Project 2

Anna Moreno
Vaishnavi Medikundam
Syeda Ali

Part A) ISA intro

1. Introduction.

Name of Architecture: AVS which is the first letter of all of our names; Anna, Vaish, Syeda

Overall Philosophy: Have an efficient ISA Design for two programs completed in the first project.

Specific goals strived for and achieved:

Modular Exponentiation

Best match score and count

2. Instruction list. Give all the instructions, their formats, opcodes, and an example.

Instruction	**Functionality**	**Opcode**	**Parity Bit**
loadi *imm*	$R4 = Mem[imm]	011 iiii	0
load Rx	Rx = $R4	01110 xx	1
blt4 *Rx*	$R5 = $R4 (<) Rx	00001 xx	0
beq4 *Rx*	$R5 = $R4 (==) Rx	10000 xx	1
store *Rx, Ry*	Mem[Ry] = Rx	000 xx yy	1
srl Rx	Rx shift right one bit, 0 shifted into MSB	001 00 xx	0
addi Rx, *imm*	Rx = Rx + imm	100 xx ii	0
subi Rx	Rx = Rx + (-imm)	101 10 xx	0
bne Rx	$R5 = Rx (!=) 0	110 11 xx	1
slt Rx	$R5 = Rx (<) 0	100 01 xx	1
BezDec *imm*	If $R5 == 0, then PC = PC + imm, else $R5 = $R5 – 1, PC = PC + 1	0100 iii	0
xor *Rx, Ry*	$R5 = Rx (EXCL) with Ry	110 xx yy	0
andi *Rx, imm*	$R5 = Rx (AND) with imm	111 xx ii	0

andi5 *imm*	$R5 = $R5 (AND) with imm	11110 ii	1
srl5 *imm*	$R5 shift right *imm* bits, 0 shifted into MSBs	01111 ii	1
jump *'branch'*	PC = PC - imm	010 iiii	1
add Rx, Ry	Rx = Rx + Ry	001 xx yy	1
sub Rx, Ry	Rx = Rx – Ry	101 xx yy	1
subIn $R4	$R4 = $R4 - 1	0110110	1
bne $R4	$R5 = $R4 (!=) 0	1110000	1
jump 'first branch'	PC = 12	1010101	0
halt	Stop	000 00 00	0

3. Register design. How many registers are supported? Is there anything special about the registers?

R0, R1, R2, R3, R4, R5
R4 and R5 are defined within the program and cannot be manipulated by the user

4. Control flow (branches). What types of branches are supported? How are the target addresses calculated? What is the maximum branch distance supported? Give examples of your assembly branch instructions and their corresponding machine code.

The following branches are supported (please see table below). The target addresses are calculated using logical instructions. The 1 or 0 bit determine the branch distance and if the instruction will branch out at all. The maximum branch distance supported per instruction is 15 lines at a time using the instruction " jump 'branch' ".

Instruction	**Functionality**	**Opcode**	**Parity Bit**
blt4 *Rx*	$R5 = $R4 (<) Rx	00001 xx	0
beq4 *Rx*	$R5 = $R4 (==) Rx	10000 xx	1

bne Rx	$R5 = Rx (!=) 0	110 11 xx	1
slt Rx	$R5 = Rx (<) 0	100 01 xx	1
BezDec *imm*	If $R5 == 0, then PC = PC + imm, else $R5 = $R5 – 1, PC = PC + 1	0100 iii	0
jump *'branch'*	PC = PC - imm	010 iiii	1
bne $R4	$R5 = $R4 (!=) 0	1110000	1
jump 'first branch'	PC = 12	1010101	0

Examples of assembly branch instructions and corresponding machine code:

- BezDec 7 0100 111
- jump *'next2'* 010 0111

5. Data memory addressing modes. What addressing modes are supported for data memory? How are the addresses calculated? Give examples of your assembly load / store instructions and their corresponding machine code.

The addressing modes supported for data memory are:

Instruction	**Functionality**	**Opcode**	**Parity Bit**
loadi *imm*	$R4 = Mem[imm]	011 iiii	0
load Rx	Rx = $R4	01110 xx	1
store *Rx, Ry*	Mem[Ry] = Rx	000 xx yy	1

The addresses are calculated manually for each instruction and are converted to bits from MIPS assembly.

