Bangladesh University of Engineering and Technology

CSE 306
Computer Architecture Sessional

Offline 3: 4-bit processor implementing MIPS instruction set

Section - B1
Group - 5

2005061 - Farriha Afnan
2005072 - Abrar Rahman Abir
2005079 - Ananya Shahrin Promi
2005087 - Iffat Bin Hossain
2005089 - Wahid Al Azad Navid

February 12, 2024

1 Introduction

MIPS, which stands for Microprocessor without Interlocked Pipeline Stages, is a type of microprocessor architecture that has played a significant role in the evolution of computing systems. The MIPS architecture is known for its simplicity, efficiency, and elegance. It follows the Reduced Instruction Set Computing (RISC) philosophy, emphasizing a small set of simple and regularly structured instructions. This design choice allows for streamlined execution and improved performance, making MIPS a popular choice for various applications, including embedded systems and educational purposes.
In adherence to the MIPS instruction conventions, mathematical operations exclusively operate on register operands, with each register having a fixed size of 32 bits. The architectural framework involves constructing a datapath equipped with an ensemble of registers, ALUs, MUXs, memories, and control elements. This intricately designed datapath is responsible for processing data and addresses within the CPU. By channeling MIPS instructions through this datapath, the CPU executes a diverse array of operations, including arithmetic computations, load/store operations, branching decisions, and jump instructions.
Understanding the fundamental principles of MIPS architecture provides insights into the broader landscape of computer architecture and helps in the exploration of various processor designs.
In this report, we will delve into the design and implementation of a 4-bit MIPS processor, exploring the key components and considerations involved in creating a functional and efficient microprocessor that aligns with MIPS architecture principles.

2 Instruction Set

2.1 Problem Description

In this assignment, we designed, simulated (in Software), and implemented (in Hardware) a 4-bit processor that implements the MIPS instruction set. Here, each instruction takes 1 clock cycle to be executed. The length of the clock cycle is long enough to execute the longest instruction in the MIPS instruction set. The main components of the processor are:

- Instruction Memory
- Data Memory
- Register File
- ALU (Arithmetic Logic Unit)
- Control Unit

2.2 Design Specification

Here, The Address bus has a size of 8 bits, the Databus has a size of 4 bits. A 4-bit ALU has been used.
Also, the register file includes temporary registers $zero, $t0, $t1, $t2, $t3, $t4. Each register has a size of 4 bits.
The control unit here is micro-programmed. The control signals associated with the operations are

stored in a special memory unit as Control Words.
All clocks required in the circuit are provided from a single clock source. Each MIPS instruction is fetched and executed in a single clock cycle.

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	S	sll
J	Logic	S	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 description

Our MIPS Instructions will be 16 bits long with the following four formats:

R-Type	**Opcode**	**Src Reg-1**	**Src Reg-2**	**Dst Reg**
	4-bits	4-bits	4-bits	4-bits
S-Type	**Opcode**	**Src Reg-1**	**Dst Reg**	**Shamt**
	4-bits	4-bits	4-bits	4-bits
I-Type	**Opcode**	**Src Reg-1**	**Src Reg-2/Dst Reg**	**Address/Immediate**
	4-bits	4-bits	4-bits	4-bits
J-Type	**Opcode**	**Target Jump Address**		**0**
	4-bits	8-bits		4-bits

Table 2: MIPS Instruction Format

2.3 Memory Considerations

For a single-cycle implementation of the MIPS instruction set, we used two separate memory units for instruction and data. Instruction Memory is accessed through the Program Counter (PC) register. And the data Memory is accessed through the address.

2.4 Instruction set assignment

The opcodes of the instruction are 4 bits, hence between 0 to 15. We are given the sequence PLIADGFBKJNCEMOH, which means:

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

Table 3: Instruction set for Group-5

2.5 Extra Features

In addition to the standard MIPS instructions, we also implemented push and pop operations in our design using a Stack. To achieve this task, the stack memory is shared with the data memory in the following way. The data memory starts from the minimum address and grows to the increasing memory addresses. On the contrary, the stack memory starts from the maximum address and grows to the decreasing memory addresses. The top of our stack is held by a stack pointer ($sp). Initially, $sp holds an address of the highest address of the stack memory. The push and pop operations are used in the provided assembly code according to the following table:

Instruction	Description
push $t0	mem[$sp] = $t0
push 3($t0)	mem[$sp] = mem[$t0+3]
pop $t0	$t0 = mem[$sp]

Table 4: Stack instruction set

2.6 Simulation

To simulate our design, an assembly code is used. Before starting the simulation, we convert the given assembly code into MIPS machine code and load the machine code into the instruction memory. The conversion process is performed using a *Cpp* code.

3 Complete Block Diagram of MIPS Processor

According to the provided instructions, we have adapted our MIPS architecture to feature a 16-bit instruction format and an 8-bit program counter, deviating from the standard 32-bit MIPS configuration. This modification allows for a more compact representation of instructions and a smaller program-counter size.

