CSE 306 OFFLINE 3 LABORATORY REPORT

[1]Sadia Tabassum, [2]Alina Zaman, [3]Md.Shafiul Haque ,
[4]Mayesha Rashid, [5]Saha Kuljit Shantanu

1 March 2022

1 Introduction

A *4 bit MIPS* is a instruction set that contains a databus of only 4 bits, that is , a register can store only four bits of data(Signed or Unsigned) in it.

For this assignment, we have done or attempted the following things:

- Designing a 4 bit MIPS structure with assigned 16-bit instruction set
- Simulating the design in a simulating software(In our case we hve used Logisim-win-2.7.1)
- Implementing the design in hardwares

2 Properties of our Instruction Set Architecture

For this assignment, our instruction set is of following features:

- It is a 16-bit instruction set
- Operational code is of 4 bit
- The data bus are 4 bits each and the address bus is 8 bits
- The conditional Jumps are of 4 bit offset
- Only logical shift is implemented
- The shift amount for logical shifts is of 4 bit
- The register identifiers are of 4 bit
- The following registers are used:

$zero	**$t3**	**$a0**	**$s0**
$t0	**$t4**	**$a1**	**$s1**
$t1	**$sp**	**$v0**	
$t2	**$ra**	**$v1**	

- The following instructions are implemented(In Opcode Order) :

sub	**andi**
lw	**sw**
bneq	**nor**
beq	**addi**
j, jal(Shared opcode)	**or**
ori	**srl**
add	**sll**
subi	**and,jr(Shared opcode)**

Instruction Type	BIT15 to BIT12	BIT11 to BIT8	BIT7 to BIT4	BIT3 to BIT0
R-type	Opcode	SrcReg1	SrcReg2	DstReg
S-type			DstReg	ShAmt
I-type(memory)			MemReg	Offset
I-type(branch)			SrcReg2	
I-type(AL)			DstReg	Immediate
J-type		Target Jump Address		Reusable

Table 1: Instruction Set Structure for various types

3 Instruction Sets

3.1 Opcode Specific Instructions(Opcode Ordered)

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

Table 2: Opcode specific Instruction Set

3.2 Set Specific Instructions(Opcode Shared)

Instruction Name	**BIT15 to BIT12**	**BIT11 to BIT8**	**BIT7 to BIT4**	**BIT3 to BIT0**
jal	4	Jump Address		Self Address
jr	F	F	F	7(recognizer for **$ra**)

Table 3: Set Specific Instruction Set

4 Register Set

Register	**Recognizer**	**Register Type**
$zero	0000	Zero
$t0	0001	Temporary
$t1	0010	
$t2	0011	
$t3	0100	
$t4	0101	
$sp	0110	Stack Pointer
$ra	0111	Return Address
$a0	1000	Argument
$a1	1001	
$v0	1010	Return
$v1	1011	
$s0	1100	Saved
$s1	1101	

Table 4: Register Set

5 Components in implementation of Project

- ➢ An assembler code written in c++ to convert user provided assembly code to machine language (*MIPS instruction Set*)
- ➢ A simulator (*logisim-win-2.7.1*)
- ➢ 5 Atmega for hardware implementation and their codes
- ➢ D- flipflops, Push switches, Logic and Arithmetic IC's, Jumper wires and Professional wires

6 Block and DataPath Analysis

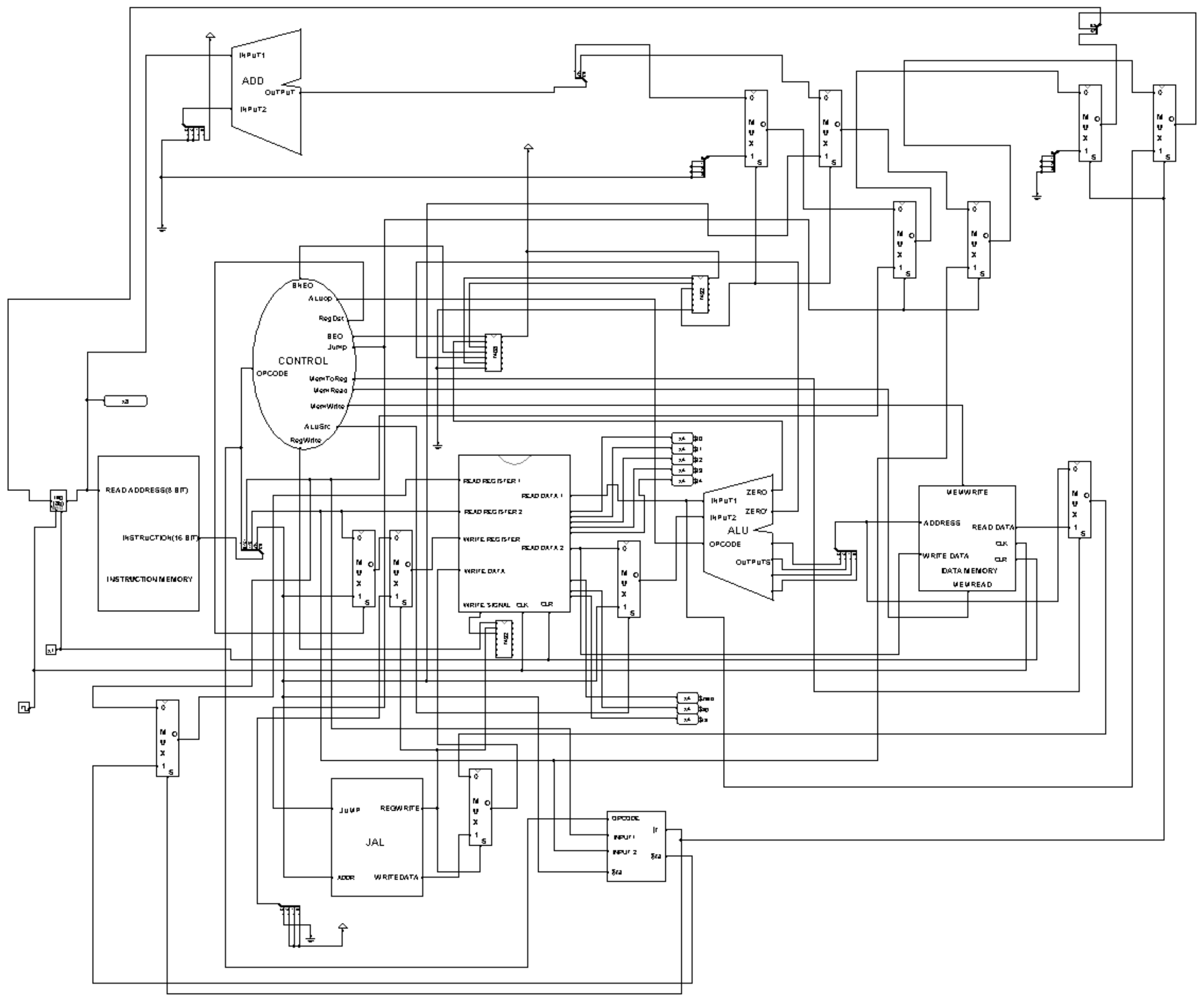


Figure 1: **Overall DataPath**

6.1 Instruction Memory

In the instruction memory, we have an *EEPROM* that takes a **8 bit address** as input and **16 bit data** as output. Here the **8 bit address** is the **Program Counter**, while each **16 bit data** denotes an instruction set.

