Faculty of Information Technology
Computer Systems Engineering Department

Computer Architecture

Project 2

Saleem Hamo 1170381
Mohammad Abbas 1170011

Insructor: Dr. Aziz M. Qaroush
Section 1

ENCS437
2019/2020 Second Semister

Project 2 Report

I. Design and Implementation

A. Datapath

The design of the CPU is a basic MIPS design with a subset of the instruction set. We Used a pre-implemented system (attached in the referenes) containing the basic components and implementing 16 basic instruction. In our modification we implemented the five mips instrucions (LWS, CAS, LLB, JAL, JALR).
The general data path starts with the PC which feeds into instruction memory and gets the instruction, then goes into the register file and reads the source register(s) used for the instruction. Next it depends on the instruction OPCODE.
The control unit control which path it will take based upon the instruction and feeds the appropriate control signal to where they need to go. In the newly implemented instructions, the datapath is modified to accept them and achieve there functionality in a proper way which will be descriped in the next sections.

1) Components

The components for the whole CPU include the Register File, the PC module, the Instruction memory, the data memory, the ALU, the ALU Control, the PC control, and the Main control unit. Also, there are a set of wires, multiplexers and a comparator.

2) Implementation

There are three units for controlling the CPU: The Main Control unit, the ALU Control, and the PC Control. The Main Control unit handles interpreting the opcodes from the current instruction and enabling all of the relevant signals so that the rest of the processor will work as expected as described by the MIPS spec. Some of the signals of the Main Control unit are also fed into the PC Control unit (Jal, Jalr, Jump, BEQ, BNE) and the ALU unit (ALUSrc, ALUOp). The PC Control unit handles getting the next address that the PC register should save for the next cycle of the processor, which will in turn dictate which instruction in instruction memory that is extracted and fed into the remaining components of the processor. The PC Control unit takes the signals of Jal, Jalr, Jump, BNE, and BEQ, as well as ALU's Zero signal and computes from it a selection value to activate a channel in its multiplexer to select whether the PC will increment, jump, or branch. The ALU Control unit takes the signals ALUSrc and ALUOp (alternatively if the instruction handles immediate mode instructions, it will retrieve the opcode from the func bits in the original instruction retrieved from instruction memory) from the Main control unit, and based on those signals, retrieves two data pieces A and B, where A is always from a register, and B could be either from a register or directly from the immediate value embedded in the instruction as selected by ALUSrc, and then based on the ALUOp value the ALU Operation is then selected and then performed. Depending on the operation performed, there are additional signals that are created such as the ALU Zero signal.

B. Control Unit

There are three units for controlling the CPU: The Main Control unit, the ALU Control, and the PC Control. The Main Control unit handles interpreting the opcodes from the current instruction and enabling all of the relevant signals so that the rest of the processor will work as expected as described by the MIPS spec. Some of the signals of the Main Control unit are also fed into the PC Control unit (Jal, Jalr, Jump, BEQ, BNE) and the ALU unit (ALUSrc, ALUOp). The PC Control unit handles getting the next address that the PC register should save for the next cycle of the processor, which will in turn dictate which instruction in instruction memory that is extracted and fed into the remaining components of the processor.

TABLE I
Instructions Opcode

000001	Lws
000011	Cas
000111	Llb
111000	Jal
111110	Jalr
100011	lw
101011	sw

Here are the logic equations from our program.
RegDest = R-type + LWS + CAS + JALR
BNE = BNE
BEQ = BEQ
MemWrite = SW
MemRead = (lw + LWS + LLB)
MemToReg = (lw + LWS + LLB)
RegWrite = ! (sw + beq + bne + j)
ALUSrc =!(R-Type + beq + bne + LWS + CAS)
ExtOp = !(andi + ori + xori + LLB)

