Department of Computer and Electrical Engineering

Computer Architecture

ENCS437

Project 2 Report

Prepared by:

Ahmad Nidal Hamad 1180060

Baraa Atta 1180445

Supervised by:

Mr. Aziz Qaroush

June 2021

Contents

Figures

Abstract

In this project we are going to implement a full pipelined data path using MIPS instruction set.

So, we will start by introducing each component alone, and then we will build the full data path structurally from them. And at the end we will give our own conclusion about the design.

Procedure and discussion

Implementation of each component

In our design, we built each component separately and then we connected the final data path structurally from them. In this section we are going to explain the internal implementation of each component separately as follows.

Register file

The register file was built as shown in figure 1

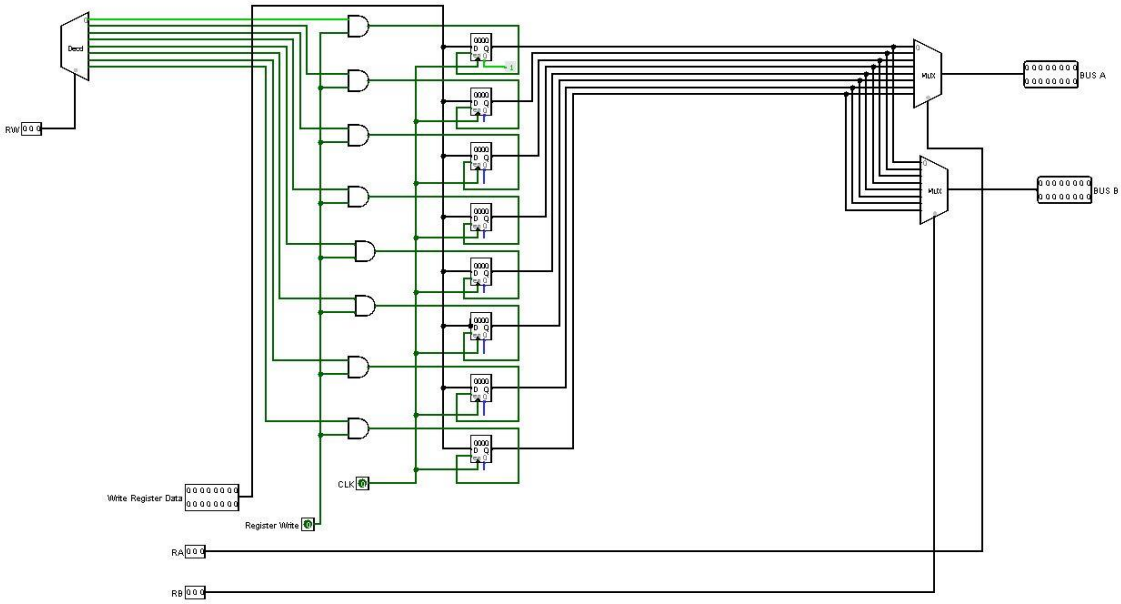

Figure 1

As we can see from the previous figure, the register file was built using 8 registers, each one of them can store 16 bits of data. This component has 5 inputs, the first one is the RW which has the index of the register we are going to write on, Register Write which is a control signal indicated if a we are allowed to write to the register file or not depending on the instruction, Write Register Data that contains the data to be written to the desired register, RA which represents the index of the first register to be read, and RB which contains the address of the second register to get read. Next, all the registers were connected to a decoder, which has the RW as input, and each output is connected to its corresponding register after making sure that the instruction can write data using the and gates. Finally, all registers are connected to two muxes, each one of them can read the data of one register at time, specifying it by the inputs RA and RB.