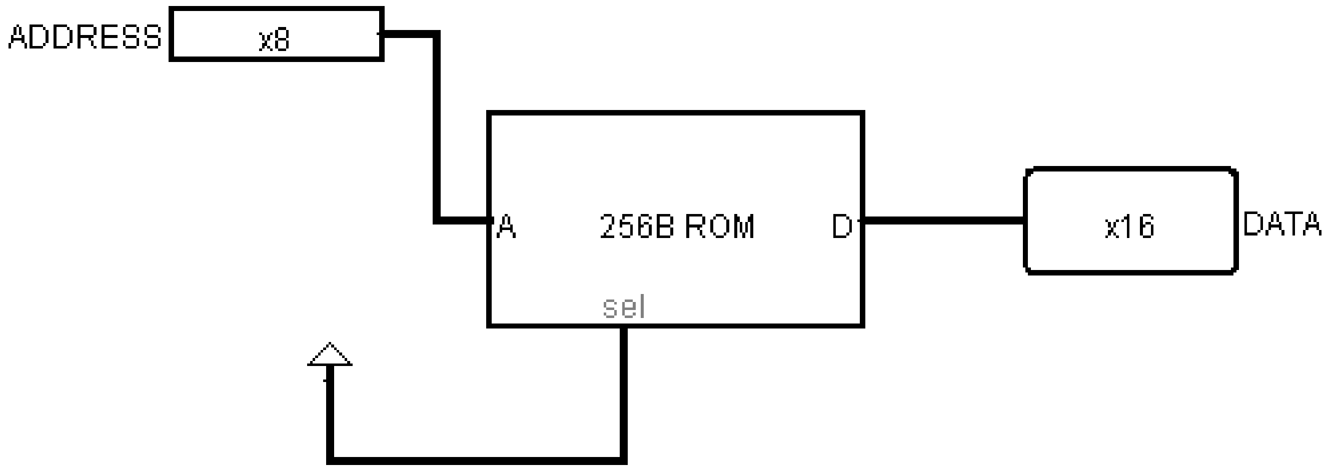


Figure 2: **Instruction Memory**

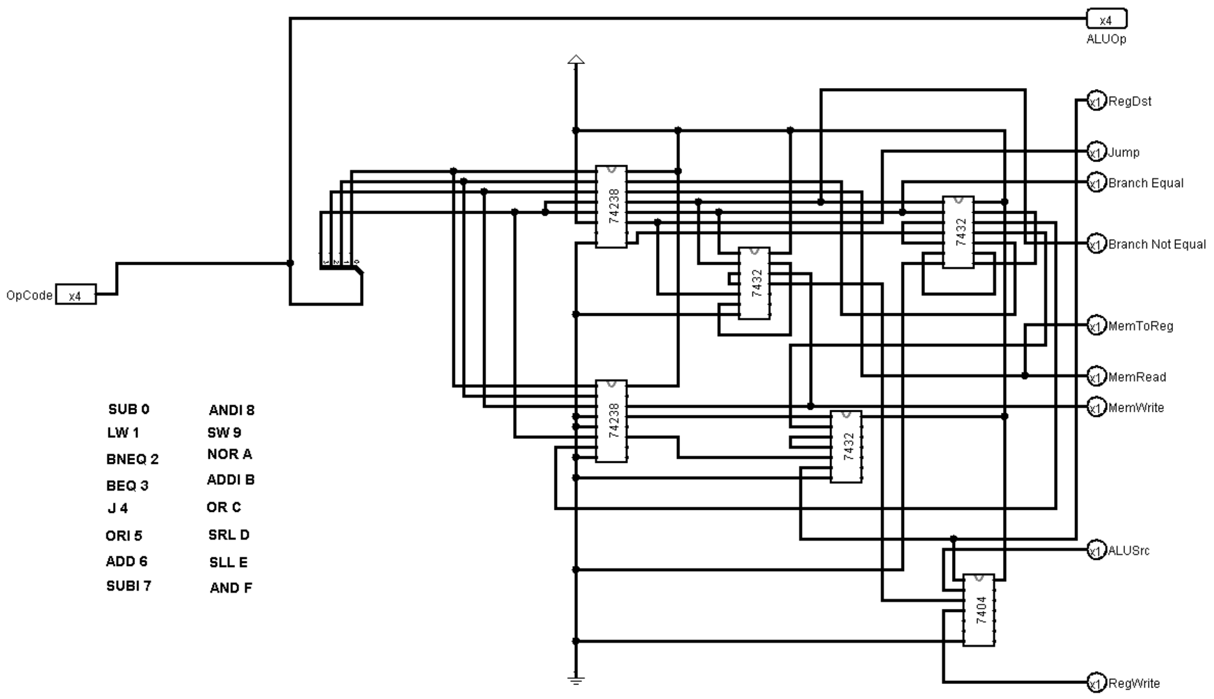
6.2 Control Unit

6.2.1 Opcode Dependent Control

In the Control Unit, we have a *decoder* that takes the **4 bit Opcode** as input and **1 bit control** as output. Here the **4 bit Opcode** passes to the **ALU**.The opcode is decoded and then by **OR** logic, the Controls are generated. The functions of the Controls will be discussed in section **9**.

Instruction	**Opcode**	**RegDst**	**BranchEqual**	**BranchnotEqual**	**Jump**	**MemtoReg**	**MemRead**	**MemWrite**	**ALUSrc**	**RegWrite**
sub	0000	1	0	0	0	0	0	0	0	1
lw	0001	0	0	0	0	1	1	0	1	1
bneq	0010	1	0	1	0	0	0	0	X	0
beq	0011	1	1	0	0	0	0	0	X	0
j	0100	X	X	X	1	0	0	0	X	0
ori	0101	0	0	0	0	0	0	0	1	1
add	0110	1	0	0	0	0	0	0	0	1
subi	0111	0	0	0	0	0	0	0	1	1
andi	1000	0	0	0	0	0	0	0	1	1
sw	1001	0	0	0	0	0	0	1	1	0
nor	1010	1	0	0	0	0	0	0	0	1
addi	1011	0	0	0	0	0	0	0	1	1
or	1100	1	0	0	0	0	0	0	0	1
srl	1101	0	0	0	0	0	0	0	1	1
sll	1110	0	0	0	0	0	0	0	1	1
and	1111	1	0	0	0	0	0	0	0	1

Table 5: Control(Opcode Ordered)



SUB 0	ANDI 8
LW 1	SW 9
BNEQ 2	NOR A
BEQ 3	ADDI B
J 4	OR C
ORI 5	SRL D
ADD 6	SLL E
SUBI 7	AND F

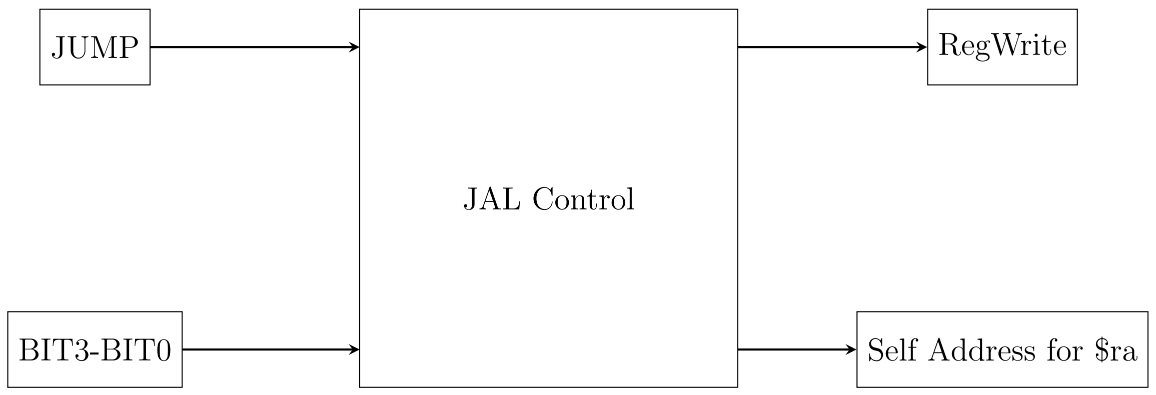
Figure 3: **Control**

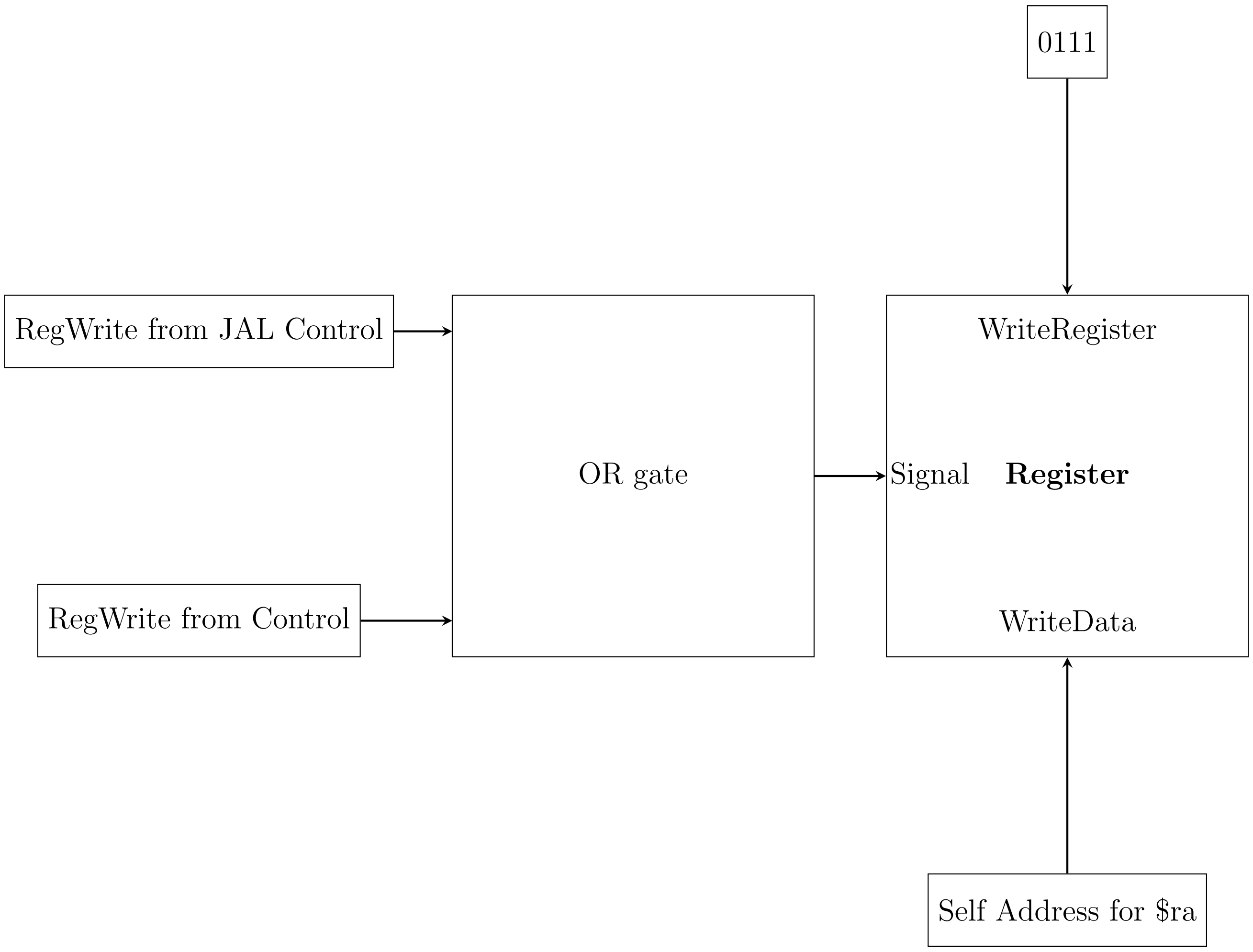
6.2.2 Additional Controls

- **JAL_Control :** This control is dependent on the Jump control and the value of **$ra** register.

