# Bangladesh University of Engineering and Technology

## Department of Computer Science and Engineering

**CSE 210:** Computer Architecture Sessional

**Assignment-3 : MIPS Design and Simulation**


**Submitted By:** (A1-Group 06)

- 2105002 - Khalid Hasan Tuhin
- 2105014 - K.M. Mehemud Azad
- 2105015 - Arnob Biswas


**Date of Submission: 11 December, 2024**

# Contents

# 1 Introduction

A processor is an important part of a computer. It follows instructions and performs tasks like math, decision-making, and controlling other parts of the computer. It's also called the Central Processing Unit (CPU). The CPU is like the brain of the computer.One type of processor is called a MIPS processor. MIPS stands for "Microprocessor without Interlocked Pipeline Stages." It is known for being simple, fast, and efficient. In this project, we are building a small 8-bit MIPS processor.

The CPU has the following main parts:

1. **Program Counter (PC):** The program counter keeps track of where the next instruction is stored. After every clock it adds 1 to its previous address. The value it stores is used as the Instruction Memory Address.

2. **Arithmetic Logic Unit (ALU):** This part solves math problems (like adding and subtracting) and checks conditions (like comparing numbers).

3. **Registers:** These are small memory spaces where the CPU temporarily stores data while working.

   - **$zero:** This register always holds the value 0. It's used for assigning one register value to another register.
   - **$t0,$t1,$t2,$t3,$t4:** General-purpose registers used for calculations and storing data.
   - **$sp:** This register is used for stack memory operations(store and load).

4. **Pipelining Registers:** These registers are used for storing necessary data and controls for pipelining

   - **IF-ID:** The IF/ID register holds instruction and program counter (PC) values fetched in the Instruction Fetch (IF) stage, passing them to the Instruction Decode (ID) stage for processing.
   - **ID-EX:** The ID/EX register stores control signals, operands, and instruction-related data from the Instruction Decode (ID) stage, forwarding them to the Execution (EX) stage for further processing.
   - **EX-MEM:** The EX/MEM register holds the results of the Execution (EX) stage, such as computed addresses, ALU results, and control signals, forwarding them to the Memory Access (MEM) stage.
   - **MEM-WB:** The MEM/WB register stores the data read from memory or the ALU results from the Memory Access (MEM) stage, passing them to the Write-Back (WB) stage for final updates to registers.

5. **Control Unit:** This part acts like a manager, telling the ALU and registers what to do.

6. **Hazard Detection Unit:** This part identifies and resolves hazards (data, control, or structural) to maintain the correct instruction execution order. It stalls or forwards instructions as needed to prevent conflicts in the pipeline.

7. **Forwarding Unit :** This part resolves data hazards by rerouting data directly from pipeline stages to dependent instructions, avoiding stalls.

8. **Data Memory:** This is the memory used to store data and results. It can hold up to 256 bytes of information.

9. **Stack Memory:** The stack is like a pile where the processor keeps temporary data. The stack pointer $sp helps keep track of the stack. For example :

    - **sw $t0, 0($sp):** This instruction saves the value in register $t0 into the memory address currently pointed to by the stack pointer ($sp).
    - **lw $t0, 2($sp):** This instruction loads the value from the memory address at $sp + 2 into the register $t1.

**Project Overview:** This project involves the design and implementation of an 8-bit MIPS processor with pipelining to enhance instruction throughput. The processor is divided into five pipeline stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write-Back (WB), enabling simultaneous execution of multiple instructions. Key components include pipeline registers for inter-stage communication, a hazard detection unit to manage control and data hazards, and a forwarding unit to resolve data dependencies efficiently. The project demonstrates a functional, reduced instruction set capable of arithmetic, logical, memory, and control operations, focusing on performance improvement through pipelining while addressing hazards effectively.

## 2 MIPS Instruction Format

MIPS instructions are 20 bits long and follow these three formats:

| Instruction Type | Format | | | | |
|---|---|---|---|---|---|
| **R-type** | Opcode<br>4-bits | Src Reg 1<br>4-bits | Src Reg 2<br>4-bits | Dst Reg<br>4-bits | Shft Amnt<br>4-bits |
| **I-type** | Opcode<br>4-bits | Src Reg 1<br>4-bits | Src Reg 2<br>4-bits | Address / Immediate<br>8-bits | |
| **J-type** | Opcode<br>4-bits | Target Jump Address<br>8-bits | | 0<br>4-bits | 0<br>4-bits |

MIPS Instruction Format

## 3 Instruction Set Assignment

Our group (A1 - group 6) is using the following instruction codes:

- **andi** - Opcode: `0` (AND with a constant value)
- **addi** - Opcode: `1` (Add a constant to a number)
- **or** - Opcode: `2` (Logical OR operation)
- **sll** - Opcode: `3` (Shift bits to the left)
- **subi** - Opcode: `4` (Subtract a constant from a number)
- **nor** - Opcode: `5` (NOR operation)
- **lw** - Opcode: `6` (Load data from memory)
- **add** - Opcode: `7` (Add two numbers)
- **bneq** - Opcode: `8` (Check if two numbers are not equal)
- **sw** - Opcode: `9` (Store data into memory)
- **sub** - Opcode: `10` (Subtract two numbers)
- **ori** - Opcode: `11` (OR with an immediate value)
- **j** - Opcode: `12` (Jump to a different instruction)
- **and** - Opcode: `13` (Logical AND operation)

- **beq** - Opcode: `14` (Check if two numbers are equal)
- **srl** - Opcode: `15` (Shift bits to the right)

These instructions allow the processor to perform math, logic, memory, and control tasks efficiently.

