CSE 306
Computer Architecture Sessional

Assignment-3
4-bit MIPS Implementation

Section - A1
Group - 01

Group Members:

1. 2005001 - Anik Saha
2. 2005012 - Abrar Jahin Sarker
3. 2005013 - Al Muhit Muhtadi
4. 2005017 - Abdullah Muhammed Amimul Ehsan
5. 2005023 - Jaber Ahmed Deeder

1 Introduction

This assignment presents the design and implementation of a 4-bit MIPS (Microprocessor without Interlocked Pipeline Stages) architecture in hardware. The objective is to develop a simplified version of the MIPS architecture suitable for educational purposes, focusing on understanding fundamental concepts such as instruction formats, data paths, and control units.

The 4-bit MIPS architecture is characterized by its limited data bus size of 4 bits and an address bus size of 8 bits. The design incorporates essential components such as a 4-bit ALU (Arithmetic Logic Unit), register file with temporary registers, and separate memory units for instructions and data.

The instruction set of the 4-bit MIPS architecture consists of various categories, including arithmetic, logic, memory access, and control instructions. Each instruction is represented by a unique opcode and follows one of the four instruction formats: R-type, S-type, I-type, or J-type. These formats specify the arrangement of opcode, source and destination registers, immediate values, and shift amounts within a 16-bit instruction word.

To facilitate the execution of instructions, the MIPS architecture utilizes two distinct memory units: instruction memory and data memory. Instruction memory is accessed through the Program Counter (PC) register, while data memory is accessed based on the provided address.

Our implementation of MIPS contains these key components:

1. **Program Counter** The Program Counter (PC) is a critical component of the MIPS (Microprocessor without Interlocked Pipeline Stages) architecture, serving as a register that stores the memory address of the next instruction to be fetched and executed. Below are key points explaining the role and functionality of the Program Counter in a MIPS processor:

   - **Incrementing:** After fetching an instruction from memory, the PC is automatically incremented to point to the next sequential memory address where the next instruction resides. This ensures the sequential execution of instructions in the program.
   - **Branching:** The PC can be modified during program execution to implement control flow operations such as branches and jumps. Conditional branches and unconditional jumps change the PC value to redirect the flow of control to different memory addresses based on certain conditions or program requirements.
   - **Jumping :** The PC holds the address of the instruction to jump in case of jump instructions.
   - **Reset:** Upon system startup or reset, the PC is typically initialized to a predefined memory address, usually the beginning of the program or the operating system kernel, to start program execution from a known starting point.

   In summary, the Program Counter (PC) in a MIPS processor is responsible for maintaining the address of the next instruction to be executed, facilitating sequential instruction execution, controlling program flow through branches and jumps, and participating in lots of other mechanisms. It is a critical component of the MIPS architecture, enabling efficient and orderly execution of programs.

2. **Instruction Memory**

   Instruction Memory is a component of the MIPS processor responsible for storing machine instructions that the processor fetches and executes during program execution.Below are key points explaining the role and functionality of Instruction Memory in a MIPS processor:

- **Purpose:** The primary purpose of Instruction Memory is to hold the program instructions that the processor needs to execute. These instructions are fetched sequentially from memory and transferred to the instruction pipeline for decoding and execution.
- **Organization:** Instruction Memory is organized as a sequential array of memory cells or locations, each containing one instruction word. In a typical MIPS architecture, each instruction word is 32 bits (4 bytes) long.Here,our instructions are 16 bits(2 Bytes) long.
- **Access:** During the fetch stage of the instruction pipeline, the processor retrieves the instruction at the current Program Counter (PC) address from the Instruction Memory. The fetched instruction is then decoded and executed in subsequent stages of the pipeline.
- **Addressing:** Instruction Memory is accessed using memory addresses provided by the Program Counter (PC) register. The PC holds the address of the next instruction to be fetched, and this address is used to access the corresponding instruction in memory.
- **Performance Considerations:** Efficient access to Instruction Memory is critical for overall processor performance. Techniques such as prefetching, branch prediction, and cache optimization may be employed to enhance instruction fetch efficiency and reduce stalls in the pipeline.
- **Instruction Alignment:** In MIPS architectures, instructions are typically aligned in memory, meaning they start at addresses that are multiples of their size (e.g., 4-byte aligned for 32-bit instructions or 2-byte aligned for 16-bit instructions). This alignment ensures efficient memory access and simplifies instruction fetching.