ALU unit

The ALU unit was implemented as shown in figure 2

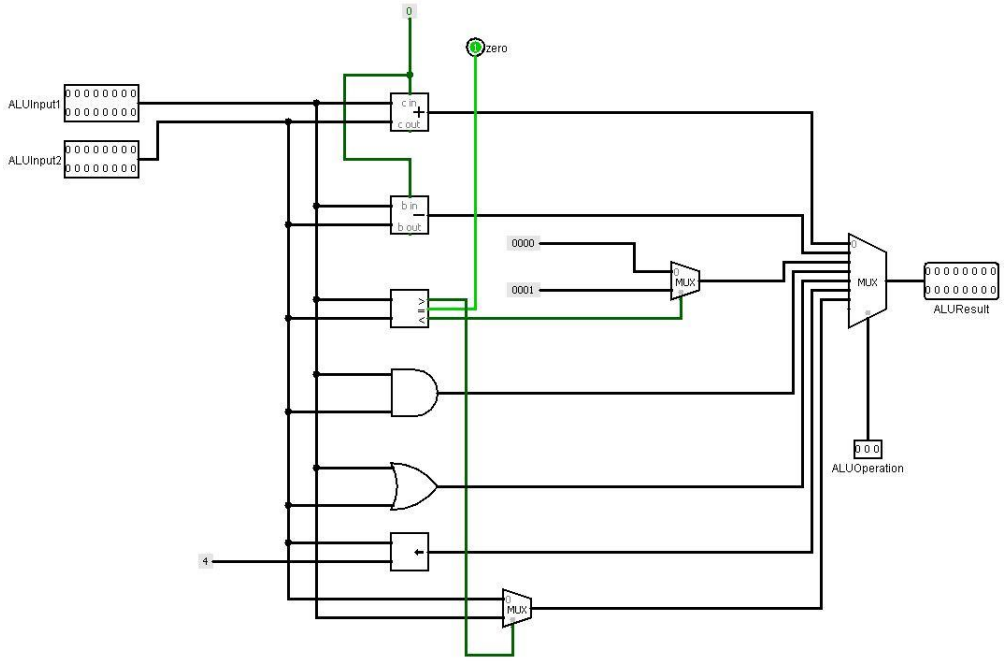

Figure 2

As shown in the previous figure, the implementation of the ALU unit is very simple, it just contains all the arithmetic components needed in our instruction set which are: an adder, and subtractor, and comparator, and gate, or gate, and a left shifter. 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. Note that this signal will be explained in details when we move to the control using section.

The extender

The extender was built as shown in figure 3

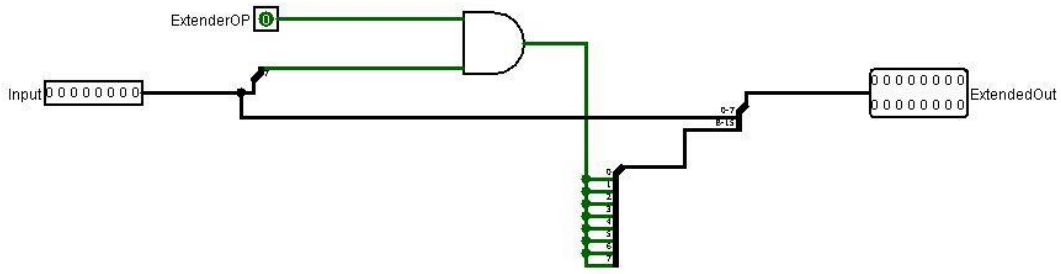

Figure 3

As we can see from the previous figure, the implementation is very simple, it just takes two inputs, one for the number we want to extend, and the another one represents a control signal to determine if we want a signed or unsigned extension, then the control signal is inserted

with the last bit of the number to and gate, which will have required extension bit at its output, and finally this bit is added to the 8 most significant bits of the input number.

control unit

This is the most important unit in the design, since it will generate all the signals which will control the operation of all other components. The implementation of this components is shown in figures 4 and 5.

