Design Document

Zeen Wang, Jermaine Brown,
Helen Wang, Athena Henderson
Team: Yellow-2122a-01

Rose-Hulman Institute of Technology
CSSE 232 Computer Architecture I
Prof. Williamson
April 3, 2022

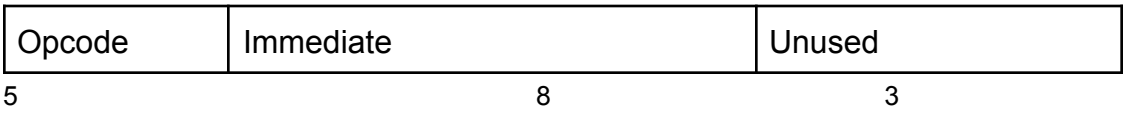
Link to the Google Doc DesignDocument

Description:

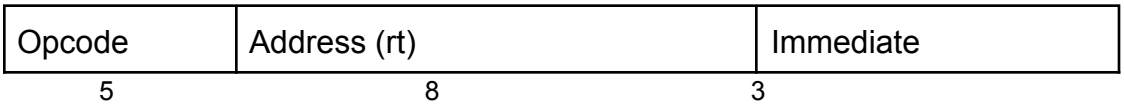
Our design uses a single register accumulator to store data and compare it to inputs. At all times we only use one register and use an allocated space in Memory for data. For our addresses we will use sign extensions to target specific places in memory to receive either data for destination. The input must have the correct first bit to target the proper place in memory.

We are going to use 2 registers, the accumulator($acc) and stack pointer($sp). The accumulator is the only register available by the programmer.

I:

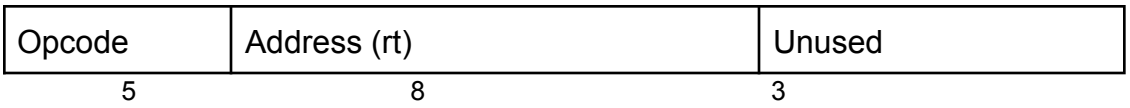


AI:

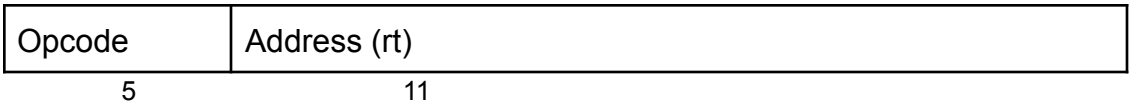


PC relative for bne and beq

A:



J:

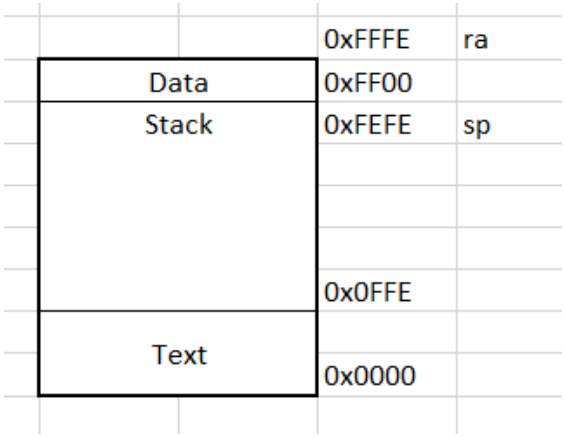


We left shift by 1 bit then we sign extent (the most significant bit will be 1 if it is a data and 0 if it is a instruction)

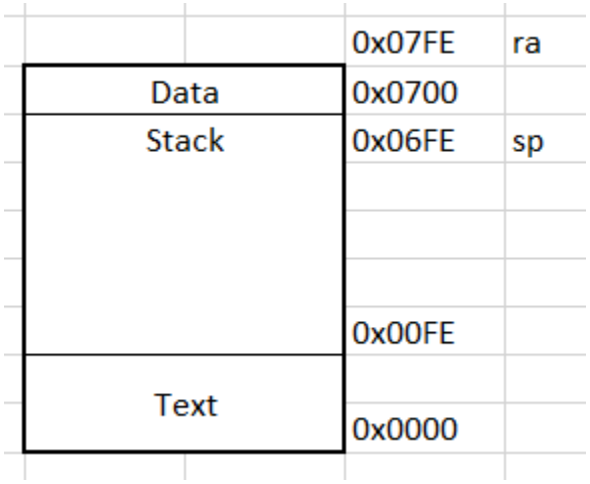
Memory Map:

previous:

