

CSE 306  
Computer Architecture Sessional

Assignment-3: 4-bit MIPS Design, Simulation, and  
Implementation

Section: A1

Group: 4

Group Members:

ID	Name
2005005	Anup Halder Joy
2005021	Afzal Hossan
2005022	Ekramul Haque Amin
2005024	Asif Karim
2005028	Md. Tamim Iqbal

## **Introduction:**

A processor, or central processing unit (CPU), is the core logic circuitry responsible for executing instructions in a computer system. In this implementation, we focus on a 4-bit processor utilizing a reduced MIPS instruction set architecture (ISA). Let's delve into the essential components required for this processor's functionality.

### **1. Arithmetic Logic Unit (ALU):**

The 4-bit ALU serves as the fundamental component for executing arithmetic and logical operations. It handles arithmetic operations like addition and subtraction, essential for various tasks such as address calculation and data manipulation.

### **2. Program Counter (PC):**

An 8-bit register, the Program Counter (PC) tracks the current instruction being executed. After each instruction, the PC is incremented by one, facilitating sequential instruction fetching from the ROM. The PC stores the memory address of the next instruction to be fetched.

### **3. Instruction Memory:**

The Instruction Memory stores the binary representation of instructions fetched from the ROM. It holds the hex code of instructions, which are then converted into binary form for execution by the processor.

### **4. Data Memory:**

The Data Memory serves as the primary storage for 4-bit data within the processor. It stores data used in computations and other operations during program execution.

### **5. Register File:**

Comprising six registers, namely \$zero, \$t0, \$t1, \$t2, \$t3, and \$t4, the Register File plays a crucial role in storing temporary data and facilitating efficient data manipulation. The \$zero register holds the value 0H, while the others function as general-purpose registers for storing data temporarily during computations.

## 6. Control Unit:

The Control Unit is responsible for decoding instructions fetched from memory. It provides selection inputs to various multiplexers (MUXs), the Register File, Data Memory, and the ALU, orchestrating the flow of data and control signals within the processor.

This implementation of a 4-bit processor enables the execution of instructions from a modified MIPS instruction set architecture, offering fundamental computational capabilities for handling arithmetic, logic, and I/O operations within a computer system.

**Instruction Set:** Our assigned sequence is given below:

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

## MIPS INSTRUCTION FORMAT:

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	Addr./Immdt.
4-bits	4-bits	4-bits	4-bits