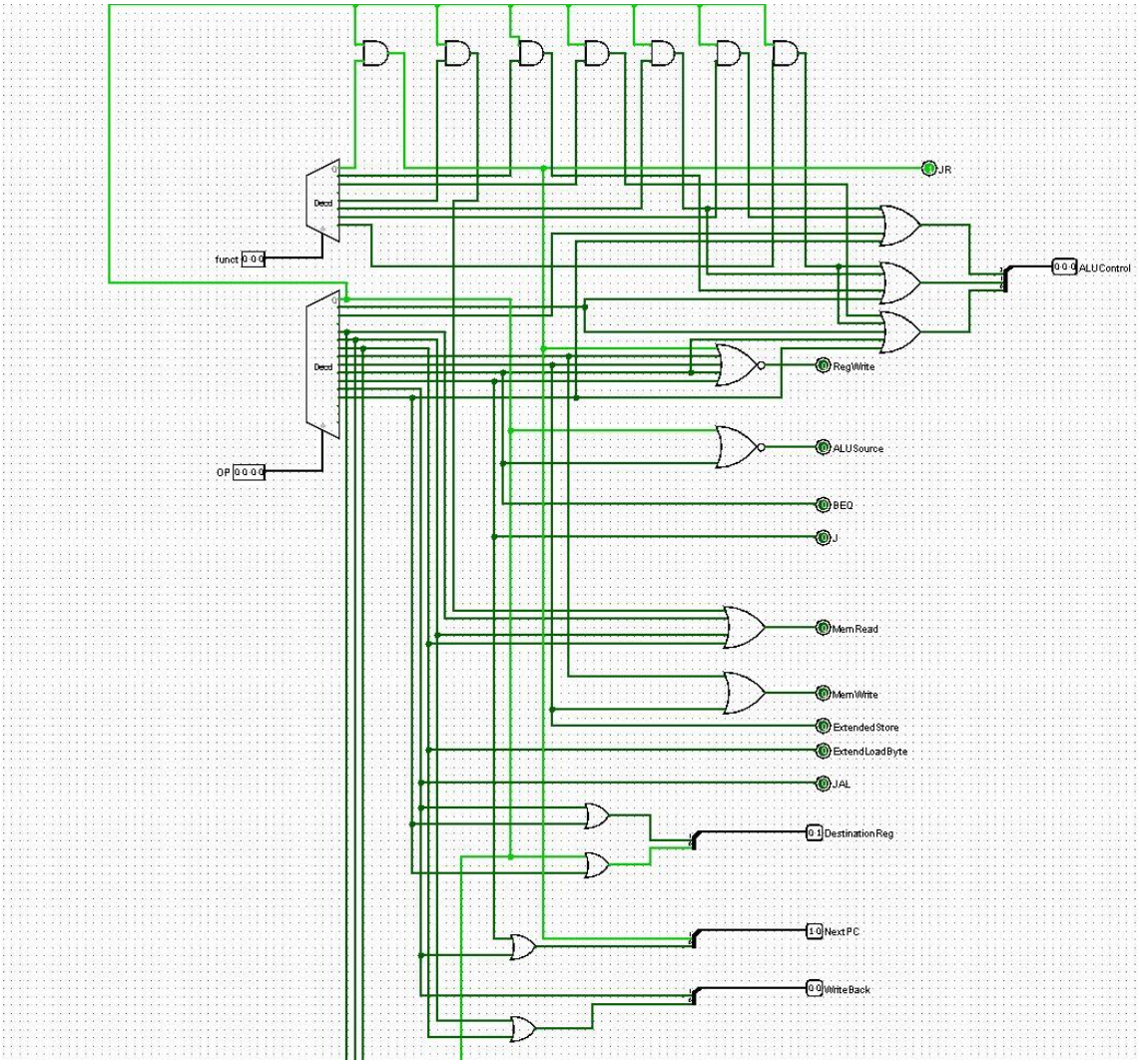

Figure 4

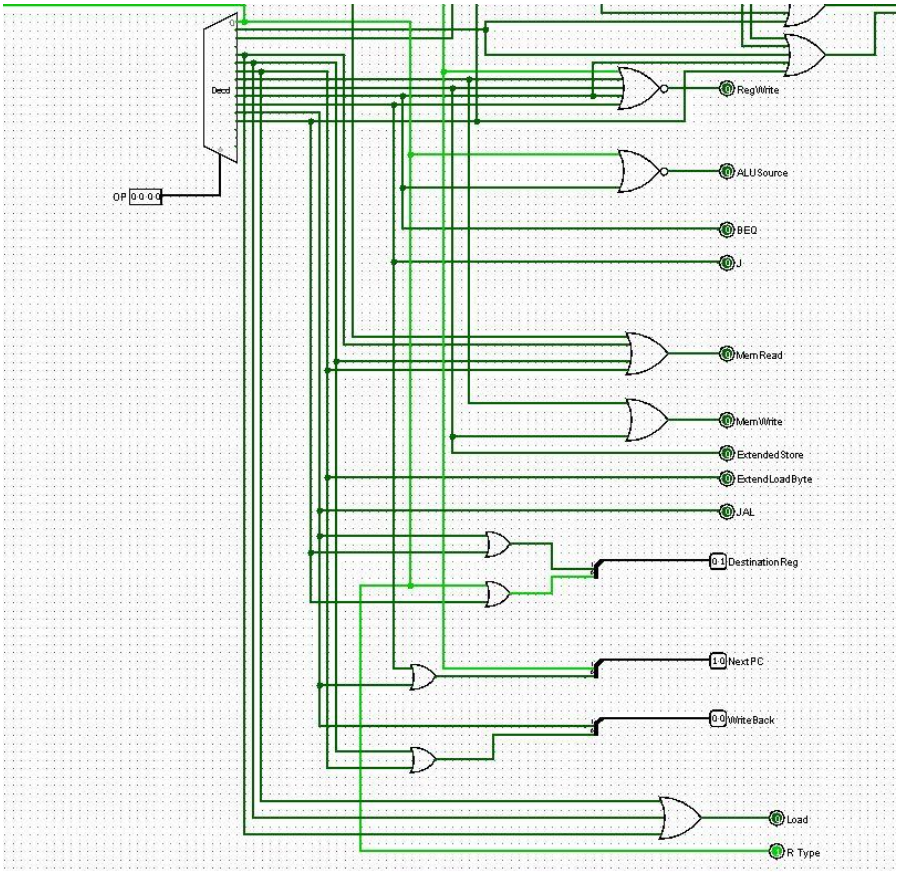

Figure 5

As shown in the previous figure, this component has a lot of outputs, so to make things easier we will explain each one of them separately with its functionality and equation as follows:

1- JR, J, BEQ, JAL: those are simple signals that are taken directly from the decoder to indicate if the instructions are the one indicated by the output name.
2- Load: this is also a signal the instruction is a load instruction or not, but it is taken from the output of an or gate, since there are 3 different types of load instructions in our instruction set.
3- R Type: this signal indicates if we are dealing with R type instruction or not, by simply taken the first output of the decoder, since all R type instructions will have an op code of 000.
4- Reg Write: this signal is used to determine if the instruction is allowed to write on the register file or not. Since the number of instructions that write on the register file is big, we determine to insert all the instructions that are not allowed to write on the register file into nor gate instead of or gate.
5- ALU Source: this signal will determine if another operand should be Rt or the immediate. As we already know, the second input for the ALU will be RT only if it was an R type instruction or a BEQ instruction, so we put the decoder outputs corresponding to the R type (output 0) and BEQ (output 9) ones into an or gate, since in the data path Rt was connected to the first input of the mux.
6- Mem write and Mem read: the first one indicates if the instruction is going to write on the memory, while the second one indicates if the instruction is going to read from memory. So, the first one is calculated by inserting all the decoder outputs

corresponding to store instruction, and the Mem read is done the same way, but with replacing the store instructions with load ones.

7- Extended Load Byte: this signal is used when we read one byte from memory, and we want to know to extend it with zeros or with the sign bit, so its simply the decoder output corresponding to normal Load byte instruction.