Link to the Google Doc DesignDocument



new:



Instructions:

Name	Type	Operation	Opcode
load a	A	acc = Mem[getAddr(rt)]	00001
	Takes an 8 bit address a and loads the value at memory address a to the accumulator, using the address rule. If rt = 0xFD = 11111101 (corresponding address 0xFFFC), then the memory is going to read from input		
save a	A	Mem[getAddr(rt)] = acc	00010
	Take an 8 bit address a and save the value in the accumulator into the memory with address a, using the address rule. If rt = 0xFD, then the memory is going to sent to output		
loadui imm	I	acc = {imm, 8b'0}	00011

Link to the Google Doc DesignDocument

	Takes an 8 bit immediate and load it to the upper 8 bits of the accumulator		
bne a, imm	AI	if(acc != Mem[getAddr(rt)]) PC = PC + 2 + getAddr(imm)	00100
	Takes an 8 bit address and a 3 bit immediate. If the value stored at address a is not equal to the value of the accumulator, then jump to the address calculated from the immediate using the branch address rule.		
beq a, imm	AI	if(acc == Mem[getAddr(rt)]) PC = PC + 2 + getAddr(imm)	00101
	Takes an 8 bit address and a 3 bit immediate. If the value stored at address a is equal to the value of the accumulator, then jump to the address calculated from the immediate using the branch address rule.		
slt a	A	acc = acc < Mem[getAddr(rt)] ? 1:0	00110
	Compare the value in the accumulator with the value stored at address a, if the accumulator is less than a then we set the accumulator to 1, else we set the accumulator to 0.		
slti imm	I	acc = acc < SignExtent(imm) ? 1:0	00111
	Compare the value in the accumulator with the immediate, if the accumulator is less than the immediate then we set the accumulator to 1, else we set the accumulator to 0.		
j a	J	PC = jumpAddr(rt)	01000
	Jump to the instruction with address a, calculated using the address rule.		
jal a	J	Mem[ra] = PC + 2 PC = jumpAddr(rt)	01001
	Jump to the instruction with address a, calculated using the address rule. Store the current PC + 2 to a fixed memory location.		
sw imm	I	Mem[sp + SignExtent(imm)] = acc	01010
	Stored the value in the accumulator onto the stack where it is offset imm to the stack pointer.		
lw imm	I	acc = sp + SignExtent(imm)	01011
	Stored the value from the stack where it is off offset imm to the stack pointer to the accumulator.		

Link to the Google Doc DesignDocument

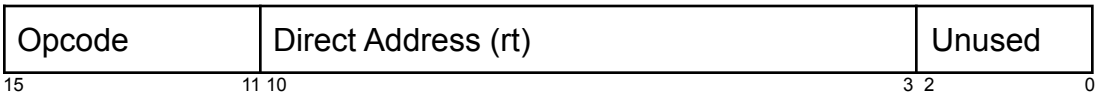
ms	imm	I	sp = sp + SignExtent(imm)	01100
		Move the stack pointer with the sign extended immediate.		
sub	a	A	acc = acc - Mem[getAddr(rt)]	01101
		Subtract the value stored at address a from the accumulator and store the result in the accumulator		
add	a	A	acc = acc + Mem[getAddr(rt)]	01110
		Add the value stored at address a to the accumulator and store the result in the accumulator		
addi	imm	I	acc = acc + SignExtent(imm)	01111
		Add the sign extended immediate to the accumulator and store the result in the accumulator		
and	a	A	acc = acc & Mem[getAddr(rt)]	10000
		And the value stored at address a to the accumulator and store the result in the accumulator		
or	a	A	acc = acc | Mem[getAddr(rt)]	10001
		Or the value stored at address a to the accumulator and store the result in the accumulator		
ori	imm	I	acc = acc | ZeroExtent(imm)	10010
		Or the zero extended immediate to the accumulator and store the result in the accumulator		
loadi	imm	I	acc = SignExtent(imm)	10011
		Load the sign extended immediate to the accumulator.		
getAddr = {7{address[7]}, address, 1'b0} jumpAddr = {4b'0, address, 1'b0} ZeroExtent = {8b'0, imm} SignExtent = {8{address[7]},imm} ra = 0x07FE sp start at 0x06FE				

Address rule: We left shift by 1 bit then we sign extent (the most significant bit will be 1 if it is a data and 0 if it is a instruction)
Branch address: Left shift the immediate by 1, sign extend it to 16 bits then add it to the value of the current PC plus 2.