# 4 Instruction Set Decode

## 4.1 Instruction Set With Instruction ID

| Instruction ID | Category | Type | Instruction |
|---|---|---|---|
| A | Arithmetic | R | add |
| B | Arithmetic | I | addi |
| C | Arithmetic | R | sub |
| D | Arithmetic | I | subi |
| E | Logic | R | and |
| F | Logic | I | andi |
| G | Logic | R | or |
| H | Logic | I | ori |
| I | Logic | R | sll |
| J | Logic | R | srl |
| K | Logic | R | nor |
| L | Memory | I | sw |
| M | Memory | I | lw |
| N | Control-conditional | I | beq |
| O | Control-conditional | I | bneq |
| P | Control-unconditional | J | j |

Table 1: Instruction Set With Instruction ID

## 4.2 Instruction Set With Op-Code

| Opcode | Instruction ID | Category | Instruction |
|---|---|---|---|
| 0000 | F | Logic | andi |
| 0001 | B | Arithmetic | addi |
| 0010 | G | Logic | or |
| 0011 | I | Logic | sll |
| 0100 | D | Arithmetic | subi |
| 0101 | K | Logic | nor |
| 0110 | M | Memory | lw |
| 0111 | A | Arithmetic | add |
| 1000 | O | Control-conditional | bneq |
| 1001 | L | Memory | sw |
| 1010 | C | Arithmetic | sub |
| 1011 | H | Logic | ori |
| 1100 | P | Control-unconditional | j |
| 1101 | E | Logic | and |
| 1110 | N | Control-conditional | beq |
| 1111 | J | Logic | srl |

Table 2: Instruction Set with Opcode

# 5 Complete Circuit Diagram (of all components)

## 5.1 Instruction Memory

The instruction memory circuit stores instructions in hexadecimal format and reads them sequentially on each clock cycle. Each 20-bit instruction is split into 5 parts, with two parts merged to create the jump address. An AND gate is used to identify the special-purpose ($sp) register, which is the 7th register in this implementation. This differentiation allows the circuit to distinguish between stack and data memory during memory operations.

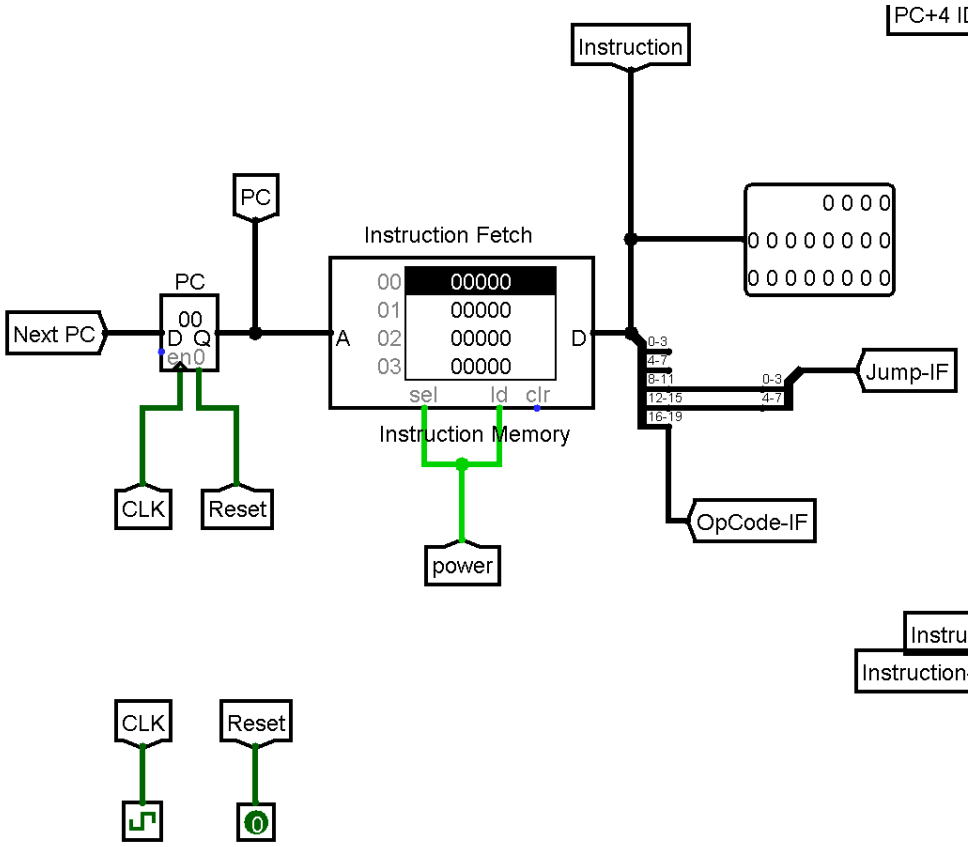



Figure 1: Instruction Memory

## 5.2 Register File

The register file stores the following registers:

- $zero (0000)
- $t0 (0001)
- $t1 (0010)
- $t2 (0011)

- $t3 (0100)
- $t4 (0101)
- $sp (0111)

It performs read and write operations as specified by the instructions provided to it.