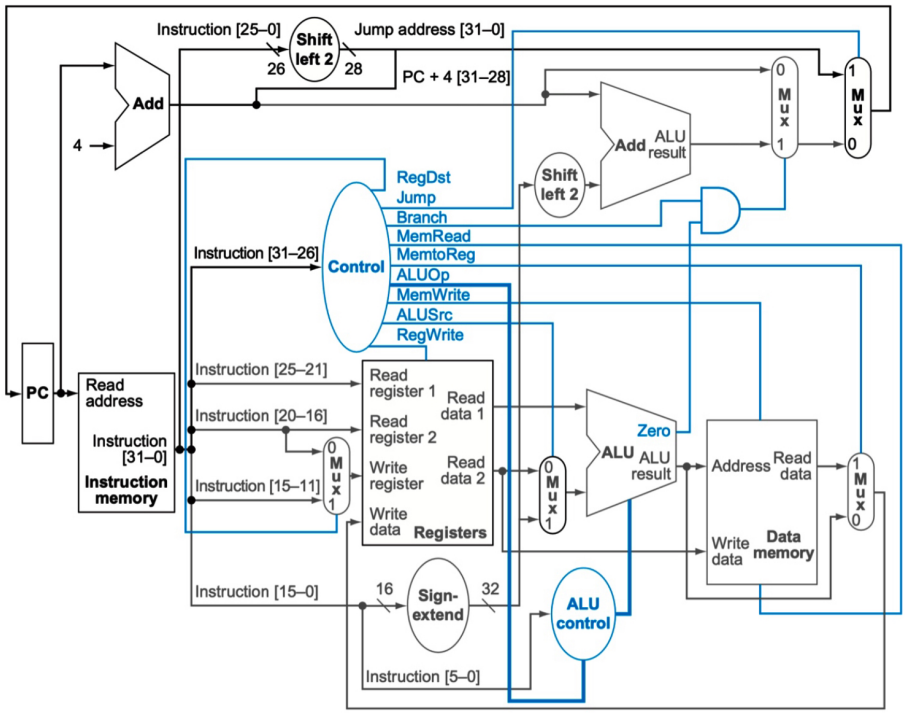


Figure 1: Complete Block Diagram of 32-bit MIPS Processor

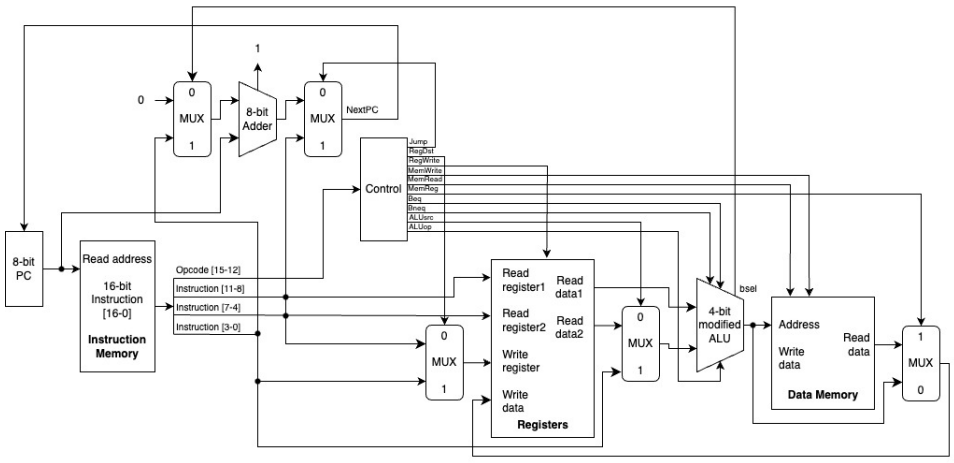


Figure 2: Complete Block Diagram of 4-bit MIPS Processor

4 Block diagrams of the main components

We will now detail each of the main parts as we describe the design steps.

4.1 Program Counter

Within our MIPS architecture, we have implemented a straightforward 8-bit Register operating on the negative edge. This register serves as the Program Counter (PC) storage unit, responsible for holding the current memory address of the instruction being executed. Comprising four D-flip flops, the PC register effectively stores the PC value.
During each clock cycle, the PC register outputs the stored PC value while the current instruction is in the execution phase. Concurrently, the ongoing instruction determines the next PC value, serving it as an input for the PC register. Depending on the specific instruction, the next value can either be PC+1 or PC+jump amount. This dynamic behavior ensures the sequential flow of program execution.
To emulate this behavior, we have utilized an Atmega microcontroller, enhancing the efficiency and reliability of our register implementation.

