure:** Contains 32 general-purpose registers (R0–R31).
- **Ports:**
  - Two 5-bit read addresses and one 5-bit write address.
  - Two 32-bit outputs (`Read Data 1, Read Data 2`) and one 32-bit input (`Write Data`).
  - A control signal (`RegWrite`) and a clock signal.
- **Special Registers:**
  - `$im0` – stores the instruction to execute.
  - `$im1` – stores the branch address for `beq`.
- **R0 Behavior:** Register R0 is always zero (writes are ignored).
- **Non-trivial Design:** The register file was made clock-driven, ensuring that data is only written on the clock's rising edge.

In simple words: The register file works like a small memory bank for your processor. It provides data to the ALU and stores the output when an instruction is executed.

—

5.3. B23EE1006_ALU.circ

The Arithmetic Logic Unit (ALU) performs all the calculations in the processor. It takes two 32-bit inputs and produces a 32-bit result along with a Zero flag.

- **Inputs:**
  - Operand A from Register File.
  - Operand B from either Register File or immediate (based on `ALUSrc` signal).
- **Outputs:**
  - 32-bit result.
  - 1-bit `Zero` flag (set when result = 0).

- **Supported Operations:**

Control Code	Operation
0000	AND
0001	OR
0010	ADD
0110	SUB
0111	SLT (Set on Less Than)

- **Special Behavior:** The ALU automatically sets `Zero = 1` for `beq` when both operands are equal.

In simple words: The ALU is like the calculator of the processor — it adds, subtracts, compares, or performs logic operations on the data coming from registers.

—

5.4. ALUCtrlUnit.circ

The ALU Control Unit translates the signals coming from the Control Unit and the instruction's function field into a 4-bit control code that the ALU can understand.

- **Inputs:**
  - 2-bit `ALUOp` from the Control Unit.
  - 6-bit `funct` field (for R-type instructions).
- **Output:** A 4-bit ALU control signal that selects the specific operation.
- **Logic:**
  - If `ALUOp` = 00 → Perform addition (for immediate arithmetic).
  - If `ALUOp` = 01 → Perform subtraction (for `beq`).
  - If `ALUOp` = 10 → Use `funct` field to determine operation (for R-type).

In simple words: This unit acts like a translator — it reads the type of instruction and tells the ALU exactly which operation to perform.

—

5.5. ControlUnit.circ

The Control Unit is responsible for interpreting the instruction opcode and generating all the control signals required to operate the datapath correctly.

- **Input:** 6-bit `Opcode` from the instruction.
- **Outputs:** `RegDst`, `ALUSrc`, `RegWrite`, `Branch`, and 2-bit `ALUOp`.
- **Behavior:**
  - `R-type: RegDst=1, ALUSrc=0, RegWrite=1, Branch=0.`
  - `I-type (addi/subi): RegDst=0, ALUSrc=1, RegWrite=1, Branch=0.`

  - Branch (beq): RegDst=X, ALUSrc=0, RegWrite=0, Branch=1.

- **Non-trivial Design:** The control unit ensures synchronization between arithmetic and branch operations using the same control logic, which avoids unnecessary duplication of signals.

In simple words: This is the decision-maker of the processor. It looks at the instruction and turns on or off the right parts of the circuit so the operation happens correctly.

—

Overall Summary: Each of these components — the Complete Processor, Register File, ALU, ALU Control, and Control Unit — plays a specific role. The Control Unit decides what needs to be done, the Register File stores data, the ALU performs computations, and the Complete Processor connects them all into a functioning single-cycle MIPS processor.

6. Instruction Formats and Example Encodings

Standard MIPS formats used:

6.1. R-type

opcode[31:26] rs[25:21] rt[20:16] rd[15:11] shamt[10:6] funct[5:0]

6.2. I-type

opcode[31:26] rs[25:21] rt[20:16] immediate[15:0]

Example Instructions (hex and description):

- `add $t0, $t1, $t2` (R-type): opcode=0x00, funct=0x20.
- `addi $t0, $t1, 10` (I-type): opcode=0x08, immediate=0x000A.
- `beq $t0, $t1, -4` (I-type): opcode=0x04, immediate=0xFFFC.

Include the exact binary/hex encodings you used in your testbench here. (Replace the placeholders below with the encodings used in your Logisim testbench.)

```
# Example (replace with your actual encodings used in .circ)
# add $8, $9, $10  -> 0x012A4020   (example)
# addi $8, $9, 5   -> 0x21280005   (example)
```

7. Test Instructions and Simulation Results

This section should present the tests you ran and include screenshots of the register file before and after instruction execution. Below are recommended test cases and a template for documenting results.

7.1. Test Cases

1. **R-type add:** Initialize `$t1`=5, `$t2`=7; instruction `add $t0, $t1, $t2`; expected `$t0`=12.
2. **I-type addi:** Initialize `$t1`=3; instruction `addi $t0, $t1, 10`; expected `$t0`=13.
3. **R-type sub:** Initialize `$t1`=15, `$t2`=6; instruction `sub $t0, $t1, $t2`; expected `$t0`=9.
4. **R-type slt:** Initialize `$t1`=2, `$t2`=5; instruction `slt $t0, $t1, $t2`; expected `$t0`=1.

5. **beq branch:** Initialize `$t0`=4, `$t1`=4; instruction `beq $t0, $t1, offset`; expected: `$im1` contains computed branch address (PC+4 + (offset « 2)).