Jump	**BIT3-BIT0 of Instruction set**	**RegWrite**	**Data to be put in $ra**
0	$(X)_H$	0	0
1	$(0)_H$	0	0
1	Y_H(can be anything but 0)	1	Y_H

Table 6: JAL Control Truth Table





Writing Register $ra by jal

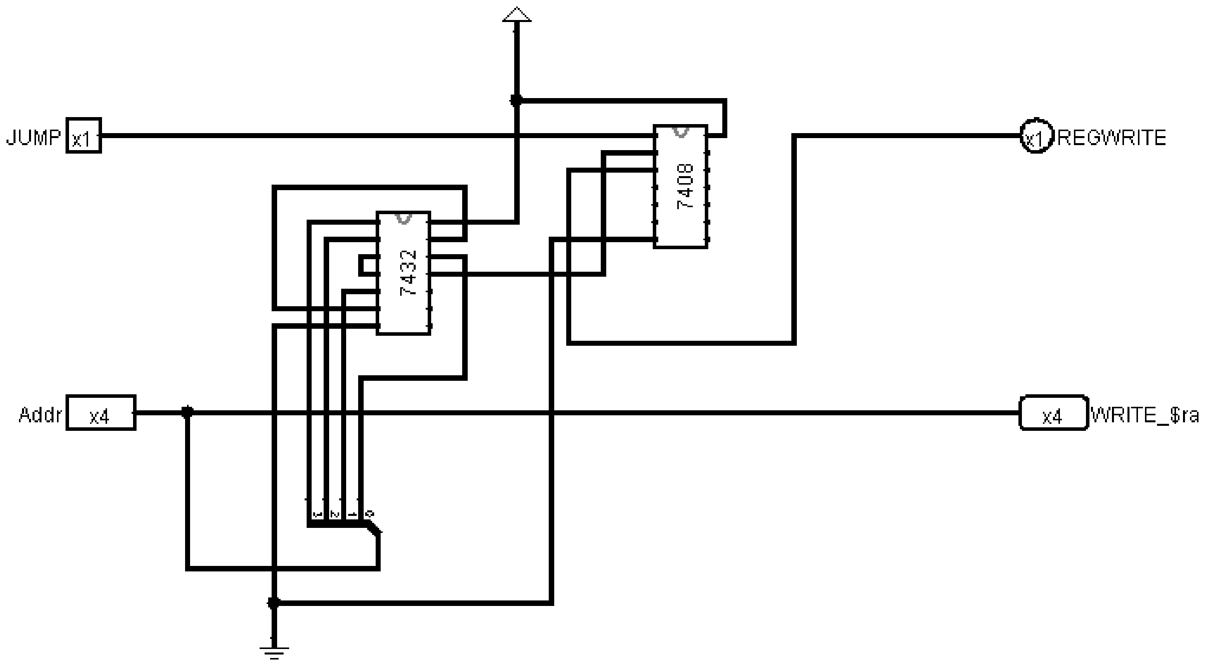


Figure 4: **Jump And Link Control**

- **JR_Control :** This control will take 16 bit data as input and give JR flag as output. Apparently, it will be activated only when it gets FFF7 as input. Only when it gets FFF, JR becomes 1. JR activates Jump by doing OR to the jump flag. And it takes the value in register7(**$ra**) to jump at that address

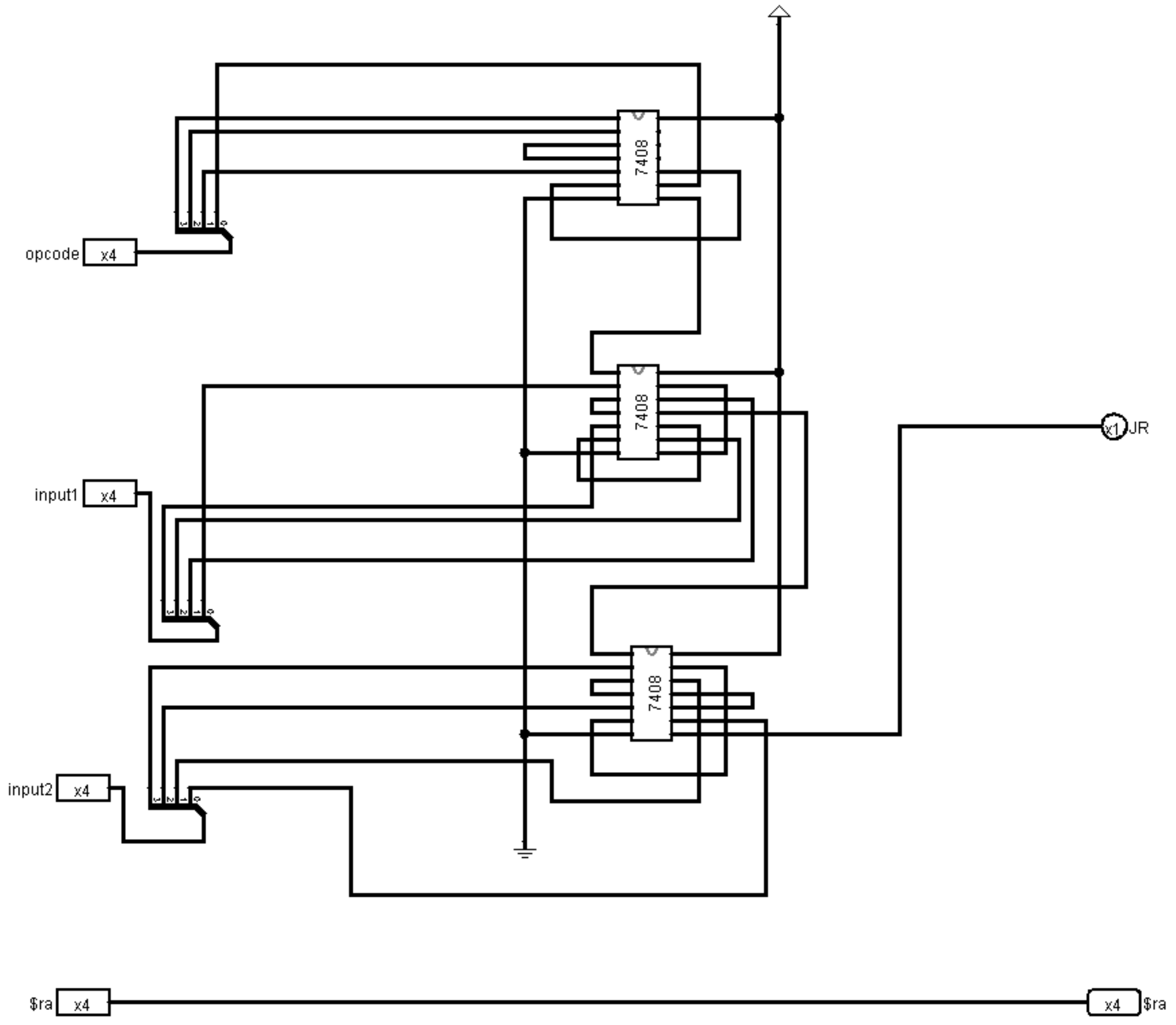


Figure 5: **Jump Return Control**

6.3 Register File

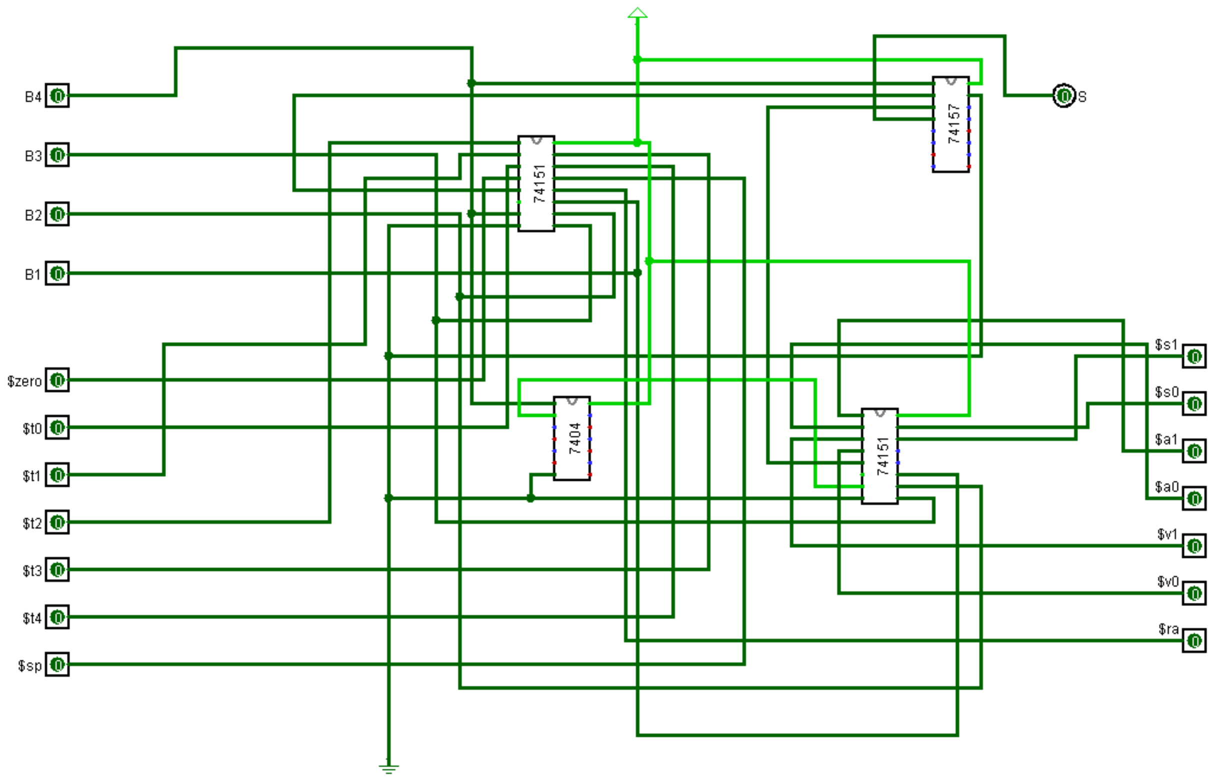


Figure 6: **Read Selector**

In register file, 2 registers of 4bit can be read at a time, while only one register can be written.

