Faculty of Engineering and Technology

Electrical & Computer Engineering Department

Computer Architecture ENCS4370

Project 2

Prepared by:

Name: mahmoud nobani **Id:**1180729

Name: Abdelghafur mousa **Id:**1180628

Name: masoud ajouli **Id:**1181621

Instructor: Dr. aziz qaroush

Section: 2

Date:12/6/2022

Abstract:

In this project we aim to implement a 24 bit, mips machine using the Logisim simulator, and we aim to have 15 instructions in the instructions set.

Table of Contents

Figures:

Tables:

Theory and Procedure:

The Datapath for mips in general is divided to 5 phase, IF, ID, EXEC, M, WB, so we first started with the aim of implementing each phase with the components it needs.

1) Instruction fetch (IF):

For this phase we need the instruction memory unit which takes the address as an input and gives the instruction as an output, we implemented it as shown below:

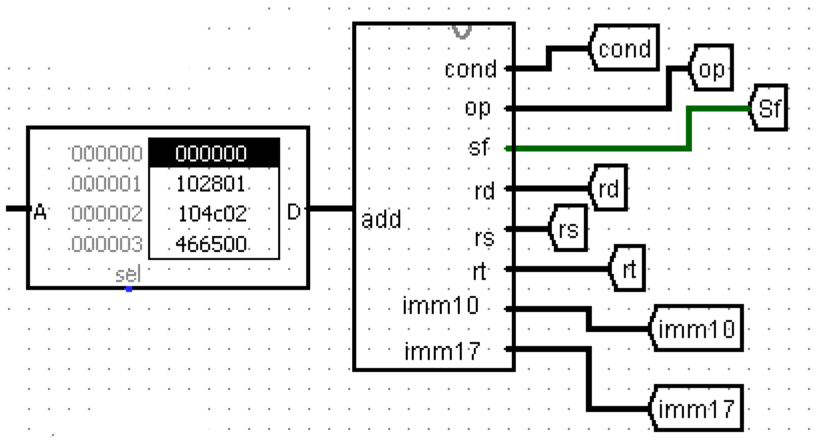


Figure 1: instruction memory

Where the first unit is the memory or ROM, and the second unit is the instruction memory which takes the address from the memory and giving us all the parameters, we need for the different types of instruction.

2) Instruction decode (ID):

This phase is inseparable from the previous one, we just added a new unit which is register file that helps us getting the need registers for our instructions.

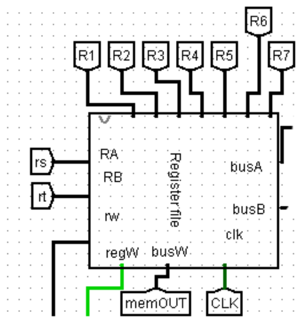


Figure 2: register file.

We were able to make the last unit using the following configuration:

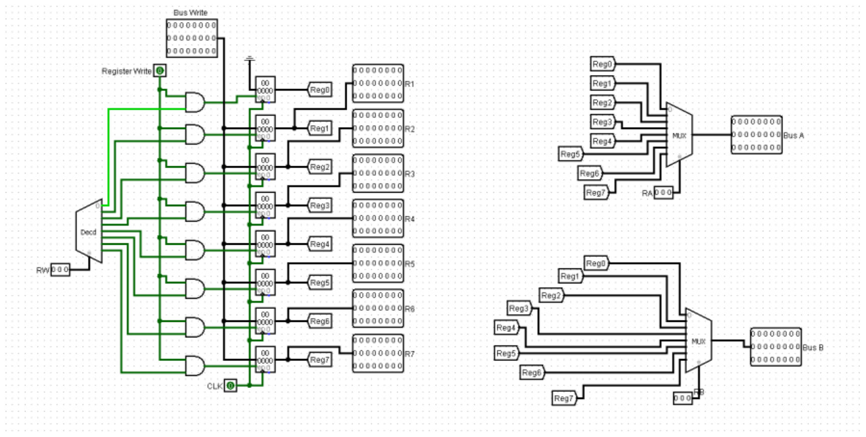


Figure 3: Register file internal representation

Where the first part is used to write on the 8 registers we have and the second part was used to get busA and busB, note that we can't edit R0 as requested.

