FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER ENGINEERING

Computer Architecture ENCS437

Project 2 Report

Prepared by:

Weaam Hjijah 1181988

Siham Abu Rmaileh 1180548

Israa Haseeba 11812467

Supervised by:

Dr. Aziz Qaroush

June 2022

Abstract

The aim of this project is to use the Logisim simulator to design and test a single 24-bit RISC processor.

The processor consists of eight 24-bit general-purpose registers: R0 through R7, 24-bit program counter and 8-bit status register, it supports three address formats (R-type format, I-type format, J-type format) and different addressing modes.

The data path will consist of five stages, each one will be implemented individually, then the control and the functional units will be connected to form the whole data path.

Contents

Design and Implementation

In order to implement the data path, each functional unit was implemented individually as follows.

PC

This unit determines the PC value, which is the address of the next instruction to be executed. It depends on some of the generated signals like BEQ, J and JAL, the other inputs of this unit are: jump address, branch address, clock and clear bit.

The next, jump and branch addresses are given as follows:

Next address = PC +1

Jump address = PC + Immediate17

Branch address = PC + Immediate17

Three adders were used to calculate the above three addresses, then a combination of multiplexers was used to select one of the three addresses to be written on the PC. Figure below shows the implementation of this unit.

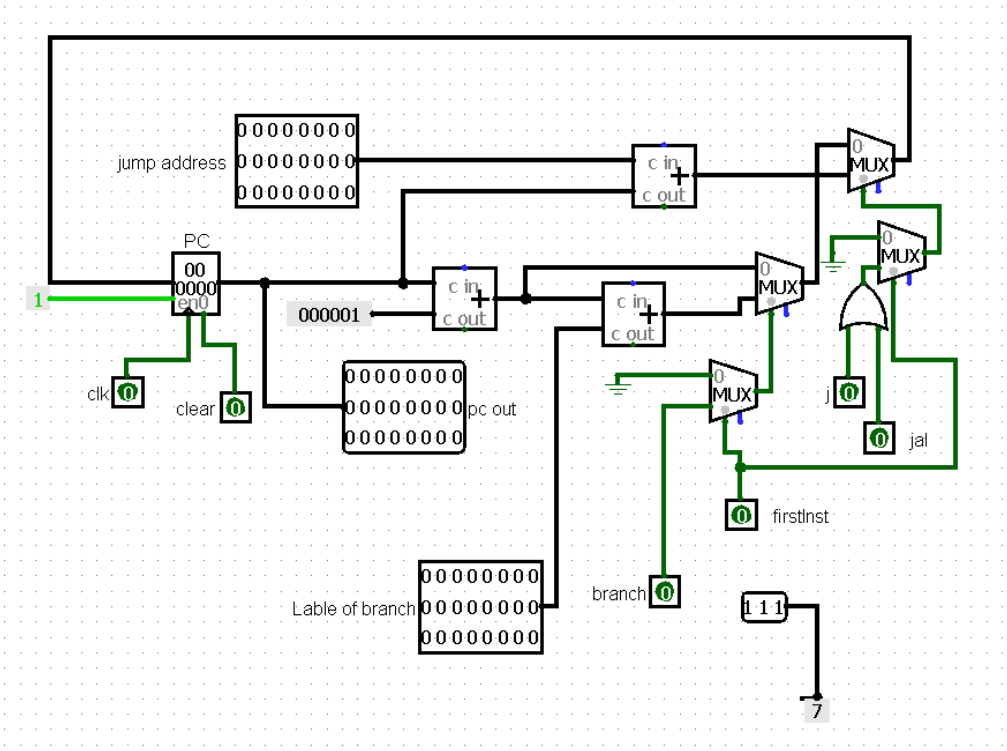


Figure 1:PC Unit

10-bit extender

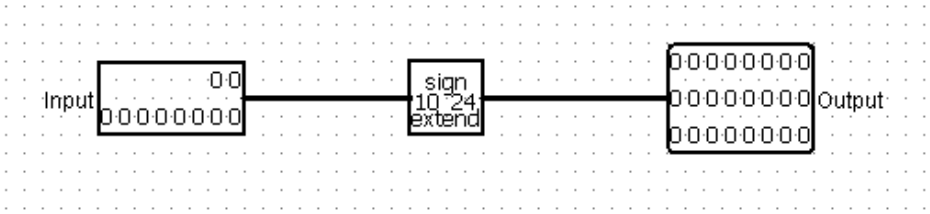


Figure 2: 10-bit extender

17-bit extender

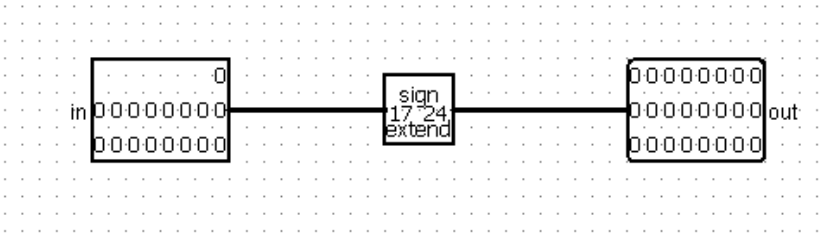
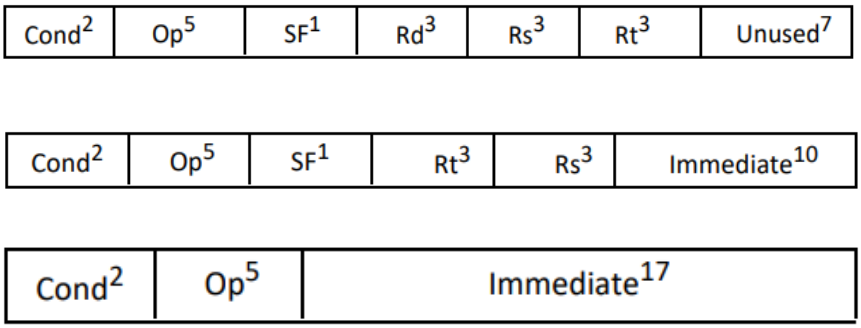


Figure 3: 17-bit extender

Decode Stage

This unit splits the fetched instruction into different fields based on the given formats, in order to use them in the later stages.



