



**Department of Electrical and  
Computer Engineering**

**Computer Architecture - ENCS437**

**Project#2**

**Instructor: Aziz M. Qaroush**

**Names:**

**Tala Skafi-1161242**

**Zaid Khateeb-1151675**

**May 16, 2019**

## **Abstract:**

This project is to Implement the design we made for question one in Assignment#1(the modified Datapath) using logisim simulator, so we have to be familiar with this program, and test all the modified and added instruction .

# Table of contents:

Abstract: .....	2
<b>1 Design and Implementation .....</b>	<b>4</b>
1.1 The original single-cycle data-path.....	4
1.2 The Datapath design and implementation before adding the needed instructions:.....	5
1.2.1 32x32 Bit Register File:.....	7
1.2.2 Memories: .....	9
1.2.3The ALU component: .....	10
1.2.4 Main control Unit: .....	12
1.2.5 PC control Unit:.....	14
1.2.6 ALU Control Unit:.....	15
1.3 Starting modify: .....	20
<b>2 Simulation and Testing:.....</b>	<b>27</b>
1. Lws: .....	27
2.CAS.....	30
3.Llb .....	32
4.jal .....	34
5.jalr.....	36
<b>3 Teamwork:.....</b>	<b>38</b>
<b>Conclusion: .....</b>	<b>39</b>

# 1 Design and Implementation

## 1.1 The original single-cycle data-path

This in the original single-cycle Datapath, see figure 1.1

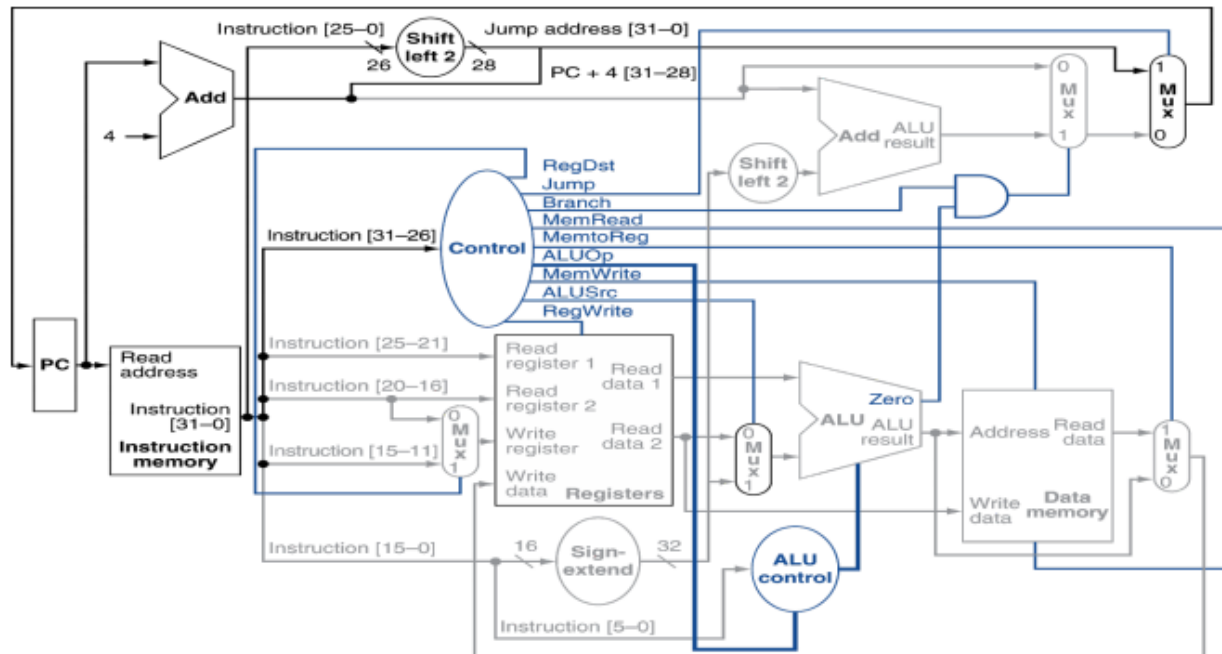


Figure 1.1 the original single cycle Datapath before modifying

## 1.2 The Datapath design and implementation before adding the needed instructions:

We designed the needed single-cycle data-path and implemented it using Logisim tool, and we will describe every component wildly, see fig1.2

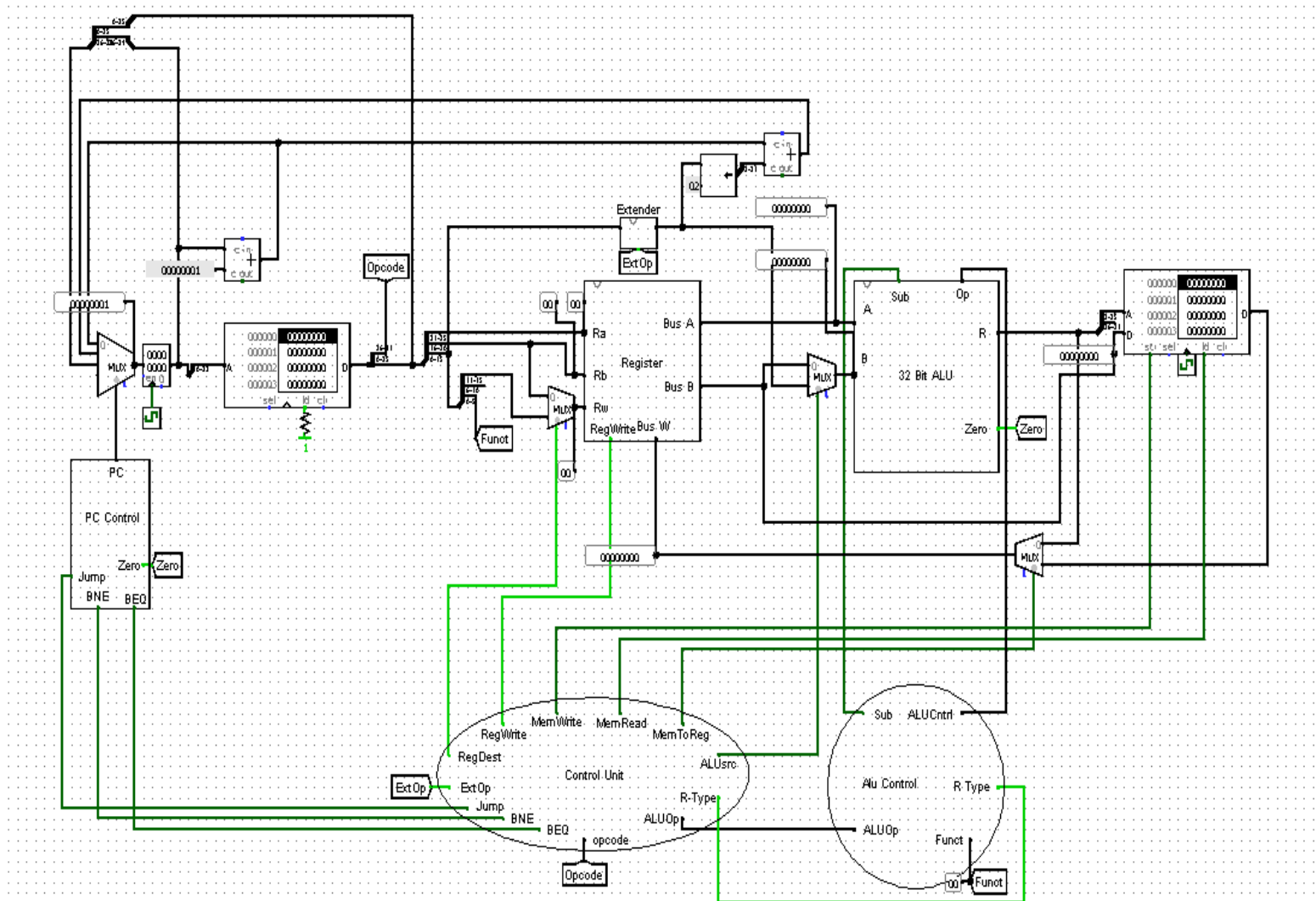


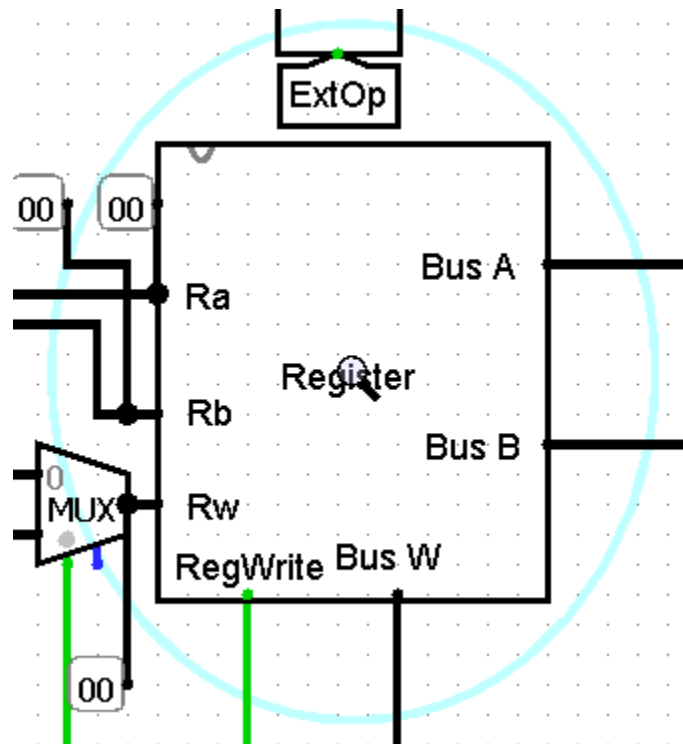
Figure1.2 designed Datapath before modifying

## **What we need:**

- ◆ Memory
  - Instruction memory where instructions are stored
  - Data memory where data is stored
- ◆ Registers
  - $32 \times 32$ -bit general purpose registers, \$0 is always zero
  - Read source register Rs
  - Read source register Rt
  - Write destination register Rt or Rd
- ◆ Program counter PC register and Adder to increment PC
- ◆ Sign and Zero extender for immediate constant
- ◆ ALU for executing instructions
- ◆ Multiplexers
  - To select the write destination, register Rt or Rd
  - To select the ALU operand inputs from register file or immediate operand
  - To update PC address with  $PC + 4$  or branch destination or jump destination.

