Birzeit University

Electrical and Computer Systems Engineering

Department

Computer Architecture ENCS437

Project 1 Report

Single Cycle MIPS Processor

Student name: Ghassan Syam Student No.: 1131486

Student name: Abdallatif Sulaiman Student No.: 1131090

Student name: Hussein Dahir Student No.: 1131138

Instructor name: Dr. Emad Hamadah

# Table of Contents

**Abstract……………………………………………………………………………………………………………….……**1

**System Description (Design and Implementation)…….………………………………………………**2

Arithmetic Logical Unit.………………………………………………………………………………….………………2

Control Unit………………………………...........……………………………………………………………..2

Register File…………………..............................................................……………..2

NextPC…………………………...............................................................…2

ALUOP…………………….................................................………..2

Final Datapath…….................……………………………...2

**Simulation and testing………………………………………………………………………………………………**3

**Conclusion……………………...………………………………………………………………..……………………..**4

Abstract:

In this project we used Logisim Program to design a single cycle MIPS processor, with instruction size of 32 bits.

System Description (Design and Implementation):

The single cycle MIPS Processor that we have designed consists of 6 blocks as follow:

* Arithmetic Logical Unit ( 32 bit ALU )
* Register File ( 32 bit )
* Control Unit
* ALUOP
* NextPC
* Memory ( ROM & RAM )

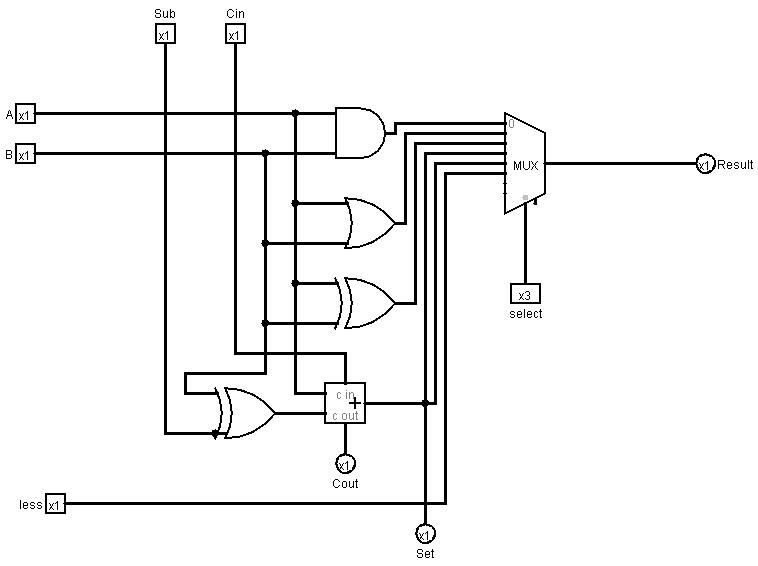
Each one of these blocks will be used in designing the Whole DataPath to form the final picture of the single MIPS Processor.

* **Arithmetic Logical Unit ( 32 bit ):**

In Order to get the full design for 32 bit ALU, we built a 1-bit ALU as the base ALU, then we built 8-bit ALU from 8 blocks of 1-bit ALU, and we used 4 blocks of 8-bit ALU to get the 32-bit ALU.

The ALU performs six operations: (AND, OR, XOR, ADD & SUB, SLT). We used a MUX to multiplex the needed operation according to the 3-bit select segment, and we have the enable bin which is value is true to enable the MUX therefore is used to enable the ALU. We have two inputs (A, B), and the Result with 32 bit for both (inputs and outputs).

**Our 1-bit ALU Design:**



The 3-bit select segment in the MUX determines the operation that will be performed:

|  |  |
| --- | --- |
| Select Segment | Operation |
| 000 | AND |
| 001 | OR |
| 010 | XOR |
| 011 | ADD |
| 100 | SUB |
| 101 | SLT |

**SLT Operation:** It takes the two inputs (A-first input , B-second input), and subtracts them from each other, then observes the result we got, especially the MSB, so if (MSB = 0 ), then A > B and the ( SET = 0 as an indication that A > B ), but if ( MSB = 1 ) then A < B and the ( SET = 1 as an indication that A < B ). We should make the input (SUB = 1) to perform the subtraction operation between the inputs.