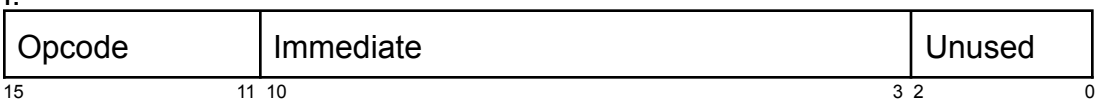
Link to the Google Doc DesignDocument

Types:

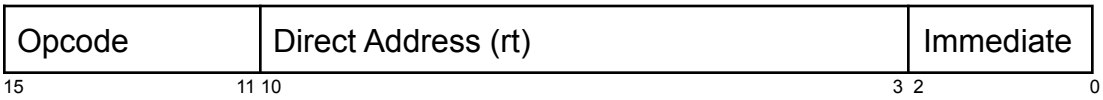
A:



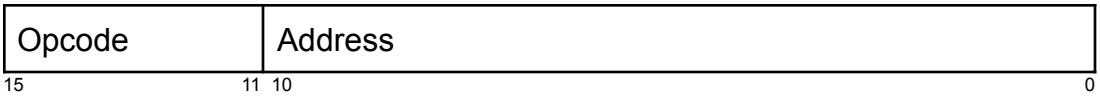
I:



AI:



J:



Call procedure:

For the callers, they are responsible to store the $acc register value, and put the return address on the $acc. For callees, they are responsible to restore the value in the Data memory, and callees will move the stack to store the original value of the data in the stack memory, and restore them back before return. Also, it's callee's responsibility to store the return address in the stack memory and use them for return.

Example program(s):

High Level Code	Assembly	Machine Code	Addresses
n = 6; c = relPrime(n);	loadi 6 save n jal relPrime load m save c	10011 00000110 000 00010 10000100 000 01001 00011000 000 00001 10000011 000 00010 10000111 000	0x 0066 0x 0068 0x 006A 0x 006C 0x 006E

int **relPrime**(int n) {
int m;
m = 2;
while (gcd(n, m) != 1) {
m = m + 1;
}
return m;
}
	relPrime:
	loadi 2
	save m
	ms -6
	loop: load m
	sw 0
	save a
	load n
		
		10011 00000010 000
		00010 10000011 000
		01100 11111010 000
		00001 1000001 000
		01010 00000000 000
		00010 10000000 000
		00001 10000100 000
			
			0x 0030
			0x 0032
			0x 0034
			0x 0036
			0x 0038
			0x 003A
			0x 003C

Link to the Google Doc DesignDocument

	sw 2	01010	00000010	000	0x 003E
	save b	00010	10000001	000	0x 0040
	load ra	00001	11111111	000	0x 0042
	sw 4	01010	00000100	000	0x 0044
	jal gcd	01001	11100111	000	0x 0046
	save o	00010	10000101	000	0x 0048
	lw 0	01011	00000000	000	0x 004A
	save m	00010	10000011	000	0x 004C
	lw 2	01011	00000010	000	0x 004E
	save n	00010	10000100	000	0x 0050
	lw 4	01011	00000100	000	0x 0052
	save ra	00010	11111111	000	0x 0054
	loadi 1	10011	00000001	000	0x 0056
	bne o, end	00100	10000101	100	0x 0058
	load m	00001	10000011	000	0x 005A
	add 1	01110	00000001	000	0x 005C
	save m	00010	10000011	000	0x 005E
	j loop	01000	00011011	000	0x 0060
end:	ms 6	01100	00000110	000	0x 0062
	j ra	01000	11111111	000	0x 0064
int **gcd**(int a, int b) {	gcd:				
if (a == 0) {	loadi 0	10011	00000000	000	0x 0002
return b;	bne a, loop	00100	10000000	010	0x 0004
}	load b	00001	10000001	000	0x 0006
while (b != 0) {	j ra	01000	11111111	000	0x 0008
if (a > b) {	loop:				0x 000A
a = a - b;	loadi 0	10011	00000000	000	0x 000C
} else {	bne b, go	00100	10000001	001	0x 000E
b = b - a;	j end	01000	00010110	000	0x 0010
}	go: load b	00001	10000001	000	0x 0012
}	slt a	00110	10000000	000	0x 0014
return a;	save i	00010	10000010	000	0x 0016
}	loadi 1	10011	00000001	000	0x 0018
	bne i, else	00100	10000010	011	0x 001A
	load a	00001	10000000	000	0x 001C
	sub b	01101	10000001	000	0x 001E
	save a	00010	10000000	000	0x 0020
	j loop	01000	00000101	000	0x 0022
	else: load b	00001	10000001	000	0x 0024
	sub a	01101	10000000	000	0x 0026
	save b	00010	10000001	000	0x 0028
	j loop	00010	00000101	000	0x 002A
	end: load a	00001	10000000	000	0x 002C
	j ra	00010	11111111	000	0x 002E