### 1.2.1 32x32 Bit Register File:

Our register file is 32x32bit size, that most instructions read and write data from and to it. As you can see from the data-path, the register file has two Read Address ports and two Read Data ports. This means we need to be able to read two values at once. There is also a Write Address port, a Write Data port, and a Write Enable port. These are used for writing the result from completed instructions back into the register file, it must be able to read two values simultaneously, and asynchronously (no clock required for reading). And it Writes must happen only on the positive clock edge and only when write enable is asserted. And it Correctly handles reads and writes from register zero, see the implementation below in fig1.2.1 , and you can see it clearly from the attached file on Ritaj for the project.







## 1.2.2 Memories:

In our project, we used two memories component (defined as RAM component in Logisim see fig 1.2.2)

The first memory is for instructions where the instructions will be stored , the second memory is for Data, where data are stored , and both size are :Address bit width=24 , data bit width=32

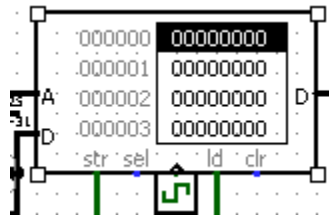
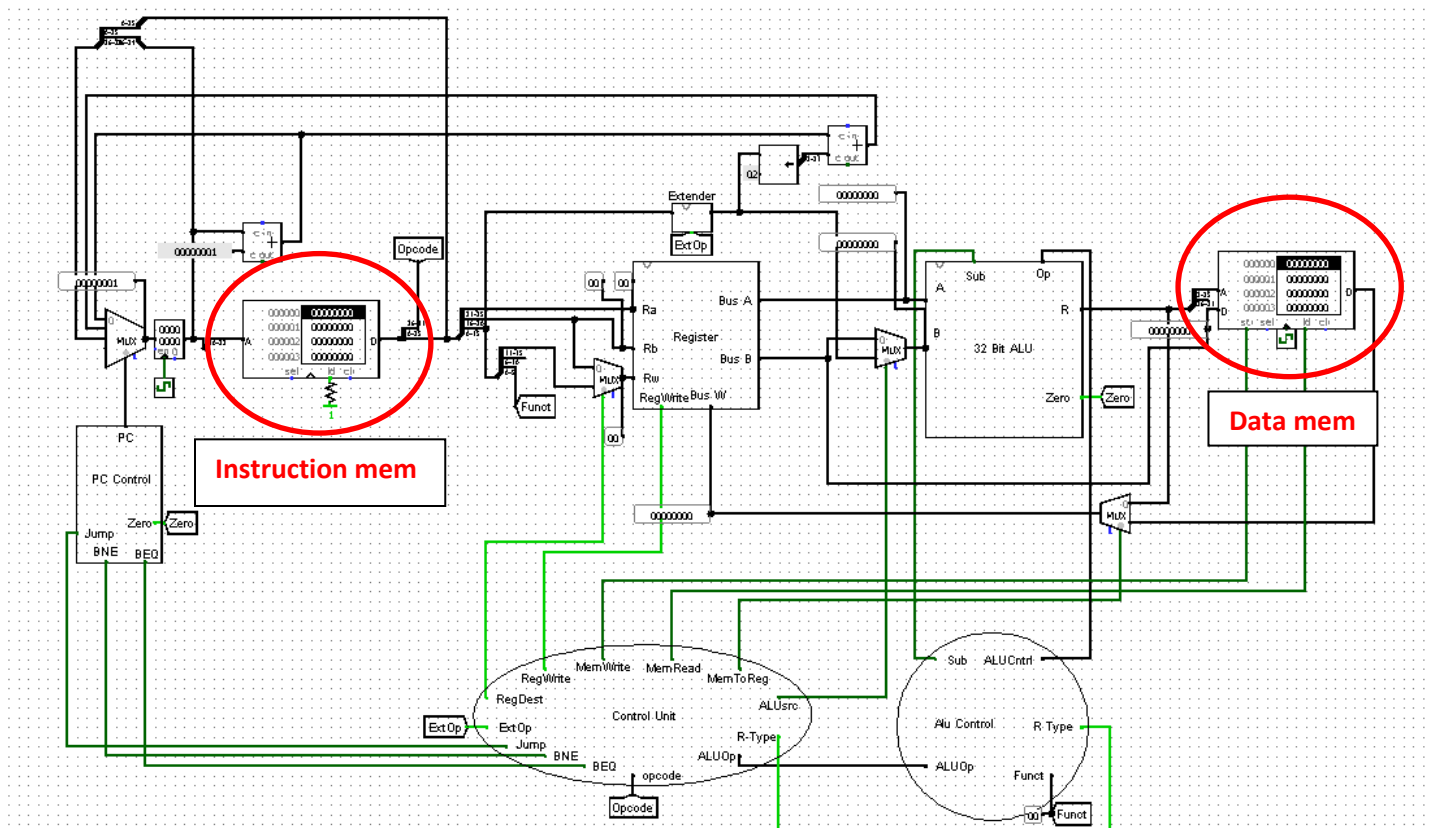


Figure 1.2.2 RAM component



### 1.2.3 The ALU component:

An important component we provide is an ALU. The 32 bit alu we used is responsible for doing all of the calculations in your processor.

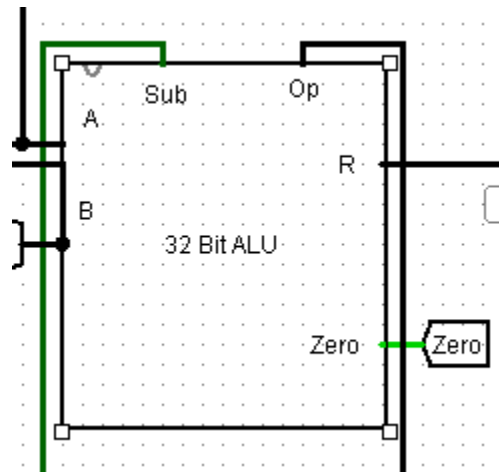


Figure 1.3.3 32bit ALU component

The ALU control receives a 6-bit function field from the instruction and the ALUOp signal from the main control. ALUOp indicates whether the operation to be performed should be ADD (100), SUB (101), AND (001) ... etc. or determined by the operation encoded in the Funct field. Similar to the procedure to generate the control signal Boolean functions for the main control unit, the ALU control signals equations can be derived based on the 6-bit function field and the ALUOp signal generated by the Main Control unit. This is the table for the ALU signals (see table 2.3.2)

<b>Op</b>	<b>Funct</b>	<b>ALU</b>	<b>4-bit ALU</b>
R-type	AND	AND	0001
R-type	OR	OR	0010
R-type	XOR	XOR	0011
R-type	ADD	ADD	0100
R-type	SUB	SUB	0101
R-type	SLT	SLT	0110
ADDI	X	ADD	0100
SLTI	X	SLT	0110
ANDI	X	AND	0001
ORI	X	OR	0010
XORI	X	XOR	0011
LW	X	ADD	0100
SW	X	ADD	0100
BEQ	X	SUB	0101
BNE	X	SUB	0101
J	X	X	X

Table 2.3.2 ALU signals

It's too large to put its' implementation here, so you can see 32-bit ALU from the attached file for our project.

## 1.2.4 Main control Unit:

The Main Control unit handles interpreting the opcodes from the current instruction and enabling all of the relevant signals so that the rest of the processor will work as expected as described by the MIPS spec (see fig 1.2.4). Some of the signals of the Main Control unit are also fed into the PC Control unit (Jump, BEQ, BNE) and the ALU unit (ALUSrc, ALUOp) see this connection in the design in fig 1.2 above.

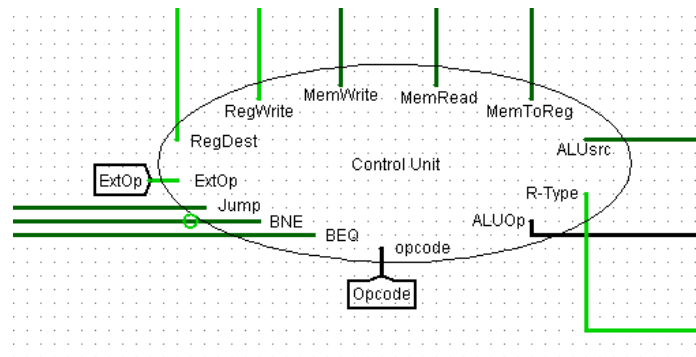


Figure 1.4.4 Main control unit

### The truth table :

OP	Reg dst	Reg write	Ext op	Alu src	beq	bne	j	Mem read	Mem write	Mem toreg
R-type	<u>1</u>	<u>1</u>	<u>x</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>
addi	<u>0</u>	<u>1</u>	<u>1</u>	<u>1</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>
slti	<u>0</u>	<u>1</u>	<u>1</u>	<u>1</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>
Andi	<u>0</u>	<u>1</u>	<u>0</u>	<u>1</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>
Ori	<u>0</u>	<u>1</u>	<u>0</u>	<u>1</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>
xori	<u>0</u>	<u>1</u>	<u>0</u>	<u>1</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>
lw	<u>0</u>	<u>1</u>	<u>1</u>	<u>1</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>1</u>	<u>0</u>	<u>1</u>
sw	<u>x</u>	<u>0</u>	<u>1</u>	<u>1</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>1</u>	<u>x</u>
Beq	<u>x</u>	<u>0</u>	<u>x</u>	<u>0</u>	<u>1</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>X</u>
bne	<u>x</u>	<u>0</u>	<u>x</u>	<u>0</u>	<u>0</u>	<u>1</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>x</u>
I	<u>X</u>	<u>0</u>	<u>x</u>	<u>x</u>	<u>0</u>	<u>0</u>	<u>1</u>	<u>0</u>	<u>0</u>	<u>x</u>

The Boolean functions for the control signals are listed as follows: (Note: in the following equations, the right hand represents the decoder output)

RegDst	=	R-type
RegWrite	=	(sw + beq + bne + j)'
ExtOp	=	(andi + ori + xori)'
ALUSrc	=	(R-type + beq + bne)'
MemRead	=	lw
MemWrite	=	sw
MemtoReg	=	lw

The control signals of Beq, Bne and J can be forwarded directly from the decoder outputs

