Bangladesh University of Engineering and Technology

MIPS Design and Simulation

CSE-210 Computer Architechture Sessional

2105123-Shatabdi Dutta Chowdhury

2105137-Arpita Dhar

2105141-Mehedi

Date of Submission: December 7, 2024

1 Introduction

MIPS (Microprocessor without Interlocked Pipeline Stages) is a Reduced Instruction Set Computing (RISC) architecture widely used in the design of microprocessors and embedded systems due to its simplicity, efficiency, and adaptability.

This report presents the architecture and setup of an 8-bit MIPS computer with pipelining. Our design includes five major components: instruction memory, control unit, register file, ALU, data memory, and stack memory.

The 8-bit computer features an 8-bit data bus, an 8-bit address bus, 8-bit temporary registers, and an 8-bit Arithmetic and Logic Unit (ALU). To adhere to standard design principles, we have incorporated separate instruction and data memory in our system design.

In this assignment, we have designed an 8-bit processor implementing the MIPS ISA, where each instruction executes in a single clock cycle. The processor includes key components such as instruction memory, data memory, stack memory, register file, ALU, and a control unit.

The processor is composed of the following six components:

Program Counter (PC):
The program counter is an 8-bit register that activates on the falling edge of the clock signal. After each clock cycle, it increments its stored value by 1. This value is used as the address for the instruction memory.

Register File:
The register file consists of seven registers, denoted as $zero, $t0, $t1, $t2, $t3, $t4, $t5, and $sp. The $zero register is hardwired to store 00H, while the remaining registers function as general-purpose registers.

ALU:
The Arithmetic and Logic Unit (ALU) performs calculations using two 8-bit binary inputs. It is controlled by a 3-bit ALUop code, which determines the type of operation to execute.

Control Unit:
The control unit decodes instructions and provides selection inputs to all multiplexers (MUXs), the register file, data memory, stack memory, and the ALU, ensuring coordinated execution of instructions.

Data Memory:
The data memory functions as the main memory with a storage capacity of 256 bytes, storing data as 8-bit values. It is used for storing variables and other general-purpose data required by the processor.

Stack Memory:
The stack memory is a specialized section of memory used for temporary data storage during subroutine calls and returns. It is managed by the $sp (stack pointer) register, which keeps track of the top of the stack. This allows efficient storage and retrieval of function parameters, return addresses, and local variables.

This comprehensive design integrates these components to implement a fully functional 8-bit MIPS processor.

2 Instruction Set

Here is the description of the instruction set.

2.1 Instruction set with Instruction ID

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

Figure 1: Instruction Set.

2.2 Instruction Set with Opcode

Op-Code	Instruction	Category	Type
0000	bneq	Control-conditional	I
0001	and	Logic	R
0010	or	Logic	R
0011	nor	Logic	R
0100	lw	Memory	I
0101	andi	Logic	I
0110	sw	Memory	I
0111	sll	Logic	R
1000	add	Arithmetic	R
1001	sub	Arithmetic	R
1010	subi	Arithmetic	I
1011	j	Control-unconditional	J
1100	ori	Logic	I
1101	addi	Arithmetic	I
1110	srl	Logic	R
1111	beq	Control-conditional	I