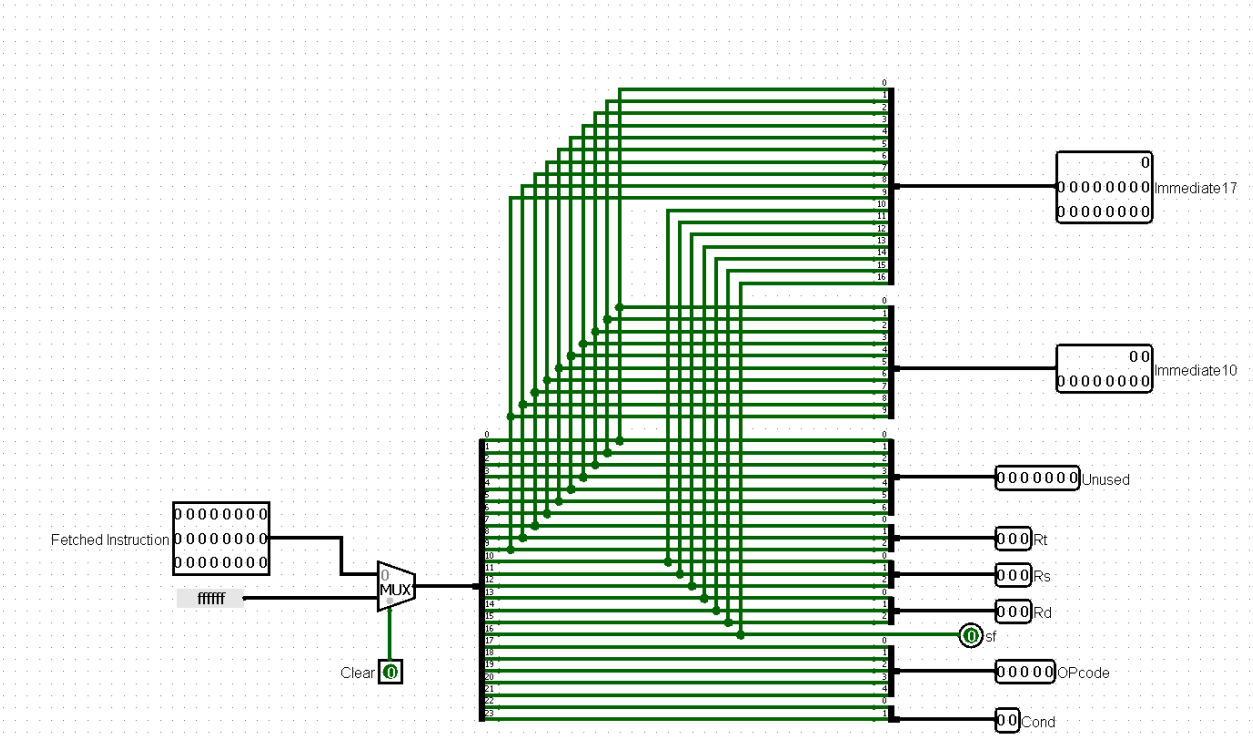


Figure 4: Decode stage

Register File

Register file consists of a set of registers which are used to stage data between memory and the functional units, in addition to the circuits that are responsible for writing and reading from these registers. The eight 24-bit general-purpose registers were represented as an array as shown in figure below.

A 3*8 decoder was used to decode the address of the desired register. In order to perform the writing on a register, two pins must be activated, the first one is the write enable pin which is connected to the all registers, the second one is the address of the that register, write operation is synchronized with the system clock. Reading from a register is performed by selecting the address of the desired register using a multiplexer. As one of the supported formats has to read two sources concurrently, two multiplexers were used.

