BERZIET UNIVERSITY

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER ENGINEERING

ENCS4370

COMPUTER ARCHITECTURE

INTERFACING LABORATORY

Project 2:

“Single Cycle Datapath”

Prepared by:

Name: Yazan Daibes **Number:** 1180414

Name: Abdallah Afifi **Number:** 1182972

Name: Mohammad Muwafi **Number:** 1180491

Instructor:

Mr. Aziz Qaroush

BIRZEIT

June– 2022

1. Abstract:

The aim of the project is to design, analyze, implement and test a single cycle Datapath 24-bit processor based on a RISC machine using the Logisim tool. A 24-bit RISC processor was built with 8 24-bit general purpose registers, 24-bit width of instruction memory as well as data memory, it supports five different addressing modes, it provides 3 different types of common formats: R-Type, J-Type, I-Type. The 24-bit RISC processor was built in five different stages as following: instruction fetch, instruction decode, instruction execution, instruction memory, and write back, the x stage takes the input from the output of x - 1 stage and then makes its job then propagate the its output to the input of the x + 1 stage as we will see later.

Contents

List Of Figures

List Of Tables

2. Data Path Components:

2.1. Program Counter Unit (PC):

This unit is responsible for determining the next PC that will be used to fetch the next instruction. *Figure* 1 shows the PC unit. As it can be seen, the PC has 4 different inputs and the PCSrc signal will determine which of these inputs will be propagated to the output of the MUX. The first input is the normal next PC, the second is the BTA which stands for branch target address, the third input can be used either for J instruction or for JAL instruction knowing that the only difference between them is that JAL will save the normal next PC to register file, finally, the last input is used for JR instruction.

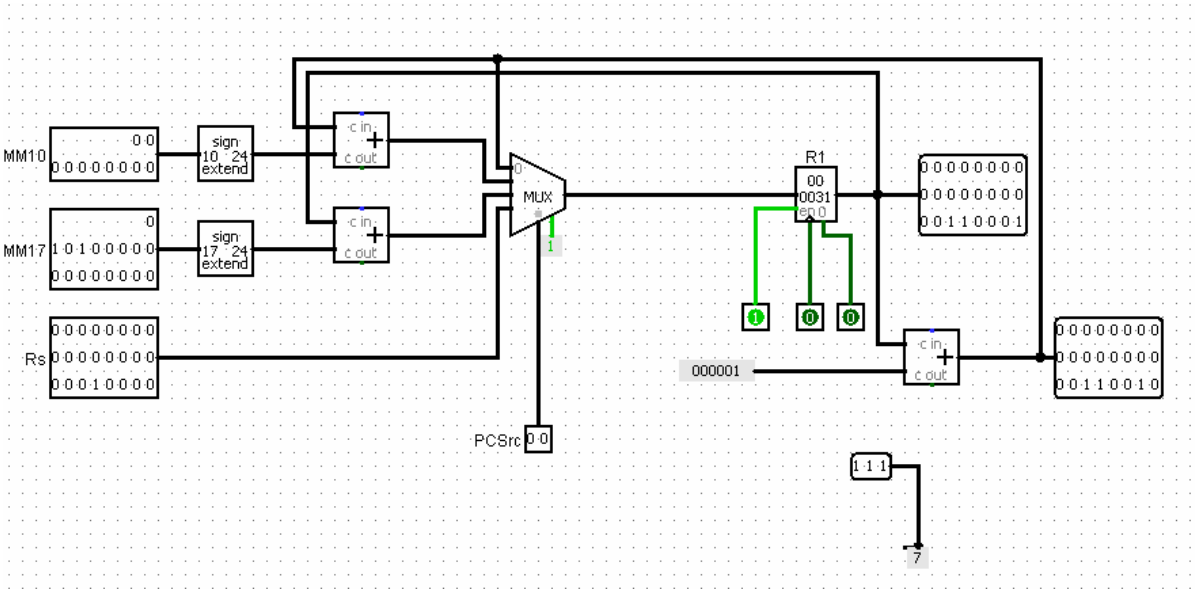

Figure 1: PC Unit

2.2. Register File

As shown in *figure 2*, the register file consists of 3 to 8 decoder, 8 registers and two MUXs. The decoder takes three bits as it's input and three bits are because the register consists of 3 bits from the instruction format. Since the register consists of 3 bits then we have 8 registers ($2^3 = 8$). Register R0 is always zero that's why one of its AND gate input is connected to zero (always disabled). The other registers take the decoded output ANDED with the enable bit and are input to the registers input. Of course, the registers take 24 data bits to be written on it as meaning that the register size is 24-bit. Two 8x1 MUXs were needed, the upper one has 3 bits RA as its selection line and output is 24 bit (BUSA = 24 bit) that were written on the register and selected by the selection line. Similarly, the lower MUX has 3 bits RB as its selection line and output is 24 bit (BUSB = 24 bit) that were written on the register and selected by the selection line.

