CSE **210**
Computer Architecture Sessional

Project: MIPS Design & Simulation

Section - B1
Group - 05

Members of the Group:

i 2105068 - Anika Morshed

ii 2105069 - Sheikh Iftikharun Nisa

iii 2105081 - Diganta Saha Tirtha

Report Prepared By:
Sheikh Iftikharun Nisa (2105069)

1 Introduction

MIPS is a RISC (Reduced Instruction Set Computer) ISA (Instruction Set Architecture). Instructions of MIPS are fixed, thus ensuring regularity.

Here is an example of an `add` instruction.

Operation	Instruction	Action
Addition	`add $t2, $t1, $t3`	`$t2 = $t1 + $t3`

Here, $t1, $t2, $t3 are registers that hold values. To evaluate an expression $x = a+b-c$, we would do the following.

```
add $t0, $t1, $t2      [x = a + b]
add $t0, $t0, $t3      [x = x + c  or  x = a + b + c]
```

According to MIPS instruction rules, arithmetic operations can only take registers as arguments, size of a register is 32 bits and there are 32 registers in total.
A datapath is built with registers, ALUs, MUXs, Memory and Control elements that can process data and addresses in the CPU. MIPS instructions are fed through a datapath to perform various instructions such as addition, logical operations (OR, AND etc.) load/store word, branching, or jump.

This report details the design and simulation of an 8-bit MIPS processor using Logisim. Key components include an Instruction Memory, 8-bit Arithmetic Logic Unit (ALU), a Register File, a Data Memory and a micro-programmed Control Unit. The ALU performs essential arithmetic and logic operations, while the Instruction and Data Memory manage program instructions and data, respectively. The Control Unit, driven by a micro- programmed ROM, regulates instruction execution through predefined control words. The processor's 20-bit instruction format supports R-type, I-type, and J-type instructions.

For pipelining and handling hazards, we implemented a Program Count Register, a Forwarding Unit and a Hazard Detection Unit.

2 Instruction Set

For this project, we have been tasked with implementing a modified and reduced version of the MIPS instruction set. Our implementation will feature an 8-bit address bus and a 8-bit data bus, as well as a 8-bit ALU (provided in the Project specifications), hence the name 8-bit MIPS.

As part of our design, we need to include several 8-bit temporary registers, including `$zero, $t0, $t1, $t2, $t3,` and `$t4`. A `$sp` register was added in order to keep track of the stack pointer in the stack memory.

The Instruction Set for our MIPS is given below.

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

Table 1: Instruction Set Description

2.1 MIPS Instruction Format

Our MIPS instruction would be 20 bits long following these three formats.

- **R-type**

Opcode	Src Reg 1	Src Reg 2	Dst Reg	Shft Amnt
4-bits	4-bits	4-bits	4-bits	4-bits

Table 2: R-type Instruction Format

- **I-type**

Opcode	Src Reg 1	Dst Reg	Address / Immediate
4-bits	4-bits	4-bits	8-bits

Table 3: I-type Instruction Format

- **J-type**

Opcode	Target Jump Address	0	0
4-bits	8-bits	4-bits	4-bits

Table 4: J-type Instruction Format

Opcodes of the instructions are 4 bits, so between 0 and 15. We are given the instruction assignment **LEFPKGIMDAHJNCBO**. So, our instruction set would be:

Opcode	Instruction Type	Instruction
0000	Memory	sw
0001	Logic	and
0010	Logic	andi
0011	Control	j
0100	Logic	nor
0101	Logic	or
0110	Logic	sll
0111	Memory	lw
1000	Arithmetic	subi
1001	Arithmetic	add
1010	Control	ori
1011	Logic	srl
1100	Control	beq
1101	Arithmetic	sub
1110	Arithmetic	addi
1111	Logic	bneq

Table 5: Instruction Set

3 MIPS Processor Block Diagram

Here is a figure showing the block diagram of a 32-bit mips.