7.2. Simulation Result Template

Insert your actual simulation screenshots in the following placeholders. For each test case include:

- Image: Register file before execution (show the relevant registers).
- Image: Register file after execution (show the updated register).
- For branch tests: screenshot showing `$im1` value.
- A short table with expected vs actual values.

Table 2: Example Result Table (replace with your actual results)

Test	Instruction	Expected Result	Actual Result
1	add $t0,$t1,$t2	$t0 = 12	$t0 = 12
2	addi $t0,$t1,10	$t0 = 13	$t0 = 13
3	beq $t0,$t1,offset	$im1 = 0x00000010	$im1 = 0x00000010

8. Challenges Faced and Solutions

Throughout the design and simulation of the MIPS processor, several challenges were encountered. Each issue required careful debugging, signal tracing, and validation within Logisim to ensure the correct working of every module. The following points describe the main difficulties faced along with the approaches used to overcome them:

- **Wiring Complexity:** As the number of connections between the Control Unit, ALU, and Register File increased, the datapath became visually cluttered and confusing. To solve this, tunnels and clearly labeled wires were used extensively in Logisim to organize signals. Grouping related connections (e.g., all control signals) also made debugging faster and improved readability.
- **RegDst and ALUSrc Selection Errors:** During early integration, incorrect bit selection for `rd` and `rt` fields caused wrong register writes for I-type instructions. The issue was resolved by verifying instruction field mappings (bits [25:21], [20:16], [15:11]) and testing both R-type and I-type instructions individually to confirm proper MUX operation.
- **Sign-Extension and Branch Shift Issues:** Incorrect sign-extension of immediate values, especially for negative constants, caused unexpected arithmetic errors and branch miscalculations. This was corrected by implementing a dedicated sign-extension unit that preserves the sign bit and by ensuring the branch offset is shifted left by 2 before addition with (PC + 4), following MIPS standards.
- **Zero Detection for Branch Instructions:** Initially, the `Zero` flag from the ALU did not assert properly during subtraction, resulting in missed branches. The problem was traced to an incomplete zero-comparison condition within the ALU. The fix involved adding an equality comparator that sets the `Zero` output high only when all 32 result bits are 0. This ensured correct branch execution for `beq` instructions.

- **Clock Synchronization and Timing:** Ensuring correct timing for register writes was crucial. Early tests showed inconsistent data updates due to asynchronous signal changes. The issue was resolved by connecting all sequential elements (especially the Register File) to a single global clock and verifying that data was sampled on the correct rising edge. Simulation snapshots were captured only after the clock pulse to confirm stable data writes.
- **Control Signal Mismatches:** The ALU Control initially did not match the ALU's internal operation codes, causing incorrect arithmetic results. The control mapping between `ALUOp` and the 4-bit ALU control signal was reviewed and standardized across all modules, ensuring consistency between the Control Unit and ALU Control Unit.
- **Testing and Debugging:** Debugging multi-component circuits without waveforms required visual inspection of data paths. To simplify this, labels were placed on key buses such as `ReadData1`, `ReadData2`, `ALUResult`, and `Zero`. Step-by-step simulation with manual clock pulses helped confirm the correct flow of data through each component.

—

9. Takeaways

The completion of this lab provided practical insight into how individual digital components combine to form a working processor. It emphasized both the conceptual understanding and hands-on implementation of MIPS architecture principles.

- Designing a processor highlights the importance of **modular integration** — even correctly working submodules can fail when combined if signal interfacing is inconsistent.
- **Control signal timing** and proper bit-field decoding are critical for stable and predictable operation in digital systems.
- The assignment reinforced how each block (ALU, Control Unit, Register File) plays a coordinated role, and even simple errors like reversed wiring or missing sign bits can affect the entire datapath.
- Working without memory modules demonstrated how instruction and data storage can be abstracted using registers like `$im0` and `$im1`, providing a deeper understanding of MIPS execution flow.
- **Debugging experience** in Logisim developed a strong appreciation for design clarity — clean wiring, consistent naming, and modularity are essential for scalable CPU design.
- Overall, the project bridged theory and practice, helping connect instruction-level concepts with their real hardware representation.

10. Concluding Remarks

This report presents the complete design, integration, and testing of a simplified single–cycle MIPS processor constructed using modular digital components. The design integrates the core building blocks—`ALU`, `ALUCtrlUnit`, `ControlUnit`, and `RegisterFile`—within the top-level circuit `B23EE1006_CompleteProcessor.circ`. The processor successfully executes arithmetic, logical, and branch instructions, following MIPS instruction set conventions, and adheres strictly to the "no external memory" requirement.

Each module was designed and verified individually before integration to ensure signal compatibility and functional correctness. The `RegisterFile` manages register-level data communication and includes dedicated registers (`$im0`, `$im1`) for instruction input and branch address output. The `ControlUnit` and `ALUCtrlUnit` collaboratively generate precise control signals to steer the ALU's operation, while the `ALU` performs the core arithmetic and logical computations. All modules were connected in a clean and readable datapath within the complete processor circuit.

Through extensive simulation, the processor was verified for R-type, I-type, and branch instructions (`add`, `sub`, `addi`, `slt`, and `beq`). The outcomes confirmed correct data flow, proper Zero flag detection, and accurate branch address calculation.

This assignment reinforced the importance of modular design, synchronization, and interconnection between digital components. It also demonstrated how theoretical MIPS concepts translate into functional hardware behavior through structured circuit implementation and testing.

Prepared by: B23EE1006