Examples of assembly load / store instructions and their corresponding machine code:

- loadi 0(0x2000) 011 0000
- store $t1, 0(0x2004) 000 0100

Part B) Answers to questions

1. Comparing to the sample of "My_straightforward_ISA", what are the unique features of your ISA? Explain why your ISA is better.

A unique feature is that we have a variety of logic based instructions. Our ISA is better because we use parity bits.

2. In what ways did you optimize for the two goals? If you optimized for anything additional, what and how?

We optimized for the two goals by using a limited of registers so that the code would be more concise and organized in addition to using parity bits which helped reduce significant technical errors.

3. What would you have done differently if you had 1 more bit for instructions? How about 1 fewer bit?

We would've used them in immediate values so that we can implement bigger numbers. If we had one fewer bit we would try to store most immediate values into registers before implementing them.

4. How did your team work together to accomplish this project? (Role of each team member, progress milestones, time spent individually and together?)

We were in constant communication on a similar platform that allowed us to tackle jobs separately but had clarity throughout our issues.

5. If you had a chance to restart this project afresh with 3 weeks' time, how would your team have done differently?

We could have had worked to obtain a more simplified ISA and would have reduced the lines significantly in our part 2 code.

Part C) Software package:

1. Algorithms (in assembly code) of the two programs. Make sure your assembly format is either obvious or well described, and that the code is well commented.
Extra credits: provide a convincing estimation:
i. on the dynamic instruction count for P1 (ME) with P = ~1000 and Q = ~500
ii. on the worst-case scenario of dynamic instruction count for P2(BMC).

2. Machine code for each of the programs. We will not correct/grade the machine code. You will also not be able to verify whether your code works correctly or not in this project (without a simulator). Therefore, you have to rely on the help of the disassembler, strive to make your algorithm simple and easy to understand, as well as pursue the sw-hw "codesign" – this will avoid putting tremendous complexity at either the software or the hardware end.

Algorithm and Machine Code for Program 1:

```
#Assume everything is equal to zero at first
#$t0 = 00
```

#$t1 = 01
#$t2 = 10
#$t3 = 11

addi $t0, 1	0 100 00 01
addi $t1, 1	0 100 01 01
loadi 0(0x2000)	0 011 0000
addi $t2, 3	0 100 10 11
addi $t2, 2	0 100 10 10
addi $t3, 3	0 100 11 11
addi $t3, 3	0 100 11 11
addi $t3, 3	0 100 11 11
addi $t3, 3	0 100 11 11
addi $t3, 3	0 100 11 11
addi $t3, 2	0 100 11 10
loop:	
bne $s0	1 1110000
BezDec 7	0 0100 111
next:	
bne $t2	1 110 11 10
BezDec 4	0 0100 100
add $t1, $t0	1 001 01 00
subi $t2	0 101 10 10
jump ‘*next*’	0 010 1000
next2:	
sub $t3, $t1	1 101 11 01
bne $s0	1 1110000
BezDec 7	0 0100 111
slt $t3	1 100 01 11
add $t3, $t1	1 001 11 01
BezDec 3	0 0100 011
sub $t1, $t3	1 101 01 11
jump ‘next2’	0 010 0111
down:	
addi $t2, 3	0 100 10 11
addi $t2, 2	0 100 10 10
bne $s0	1 1110000
BezDec 5	0 0100 101
subIn $s0, 1	1 0110110

sub $t0, $t0	1 101 00 00
add $t0, $t1	1 001 00 01
jump *'loop'*	0 1010101
exit:	
store $t1, 0(0x2004)	1 000 0100
halt	0 000 0000

Algorithm and Machine Code for Program 2:

#Assume everything is equal to zero at first
#$t0 = 00
#$t1 = 01
#$t2 = 10
#$t3 = 11

addi $t0, 3	0 100 00 11
addi $t0, 3	0 100 00 11
addi $t0, 3	0 100 00 11
addi $t0, 3	0 100 00 11
addi $t0, 3	0 100 00 11
addi $t0, 3	0 100 00 11
addi $t0, 2	0 100 00 10
addi $t1, 3	0 100 01 11
addi $t1, 3	0 100 01 11
addi $t1, 3	0 100 01 11
addi $t1, 3	0 100 01 11
addi $t1, 3	0 100 01 11
addi $t1, 3	0 100 01 11
addi $t1, 3	0 100 01 11
addi $t1, 3	0 100 01 11
addi $t1, 3	0 100 01 11
addi $t1, 3	0 100 01 11
addi $t1, 3	0 100 01 11
addi $t1, 2	0 100 01 10
next:	
loadi 0(0x200C)	0 011 1100
next2:	
loadi 0(0x2000)	0 011 0000
load $t2	1 01110 10
subi $t1	0 101 10 01
xor $t2, $t3	0 110 10 11

sub $t0, $0	1 101 00 00
bne $t0	1 110 11 00
BezDec 7	0 0100 111
next3:	
bne $t1	1 110 11 01
BezDec 5	0 0100 101
subi $t1	0 101 10 01
andi5 1	1 11110 01
srl5 1	1 01111 01
jump ‘next3’	1 010 0101
next4:	
bne $t0	1 110 11 00
BezDec 7	0 0100 111
loadi 0(0x2010)	0 011 0000
load $t3	1 01110 11
subi $t0	0 101 10 00
loadi 0(0x2004)	0 011 0100
blt4 $t1	0 00001 01
bne $t1	1 110 11 01
bne $t0	1 110 11 00
BezDec 7	0 0100 111
BezDec 5	0 0100 101
beq4 $t1	1 1000 01
bne $t1	1 110 11 01
BezDec 7	0 0100 111
jump ‘first branch’	0 1010101
score:	
sub $t2, $t2	1 101 10 10
bne $t0	1 110 11 00
BezDec 7	0 0100 111
addi $t2, 1	0 100 01 01
store $t1, 0(0x2004)	1 000 0100
bne $t1	1 110 11 01
BezDec 3	0 0100 011
store $t2, 0(0x2008)	1 000 1000
jump ‘first branch’	0 1010101
best count:	
sub $t2, $t2	1 101 10 10
addi $t2, 1	0 100 01 01

bne $t0	1 110 11 00
BezDec 3	0 0100 011
store $t2, 0(0x2008)	1 000 1000
jump ‘first branch’	0 1010101
exit:	
halt	0 000 0000

3. Output of your Python disassembler for each program. This should be a line-by-line explanation of the machine code, what is done by each line of code.

P1

```
ECE 366 Group 8
 2 = disassembler
Please enter the mode of Program: 2
Mode selected:  ECE 366 Group 8 Disassembler
----------
Machine code: 01000001

addi R0, 1
----------
Machine code: 01000101

addi R1, 1
----------
Machine code: 00110000

load 0
----------
Machine code: 01001011

addi R2, 3
----------
Machine code: 01001010

addi R2, 2
----------
Machine code: 01001111

addi R3, 3
----------
Machine code: 01001111

addi R3, 3
----------
Machine code: 01001111

addi R3, 3
----------
Machine code: 01001111

addi R3, 3
----------
Machine code: 01001111

addi R3, 3
----------
Machine code: 01001110
```

addi R3, 2

Machine code: 11110000

bne $R4

Machine code: 00100111

bezDec 7

Machine code: 11101110

bne R3

Machine code: 00100100

bezDec 4

Machine code: 10010100

add R1, R0

Machine code: 01011010

subi R2

Machine code: 10101000

jump1,8

Machine code: 11011101

sub R3, R1

Machine code: 11110000

bne $R4

Machine code: 00100111

bezDec 7

Machine code: 11000111

slt R3

Machine code: 10011101

add R3, R1

Machine code: 00100011

bezDec 3

Machine code: 11010111

sub R1, R3

Machine code: 10100111

jump1,7

Machine code: 01001011

addi R2, 3

Machine code: 01001010

addi R2, 2

Machine code: 11110000

bne $R4

Machine code: 00100101

bezDec 5

Machine code: 10110110

subln $R4

Machine code: 11010000

sub R0, R0

Machine code: 00010001

srl R1

Machine code: 01010101

jump 'first branch'

Machine code: 10000100

store R1, R0

Machine code: 00000000

halt
---
disassembler is being done

P2 output

ECE 366 Group 8
 2 = disassembler
Please enter the mode of Program: 2
Mode selected: Please enter the which program 1 or 2: 2
Mode selected: ECE 366 Group 8 Disassembler