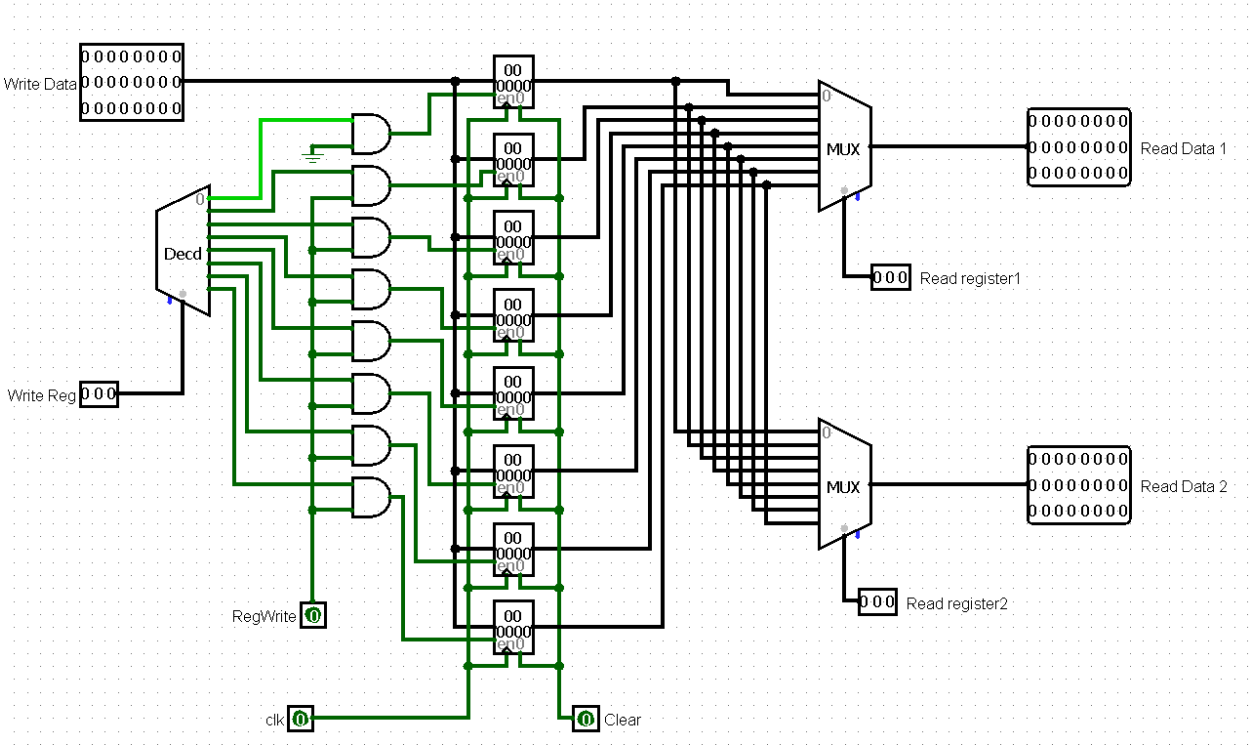


Figure 5: Register file

Control Unit

The control unit or CU is responsible for generating the signals that tells the processor components how to respond to a specific instruction. Generating the signals was done based on the instruction opcode, so a binary decoder was used with the opcode as the input to perform that.

To build the unit correctly, a truth table was implemented to obtain all signals equations as follows:

Table 1: Control unit signals

	RegDes	RegWr	ExtOp	ALUSrc	BEQ	JR	JAL	J	MemRd	MemWr	LUI
AND	00=Rd	1	00	0=Rd	0	0	0	0	0	0	0

CAS	00=Rd	1	00	0=Rd	0	0	0	0	0	0	0
Lws	00=Rd	1	00	0=Rd	0	0	0	0	1	0	0
ADD	00=Rd	1	00	0=Rd	0	0	0	0	0	0	0
SUB	00=Rd	1	00	0=Rd	0	0	0	0	0	0	0
CMP	X	0	00	0=Rd	0	0	0	0	0	0	0
JR	X	0	00	0=Rd	0	1	0	0	0	0	0
ANDI	01=Rt	1	10	1=imm	0	0	0	0	0	0	0
ADDI	01=Rt	1	10	1=imm	0	0	0	0	0	0	0
Lw	01=Rt	1	10	1=imm	0	0	0	0	1	0	0
Sw	X	0	10	1=imm	0	0	0	0	0	1	0
BEQ	X	0	00	1=imm	1	0	0	0	0	0	0
J	X	0	01	1=imm	0	0	0	1	0	0	0
JAL	10=R7	1	01	1=imm	0	0	1	0	0	0	0
LUI	11=R1	1	01	1=imm	0	0	0	0	1	0	1

RegWr = (AND + CAS + Lws + ADD + SUB + ANDI + ADDI + Lw + JAL + LUI)

10-bit extension = (ANDI + ADDI + Lw + Sw + BEQ)

17-bit extension = (J + JAL + LUI)

ALUSr = 1 when (ANDI + ADDI + Lw + Sw + BEQ + J + JAL + LUI)

BEQ = BEQ

JR = JR

J = J

JAL = JAL

MemRd = (Lws + Lw + LUI)

MemWr = Sw

LUI = LUI

After that, the control unit was implemented as shown in figure below.

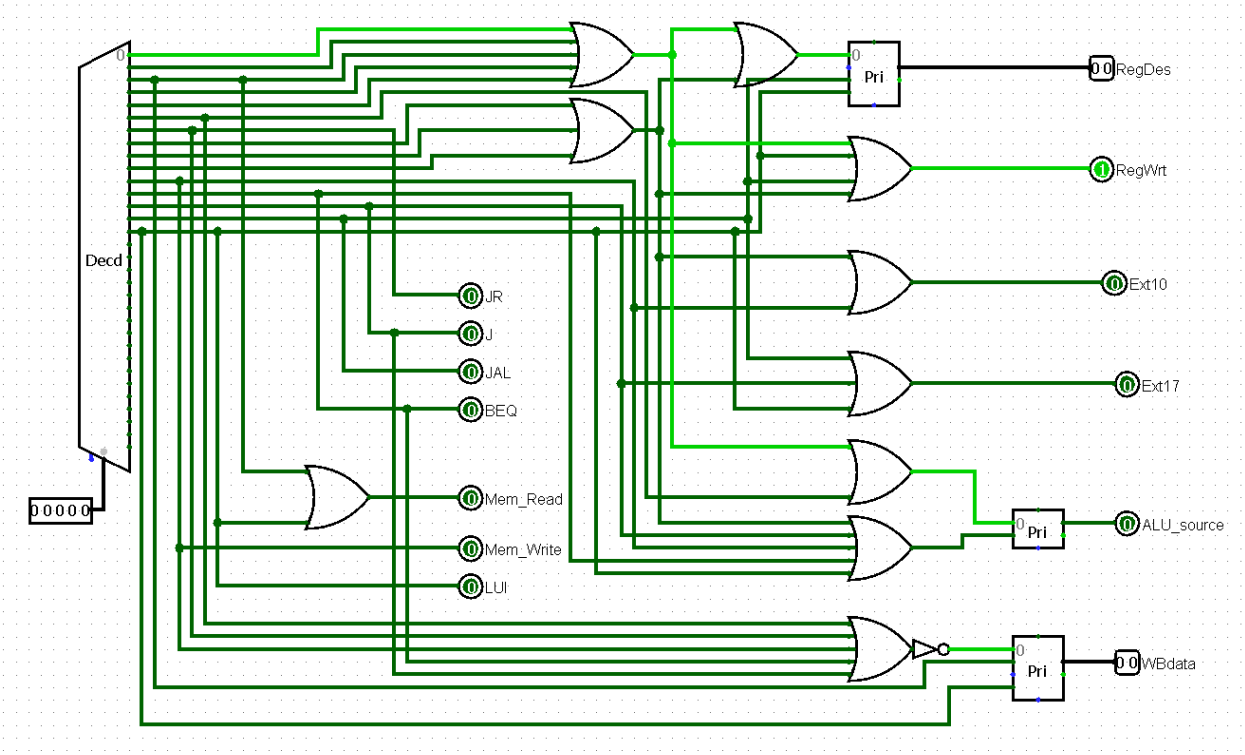


Figure 6: Control unit

ALU Control Unit

As shown in the figure below the ALU Control Unit was designed to choose the required operation in order to execute our instruction. The implementation was very simple, it was basically depended on decoding the OP code of the instruction then adding all the OP codes that shared the same operation into OR gate, then all the ORS' output were used as input into an encoder to decide the operation and out "ALU_oper" signal.