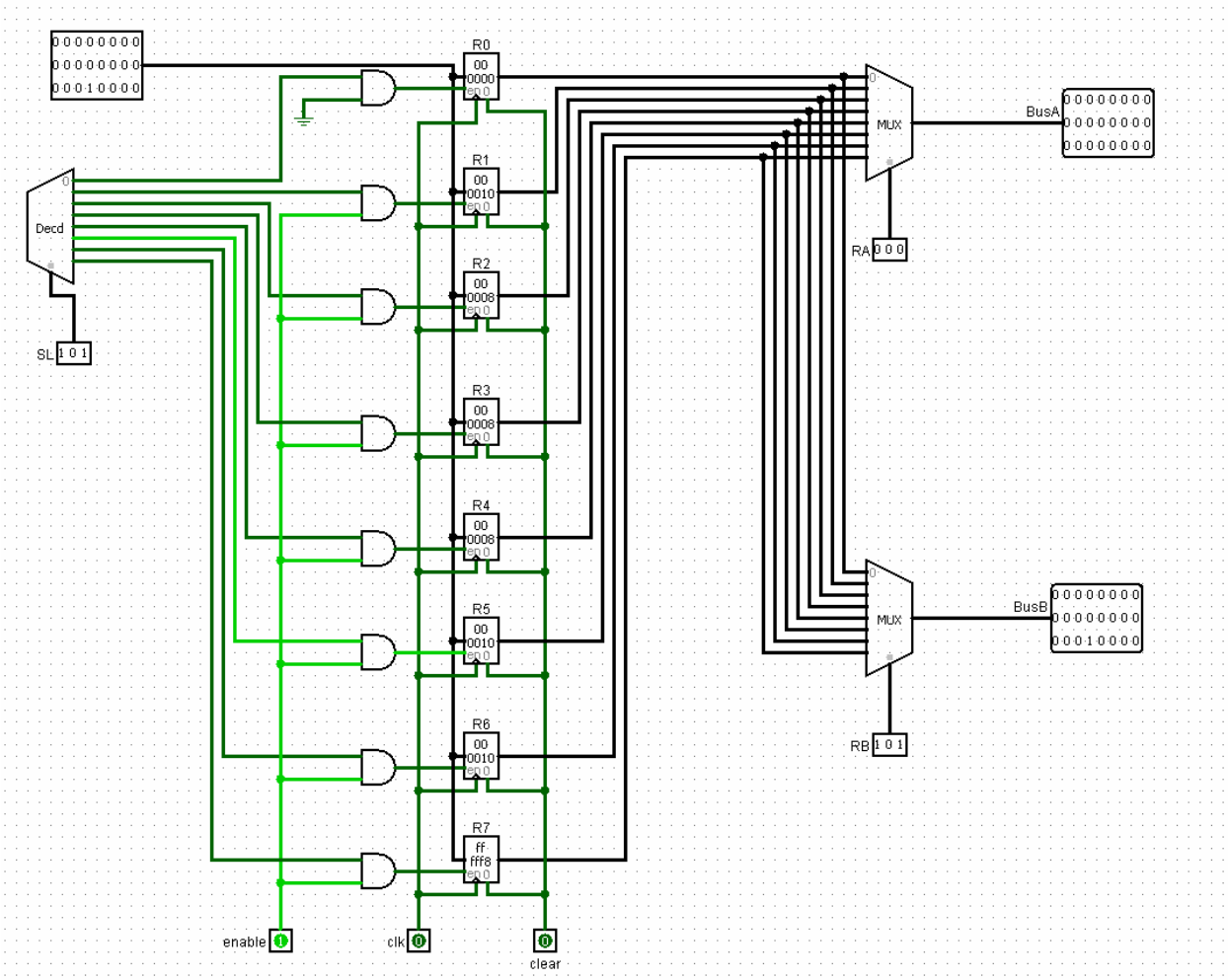

Figure 2: Register File

2.3. Arithmetic-logic Unit (ALU):

The Arithmetic and Logical unit was not hard to build at first. At first sight, we can see that there will only be AND, SUB, ADD operations only, however, when we dive deeper into the project, we see that there will also be SHIFT, CAS. Shifting required a newer hardware to allow for shifting, and since we were adding a shift bit, we used it to allow for CAS. Compare and Swap is initially a subtraction operation. However, if the first operand was not greater than the second operand, then we will output the second operand, otherwise the first operand. Finally, CMP command was implemented, however, we found that the ZF is turned on in case of normal subtraction where operand 1 is less than operand 2, so we needed to add a CMP bit to control when the compare operation will affect the ZF.

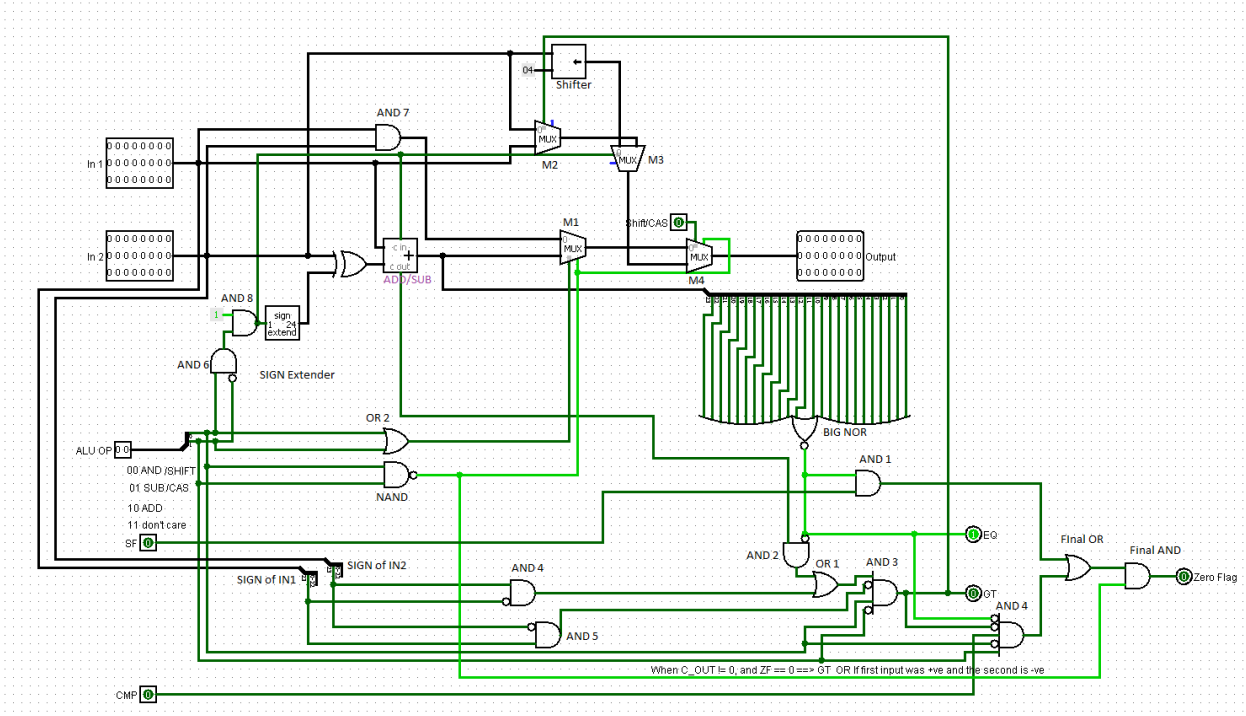

Figure 3: ALU design

In *figure 3* above, we can see the full construction of the ALU, so let's handle the units 1 by 1:

1. ALUOP = 00, Shift/CAS = 0, CMP = X, SF=X: This is implemented directly through a 2 input AND Gate (AND 7), with 24 data lines on each input. The result is then driven to MUX M1 where we choose between it and the output of the Adder/Subtractor. In here the selection line is the OR (OR2) between the 2 bits of ALU OP, so if either is 1, it will choose the output of the Adder/Subtractor. Then it gets driven to MUX M4. In here the selection line is Shift/CAS, where in case of Shift or CAS it will choose the other output, however, in this case it will be 0 and it will choose the output of the AND gate, and finally it is driven to the output. We don't have cases where we have AND with Shift/CAS = 1.
2. ALUOP = 01, Shift/CAS = 0, CMP = 0, SF = ?: In here it is a normal subtraction without affecting the Zero Flag. In AND 6 gate, we take the 2 bits of ALU OP, and if they were 01 (notice the not on the MSB input) it will give out 1. This signal will be extended using the sign extender to 24 bits of 1s, then it will be XORed with In2. This means that when it is 1 (which it is in case of subtraction) it will do 1s compliment, and in case of 0 (in case of addition) it will not compliment the number. Then this sign will enter the C_IN of the Adder in order to now make it a 2's compliment and convert the adder to a subtractor. Then the result passes through MUX M1 and is selected (selection line will be 1) and then through MUX M4 and will also be selected (selection line = 0) to the output. Now, in the same time, this result will be driven to the BIG NOR gate before entering any Muxes. In the BIG NOR gate, it will turn 1 when all the 24 bits are 0s, thus indicating equality. Then this value is given to the EQ output bit as we will need it later in Control unit, and it will also enter AND 1, where with the SF it will determine the ZF. If both are 1 then the ZF will be 1, and 0 otherwise. This is important as the SF is 1 only in case of SUBSF, so it

will always reset the ZF after 1 cycle, unless it was a series of SUBSF where the results are equal (or a CMP which will be discussed later)

3. ALUOP = 10, Shift/CAS = 0, CMP = 0, SF = x: In here, it is normal addition, The 2nd operand will be XORed with 0 so it will not be changed and c_in will be 0. Then it goes through the normal path again.
4. ALUOP = 11, Shift/CAS = X, CMP = X, SF = X: In here it is used in case we needed Stall Cycles. Once it is 11 there will be no output and nothing will be calculated. (That is why when we needed a shift operation, we didn't add it to 11, we needed instead to add a separate bit)
5. ALUOP = 01, Shift/CAS = 1, CMP = 0, SF = X. In here, the operation will be normal subtraction. However, now we will look at a different path. First, the subtraction will be done as usual, but then we will look at the result of GT. Now GT is a signal that turns on when the result is all 0s with no Carry, OR when 1st operand is +ve while the other is -ve (which can be seen in Sign of IN1 and Sign of IN2 that pass through AND 4), AND it is not the opposite (Sign of IN1 is -ve [1] while Sign of IN2 is +ve [0]) AND finally, the last 2 inputs (bottom 2 ones) Of AND 3 is to check that the operation is subtraction, and not any other operation. So now, we got the Greater Than (GT) signal, we will drive it to be the signal for MUX M2. In here we will choose between the 2 inputs based on the signal, if GT is 1 then we choose In1, otherwise In2. Then in MUX M3 we check to see if the operation is subtraction, so that we will see if we were shifting or Comparing and Swapping. Then it is driven to MUX M4 Where it will be selected (selection line is 1) to be driven to the output.
6. ALUOP = 00, Shift/CAS = 1, CMP = X, SF = X: In here it will be shifting. So, the operation will be addition just so that we can differentiate between it and CAS. The actual thing that goes on is that In2 will get shifted by 4 and then it will be chosen at MUX M3 (selection line is 0 now) and then It will be chosen at MUX M4 (selection line is 1) to be driven to the output.
7. ALUOP = 01, Shift/CAS = 0, CMP = 1, SF = X: In here it will be comparison. Compare will set ZF if In1 < In2. So now, we already have GT, and we also have EQ, so we can implement LT by checking if GT and EQ are both 0, and also check that the operation is Subtraction (to be surer). And that is how AND 4 works. The thing that we check at Final AND is to see that ALUOP is not 11, which is a NOP.

2.4. Control Unit:

Here we had a simple design. It was a decoder and based on the instruction we added the corresponding logical gates or hardware.