**32-bit ALU:**

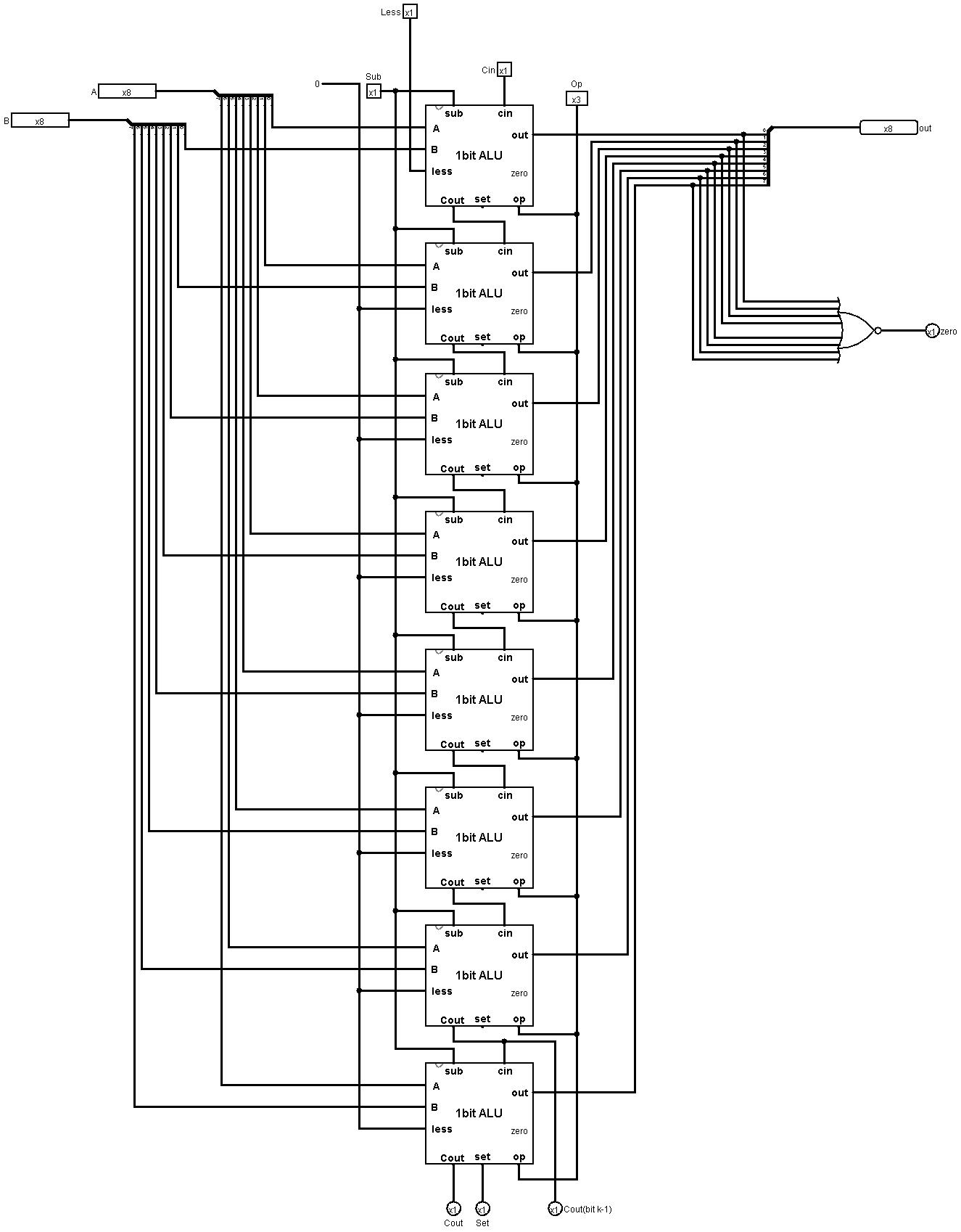
We have implemented a hardware circuit in 32-bit ALU to deal with SLT operation as follow:

If we have two numbers A and B, and the subtraction result of (A - B) caused an overflow, then the SLT result is wrong, so this circuit handled this problem and correct the answer.