In summary, Instruction Memory in a MIPS architecture serves as the storage space for program instructions, facilitating their sequential retrieval and execution by the processor. It plays a crucial role in overall processor performance and is accessed using memory addresses provided by the Program Counter.

3. **Register File :** The register file is a crucial component of the MIPS (Microprocessor without Interlocked Pipeline Stages) architecture, providing storage space for operands, intermediate results, and other data during program execution. Below are key points explaining the register file in the context of the 4-bit MIPS architecture:

   - **Size and Composition:** The register file consists of several 4-bit registers, each capable of storing values from 0 to 15 ($2^4 - 1$). Specifically, it includes the following temporary registers:
     - $zero: A special register that always holds the value zero.
     - $t0,$ t1,$ t2,$ t3,$ t4: Temporary registers used for general-purpose storage and computation.
   - **Purpose:** The register file serves as a fast and efficient storage location for data that the processor needs to access frequently during program execution. It allows for quick retrieval and manipulation of operands for arithmetic, logic, and data movement operations.
   - **Usage in Assembly Code:** The assembly code provided to simulate the MIPS design will utilize the above-mentioned registers for data manipulation. Instructions in the code will specify these registers as the source and destination operands for various operations.
   - **Access and Operation:** During program execution, the processor accesses the register file to read operand values and store results. The control unit coordinates the read and write operations to the register file based on the instructions being executed.
   - **Benefits:** Using registers for temporary storage offers several advantages, including fast access times, efficient data movement, and support for parallel execution of instructions.

   In summary, the register file in the 4-bit MIPS architecture provides storage space for operands and intermediate results, facilitating efficient data manipulation and computation during program execution.

4. **ALU :** The Arithmetic Logic Unit (ALU) is a crucial component of the MIPS (Microprocessor without Interlocked Pipeline Stages) architecture. Below are key points explaining the role and functionality of the ALU in a MIPS processor:

   - **Functionality:** The ALU executes arithmetic operations such as addition, subtraction, multiplication, and division, as well as logical operations such as AND, OR, XOR, and shift operations.
   - **Data Width:** In the context of the MIPS architecture, the ALU typically operates on data of fixed width. For example, in a 32-bit MIPS processor, the ALU handles 32-bit data.Here,Our ALU handles 4 bit data.
   - **Inputs:** The ALU receives input operands from registers or immediate values specified in instructions. These operands are passed to the ALU for processing.
   - **Operation Selection:** The operation to be performed by the ALU is determined by the opcode field of the instruction being executed. The opcode specifies the type of operation to be executed (e.g., addition, subtraction, logical AND, logical OR).
   - **Output:** After performing the specified operation, the ALU produces a result, which is then stored in a destination register or used as input for subsequent instructions.
   - **Flags:** In addition to the result, the ALU may set status flags based on the outcome of the operation. Common flags include zero flag (indicating if the result is zero), carry flag (indicating if a carry occurred in arithmetic operations), and overflow flag (indicating if the result exceeds the representable range).Zero flag is necessary for branching.
   - **Speed and Efficiency:** The ALU is designed to execute operations quickly and efficiently, as it is a critical component of the processor responsible for performing the bulk of the computational tasks.

   In summary, the Arithmetic Logic Unit (ALU) in the MIPS architecture is responsible for executing arithmetic and logical operations on data. It plays a crucial role in performing computational tasks and manipulating data within the processor.

5. **Data Memory :** Data Memory, also known as the Data Cache or Data Cache Memory, is a fundamental component of the MIPS (Microprocessor without Interlocked Pipeline Stages) architecture. It serves as the main storage space for variables, arrays, and other data structures.Below are key points explaining the role and functionality of Data Memory in a MIPS processor:

   - **Purpose:** The primary purpose of Data Memory is to hold data that the processor reads from or writes to during program execution. This includes both input data and intermediate results generated by the processor.
   - **Organization:** Data Memory is organized as a sequential array of memory cells or locations, each capable of storing one data word. The size of each data word depends on the architecture but is typically 32 bits (4 bytes) in a MIPS processor.Here,the size of each word is 4 bits.
   - **Access:** During program execution, the processor accesses Data Memory to read input data or store computed results. Data is read from or written to memory using memory addresses provided by the instruction being executed.
   - **Addressing:** Data Memory is accessed using memory addresses provided by the instruction being executed. The memory address specifies the location in Data Memory from which data is to be read or written.