Link to the Google Doc *DesignDocument*

if (n == 0) {
  n++;
} else {
  n = 2;
}
	loadi 0
	bne n, else
	load n
	add 1
	save n
	j done
	else: loadi 2
	save n
	done:
		10011 00000000 000
		00100 10000100 100
		00001 10000100 000
		01110 00000001 000
		00010 10000100 000
		01000 00001001 000
		10011 00000010 000
		00010 10000100 000
			0x 0002
			0x 0004
			0x 0006
			0x 0008
			0x 000A
			0x 000C
			0x 000E
			0x 0010
			0x 0012

while (n != 0) {
  n = n - m
}
	loop: loadi 0
	beq n, done
	load n
	sub m
	save n
	j loop
	done:
		10011 00000000 000
		00101 10000100 100
		00001 10000100 000
		01101 10000011 000
		00010 10000100 000
		01000 00000001 000
			0x 0002
			0x 0004
			0x 0006
			0x 0008
			0x 000A
			0x 000C
			0x 000E

int count = 0;
for (int i = 0; i < n; i++) {
  count++;
}
	loadi 0
	save count
	save i
	loop: beq n, done
	add 1
	save count
	save i
	j loop
	done:
		10011 00000000 000
		00010 10000110 000
		00010 10000010 000
		00101 10000100 100
		01110 00000001 000
		00010 10000110 000
		00010 10000010 000
		01000 00000100 000
			0x 0002
			0x 0004
			0x 0006
			0x 0008
			0x 000A
			0x 000C
			0x 000E
			0x 0010
			0x 0012

Data: 0xFF00 a(value = m) 0xFF02 b(value = n) 0xFF04 i 0xFF06 m 0xFF08 n 0xFF0A o 0xFF0C count 0xFF0E c		Stack: 0xFEFE	

RTL:

Name	Fetch	Decode		
load a	IR = Mem[PC];	MDR = Mem[SE(IR[10-3]<<1)]	Acc = MDR	

Link to the Google Doc DesignDocument

save a	PC = PC + 2	ALUOut = PC + SE(IR[10-3]<<1)	Mem[SE(IR[10-3]<<1)] = Acc	
loadui imm			Acc = IR[10-3] << 8	
bne a, imm			if(Acc != MDR) PC = ALUOut	
beq a, imm			if(Acc == MDR) PC = ALUOut	
slt a			if(Acc < MDR) Acc = 1 else Acc = 0	
slti imm			if(Acc < SE(IR[10-3])) Acc = 1 else Acc = 0	
j a			PC = ZE(IR[10-0]<<1)	
jal a			Mem[ra] = PC PC = ZE(IR[10-0]<<1)	
sw imm			ALUOut = sp +SE(IR[10-3])	Mem[ALUOut] = Acc
lw imm			ALUOut = sp +SE(IR[10-3])	Acc = Mem[ALUOut]
ms imm			Sp = Sp + (SE(IR[10-3]))	
sub a			Acc = Acc - MDR	
add a			Acc = Acc + MDR	
addi imm			Acc = Acc + (SE(IR[10-3]))	

Link to the Google Doc DesignDocument

and a			Acc = Acc & MDR	
or a			Acc = Acc | MDR	
ori imm			Acc = Acc | (ZE(IR[10-3]))	
loadi imm			Acc = SE(IR[10-3])	

RTL Components

1. **16-bit Registers: reg16**

Input wires:

In: 16-bit bus for Input data,
E: 1-bit Enable wire,
CLK: 1-bit clock wire
Reset:

Output wire:

Out: 16-bit bus for output data

Implemented Symbols: Acc, Sp, PC, IR, ALUOut, MDR

Description: We can write to the register only when E = 1. It will read and write on rising edge

Test implementation: Set reset to 1 for 10 cycles then switch it back to one. Test if the output is 0. Set input to 1 while keeping the enable as 0, the output should be 1. Change enable to 1, now the output should be 1. Change the input to 2 between the rising edge and falling edge, the next falling edge should still read 1, while the next rising edge should read 2.