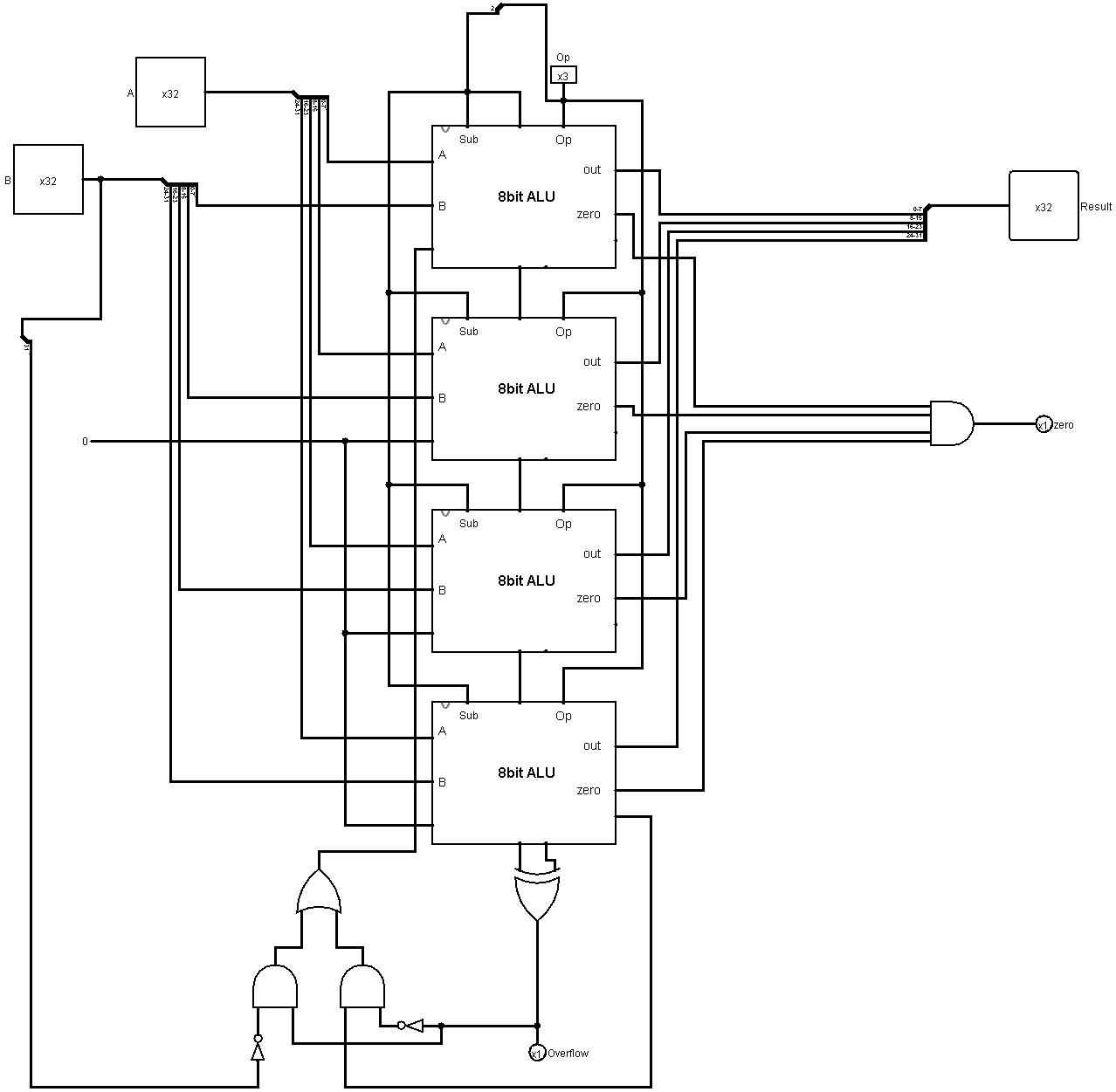
For example:

Let A= 8FFFFFFF (negative number), B= 7FFFFFFF (positive number)

A – B means that it is actually summation of two negative numbers and the result is positive, so overflow occurred, so we have this circuit to handle this overflow and make the SLT operation works well.

**8-bit ALU Design:**

**32-bit ALU Design:**



* **Register File:**

We have designed the Register File (32 × 32-bit registers) with a simple basic design as it shown in the slides controlling the output by multiplexing the register values, and using a decoder to enable the write of new data based on the address of the destination and the RegWrite control signal.

**Registers Block:**

We have implemented this Block as it shown in the slides for choosing the register you want to write it on using decoder, it takes the address that you want to write it on, then it enables only the register you called, RegWrite control signal must be equal to 1 for the AND gate, if RegWrite was equal to 1 , then the desired register will work and take the input from the control signal (BusW 32-bit ), but if it was equal to 0 , then you can’t write to any register, instead of it can be read any register you want.