   In summary, Data Memory in the MIPS architecture serves as the primary storage space for data manipulated by the processor during program execution. It plays a crucial role in storing input data, intermediate results, and program variables, and efficient access to Data Memory is essential for overall processor performance.

6. **Control Unit :** The control unit in a MIPS (Microprocessor without Interlocked Pipeline Stages) processor is a vital component responsible for generating control signals that coordinate the execution of instructions and manage data flow within the processor. Below are key points explaining the control unit in the context of the provided MIPS instructions:

    - **Functionality:** The control unit interprets the opcode of each instruction and generates the necessary control signals to coordinate the operation of other components in the processor, such as the ALU, register file, and data memory.Multiplexers receives their control bits from control unit and provides the expected bits.
    - **Instruction Categories:** The provided instructions can be categorized into arithmetic, logic, memory access, and control instructions. Each instruction category may require specific control signals to be generated by the control unit.
    - **Control Signals:** Based on the opcode of each instruction, the control unit generates control signals to perform various tasks, including selecting the appropriate operation for the ALU, enabling data transfers between the register file and data memory, and managing program flow based on conditional branches and unconditional jumps.
    - **Microprogramming:** In the described MIPS architecture, the control unit should be microprogrammed. This means that the control signals associated with different operations are stored in a special memory unit, such as EEPROM, as control words. The control unit fetches these control words based on the opcode of the current instruction being executed.
    - **Integration with Other Components:** The control unit interacts closely with other components of the processor, such as the ALU, register file, program counter (PC), and data memory. It coordinates the timing and sequencing of operations to ensure correct execution of instructions.

    In summary, the control unit in the MIPS architecture plays a crucial role in coordinating the execution of instructions by generating appropriate control signals, interpreting opcodes, and managing data flow within the processor.

Apart from these,we needed 6 multiplexers.Multiplexers are used to send the correct bits to the next unit.These multiplexers are controlled by control unit.

Adders were required to calculate address of the next instruction where Program counter will point to.Two adders were needed here.