- Whether to write in a register is controled by a write signal. If an instruction intends to write some data into the register, then the **regWrite** for that control signal will be 1.
- To determine whether the BIT7-BIT4 will be the register to write(S or I type) or the BIT3-BIT0 are the destination registers(R type), this decision is made by **RegDst** .

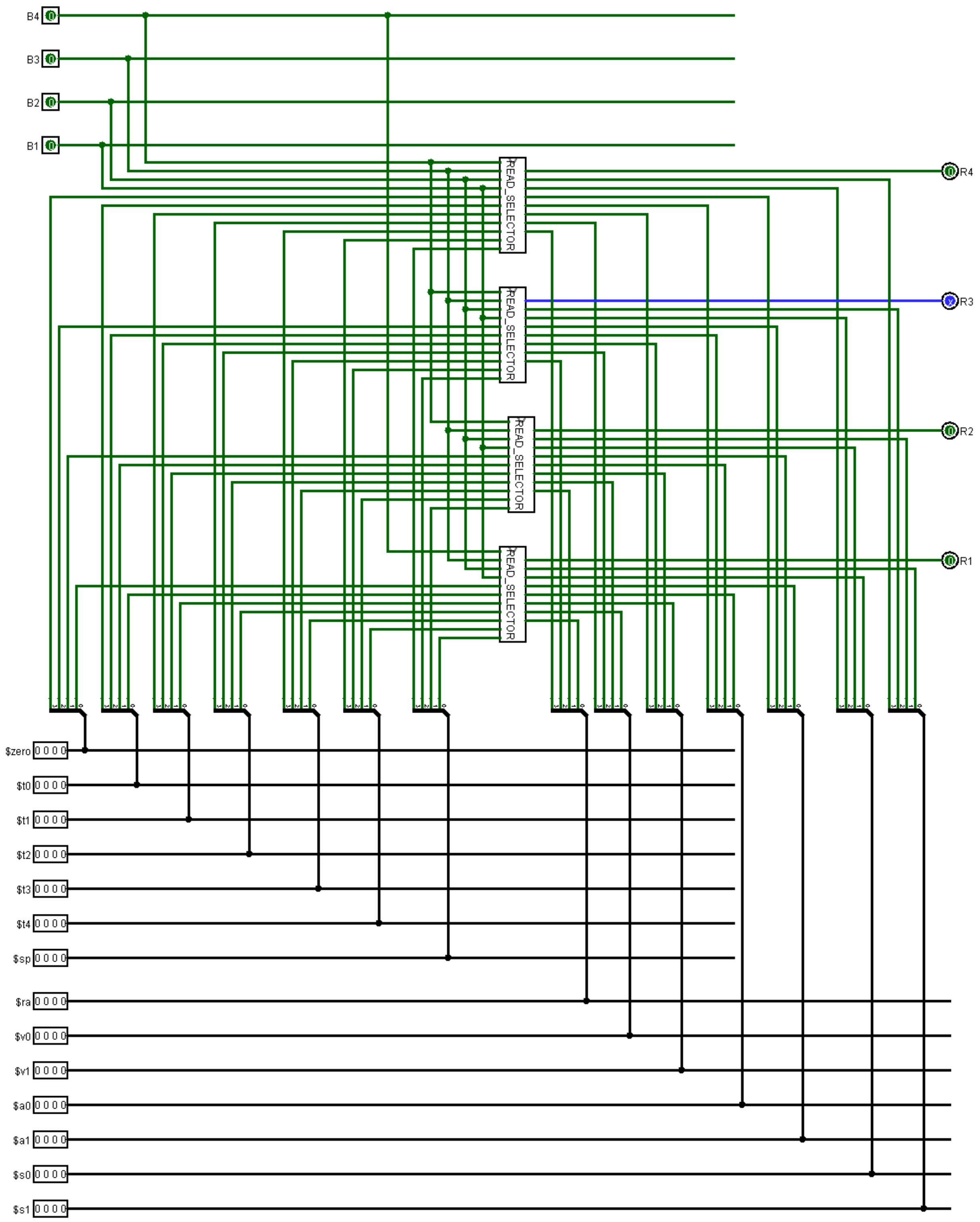


Figure 7: **Reader**

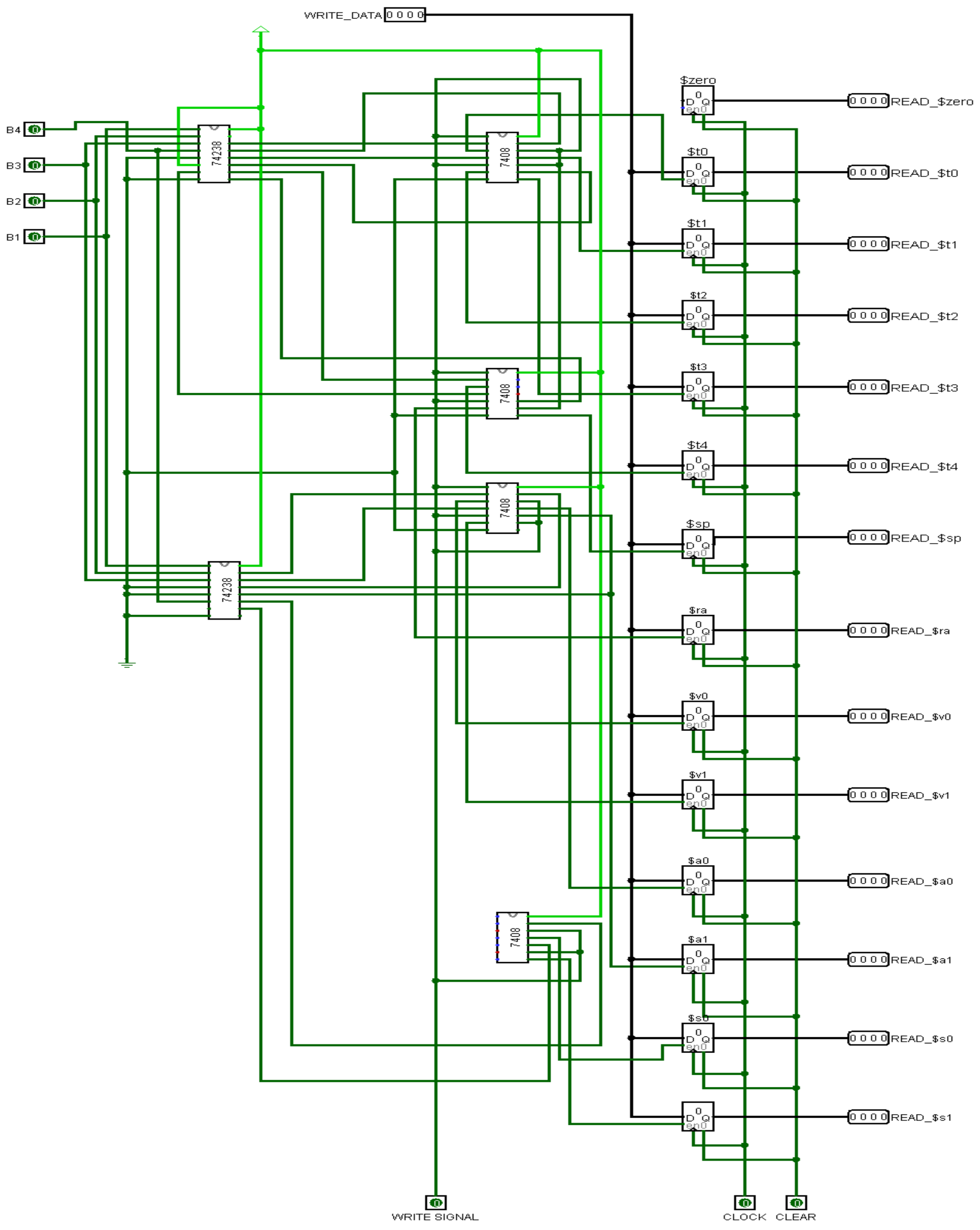


Figure 8: **Writer**

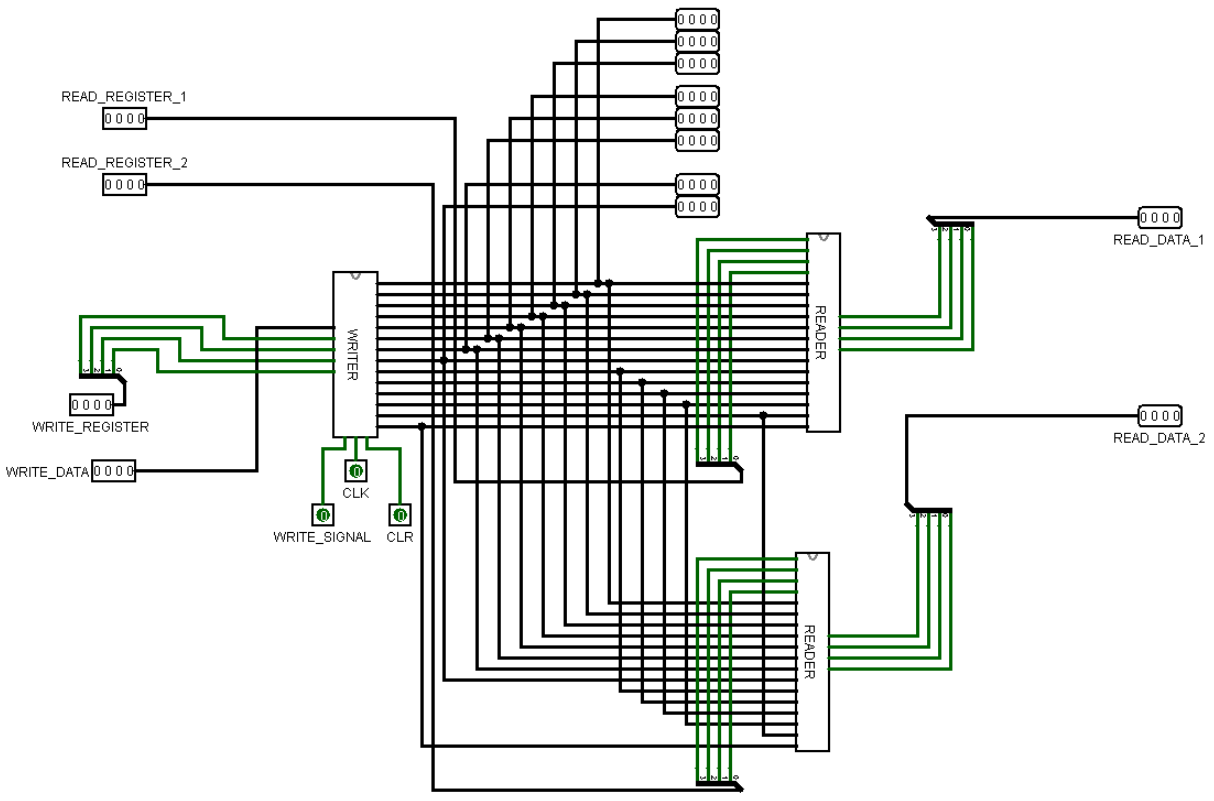


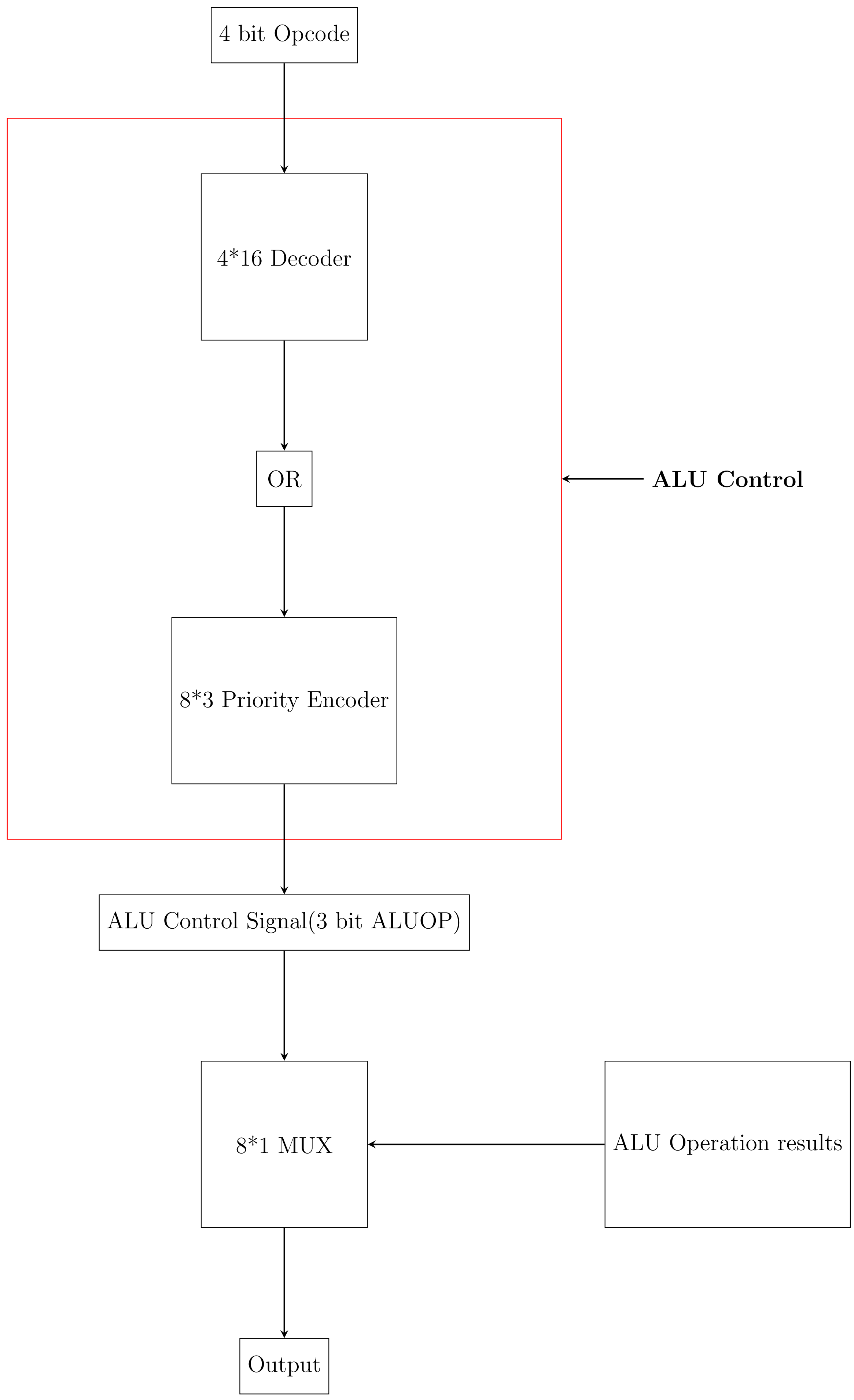
Figure 9: **Register Circuit**

6.4 ALU

6.4.1 ALU Control

In this block, the ALUOp is taken out from ALU through a decoder and the output of the decoder is passed to a priority encoder, so that the ALUop is converted to a 3bit MUX Control. The operation we choose from the mux will be output of the ALU.

The ALU implements 7 operations in total. Addition, Subtraction, Logical AND, Logical OR, Logical NOR, Shift Logical Left, Shift Logical Right.


4 bit Opcode
4*16 Decoder
OR
8*3 Priority Encoder
ALU Control
ALU Control Signal(3 bit ALUOP)
8*1 MUX
ALU Operation results
Output

Instructions with Opcode	ALUOP
lw(1)	000
add(6)	
sw(9)	
addi(B)	
sub(0)	001
bneq(2)	
beq(3)	
subi(7)	
andi(8)	010
and(F)	
ori(5)	011
or(C)	
nor(A)	100
srl(D)	101
sll(E)	110