**See the implementation of this unit:**

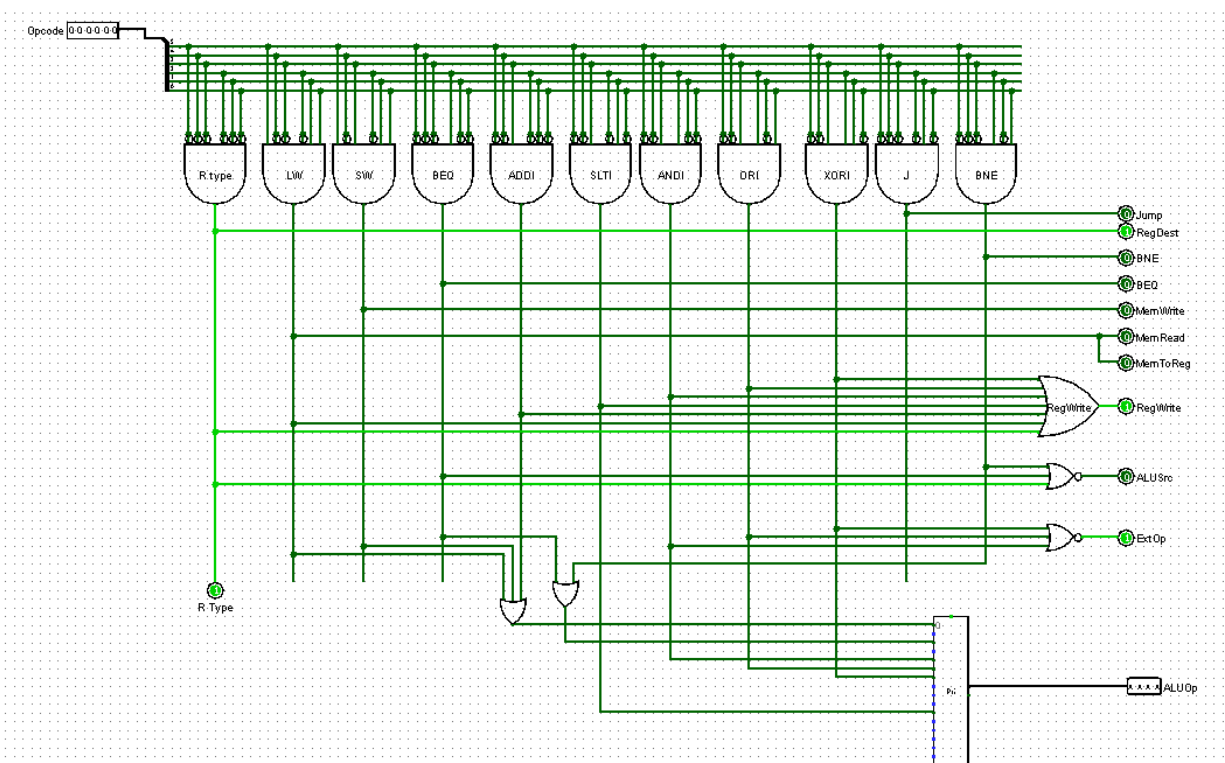


Figure 1.5.5 Main control unit implementation

## 1.2.5 PC control Unit:

The PC Control unit handles getting the next address that the PC register should save for the next cycle of the processor, which will in turn dictate which instruction in instruction memory that is extracted and fed into the remaining components of the processor. The PC Control unit takes the signals of Jump, BNE, and BEQ, as well as ALU's Zero signal and computes from it a selection value to activate a channel in its multiplexer to select whether the PC will increment, jump, or branch. see fig1.2.5

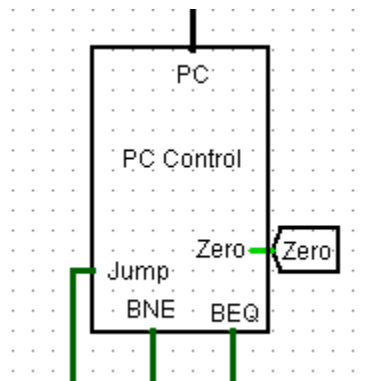


Figure 1.6.5 PC control unit

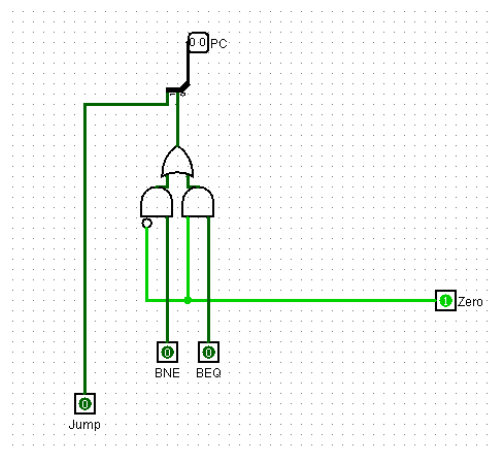


Figure 1.7.6 PC control unit implementation

## 1.2.6 ALU Control Unit:

The ALU Control unit takes the signals ALUSrc and ALUOp (alternatively if the instruction handles immediate mode instructions, it will retrieve the opcode from the func bits in the original instruction retrieved from instruction memory) from the Main control unit, and based on those signals, where A is always from a register, and B could be either from a register or directly from the immediate value embedded in the instruction as selected by ALUSrc, and then based on the ALUOp value the ALU Operation is then selected and then performed. Depending on the operation performed, there are additional signals that are created such as the ALU Zero signal. See fig1.2.6

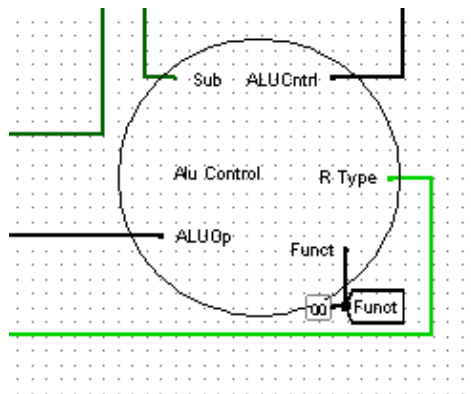


Figure 1.8.6 ALU control unit

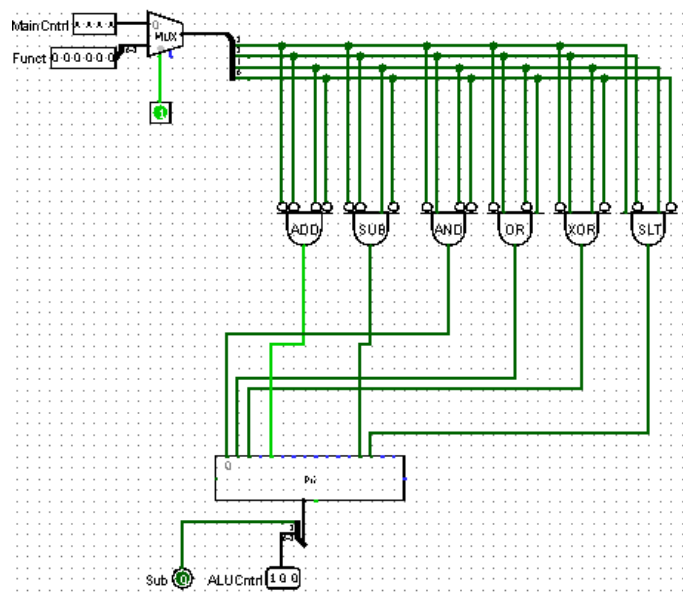


Figure 1.9.7 ALU control unit implementation

Then we put all of these components together as shown above in fig1.2 and connect all of them with Sign and zero extender Which will extend the immediate constant and make it 32 bit instead of 16 , as the original single-cycle data-path shown in fig 1.1

### **How to test the correctness of our single-cycle data-path design?**

We loaded a small program (called: The Sum Program) this program adds the numbers between A and B, inclusive, where A originally resides at address 4 in data memory and B resides in address 8 in data memory. The result is placed at data memory location 0 so the program will compute the sum of the numbers from 1 to 10 , (the result will be 55 in decimal , =37 in Hex)

(written in assembly to be understood):

```
sub r0,r0,r0    ; set reg[0] to 0, use as base
lw  r1,0(r0)    ; reg[1] <- mem[0] (= 1)
lw  r2,4(r0)    ; reg[2] <- mem[4] (= A)
lw  r3,8(r0)    ; reg[3] <- mem[8] (= B)
sub r4,r4,r4    ; reg[4] <- 0, running total
add r4,r2,r4    ; reg[4] += A
slt r5,r2,r3    ; reg[5] <- A < B
beq r5,r0,2     ; if reg[5] = FALSE, go forward 2 instructions
add r2,r1,r2    ; A++
beq r0,r0,-5    ; go back 5 instructions
sw  r4,0(r0)    ; mem[0] <- reg[4]
beq r0,r0,-1    ; program is over, keep looping back to here
```



this code of instructions will be converted to machine code then loaded to instruction memory:

v2.0 raw

00000022

8c010000

8c020004

8c030008

00842022

00822020

0043282a

10a00002

00411020

1000fffb

ac040000

1000ffff

And we must here fill the memory data specially at address 0 (value=1) and at address 4 (named A with value 1) and address 8 (B with value a ) :