In the implementation there was some special cases that required from us to deal with it using separate small units:

Condition Decoder: This unit was used to choose the selected condition of the instruction. The implementation was done using decoder which takes the two-bit condition and out "selected_condition" signal as shown in the following figure.

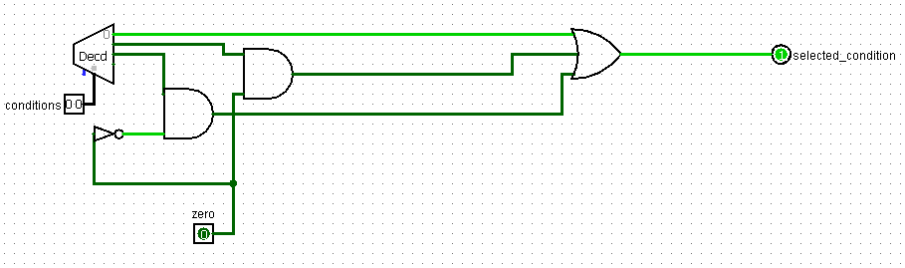


Figure 7: Condition decoder

Addition Condition

This block works the same as the general condition block but with taking into account some special cases that are related to the addition operation such as the existence of SF signal.

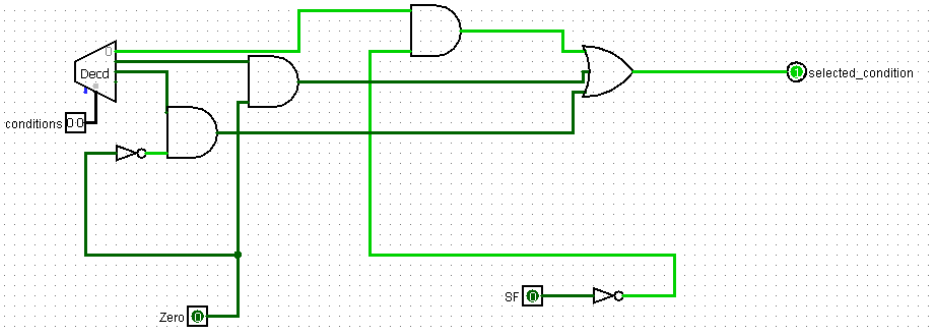


Figure 8: Addition condition

Sub Condition

This block works the same as the general condition block but with taking into account some special cases that are related to the subtraction operation such as the existence of SF signal.

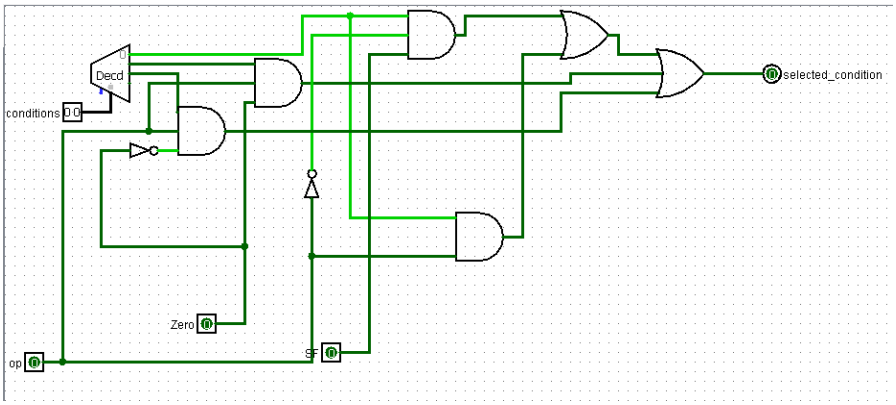


Figure 9: Subtraction condition

Full implementation of the ALU control unit.