Another control signal is important which is (ResetAllRegs), it connected to the (Reset pin) for all registers to make them zero, (CLK control signal) must be connected to the clock pin for all registers, so nothing will work unless CLK is enabled for all registers.

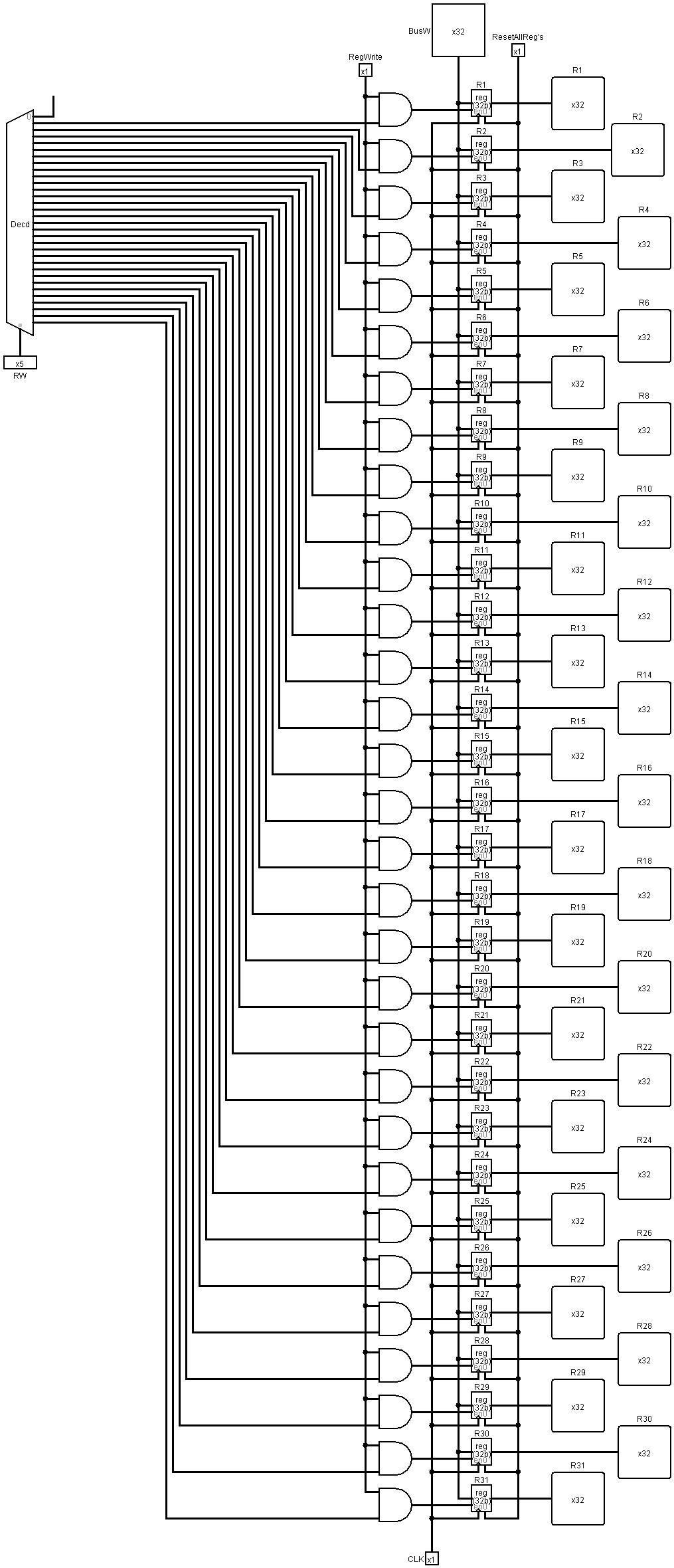
Register 0 always equal to “zero”, so we can’t write it on.