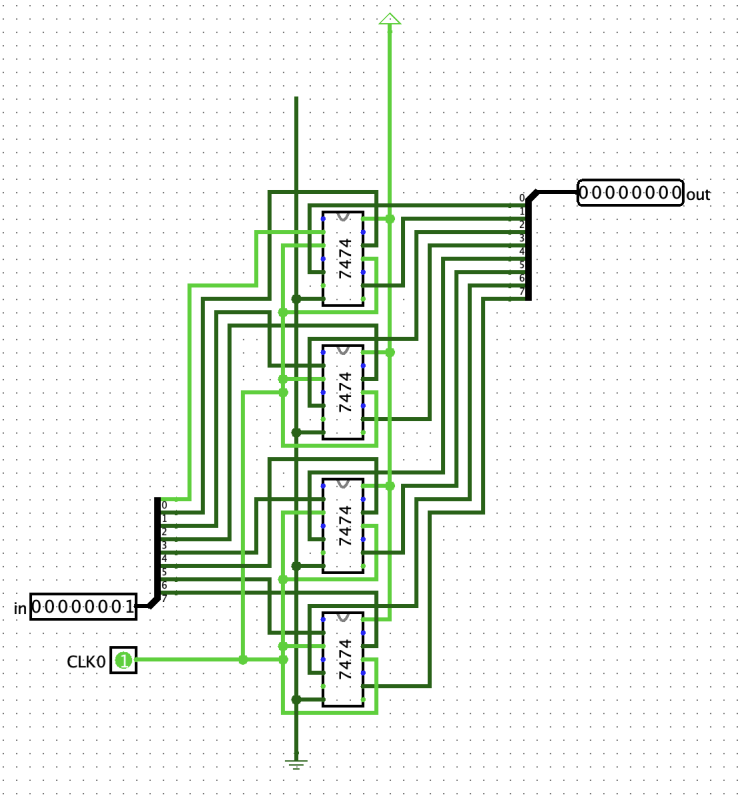


Figure 3: Program Counter

4.2 Instruction Memory

In our implementation, we harnessed the capabilities of Atmega32A arrays to store the MIPS instructions in our instruction memory. The initial set of instructions, provided as input, underwent a conversion process to generate corresponding 16-bit MIPS instruction codes. These 16-bit

instructions were further segmented into four 4-bit values to facilitate decoding: the first 4 bits denoted the destination register or immediate value, the subsequent 4 bits represented the second source register or another destination register, followed by 4 bits for the first source register, and finally, 4 bits for the Opcode indicating the instruction type.

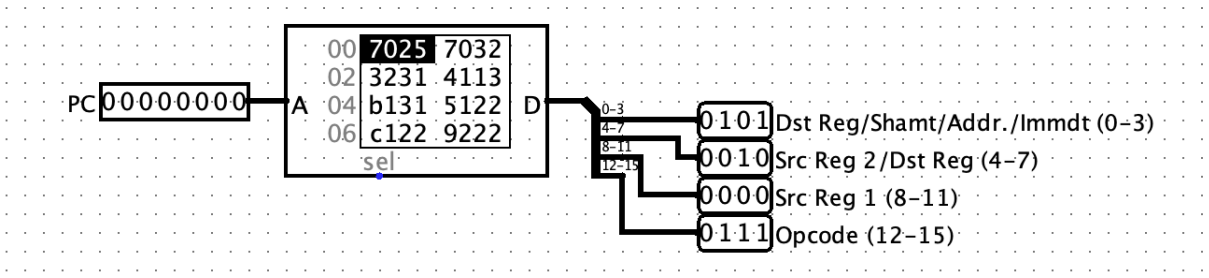


Figure 4: Instruction Memory

To facilitate the sequential execution of instructions, the PC value is incremented, prompting the instruction memory to output the next instruction in the sequence. It's important to note that instruction memory operates independently of the clock signal. To synchronize the various components and devices relying on a clock, we leveraged one of the pins on the Atmega32A to generate a clock pulse, ensuring proper coordination and timing in the system. This approach enhances the overall efficiency and synchronization of our MIPS architecture.

4.3 Control

Within our control unit, we meticulously configure the settings for our Multiplexer (MUX) and branch operations, tailoring them to the unique characteristics of each instruction group. A pivotal component in this configuration is a 16-bit Read-Only Memory (ROM) that serves as a repository for hexadecimal values corresponding to distinct operations.

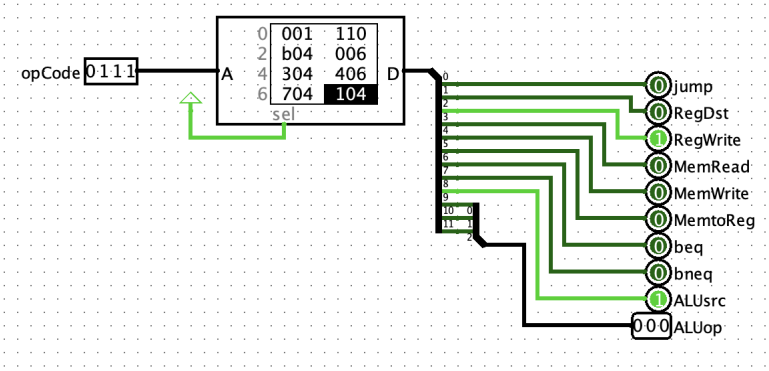


Figure 5: Control

Upon receiving the 4-bit OpCode, we employ it as a selector to pinpoint the specific operation stored in the ROM. This operation retrieval from the ROM yields not only the 9 selector bits necessary for the operation but also a crucial 3-bit ALUop. This ALUop plays a pivotal role in dictating the operation of our 4-bit Arithmetic Logic Unit (ALU). By intricately orchestrating

these components, our control unit ensures precise and efficient handling of diverse instructions, contributing to the overall effectiveness of our MIPS architecture.