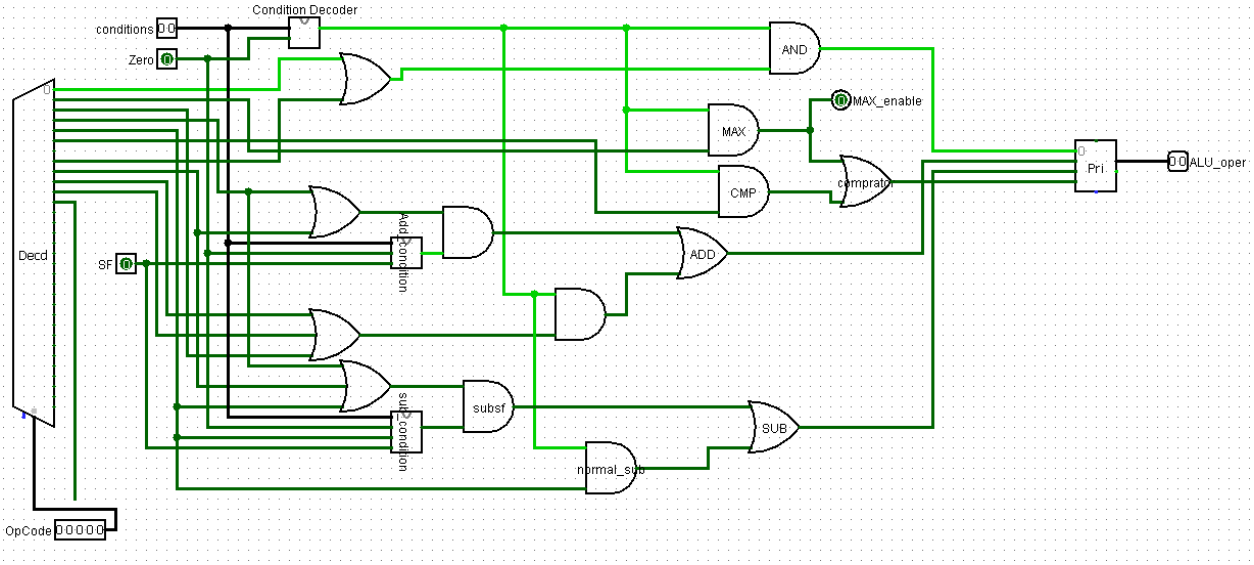


Figure 10: ALU control unit

Table 2: ALU operation decode

OP	Name	Condition	SF	ALU operation
00000	AND	00	0	AND
00000	ANDEQ	01	X	AND
00000	ANDNE	10	X	AND

00001	CAS	00	0	MAX
00001	CASEQ	01	X	MAX
00001	CASNE	10	X	MAX
00010	Lws	00	0	ADD
00010	LwsEQ	01	X	ADD
00010	LwsNE	10	X	ADD
00011	ADD	00	0	ADD
00011	SUBSF	00	1	SUB
00011	ADDEQ	01	X	ADD
00011	ADDNE	10	X	ADD
00100	SUB	00	0	SUB
00100	SUBEQ	01	X	SUB
00100	SUBNE	10	X	SUB
00101	CMP	X	0	CMP
00110	JR	00	0	-
00110	JREQ	01	X	-
00110	JRNE	10	X	-
00111	ANDI	00	0	AND
00111	ANDIEQ	01	X	AND
00111	ANDINE	10	X	AND
01000	ADDI	00	0	ADD
01000	SUBISF	00	1	SUB
01000	ADDIEQ	01	X	ADD
01000	ADDINE	10	X	ADD
01001	Lw	00	0	ADD
01001	LwEQ	01	X	ADD
01001	LwNE	10	X	ADD
01010	Sw	00	0	ADD
01010	SwEQ	01	X	ADD
01010	SwNE	10	X	ADD

01011	BEQ	X	0	COM
01100	J	00	0	-
01100	JEQ	01	X	-
01100	JNE	10	X	-
01101	JAL	00	0	-
01101	JALEQ	01	X	-
01101	JALNE	10	X	-
01110	LUI	00	0	
01110	LUIEQ	01	X	
01110	LUINE	10	X	

ALU Unit

As shown in the previous figure, the implementation of the ALU unit was very simple, it just contains all the arithmetic components needed in our instruction set which are: an adder, subtractor, comparator, and gate and or gate, Also, we can see a mux, which is used for the CAS instruction, to output the maximum number between the inputs. Finally, all those components were connected to a mux having the ALU operation control signal on its selection line.

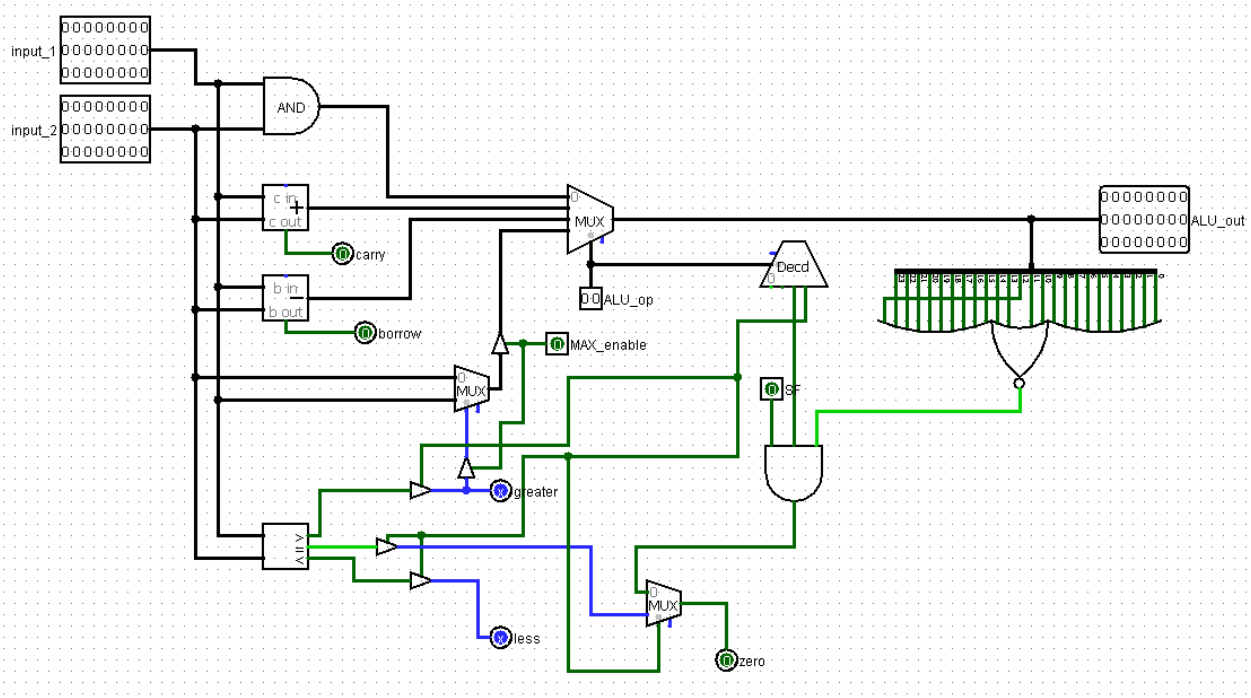


Figure 11: ALU unit

Instruction Memory

The Instruction Memory stores all the prefetch instructions. The input for the 24-bit data width memory is the address of the instruction fetched and stored in the instruction register. The output is simply the 24-bit instruction stored in that address. The memory has a pin which is responsible for enabling or disabling the component.