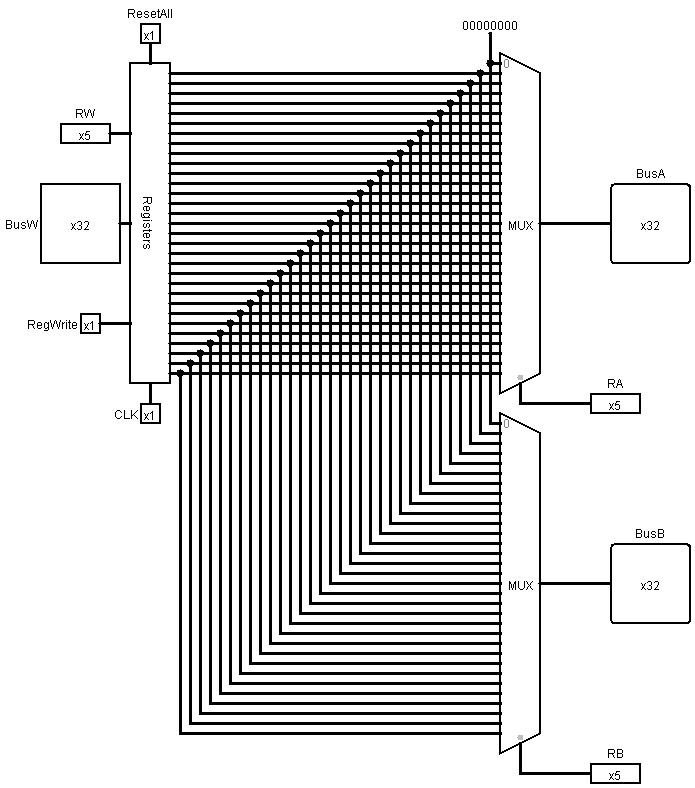
**Register File:**

We used (Registers Block) to build Register File, Registers Block represent as an input for the MUXES, then the upper MUX choose the registers that it must pass to BusA according to the address in RA control signal, same way in getting BusB according to the address in RB control signal.

We have these control signal again (RegWrite, CLK, ResetAll, BusW) from the Registers Block that we have explained before.

**Registers Block:**

****

**Register File:**

* **Memory:**

We have used two types of memory in our design as follow:

**ROM:**

It represents the instruction memory that needs only provide read access, because datapath does not write instruction, also it behaves like a combinational logic for read, address which is input from PC selects instruction after access time and then take them to the output of the ROM.

**RAM:**

It represents the actual memory (Data Memory) that used for load and store data, it takes the address from the output of the ALU, it consist of many signals such as:

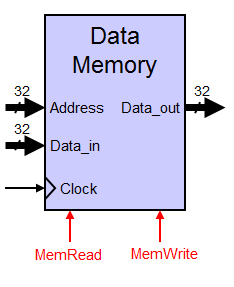
**MemRead**: enables output on Data\_out

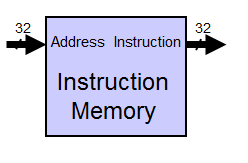
Address selects the word to put on Data\_out

**MemWrite**: enables writing of Data\_in

Address selects the memory word to be written

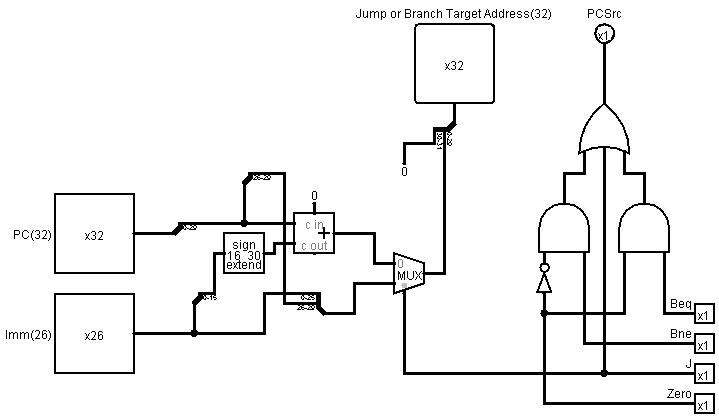
**CLK**: It synchronizes the write operation.





* **NextPC:**

We considered the design in the lecture slide to build the NextPC Block as follow:



PCSrc means that from where the (NextPC Address) should be taken, but here it depends on three control signals, (Beq, Bne, Jump) for three cases:

* If Beq was set and the result of the ALU is zero, then PCSrc will show.
* If Bne was set and the result of the ALU isn’t zero, then PCSrc will show.
* Also if it is (J), PCSrc will show

All these three cases are an input to OR gate in order to get the PCSrc.

On the left side of the figure, here we designed how to compute NextPC Address, we have a MUX that decides what to take to compute the NextPC Address.