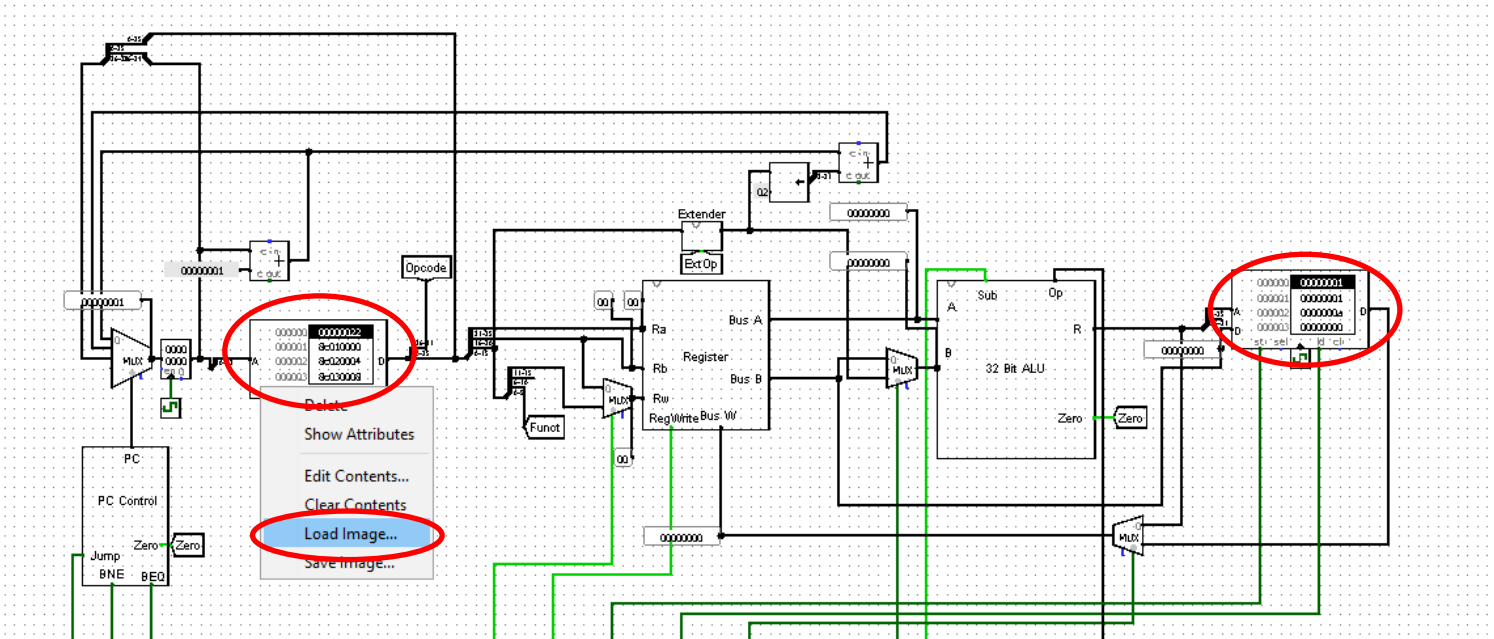
v2.0 raw

00000001

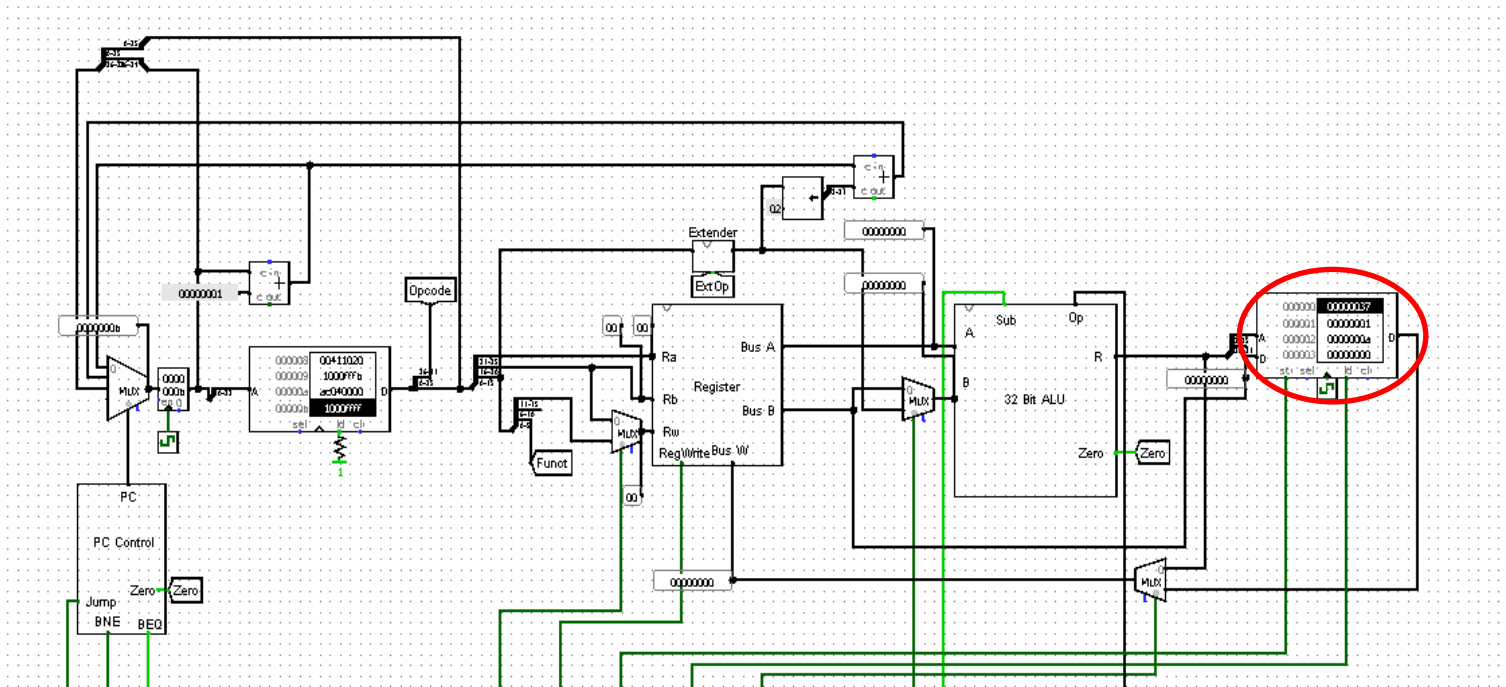
00000001

0000000a

See how we load them by select “load image” and chose the text file that contains the code written in machine language, note how the memories where filled with codes:



See how result “37hex” placed at location 0 in data memory!



### **Why we chose this simple program for testing?**

Simply, because it contains instruction that will use all the component, so now we can make sure that our design **is ready to modify**.

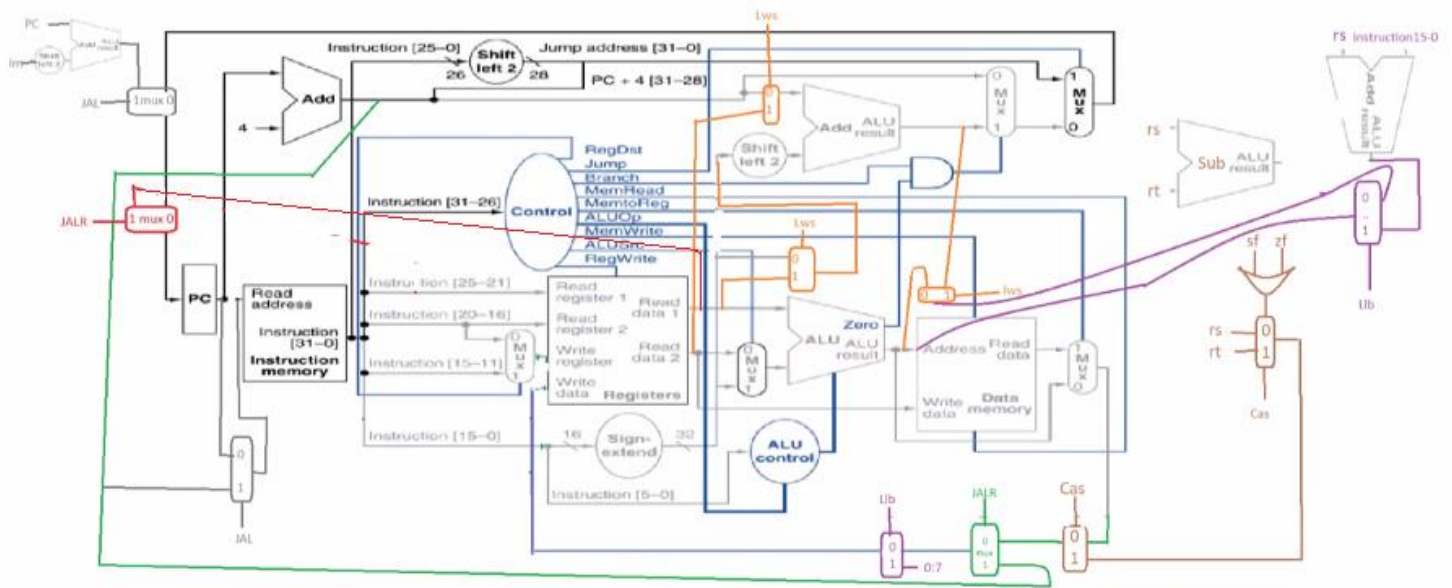
## 1.3 Starting modify:

This is our solution for Question 1 to add the following instruction to the data-path:

<b>Lws \$rd, \$rs, \$rt</b>	<b># rd = Mem[4*rs + rt]</b>
<b>Cas rd, rs, rt</b>	<b># Reg(Rd) = Max[Reg(Rs) , Reg(Rt)]</b>
<b>Llb rt, imm(rs)</b>	<b># Reg(Rt[0:7]) = Mem(Reg(Rs) + Imm16)</b>
<b>Jal</b>	<b># R31 = PC + 4, PC = PC + Imm*4</b>
<b>Jalr</b>	<b># rd = PC + 4, PC = rs</b>

## ALU signals:

Op	Funct	ALU	4-bit ALU
R-type	AND	AND	0001
R-type	OR	OR	0010
R-type	XOR	XOR	0011
R-type	ADD	ADD	0100
R-type	SUB	SUB	0101
R-type	SLT	SLT	0110
ADDI	X	ADD	0100
SLTI	X	SLT	0110
ANDI	X	AND	0001
ORI	X	OR	0010
XORI	X	XOR	0011
LW	X	ADD	0100
SW	X	ADD	0100
BEQ	X	SUB	0101
BNE	X	SUB	0101
J	X	X	X
lws	X	ADD	0000
cas	X	SUB	0010
llb	X	ADD	0000
jal	X	X	X
jalr	X	X	X



## Truth table:

OP	Reg dst	Reg write	Ext op	Alu src	beq	bne	j	Mem read	Mem write	Mem toreg	Mux lw	Mux cas	Mux Lb	Mux JAL	Mux JALR
R-type	<u>1</u>	<u>1</u>	<b>x</b>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>
addi	<u>0</u>	<u>1</u>	<u>1</u>	<u>1</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>
slti	<u>0</u>	<u>1</u>	<u>1</u>	<u>1</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>
Andi	<u>0</u>	<u>1</u>	<u>0</u>	<u>1</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>
Ori	<u>0</u>	<u>1</u>	<u>0</u>	<u>1</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>
xori	<u>0</u>	<u>1</u>	<u>0</u>	<u>1</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>
<u>lw</u>	<u>0</u>	<u>1</u>	<u>1</u>	<u>1</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>1</u>	<u>0</u>	<u>1</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>
<u>sw</u>	<b>x</b>	<u>0</u>	<u>1</u>	<u>1</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>1</u>	<b>x</b>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>
<u>Beq</u>	<b>x</b>	<u>0</u>	<b>x</b>	<u>0</u>	<u>1</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<b>x</b>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>
<u>bne</u>	<b>x</b>	<u>0</u>	<b>x</b>	<u>0</u>	<u>0</u>	<u>1</u>	<u>0</u>	<u>0</u>	<u>0</u>	<b>x</b>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>
<u>J</u>	<b>X</b>	<u>0</u>	<b>x</b>	<b>x</b>	<u>0</u>	<u>0</u>	<u>1</u>	<u>0</u>	<u>0</u>	<b>x</b>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>
<u>Lws</u>	<u>1</u>	<u>1</u>	<b>x</b>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>1</u>	<u>0</u>	<u>1</u>	<u>1</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>
<u>Cas</u>	<u>1</u>	<u>1</u>	<b>x</b>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>1</u>	<u>0</u>	<u>0</u>	<u>0</u>
<u>Llb</u>	<u>0</u>	<u>1</u>	<u>1</u>	<u>1</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>1</u>	<u>0</u>	<u>1</u>	<u>0</u>	<u>0</u>	<u>1</u>	<u>0</u>	<u>0</u>
<u>Jal</u>	<b>x</b>	<u>1</u>	<b>x</b>	<b>x</b>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>1</u>	<u>0</u>
<u>Jalr</u>	<u>1</u>	<u>1</u>	<b>x</b>	<b>x</b>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	<u>1</u>