Table 7: Caption

6.4.2 ALU Operation

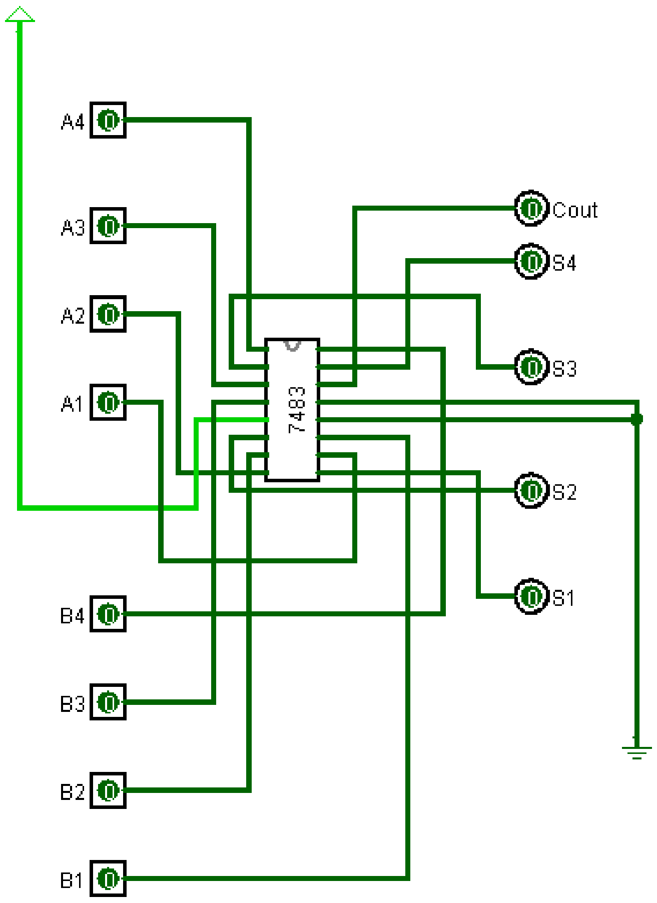


Figure 10: **4 bit Adder**

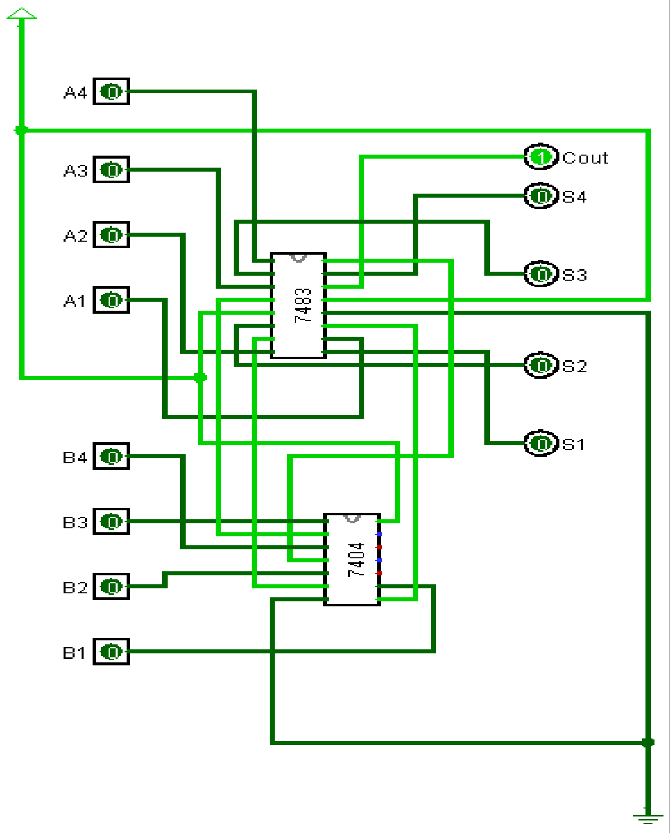


Figure 11: **4 bit Subtractor**

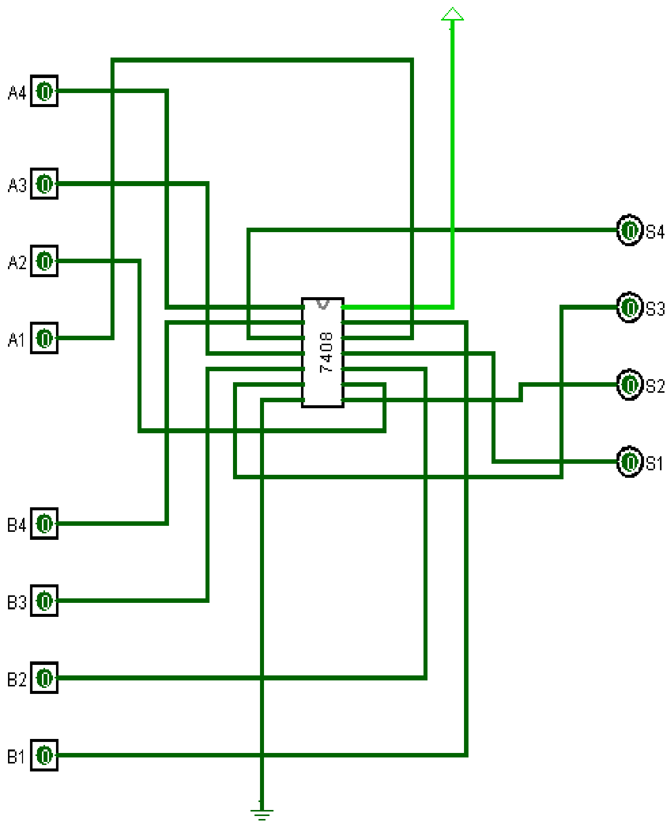


Figure 12: **4 bit AND**

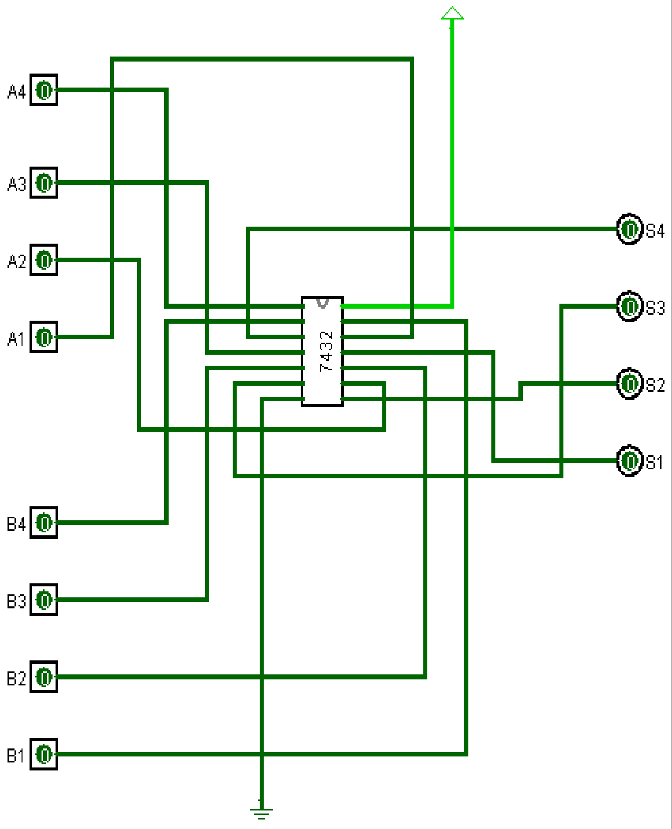


Figure 13: **4 bit OR**

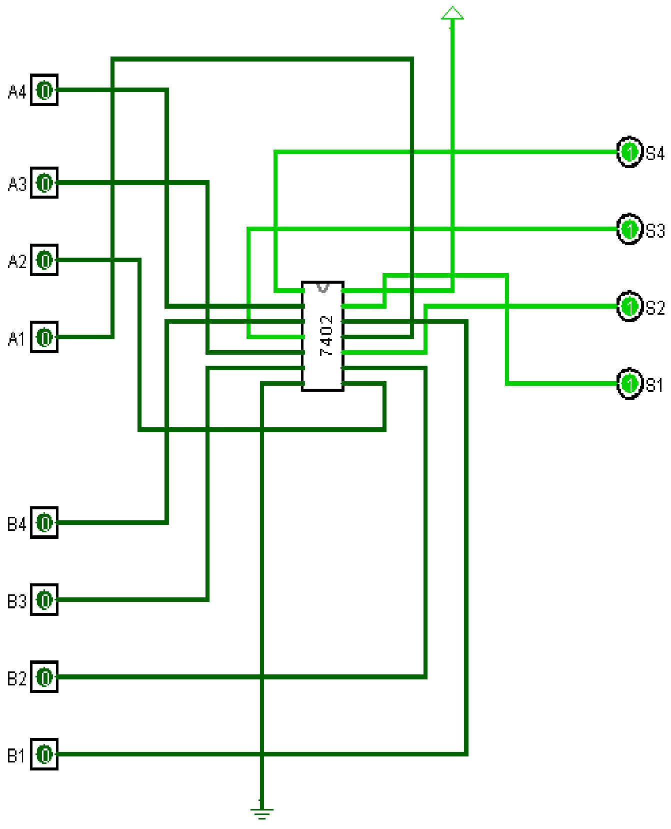


Figure 14: **4 bit NOR**

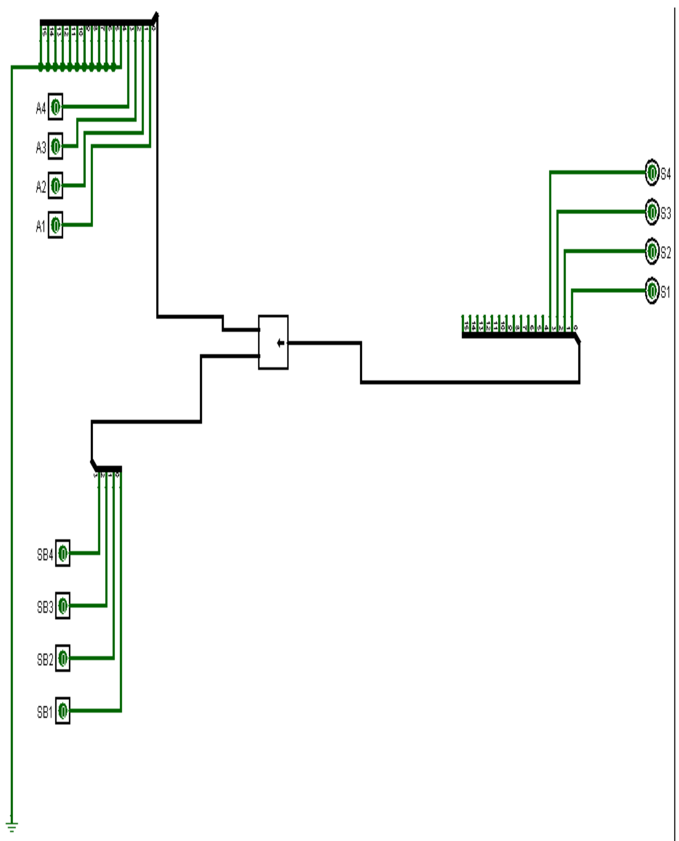


Figure 15: **4 bit left Shifter**

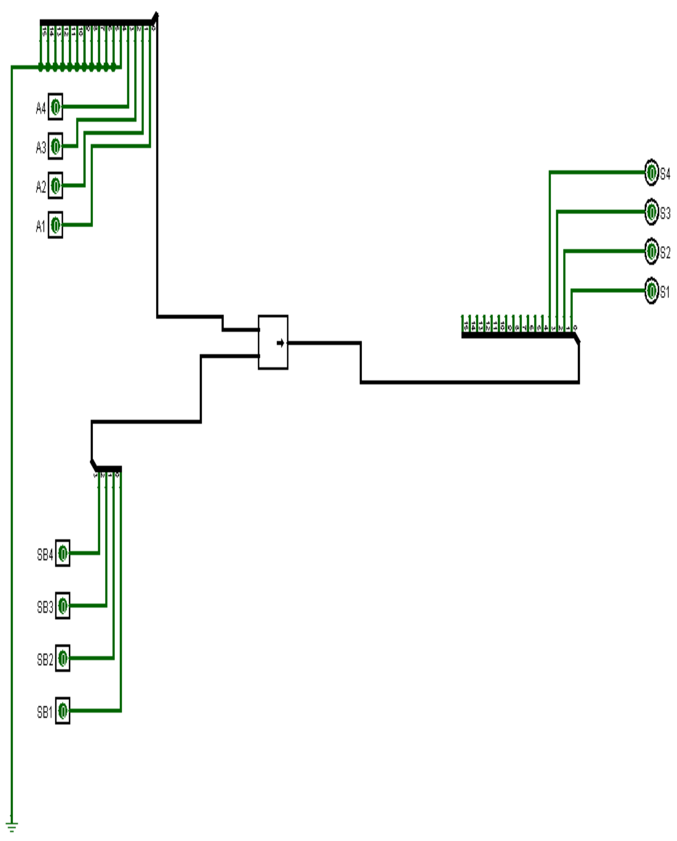