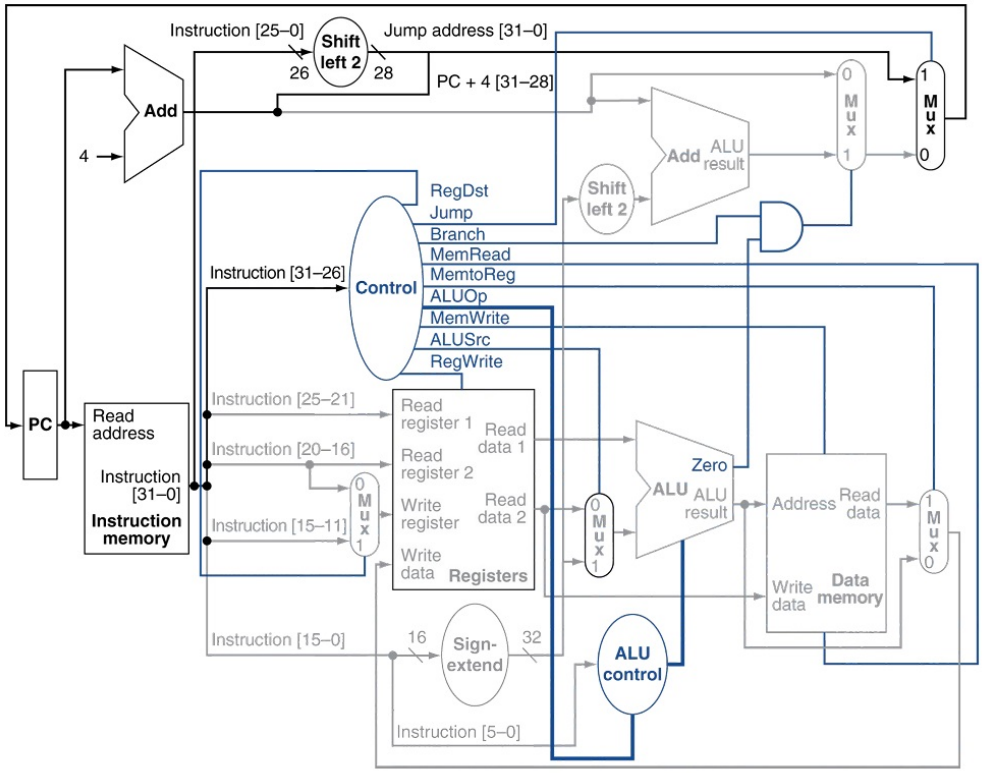

Figure 1: 32-bit PC Block Diagram

4 Complete Circuit Diagram

The complete circuit diagram of our mips is shown below.