C. ALU Control Unit

II. Simulation and Testing

A. The Whole System

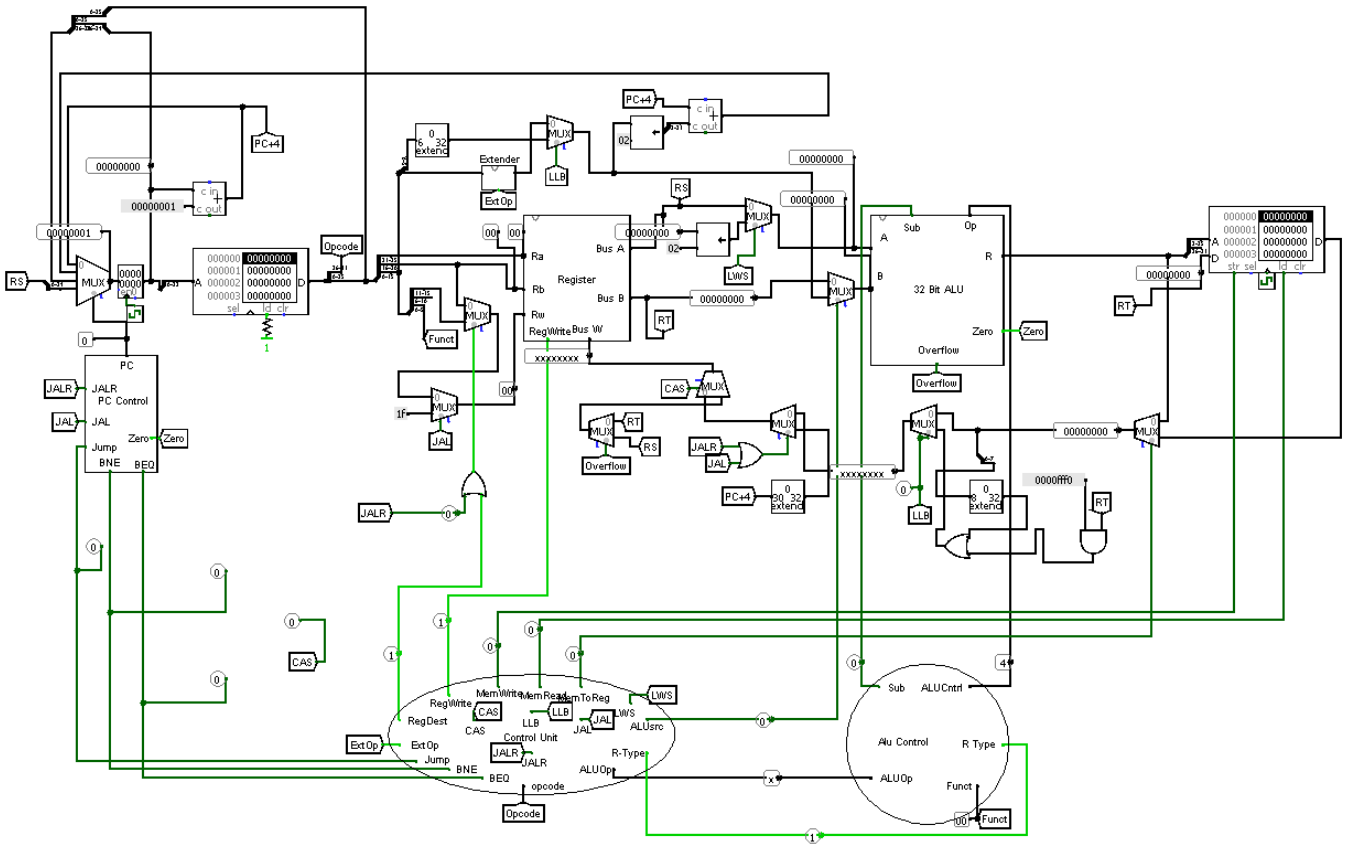

Fig. 1. Datapath

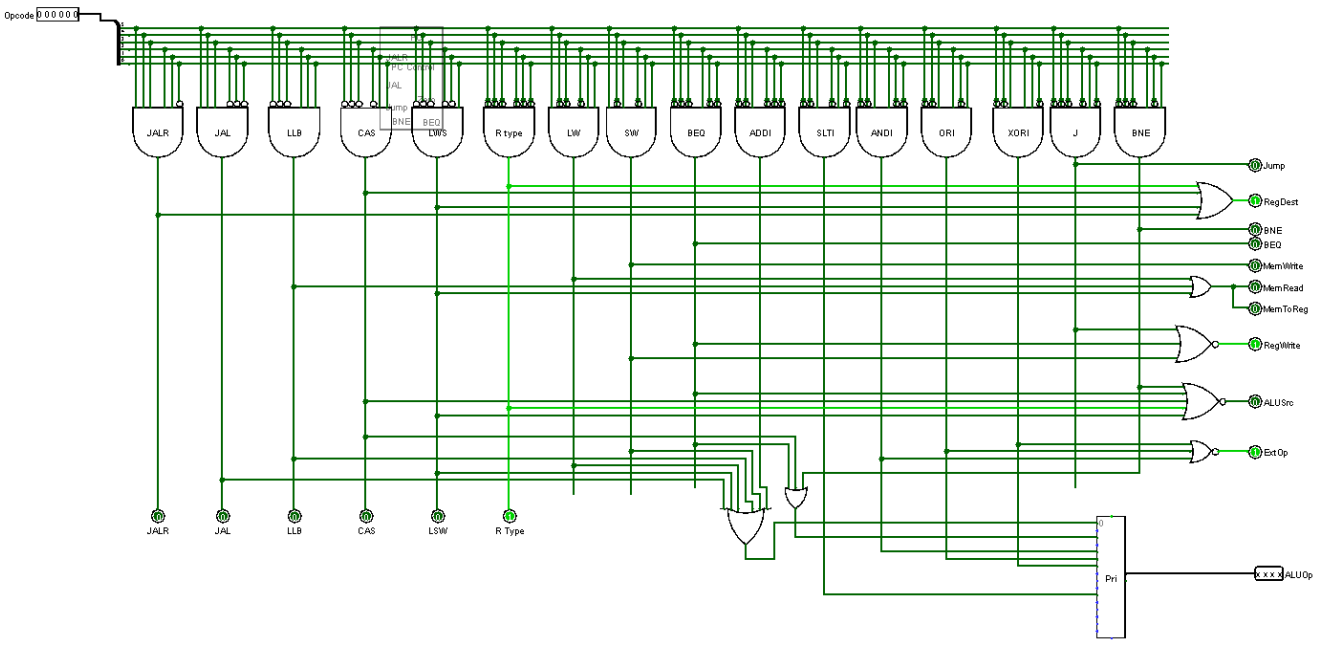

Fig. 2. Control Unit

TABLE II
CONTROL SIGNALS

Op	RegDst	RegWrite	ExtOp	ALUSrc	Beq	Bne	J	MemRead	MemWrite	MemToReg
R-type	1 = Rd	1	x	0=BusB	0	0	0	0	0	0
addi	0 = Rt	1	1=sign	1=Imm	0	0	0	0	0	0
slti	0 = Rt	1	1=sign	1=Imm	0	0	0	0	0	0
andi	0 = Rt	1	0=zero	1=Imm	0	0	0	0	0	0
ori	0 = Rt	1	0=zero	1=Imm	0	0	0	0	0	0
xori	0 = Rt	1	0=zero	1=Imm	0	0	0	0	0	0
lw	0 = Rt	1	1=sign	1=Imm	0	0	0	1	0	1
sw	x	0	1=sign	1=Imm	0	0	0	0	1	x
beq	x	0	x	0=BusB	1	0	0	0	0	x
bne	x	0	x	0=BusB	0	1	0	0	0	x
j	x	0	x	x	0	0	1	0	0	x
LWS	1 = Rd	1	x	0=BusB	0	0	0	1	0	1
CAS	1 = Rd	1	x	0=BusB	0	0	0	0	0	x
LLB	0 = Rt	1	x	1=Imm	0	0	0	1	0	1
JAL	x	1	x	x	0	0	1	0	0	x
JALR	1=Rd	1	x	x	0	0	0	0	0	x

B. LWS

Lws $rd, $rs ,$ rt
rd = Mem[4*rs + rt]

For LWS instruction, we added the shift register as shown in the figure to multiply the value in rs by 2, then it is added to the value of rt in the ALU, after that the address is ready and the data from memory is written back to rd.