Op Code	Jump	Reg Dst	Reg Write	Mem Read	Mem Write	Mem Reg	beq	bneq	ALU Src	ALU Op	Hex Val
0000	1	0	0	0	0	0	0	0	0	000	001
0001	0	0	0	0	1	0	0	0	1	000	110
0010	0	0	1	0	0	0	0	0	1	101	B04
0011	0	1	1	0	0	0	0	0	0	000	006
0100	0	0	1	0	0	0	0	0	1	001	304
0101	0	1	1	0	0	0	0	0	0	010	406
0110	0	0	1	0	0	0	0	0	1	011	704
0111	0	0	1	0	0	0	0	0	1	000	104
1000	0	1	1	0	0	0	0	0	0	100	806
1001	0	0	1	0	0	0	0	0	1	110	D04
1010	0	0	0	0	0	0	1	0	0	001	240
1011	0	1	1	0	0	0	0	0	0	001	206
1100	0	1	1	0	0	0	0	0	0	011	606
1101	0	0	1	1	0	1	0	0	1	000	12C
1110	0	0	0	0	0	0	0	1	0	001	280
1111	0	0	1	0	0	0	0	0	1	010	504

Table 5: Flag enables for every operation

4.4 Register Memory

In laying the foundation for our register file, we meticulously crafted a structure housing a total of 7 registers.

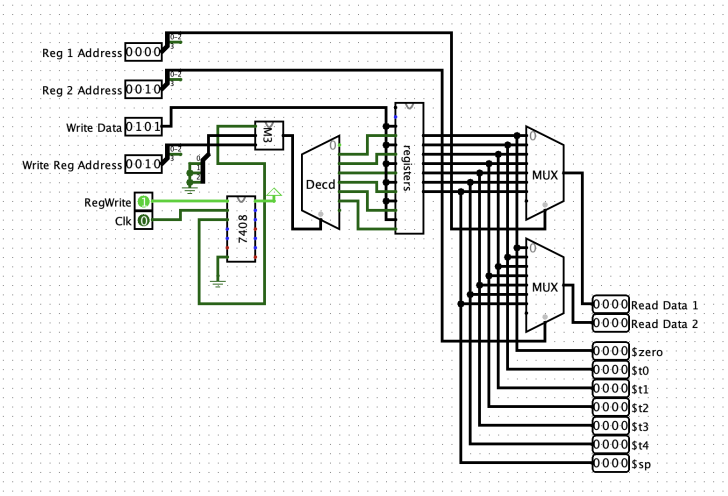


Figure 6: Caption

This ensemble includes 5 specialized temporary registers—$t0, $t1, $t2, $t3, and $t4. The repertoire extends further to encompass $sp, instrumental in managing stack-related instructions, and

$zero, a dedicated register for arithmetic operations involving a zero value.
To facilitate seamless read or write operations, the register file necessitates the provision of two register addresses as inputs. When executing a write operation, additional inputs include the data to be written and the corresponding register address. Notably, during the execution of a store word instruction, no writing operation is required in the register file. To navigate this, a selection bit termed RegWrite serves as an input, guiding the system in determining whether a value needs to be written. Moreover, we have integrated a showReg flag, allocated to a specific pin, enabling us to interrupt register file operations and display the data of a specified register, one at a time. This feature enhances the observability of register contents during the execution of our MIPS architecture.

4.5 Arithmetic Logic Unit

Embarking on the next phase of our design, we delve into the intricate architecture of a 4-bit ALU meticulously crafted to navigate the realm of arithmetic and logical instructions. The ALU, a pivotal component of our system, gracefully accepts two input values and an ALUop (ALU operation) value, orchestrating a symphony that culminates in a 4-bit output. This output, a harbinger of further computational endeavors, propels the intricate dance of data within our MIPS architecture.