## Equations from truth table:

$$\text{MemRead} = \text{lw} + \text{LWS} + \text{Llb}$$

$$\text{MemWrite} = \text{sw}$$

$$\text{MemtoReg} = \text{lw} + \text{LWS} + \text{Llb}$$

$$\text{RegDst} = \text{R-type} + \text{Lws} + \text{Cas} + \text{Jalr}$$

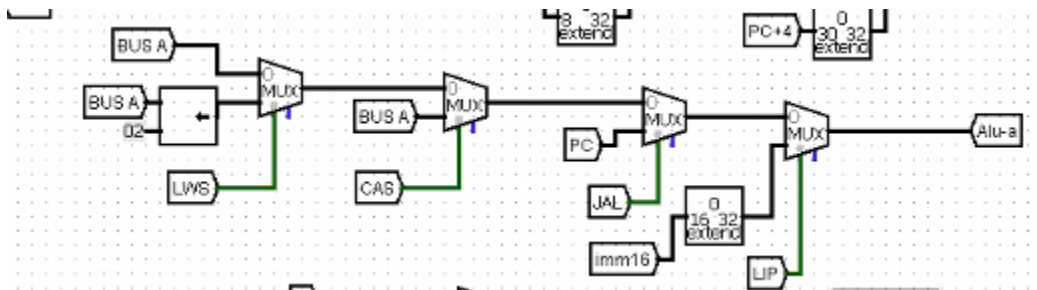
$$\text{RegWrite} = (\text{sw} + \text{beq} + \text{bne} + \text{j})$$

$$\text{ExtOp} = (\text{andi} + \text{ori} + \text{xori})$$

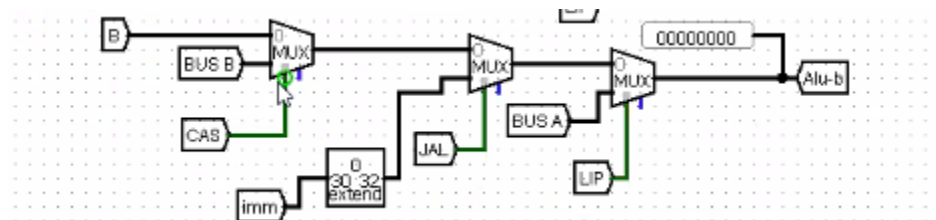
$$\text{ALUSrc} = (\text{R-type} + \text{beq} + \text{bne} + \text{Lws} + \text{Cas})$$

## How we modified?

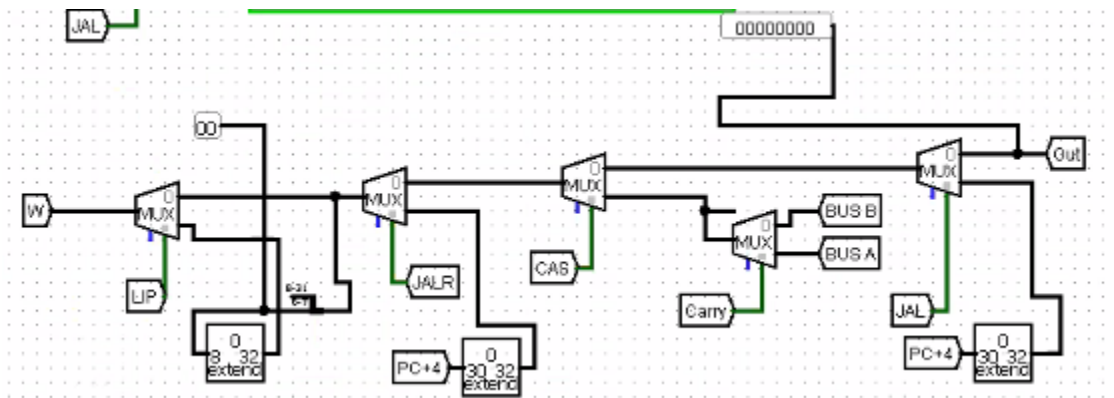
We made the First input for ALU (A) in this way:



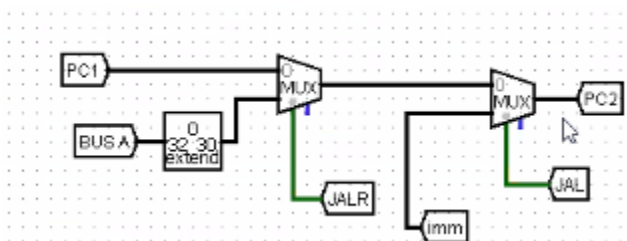
Second input for ALU (B)



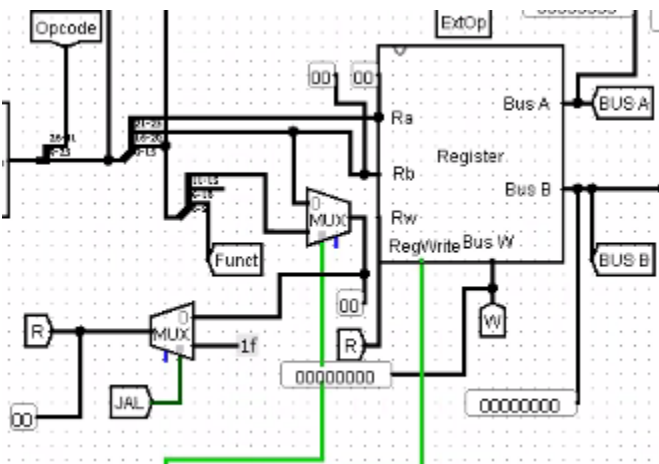
The 4-stages of muxes that controlling the write data signal depending on the instruction



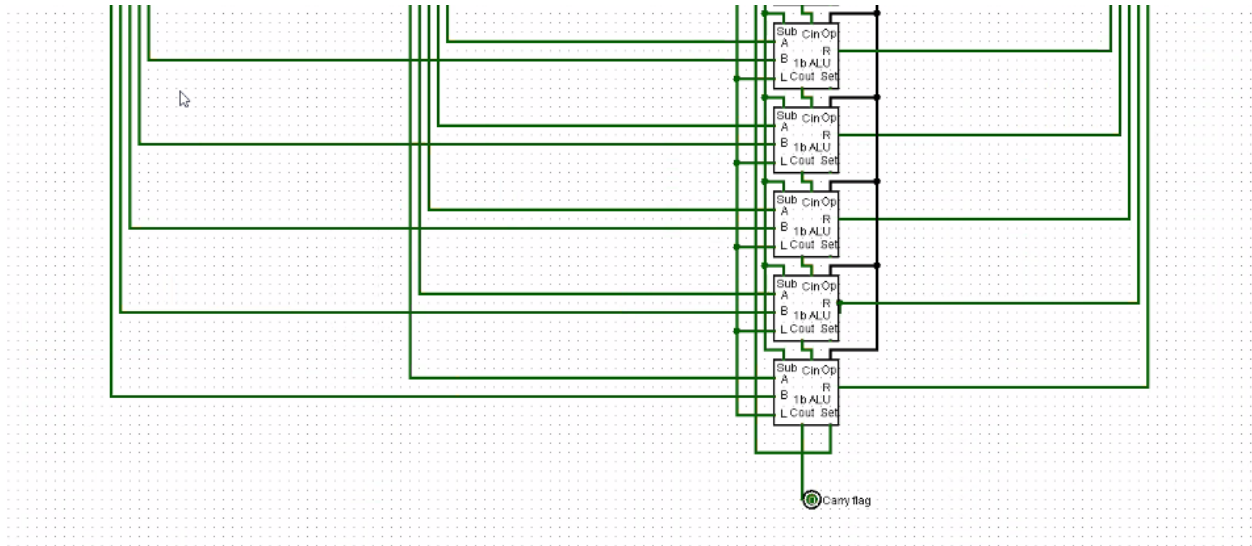
To determine the PC value according to the instruction .



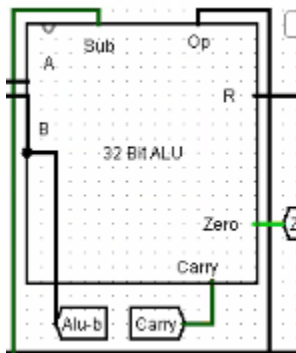
**Set the wright register to 31 in order to write on the R31**