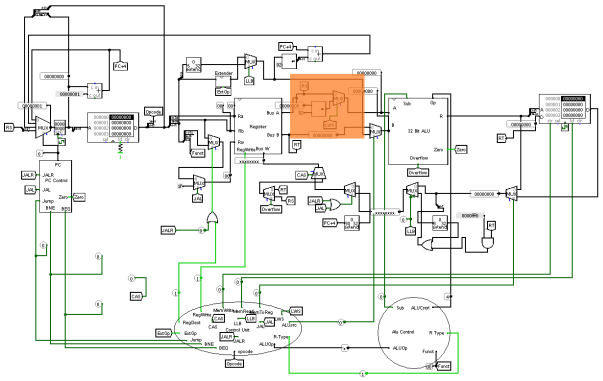

Fig. 3. LWS Datapath

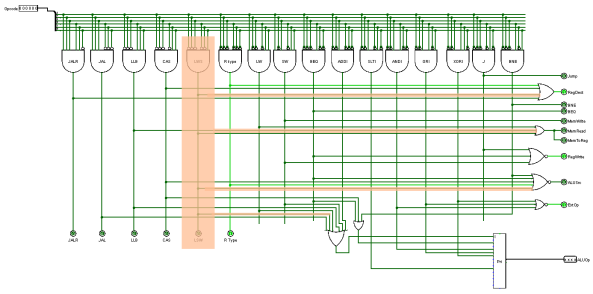

Fig. 4. LWS Control

We used this piece of code and this data in memory to test the functionality of this unit.

Program
This program loads the value 0 to R0 to use it as base in the load instructions, then it loads the value 1 from mem[0] to R1, After that, it loads thre value 8 from mem[1*4] to R2, then, lws is executed where it loads the value of mem[2*4] to R3, Finally R3 is stored in mem[3*4] in order to observe the value after the program.

Assembly:
stli R0, R0, 0 # R0 <- 0
lw R1, 0(R0) # R1 <- mem[0]
lw R2, 4(R0) # R2 <- mem[1]
lws R3, R1, R2 # R3 = Mem[4*R1 + R2]
sw R3, 12(R0) # mem[3] <- R3

Machine Language:
v2.0 raw
00000022 -> 000000 00000 00000 00000 00000 100010
8c010000 -> 100011 00000 00001 00000 00000 000000
8c020004 -> 100011 00000 00010 00000 00000 000100
04221800 -> 000001 00001 00010 00011 00000 000000
ac03000c -> 101011 00000 00011 00000 00000 001100

Memory Data

```
000000 00000001 00000008 00000000 12345678 00000000 00000000
000010 00000000 00000000 00000000 00000000 00000000 00000000
000020 00000000 00000000 00000000 00000000 00000000 00000000
000030 00000000 00000000 00000000 00000000 00000000 00000000
000040 00000000 00000000 00000000 00000000 00000000 00000000
000050 00000000 00000000 00000000 00000000 00000000 00000000
000060 00000000 00000000 00000000 00000000 00000000 00000000
```

Fig. 5. Memory Before

```
000000 00000001 00000008 00000000 00000000 00000000 00000000
000010 00000000 00000000 00000000 00000000 00000000 00000000
000020 00000000 00000000 00000000 00000000 00000000 00000000
000030 00000000 00000000 00000000 00000000 00000000 00000000
000040 00000000 00000000 00000000 00000000 00000000 00000000
000050 00000000 00000000 00000000 00000000 00000000 00000000
000060 00000000 00000000 00000000 00000000 00000000 00000000
```

Fig. 6. Memory After

As you can see the value of mem[3] after the execution holds the value of mem[2] before the executions.

C. CAS

Cas rd, rs, rt
#Reg(Rd) = Max[Reg (Rs) , Reg (Rt)]

CAS instrustion loads the max of Rs and Rt into Rd by subtractions the value and based on the overflow flag the value is written back to the register file, with rd as a destination.