8- Extend Store: this signal is used as a selection for the mux at the input of the data memory, in order to select the normal ALU output or the extended version of it. So, this signal will be needed when dealing with the store byte instruction, as a result it just takes the decoder output corresponding to the store byte instruction since the extended version ins connected to the second output of the mux.

The other control signals are more than one-bit signals, so it is hard to explain them without referring to the muxes which are connected to. So, I will keep their explanation to the discussion of the project. But in brief they function as the following:

9- ALU control: this signal will choose the suitable arithmetic operation based on the instruction.
10- Destination Reg: this signal will choose between R7, R1, Rt, and Rd to write on.
11- Next pc: this signal will choose the source of the next instruction pointer.
12- Write back: this signal will choose between the different possibilities of the sources to write it back on the register file, like the ALU output or the memory word or the extended version of the memory byte.

Control signals table

Op	Reg Dst	Reg Write	Ext Op	ALU Src	Beq	JR	JAL	J	Next Jump PC	Mem Read	Mem Write	Mem toReg	Load byte Extender	Mux Write back	store byte Zero Extender Mux
R-type	01 = Rd	1	x	0=Bus B	0	0	0	0	XX	0	0	0	X	00=normal	X
R – type LWS	01=Rd	1	x	0=Bus B	0	0	0	0	XX	1	0	1	X	00=normal	X

JR	PC	0	x	x	0	1	0	0	10	0	0	0	X	X	X
addi	00 = Rt	1	1=sign	1=Imm	0	0	0	0	XX	0	0	0	X	00=normal	X
andi	00 = Rt	1	0=zero	1=Imm	0	0	0	0	XX	0	0	0	X	00=normal	X
ori	00 = Rt	1	0=zero	1=Imm	0	0	0	0	XX	0	0	0	X	00=normal	X
Lw	00 = Rt	1	1=sign	1=Imm	0	0	0	0	XX	1	0	1	X	00=normal	X
Lbu	00=Rt	1	x	1=Imm	0	0	0	0	XX	1	0	1	0=zero	01=extended byte	X
Lb	00=Rt	1	X	1=Imm	0	0	0	0	XX	1	0	1	1=sign	01=extended byte	X
Sw	x	0	1=sign	1=Imm	0	0	0	0	XX	0	1	x	X	X	0
Sb	x	0	1=sign	1=Imm	0	0	0	0	XX	0	1	x	X	X	1
beq	x	0	x	0=Bus B	1	0	0	0	00	0	0	x	X	X	X
J	x	0	x	X	0	0	0	1	01	0	0	x	X	X	X
JAL	10=111	1	x	X	0	0	1	0	01	0	0	0	X	10=PC + 2	X
LUI	11=001	1	x	1=Imm	0	0	0	0	XX	0	0	0	X	0=normal	X

Next PC unit

This unit is used to determine the next pc value as shown in figure 6.

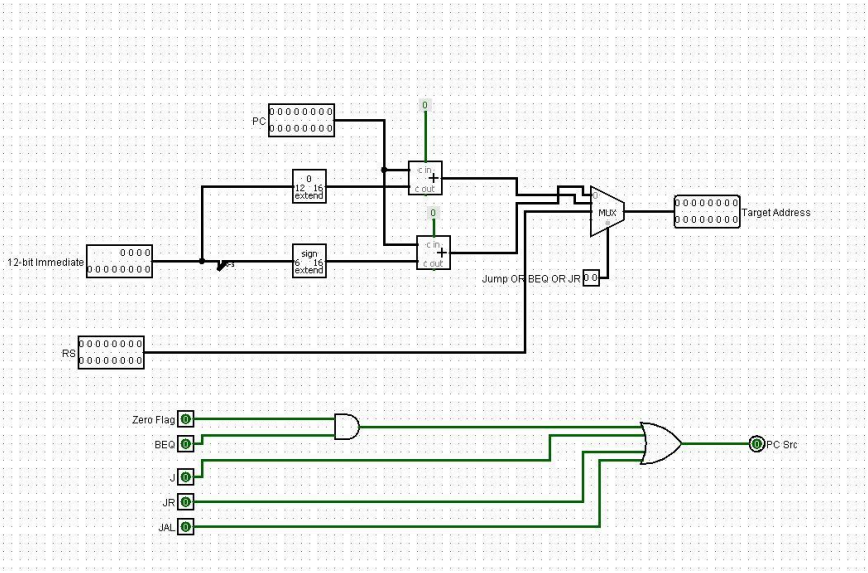

Figure 6

As we can see from the previous figure, the unit has two main inputs, the immediate and the register Rs, also it has another input for all the instructions expected to perform jump operation.

According to the output it has two outputs, which are the PC Src to indicate if there is a jump instruction or we should continue the program normally, the other output is the new value for the PC if a jump instruction is encountered, the selection is done using a special signal, which will map every instruction with its suitable PC value. Note that all jump instructions are connected directly to the or gate, except the BEQ since it isn't sufficient to have a branch equal instruction to jump, but also its condition should be satisfied.

Forwarding unit

This unit is built to reduce the number of stall cycles, by returning the output of one instruction to another one dependent on it, without waiting until it reaches the write back stage. The implementation of this unit is shown in figure 7.