**We add a pin in the ALU in order to use it when we subtract to determine the max value(CAS)**

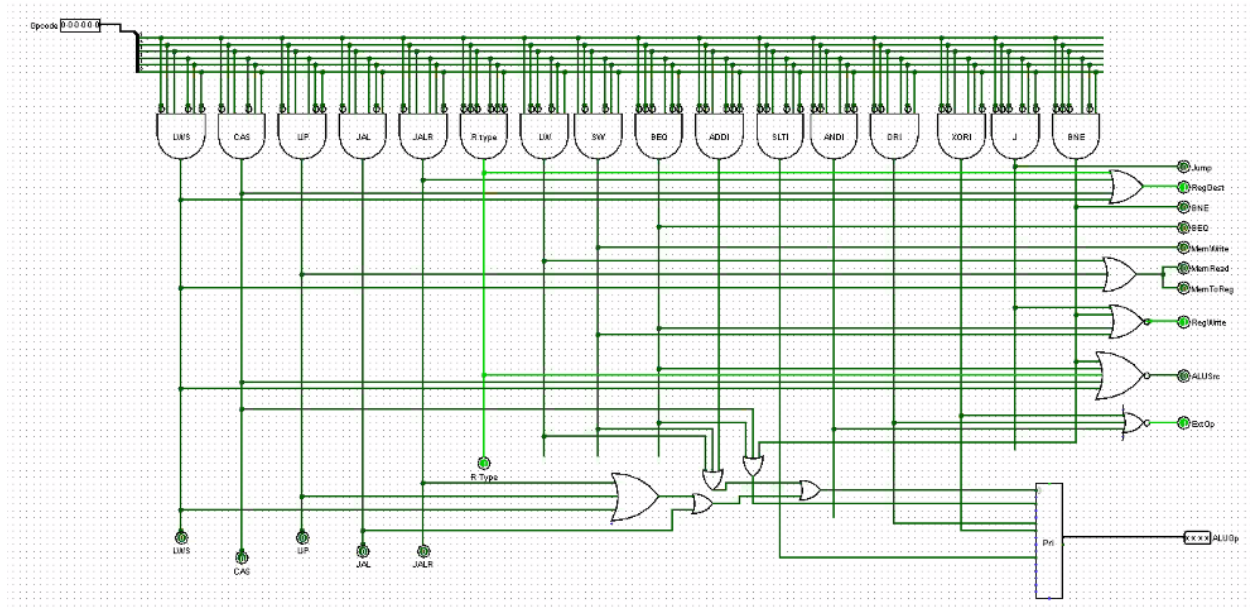


**See ALU component after adding the pin**

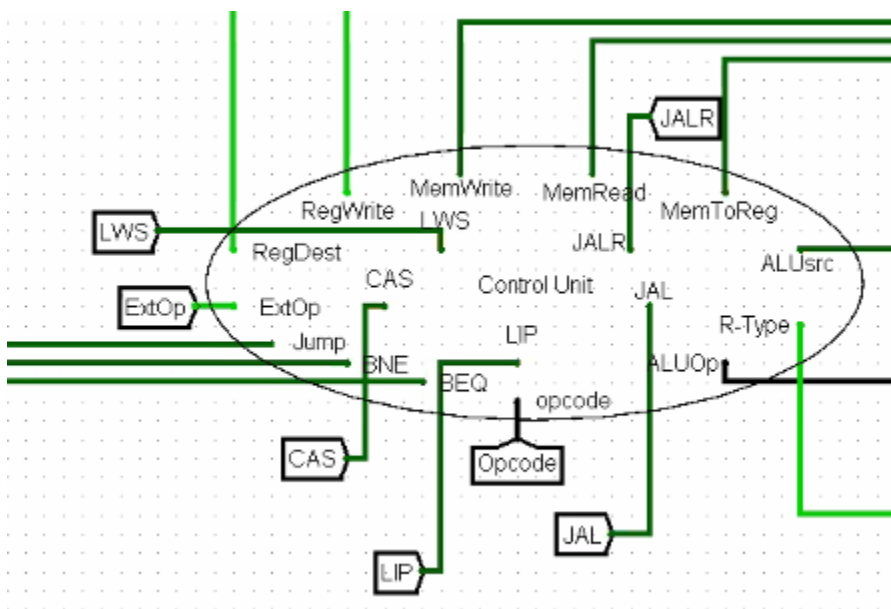


**We modified the control unit with the unique opcode for every added instruction , and we modified the signals according to the Equations.**

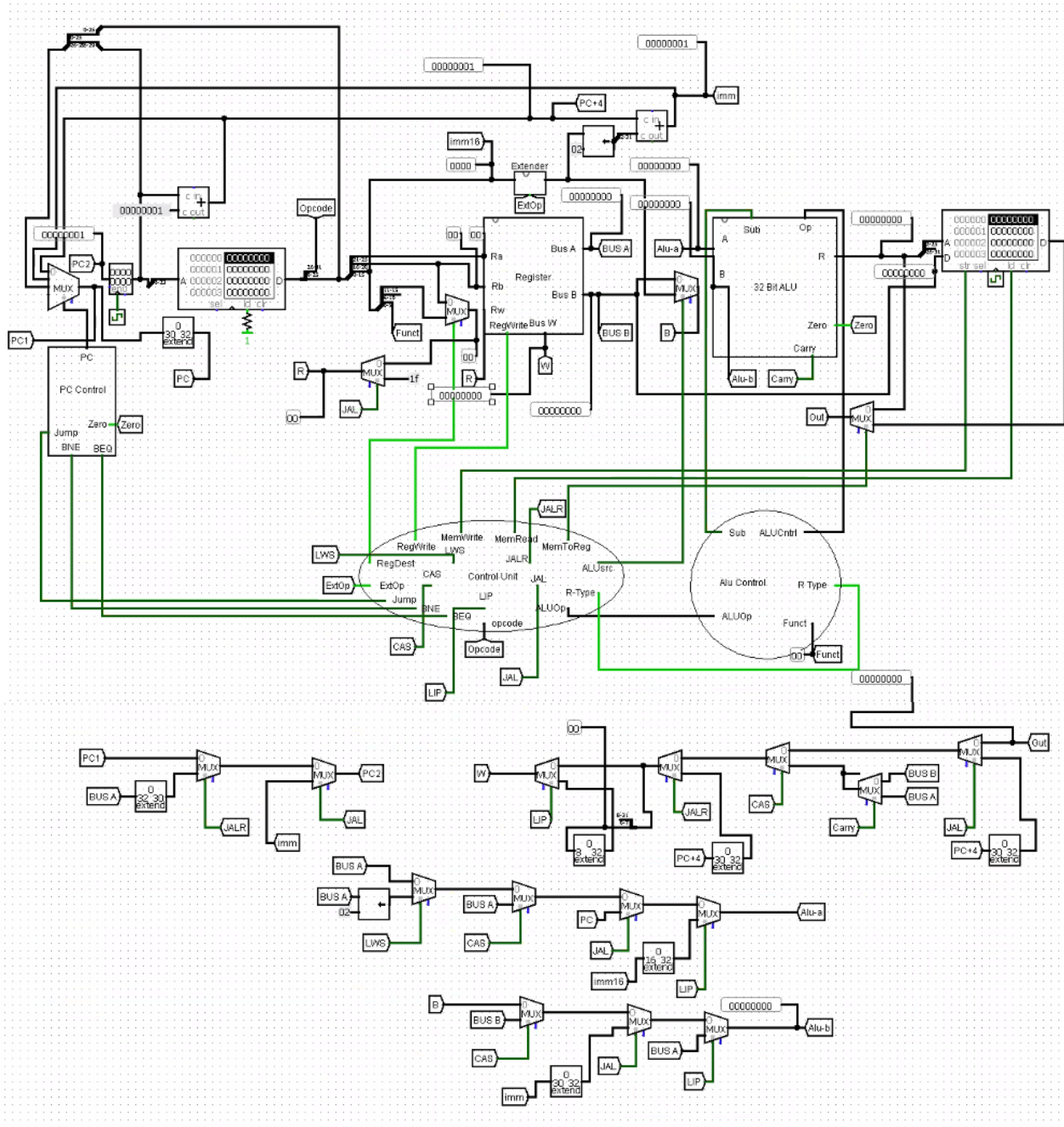




**Control unit after modified :**



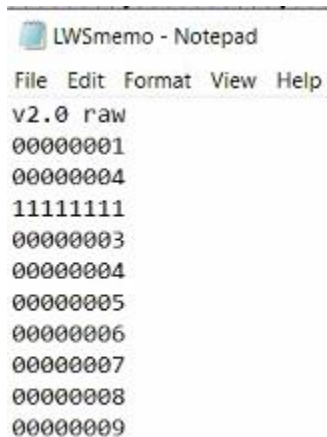
## The overall design after modification and adding the instruction:



## 2 Simulation and Testing:

### 1. Lws:

Data mem code to fill data:



```
LWSmemo - Notepad
File Edit Format View Help
v2.0 raw
00000001
00000004
11111111
00000003
00000004
00000005
00000006
00000007
00000008
00000009
```

instruction mem contains instructions:

in assembly:

```
sub r0,r0,r0 (set reg[0] to 0, use as base)
lw r1,0(r0) ( reg[1] <- mem[0] (= 1))
lw r2,4(r0) (reg[2] <- mem[4] (= 4))
lws r3,r2,r1rd ( Mem[4*rs + rt] )
beq r0,r0,-1 (program is over, keep looping back to here)
```

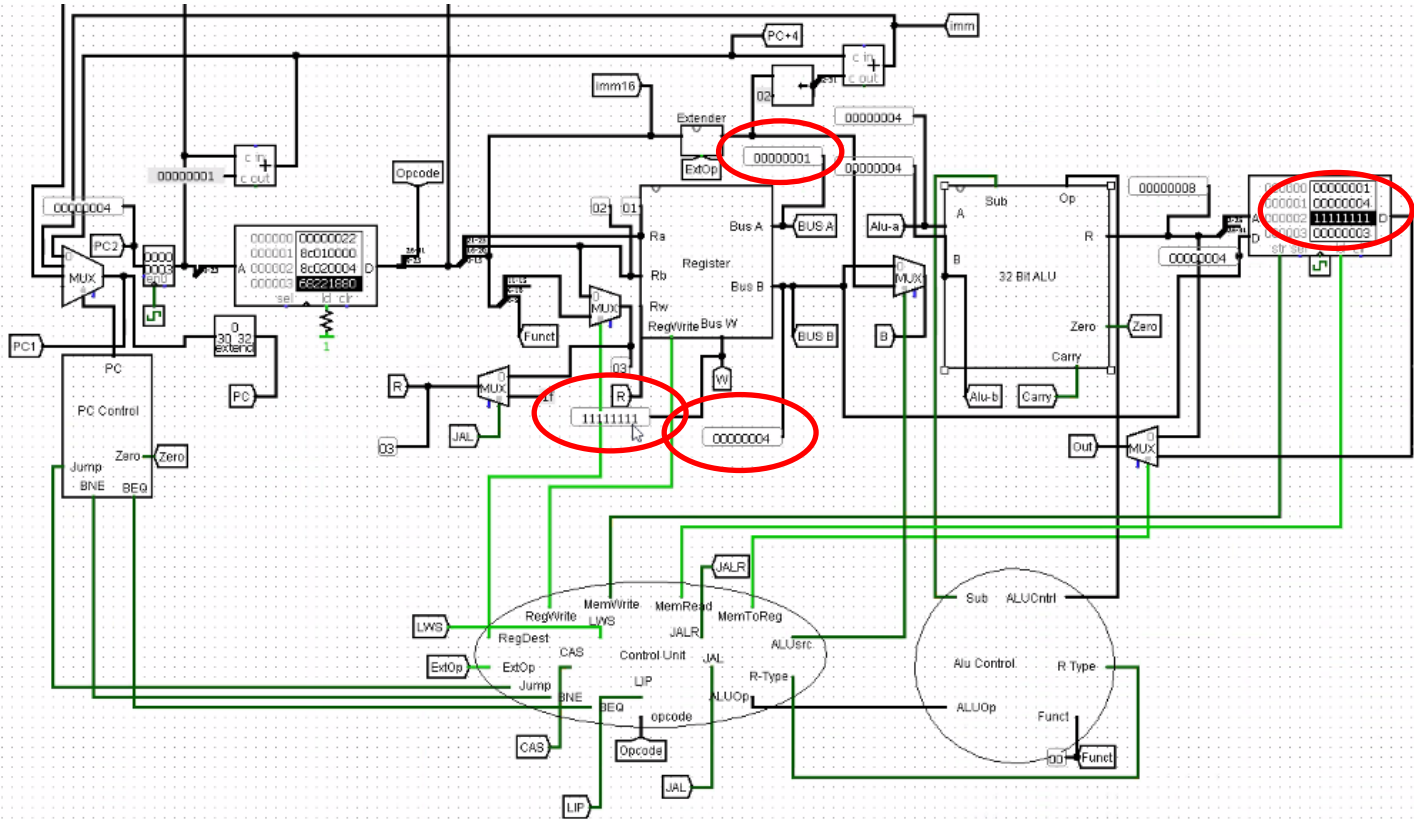
to machine language:

example to convert:

```
Lws r3 , r2 ,r1
011010 00001 00010 00011 0 0010 0000
0110 1000 0010 0010 0001 1000 1000 0000
6 8 2 2 1 8 8 0
```



**Result:**



## 2.CAS

sub r0,r0,r0 (set reg[0] to 0, use as base)

lw r1,0(r0) ( reg[1] <- mem[0] (= 1))

lw r2,4(r0) (reg[2] <- mem[4] (= 4))

cas r3,r1,r2 (Reg(Rd) = Max[Reg(Rs) , Reg(Rt)])

beq r0,r0,-1 (program is over, keep looping back to here)

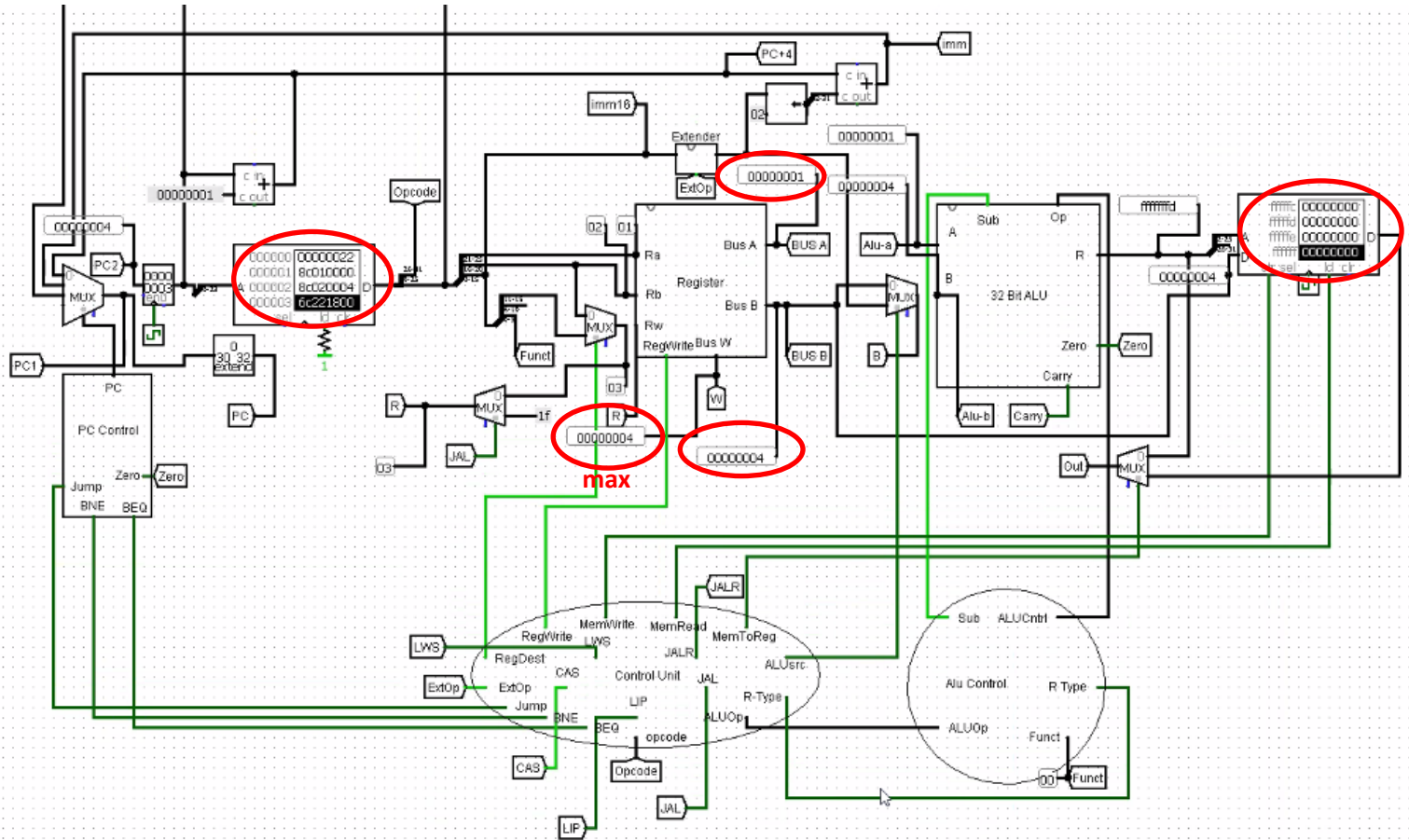
machine:

File	Edit	Format
v2.0	raw	
00000022		
8c010000		
8c020004		
6c221800		
1000ffff		

Data in memory:

00000004  
0000000a  
11111111  
00000003  
00000004  
00000005  
00000006  
00000007  
00000008  
00000009  
0000000a  
0000000b  
0000000c  
0000000d  
0000000e

## Result:



### 3.Llb

sub r0,r0,r0

lw r1,8(r0)

lw r2,4(r0)

Llb r2, imm(r1) (**Reg(Rt[0:7]) = Mem(Reg(R1) + Imm6)**)

beq r0,r0,-1 (**program is over, keep looping back to here**)

**program code:**

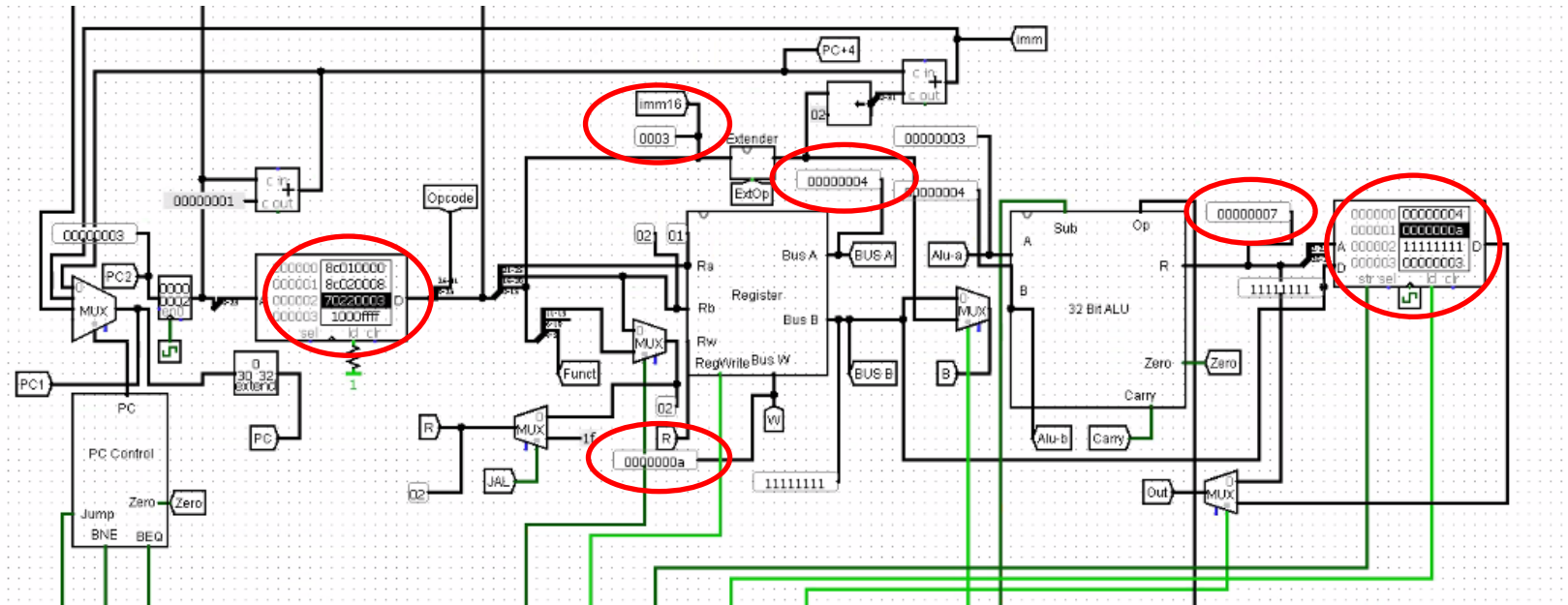
```
File Edit Forr
v2.0 raw
8c010000
8c020008
70220003
1000ffff|
```

**Data in memory:**

```
00000004
0000000a
|11111111
00000003
00000004
00000005
00000006
00000007
00000008
00000009
0000000a
0000000b
0000000c
0000000d
0000000e
```