1. If it is a Jump, then Jump target address: upper 4 bits of PC are concatenated with Imm26

PCSrc = J + (Beq **.** Zero) + (Bne **.** (Zero)`)

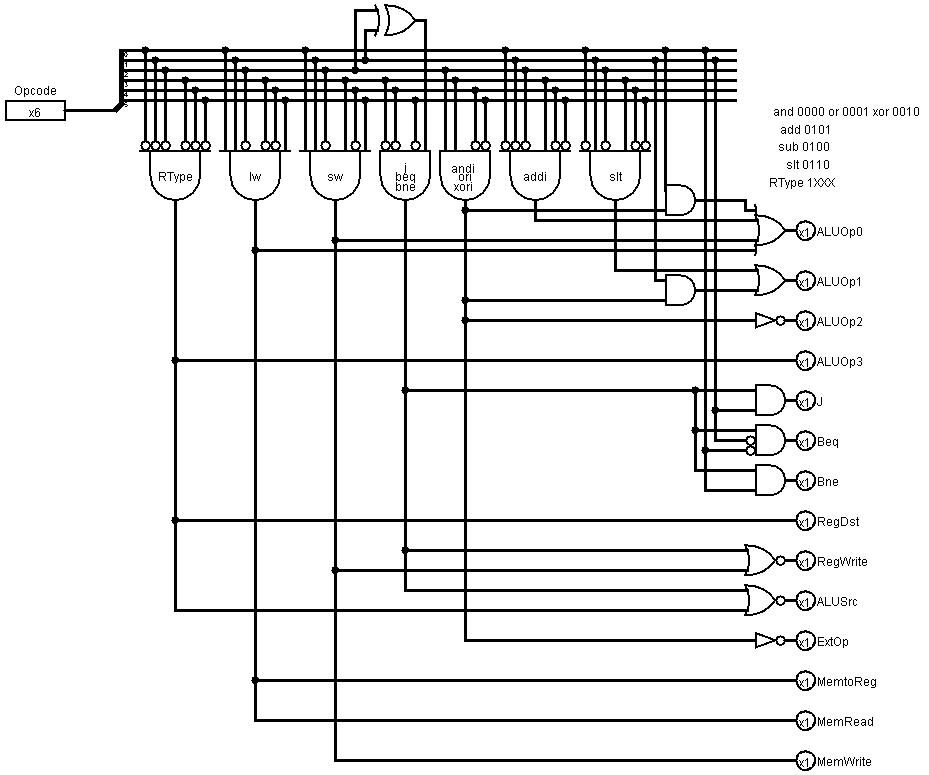
1. If it is a Branch, then Branch target Address: (PC + 4) + Imm(16).

Note: Immediate can be positive or negative.

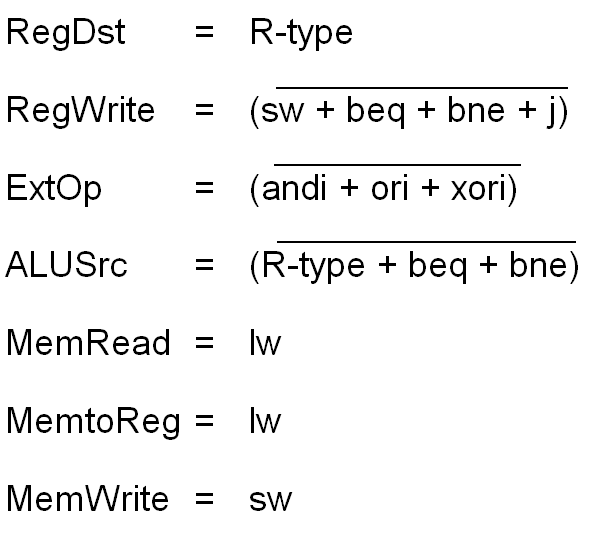
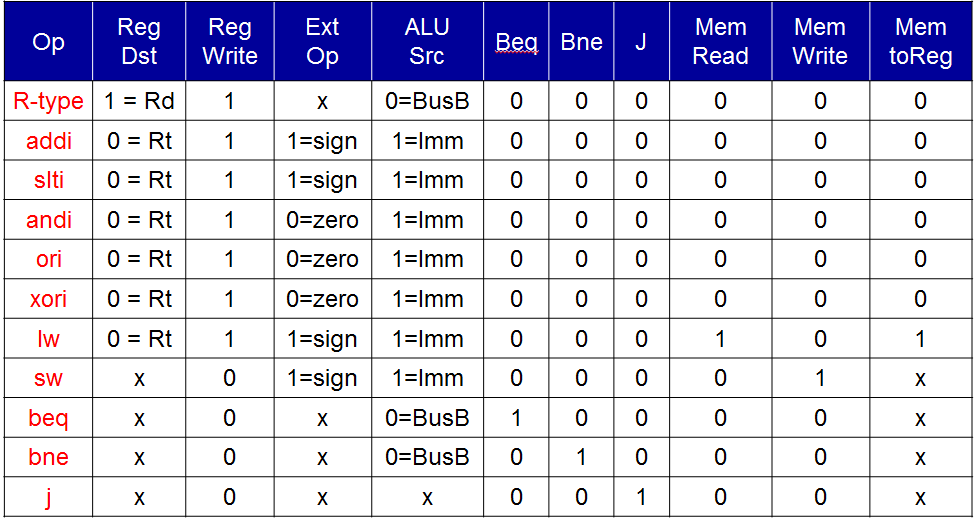
* **Control Unit and ALUOP:**