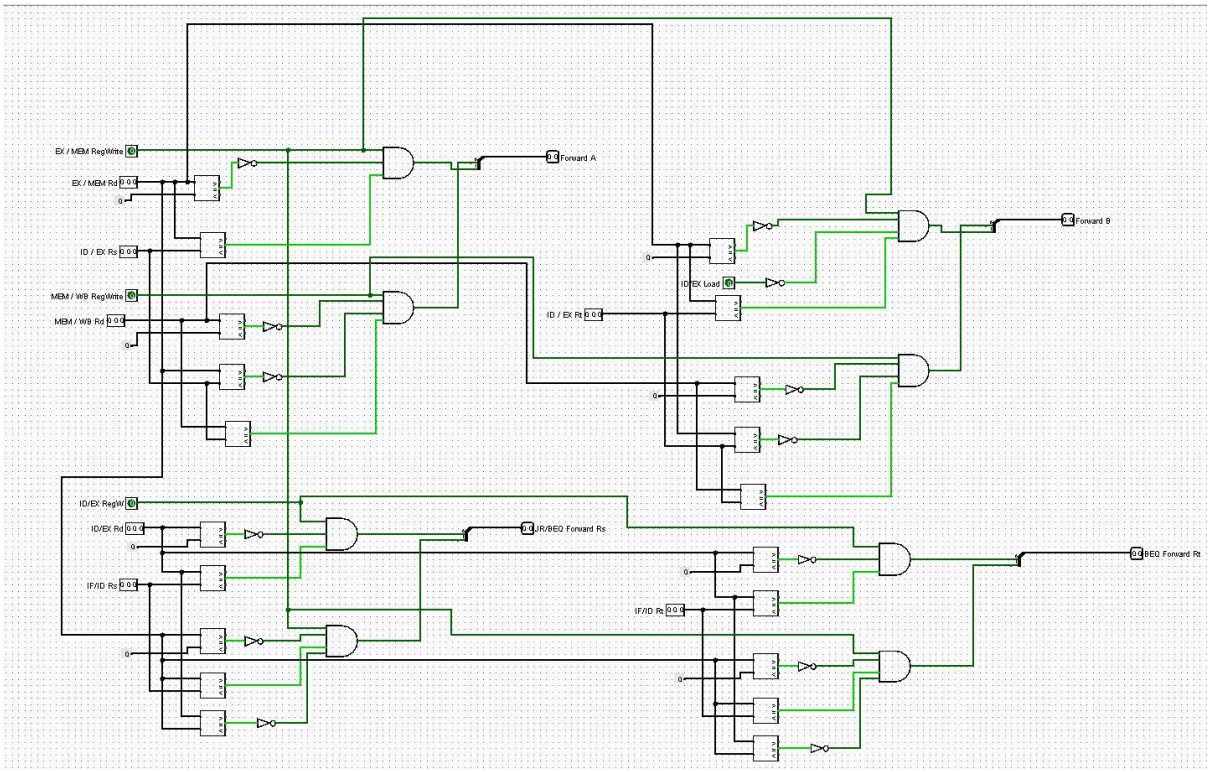

Figure 7

As we can see from the previous figure, the unit has multiple sub circuits each one of them is responsible for one of the forwarding locations as follows:

1- Forward A: this circuit is used to forward the output of the ALU and the one for the instruction in the memory write back stage to the instruction that will come to the execution stage next, if and only if there is a dependency between it and the instructions that preceded it. This is done by checking some conditions which are:
   - Part 1
     1- The instruction in the memory stage should be writing to a register.
     2- The register which the instruction is going to write on shouldn't be the zero register.
     3- The destination register for the instruction in the memory stage should be the same as the source register (Rs) for the instruction in the execution stage.

   - Part 2
     1- The instruction in the write back stage should be writing to a register.
     2- The register which the instruction is going to write on shouldn't be the zero register.
     3- The destination register for the instruction in the memory stage should be the same as the source register (Rs) for the instruction in the execution stage.
     4- There shouldn't be a dependency between the instruction in the execution stage and the one in the memory stage, or it will take a wrong input

If all the conditions of one of the parts are satisfied, the forwarding unit output will be 10 or 01 (since it's impossible to satisfy both toother). So, in the data path we will connect forwarded output from the memory stage to the input 10 of the mux, and the one from the write back stage to the input 01 of it.

2- Forward B: this has the same functionality as Forward A, but with a dependency on the Rt register not Rs, otherwise the circuit is the same.
3- JR/BEW forward Rs: this signal is used for the forwarding from the memory and the write back stages to the decode stage, particularly for the JR and BEQ instructions, since the are the only instructions that executed in the decode stage, and needs an input from a register or registers. Also, here there are some conditions that should be satisfied to make the forwarding process which are:
    - part 1
        1- The instruction in the memory stage should be writing to a register.
        2- The register which the instruction is going to write on shouldn't be the zero register.
        3- The destination register for the instruction in the memory stage should be the same as the source register (Rs) for the instruction in the decode stage.
    - Part 2
        1- The instruction in the write back stage should be writing to a register.
        2- The register which the instruction is going to write on shouldn't be the zero register.
        3- The destination register for the instruction in the memory stage should be the same as the source register (Rs) for the instruction in the execution stage A.
        4- There shouldn't be a dependency between the instruction in the execution stage and the one in the memory stage, or it will take a wrong input.