## Result:



## **4.jal**

sub r0,r0,r0

lw r1,8(r0)

lw r2,4(r0)

jal

beq r0,r0,-1

### **converting into machine language:**

Jal

0111 0100 0000 0000 0000 0000 0000 0011

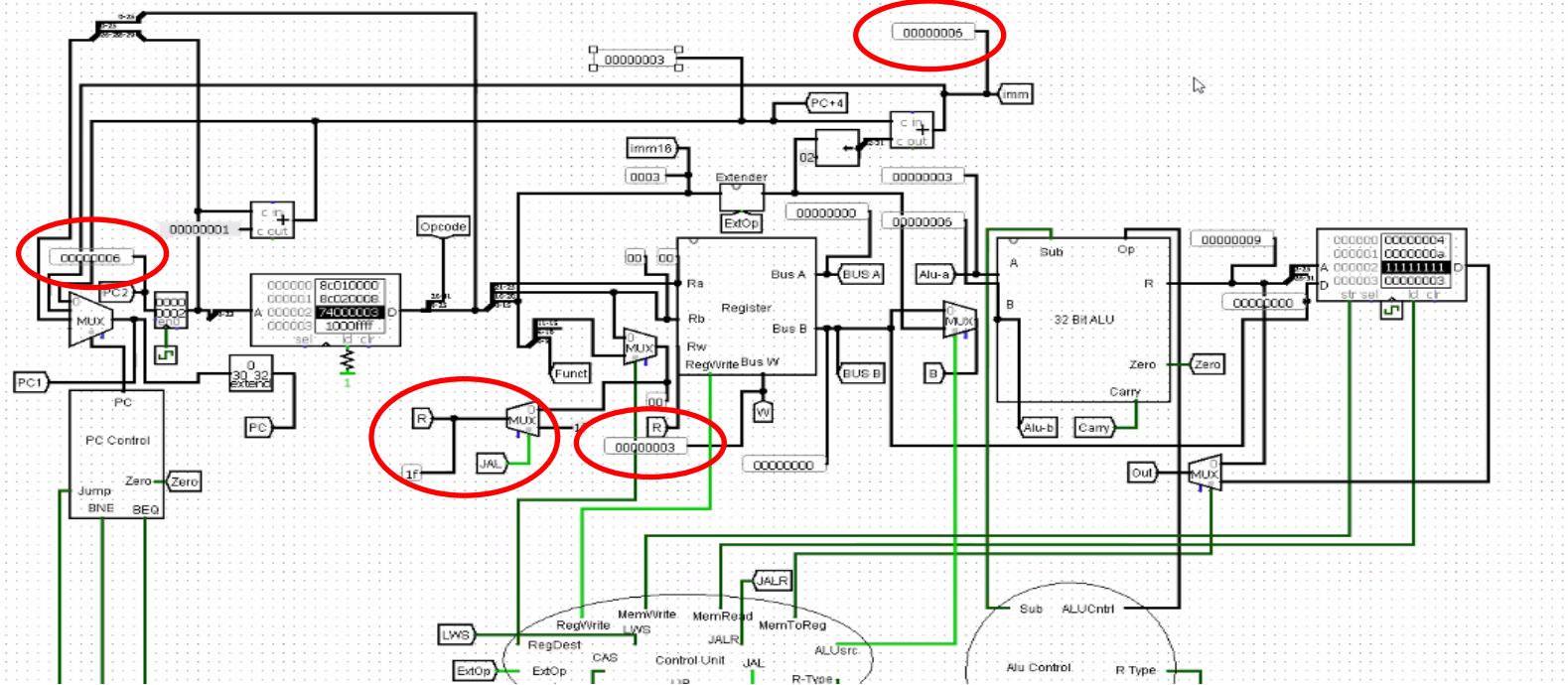
740000003

The program:

```
v2.0 raw
8c010000
8c020008
74000003
1000ffff
|
```

**No need to use data memory.**

**Result:**



## 5.jalr

sub r0,r0,r0

lw r1,8(r0)

lw r2,4(r0)

jalr r1,r2

beq r0,r0,-1

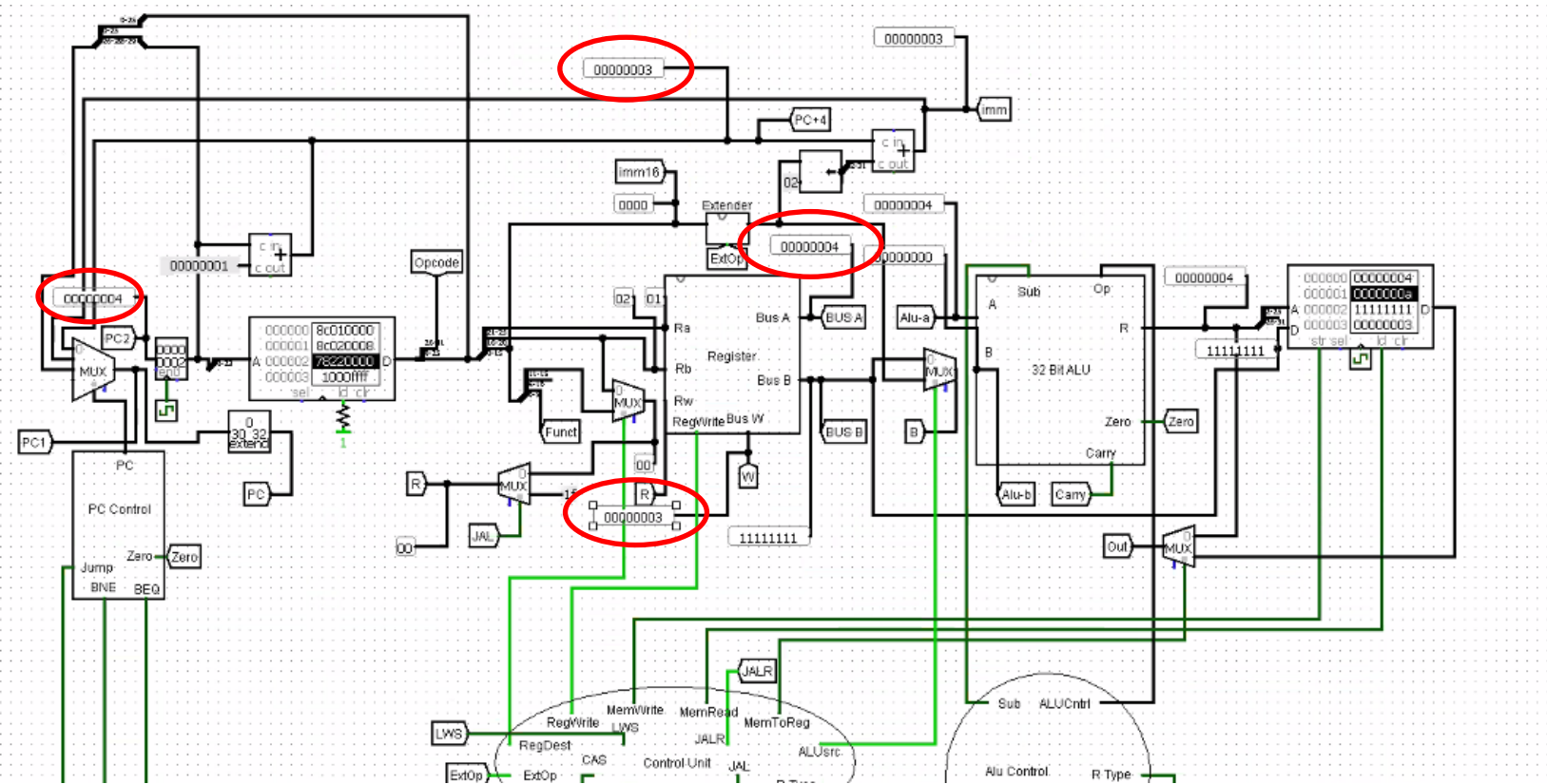
### **program code:**

```
v2.0 raw
8c010000
8c020008
78220000
1000ffff
|
```

### **Data memory:**

```
00000004
0000000a
|11111111
00000003
00000004
00000005
00000006
00000007
00000008
00000009
0000000a
0000000b
0000000c
0000000d
0000000e
```

## Result:



### 3 Teamwork:

We (Tala and Zaid) were both working on this project using zoom sessions, both controlling and modifying the design together, see fig3.1. Throughout the semester, we have been working together. Throughout the early month of May, most of the basic design of the processor was heavily researched and tested.

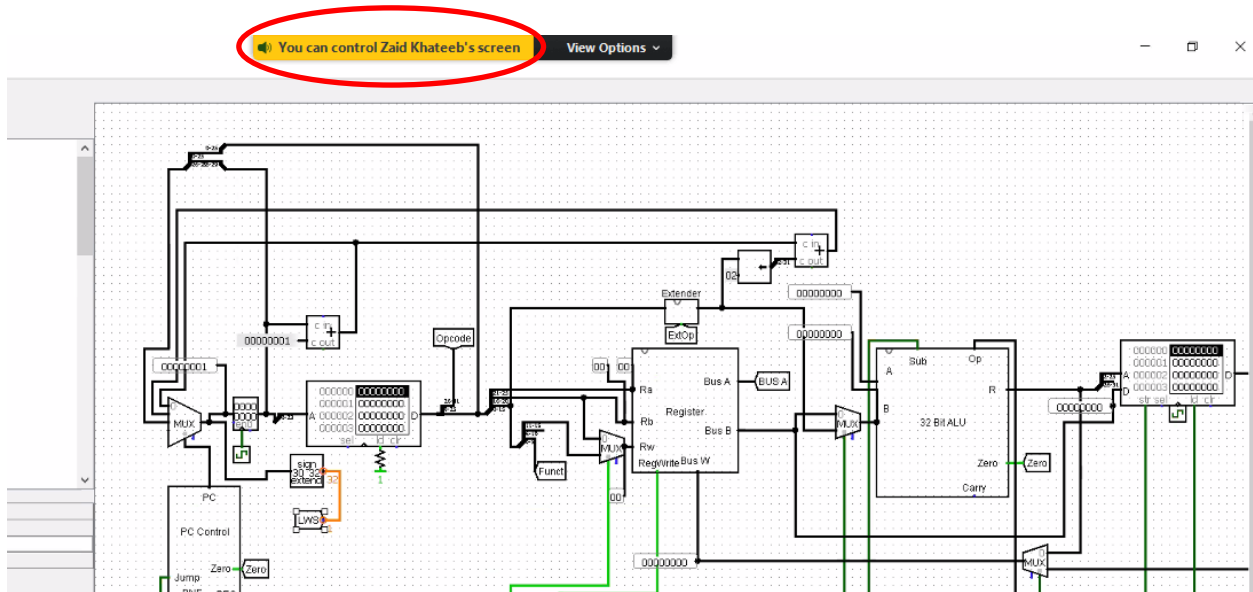


Figure 10.1 how we work together

## **Conclusion:**

This project wasn't easy , it teaches us how to make a CPU ! on single-cycle data-path, it teaches us how to modify this data-path to keep up with any changes or added instruction to be handled, executed and tested to make sure , testing process wasn't easy too , but we have to do it to compare if the expected result of an instruction is the same with what we see after executing this instruction .