We have designed the control unit and ALUOP according to the following table and equations for the necessary signals in the systems, and from it built the Boolean functions that use the Instruction bits as input and produce the needed signals. We then implemented the Boolean functions using the logic gates found in Logisim. Following are the truth table, and the logic circuit of the Control unit.

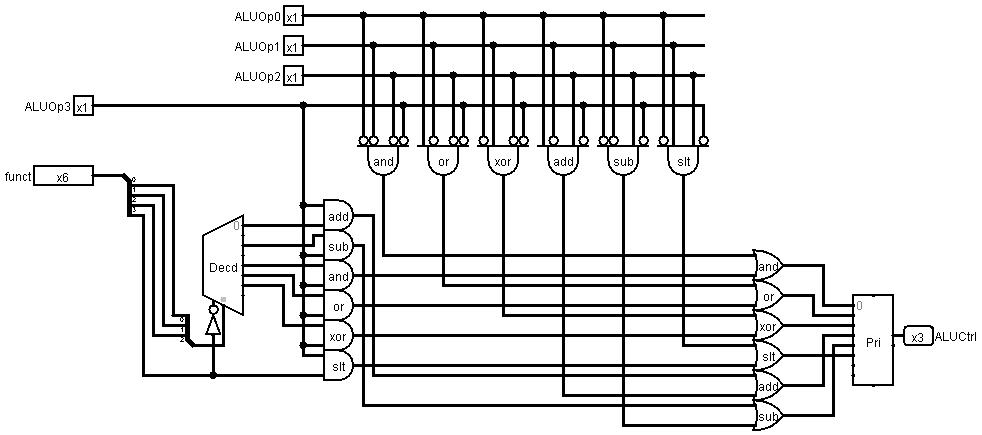
**Control Unit:**

****

On the right side of the figure, Opcode determines the type of the instruction, and only on the AND gates will turn on, then the signals on the left side of the figure turn on/off according to the type of the instruction. We consider the following table to construct the above design:



**ALUOP:**

****

We have 4 signal ALUOP [0-3], that minimize the opcode to something simple, and the ALUOP output determines according to the type of the instruction as follow:

**Instruction ALUOP**

R-Type 1xxx

ADD 0101

SUB 0100

SLT 0110

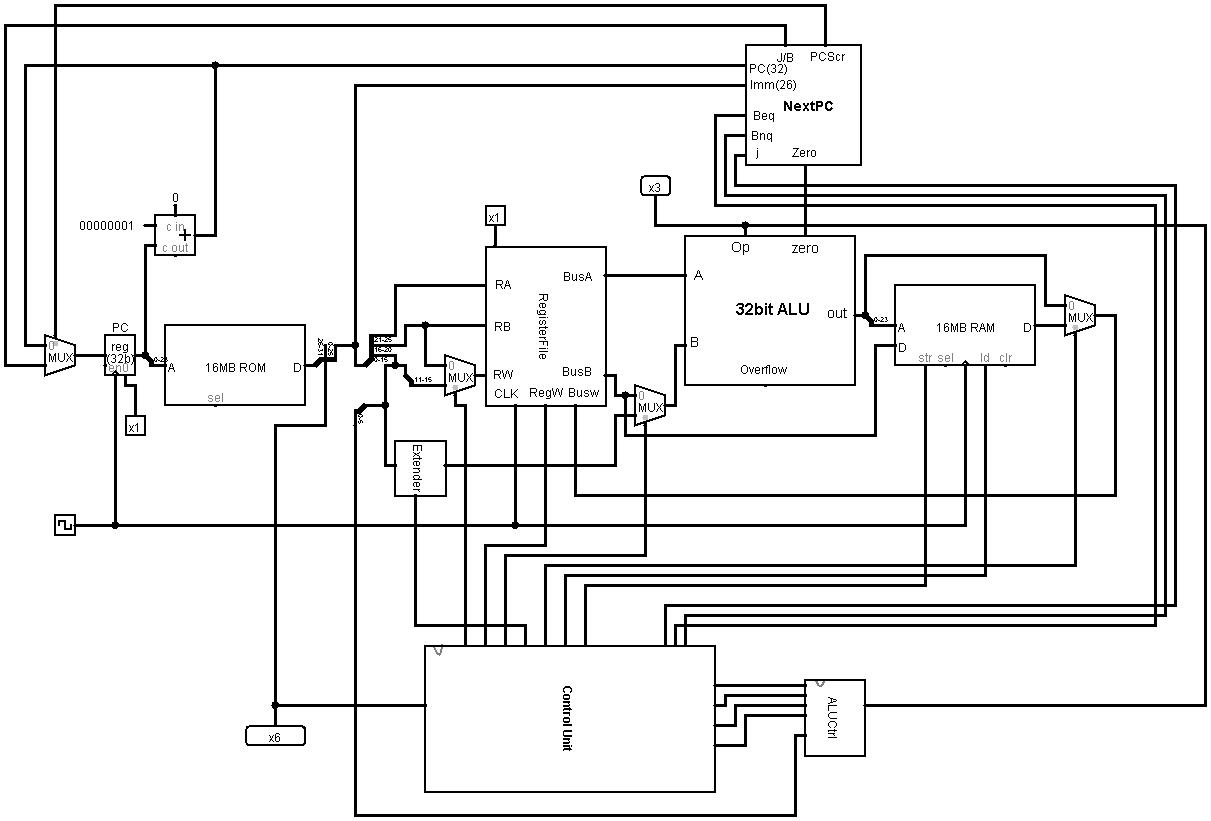
AND 0000/0001/0010

For example, if ALUOP3 = 0, all the remaining signal will be on and the decoder too, then it outputs the first 3-bit, after that it will determine the output code for ALUCtrl according to the encoder.

* We have used a table to design the ALUOP:

|  |  |  |  |
| --- | --- | --- | --- |
| instruction | op | fun | ALUOP |
| add | 0 | 20 | 0000 add 011 |
| sub | 0 | 22 | 0010 sub 100 |
| and | 0 | 24 | 0100 and 000 |
| or | 0 | 25 | 0101 or 001 |
| xor | 0 | 26 | 0110 xor 010 |
| slt | 0 | 2A | 1010 slt 101 |
| addi | 08 | x | add 011 |
| slti | 0A | x | slt 101 |
| andi | 0c | x | and 000 |
| ori | 0D | x | or 001 |
| xori | 0e | x | xor 010 |
| beq | 04 | x | sub 100 |
| bne | 05 | x | sub 100 |
| j | 02 | x | x x |
| lw | 23 | x | add 011 |
| sw | 2B | x | add 011 |

* **The Final DataPath:**

****

* Simulation and Testing:
* We tested the following instruction sequence after we loaded it into the program memory:

xor $9, $9, $9

ori $9,$0, 1

xor $8, $8, $8

ori $8,$0, 6

sw $8, 2($9)

lw $10, 2($9)

addi $10,$0,5

xor $9, $9, $9

* The program was loaded into the memory in binary as follows:

01294826

34090001

01084026

34080006

ad280002

8d2a0002

200a0005

01294826

* The code sequence is supposed to make $9 equal to zero by XOR instruction, then make an ORI instruction between number 1 and $0, and store the result in $9, then again make $8 equal to zero by XOR instruction, and perform ORI instruction between number 6 and $0 and put the result in $8. After that, it stores the content of $8 in the second address of the data memory, then loads the data from the second address and put it into $10, and performs an ADD (immediate) operation between number 5 and $0, then the output will be in $10, Finally, we again made $9 equal to zero by XOR instruction.

Conclusion:

The simulation has worked fine and it outputs the correct data in the right place in memory. The project was not hard, but it took some time, and it helped us to understand the single cycle processor in such a way that we cannot easily forget, even when it is just a simple design for a simple processor, but it can be considered as a good progress and a good project to help us to understand the processor by simply designing it.