2. **Memory: mem16**

Input wires:

Addr: 16-bit bus for Memory Address,
In: 16-bit bus for Input data,
W:1-bit Memory write control wire,
CLK: 1-bit clock wire

Output wire:

Out: 16-bit bus for output data

Implemented Symbols: Mem

Description: It will write on rising edge
Test implementation: Give a 16-bit address and set read and write to 0, should return 0. Set read to 1, the output should be the data at the given address. Set input data to a random 16 bit number then set read to 0, write to 1, set read to 1, the output should be the new data.

3. **ALU: alu16**

Input wires:

A: 16-bit bus for Input A,
B: 16-bit bus for Input B,
OP: 3-bit bus for ALUopcode,

Output wire:

Out: 16-bit bus for output data
Zero: 1-bit zero wire (high for 0)

Implemented Symbols: ALU

Opcode:

000 for And
001 for Or
010 for Add
011 for Sub
100 for Sl

Test implementation: Input 2 16-bit numbers and change the opcode every cycle, the output should be the result of and, or, add, sub, and slt. Give 0x1111 and 0x1111l and opcode 11, the zero output should be 1. Give 0xFFFF and 0x0001 that will cause an overflow and opcode 00, the overflow output should be 1.

4. **1-bit Mux: mux1b16**

Input wires:

A: 16-bit bus for Input A,
B: 16-bit bus for Input B,
OP:1-bit bus for opcode

Output wire:

Out: 16-bit bus for output data

Implemented Symbols: 1bMux

Opcode:

0 for A
1 for B

Test implementation: Give two different 16-bit numbers, change the opcode every cycle and check the output

5. **1-bit Mux: mux1b16**

   Input wires:

   A: 16-bit bus for Input A,
   B: 16-bit bus for Input B,
   C: 16-bit bus for Input C,
   D: 16-bit bus for Input D,
   OP:2-bit bus for opcode

   Output wire:

   Out: 16-bit bus for output data

   Implemented Symbols: 1bMux

   Opcode:

   0 for A
   1 for B
   10 for C
   11 for D

   Test implementation: Give four different 16-bit numbers, change the opcode every cycle and check the output

6. **16-bit Sign Extend: se16**

   Input wires:

   In: 8-bit bus for Input,

   Output wire:

   Out: 16-bit bus for output data

   Implemented Symbols:

   Test implementation: Give a 8-bit number that starts with 1, should fill the upper bits with 1. Give an 8-bit number that starts with 0, should fill the upper bits with 0.

7. **16-bit Zero Extend: ze16**

   Input wires:

   In: 8-bit bus for Input,

   Output wire:

   Out: 16-bit bus for output data

   Implemented Symbols:

   Test implementation: Give different 8-bit numbers, some starting with 1 and some starting with 0. The zero-extender should fill the upper bits with 0.

8. **1-bit Mux mux1b1**

   Input wires:

   A: 1-bit for Input A,

B: 1-bit for Input B,
OP:1-bit for opcode

Output wire:

Out: 1-bit for output data

Implemented Symbols:

Opcode:

0 for A
1 for B

Test implementation: Give different 1-bit numbers, set op to 1, then set op to 0

Integration Plan:

1. **PC:**

The first one will be small, including a 2-bit mux and PC ensuring that the mux is properly selecting the correct inputs based on the PCSrc and that the PC is changing when it should. When doing beq we will set Branch to 1 and bne/beq to 1. When doing bne we will set Branch to 1 and bne/beq to 0. This and PCWrite will select when PC will be changed. PC will also handle shifting the ZE IR input it receives.

Input Wires:

2-bit PCSrc wire

1-bit Zero? Input

1-bit Branch? wire

1 bit bne/beq wire

16-bit bus ALU

16-bit bus ALU Out

bus IR ZE

1-bit PCWrite wire

1-bit clock wire

Output Wires:

16-bit bus for PC out

bus IR ZE << 1

Link to the Google Doc DesignDocument

Tests:

Give 3 separate values for the inputs and for each value of PCSrc the clock will do 2 cycles to test when PCWrite is off or on. When PCWrite is on the output should be the selected input from the mux and if PCWrite is off the output should be the previous cycle's output. Next, set PCWrite to 0, Branch to 1 and bne/beq to 1 and check the output. Then switch bne/beq = 0 and check the output.