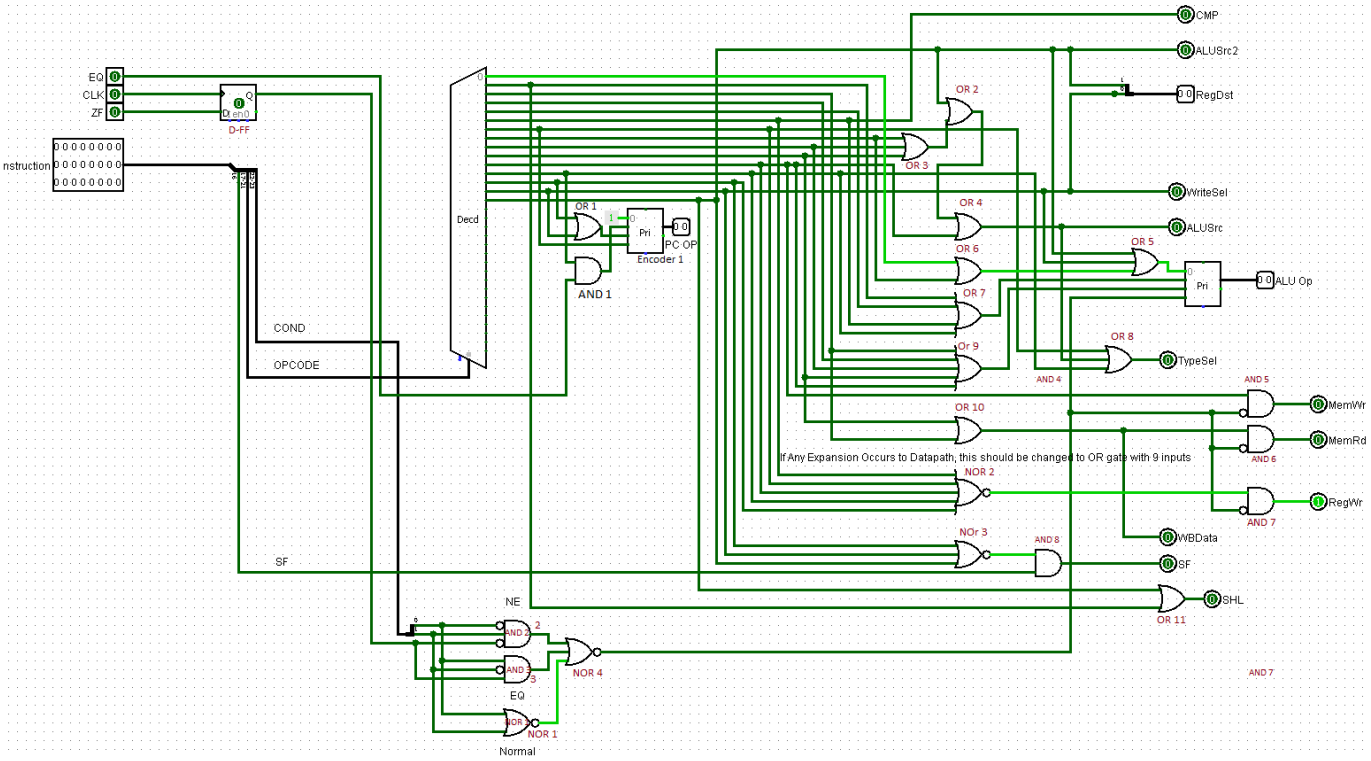

Figure 4:Control Unit

In *figure 4* above, we can see the complete Datapath. It is straight forward, however, I will be going through some of the signals.

1. Encoder 1: in here we choose the PC OP. It is usually 00 (thus we put permanent 1 to input 0). Otherwise, in case of BEQ and the EQ bit was 1 due to subtraction, PC OP will be 01. If it was JAL or a normal J then we go for 10 (thus the need for OR 1), and finally if it was JR, it will be 11.
2. NE/EQ/Normal: Those 3 signals Control whether to execute the command or not based on the Condition parameter that it has. It can have 3 states:
    a. NE: (i.e. ADDNE). in here, we need to check that ZF from last operation is 0 (that is why we needed a D-FF), and that the Condition Bits = 10. If all 3 cases meet, then we output 1
    b. EQ: (i.e. ADDEQ). in here, we need to check that ZF from last operation is 1, and that Condition Bits = 01. If all 3 cases meet, then we output 1.
    c. Normal: (i.e. ADD). In here, we check to see if the Condition is 00, and if it was then we will not care for the ZF as it will always be executed.

    As we can see, there is a NOR gate (NOR 4) that connects with them. This turns 1 when they are all 0, in case such as COND = 10 (NE) and ZF = 1. When it is 1, it will turn RegWr, MemRd and MemWr all to 0, and it will make ALU OP = 11 in order to give 1 stall cycle.
3. ALU OP: it is one of the main signals. It is just implemented by having all operations that use AND or need ALUOP to be 0 in order to do other operations (such as Shift or in cases

of Jump where ALUOP is not needed but in the same time we don't want it to be 11, or 01 and affect the ZF). Same for SUB and ADD operations. We found which ones needed a sub operation and which ones didn't. This will be more specified in Table ?? below.

4. RegDST: this decides where the destination register will be when writing back to the register file. Usually it is RD for R-Type and Rt for I-Type, with some exceptions such as JAL where it will be 7 always (R7).

Table 1: Control Signals

Inst.	Cond	Op	SF	ZF	EQ	CMP	ALUSrc2	RegDst	WriteSel	ALUSrc	ALUOP	TypeSel	MemWr	MemRd	RegWr	WBData	SF	SHL/CAS	PCOP
AND	00	00000	0	X	X	0	0	00	0	0	00	0	0	0	0	0	0	0	00
ANDEQ	01	00000	0	0	X	0	0	00	0	0	11	0	0	0	0	0	0	0	00
ANDNE	10	00000	0	0	X	0	0	00	0	0	00	0	0	0	0	0	0	0	00
ANDEQ	01	00000	0	1	X	0	0	00	0	0	11	0	0	0	0	0	0	0	00
ANDNE	10	00000	0	1	X	0	0	00	0	0	00	0	0	0	0	0	0	0	00
CAS	00	00001	0	X	X	0	0	00	0	0	01	0	0	0	1	0	0	1	00
CASEQ	01	00001	0	0	X	0	0	00	0	0	11	0	0	0	0	0	0	1	00
CASNE	10	00001	0	0	X	0	0	00	0	0	01	0	0	0	1	0	0	1	00
CASEQ	01	00001	0	1	X	0	0	00	0	0	01	0	0	0	1	0	0	1	00
CASNE	10	00001	0	1	X	0	0	00	0	0	11	0	0	0	0	0	0	1	00
LWS	00	00010	0	X	X	0	0	00	0	0	10	0	0	1	1	1	0	0	00
LWSEQ	01	00010	0	0	X	0	0	00	0	0	11	0	0	0	0	1	0	0	00
LWSNE	10	00010	0	0	X	0	0	00	0	0	10	0	0	1	1	1	0	0	00
LWSEQ	01	00010	0	1	X	0	0	00	0	0	10	0	0	1	1	1	0	0	00
LWSNE	10	00010	0	1	X	0	0	00	0	0	11	0	0	0	0	1	0	0	00
ADD	00	00011	0	X	X	0	0	00	0	0	10	0	0	0	1	0	0	0	00
ADDEQ	01	00011	0	0	X	0	0	00	0	0	11	0	0	0	0	0	0	0	0
ADDNE	10	00011	0	0	X	0	0	00	0	0	10	0	0	0	1	0	0	0	00
ADDEQ	01	00011	0	1	X	0	0	00	0	0	10	0	0	0	1	0	0	0	00
ADDNE	10	00011	0	1	X	0	0	00	0	0	11	0	0	0	0	0	0	0	0
SUB	00	00100	0	X	X	0	0	00	0	0	01	0	0	0	1	0	0	0	00
SUBSF	00	00100	1	X	X	0	0	00	0	0	01	0	0	0	1	0	1	0	00
SUBEQ	01	00100	0	0	X	0	0	00	0	0	11	0	0	0	0	0	0	0	00
SUBEQSF	01	00100	1	0	X	0	0	00	0	0	11	0	0	0	0	0	1	0	00
SUBNE	01	00100	0	0	X	0	0	00	0	0	01	0	0	0	1	0	0	0	00
SUBNESF	10	00100	0	0	X	0	0	00	0	0	01	0	0	0	1	0	1	0	00
SUBEQ	10	00100	0	1	X	0	0	00	0	0	01	0	0	0	1	0	0	0	00