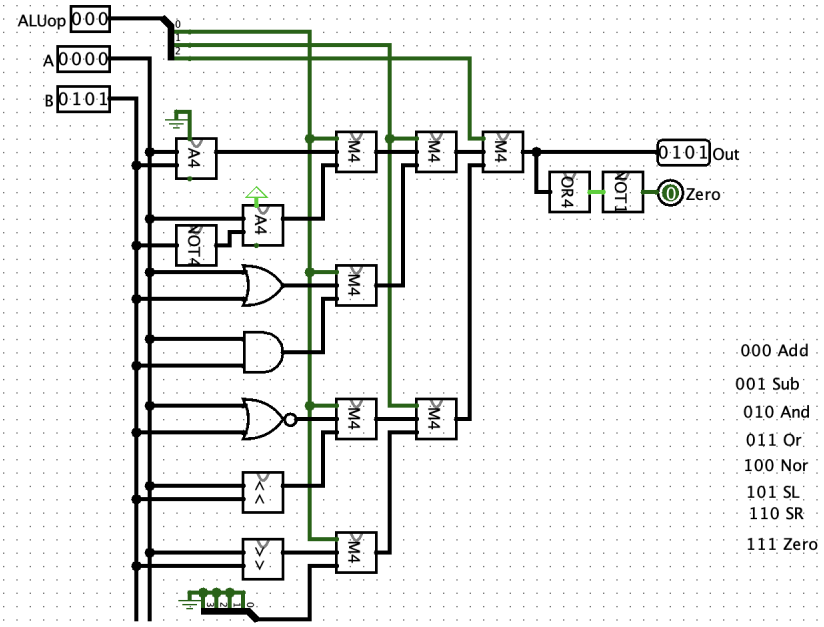


Figure 7: 4-bit ALU

Beyond its numerical prowess, our ALU extends its utility by unfurling the banner of a zero flag, a sentinel that diligently scrutinizes the output, discerning whether it gracefully aligns with the illustrious zero or not. This flag, akin to a watchful guardian, plays a crucial role in shaping the decision-making fabric of our system.
In the ballet of information flow, the circuit orchestrates a choreography where the opcode, coupled with two 4-bit input values, takes center stage. Harmoniously synchronized with this input ensemble, the ALU absorbs the essence of the operation to perform, painting an artistic tapestry of computation.

However, the stage isn't solely reserved for arithmetic and logical maneuvers. The ALU graciously opens its gates to the beq and bneq flags, and ambassadors from the control unit. These flags, bearing the imprints of branching decisions, fuse with the zero flag in a collaborative effort. From this union emerges the bsel flag, a herald signaling the grand tableau of branching necessity. It encapsulates the essence of whether our system must tread the path of a branch or gracefully proceed along its sequential journey.

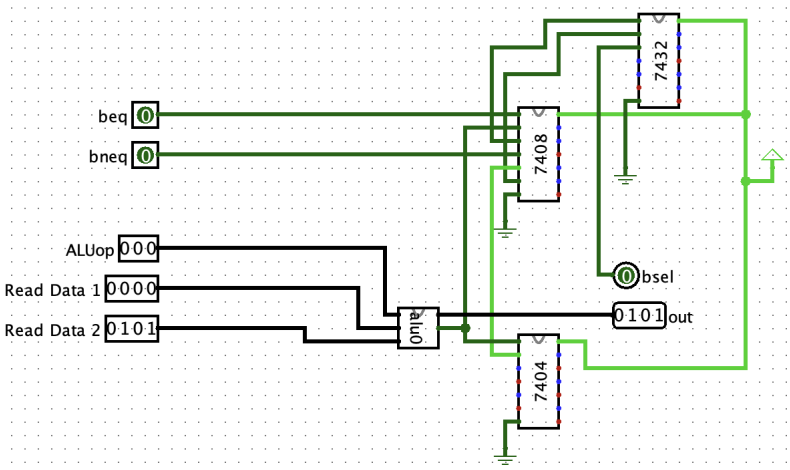


Figure 8: Modified ALU

This intricate dance between inputs, ALU operations, and branching decisions lays the groundwork for a dynamic and responsive MIPS architecture, where computational finesse converges with decision-making prowess.

4.6 Data Memory

In our MIPS system, Data Memory serves as a storage unit, using a 4-bit address to manage data in a 16-byte RAM. When it's time to operate, like in a *load word* instruction, it reads data based on the MemRead flag. Conversely, for a *store word* instruction, it writes data with the help of the MemWrite flag. The clock regulates these actions, ensuring a synchronized and efficient process. Think of it as a data handler, smoothly processing read and write operations in response to specific instructions and clock signals.

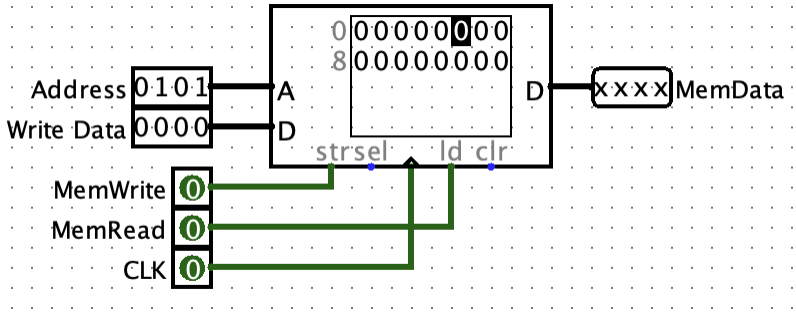


Figure 9: Data Memory

4.7 Additional MUX and Adder

To manage our 4-bit Program Counter (PC), we employed a 7483 Adder for efficient value calculation. The PC value can either be incremented by 1 or adjusted by adding a branch distance, depending on the instruction. For simplicity, we set the carry-in (cin) value to 1, streamlining the implementation to only require two 4-bit adders. Additionally, we incorporated several multiplexers to control the data flow to input pins, enhancing the overall flexibility and functionality of the system. This design choice allowed for dynamic data routing, optimizing the system's adaptability in various scenarios. Furthermore, the use of multiplexers contributed to the efficient management of input data, ensuring precise control over the direction of information flow.

4.8 Registers

In our simulation, D-flipflops were utilized as 8-bit registers, effectively storing and handling data within the system. This approach ensured reliable and synchronized storage, enhancing the overall performance of the design.