2 Instruction Set

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

Table 1: Instruction Set

3 Complete Block diagram of a 4-bit MIPS processor

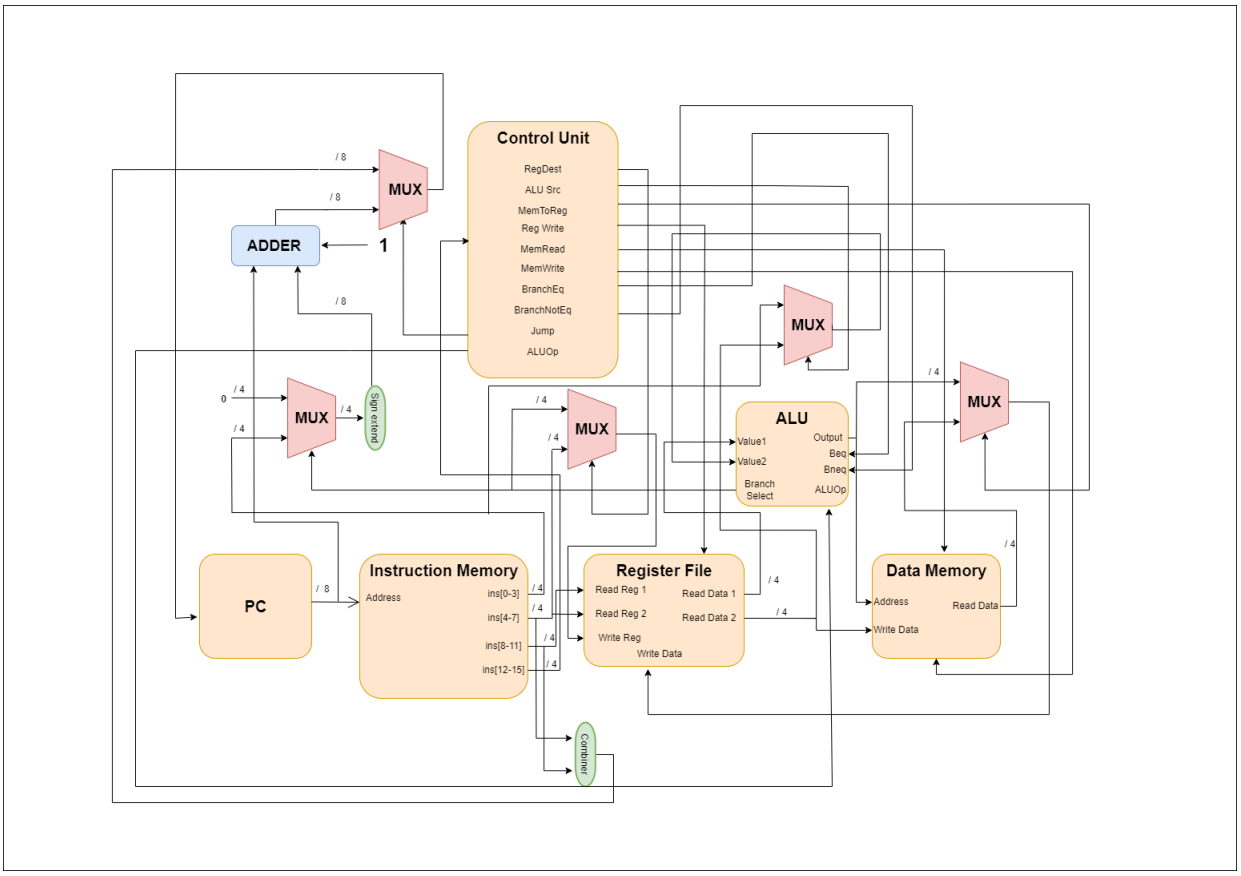

Figure 1: Complete Block Diagram

4 Block diagrams of the main components

4.1 PC Register

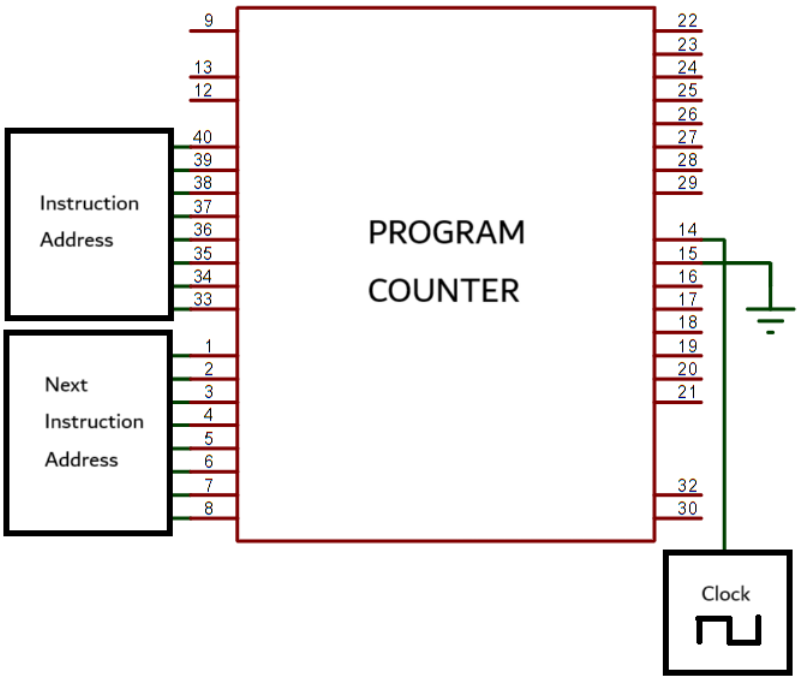

Figure 2: PC Register

4.2 Instruction Memory

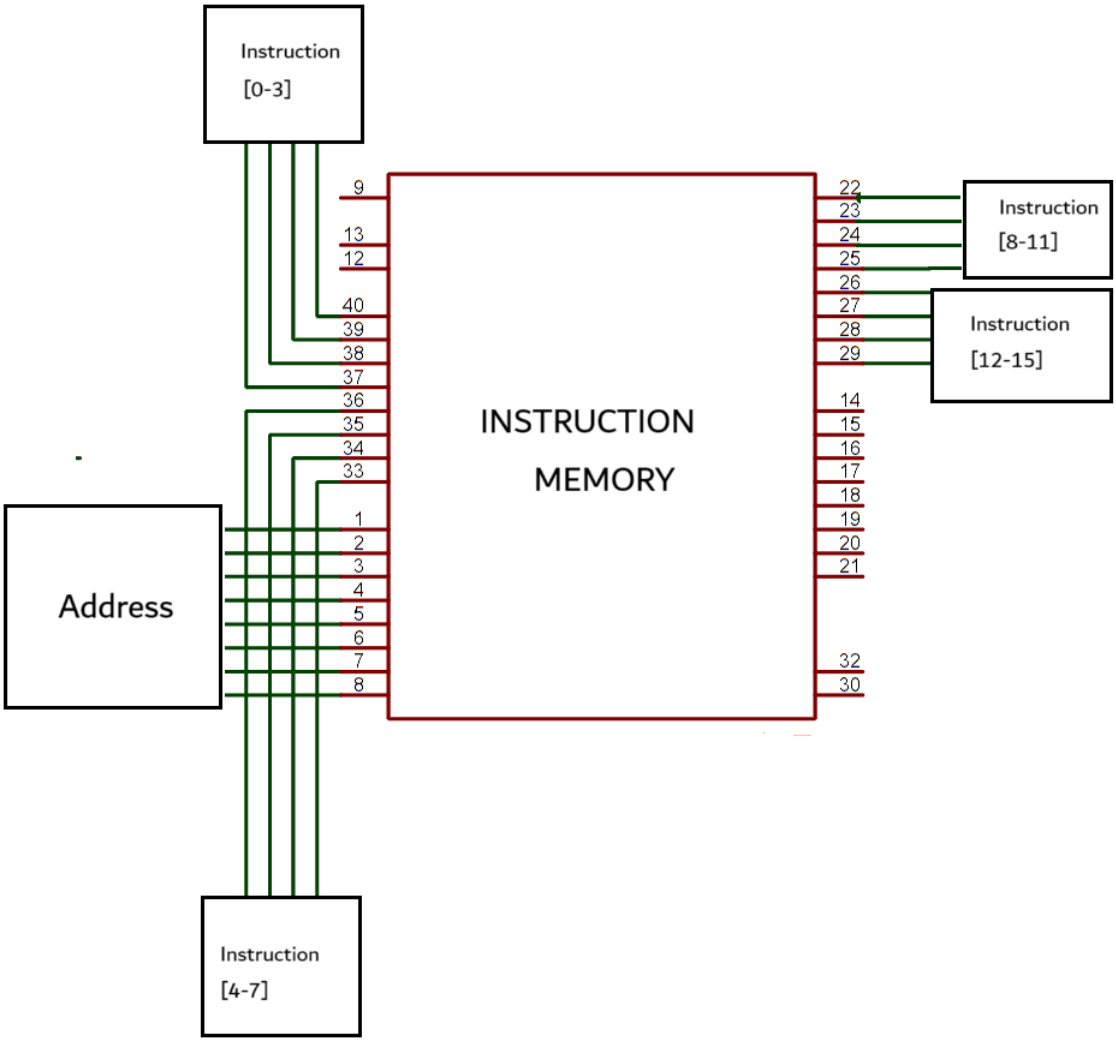

Figure 3: Instruction Memory

4.3 Register Files

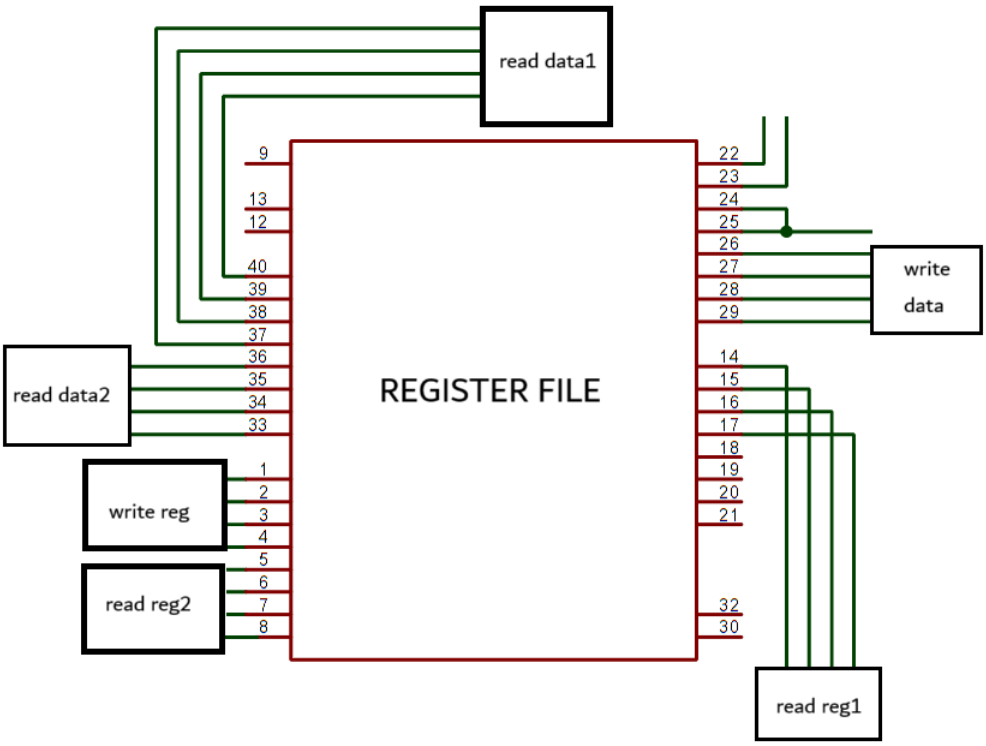

Figure 4: Register Files

4.4 Arithmetic and Logic Unit

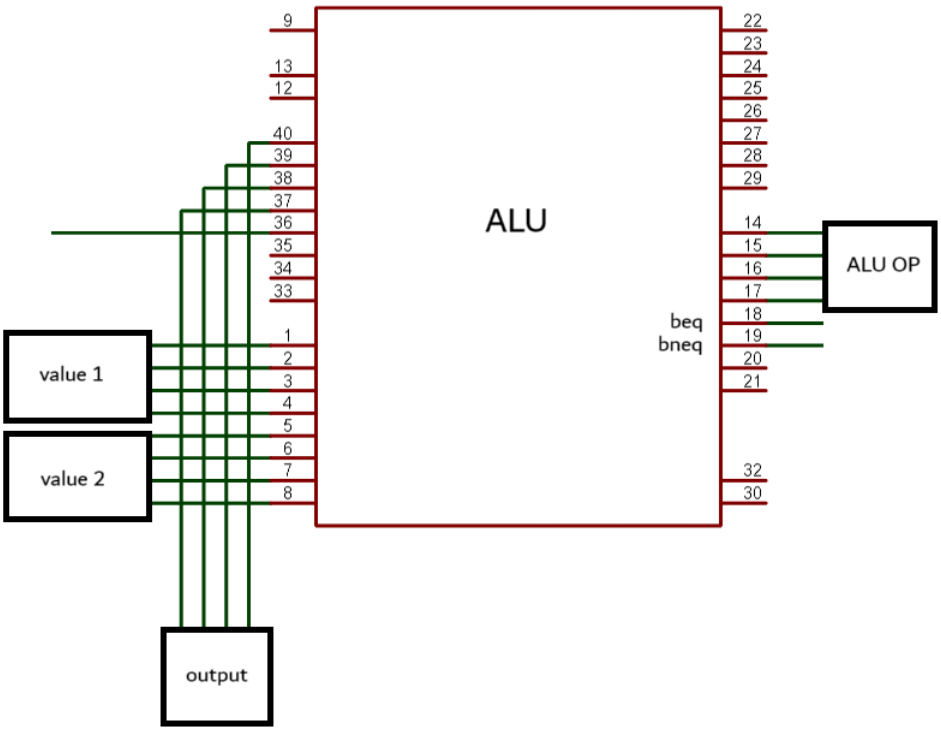

Figure 5: Arithmetic and Logic Unit

4.5 Data Memory

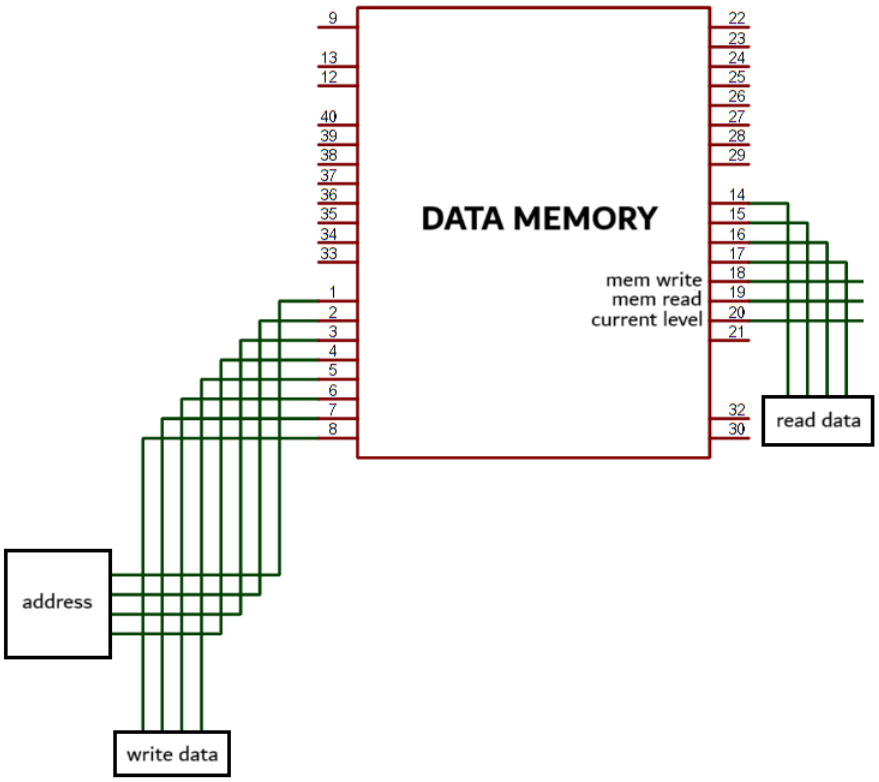

Figure 6: Data Memory

5 Approach to implement the push and pop instructions

According to the specification, following are the formats of push and pop instructions which need to be incorporated into our design.

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

Table 2: Push and Pop Instruction Formats

The salient idea behind implementing each of the operations is that these instructions can be viewed as compound instructions consisting of two or more simple instructions that are directly supported by our design. Let us explore the details of the approach for each of them.