SUBNE	10	00100	1	1	X	0	0	00	0	0	11	0	0	0	0	0	0	0	00
CMP	00	00101	0	X	0	1	0	00	0	0	01	0	0	0	0	0	0	0	00
CMP	00	00101	0	X	1	1	0	00	0	0	01	0	0	0	0	0	0	0	00
CMPEQ	01	00101	0	0	0	1	0	00	0	0	11	0	0	0	0	0	0	0	00
CMPNE	10	00101	0	0	0	1	0	00	0	0	01	0	0	0	0	0	0	0	00
CMPEQ	01	00101	0	1	0	1	0	00	0	0	01	0	0	0	0	0	0	0	00
CMPNE	10	00101	0	1	0	1	0	00	0	0	11	0	0	0	0	0	0	0	00
JR	00	00110	0	X	X	0	0	00	0	0	XX	1	0	0	0	0	0	0	11
JREQ	01	00110	0	0	X	0	0	00	0	0	11	1	0	0	0	0	0	0	11
JRNE	10	00110	0	0	X	0	0	00	0	0	XX	1	0	0	0	0	0	0	11
JREQ	01	00110	0	1	X	0	0	00	0	0	11	1	0	0	0	0	0	0	11
JRNE	10	00110	0	1	X	0	0	00	0	0	XX	1	0	0	0	0	0	0	11
ANDI	00	00111	0	X	X	0	0	00	0	1	00	1	0	0	1	0	0	0	00
ANDIEQ	01	00111	0	0	X	0	0	00	0	1	11	1	0	0	0	0	0	0	00
ANDINE	10	00111	0	0	X	0	0	00	0	1	00	1	0	0	1	0	0	0	00
ANDIEQ	01	00111	0	1	X	0	0	00	0	1	00	1	0	0	1	0	0	0	00
ANDINE	10	00111	0	1	X	0	0	00	0	1	11	1	0	0	0	0	0	0	00
ADDI	00	01000	0	X	X	0	0	00	0	1	10	1	0	0	1	0	0	0	00
ADDIEQ	01	01000	0	0	X	0	0	00	0	1	11	1	0	0	0	0	0	0	00
ADDINE	10	01000	0	0	X	0	0	00	0	1	10	1	0	0	1	0	0	0	00
ADDIEQ	01	01000	0	1	X	0	0	00	0	1	11	1	0	0	0	0	0	0	00
ADDINE	10	01000	0	1	X	0	0	00	0	1	10	1	0	0	1	0	0	0	00
LW	00	01001	0	X	X	0	0	00	0	1	10	1	0	1	1	1	0	0	00
LWEQ	01	01001	0	0	X	0	0	00	0	1	11	1	0	0	0	1	0	0	00
LWNE	10	01001	0	0	X	0	0	00	0	1	10	1	0	1	1	1	0	0	00
LWEQ	01	01001	0	1	X	0	0	00	0	1	10	1	0	1	1	1	0	0	00
LWNE	10	01001	0	1	X	0	0	00	0	1	11	1	0	0	0	1	0	0	00
SW	00	01010	0	X	X	0	0	00	0	1	10	1	1	0	0	0	0	0	00
SWEQ	01	01010	0	0	X	0	0	00	0	1	11	1	0	0	0	0	0	0	00
SWNE	10	01010	0	0	X	0	0	00	0	1	10	1	1	0	0	0	0	0	00
SWEQ	01	01010	0	1	X	0	0	00	0	1	10	1	1	0	0	0	0	0	00
SWNE	10	01010	0	1	X	0	0	00	0	1	11	1	0	0	0	0	0	0	00
BEQ	00	01011	0	X	1	0	0	00	0	0	01	1	0	0	0	0	0	0	01
BEQ	00	01011	0	X	0	0	0	00	0	0	01	1	0	0	0	0	0	0	00
BEQEQ	01	01011	0	0	X	0	0	00	0	0	11	1	0	0	0	0	0	0	00