### J-type

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

## Complete Block diagram of a 4-bit MIPS:

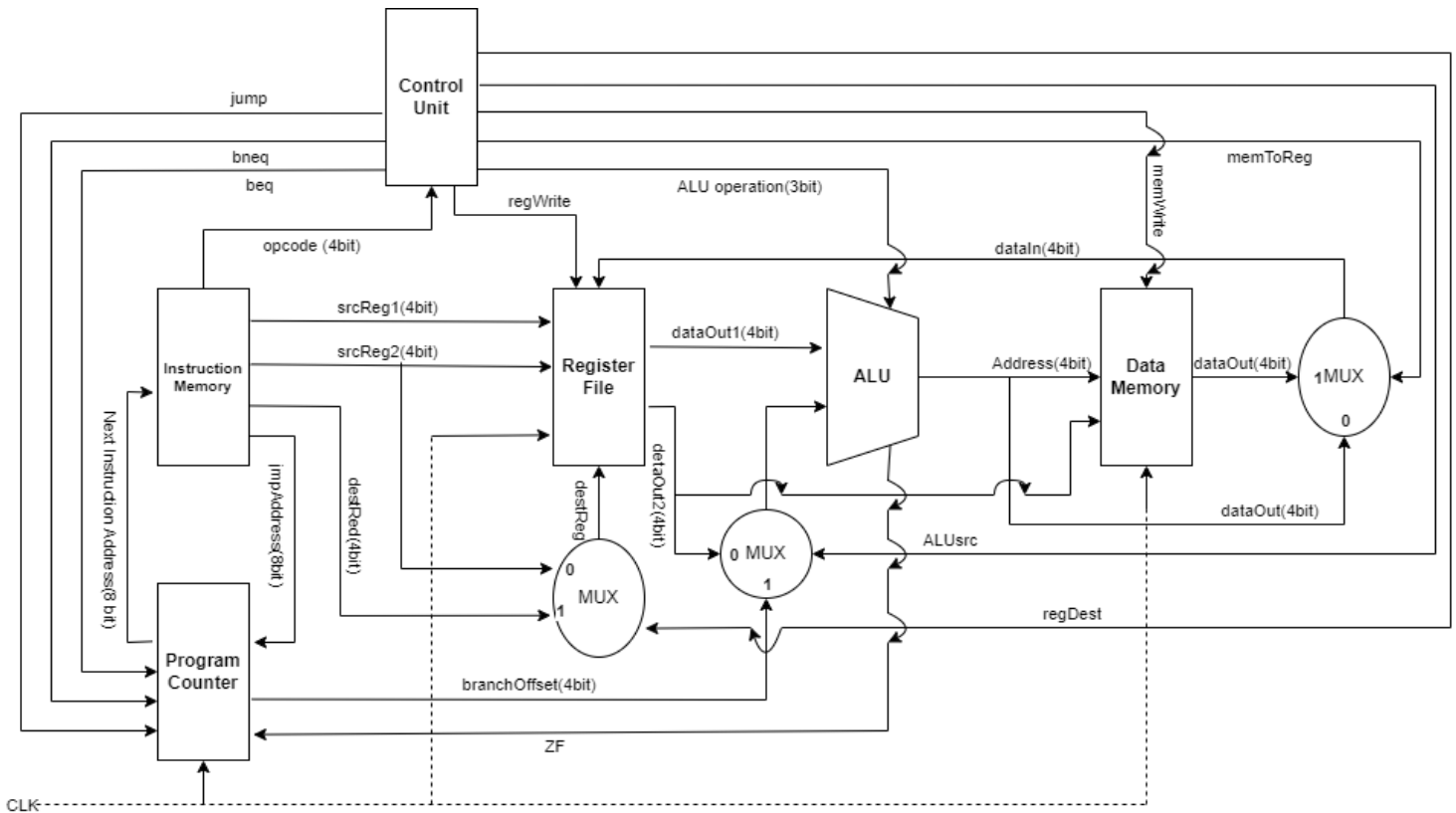


Figure: Complete Block diagram of 4-bit MIPS

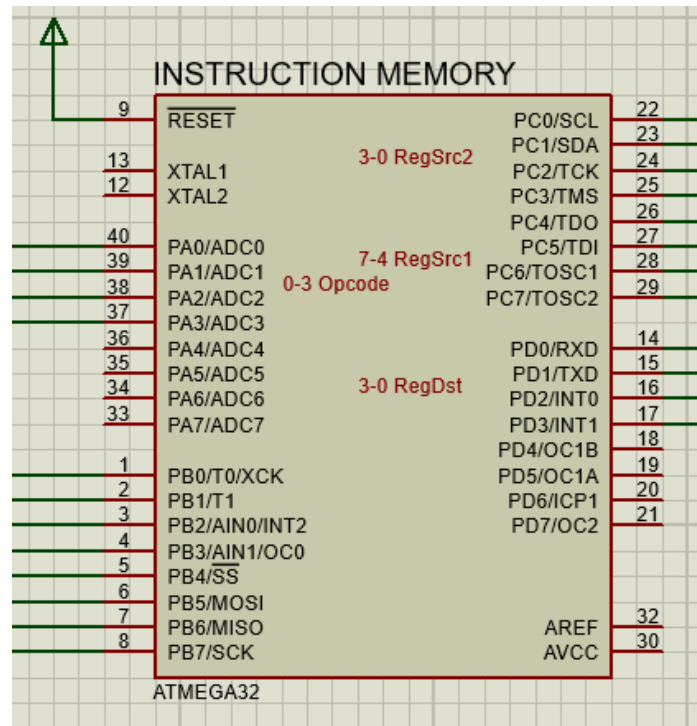
## Diagrams of the main components:

### Instruction Memory:

To implement “Instruction Memory” of MIPS, we have used an ATmega-32. We used a array to store the instruction in **Instruction Memory**.

- PORT A - Port-A (A0 to A3) has been used as an output port. The output of PORTA is opcode and being sent to **Control Unit**.
- PORT B - Port-B has been used to get the address of next instruction in Instruction Memory from **Program Counter**.

- PORT C - Port-C has been used to send address of source Register-1(C4 to C7) and source Register 2(C0 to C3) to the **Register File**. Again, this port is used to send the jump address (C0 to C7) when need to jump to the **Program Counter**.
- PORT D – Lower Nibble of Port-D (D0 to D3) has been used to send the address of destination Register to the **Register File**.