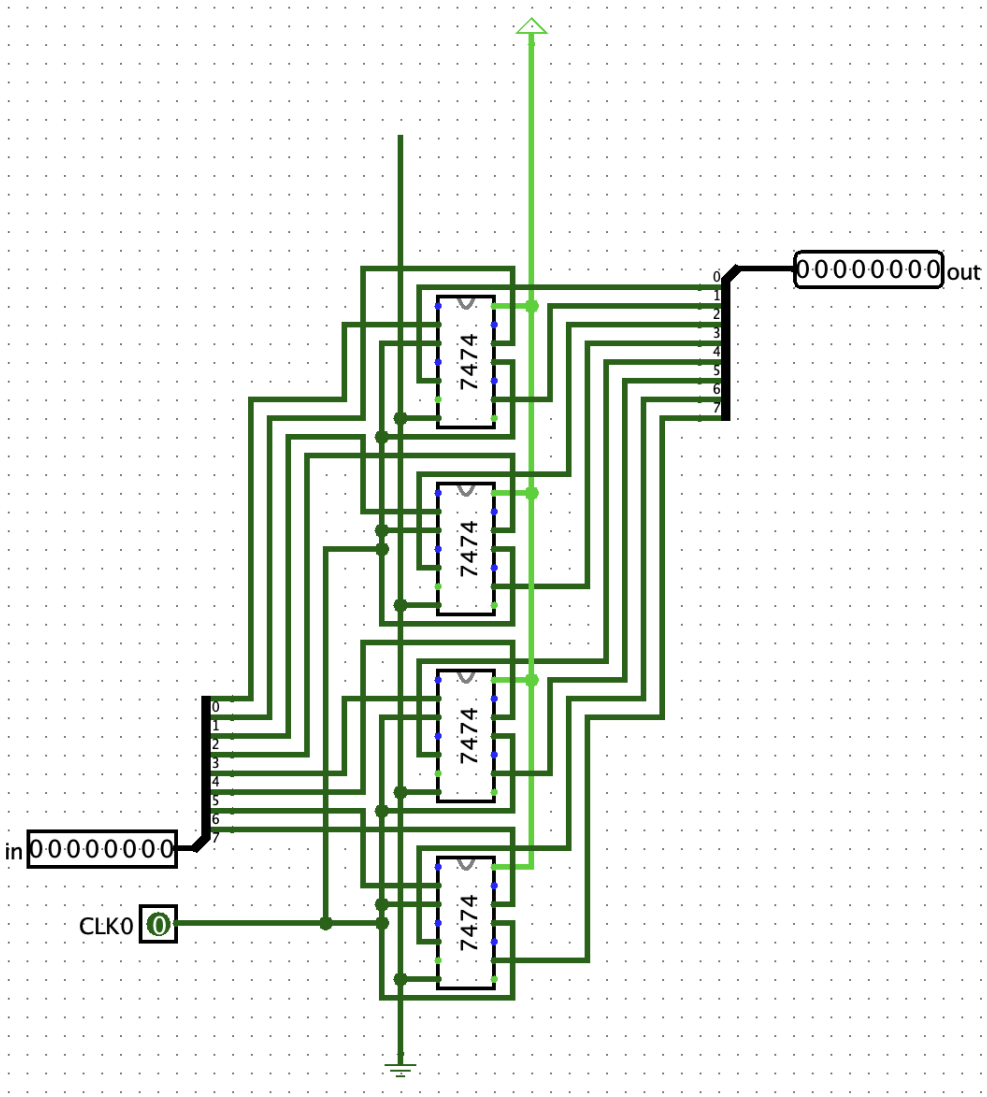


Figure 10: 8-bit Register

4.9 Full Circuit