2. **Memory:**

The Second will be MEM, a 1-bit mux, 2-bit mux, IR, and MDR. Once again checking the mux functionality. However, this subsystem will ensure that the control of memwrite and memread are working properly with Mux outputs and that the memory block of our Datapath is working.

Input Wires:

16-bit bus IO Input

bus IR SE << 1

16-bit bus ACC

16-bit bus ALU Out

16-bit bus PC

1-bit reset wire

1-bit clock wire

Output Wires:

bus for IR out

16-bit bus for MDR out

16-bit bus for Memory out

1-bit PCWrite wire

1-bit Branch wire

1-bit bneOrbeq wire

1-bit PCSrc wire

1-bit AccWrite wire

1-bit SpWrite wire

1-bitALUSrcA wire

1-bit ALUSrcB wire

3-bit bus ALUOp

16-bit bus for IO Output

Tests:

We will once again create 4 unique values for the bus inputs to use for this test. MemAddr will increment every 8 cycles while MemData increments every 4 clock cycles to test MemRead and MemWrite together for all possible combinations of the selected values.

3. **Wires:**

The Third subsystem will take 3 inputs and test output for the SE <<1, SE, ZE, <<8, the ACCSrc 3-bit mux, ACC, and SP. This subsystem ensures a functional modification of data via bit shifts and Zero and Sign extensions, and that this modified data is properly selected and stored in the ACC and SP when it should be.

Input Wires:

3-bit ACCSrc wire

1-bit ACCWrite wire

1-bit SPWrite wire

bus IR

bus MDR

bus Memory Out

1-bit clock wire

Output Wires:

16-bit bus for ACC out

16-bit bus for SP out

bus for IR SE <<1 out

bus for IR SE out

bus for IR ZE output data

Tests:

The main focus of testing the subsystem is not the ZE, SE, or bit shift since they are tested individually. The tests will increment through selections for ACCSrc every 2 cycles to alternate ACCWrite and confirm the proper outputs. At the same time SPWrite will alternate for a secondary check.

4. **ALUPath:**

The Fourth subsystem will include the ALUSrcA, ALUSrcB, ALU, and ALU out. This subsystem guarantees the ALU operations being performed correctly, and the output is stored with no issue.

Input Wires:

2-bit ALUSrcA wire

3-bit ALUSrcB wire

16-bit bus PC

16-bit bus ACC

16-bit bus SP

16-bit bus MDR

bus IR SE

bus IR ZE

bus IR SE << 1

3-bit bus for ALUopcode

Output Wires:

16-bit bus for ALU out

16-bit bus for ALU Out output

1-bit Zero? output data

Tests:

2 of the 6 inputs are going to be the same to test Zero?. But the 6 inputs will remain the same through the test. We'll increment ALUSrcA every time we try

every op code with every selection in ALUSrcB to assure that the ALUOut and ALU are working with proper selection. When we select subtract in the opcode and the 2 inputs that are valued the same we will check that Zero gives a 1 as an output.

5. **Overall Integration**

   Finally you test the four subsystems left as one whole system by connecting the Outputs to the inputs in order that the subsystems are listed like output of PC to inputs of Memory as shown in the Datapath. Lastly connect the last output to the inputs of the PC subsystem. With tests written for the subsystems the tests for the integration are extremely similar.