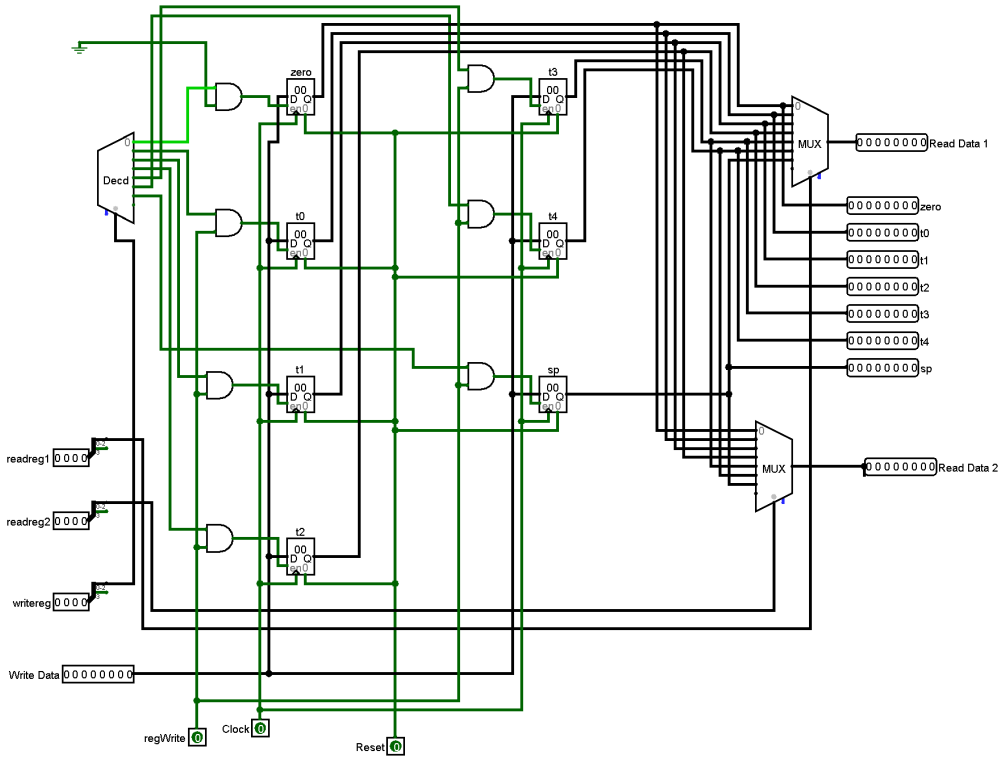



Figure 2: Register File

## 5.3 Control Unit

The **Control Unit** is a microprogrammed circuit responsible for generating the control signals needed to execute various operations in the processor. **Microprogramming** is a technique where the control sequences are stored as **binary instructions** in a **ROM** (Read-Only Memory). These sequences are accessed using specific **addresses**, allowing the control unit to provide precise control signals for different instructions.

In our design, we first calculated the **binary control sequences** for each instruction based on their functionality. These binary sequences were then **converted into hexadecimal format** and stored in the ROM. When an instruction is executed, the control unit retrieves the corresponding control sequence from the ROM, ensuring that the processor functions correctly.

### 5.3.1 Control Unit BreakDown

| Instruction | RegDst | AluSrc | MemRead | RegWrite | MemToReg | MemWrite | beq | bne | ALUOp | jump | AluOp | Hex Value |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| andi | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 011 | 0 | 0 | 506 |
| addi | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 000 | 0 | 000 | 500 |
| or | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 010 | 0 | 010 | 904 |
| sll | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 101 | 0 | 101 | 90a |
| subi | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 001 | 0 | 001 | 502 |
| nor | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 100 | 0 | 100 | 908 |
| lw | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 000 | 0 | 000 | 780 |
| add | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 000 | 0 | 000 | 900 |
| bneq | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 001 | 0 | 001 | 012 |
| sw | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 000 | 0 | 000 | 440 |
| sub | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 001 | 0 | 001 | 902 |
| ori | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 010 | 0 | 010 | 504 |
| j | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 000 | 1 | 000 | 001 |
| and | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 011 | 0 | 011 | 906 |
| beq | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 001 | 0 | 001 | 022 |
| srl | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 011 | 0 | 110 | 90c |

Table 3: Control Sequence for Instructions

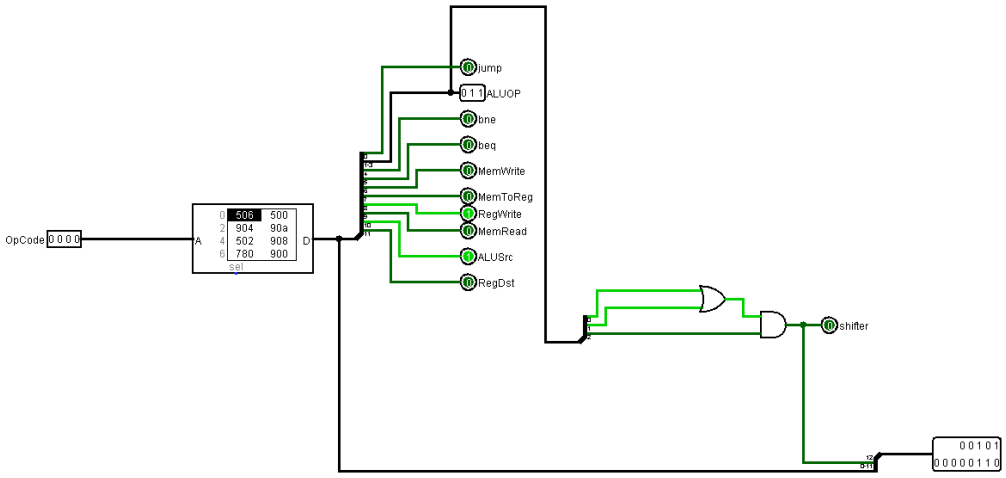



Figure 3: Control

## 5.4 Memory

The memory system distinguishes between stack and data memory using the `sp` flag from the instruction. When the `$sp` register is identified, the system accesses the stack memory. The stack memory follows the standard stack behavior, storing data starting from the last address `1111 1111` and decreasing the `$sp` register as new values are pushed. In contrast, the data memory begins storing values from address `0000 0000`.