BEQNE	10	01011	0	0	X	0	0	00	0	0	01	1	0	0	0	0	0	0	00
BEQEQ	01	01011	0	1	X	0	0	00	0	0	01	1	0	0	0	0	0	0	00
BEQNE	10	01011	0	1	X	0	0	00	0	0	11	1	0	0	0	0	0	0	00
J	00	01100	0	X	X	0	0	00	0	0	XX	0	0	0	0	0	0	0	10
JEQ	01	01100	0	0	X	0	0	00	0	0	11	0	0	0	0	0	0	0	10
JNE	10	01100	0	0	X	0	0	00	0	0	XX	0	0	0	0	0	0	0	10
JEQ	01	01100	0	1	X	0	0	00	0	0	XX	0	0	0	0	0	0	0	10
JNE	10	01100	0	1	X	0	0	00	0	0	11	0	0	0	0	0	0	0	10
JAL	00	01101	0	X	X	0	0	01	1	0	00	0	0	0	1	0	0	0	10
JALEQ	01	01101	0	0	X	0	0	01	1	0	11	0	0	0	0	0	0	0	10
JALNE	10	01101	0	0	X	0	0	01	1	0	00	0	0	0	1	0	0	0	10
JALEQ	01	01101	0	1	X	0	0	01	1	0	00	0	0	0	1	0	0	0	10
JALNE	10	01101	0	1	X	0	0	01	1	0	11	0	0	0	0	0	0	0	10
LUI	00	01110	0	X	X	0	1	10	0	1	00	1	0	0	1	0	0	1	00
LUIEQ	01	01110	0	0	X	0	1	10	0	1	11	1	0	0	0	0	0	1	00
LUINE	10	01110	0	0	X	0	1	10	0	1	00	1	0	0	1	0	0	1	00
LUIEQ	01	01110	0	1	X	0	1	10	0	1	00	1	0	0	1	0	0	1	00
LUINE	10	01110	0	1	X	0	1	10	0	1	11	1	0	0	0	0	0	1	00

3. Single Cycle Datapath:

All the explained components (PC, Register File, ALU, Control Unit) in section 2 were connected to each other as shown in *figure 5*. In addition to adding instruction memory (ROM) between the PC and Register File and data memory (RAM) between the ALU and the last MUX on the right. The data path is clocked and the instruction should be loaded int the instruction memory. The PC after it's enabled will start moving inside the instruction memory depending on the instruction type (i.e., jump, add…etc.). A component called sign extend (there was no need for unsigned extension) is added in which it extends the 17 bits to 24 and the other extender extends the 10 bits to 24 bits. MUXs were added to control the entered logic when needed, also a spitter was added to split the bits into desired length.

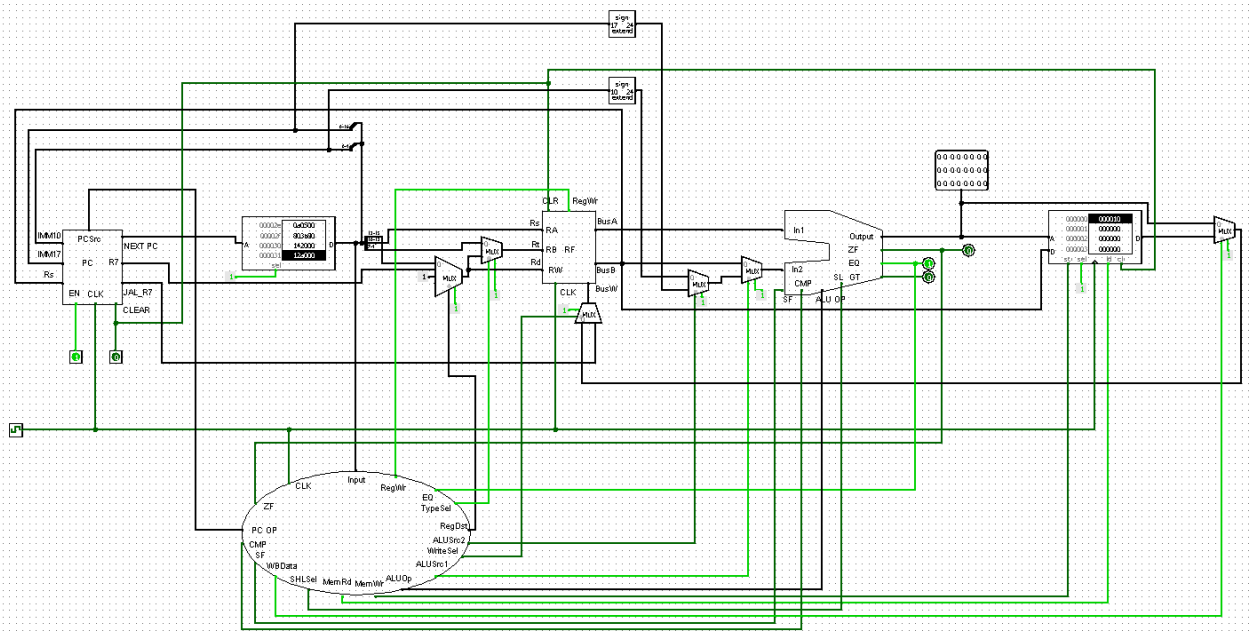

Figure 5: Single Cycle Datapath

4. Testing and Results:

In order to test the system, we first needed to convert the instructions into machine code, which usually takes time, however, we found a custom compiler online where we defined the rules and based on them we created the tests. The site is called [customasm Web (hlorenzi.github.io)] and for the *#ruledef* you can look it up in Appendix A.

Test Case:

```
================================================================

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 = 24

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 = 24, r2 = 8, cmp ==> GT ==> ZF goes from 0 to 1

ADDNE r1,r6,r7 ;shouldn't work

=====================================================================
```

The test case above as we can see is almost universal. We tried jumping and returning to an address, Tried the EQ (condition field) instructions and even instructions like CMP and LW and SW operations. Next to each command we wrote what is expected to be.

Here is the binary representation in the Instruction Memory shown in figure ??. We can see that the instructions are spread accordingly depending on the jump operation.