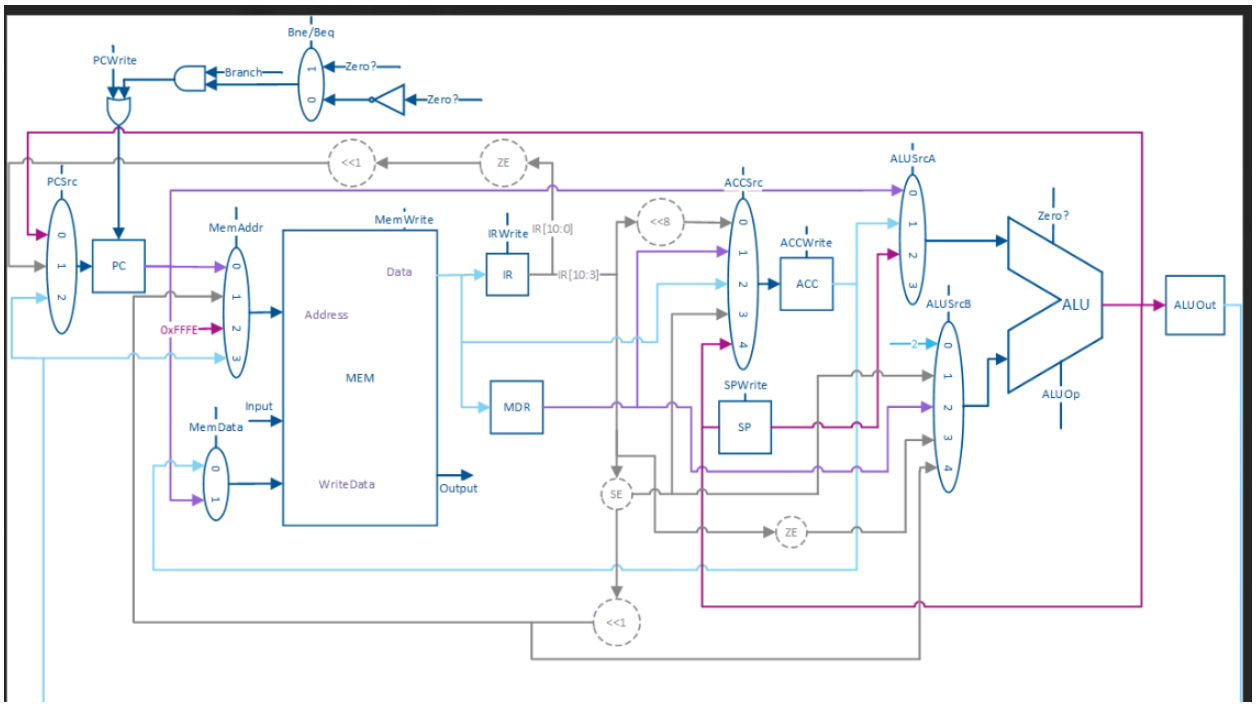
Input/Output/Control:

Name:	Bit Size:	I/O/C	Description:
PCSrc	2	Control	Used to determine between ALU output, the zero-extended immediate shifted by 1, and ALUOut for the input to PC.
MemAddr	2	Control	Used to determine between PC, the sign-extended immediate shifted by 1, 0xFFFE, and ALUOut for the address we are writing to in memory
MemData	1	Control	Used to determine between Acc output and PC for input to memory
MemWrite	1	Control	Is 1 when we are writing to memory, 0 otherwise
SPWrite	1	Control	Is 1 when we are writing to Sp, 0 otherwise
ACCSrc	3	Control	Used to determine between the immediate shifted by 8, MDR output, data from memory, the sign-extended immediate, or Sp output for the input to the accumulator
ACCWrite	1	Control	Is 1 when we are writing to the accumulator, 0 otherwise
ALUSrcA	2	Control	Used to determine between PC, Acc output, and Sp output for the input to A
ALUSrcB	3	Control	Used to determine between 2, the sign-extended immediate, MDR output, the zero-extended immediate, and the sign-extended immediate shifted by 1 for the input to for B
Zero?	1	Control	Flag that is set to 1 when the output of the ALU is zero
ALUOp	3	Control	Used to determine which operations should be done by the ALU [000 - And,0 01 - Or, 010 - Add, 011 - Sub, 100 - Slt]
Bne/Beq	1	Control	Used to determine what control bit gets passed into the PC to determine exactly when to write to the PC (specifically for determining bne/beq) [1 - Beq, 0 - Bne]