### Program Counter:

To implement “Program Counter” of **MIPS**, we have used an ATMEga-32.

- PORT-A - Port A has been used to output the next instruction’s address. This address goes to **Instruction Memory**.
- PORT-B - Port B has been used to get clock pulse at 0<sup>th</sup> bit.
- PORT-C - Port C has been used to get necessary data for address calculation of next instruction. The lower nibble is the branch offset to execute conditional jump instructions ‘beq’ and ‘bne’. The upper nibble contains control bits from **Control Unit** for beq(5<sup>th</sup> bit), bne(6<sup>th</sup> bit), jump(7<sup>th</sup> bit) and Zero Flag(4<sup>th</sup> bit) from **ALU**.
- PORT-D - Port D has been used to get the jump address to execute the unconditional jump instruction ‘j’.

At first, we initialize the data direction registers. When we get a clock pulse, we check if jump instruction is set from Control Unit. If jump instruction is set, we set the value of next instruction to jump address found from Port D.

Otherwise, we check if branch instructions are set true. If so, we check for the conditions to be true. If the conditions are true then we add branch offset value found from Port C with our current instruction's address.

If no jump instructions are set true, then we normally increase instruction address by 1.

At last, calculated next instruction's address is set to Port A which is sent to Instruction Memory to execute next instruction.

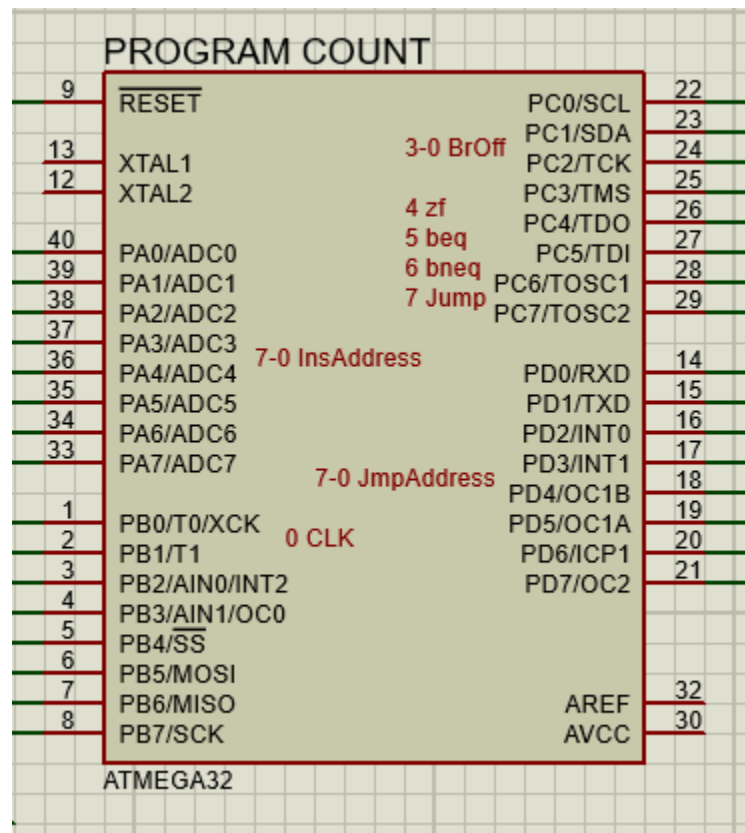


Fig: ATMEGA32 as Program Count

### Data Memory:

To implement “Data Memory” of MIPS, we have used an ATmega-32.

- **PORT A** - Port-A has been used to output data. The data is sent to **Register Files** from a specific memory location. Lower Nibble of output of Port A is the desired data to be sent.

- PORT B – Port-B has been used to get the address of a specific memory location. The lower nibble of input is the desired address.
- PORT C – Port-C has been used to get clock input (0<sup>th</sup> bit) and Control Bit for Memory write (1<sup>st</sup> bit) from control unit.
- PORT D – Lower Nibble of Port-D has been used to get the calculated data from ALU.

At First, Data Direction Registers have been initiated accordingly. When a clock pulse is given, data is sent to Port A from the address got from PINB. Also, if the control bit for memory write is got set by **Control Unit**, Data got from **ALU** is written in the specified address got from PINB.