Figure 16: **4 bit right Shifter**

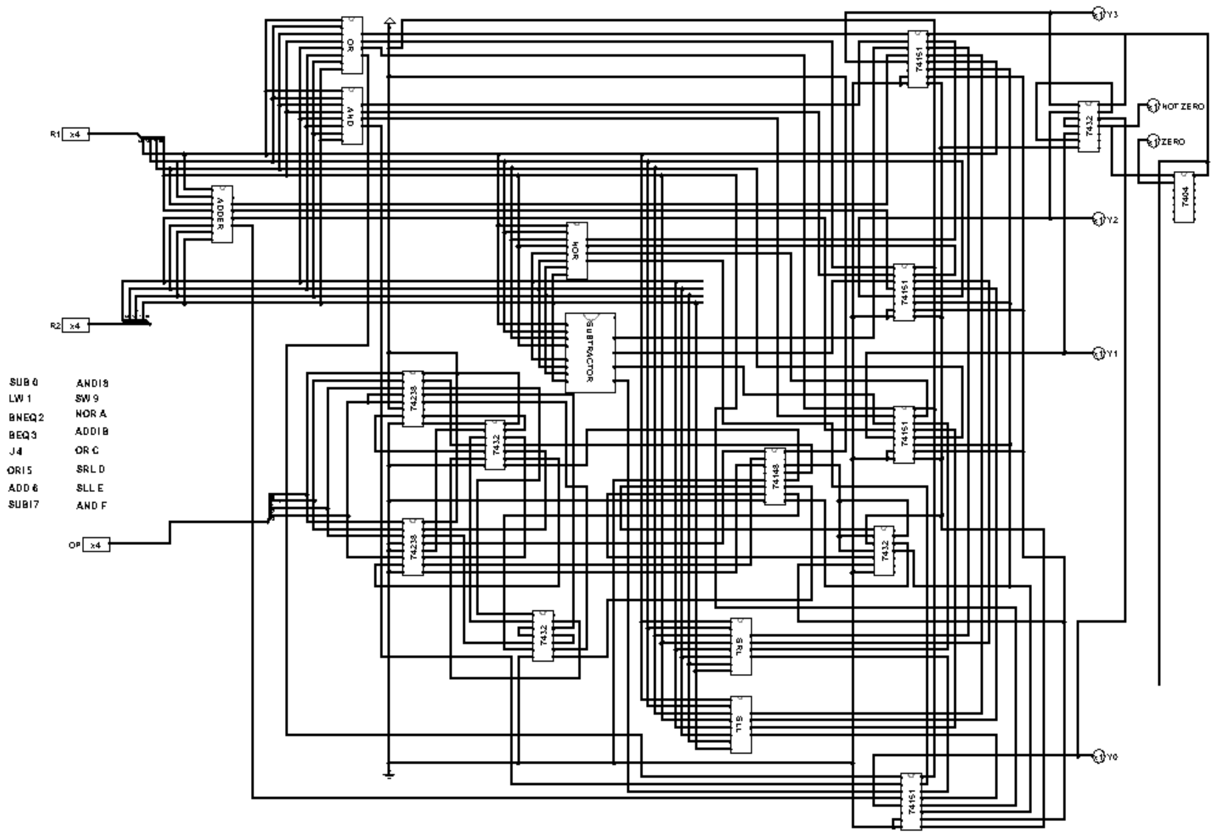


Figure 17: **4 bit ALU**

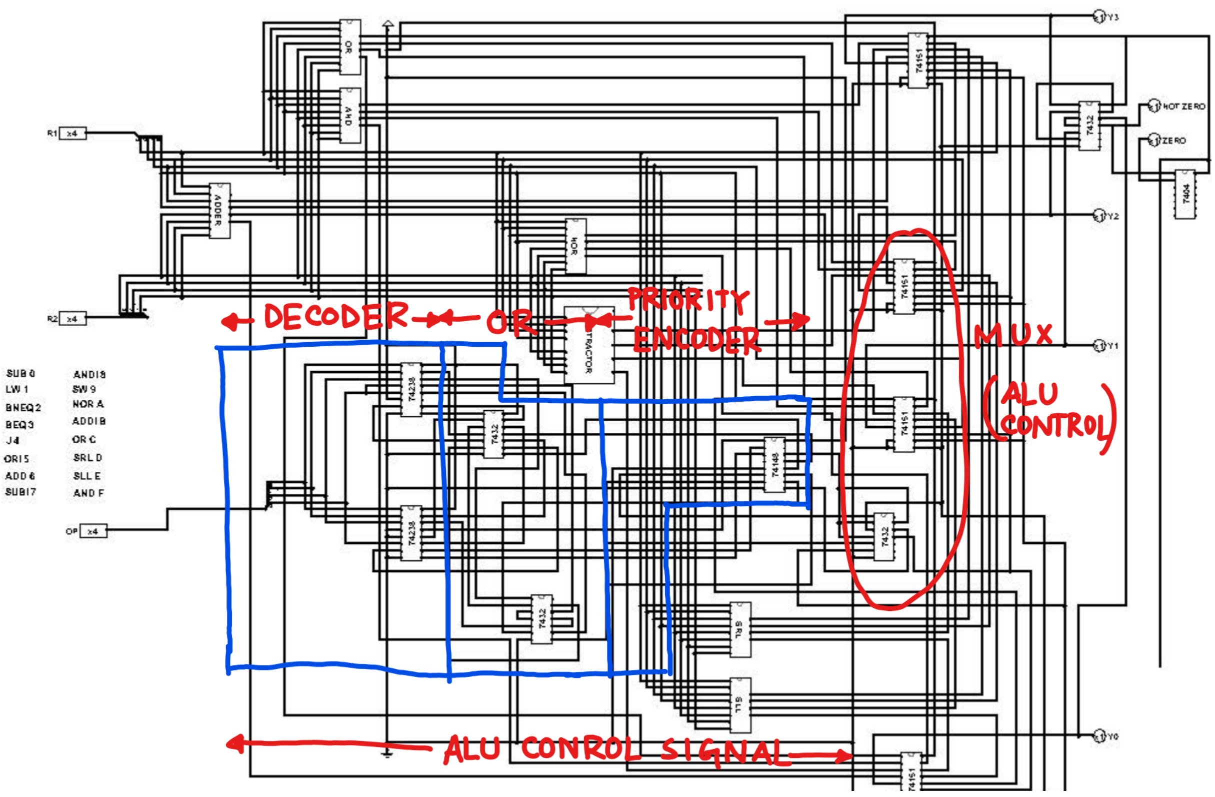


Figure 18: **ALU Control Marked**

6.5 Data Memory

In the Data memory, we have a *RAM* that takes a **4 bit address** as input and **4 bit data** as output. Here the **4 bit address** is the **Memory address** in which we want to store our data, either by *store word* or by *stack*, while each **4 bit data** denotes the **data kept in the input address**.