3) Execute:

For the third phase we added Alu, the Alu configuration is as shown below:

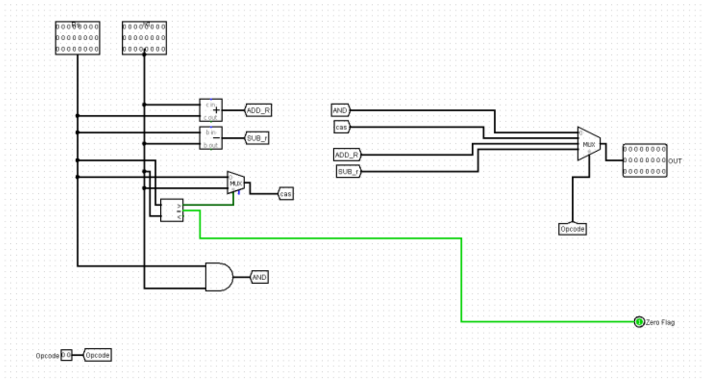


Figure 4: Alu

The Alu is used to perform various arithmetic operations, for this project we need only 4 main operations which are and, sub, add and cas.

4) Memory:

The fourth phase introduced to us an extra new component which is the memory, it will be utilized for loading and storing data, to implement it we used RAM memory component.

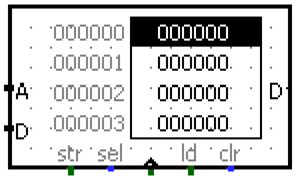


Figure 5: RAM.

5) Write Back:

Lastly the fifth phase, the implementation for it was simple as it was just a mux between Alu result and the memory output.

Extra components:

Logisim didn't provide a good extending unit for immediate values which supports both unsigned and signed, so we had to implement and extending unit as shown here:

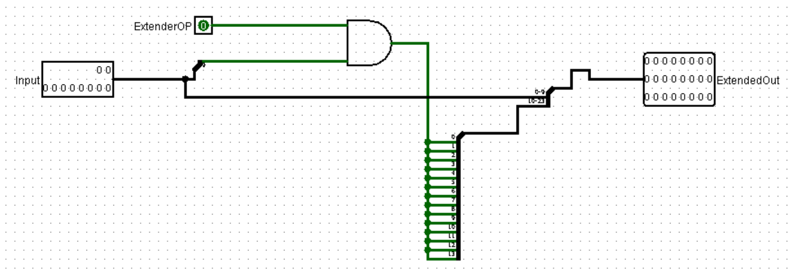


Figure 6: extender.

<u>data path:</u>

the supported instructions:

R-type						
0	AND	Reg(Rd) = Reg(Rs) & Reg(Rt)	Op = 00000	Rs	Rt	Rd
1	CAS	Reg(Rd) = Max[Reg(Rs) , Reg(Rt)]	Op = 00001	Rs	Rt	Rd
2	Lws	Reg(Rd) = Mem[Reg(Rs) + Reg(Rt)]	Op = 00010	Rs	Rt	Rd
3	ADD	Reg(Rd) = Reg(Rs) + Reg(Rt)	Op = 00011	Rs	Rt	Rd
4	SUB	Reg(Rd) = Reg(Rs) – Reg(Rt)	Op = 00100	Rs	Rt	Rd
5	CMP	zero-flag = Reg(Rs) < Reg(Rt)	Op = 00101	Rs	Rt	00
6	J	PC = Reg(Rs)	Op = 00110	Rs	00	00

Table 1: R-type

I-Type						
7	ANDI	Reg(Rt) = Reg(Rs) & Immediate^{10}	Op = 00111	Rs	Rt	Immediate^{10}
8	ADDI	Reg(Rt) = Reg(Rs) + Immediate^{10}	Op = 01000	Rs	Rt	Immediate^{10}
9	Lw	Reg(Rt) = Mem(Reg(Rs) + Imm^{10})	Op = 01001	Rs	Rt	Immediate^{10}
10	Sw	Mem(Reg(Rs) + Imm^{10}) = Reg(Rt)	Op = 01010	Rs	Rt	Immediate^{10}