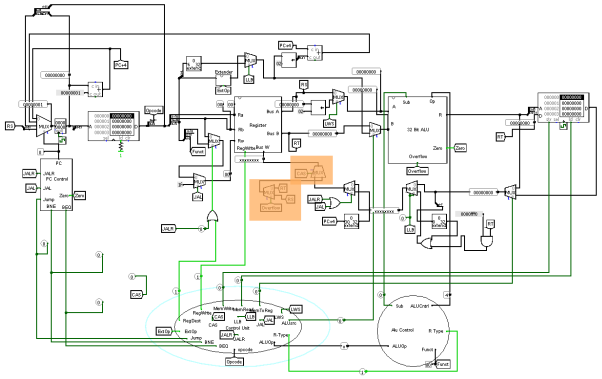

Fig. 7. CAS Datapath

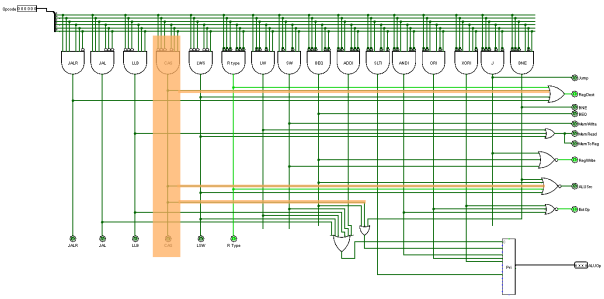

Fig. 8. CAS Control

We used this piece of code and this data in memory to test the functionality of this unit.

Program

This program loads the value 0 to R0 to use it as base in the load instructions, then it loads the value 5 from mem[0] to R1, After that, it loads thre value 6 from mem[1*4] to R2, then, cas is executed and the wrritten back value depends on the overflow flag in the control unit. Finally R3 is stored in mem[2*4] in order to observe the value after the program.

Assembly:

```
stli R0, R0, 0 # R0 <- 0
lw R1, 0(R0) # R1 <- mem[0]
lw R2, 4(R0) # R2 <- mem[1]
cas R3 , R1,R2 #Reg(Rd)=Max[Reg(Rs),Reg(Rt)]
sw R3, 8(R0) # mem[2] <- R3
```

Machine Language:

```
v2.0 raw
00000022 -> 000000 00000 00000 00000 00000 100010
8c010000 -> 100011 00000 00001 00000 00000 000000
8c020004 -> 100011 00000 00010 00000 00000 000100
0c221800 -> 000011 00001 00010 00011 00000 000000
ac030008 -> 101011 00000 00011 00000 00000 001000
```

Memory Data

```
000000 00000005 00000006 00000000 00000000 00000000 00000000
000010 00000000 00000000 00000000 00000000 00000000 00000000
000020 00000000 00000000 00000000 00000000 00000000 00000000
000030 00000000 00000000 00000000 00000000 00000000 00000000
000040 00000000 00000000 00000000 00000000 00000000 00000000
000050 00000000 00000000 00000000 00000000 00000000 00000000
000060 00000000 00000000 00000000 00000000 00000000 00000000
```

Fig. 9. Memory Before

```
000000 00000005 00000006 00000006 00000000 00000000 00000000
000010 00000000 00000000 00000000 00000000 00000000 00000000
000020 00000000 00000000 00000000 00000000 00000000 00000000
000030 00000000 00000000 00000000 00000000 00000000 00000000
000040 00000000 00000000 00000000 00000000 00000000 00000000
000050 00000000 00000000 00000000 00000000 00000000 00000000
000060 00000000 00000000 00000000 00000000 00000000 00000000
```

Fig. 10. Memory After

As you can see the max value 6 is stored in mem[2].

D. LLB

Llb rt, imm(rs)
#Reg (Rt [0 :7]) = Mem(Reg (Rs) + Imm16)

LLB instruction loads the fisrt byte of Mem(Reg (Rs) + Imm16) to re register Rt, by useng and gate to clear the first byte of Rt than concatinating it with the first byte of mem[rs+imm] using or gate and Finally writing it back to Rt.

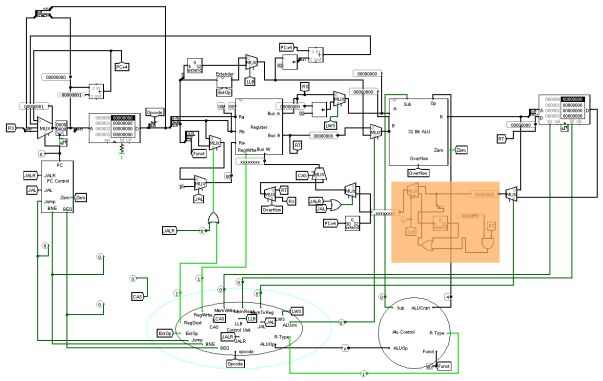

Fig. 11. LLB Datapath

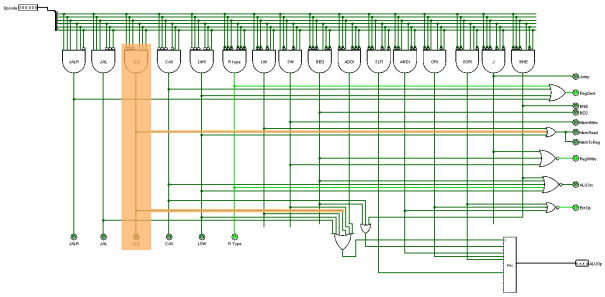

Fig. 12. LLB Control

We used this piece of code and this data in memory to test the functionality of this unit.