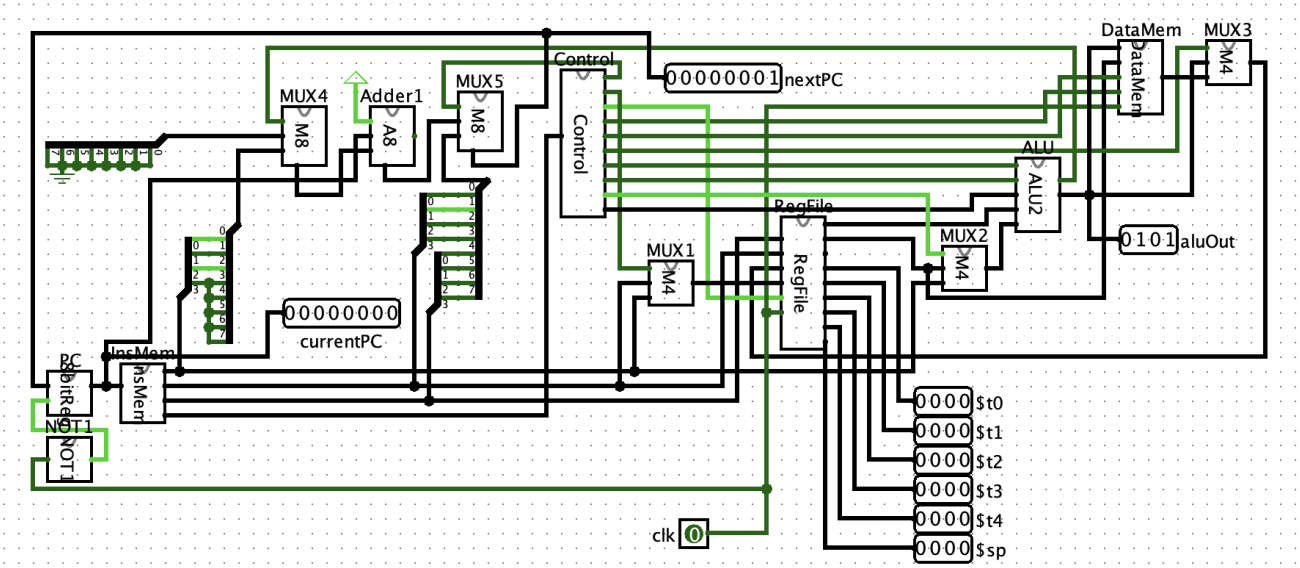
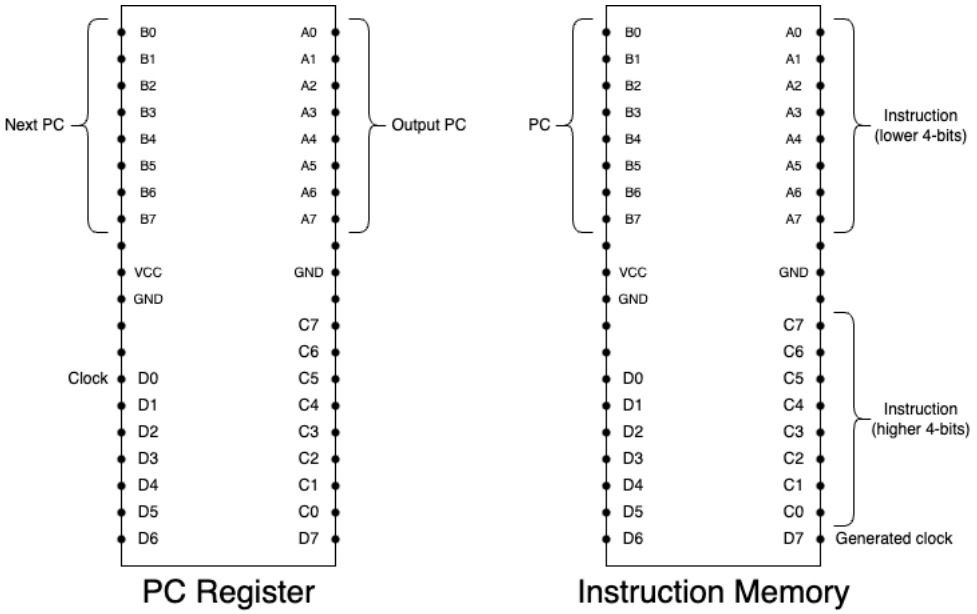
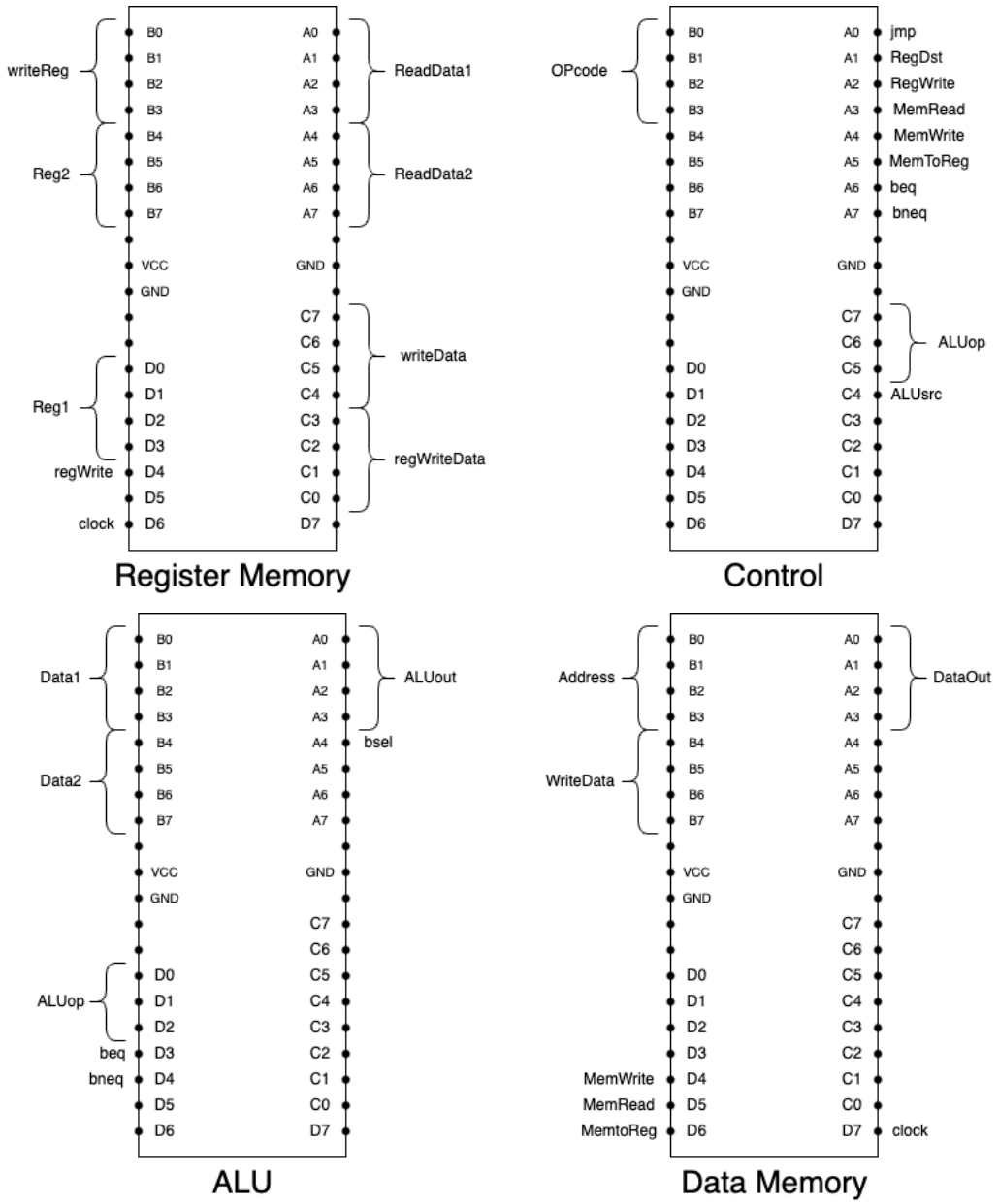