    If all the conditions of one of the parts are satisfied, the forwarding unit output will be 10 or 01 (since it's impossible to satisfy both toother). So, in the data path we will connect forwarded output from the memory stage to the input 10 of the mux, and the one from the write back stage to the input 01 of it.

4- BEQ Forward Rt: this has the same functionality as Forward A, but with a dependency on the Rt register not Rs, otherwise the circuit is the same. Note that this forwarding process in only necessary for the BEQ instruction, since it's the only one that takes Rt as input register.

Hazard detection unit

The function of this unit is to insert a stall cycles whenever needed, as we already know there are two cases where stall cycles are needed, which are the load instruction and an instruction following it and dependent on it, and the other case when any type of jump instructions occurred. The implementation of this unit is shown in figure 8.

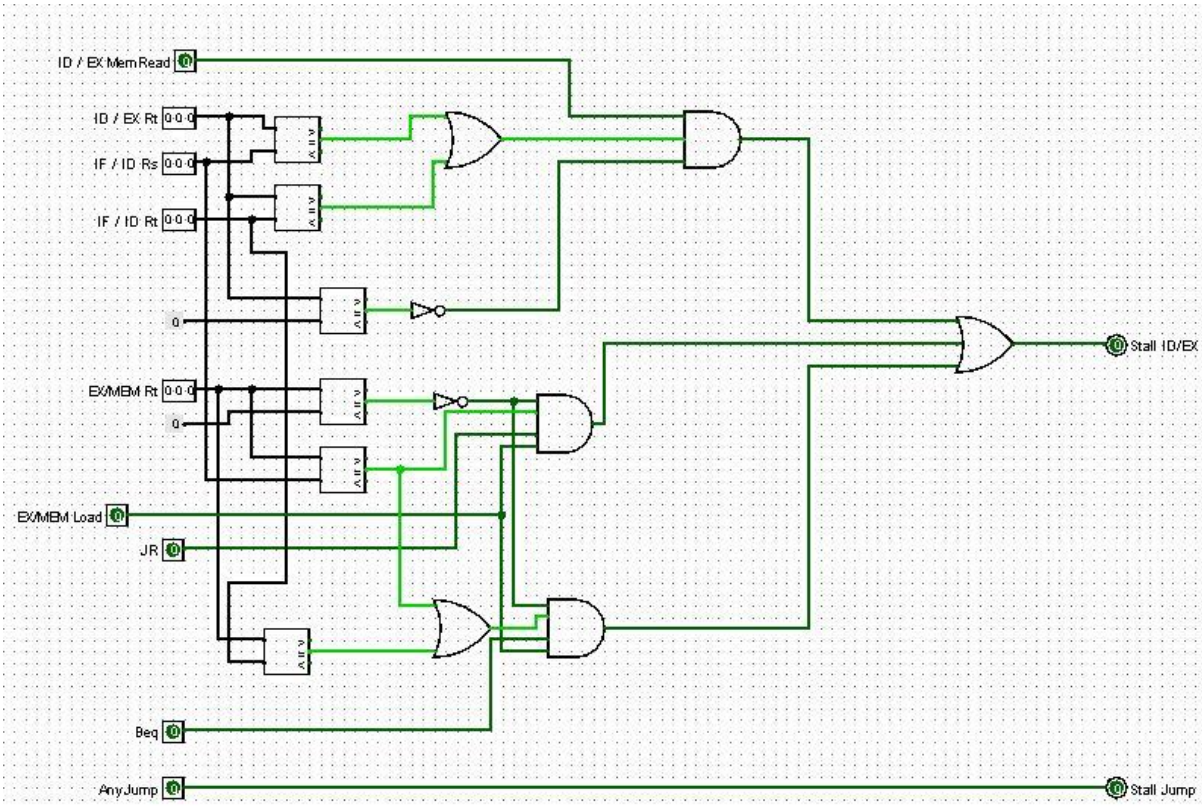

Figure 8

As we can see from the previous figure, this unit has two inputs, one to stall a cycle in the decode stage and on to stall cycle in the execution stage. The details of each output are shown in the next lines:

1- Stall Jump: this signal is used to add a stall cycle to the decode stage whenever any instruction that will perform jump operation occurs.
2- Stall ID/EX: this signal is used to stall a cycle in the execution stage whenever one of the following three cases satisfied:
   - Case 1:
     1- If the instruction in the execution stage is a load instruction and is going to write on a register.
     2- If the destination not the zero register.
     3- If the destination register is as same as at least one of the sources registers for the instruction in the decode stage.

   - Case 2:
     1- If the instruction in the memory stage is a load instruction and is going to write on a register.
     2- If the destination not the zero register.
     3- If the destination register is as same as at least one of the sources registers for the instruction in the decode stage and the instruction in the decode stage is jump instruction or branch equal.

- Case 3:

  Case 3 is as same as case 2 but with a dependency on Rt only not Rt or Rs, also the instruction in the decode stage should be BEQ, since it's the only one that takes Rt as input register.

Building the final data path

The final step now is to connect the pre-impelemnted components together to the final data path from them. The final path is shown in figures 9 and 10.