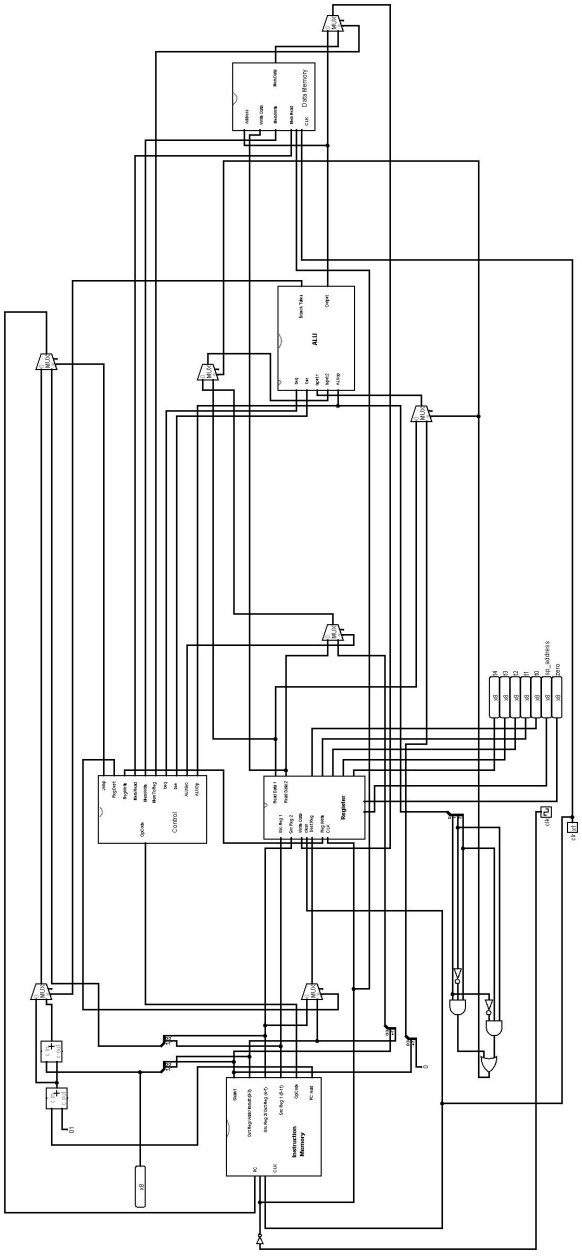

Figure 2: 8-bit MIPS

We shall describe the design steps by listing and detailing each of the major individual circuit components.

4.1 Instruction Memory

Using our MIPS to assembly code assembler, we generated the hex image of our instruction set. We then loaded this hex image into the built-in ROM of `Logisim`.

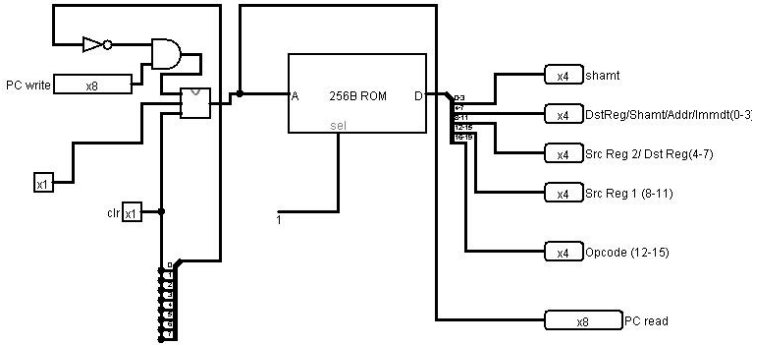

Figure 3: Instruction Memory

4.2 Arithmetic and Logical Unit (ALU)

Our next task is to design an 8-bit ALU to handle arithmetic and logical instructions. The ALU takes two 8-bit input values (Data1 and Data2), an ALUOP value, along with the beq and bneq flag, and produces an 8-bit output value, which is then used for further calculation in other stages along with a flag that indicates whether a branch is supposed to be takena or not. It was made using the 8-bit basic ALU provided in the project. This ALU also generates a zero flag that verifies whether the output is zero or not.

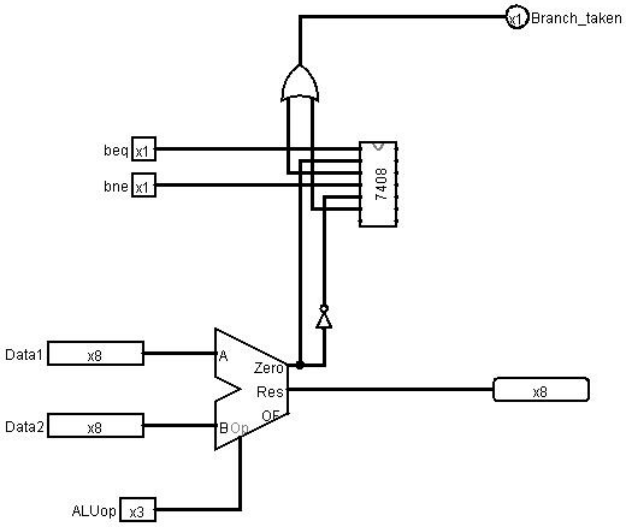

Figure 4: ALU encapsulated

4.3 Control Unit

In the control unit, we set the values for our MUXs and branch operations based on the provided instruction set for each group. To do this, we use a 16-bit ROM, where we store the hexadecimal values for the different operations. When we receive the 4-bit OpCode, we use it to select the corresponding operation from the ROM. The ROM outputs 9 selector bits for the operation, as well as a 3-bit ALUop that is used to control the operation of the 8-bit ALU.