11	BEQ	Branch if (Reg(Rs) == Reg(Rt))	Op = 01011	Rs	Rt	Immediate^{10}
J-Type						
12	J	PC = PC + Immediate^{17}	Op = 01100	Immediate^{17}		
13	JAL	R7 = PC + 1, PC = PC + Immediate^{17}	Op = 01101	Immediate^{17}		
14	LUI	R1 = Immediate^{17} << 4	Op = 01110	Immediate^{17}		

Table 2: I and J-type

Then we started analayzing each instrustion:

- Instruction 6,11,12,13: these instruction are typical, they were added to the jump instrction part, the only difference was with JAL, so we added another value to the RegDst mux equal 7.
- Instructions 0.1,2,3,4: these are typical r-type instruction which works as usual, they will affect RegDst and MemToReg and MemR.
- Instructions 7,8,9,10: the normal I-type instruction, with the change to the control signals
- Instruction 5: not ur typical r-type, but doesn't change much, we just need to update the zero flag with not writebacks.
- LUI: which is the tricky one, we added a value for RegDst and and a shift left components.

Looking back at all of those instructions we implemented the following datapath:

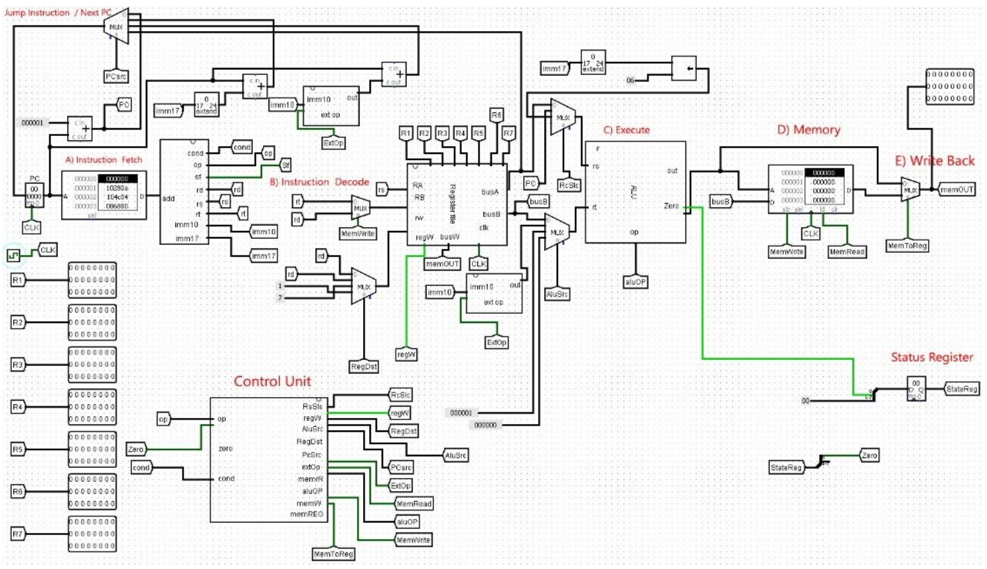


Figure 7: final Datapath

Lastly, we need to implement the control unit with a consideration to the condition for each instruction, the following table represent the truth table for the control unit:

OP Code	Instruction	Reg. Dst.	RegW	ExtOP	ALU Src	RsSlc	ALU Op.	MemR	MemW	MemToReg	PCSlc
00000	AND	0	1	X	0	0	And	0	0	0	0
00001	CAS	0	1	X	0	0	Cas	0	0	0	0
00110	LWS	0	1	X	0	0	Add	1	0	1	0
00111	ADD	0	1	X	0	0	Add	0	0	0	0
00100	SUB	0	1	X	0	0	sub	0	0	0	0
00101	CMP	X	0	X	0	0	X	0	0	X	0
00110	JR	X	0	X	X	0	X	0	0	X	3
00111	ANDI	1	1	0	1	0	And	0	0	0	0
01000	ADDI	1	1	1	1	0	Add	0	0	0	0
01001	LW	1	1	1	1	0	Add	1	0	1	0
01010	SW	1	0	1	1	0	Add	0	1	X	0