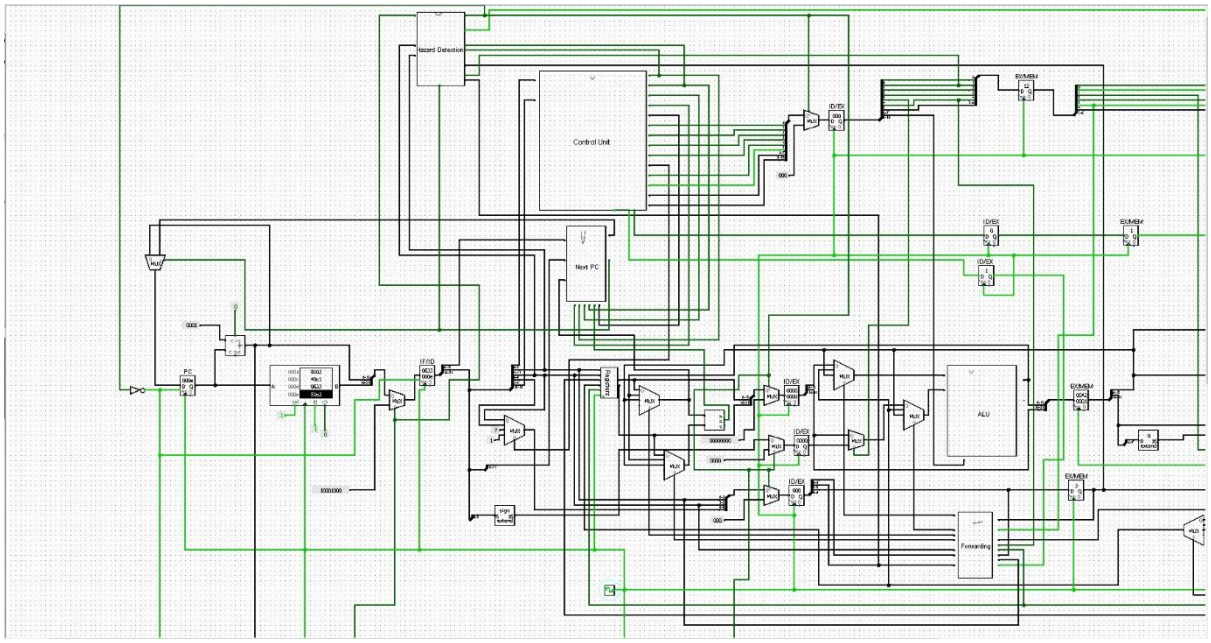

Figure 9

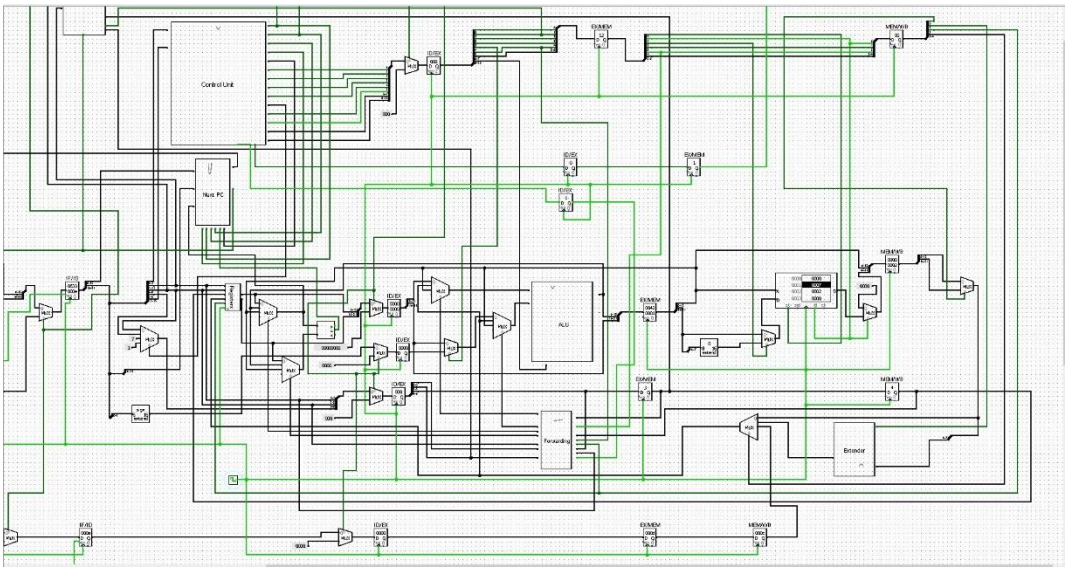

Figure 10

As we can see from the previous figures, the final data path was built structurally from the components we built before. The data path was built under the pipelined approach, for example we can see that there is a register between every two stages, also we can see a lot of muxes in the system, each one of them has one of the forwarding signals on its selection line in order to maintain the consistency on our data and results.

Before ending with some simulation results, we should pay your attention to some important notes which are:

1- The structural hazards were solved by making the writing operation on the register file done on the falling edge, while the reading on the rising edge.
2- The data memory is word addressable, so when dealing with store byte instruction, we determine to extend the byte with zeros before saving it.
3- In out design the first instruction in the instruction memory shouldn't be zero, since it represents a jump instruction to address zero, which will make the program stuck at the first address, so we advise you to use the instruction 1000 which represents adding zero to register zero; to make sure that nothing will be affected.
4- When dealing with stall cycles, we put zeros when stalling in the ID/EX registers, while we put 100 when stalling in the IF/ID registers, since the zeros in the instruction decode stage will mean a jump instruction.