Table 1: Instruction Set Table with Op-Codes

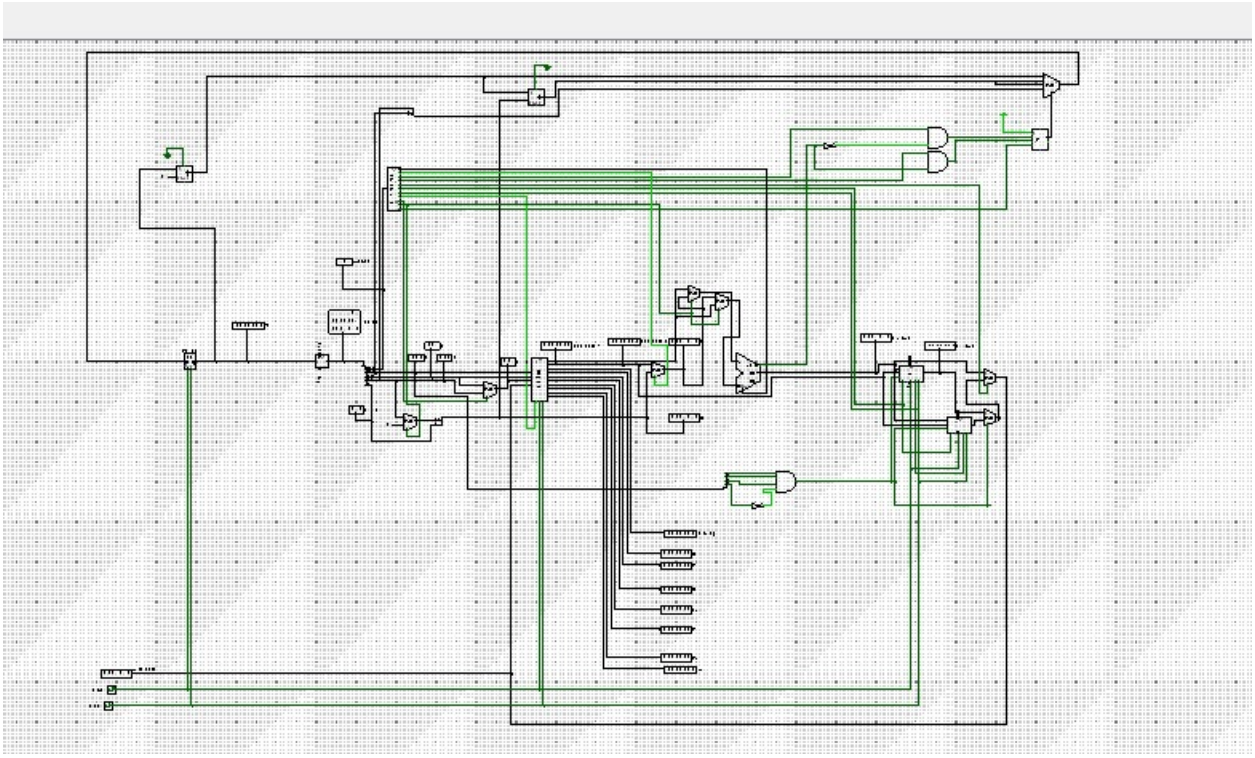
2.3 With control Signal

Instruction ID	Op-Code	Instruction	Jump	Shift	RegDst	RegWrite	MemRead	MemWrite	MemToReg	Beq	Bneq	AluSrc	AluOp
O	0000	bneq	0	0	0	0	0	0	0	0	1	0	001
E	0001	and	0	0	1	1	0	0	0	0	0	0	011
G	0010	or	0	0	1	1	0	0	0	0	0	0	010
K	0011	nor	0	0	1	1	0	0	0	0	0	0	100
M	0100	lw	0	0	0	1	1	0	1	0	0	1	000
F	0101	andi	0	0	0	1	0	0	0	0	0	1	011
L	0110	sw	0	0	0	0	0	1	0	0	0	1	000
I	0111	sll	0	1	1	1	0	0	0	0	0	1	101
A	1000	add	0	0	1	1	0	0	0	0	0	0	000
C	1001	sub	0	0	1	1	0	0	0	0	0	0	001
D	1010	subi	0	0	0	1	0	0	0	0	0	1	001
P	1011	j	1	0	0	0	0	0	0	0	0	0	000
H	1100	ori	0	0	0	1	0	0	0	0	0	1	010
B	1101	addi	0	0	0	1	0	0	0	0	0	1	000
J	1110	srl	0	1	1	1	0	0	0	0	0	1	110
N	1111	beq	0	0	0	0	0	0	0	1	0	0	001