Machine code: 01000011

addi R0, 3

Machine code: 01000011

addi R0, 3

Machine code: 01000011

addi R0, 3

Machine code: 01000011

addi R0, 3

Machine code: 01000011

addi R0, 3

Machine code: 01000011

addi R0, 3

Machine code: 01000010

addi R0, 2

Machine code: 01000111

addi R1, 3

Machine code: 01000111

addi R1, 3

Machine code: 01000111

addi R1, 3

Machine code: 01000111

addi R1, 3

Machine code: 01000111

addi R1, 3

Machine code: 01000111

addi R1, 3

Machine code: 01000111

addi R1, 3

Machine code: 01000111

addi R1, 3

Machine code: 01000111

addi R1, 3

Machine code: 01000111

addi R1, 3

Machine code: 01000110

addi R1, 2

Machine code: 00111100

loadi 12

Machine code: 00110000

loadi 0

Machine code: 10111010

load R2

Machine code: 01011001

subi R1

Machine code: 01101011

xor R2,R3

Machine code: 11010000

sub R0, R0

Machine code: 11101100

bne R0

Machine code: 00100111

bezDec 7

Machine code: 11101101

bne R0

Machine code: 00100101

bezDec 5

Machine code: 01011001

subi R1

Machine code: 01111001

xori R2, R1

Machine code: 10111101

srl5 1

Machine code: 10100101

jump 5

Machine code: 11101100

bne R2

Machine code: 00100111

bezDec 7

Machine code: 00110000

loadi 0

Machine code: 10111011

load R3

Machine code: 01011000

subi R0

Machine code: 00110100

loadi 4

Machine code: 00000101

blt4 R1

Machine code: 11101101

bne R1

Machine code: 11101100

bne R1

Machine code: 00100111

bezDec 7

Machine code: 00100101

bezDec 5

Machine code: 11000001

beq R1

Machine code: 11101101

bne R1

Machine code: 00100111

bezDec 7

Machine code: 11010101

sub R1, R1

Machine code: 11011010

sub R2, R2

Machine code: 11101100

bne R2

Machine code: 00100111

bezDec 7

Machine code: 01000101

addi R1, 1

Machine code: 10000100

store R1, R0

Machine code: 11101101

bne R1

Machine code: 00100011

bezDec 3

Machine code: 10001000

store R2, R0

Machine code: 11010101

sub R1, R1

Machine code: 11011010

sub R2, R2

Machine code: 01000101

addi R1, 1

Machine code: 11101100

bne R1

Machine code: 00100011

bezDec 3

Machine code: 10001000

store R2, R0

```
----------
Machine code: 11010101

sub R1, R1
----------
Machine code: 00000000
halt
----------
disassembler is being done
```

4. Python code for your ISA’s disassembler.

ISA.py

```
#ISA design for AVS

def disassembler(M,Nlines):
        print("ECE 366 Group 8 Disassembler")
        print("----------------")
   #print the instructions
        Rx = 0
        Ry= 0
        imm = 0
        for i in range(Nlines):

        fetch =M[i]
        print("Machine code: " + M[i])
        if(fetch[0:4] == "0011"): # load imm
        imm= int(fetch[4:8],2)
        print("loadi "+ str(imm))

        elif(fetch[0:4] == "1000"):#store Rx, Ry
        Rx= int(fetch[4:6],2)
        Ry= int(fetch[6:8],2)
        print("store R" + str(Rx)+ ", R" + str(Ry))

        elif(fetch[0:6] == "000100"):#srl Rx
        Rx= int(fetch[6:8],2)
        print("srl R" + str(Rx))

        elif(fetch[0:4] == "0100"):#addi Rx, imm
        Rx= int(fetch[4:6],2)
        imm= int(fetch[6:8],2)
        print("addi R" + str(Rx)+ ", " + str(imm))

        elif(fetch[0:6] == "010110"):#subi Rx
        Ry= int(fetch[6:8],2)
        print("subi R" + str(Ry))

        elif(fetch[0:6] == "111011"):#bne Rx
        print("bne R" + str(Rx))

        elif(fetch[0:6] == "110001"):#slt Rx
        Rx= int(fetch[6:8],2)
        print("slt R" + str(Rx))

        elif(fetch[0:5] == "00100"):#BezDec imm
        imm= int(fetch[5:8],2)
```