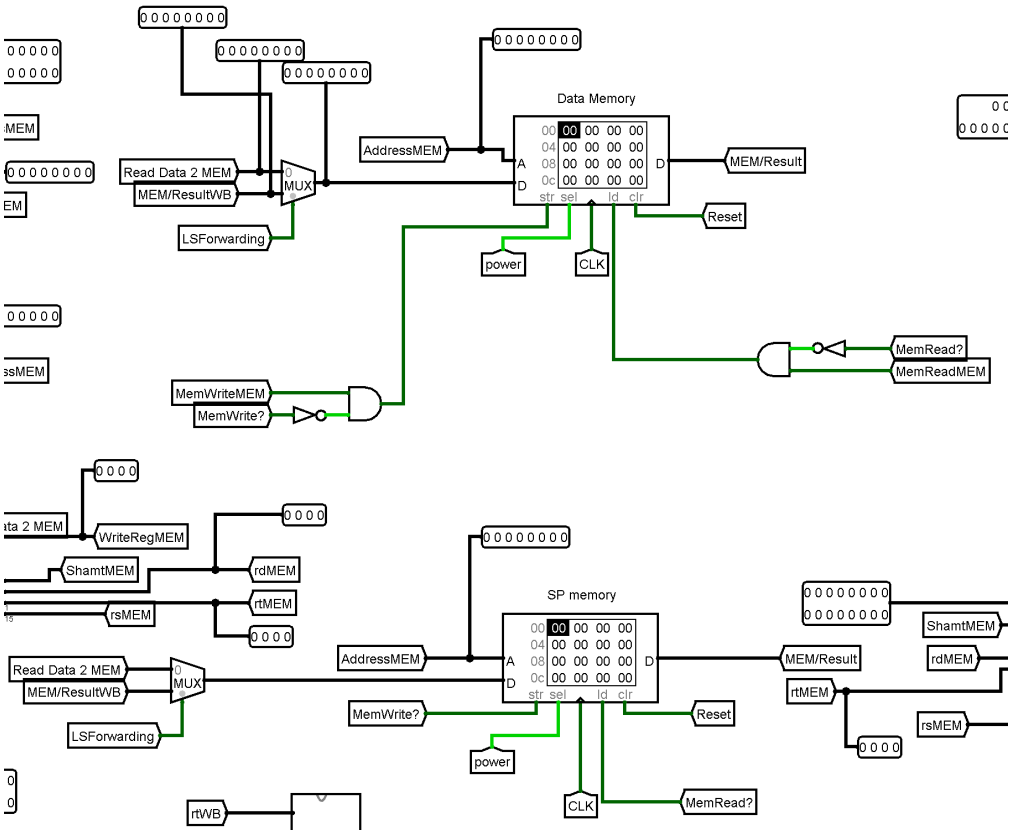
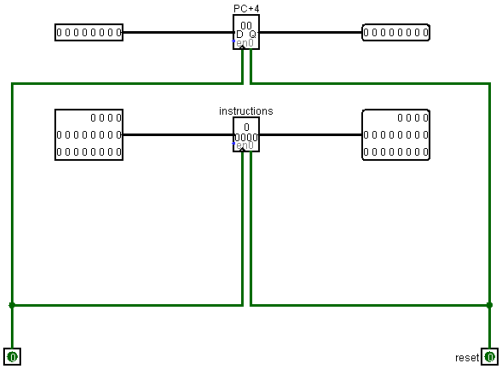