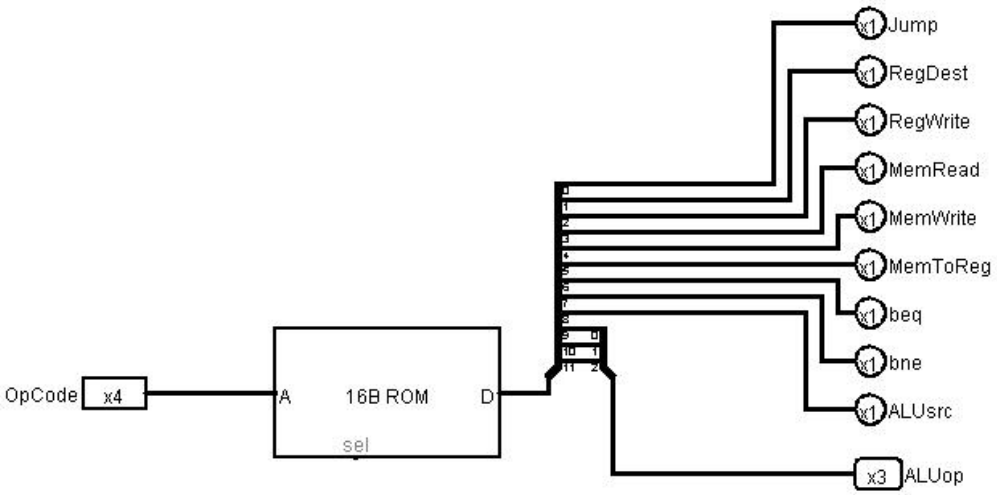

Figure 5: Control Unit

4.4 Register File

Registers are simulated using D-flipflops (IC 7474) in our mips. Figure 6 is an instance of how a single 8-bit register looks like in our MIPS.

To implement our register file, we first created a file consisting of 7 registers. The base registers, ranging from $t0 - $t4 and $zero registers, were implemented using our 8-bit register (Figure 6) from which data could be read or written as per necessity. Along with that, an $sp register was implemented to store the current address of the stack pointer.

In order to read from or write to a register, the register file, shown in Figure 7 takes the two register address (ReadReg1 and ReadReg2) as inputs. We also need to provide a write data value and the corresponding write register address as input when writing to a register. However, if we are executing a Store Word (`sw`) instruction, we do not need to write in the register file. Therefore, we take a selection bit, RegWrite, as an input to determine whether we need to write the value or not.

In the register file, the outputs represent the values stored in the corresponding registers at a given instant, and the `$sp` holds the address of the stack pointer in memory.