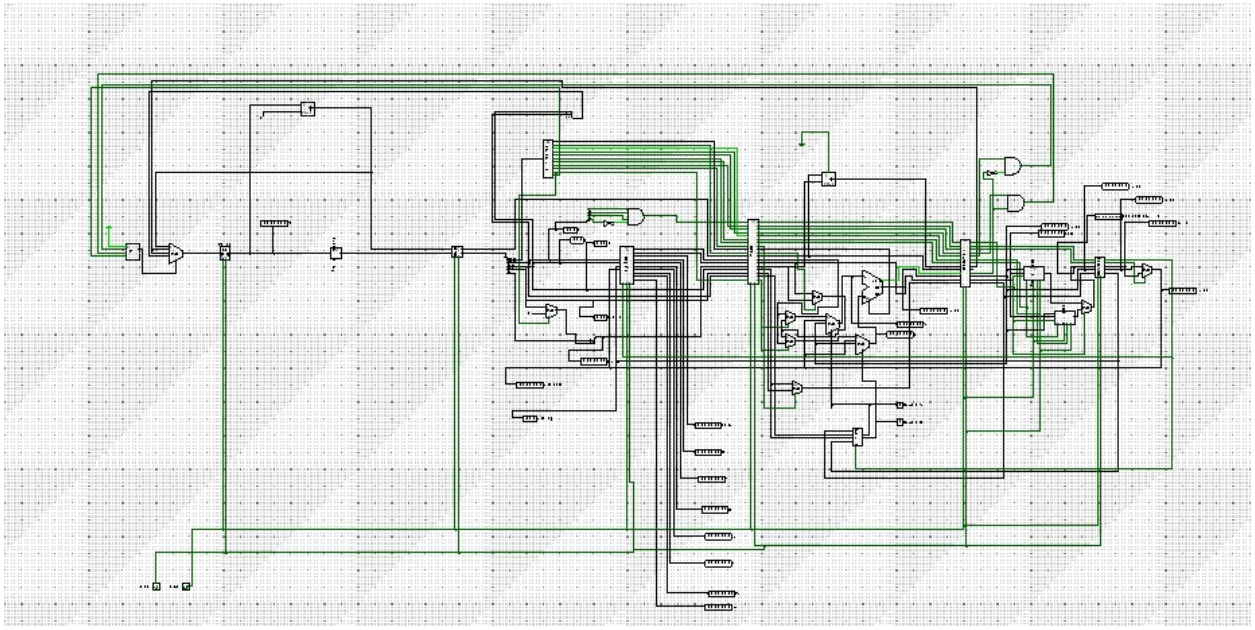
Table 2: Instruction Table with Control Signals

3 Cicuit Diagram

3.1 Circuit Without pipelining



3.2 Circuit With pipelining



4 How to write and Execute a Program in my Machine

4.1 For PC

The program counter will be increasing 1 for each clock cycle.

4.2 Intruction memory

Here we edit the content of the ROM with Hexadecimal number using the asssembly code .From heer we get 20 bits of instruction.

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

Figure 2: Mips instruction format

4.3 How to see the output

For each clock cycle we should see the corresponding output from the instruction in the register and the value will be seen in that register. But for the push and pop operation there is some difference .As they are not the direct instructions.

4.3.1 Push and Pop

For implementing this part, we introduced stack memory and sp pointer.

- `$sp` (as stack pointer register) which will be initialized with `0xFF`.

For the first instruction in this part:

Instruction	**Description**
push `$t0`	`mem[$sp] = $t0`

Equivalent MIPS Code we generated:

```
sw $t0, 0($sp)
subi $sp, $sp, 1
```

For the second instruction in this part:

Instruction	**Description**
push offset(`$t0`)	`mem[$sp] = mem[$t0 + offset]`

Equivalent MIPS Code we generated:

```
lw $t5, offset($t0)
sw $t5, 0($sp)
subi $sp, $sp, 1
```

For the third instruction in this part:

Instruction	Description
pop `$t0`	`$t0 = mem[$sp]`

Equivalent MIPS Code we generated:

```
addi $sp, $sp, 1
lw $t0, 0($sp)
```

we keep the value of the sp in the stack memory.

5 IC's used with count

IC name	IC number	Count
Register(8-bit)	Register(8-bit)	9
RAM	RAM	2
ROM	ROM	2
Decoder	74138	1
Priority Encoder	74148	1
Adder	7483	2
MUX(2X1)	74157	10
MUX(4X1)	74153	1
MUX(8X1)	74151	2
AND	7408	4
NOT	7404	1
Total		35

6 Contribution Of each memeber:

- Instruction memory - 2105137
- Instruction Fetch - 2105123
- Register File - 2105123
- Execution part - 2105123
- Data memory - 2105137
- Stack Memory - 2105123
- Full circuit merge - 2105123
- Circuit checking - 2105137
- Trying utmost -
  - Pipelined registers - 2105123
  - Forwarding unit - 2105123
  - Merging pipelined components - 2105123
  - Bug in circuit-2105123,2105137
  - Load use data hazard - 2105141
- Report
  - 2105123 -
    - Instruction set
    - How to write and execute a program in your machine.
    - Contribution of each member
    - Extra part
  - 2105137 -
    - Introduction
    - Complete Circuit Diagram
    - ICs used with their count
    - Discussion

7 Extra Part

We have also tried to implement the pipelining.There for resolving the ex-hazard,mem-hazard,double data hazard we have implemented the forwarding unit. For pipelining we have implemented the pipelined registers.