Program
This program loads the value 0 to R0 to use it as base in the load instructions, then it loads the value 8 from mem[0] to R1, After that, it loads thre value aaaaaaaa from mem[2*4] to R2, then, llb is executed and the written back value to Rt is the concatination of the fisrt byte of mem[3] bb and the rest of mem[2] aaaaaa. Finally R3 is stored in mem[1*4] in order to observe the value after the program.

Assembly:
stli R0, R0, 0 # R0 <- 0
lw R1, 0(R0) # R1 <- mem[0]
lw R2, 8(R0) # R2 <- mem[2]
llb R2, 4(R1) #Reg (Rt [0:7])=Mem(Reg(Rs)+Imm16)
sw R2, 4(R0) # mem[1] <- R2

Machine Language:
v2.0 raw
00000022 -> 000000 00000 00000 00000 00000 100010
8c010000 -> 100011 00000 00001 00000 00000 000000
8c020008 -> 100011 00000 00010 00000 00000 001000
1c220004 -> 000111 00001 00010 0000 0000 0000 0100
ac020004 -> 101011 00000 00010 00000 00000 000100

000000	00000008	00000000	aaaaaaaa	bbbbbbbb	cccccccc	00000000
000010	00000000	00000000	00000000	00000000	00000000	00000000
000020	00000000	00000000	00000000	00000000	00000000	00000000
000030	00000000	00000000	00000000	00000000	00000000	00000000
000040	00000000	00000000	00000000	00000000	00000000	00000000
000050	00000000	00000000	00000000	00000000	00000000	00000000
000060	00000000	00000000	00000000	00000000	00000000	00000000

Fig. 13. Memory Before

000000	00000008	aaaaaabb	aaaaaaaa	bbbbbbbb	cccccccc	00000000
000010	00000000	00000000	00000000	00000000	00000000	00000000
000020	00000000	00000000	00000000	00000000	00000000	00000000
000030	00000000	00000000	00000000	00000000	00000000	00000000
000040	00000000	00000000	00000000	00000000	00000000	00000000
000050	00000000	00000000	00000000	00000000	00000000	00000000
000060	00000000	00000000	00000000	00000000	00000000	00000000

Fig. 14. Memory After

As you can see the max value aaaaaabb is stored in mem[1].

E. JAL

Ja
R31 = PC + 4 , PC = PC + Imm*4

Jal loads the value of PC + 4 to R31, then PC + Imm*4 is the new PC so the execution continues from there.

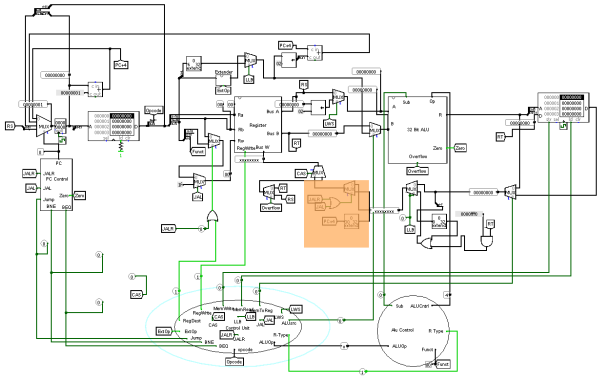

Fig. 15. JAL Datapath

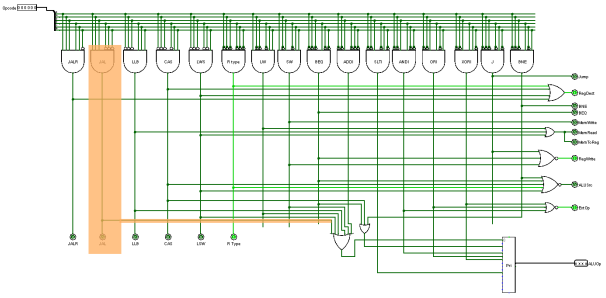

Fig. 16. JAL Control

We used this piece of code and this data in memory to test the functionality of this unit.