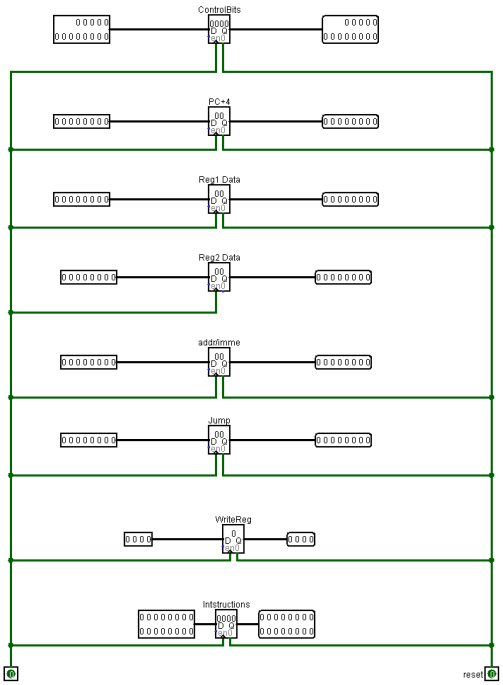

Figure 4: Memory

## 5.5 Pipeline

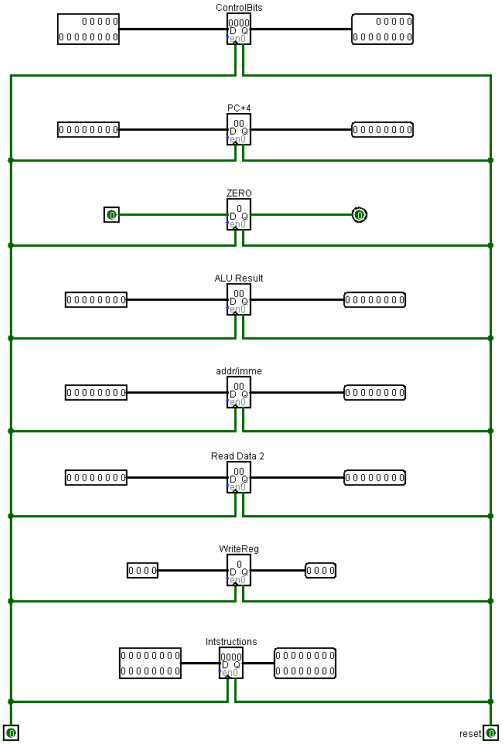
### 5.5.1 Pipeline Registers



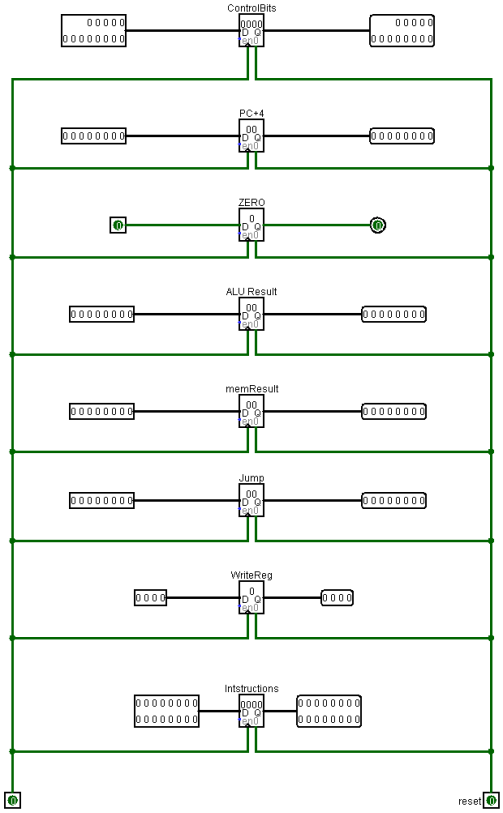

(a) IF/ID Register




(b) ID/EX Register




(c) EX/MEM Register




(d) MEM/WB Register

Figure 5: Pipeline Registers