Figure 11: Full Circuit

4.10 ATmega32A Pin Diagrams for Hardware

We used ATmega32A to implement the Program counter register, Instruction memory ROM, Control ROM, ALU (arithmetic logic unit) circuit, Register memory, and Data memory RAM in our hardware circuit.
Here are the corresponding pin diagrams:





4.11 Simulator Software

Software: Logisim
Version: logisim-win-2.7.1

5 Approach to implement the push and pop instructions

We broke down these two instructions into two or three instructions. This part is handled in our Assembly to MIPS instruction converter code. We stored the highest address of the data memory in sp at the very beginning of our each instruction cycle.
Here are the detailed steps of our push-pop implementation:

push $t0	mem[$sp] = $t0	**addi** $t1, $t0, 0 **sw** $t1, 0($sp) **subi** $sp, $sp, 1
push 3($t0)	mem[$sp] = mem[$t0+3]	**lw** $t1, 3($t0) **sw** $t1, 0($sp) **subi** $sp, $sp, 1
pop $t0	$t0 = mem[$sp]	**addi** $sp, $sp, 1 **lw** $t0, 0($sp)

Table 6: Push-Pop implementation

6 ICs used with their count

Here is the IC count of our hardware implementation:

IC Number	IC Name	Count
ATMEGA32A	40 pin microprocessor	6
SN74HC83N	4bit Binary Full Adder	2
SN74HC157N	Quad 2 to 1 MUX	6

Table 7: Caption

7 Discussion

In the software implementation of our circuit, we utilized the 7400-library integrated circuits to emulate the logic circuits. Meanwhile, at the hardware level, we executed the necessary coding specifically at the ATMEGA level, ensuring a seamless integration between the software and hardware components.

During the hardware testing phase, each ATmega microcontroller underwent thorough individual testing before being integrated into the final circuit. This meticulous testing approach aimed to identify and address any potential issues with the microcontrollers before they became part of the larger system.

To enhance the user interface and facilitate comprehension, wires of different sizes were strategically employed in the circuit layout. This organizational approach contributed to a clearer and more user-friendly design, making it easier for users to understand the various components and their connections.

In the hardware-level demonstrations, we activated the ShowReg feature in the register file to visually showcase the values of each register. The ALU outputs were displayed at each step of operation, providing insights into the intermediate results of arithmetic and logical operations. Furthermore, the values of the Program Counter (PC) and data memory were also visually accessible at the software level.

To optimize the use of integrated circuits (ICs), we implemented specific strategies. For instance, instead of incorporating a separate adder for PC+1, we leveraged the carry-in (Cin) value of the existing adder. Similarly, to determine the selector bit (bsel) for branching, we calculated the value within the ATmega designated for ALU operations. These optimization techniques aimed to reduce the overall complexity and resource requirements of the circuit.

Clock pulses, crucial for synchronization, were provided through both an SPDT switch for manual control and an automatic clock generated by an idle ATmega. This additional clock source offered flexibility, allowing users to connect it to the clock circuit as needed.

8 Contribution of Each Member

Activity	**Roll**
Planning and Steps Designing	61, 87, 89
Instruction Codes	61, 72
ATmega32 Programs	79, 89
Block Diagrams	72, 87
Software Implementation	61, 79, 87
Equipment Management	72, 87, 89
Hardware Implementation	61, 72, 79, 87, 89
Hardware Circuit Checking	61, 72, 79, 89
Debugging and Fixing Errors	61, 79, 89
Report	87, 72, 79

Table 8: Contribution Chart