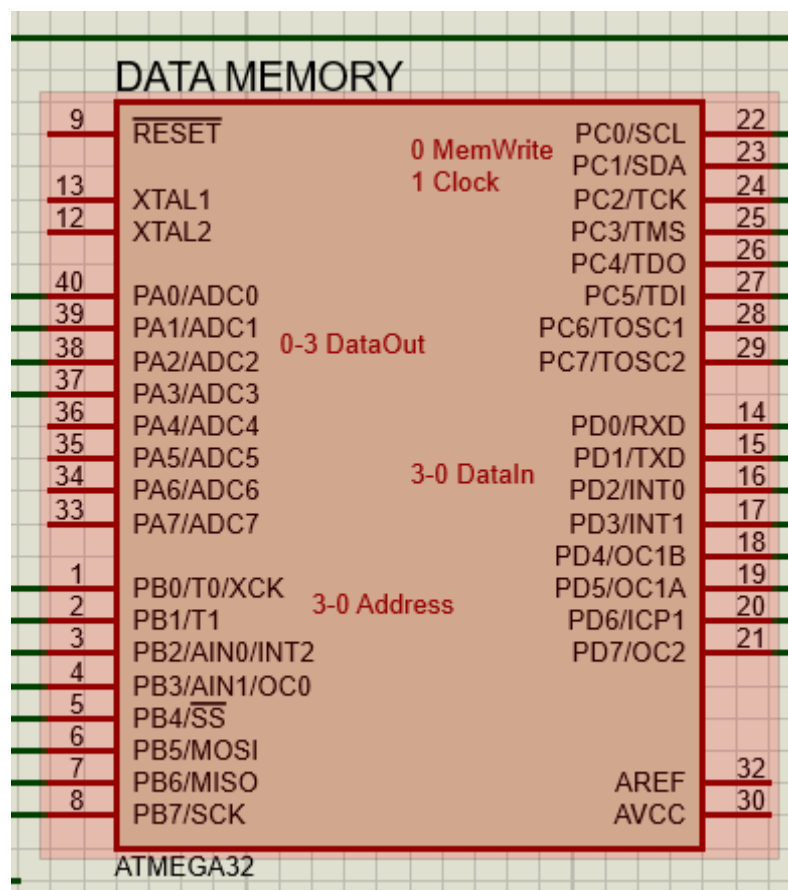


Fig: ATMEGA32 as Data Memory



## Register File:

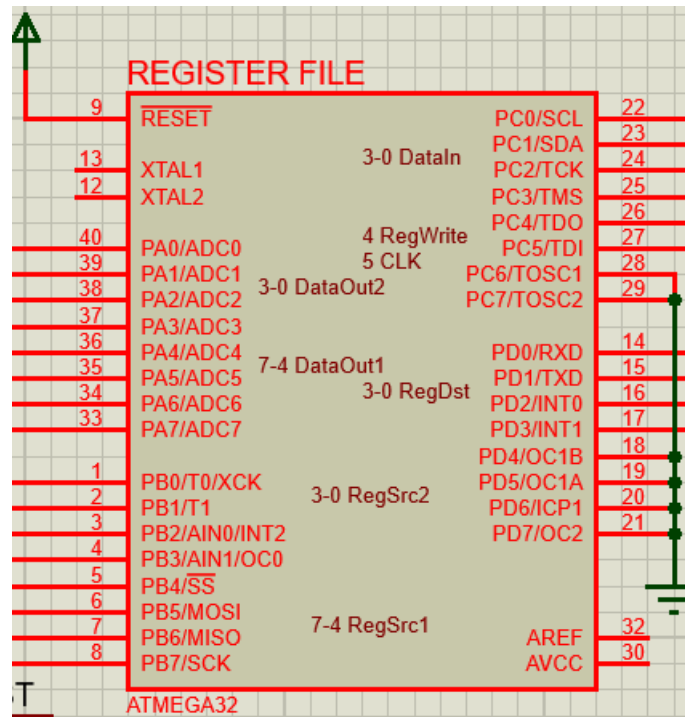


Fig: ATMEGA32 as Register File

We are using atmega32 as Register File for the MIPS.

We took an array to store the data of the registers and initialize them with zero as **Register File**. We must disable the JTAG interface to free up the corresponding I/O pins for general use. JTAG stands for Joint Test Action Group. atmega32 has one JTAG port which shares four pins with port C. To use these pins, one must disable the JTAG port.