### 5.5.2 Forwarding

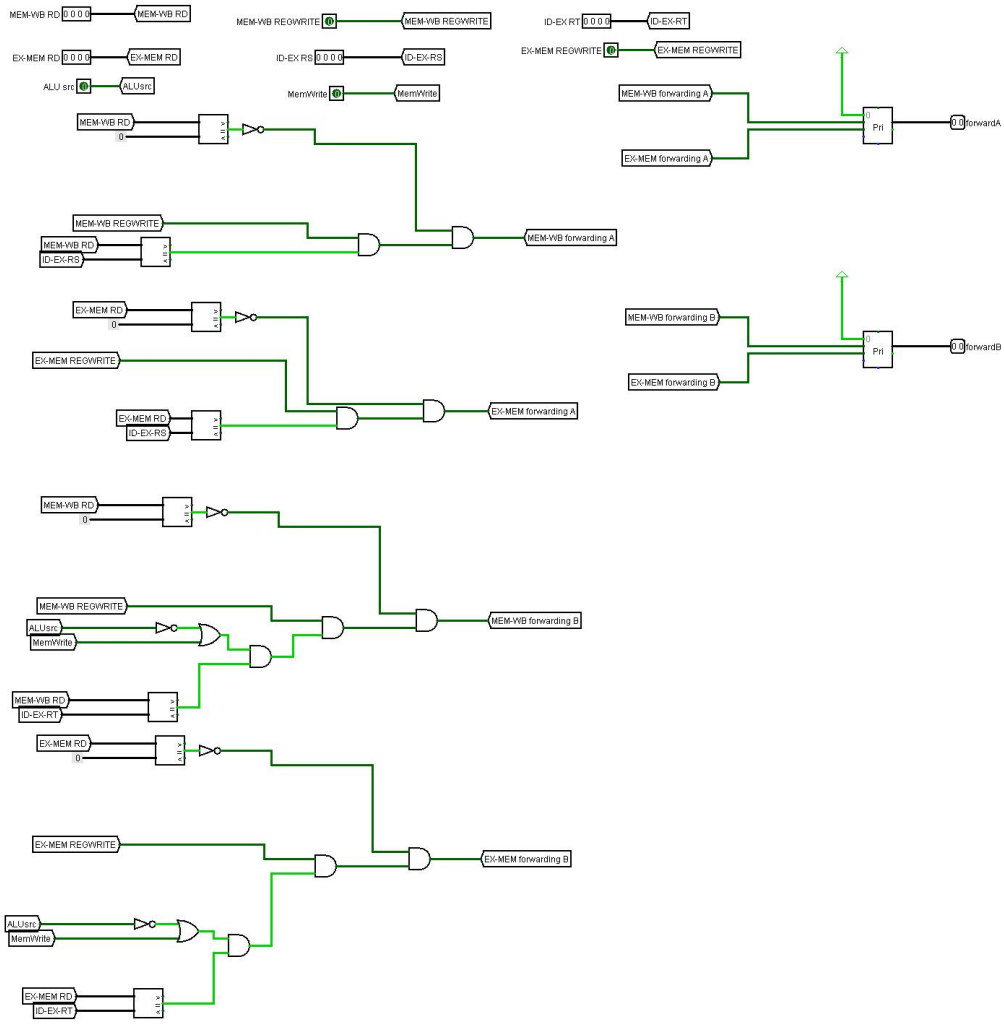



Figure 6: Forwarding Unit

### 5.5.3 Normal Forwarding

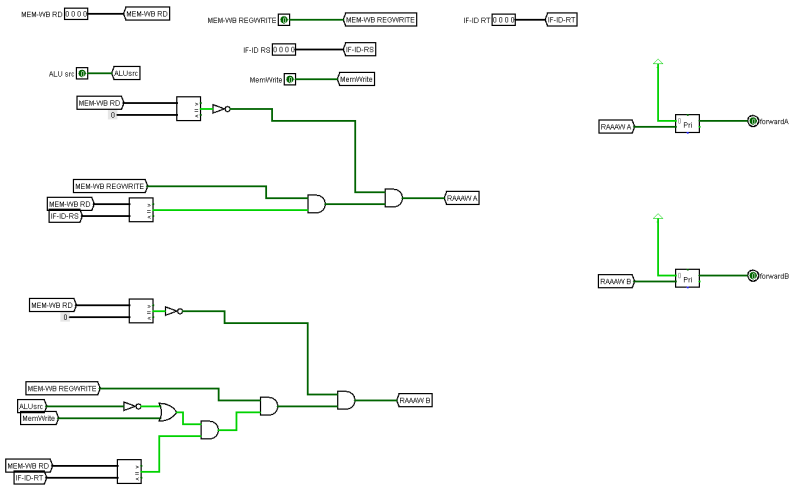
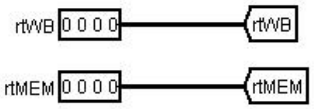



Figure 7: RAAAW Detection Unit



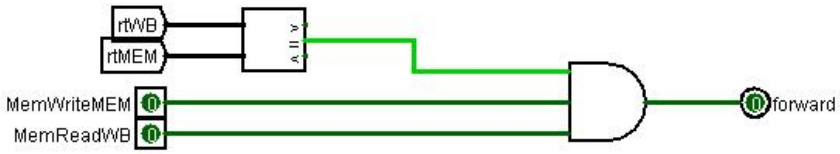
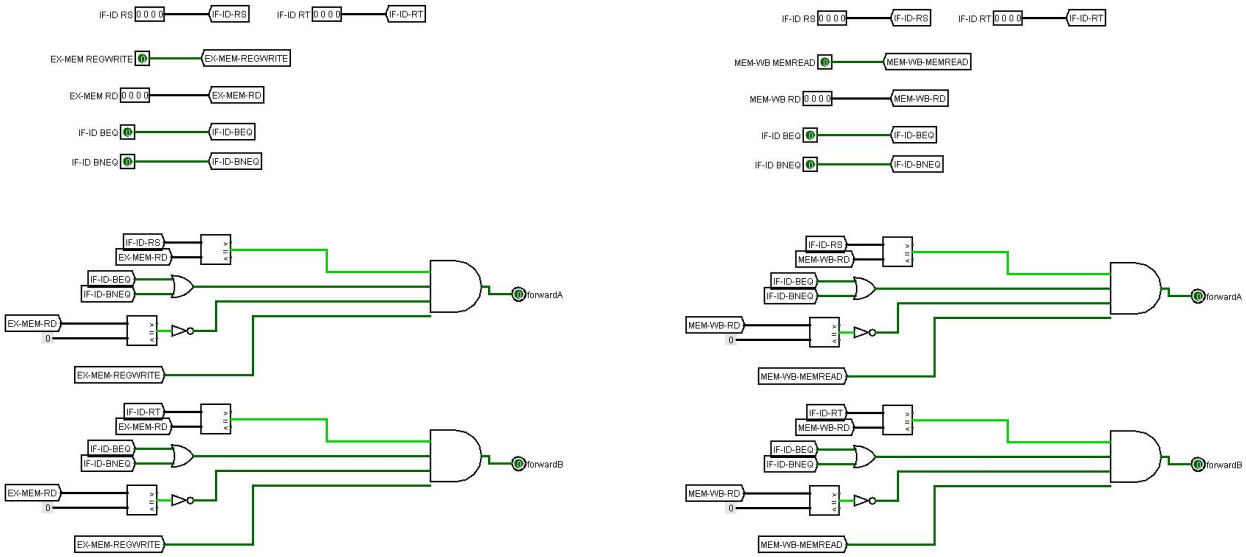