1. **EX Hazard:** The dependent instruction is in the EX stage and the prior instruction is in MEM stage.

   If (EX/MEM.RegWrite and (EX/MEM.RegisterRd $\neq 0$) and (EX/MEM.RegisterRd = ID/EX.RegisterRs))

   $$\text{ForwardA} = 10$$

   If (EX/MEM.RegWrite and (EX/MEM.RegisterRd $\neq 0$) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))

   $$\text{ForwardB} = 10$$

2. **MEM Hazard:** The dependent instruction is in the EX stage and the prior instruction is in the WB stage.

   If (MEM/WB.RegWrite and (MEM/WB.RegisterRd $\neq 0$) and (MEM/WB.RegisterRd = ID/EX.RegisterRs))

   $$\text{ForwardA} = 01$$

   If (MEM/WB.RegWrite and (MEM/WB.RegisterRd $\neq 0$) and (MEM/WB.RegisterRd = ID/EX.RegisterRt))

   $$\text{ForwardB} = 01$$

3. **Double Data Hazard:** The dependent instruction is in the EX stage and it depends on two prior instructions, one of which is in the MEM stage and the other is in the WB stage. For this new case, MEM Hazard condition is modified this way:

   If (MEM/WB.RegWrite$\wedge$(MEM/WB.RegisterRd $\neq 0$)$\wedge$(MEM/WB.RegisterRd = ID/EX.RegisterRs)

   $\wedge$not(EX/MEM.RegWrite and (EX/MEM.RegisterRd $\neq 0$) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))

   $$\text{ForwardA} = 01$$

   If (MEM/WB.RegWrite$\wedge$(MEM/WB.RegisterRd $\neq 0$)$\wedge$(MEM/WB.RegisterRd = ID/EX.RegisterRt)

   $\wedge$not(EX/MEM.RegWrite and (EX/MEM.RegisterRd $\neq 0$) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))

   $$\text{ForwardB} = 01$$

EX/MEM.reg_rd=ID/EX.reg_rs	EX/MEM.reg_rd=ID/EX.reg_rt	MEM/WB.reg_rd=ID/EX.reg_rs	MEM/WB.reg_rd=ID/EX.reg_rt	FORWARD_A	FORWARD_B
0	0	0	0	00	00
0	0	0	1	00	01
0	0	1	0	01	00
0	0	1	1	01	01
0	1	0	0	00	10
0	1	0	1	00	10
0	1	1	0	01	10
0	1	1	1	01	10
1	0	0	0	10	00
1	0	0	1	10	01
1	0	1	0	10	00
1	0	1	1	10	01
1	1	0	0	10	10
1	1	0	1	10	10
1	1	1	0	10	10
1	1	1	1	10	10

Table 3: Forwarding Conditions for Pipeline Registers

8 Discussion

The development of our MIPS architecture simulation was a journey of iteration, optimization, and learning. We began with ambitious designs that were resource-intensive, but through careful analysis and testing, we refined our approach to achieve a streamlined and efficient implementation. A significant part of this process involved thoroughly evaluating edge cases and ensuring robust performance under various scenarios.

Our solution focuses on simulating the entire MIPS datapath using primitive C code, which presented unique challenges. Emulating the intricate behavior of the MIPS architecture within a software environment required meticulous attention to detail and a deep understanding of its components and operation.

Extensive debugging and testing were integral to our success. By running diverse test cases and ensuring precise synchronization among components, we created a cohesive and reliable simulation of the MIPS architecture.

Ultimately, this project offered invaluable insights into the principles of computer architecture, particularly the inner workings of the MIPS instruction cycle. The experience of designing and refining this system has deepened our technical knowledge and reinforced the importance of thoughtful, iterative development.