There is a register in atmega32 MCUCSR (MCU control and status register). It consists of JTD(JTAG disable)bit as 7<sup>th</sup> bit of register. JTAG can be disabled by writing 1 to this bit.

$$\text{MCUCSR} = (1 \ll \text{JTD})$$

This command must be written twice withing 4 clock cycles to disable JTAG.

We set port A as output and set port B, port C and port D as input.

It reads the 1st source address from upper 4 bit of pin B and gets the corresponding data from the **RegFile** Array and stores it in dataOut\_1.

And it reads the 2nd source address from lower 4 bit of pin B and gets the corresponding data from the **RegFile** Array and stores it in dataOut\_2.

Then it puts the dataOut\_1 in upper 4 bit of port A AND dataOut\_2 in lower 4 bit of port A as output which will be input of ALU for further operation.

We read the current clock state from port C. if the previous clock was zero and the current clock is 1 which means we got a rising edge then we check if the fourth bit which we determined as **RegWrite** is 0 or 1. if it is 1 then we store the value of pin C in the destination Register. The Address of destination Register was taken from port D.

### ALU:

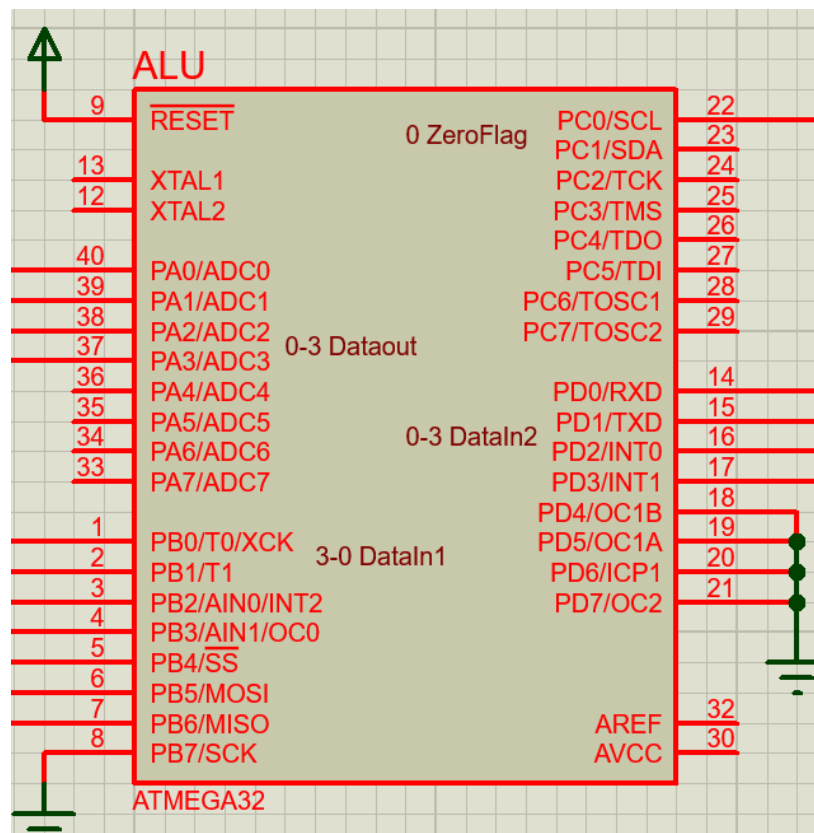


Fig: ATMEGA32 as ALU

We are using atmega32 as ALU unit for the MIPS.

First, we must disable the JTAG interface to free up the corresponding I/O pins for general use.

JTAG stands for Joint Test Action Group. atmega32 has one JTAG port which shares four pins with port C. To use these pins, one must disable the JTAG port.

There is a register in atmega32 MCUCSR (MCU control and status register). It consists of JTD (JTAG disable) bit as 7<sup>th</sup> bit of register. JTAG can be disabled by writing 1 to this bit.

$$\text{MCUCSR} |= (1 \ll \text{JTD})$$

This command must be written twice withing 4 clock cycles to disable JTAG.

We set port A and port C as output and initialize them with zero. And set port B and port D as input.

It reads the OpCode from the upper nibble of port B. It also reads the 1st operand from the lower nibble of port B and 2nd operand from port D.