Figure 8: Load-Store Forwarding

### 5.5.4 Branch Forwarding

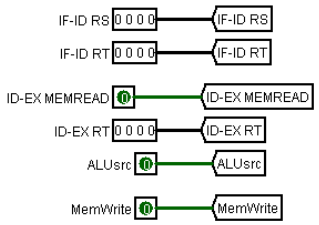



(a) Use-Branch Forwarding

(b) Load-Branch Forwarding

Figure 9: Branch Forwarding

### 5.5.5 Hazard Detection



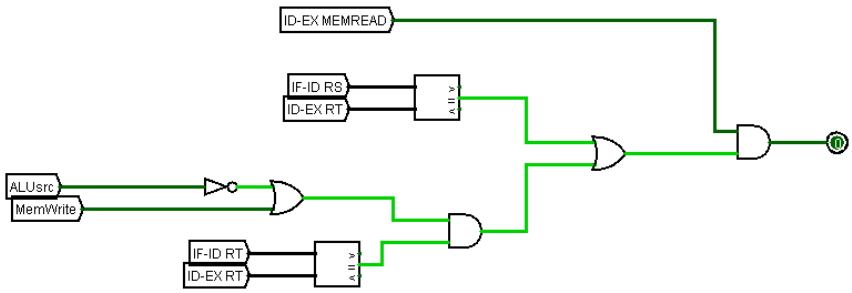
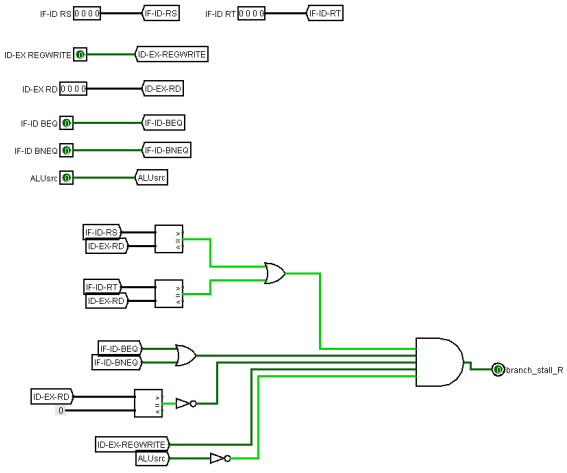
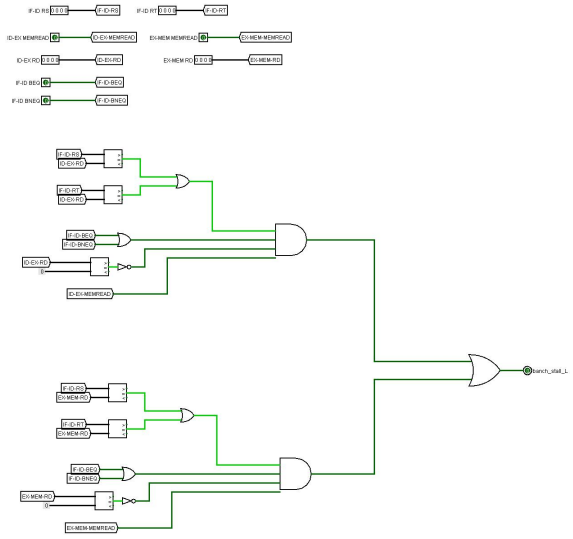




Figure 10: Hazard Detection Unit




(a) Use-Branch Hazard



(b) Load-Branch Hazard

Figure 11: Branch Hazards

## 5.6 Complete Circuit

The circuit consists of key components that work in harmony to execute instructions. The instruction memory fetches and decodes instructions, passing them to the register file and execution unit. The execution unit houses the ALU, which performs arithmetic and logical operations. The memory system handles both stack and data memory, distinguishing them using the `sp` flag. These components are interconnected to ensure the efficient processing of instructions across the system.

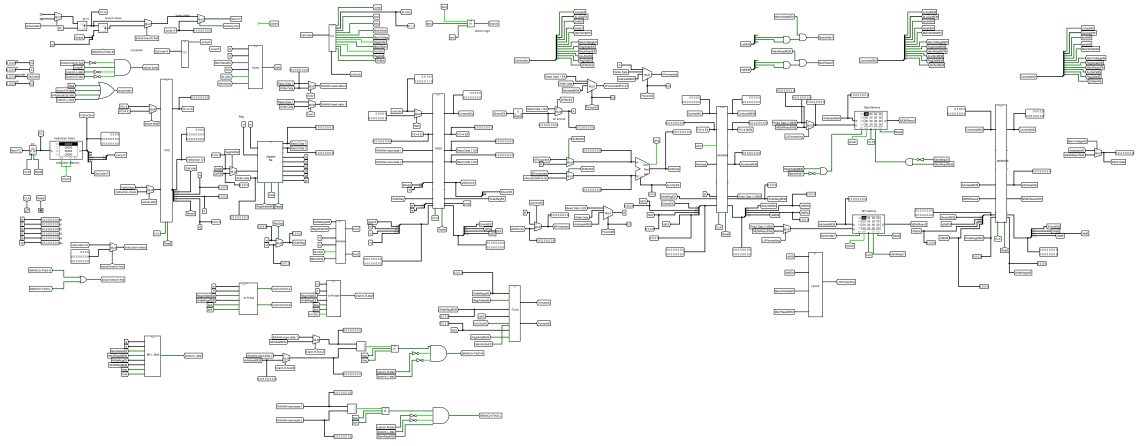


Figure 12: Complete Circuit Diagram

# 6 How to write and execute a program in your machine

A standard MIPS instruction code could be in this format:

```
start:
addi $t1, $zero, 3
subi $t2, $zero, -2
add $t0, $t1, $t2
sub $t3, $t1, $t2
nor $t4, $t0, $t2
sw $t1, 3($t2)
srl $t2, $t2, 1
beq $t2, $t3, label1
j end
label1:
sll $t3, $t3, 1
lw $t2, 4($sp)
j label2
label3:
or $t0, $t0, $t2
andi $t2, $t4, 1
ori $t1, $t1, 5
and $t1, $t2, $t4
j end
label2:
bneq $t0, $t2, label3
end:
```

The `converter.py` program was written to convert a MIPS instruction code to machine code, which can be used in the instruction memory in the MIPS circuit simulation.