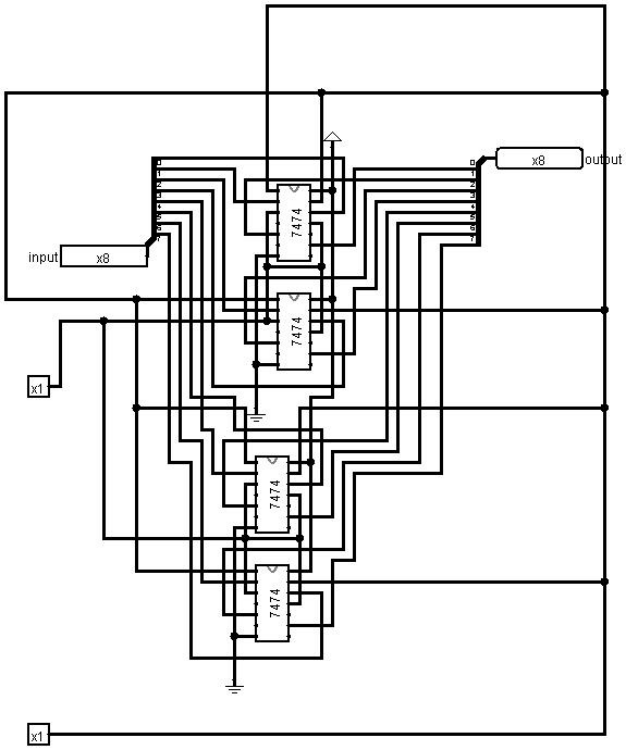

Figure 6: 8 bit Register

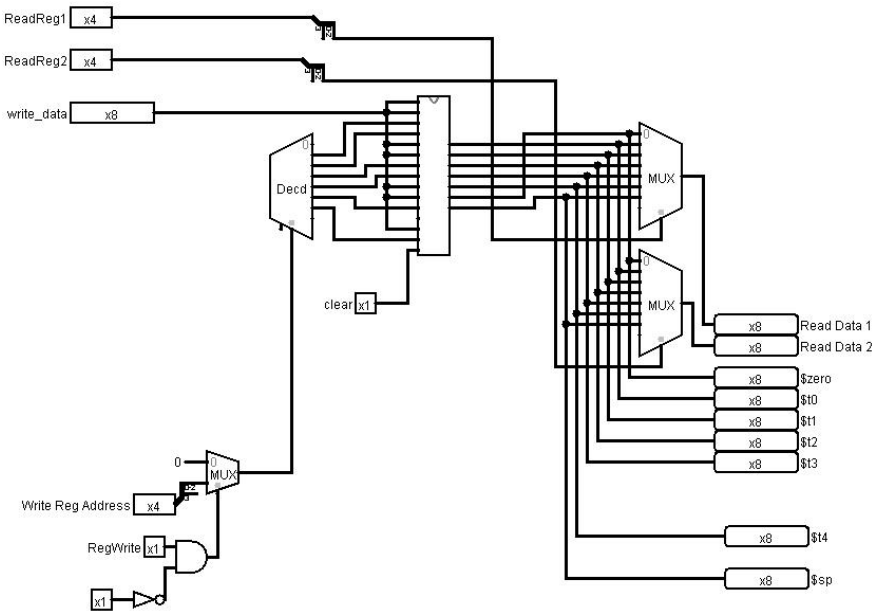

Figure 7: Register File

4.5 Data Memory

Data Memory serves as the storage for memory operations. It takes an 8-bit memory address for reading and writing, and the data written to the 256B RAM is also 8 bits. Memory read operations, such as `lw`, and memory write operations, such as `sw`, are controlled by the MemRead and MemWrite flags from the control unit, respectively. Based on these flags, we

select the operation—either to read or to write. For memory read (`lw`), the output is the 8-bit memory data when the MemRead flag is set to 1.

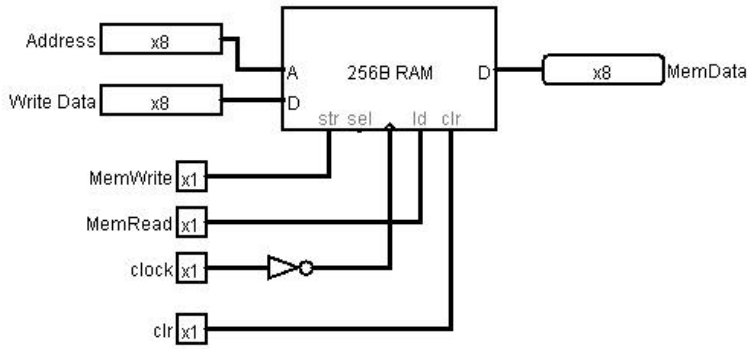

Figure 8: Data Memory

5 MIPS with Pipelining

Pipelining in MIPS (and in processors generally) is implemented to improve the instruction throughput and overall performance of the processor.

The MIPS pipeline consists of five stages:

1. IF: Instruction fetch from memory
2. ID: Instruction decode & register read
3. EX: Execute operation or calculate address
4. MEM: Access memory operand
5. WB: Write result back to register

Pipelining introduces challenges, such as data hazards (dependencies between instructions), control hazards (branch instructions), and structural hazards (resource conflicts). However, modern MIPS processors handle these challenges efficiently using techniques like data forwarding (bypassing), branch prediction, and pipeline stalls to ensure smooth operation and minimize performance loss.

The complete circuit diagram of our mips with added pipelining is shown below.