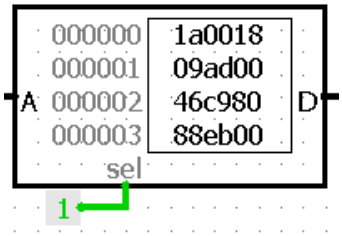


Figure 12: Instruction memory

Data Memory

The data memory consists of a series of fixed size blocks (24-bit data width).

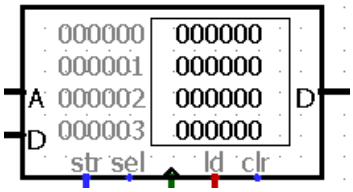


Figure 13:Data memory

Complete Data Path

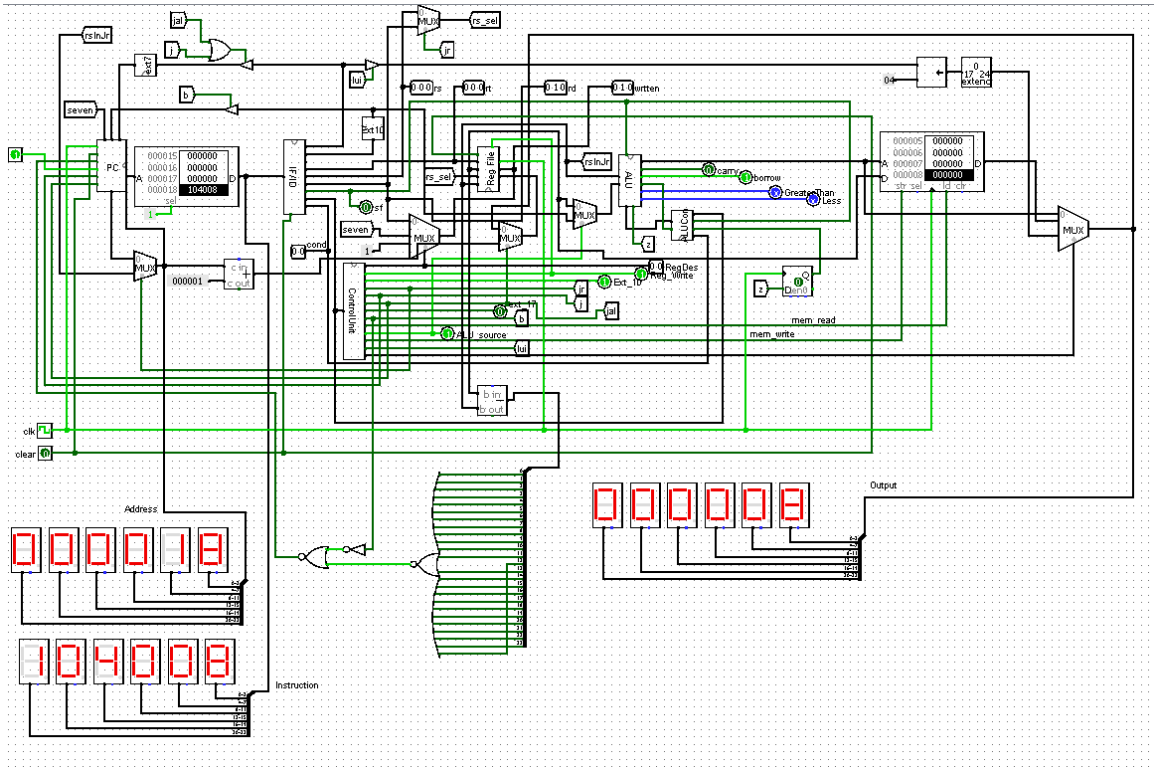


Figure 14: Data path

Simulation and Testing

There are two instruction lists tested on the data path.

First one:

```
v2.0 raw
104008 96800 98800 9ad00 46c980 88eb00 9ad00 80e800 63b80 180027
16*0 104008 96800 98800 508804 ce000 17*0 a0500
803b80 142000 12a000
```

```
ADDI r2,r0,8  ;r2 = 8
SUBSF r3,r2,r0 ;r3 = 8
SUBSF r4,r2,r0 ;r4 = 8
```

```
SUBSF r5,r3,r2 ;r5 = 0, zf goes from 0 to 1
ADDEQ r6,r2,r3 ;should work  r6 = 16, zf goes from 1 to 0
SUBNE r7,r2,r6 ;should work, r7 = -8, zf = 0
SUBSF r5,r3,r2 ;zf = 1
ANDNE r7,r2,r0 ;should not work
ADD r1,r6,r7 ;r1 = 8
j end


end:
cmp r1,r2 ;r1 = 8, r2 = 8, cmp ==> EQ ==> ZF = 0 (not LT)
ANDNE r1,r6,r7 ;should work
sw r1,r0,0
lw r5,r0,0
```

The data path works correctly. You can watch the full simulation of the program in the link below.

https://drive.google.com/file/d/1LymY4_Y4PIZ19n-iRts8y6fGOHzp99hy/view?usp=sharing