1. **Register to Stack:** The first instruction refers to pushing the value of a particular register to the stack. The format is **push** $< register\_name >$

   Example:

   ```
   push $t0
   ```

   This instruction may be viewed as comprising the following steps:

   (a) Decrease the value of **$sp** by one to indicate that a new value is pushed onto the stack

   (b) Store the value at the register **$t0** to the topmost position of the stack. The topmost position is denoted by the address pointed to by the register **$sp**.

   It is easy to see that the corresponding pair of assembly instructions for the above-mentioned steps are:

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

   Note that, since this instruction is broken down into 2 simple instruction, it requires 2 clock cycles to execute.

2. **Memory to Stack:** The second instruction refers to pushing the value of a particular memory location to the stack. The format is **push** $< memory_address >$

   Example:

   ```
   push 3($t0)
   ```

   To accomplish this instruction, we are using a hidden register which we have named as $hd.

   This instruction may be viewed as comprising the following steps:

   (a) Decrease the value of **$sp** by one to indicate that a new value is pushed onto the stack

   (b) Load the value at the topmost position of the stack to the hidden register **$hd**. The topmost position is denoted by the address pointed to by the register **$sp**

(c) Store the value at the hidden register **$hd** to the topmost position of the stack. The topmost position is denoted by the address pointed to by the register **$sp**.

It is easy to see that the corresponding trio of assembly instructions for the above-mentioned steps are:

```
subi $sp, $sp, 1
lw $hd, 3($t0)
sw $hd, 0($sp)
```

Note that, since this instruction is broken down into 3 simple instruction, it requires 3 clock cycles to execute.

3. **Stack to Register:** The third instruction refers to popping the topmost value on the stack and loading that value into a particular register. The format is **pop** $< register\_name >$

Example:

```
pop $t0
```

This instruction may be viewed as comprising the following steps:

(a) Load the value at the topmost position of the stack to the register **$t0**. The topmost position is denoted by the address pointed to by the register **$sp**

(b) Increase the value of **$sp** by one to indicate that a value is popped from the stack.

It is easy to see that the corresponding pair of assembly instructions for the above-mentioned steps are:

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

Note that, since this instruction is broken down into 2 simple instruction, it requires 2 clock cycles to execute.

6 ICs used with their count

IC category	IC Number	Count
	ATMEGA32A	6
$2 \times 1 Multiplexer$	74157	6
4 bit Adder	7483	2
Total		14

Table 3: IC Count

7 Discussion

This assignment on the implementation of a 4-bit-MIPS processor significantly helped us bridge through the chasms of our knowledge of Computer Architecture enabling us to visualize the entirety of the whole workflow from writing a piece of code in a fancy editor to carrying out bit-level instructions in real hardware. Although a shade of abstraction becomes prevalent owing to the use of microcontrollers in the design, the underlying ideas remain all the same and the stream of actions replicate the actual process to a considerable extent.