# 7 ICs used with their count

| Serial Number | Component Name | Approximate IC/Component Count |
|---|---|---|
| 1 | Register(8-bit) | 34 |
| 2 | RAM | 4 |
| 3 | ROM | 1 |
| 4 | Decoder | 1 |
| 5 | Adder | 2 |
| 6 | MUX | 25 |
| 7 | AND | 42 |
| 8 | NOT | 28 |
| 9 | OR | 20 |
| 10 | COMPARATOR | 34 |
| 11 | TOTAL | 191 |

Table 4: Approximate IC/Component Count

# 8 The Simulator Used along with the Version Number

logisim-generic-2.7.1

# 9 Discussion

The implementation of the circuit was a challenging process, requiring multiple iterations and careful consideration of various design aspects. We faced several hurdles, each of which necessitated thoughtful problem-solving to ensure the success of the project. Below are the key challenges we encountered:

1. **Microprogramming the Control Unit:** The control unit was the cornerstone of the entire design. Initially, the microprogramming approach required frequent revisions as we refined our simulation methodology. Achieving the right balance of control sequences and optimization was crucial for the operation of the processor.

2. **Handling the Shift Instructions:** The conventional implementation of the `sll` and `srl` instructions involves setting the Rs register to 0, using Rt as the source. However, this would have required multiple multiplexers (muxes) in our design. To simplify, we directly used the Rs register as the source and set Rt to 0, ensuring proper shifting with a more efficient approach.

3. **ALU Shifting Issue:** Our ALU required special attention as it performed shifts in the reverse order, such as `B<<A` instead of the typical `A<<B` for other operations. To address this, we incorporated multiplexers within the ALU to properly switch the operands and execute the shift operation as `A<<B`.

4. **Program Counter (PC) Handling:** Unlike conventional MIPS systems, where the PC is incremented by 0x4 to fetch the next instruction, we incremented the PC by 0x1 for each instruction. This adjustment was necessary because our ROM was not byte-addressed but instruction-addressed, which allowed for direct sequencing of instructions with a single-byte increment.

5. **Jump Instruction Optimization:** Unlike standard designs, our jump instruction did not require shifting since the instruction format provided the necessary 8-bit address. This simplified the design and eliminated the need for additional shift operations during jump execution.

6. **Separate Memory for Stack and Data:** To avoid overwriting important information and ensure proper allocation, we implemented two separate memory modules: one for stack memory and another for data memory. The stack memory follows the standard rule of storing data starting from the highest address, while the data memory begins at address 0.

7. **Challanges in Pipelined Design:**

   (a) **Hazard Management:**

   - **Data Hazards:** Occur when an instruction depends on the result of a previous instruction still in the pipeline.
   - **Control Hazards:** Occur during branching and jumping, causing the pipeline to fetch incorrect instructions.

(b) **Pipeline Stalling:** Stalls occur when the processor must wait for dependencies to resolve, reducing performance.

(c) **Resource Conflicts:** Limited resources like memory and the register file can lead to contention, especially in multi-stage pipelines.

(d) **Synchronization Issues:** Ensuring all pipeline stages work seamlessly without mismatches or delays.

8. **Steps to Solve Issues:**

(a) **Forwarding Unit for Data Hazards:** Introduced to bypass data from later stages (EX or MEM) to earlier stages (ID or EX), ensuring the pipeline doesn't stall unnecessarily.

(b) **Hazard Detection Unit for Stalling:**

- Automatically detects dependencies and stalls the pipeline only when forwarding is insufficient.
- Prevents incorrect execution by pausing affected stages.

(c) **Control Hazard Handling:**

- Implemented a flush mechanism to clear incorrect instructions in the pipeline when branch predictions fail.
- Ensured minimal performance impact through accurate branch handling strategies.

(d) **Efficient Resource Allocation:** Designed separate memory blocks for instruction and data to prevent resource conflicts between fetch and access stages.

(e) **Pipeline Balancing:** Optimized stage lengths and ensured uniform workloads to minimize delays.

This project highlighted the challenges of designing a pipelined architecture and demonstrated effective solutions to handle hazards and synchronization issues. By integrating hazard detection, forwarding, and resource allocation techniques, the MIPS 8-bit processor achieved seamless instruction execution with improved throughput.

# 10 Contributions of Each Member

The development of the MIPS 8-bit processor with pipelining was a collaborative effort, where each team member contributed equally to different aspects of the project. Below is a summary of each member's contributions:

## 10.1 2105002: Khalid Hasan Tuhin

- Designed and implemented the instruction fetch (IF) and instruction decode (ID) stages of the pipeline.
- Worked on the hazard detection unit to resolve data hazards and ensure pipeline consistency.
- Contributed to the testing and debugging of the overall pipeline functionality.
- Report Writing - 1,5,6

## 10.2 2105014: K.M. Mehemud Azad

- Focused on the execute (EX) and memory access (MEM) stages of the pipeline.
- Implemented the forwarding unit to resolve data dependencies and minimize stalls.
- Assisted in optimizing control hazard handling through efficient branch prediction mechanisms.
- Report Writing - 2,7,8

## 10.3 2105015: Arnob Biswaws

- Developed the write-back (WB) stage and ensured seamless integration with earlier pipeline stages.
- Worked on resource allocation and conflict resolution between instruction and data memory.
- Conducted performance evaluations and refined the pipeline to achieve balanced throughput.
- Report Writing - 3,4,5

Each member actively participated in documentation, presentation preparation, and finalizing the overall design of the MIPS processor, ensuring a well-coordinated and successful project.