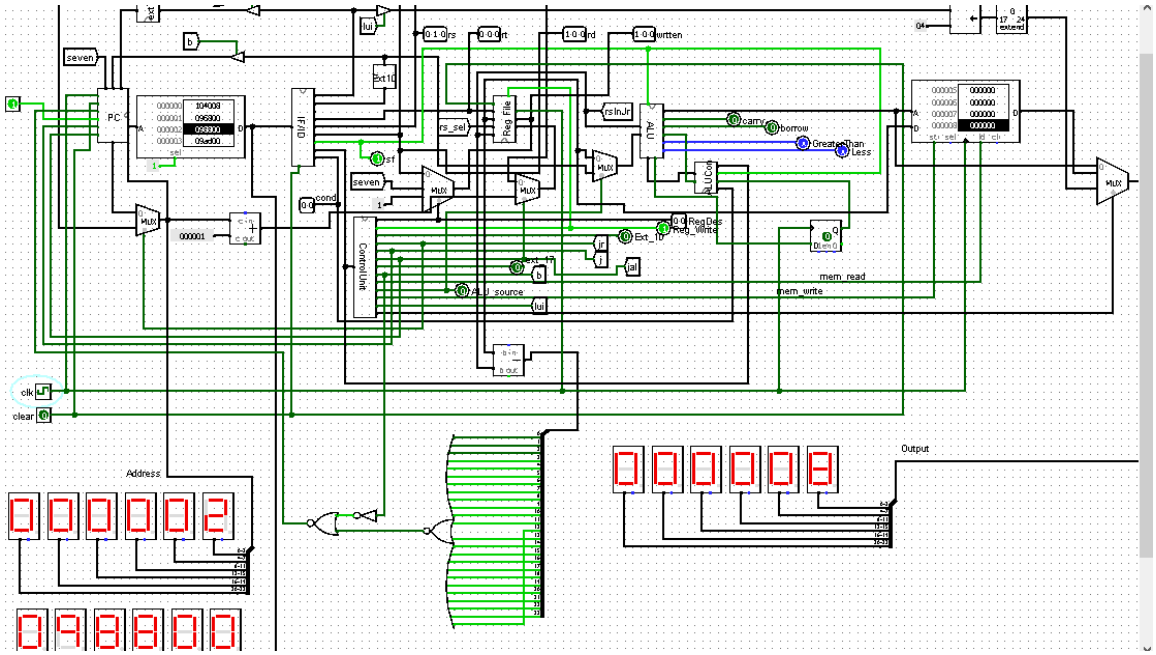


Figure 15: Test case 1

The second one: It is the same as the first one but with JAL and JR instructions.

```
v2.0 raw
1a0018 9ad00 46c980 88eb00 9ad00 80e800 63b80 180027
16*0 104008 96800 98800 508804 ce000 17*0 a0500
803b80 142000 12a000
```

And the supposed execution and the program itself are shown in the following figure.

```
JAL label
SUBSF r5,r3,r2 ;r5 = 0, zf goes from 0 to 1
ADDEQ r6,r2,r3 ;should work  r6 = 16, zf goes from 1 to 0
SUBNE r7,r2,r6 ;should work, r7 = -8, zf = 0
SUBSF r5,r3,r2 ;zf = 1
ANDNE r7,r2,r0 ;should not work
ADD r1,r6,r7 ;r1 = 8
j end


label:
        ADDI r2,r0,8  ;r2 = 8
        SUBSF r3,r2,r0 ;r3 = 8
        SUBSF r4,r2,r0 ;r4 = 8
        ADDIEQ r4,r2,4 ;should not work
        JR r7


end:
cmp r1,r2 ;r1 = 8, r2 = 8, cmp ==> EQ ==> ZF = 0 (not LT)
ANDNE r1,r6,r7 ;should work
sw r1,r0,0
lw r5,r0,0
```

Make sure when you simulate the data path that the firstinst signal is set to one. The program is loaded into the ROM and the simulation starts by triggering the clock. the supposed values that are shown in the figure above are the same as simulation.

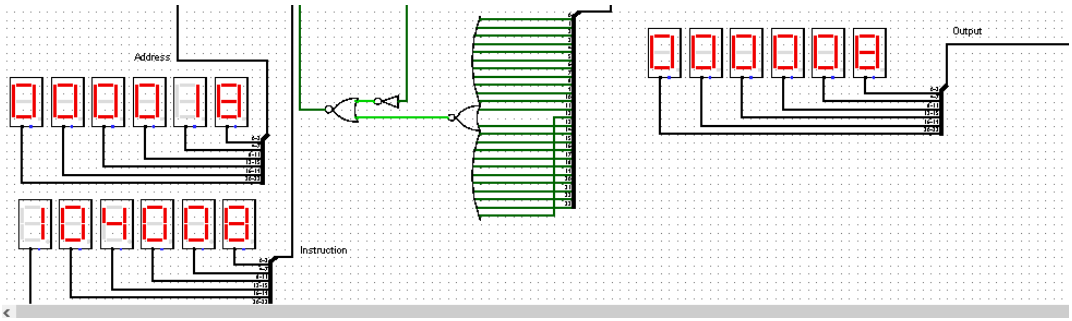


Figure 16: Output

Here is the output of the first instruction after jumping into the label. which is ADDI.

Notice that the registers are filled correctly, but the names of them are static. So that the destination register should be rt, but it appeared on rd. This is not a mistake, this is because of the order of filling the instructions in the IF/ID stage. So that Rd is always the first register to be filled.

To avoid this annoying thing, the output named "written" shows the distention register.

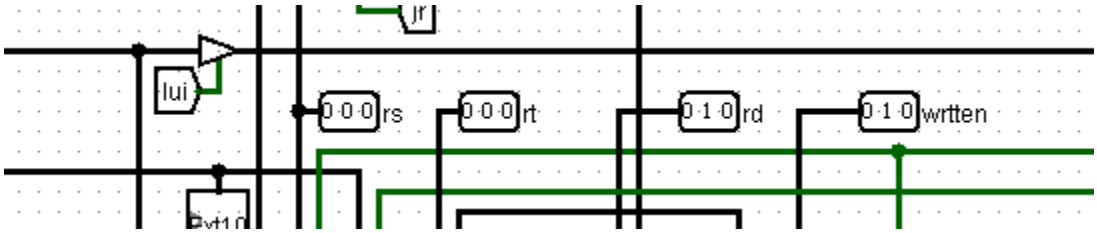


Figure 17: Registers values

However, the simulation went correctly until it has to return to the instruction that has the address stored in r7. Here, a state appeared called "Oscillation apparent". So that the simulation had to be reset.