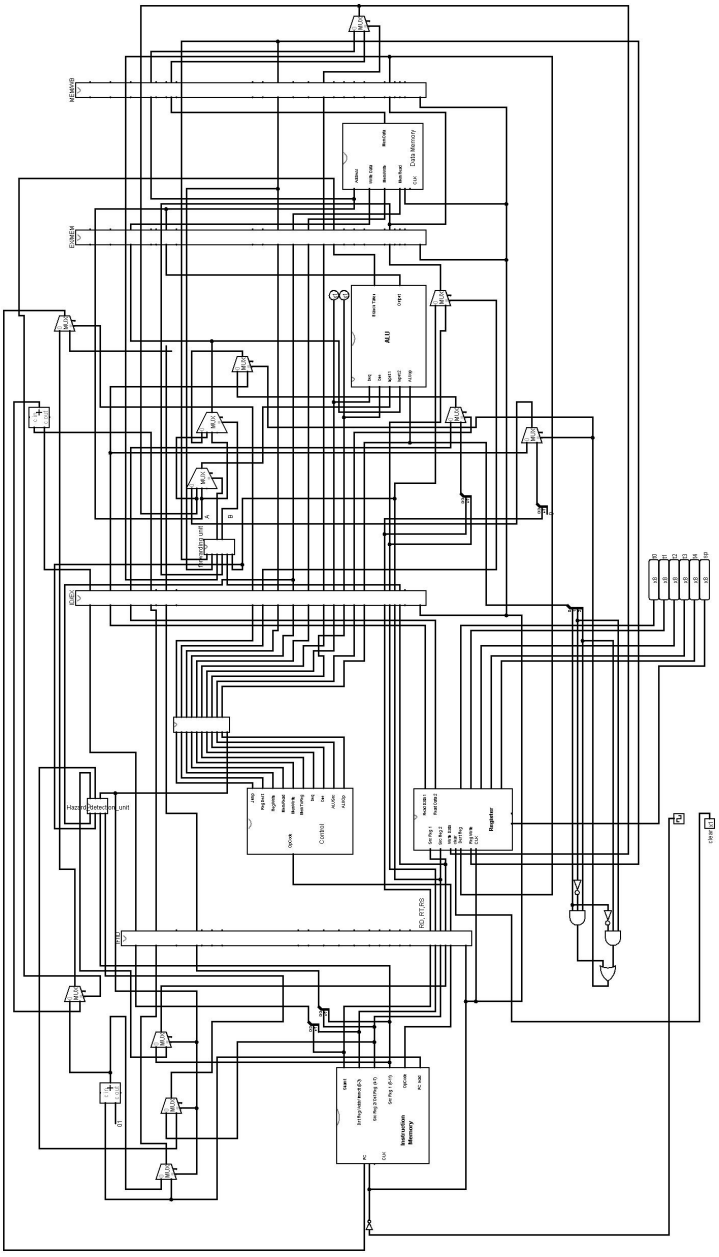

Figure 9: MIPS with pipelining

5.1 Program Count Register

This is a simple register file containing seven 8-bit registers, nine single-bit registers, one 3-bit register, five 4-bit registers.

To store the value of our PC at any given time, we use a PC register. This register consists of D-flip flops operating at falling edge trigger, which are used to store the PC value. In every clock cycle, the PC register outputs the stored value of the PC, while the current PC instruction is being executed. At the same time, the PC instruction sends the next value for the PC as an input for the PC register. The next value can either be PC+1 or PC+jump amount, depending on the specific instruction being executed.

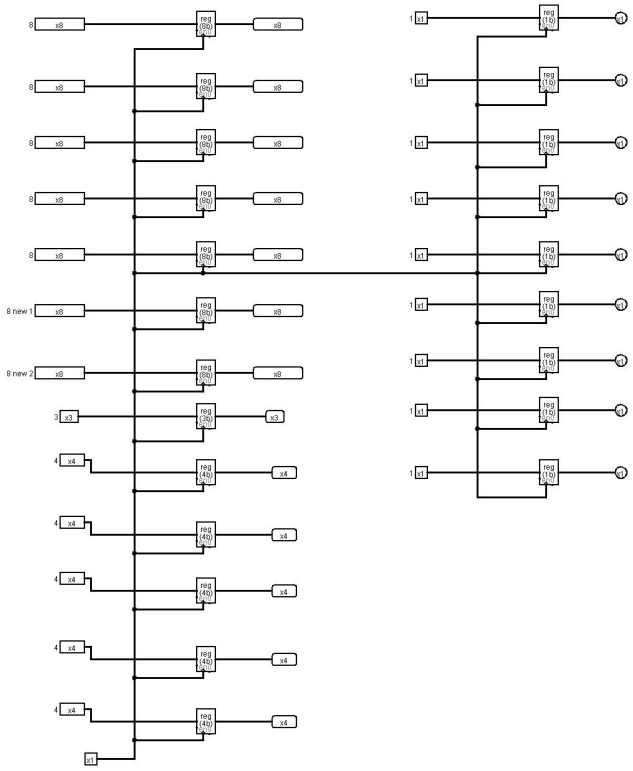

Figure 10: Program Count Register

5.2 Forwarding

In our MIPS, data hazards, including EX/MEM and double data hazards, are managed through forwarding, which allows data to be passed directly from one pipeline stage to another without waiting for it to be written back to the register file. Forwarding helps reduce pipeline stalls by ensuring that operands are available to instructions that depend on them, even if they are still in earlier stages of execution.