Program

This program loads the value 0 to R0 to use it as base in the load instructions, then jal is executed with imm = 3, so the new pc is pc+12 which means that the execution skips 3 instructons and continues after there. The is instruction right after Jal loads the value of mem[0] ti R1, but the one after skipping three instructions is loading mem[1] to R1, in both cases the value is stored at mem[2] so we can observe the difference there. Also the value of R31 is stored to mem[3] to observe it.

Assembly:
stli R0, R0, 0 # R0 <- 0
JAL 3
lw R1, 0(R0) # R1 <- mem[0]
sw R1, 8(R0) # mem[2] <- R1
sw R31, 12(R0) # mem[3] <- R31
lw R1, 4(R0) # R1 <- mem[1]
sw R1, 8(R0) # mem[2] <- R1
sw R31, 12(R0) # mem[3] <- R31

Machine Language:
v2.0 raw
00000022 -> 000000 00000 00000 00000 00000 100010
e0000003 -> 111000 00000 00000 00000 00000 000011
8c010000 -> 100011 00000 00001 00000 00000 000000
ac010008 -> 101011 00000 00001 00000 00000 001000
ac1f000c -> 101011 00000 11111 00000 00000 001100
8c010004 -> 100011 00000 00001 00000 00000 000100
ac010008 -> 101011 00000 00001 00000 00000 001000
ac1f000c -> 101011 00000 11111 00000 00000 001100

Memory Data

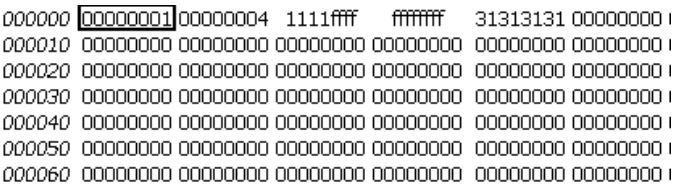

Fig. 17. Memory Before

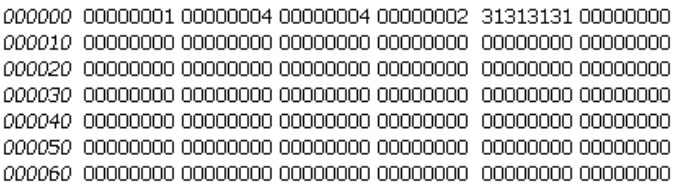

Fig. 18. Memory After

If we look at mem[2] we can see the value 4 which was at mem[1] so the first part is working as expected, Also mem[3] contains 2 which was supposed to be the nexr pc but is loaded to 31 due to jal instructin.
NOTE: 2 means 2*4 since each instruction is 4 bytes long and the memory is using bytes unit to address.

F. JALR

Jalr
rd = PC + 4 , PC = rs

Jal loads the value of PC + 4 to R31, then PC + Imm*4 is the new PC so the execution continues from there.

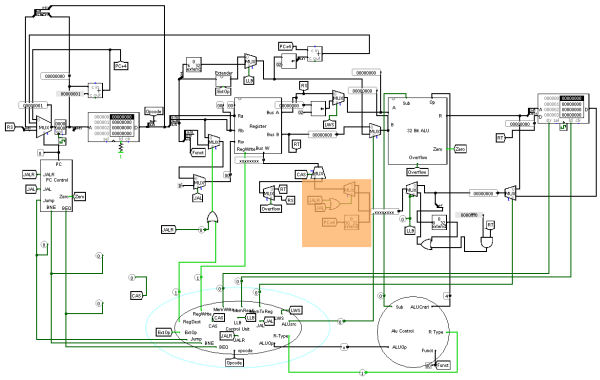

Fig. 19. JALR Datapath

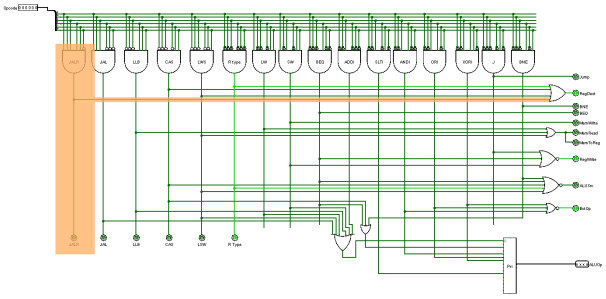

Fig. 20. JALR Control

We used this piece of code and this data in memory to test the functionality of this unit.