The instruction before the error.

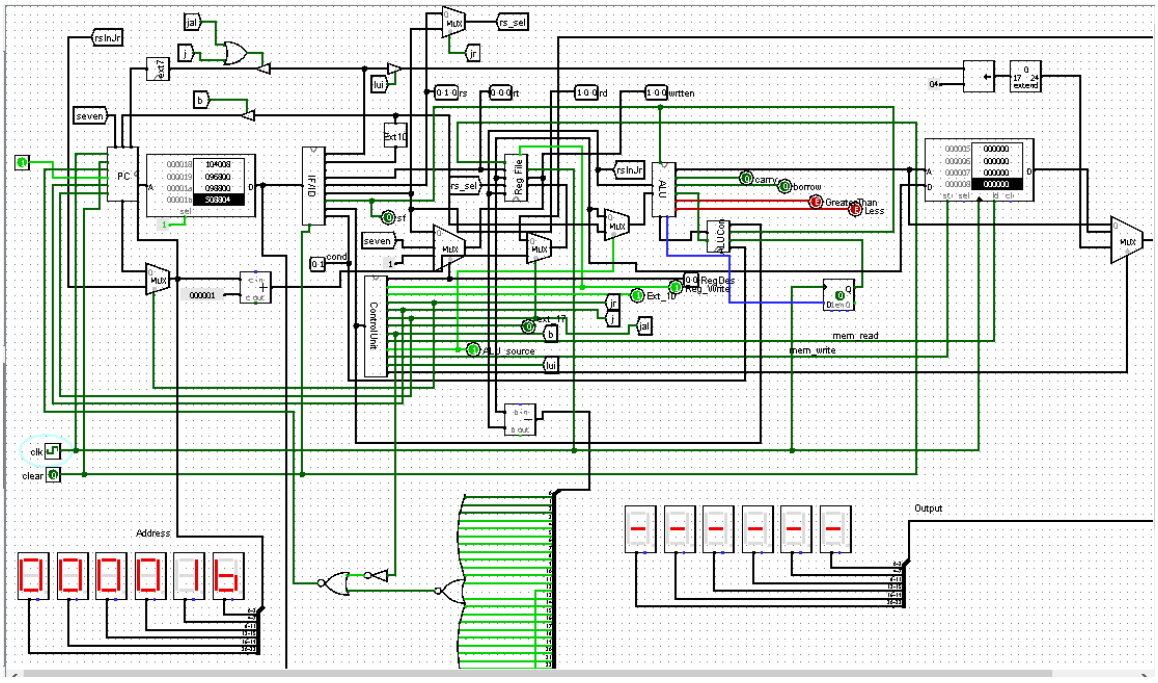


Figure 18: Test case 2

The error:

This may be due to a clocking issue.

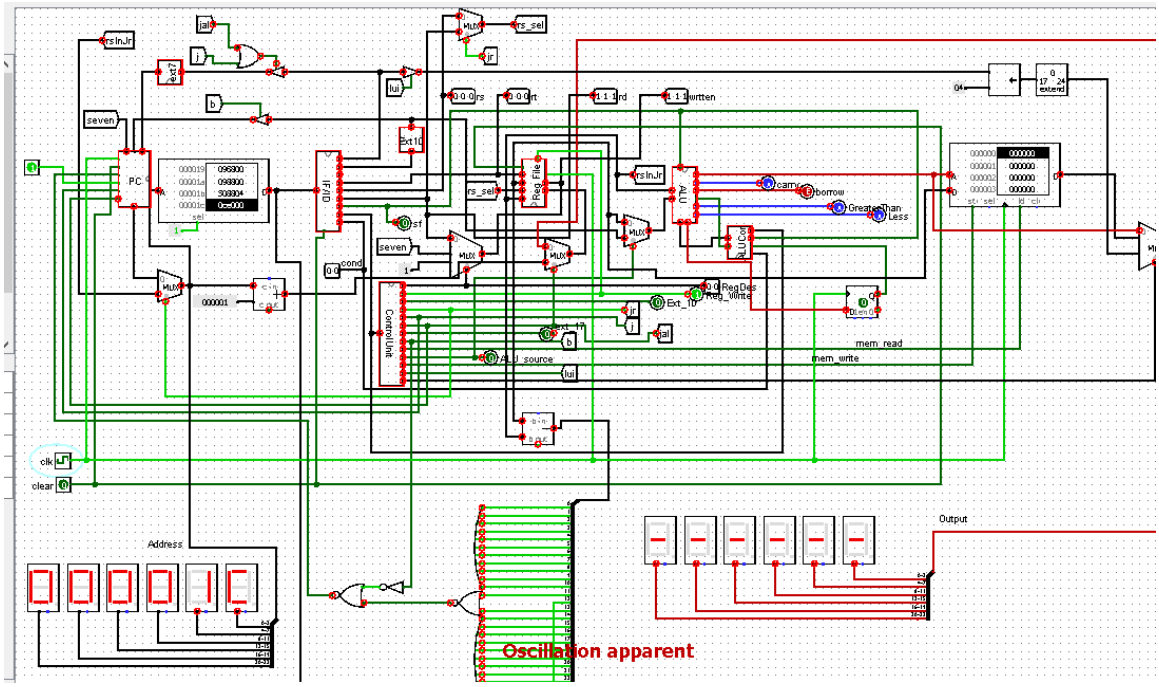


Figure 19: Test case 2 result

Teamwork

As a team, we cooperated together to make this project, first we start with thinking of the design for each component, by sharing ideas and correct each other, then we agreed on the design and put the control unit signals table. After that, one of us built and test PC, IF/ID and register file, one built and test Decode Condition, sub-Condition, add Condition and ALU control unit and the last one built ALU unit, Control unit and the extenders, finally after all units were ready and worked in the right way, we all put our effort, and built the data path together then testing it to make sure that it works as required.