Prior to designing the circuitry, we needed to have developed a codebase for parsing an assembly file written in MIPS assembly language to machine-readable code. The codebase, written in python, parses the given assembly file, reports errors(if any), and converts it to a machine code as per the given instruction set corresponding to our assigned permutation. Since the control bits have been micro-programmed, it also generates a hard-coded array of binary strings representing the control bits for different opcodes. These binary arrays for control bits and instructions are in turn used in the software simulation.

To perform the software simulation, we preferred to resort to Proteus since we used the microcontroller named ATmega32 in our design. To ensure that the use of microcontroller does not trivilalize our design, we used separate ATmega's for each of the stages of our MIPS processor. One additional ATmega was used to serve as the Control Unit. Furthermore, we used multiplexers and 4-bit adders for the calculation of the next value of program counter. For showing the 'write-back-value' of the current instruction, we connected 4 LEDs to the corresponsing pins of the relevant ATmega32. To trace the workflow of the jump and branch instructions, we also connected 4 LEDs showing the lower 4 bits of the program counter(assuming that the number of instructions will not exceed 16, for demonstration purposes).

Then comes the hardware implementation filled with multi-dimensional challenges. To parallelize our activities, we kept building the two halves of the circuit simultaneously and merged them afterwards. Extensive care was taken to avoid wire-crossings so that the circuit looks cleaner. Wires were meticulously inserted into breadboard pins so that they don't touch other adjacent wires and thereby cause short circuits. Since ATmega32 is highly sensitive to the provided voltage and a sudden uprise of voltage may rupture the internal circuitry, preventing unintended shorting connections was of paramount importance. After the circuit was built, the hex files were burnt into the microcontrollers with the help of 'Extreme Burner'. Although 5 of them need to be burnt only once, the ATmega acting as the Instruction Memory need to be burnt every time every time the intended assembly code changes. The corresponding machine code is generated for the given assembly code and is then loaded onto that microcontroller.

To assess the outputs, we speculated the 'write-back-value's of each instruction beforehand and later matched them with the outputs in LEDs. In case of branch and jump instructions, we also kept an eye on the program counter LEDs to detect whether the program control switches to the appropriate place.

One particular hurdle we needed to battle against was themechanical bouncing of push button switch that we used to manually trigger clock edges. To introduce debouncing, we added a delay of 500ms right after the detection of a clock edge so that subsequent unintended bouncing do not cause the program counter to leap forward.

In a nutshell, this implementation helped us gain invaluable pieces of experience that will guide us through further explorations in the coming days. We hope to explore pipelining and other optimization techniques in future to widen our understanding of processor design.

8 Contribution of Each Member

8.1 Software Simulation

Student ID	Surname	Contribution
2005001	Anik	Developed the python codebase for a MIPS assembler, jointly prepared the code for the microcontroller in Control Unit
2005012	Jahin	Prepared the code for the microcontroller in Data Memory
2005013	Muhit	Prepared the code for the microcontroller in ALU
2005017	Amim	Prepared the code for the microcontroller in Register Files, jointly prepared the code for the microcontroller in Control Unit
2005023	Jaber	Prepared the code for the microcontroller in PC Register

8.2 Hardware Implementation

Student ID	Surname	Contribution
2005001	Anik	Jointly implemented Register Files, ALU, Data Memory
2005012	Jahin	Jointly implemented PC Register, Instruction Memory, Control Unit
2005013	Muhit	Jointly implemented PC Register, Instruction Memory, Control Unit
2005017	Amim	Jointly implemented Register Files, ALU, Data Memory
2005023	Jaber	Jointly implemented PC Register, Instruction Memory, Control Unit

8.3 Verification and Testing

Student ID	Surname	Contribution
2005001	Anik	Jointly generated extensive testcases for simulation and performed relevant debugging in hardware
2005012	Jahin	Verified the procedure for implementing push and pop instructions and the control bits for micro-programming into Control Unit
2005013	Muhit	Jointly generated extensive testcases for simulation and performed relevant debugging in hardware
2005017	Amim	Jointly generated extensive testcases for simulation and performed relevant debugging in hardware
2005023	Jaber	Tested and debugged the assembler codebase and assisted in hardware testing