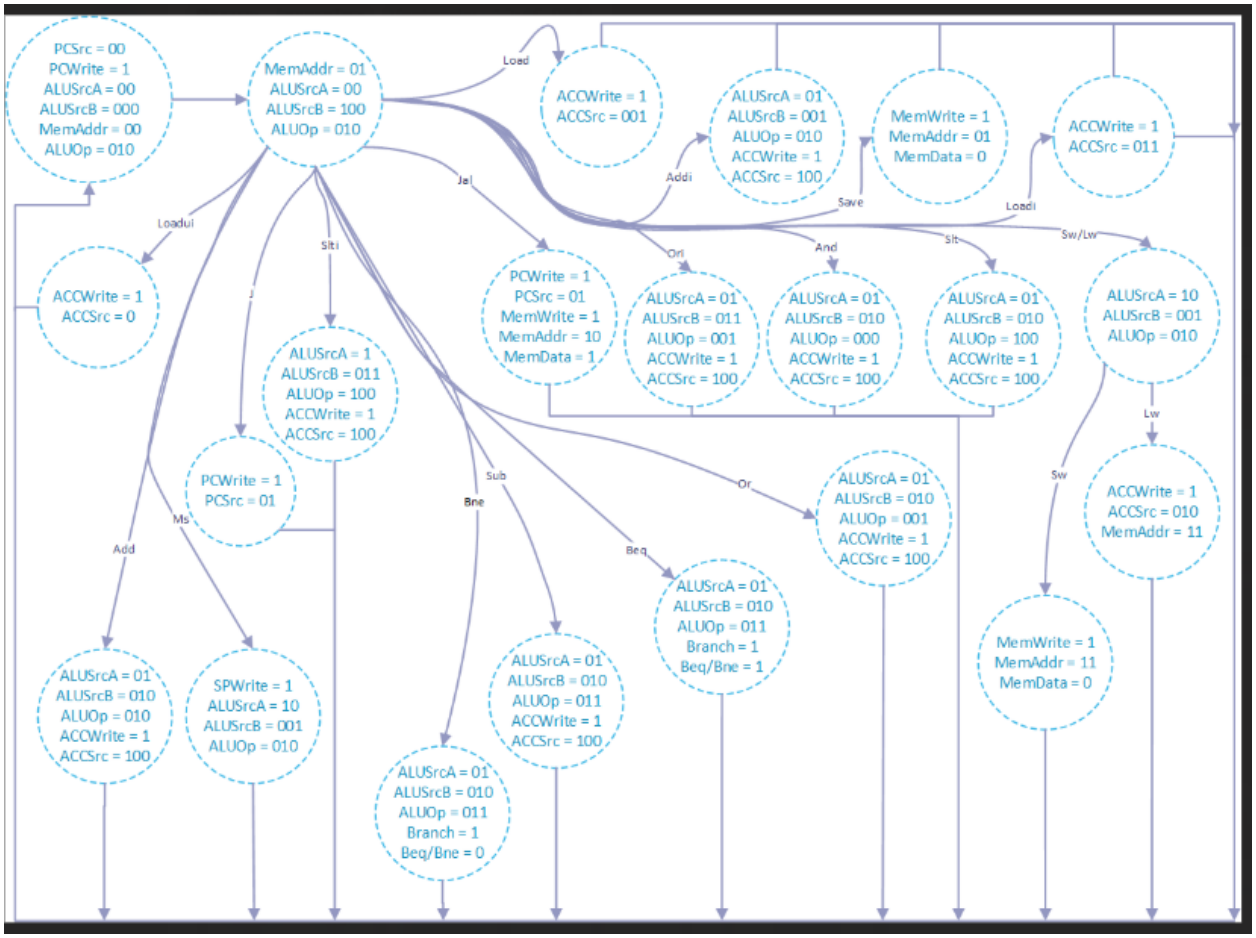
Link to the Google Doc DesignDocument

PCWrite	1	Control	Used to determine what control bit gets passed into the PC to determine exactly when to write
Branch	1	Control	Used to determine what control bit gets passed into the PC to determine exactly when to write (specifically if it's a branch instruction)
IRWrite	1	Control	Used to determine whether we can write to IR or not

Datapath:



Link to the Google Doc DesignDocument



All read/write enable bits is 0 if not mentioned

Datapath test implementation

constant io = 1111 1101
constant a = 1000 0000 //a = 1
constant b = 1000 0001 //b = 10

//Test reset

Loadi 1
Save a
Save io

//Test I/O
load io
save io

//Test load/save

Link to the Google Doc DesignDocument

```
load io
save a
load io
save b
load a
save io

//Test loadui
loadui 8
save io

//Test loadi
loadi 20
save io

//Test bne (should print b)
load a
bne b 1
save io
load b
save io

//Test beq (should print b)
load a
beq a 1
save io
load b
save io

//Test slt
load a
slt b
save io
load b
slt a
save io

//Test slti
load a
slti 0
save io
load a
slti 1000 // a number greater than a
save io
```

```
//Test j
load a
j TAG
save io
TAG:
load b
save io

//Test jal (should output b)
load a
jal TAG
save io
TAG:
load b
j ra

//Test sw/lw/ms
ms 4
load a
sw 0
load b
sw 2
loadi 0
lw 0
save io
lw 2
save io

//Test sub
load a
sub b
save io

//Test add
load a
add b
save io

//Test addi
load a
addi 1
save io
load a
```

```
addi -1
save io

//Test and
load a
and b
save io

//Test or
load a
or b
save io

//Test ori
load a
ori 0x0F
save io
```