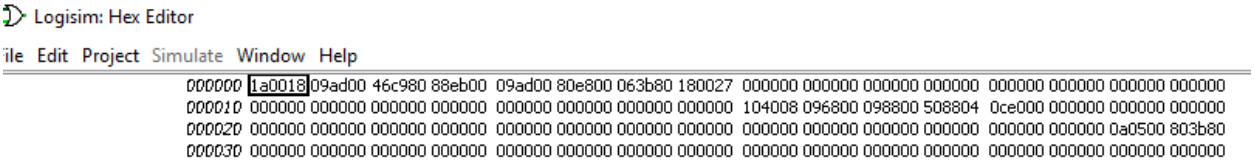

Figure 6:Binary Representation in Instruction Memory

As for the result, here is the register file with the Data memory:

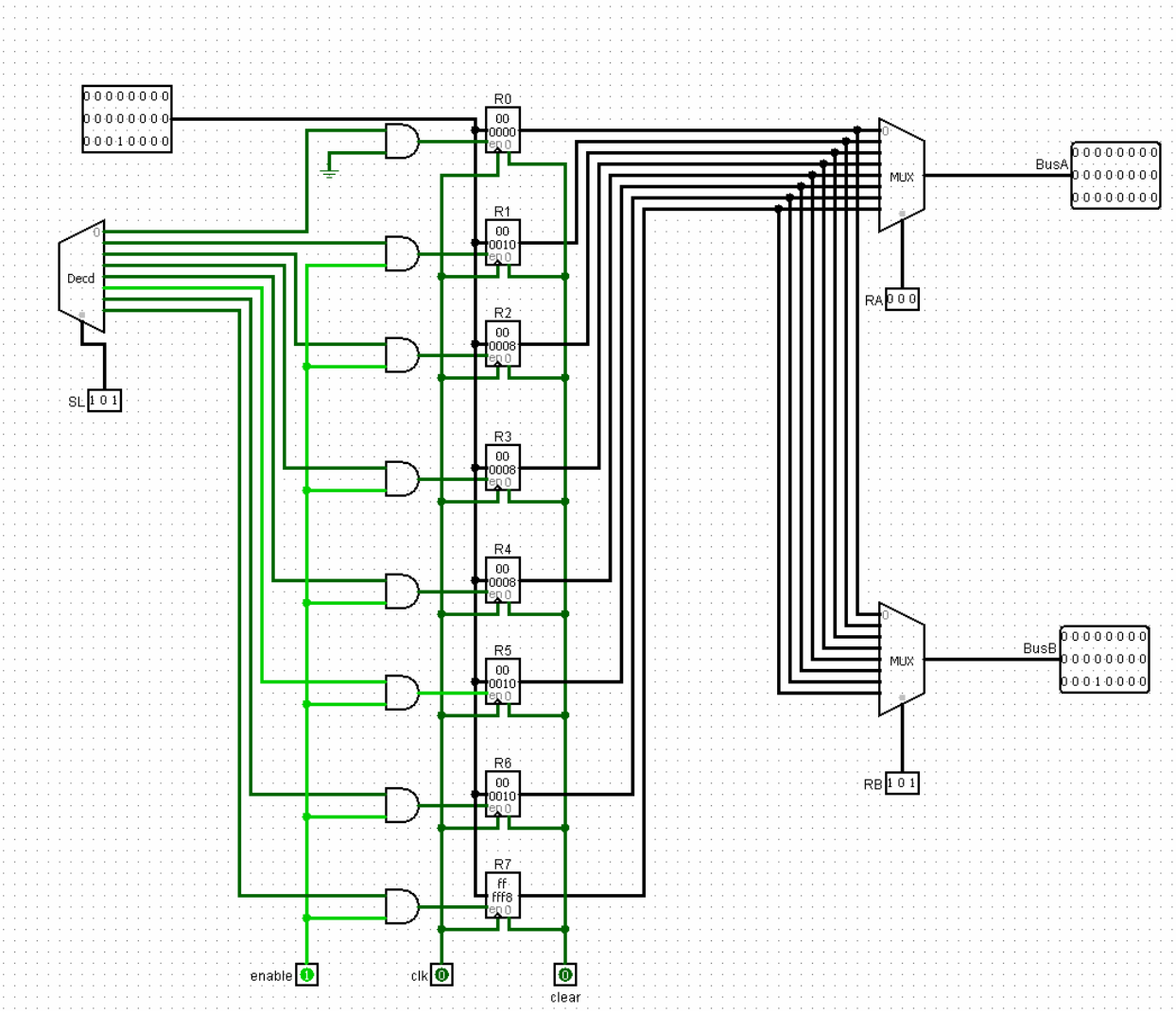

Figure 7:Register File Content

5. Conclusion

In this project we have learned extensively about the MIPS-RISC structure and its implementation. We have succeeded in implementing a single cycle 24-bit processor and testing it using the Logisim simulator. As a future plan, this project encourages us to dive more in computer architecture techniques and how to optimize our processor to pipelined one which was expected to be built but due to the very big load in this semester, we did not have any extra time to complete our design as expected so we hope to take this issue into consideration when grading our work.