01011	**BEQ**	**X**	**0**	**1**	**0**	**0**	**X**	**0**	**0**	**X**	**2**
01100	**J**	**X**	**0**	**X**	**X**	**0**	**X**	**0**	**0**	**X**	**1**
01101	**JAL**	**2**	**1**	**X**	**2**	**1**	**Add**	**0**	**0**	**0**	**1**
01110	**LUI**	**3**	**1**	**X**	**3**	**2**	**add**	**0**	**0**	**0**	**0**

Table 3: Truth table for control unit

Testing:

Now we need to do some testing so we will test the following code:

1) addi r1,r2,10
2) addi r2,r3,4
3) sub r3,r2,r1
4) jEQ 15
5) sw r2,r0,10
6) lw r4,r0,10
7) luiNE 1
8) jr r0

the first 3 instruction gave us the following:

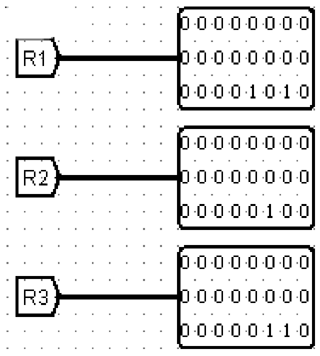
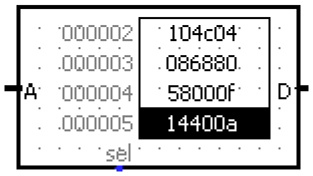


Figure 8: output1

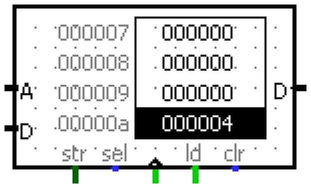
The 4th instruction didn’t work as expected because the condition wasn’t met, we continue to 000005



```
000002  104c04
000003  086880
000004  58000f
000005  14400a
```

Figure 9: output2

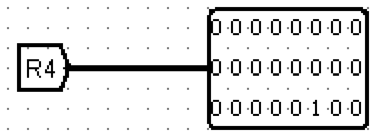
Instruction 5 gave us the right result and did store r2 in location 10.



```
000007  000000
000008  000000
000009  000000
00000a  000004
```

Figure 10: output3

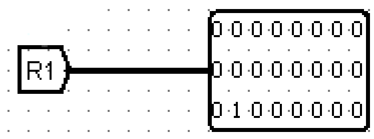
As for the 6th instruction, it loaded from location 10, value 4 to r4.



```
00000000
00000000
00000100
```

Figure 11: output 4

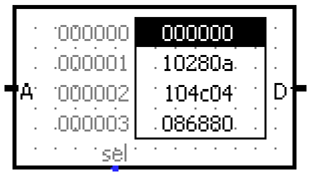
And instruction 7 did work well, because the previous operation didn't have zero flag.



```
00000000
00000000
01000000
```

Figure 12: output 5

Lastly jr worked well and we were back to location 0.



```
000000  000000
000001  10280a
000002  104c04
000003  086880
```

Figure 13: output 6

And the final output is:

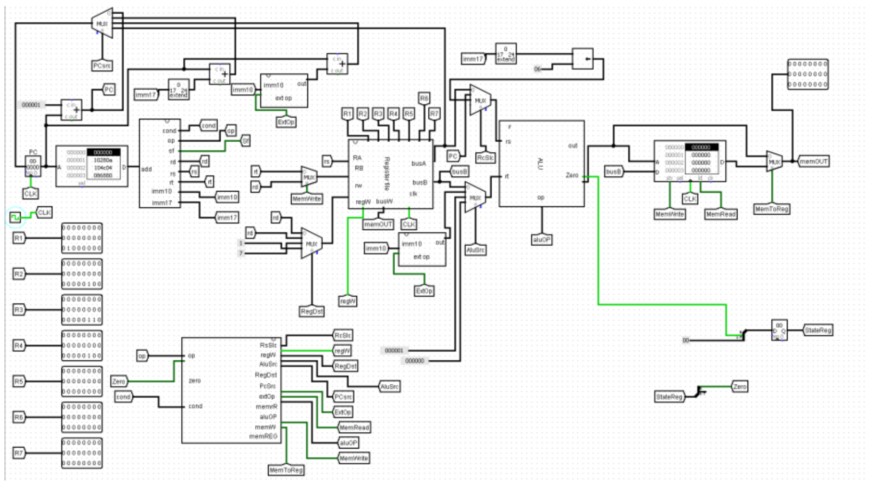


Figure 14 :final output

Hex values for the used instructions:

10 28 0A ; ADDI R1,R2,10
10 4C 04 ; ADDI R2,R3,4
08 68 80 ; SUB R3,R2,R1
58 00 0F ; JEQ 15
14 40 0A ; SW R2,R0,10
12 80 0A ; LW R4,R0,10
9C 00 01 ; LUINE 1
0C 00 00 ; JR R0

Pipelining:

In order to enable the pipelining, we need to implement registers that aim to move data and control signals between the sections on clock in order to reduce the time taken for a program.

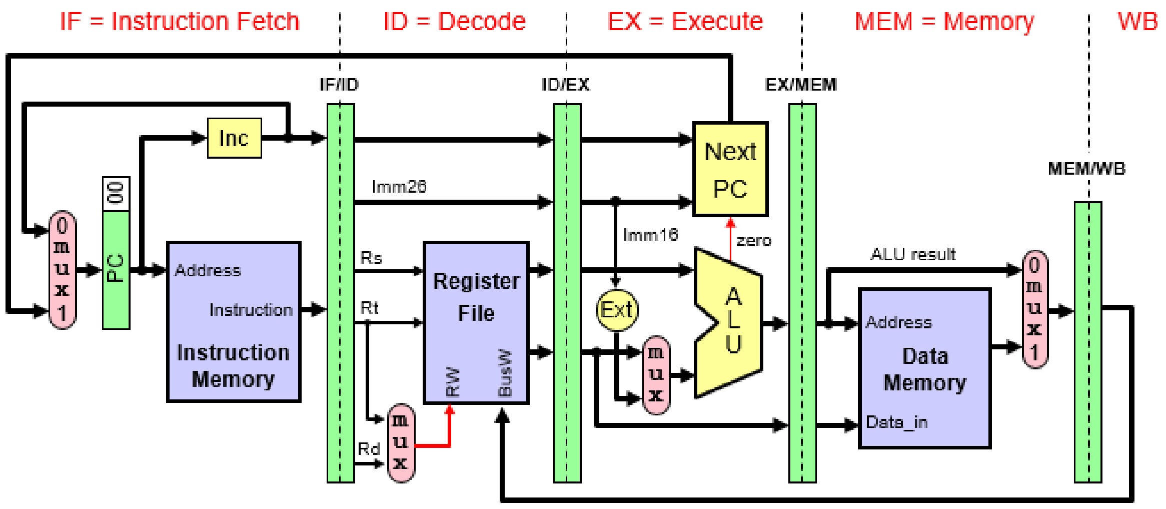


Figure 15: pipeline machine

In theory if we want to implement pipelining on our machine it should something like this:

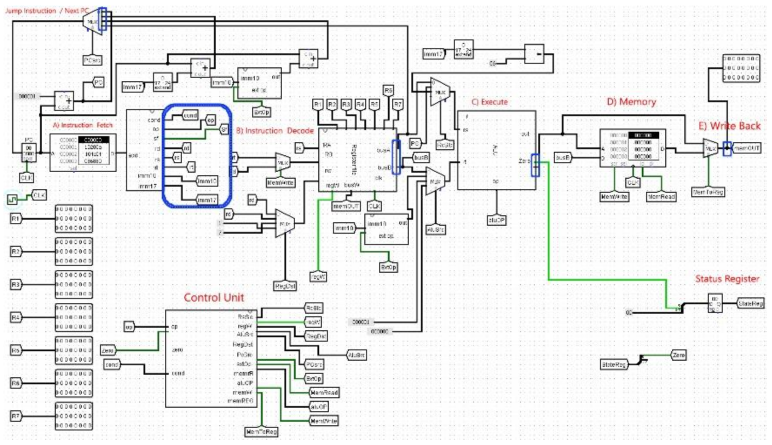


Figure 16: theoretical pipeline

Note the control signal should also have blue rectangles (registers).