Program

This program loads the value 0 to R0 to use it as base in the load instructions, then the value 6 is loaded from mem[0] to R1, after that jalr is executed, so the new pc 6 which means that the execution is moved to the 5th instruction in the instructons below and continues after there.Also, the value of the next pc is moved to R4, The is instruction right after Jalr loads the value of mem[0] to R1, but the one after changing pc to 6 is is loading mem[1] to R3, in both cases the value is stored at mem[2] so we can observe the difference there. Also the value of R4 is stored to mem[3] to observe it.

Assembly:

```
stli R0, R0, 0 # R0 <- 0
lw R1, 0(R0) # R1 <- mem[0]
jalr R1, R4 # R4 = PC + 4 , PC = R1
lw R3, 0(R0) # R3 <- mem[0]
sw R3, 8(R0) # mem[2] <- R3
sw R3, 12(R0) # mem[3] <- R3
lw R3, 4(R0) # R3 <- mem[1]
sw R3, 8(R0) # mem[2] <- R3
sw R4, 12(R0) # mem[3] <- R4
```

Machine Language:

```
v2.0 raw
00000022 -> 000000 00000 00000 00000 00000 100010
8c010000 -> 100011 00000 00001 00000 00000 000000
f8202000 -> 111110 00001 00000 00100 00000 000000
8c030000 -> 100011 00000 00011 00000 00000 000000
ac030008 -> 101011 00000 00011 00000 00000 001000
ac04000c -> 101011 00000 00100 00000 00000 001100
8c030004 -> 100011 00000 00011 00000 00000 000100
ac030008 -> 101011 00000 00011 00000 00000 001000
ac04000c -> 101011 00000 00100 00000 00000 001100
```

Memory Data

```
000000 00000006 00000004 1111ffff ffffffff 00000000 00000000
000010 00000000 00000000 00000000 00000000 00000000 00000000
000020 00000000 00000000 00000000 00000000 00000000 00000000
000030 00000000 00000000 00000000 00000000 00000000 00000000
000040 00000000 00000000 00000000 00000000 00000000 00000000
000050 00000000 00000000 00000000 00000000 00000000 00000000
000060 00000000 00000000 00000000 00000000 00000000 00000000
```

Fig. 21. Memory Before

```
000000 00000006 00000004 00000004 00000003 00000000 00000000
000010 00000000 00000000 00000000 00000000 00000000 00000000
000020 00000000 00000000 00000000 00000000 00000000 00000000
000030 00000000 00000000 00000000 00000000 00000000 00000000
000040 00000000 00000000 00000000 00000000 00000000 00000000
000050 00000000 00000000 00000000 00000000 00000000 00000000
000060 00000000 00000000 00000000 00000000 00000000 00000000
```

Fig. 22. Memory After

If we look at mem[2] we can see the value 4 which was at mem[1] so the first part is working as expected, Also mem[3] contains 3 which was supposed to be the next pc but is loaded to R4 due to jalr instructin.
NOTE: 3 means 3*4 since each instruction is 4 bytes long and the memory is using bytes unit to address.

III. Teamwork

As menttioned above we used a pre-implemented system containig 16 instrucions and the main components, then we adapted it to accept our newly added instructions. So each one of the team almost participated in every thing but to be more specific here is what each one mainly did. For each instructions in the table, the one who made it has done the implementation and the testing.

TABLE III
Work Distribution

LWS	Saleem and Mohammad
CAS	Mohammad
LLB	Mohammad
JAL	Saleem
JALR	Saleem
Report	Mohammad and Saleem

IV. References

References

[1] Course Material.

[2] Exam 1 - Question 1.
https://drive.google.com/file/d/1COfm-1U1u2sZy4apC-Gd8FOYOQZW4EGc/view?usp=sharing

[3] Pre-Implemented System.
https://github.com/sawyermade/architecture/tree/master/project-final