In Figure 11, the ForwardA (FwdA) and ForwardB (FwdB) are set depending on the Rs, Rt, Rd values at different stages (ID/EX, EX/MEM and MEM/WB).

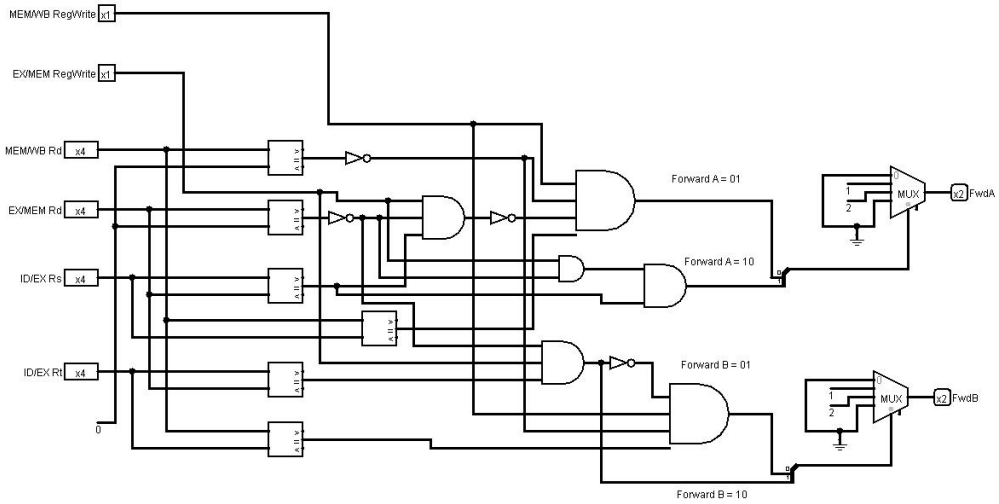

Figure 11: Forwarding Unit

5.3 Hazard Detection Unit

We begin by implementing a Control 0 module, as shown in Figure 12, which sets all the control bits to zero, effectively simulating a no-op. This is crucial for handling cases such as the Load-Use Data Hazard, which is determined by checking the MemRead signal along with the Rs and Rt values from different stages in our Hazard Detection Unit.