Then we perform the corresponding operation based on the OpCode received from port B. Which are:

- 000 -> Addition operation.
- 001 -> SRL (logical right shift operation.)
- 010 -> Bitwise OR operation.
- 100 -> SLL (logical left shift operation.)
- 101 -> Bitwise AND operation.
- 110 -> Bitwise NOR operation.
- 111 -> Subtraction operation.

It outputs the result to port A which will be going in Data memory PIN B and "MEM TO REG" MUX. And sets port C based on whether the result is zero or not which will be used as Zero flag

### Control Unit:

Control Unit of MIPS is implemented by another ATmega32.

- PORT A: has been set for giving output the control bits.

Bit 0 -> MemToReg which controls our **memToReg MUX** and it only works when it is a load word instruction and if it is set to 1 then it takes data from the **Data Memory**

Bit 1 -> MemWrite which controls whether we need to write to the **Data Memory** or not. It will work for our store word instruction.

Bit 2 -> Reg Write if it is set to 1 then we need to write data in our destination register of **Register File**. It will be set to 0 if it is jump, store, beq or bneq instruction.

Bit 3-> RegDst this control bit goes to the REG DST MUX which selects which is the destination register for a given instruction. If it is set to 1 then RegDst of INSTRUCTION MEMORY is the destination otherwise Src-2 Reg is the destination.

Bit 4 -> AluSrc this control bit goes to the **ALUSrcMUX** which helps us to select whether we need to do arithmetic operation for address or the data of the register.

Bit 5 -> beq, Bit 6 -> bneq, Bit 7 -> Jump these control bits direct the **Program Count** unit to do respective operation.

- PORT B: The opcode from INSTRUCTION MEMORY comes to this port as an input from which we can set the control bits.
- PORT C: Using this we are giving instruction to the ALU to do the required operation. We have 7 types of general operation eg: ADD, SRL, OR, SLL, AND, NOR, SUB. We have enumerated this operation like for ADD = 0b00000000, SRL=0b00000001 so that we can easily understand the specific operation which is needed to be executed.

At first, we need to do the initialization for setting the port. From input pin PORT B we are getting the opcode. By analyzing the opcode we are setting the control bits to PORT A and specific general operation to PORT C. We have enumerated the control bits for all operation like or\_ctrl=0b00001100, ori\_ctrl=0b0010100, lw\_ctrl=0b00010101. For an example if opcode is for or operation PORT A=or\_ctrl( control bit for or operation). And PORT C=OR(specific general operation for ALU)