- Whether to read an **address data** from the data memory to load on register is activated by control **MemRead**
- Whether to transfer an **address data** from the data memory to load on register is activated by control **MemtoReg**
- Whether to write a **data** to a certain **address** of the data memory to store is activated by control **MemWrite**

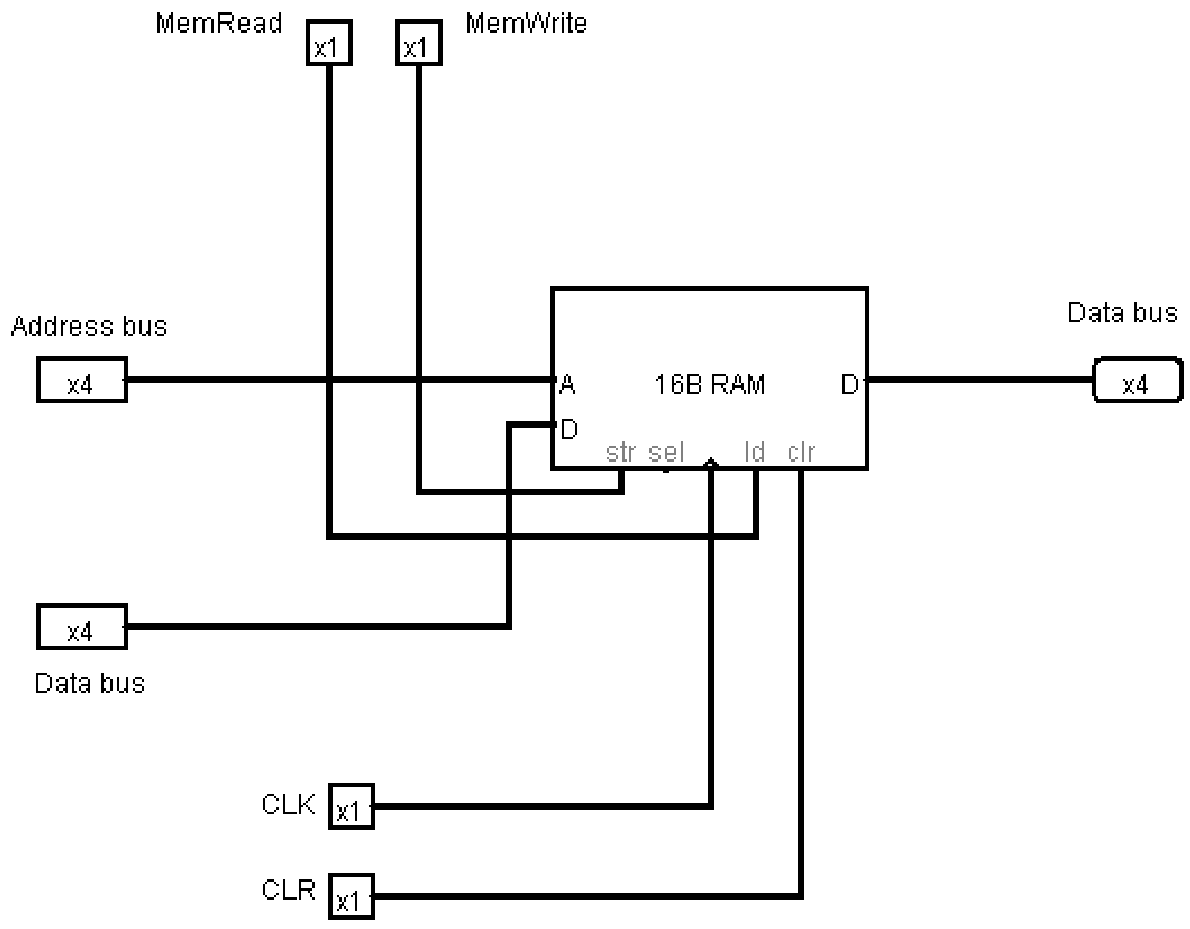


Figure 19: **Data Memory**

6.6 Branching and Program Counter

For a general instruction, the program counter should increase by 1 after execution, unless we are up for a jump.
Branching(beq,bneq) are called conditional jumps and general Jumps (j,jal,jr) are unconditional.
In case of branching or Jumping, the program counter shall proceed to a label that has been called for jump. Else the program counter will be incremented by 1 for each instruction.

7 How to write and execute a program in this machine

Step 1: At first we have to run the user programmed MIPS code (which is to be preserved in a text file input.txt) and then parsed through the assembler. This will generate a hex code of a set of mips instruction in some file output.txt)

Step 2: In case of software simulation the output.txt code will be loaded to the instruction memory(*EEPROM*) while in case of hardware implementation, the output hex file has to be passed as a group of hex strings to the *ATMEGA32* for instruction memory. The *EEPROM* of *ATMEGA32* is implemented in this case

Step 3: At each instruction the clock pulse has to be provided in both cases. Each clock pulse will either forward the registers to next instruction or branch it back or jump to another instruction based on the instruction provided to it.

Step 4: All the register values can be shown in software simulation after an instruction is executed, but only output register can shown in hardware implementation. The Program counter can be shown in both instructions.

8 Special Features Implemented

8.1 User programmable Push Pop in stack :

We have added the stack pointer in our software (**$sp**). It will rise from the bottom of Data Memory whenever user will push a data, and remove data from top whenever popped. The command has to be given in the following format.

```
addi $sp,$sp,-1 #allocating a space for stack pointer for pushing the data
sw $t0,0($sp) #pushing one data (data in $t0)in our stack pointer
lw $t1,0($sp) #popping the data from our stack pointer and assign it to $t2
addi $sp,$sp,1 #deallocating the allocated segment
```

8.2 User programmable function writing :

We have also implemented the procedure calling in our software implementation. Instructions **jal**(Jump and Link) and jr(Jump return) has been implemented.
Any jal command within 16 instructions work.
Additionally, for the purpose to serve the procedure calling additional registers are implemented. Those are **$ra,$ao,$a1,$v0,$v1,$s0,$s1**. The command has to be given in the following format.

C code:

```
int leaf(int a, int b){

    return a+b;
}

int main{

    int x=5,y=6;
    int z = leaf(x,y);

}
```

Eqivalent MIPS code:

```
addi $a0,$zero,5
addi $a1,$zero,6
jal leaf
add $t0,$s0,$zero

leaf:
add $v0,$a0,$a1
add $s0,$v0,$zero
jr $ra
```

9 Function of Different Control Signals

9.1 RegDst :

RegDst controls a mux to select from either BIT3-BIT0 or BIT7-BIT4 to feed to Writing Register of the Instruction Memory.

9.2 Branch Equal :

Branch Equal undergoes an AND operation with ALU Zero Flag to assertain whether the Branching is satisfied on Equal values or not. Based on the results it may execute a conditional Jump.

9.3 Branch not Equal :

Branch Equal undergoes an AND operation with ALU NotZero Flag to assertain whether the Branching is satisfied on inequal values or not. Based on the results it may execute a conditional Jump.

9.4 Jump :

Jumping on a label is unconditional. Jump activates the program counter to proceed to a label unconditionally.

9.5 MemRead :

While loading a data to register from a certain addres in Data Memory, memory has to be read. MemRead activates the Data Memory to enable data reading

9.6 MemWrite :

While storing a data from a register to a certain address in data memory, memory has to be written. MemWrite enables the data memory for writing data to a specified address.

9.7 MemtoReg :

Only reading memory is not enough to write a value obtained from memory to register, for which, MemToReg will activate a mux to whether write the value to the register or not.

9.8 ALUSrc :

ALUSrc activates a mux that will select either of register data or immediate data to be passed to ALU. I-type instruction have an ALUSrc 1, to select immediate value for writing data. Or it may be 0, which will redirect it to select the value passing from the Register.

9.9 RegWrite :

RegWrite enables the write signal for the register. RegWrite from Control unit or RegWrite from jal activates the write signal. If jal activates write signal, data will be written specifically to $ra.

9.10 JR :

JR activates a mux to chose whether the jump is a branch jump, or unconditional jump or the return jump. Branch Jump passes through the first mux, Unconditional Jump passes throgh the second mux with the output of Branch Mux. While the output goes to another mux with the jump return.

10 IC's with their Count:

10.1 Software

- **IC 7402: Quad 2input NOR :** 1
- **IC 7404: Hex 1input NOT :** 4
- **IC 7408: Quad 2input OR :** 8
- **IC 7432: Quad 2input AND :** 12
- **IC 7483: 4 bit adder :** 4
- **IC 74148: 8*3 Priority Encoder :** 1
- **IC 74151: 8*1 MUX :** 20
- **IC 74157: Quad 2*1 MUX :** 20
- **IC 74238: 3*8 Decoder :** 6
- **IC 74273: Register :** 16
- **EEPROM :** 1
- **RAM :** 1
- **Left Shifter :** 1
- **Right Shifter :** 1
- **Clock :** 1

10.2 Hardware

- **IC 7404: Hex 1input NOT :** 1
- **IC 7408: Quad 2input OR :** 1
- **IC 7432: Quad 2input AND :** 2
- **IC 74273: Register :** 1
- **Push Switch :** 1
- **ATMEGA32 :** 5
- **Clock :** 1

11 Discussion

In case of hardware implementation, all the equipments were tested individually and then we tried to combine them. Unfortunately the D flip flop was not being able to generate proper PC even though we used falling edge configuration to avoid debouncing. One possible reason is the given clock was not being able to sync with the clocks of Atmega32. Due to this reason, register values were not stored and only immediate or R/I type instructions were performed successfully.

All the wires and ICs were attached carefully. Wires were given connection after thorough cross checking. Still there remained some errors and our project was not fully successful.

12 References

1. **All images in full resolution :** IMAGES
2. **Assembler code:** ASSEMBLER