6. Appendix

```
#ruledef
{
    ;                   R TYPE
    AND r{reg_num1}, r{reg_num2},r{reg_num3} => 0b00000000 @ reg_num1`3 @ reg_num2`3 @ reg_num3`3 @ 0b0000000
    ANDEQ r{reg_num1}, r{reg_num2},r{reg_num3} => 0b01000000 @ reg_num1`3 @ reg_num2`3 @ reg_num3`3 @ 0b0000000
    ANDNE r{reg_num1}, r{reg_num2},r{reg_num3} => 0b10000000 @ reg_num1`3 @ reg_num2`3 @ reg_num3`3 @ 0b0000000

    CAS r{reg_num1}, r{reg_num2},r{reg_num3} => 0b00000010 @ reg_num1`3 @ reg_num2`3 @ reg_num3`3 @ 0b0000000
    CASEQ r{reg_num1}, r{reg_num2},r{reg_num3} => 0b01000010 @ reg_num1`3 @ reg_num2`3 @ reg_num3`3 @ 0b0000000
    CASNE r{reg_num1}, r{reg_num2},r{reg_num3} => 0b10000010 @ reg_num1`3 @ reg_num2`3 @ reg_num3`3 @ 0b0000000

    LWS r{reg_num1}, r{reg_num2},r{reg_num3} => 0b00000100 @ reg_num1`3 @ reg_num2`3 @ reg_num3`3 @ 0b0000000
    LWSEQ r{reg_num1}, r{reg_num2},r{reg_num3} => 0b01000100 @ reg_num1`3 @ reg_num2`3 @ reg_num3`3 @ 0b0000000
    LWSNE r{reg_num1}, r{reg_num2},r{reg_num3} => 0b10000100 @ reg_num1`3 @ reg_num2`3 @ reg_num3`3 @ 0b0000000

    ADD r{reg_num1}, r{reg_num2},r{reg_num3} => 0b00000110 @ reg_num1`3 @ reg_num2`3 @ reg_num3`3 @ 0b0000000
    ADDEQ r{reg_num1}, r{reg_num2},r{reg_num3} => 0b01000110 @ reg_num1`3 @ reg_num2`3 @ reg_num3`3 @ 0b0000000
    ADDNE r{reg_num1}, r{reg_num2},r{reg_num3} => 0b10000110 @ reg_num1`3 @ reg_num2`3 @ reg_num3`3 @ 0b0000000

    SUB r{reg_num1}, r{reg_num2},r{reg_num3} => 0b00001000 @ reg_num1`3 @ reg_num2`3 @ reg_num3`3 @ 0b0000000
    SUBEQ r{reg_num1}, r{reg_num2},r{reg_num3} => 0b01001000 @ reg_num1`3 @ reg_num2`3 @ reg_num3`3 @ 0b0000000
    SUBNE r{reg_num1}, r{reg_num2},r{reg_num3} => 0b10001000 @ reg_num1`3 @ reg_num2`3 @ reg_num3`3 @ 0b0000000
```

SUBSF r{reg_num1}, r{reg_num2},r{reg_num3} => 0b00001001 @ reg_num1`3 @ reg_num2`3 @ reg_num3`3 @ 0b0000000

SUBEQSF r{reg_num1}, r{reg_num2},r{reg_num3} => 0b01001001 @ reg_num1`3 @ reg_num2`3 @ reg_num3`3 @ 0b0000000

SUBNESF r{reg_num1}, r{reg_num2},r{reg_num3} => 0b10001001 @ reg_num1`3 @ reg_num2`3 @ reg_num3`3 @ 0b0000000

CMP r{reg_num1}, r{reg_num2} => 0b00001010 @ 0b000 @ reg_num1`3 @ reg_num2`3 @ 0b0000000

CMPEQ r{reg_num1}, r{reg_num2} => 0b01001010 @ 0b000 @ reg_num1`3 @ reg_num2`3 @ 0b0000000

CMPNE r{reg_num1}, r{reg_num2} => 0b10001010 @ 0b000 @ reg_num1`3 @ reg_num2`3 @ 0b0000000

JR r{reg_num1} => 0b00001100 @ reg_num1`3 @ 0b0000000000000

JREQ r{reg_num1} => 0b01001100 @ reg_num1`3 @ 0b0000000000000

JRNE r{reg_num1} => 0b10001100 @ reg_num1`3 @ 0b0000000000000

; I TYPE

ANDI r{reg_num1}, r{reg_num2},{imm} => 0b00001110 @ reg_num1`3 @ reg_num2`3 @ imm`10

ANDIEQ r{reg_num1}, r{reg_num2},{imm} => 0b01001110 @ reg_num1`3 @ reg_num2`3 @ imm`10

ANDINE r{reg_num1}, r{reg_num2},{imm} => 0b10001110 @ reg_num1`3 @ reg_num2`3 @ imm`10

ADDI r{reg_num1}, r{reg_num2},{imm} => 0b00010000 @ reg_num1`3 @ reg_num2`3 @ imm`10

ADDIEQ r{reg_num1}, r{reg_num2},{imm} => 0b01010000 @ reg_num1`3 @ reg_num2`3 @ imm`10

ADDINE r{reg_num1}, r{reg_num2},{imm} => 0b10010000 @ reg_num1`3 @ reg_num2`3 @ imm`10

LW r{reg_num1}, r{reg_num2},{imm} => 0b00010010 @ reg_num1`3 @ reg_num2`3 @ imm`10

LWEQ r{reg_num1}, r{reg_num2},{imm} => 0b01010010 @ reg_num1`3 @ reg_num2`3 @ imm`10

LWNE r{reg_num1}, r{reg_num2},{imm} => 0b10010010 @ reg_num1`3 @ reg_num2`3 @ imm`10

SW r{reg_num1}, r{reg_num2},{imm} => 0b00010100 @ reg_num1`3 @ reg_num2`3 @ imm`10

SWEQ r{reg_num1}, r{reg_num2},{imm} => 0b01010100 @ reg_num1`3 @ reg_num2`3 @ imm`10

SWNE r{reg_num1}, r{reg_num2},{imm} => 0b10010100 @ reg_num1`3 @ reg_num2`3 @ imm`10

BEQ r{reg_num1}, r{reg_num2},{imm} => 0b00010110 @ reg_num1`3 @ reg_num2`3 @ imm`10

BEQEQ r{reg_num1}, r{reg_num2},{imm} => 0b01010110 @ reg_num1`3 @ reg_num2`3 @ imm`10

BEQNE r{reg_num1}, r{reg_num2},{imm} => 0b10010110 @ reg_num1`3 @ reg_num2`3 @ imm`10

; J TYPE

J {imm} => 0b0001100 @ imm`17

JEQ {imm} => 0b0101100 @ imm`17

JNE {imm} => 0b1001100 @ imm`17

JAL {imm} => 0b0001101 @ imm`17

JALEQ {imm} => 0b0101101 @ imm`17

JALNE {imm} => 0b1001101 @ imm`17

LUI {imm} => 0b0001110 @ imm`17

LUIEQ {imm} => 0b0101110 @ imm`17

LUINE {imm} => 0b1001110 @ imm`17

}