Some simulation results

Since it is very hard to give a screen shot for every instruction in every stage, we will give a simple program with some cases like forwarding and stalling to make sure that the system is working fine, moreover wile discussing the project all the cases will be sown clearly.

The sample code:

LUI 3H

ADDI R6,R1,2H

SW (R0+1) = R6

J 10

From address 10:

LW R3 = R0 + 1

ADDI R7,R3,2H

The machine code of the previous code

C003

3382

7181

A00A

From address 10:

40C1

37C2

Here we can see a simple code, but with different forwarding and stall cases. First of all we loaded the 4-bits shifted value of 3H to the first register, so we expect that register one will hold a value of 30H, next we added register one to a value of 2H and store the result in R6, that will contains 32H here we can see that forwarding is necessary since there is a read after write dependency, next we stored the value of R6 in the address 1 which should contain a value of 32H, and then we jumped to address 10, moreover after jumping, we load a word from address 1 which will contain a value of 32H . finally, we added a value of 2H to the value of the first register and store the result in R7, so we expect it to have a value of 34H. as we can see that the last case will cover the read after write case with load instruction, which will require a stall cycle.

After running the simulation, we expect values like this:

R1 = 30H

R6 = 32H

R3 = 32H

R7 = 34H

After running the simulation, we got the results shown in figure 11.

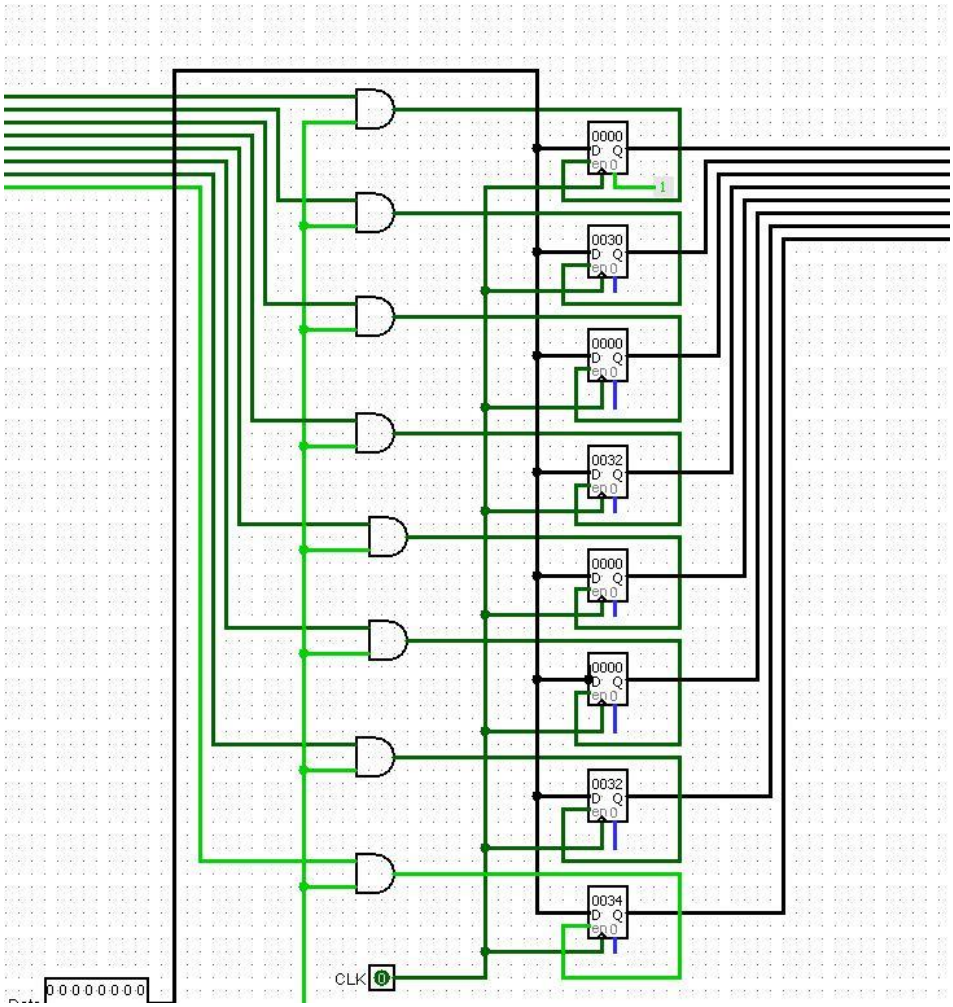

Figure 11

As we can see all the registers contain the expected values, so the most important cases like the forwarding and cycle stalling are working well, so the data in consistent.

At the end we should say that we tested all other instructions on our data path, and all of them worked fine, but it's very hard to show a simulation for every one of them in the report.

Conclusion

In this project we learned how to implement a full pipelined data path using MIPS instruction set. Also, we have seen how we can use forwarding in order to have less stall cycles to make our data path more efficient. Finally, we can say that it was very interesting to implement a pipelined data path that is close enough to the one in our daily used CPUs.