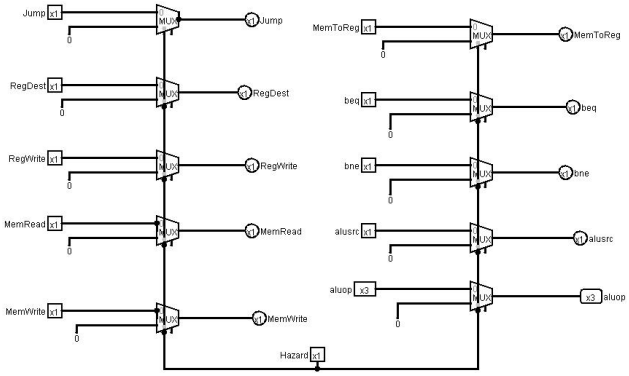

Figure 12: Control 0 Module

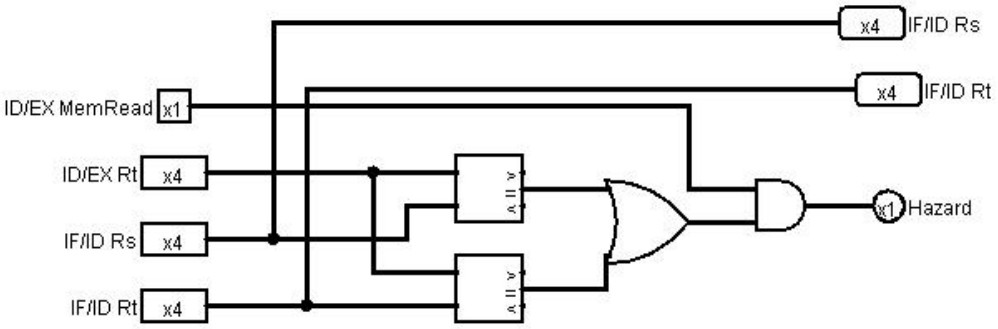

Figure 13: Hazard Detection Unit

5.4 Simulator

Software: Logisim
Version: logisim-generic-2.7.1

6 How to Write and Execute a Program

1. **Writing a Program:** All MIPS assembly instructions are to be written in `mips_code.txt` in the B1_05_Necessary Content directory. Refer to tables 5 and 2, 3 and 4 for opcodes and formats, and table 6 for register addresses.

2. **Converting to Machine Code:** Run the `mips_to_assembly.py` script to convert MIPS instructions in the `mips_code.txt` file into hexadecimal code outputted in `instruction_image.txt`.

3. **Execution:**

- Reset all components using the clear button.
- Load `instruction_image.txt` into the ROM unit in the Instruction Memory component in Logisim.
- Simulate the processor in Logisim by stepping through the instructions using the clock. A single complete clock is required to initialize the stack pointer (`$sp`) to the correct place. Afterwards each complete clock will execute a full instruction in out basic MIPS (without pipelining).

Register	**Address Bits**
$zero	0000
$t0	0001
$t1	0010
$t2	0011
$t3	0100
$t4	0101
$sp	0110

Table 6: Register Address Table

7 ICs used with their count

The IC count for the version:

IC number	Quantity used
7404	1
7408	2
7432	1
7474	32
74138	1
74153	16
74157	52
Total:	105

Table 7: ICs used with count

8 Contribution of Each Member

2105068: ALU encapsulated, Instruction Memory, Control Unit, Forwarding and Hazard Detection Unit Designing, Assembling Circuit, Pipelining, Debugging and Testing circuit, Testing assembly code.
2105069: ALU encapsulated, Instruction, Data and Stack Memory, Register file and Control Unit Designing, MIPS to Assembly Code designing, Assembling Circuit, Pipelining and Testing Circuit.
2105081: ALU encapsulated, Instruction Memory, Control unit, Forwarding and Hazard

Detection Unit Designing, Assembling Circuit, Pipelining, Debugging and Testing circuit, Testing assembly code.

9 Discussion

1. For implementing the circuit on software level, we used 7400-library integrated circuits.
2. The circuit was modularized into distinct stages of instruction execution, which facilitated debugging and testing while also reducing complexity. The tasks were distributed among team members as outlined in the previous section.
3. An 8-bit ALU was provided in the project which was used to build a more refined ALU encapsulated with flags like Branch_taken.
4. The value of each data memory was kept visible at the software level.
5. 6 registers ($t0, $t1, $t2, $t3, $t4, $zero) were implemented from scratch using the 7474 IC, from which data could be read or written as per necessity. Along with that, an $sp register was implemented to store the current address of the stack pointer. The values of the registers were kept visible at all times.
6. The outputs were checked multiple times for a large number of input test cases to ensure that our circuit satisfies the expected output values from the given input file.
7. In our MIPS we read the instructions byte-by-byte, making the size of a word only a single byte, thus requiring us only to add 1 with our current PC address to go the next instruction, unlike the standard 32-bit MIPS which reads 4 bytes at a time, as 1 word is equivalent to 4 bytes in that case.
8. We performed PC+1 using the C_{in} value of the adder instead of using a separate adder.
9. Initially, single cycle MIPS was designed and tested. Later on, Pipelining was added in order to increase the efficiency at each clock cycle.
10. The pipelined version of the MIPS extends the non-pipelined version by incorporating hazard detection units, pipeline registers, and making minimal adjustments (like adding forwarding) to the existing core components. It effectively handles EX/MEM, double data and load-use hazards.
11. The assembler program, written in Python, was submitted along with the processors to convert MIPS code into assembly code. It includes support for comments and handles basic operations.