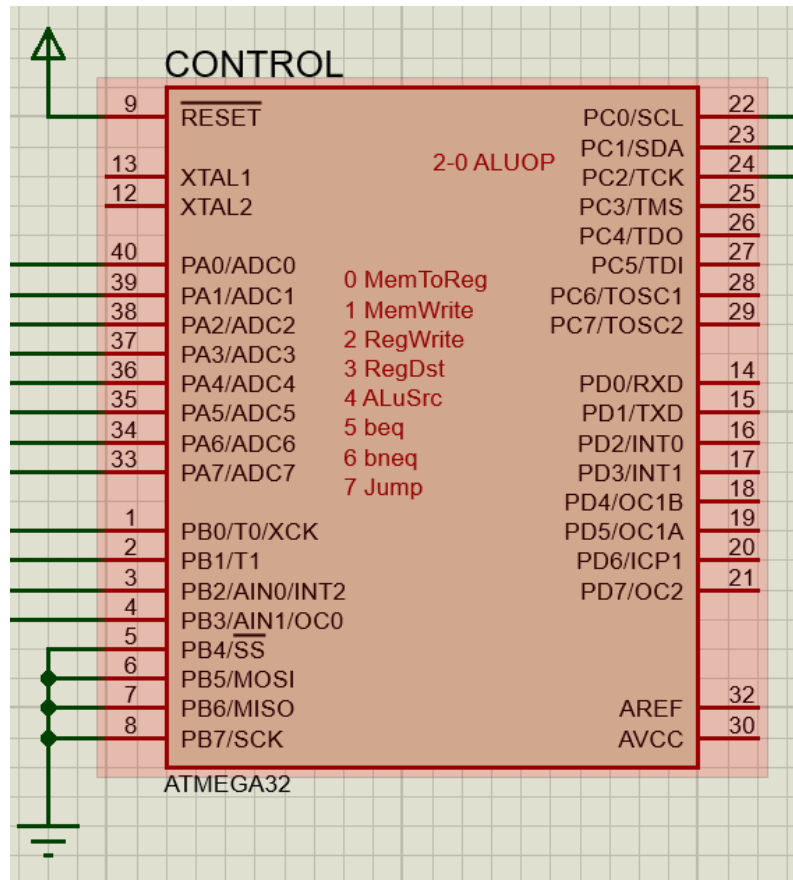


Fig: ATMEGA32 as Control Unit

## Approach to implement the push and pop instructions:

We have a dedicated register called \$sp which store a pointer to the last address (1111) of the data memory. We have two different kind of push operation and a pop operation. We implement these operations using other basic operations. Consequently, each of these operations needs more than one clock. Their implementation is as follow-

### Push type-1 (push \$reg):


Push \$reg is equivalent to  $\text{mem}[\$sp] = \$reg$ . To implement this push operation, first, we store the value of \$reg in memory pointed by \$sp using 'sw' instruction. Then we decrement the value of \$sp by 1 and store the result in \$sp using subi instruction. Hence this push operation needs two clock pulses.

push \$reg		sw \$reg 0(\$sp) subi \$sp \$sp 1
------------	---	--------------------------------------

### Push type-2 (push offset(\$reg)):


push offset(\$reg) is equivalent to operation  $\text{mem}[\$sp] = \text{mem}[\$reg + \text{offset}]$ . To implement this push operation, first,

we load the value of  $\text{mem}[\$reg + \text{offset}]$  in \$tx register (which is only used for this purpose) using 'lw' instruction. Then we store the value of \$tx register in memory pointed by \$sp using 'sw' instruction. Then we decrement the value of \$sp by 1 and store the result in \$sp using 'subi' instruction. Hence this push operation needs three clock pulses.

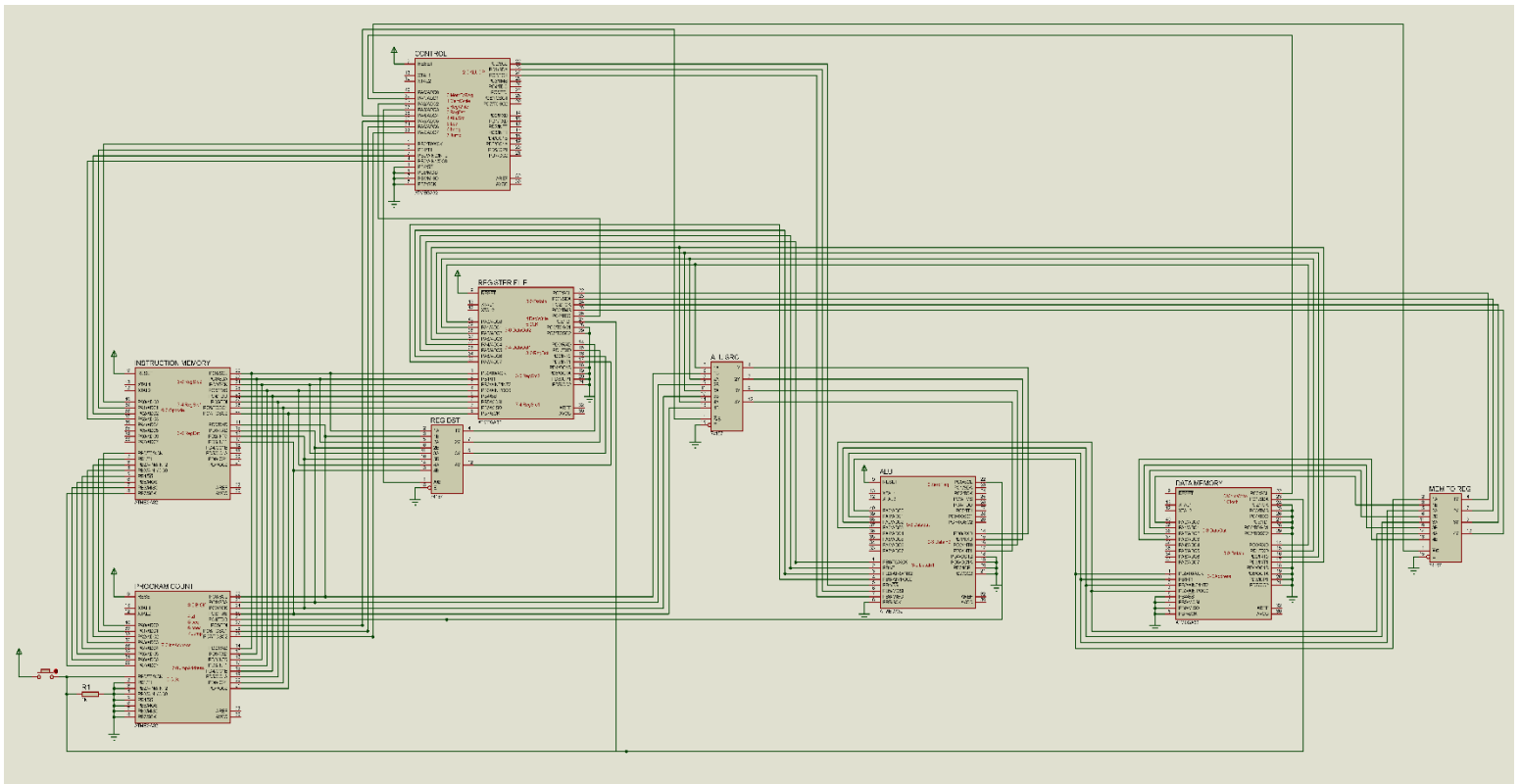
push offset(\$reg)		lw \$tx offset(\$reg) sw \$tx 0(\$sp) subi \$sp \$sp 1
--------------------	---	--