We would also need a hazard forwarding unit.

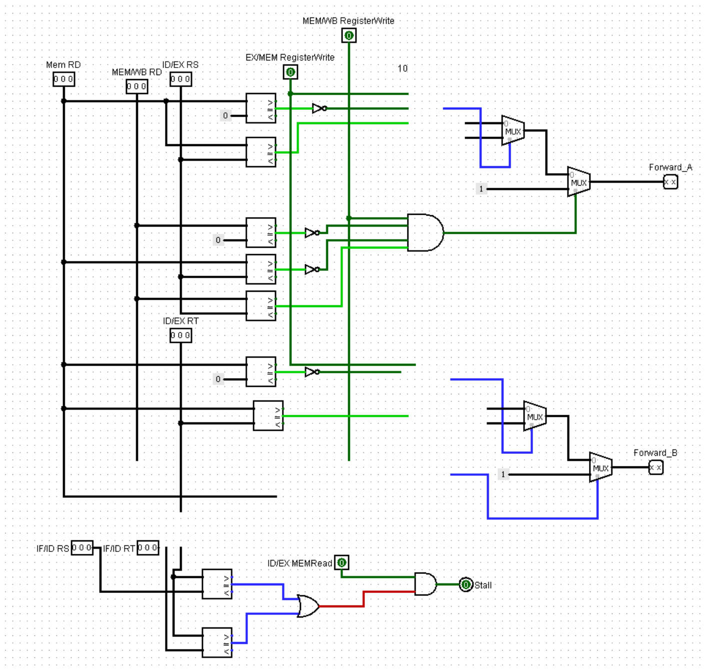
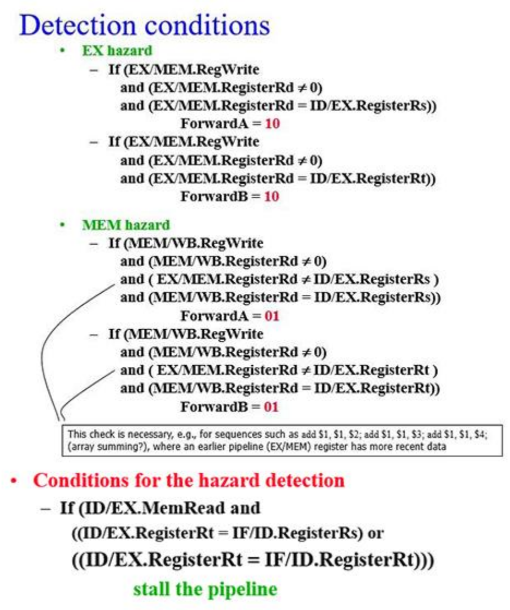


Figure 17: Hazard forward unit

Sadly, due time constrains we weren't able to continue with it, so we just added a prototype as in the picture.

The theory behind it is as shown below:



Detection conditions

- **EX hazard**
  - **If (EX/MEM.RegWrite**
    and (EX/MEM.RegisterRd $\neq$ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 10
  - **If (EX/MEM.RegWrite**
    and (EX/MEM.RegisterRd $\neq$ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10
- **MEM hazard**
  - **If (MEM/WB.RegWrite**
    and (MEM/WB.RegisterRd $\neq$ 0)
    and (EX/MEM.RegisterRd $\neq$ ID/EX.RegisterRs)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - **If (MEM/WB.RegWrite**
    and (MEM/WB.RegisterRd $\neq$ 0)
    and (EX/MEM.RegisterRd $\neq$ ID/EX.RegisterRt)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

This check is necessary, e.g., for sequences such as add $1, $1, $2; add $1, $1, $3; add $1, $1, $4; (array summing?), where an earlier pipeline (EX/MEM) register has more recent data

- **Conditions for the hazard detection**
  - **If (ID/EX.MemRead and**
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
    ((ID/EX.RegisterRt = IF/ID.RegisterRt)))
    stall the pipeline

Conclusion:

In this project we learned how to implement a single cycle data path using MIPS instruction set. We also kind of learned how to theoretically make a pipelined machine, sadly we couldn’t see the difference between these two implementations, lastly we ran into a lot of problems like time and how sometimes annoying and lacking Logisim can be.