### Pop (pop \$reg):

pop \$reg is equivalent to operation  $\$reg = \text{mem}[\$sp]$ . To implement this push operation, first, we increment the value of \$sp by 1 and store the result in \$sp using 'addi' instruction. Then we load the value of memory pointed by \$sp in \$reg using 'lw' instruction. Hence this pop operation needs two clock pulses.

pop \$reg		addi \$sp \$sp 1 lw \$reg 0(\$sp)
-----------	---	--------------------------------------

## Complete Circuit Diagram:



## ICs Used with Count:

IC	Quantity
ATmega32	6
IC 74157 (MUX)	3
	Total: 9

## The Simulator Used along with the Version Number:

Proteus - 8 - Professional

## **Discussion:**

In this assignment, the primary objective was to implement a 4-bit processor capable of executing a modified MIPS instruction set. Our task involved designing various components, including the Arithmetic Logic Unit (ALU), Program Counter (PC), Instruction Memory, Data Memory, Register File, and Control Unit, while ensuring efficient operation and minimal resource utilization. We used Atmega32 to implement all the main components described above because the ROM ICs are not that much available.

And we all know that we can never be so sure about hardware. The switch we used for clocking creates debounce effect. So, it increases the program count by 2. We tried to solve the problem by our codes for ATmega32.

Throughout the implementation process, we encountered challenges and setbacks that tested our problem-solving skills and resilience. However, despite our careful planning, we encountered unexpected issues during the testing phase. Upon closer inspection, we discovered that certain components were not properly connected, leading to erratic behavior in the processor's functionality. Rectifying these wiring errors proved crucial in restoring the proper operation of the circuit, highlighting the critical importance of attention to detail in hardware design.

The process of debugging and troubleshooting the MIPS processor was undoubtedly challenging, but it also provided valuable learning opportunities. Through perseverance and collaboration, we were able to identify and address issues effectively, gaining a deeper understanding of hardware design principles and techniques along the way. We have tested some instructions made by us as well as the corner cases. The experiment helped us better understand the overall MIPS instruction set and the data-path as a whole.

Ultimately, the successful resolution of these challenges brought a sense of accomplishment and satisfaction to the team. Celebrating the proper functionality of the MIPS processor, despite the obstacles encountered, reinforced the importance of thorough testing, attention to detail, and the iterative nature of engineering projects. This experience equipped us with valuable skills and confidence that will undoubtedly benefit us in future endeavors in hardware design and computer architecture.



## **Contribution:**

ID	Contribution
2005005	Hardware implementation, Report writing, Debugging, Diagram drawing
2005021	Software implementation, Hardware implementation, Debugging, Report writing
2005022	Hardware implementation, Report writing, Debugging
2005024	Hardware implementation, Report writing, Debugging
2005028	Hardware implementation, Report writing, Debugging