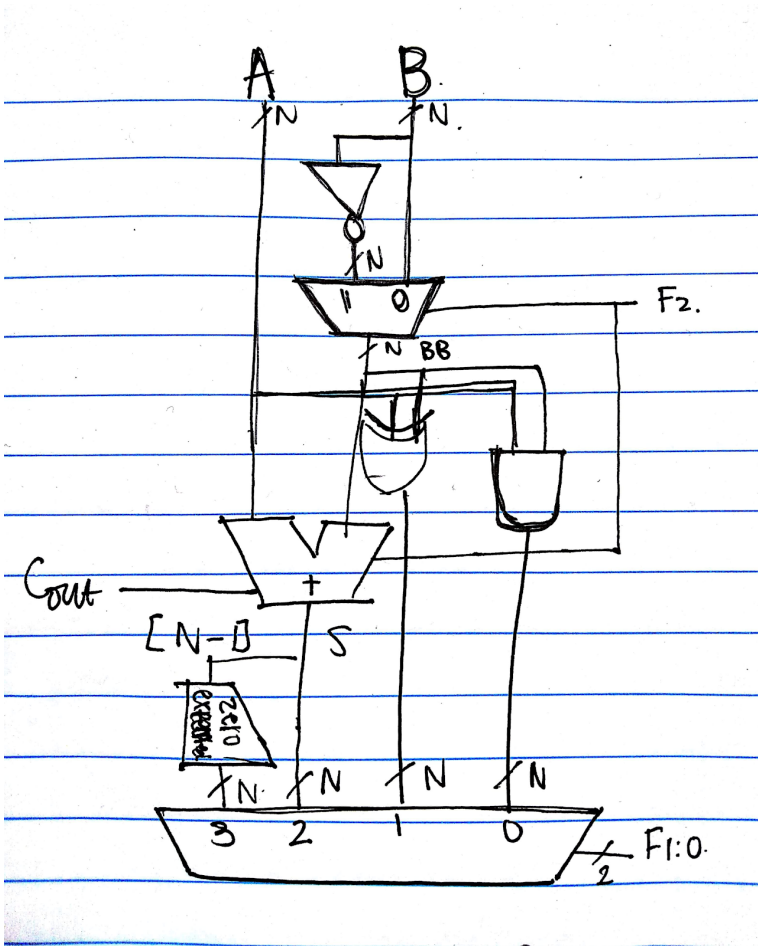
```
print("bezDec " + str(imm))

elif(fetch[0:4] == "0111"):#xori Rx, imm
Rx= int(fetch[4:6],2)
imm= int(fetch[6:8],2)
print("xori R" + str(Rx)+ ", R" + str(imm))

elif(fetch[0:4] == "0111"):#andi Rx, imm
Rx= int(fetch[4:6],2)
imm= int(fetch[6:8],2)
print("andi" + str(Rx)+ "," + str(imm))

elif(fetch[0:4] == "1010"):#jump 'branch'(imm)
imm= int(fetch[4:8],2)
print("jump " + str(imm))

elif(fetch[0:4] == "1001"):#add RX, Ry
Rx= int(fetch[4:6],2)
Ry= int(fetch[6:8],2)
print("add R" + str(Rx)+ ", R" + str(Ry))

elif(fetch[0:4] == "1101"):#sub Rx, Ry
Rx= int(fetch[4:6],2)
Ry= int(fetch[6:8],2)
print("sub R" + str(Rx)+ ", R" + str(Ry))

elif(fetch[0:8] == "10110110"):#subln R4

print("subln $R4")

elif(fetch[0:8] == "11110000"):#bne R4
print("bne $R4" )

elif(fetch[0:8] == "01010101"):# jump ' first branch'
print("jump 'first branch' ")

elif(fetch[0:8] == "00000000"):#halt
Rx= int(fetch[4:6],2)
Ry= int(fetch[6:8],2)
print("halt")

elif(fetch[0:6]=="101110"): #load Rx
Rx= int(fetch[6:8],2)
print("load R" + str(Rx))

elif(fetch[0:6]=="000001"): #blt4 Rx
Rx= int(fetch[6:8],2)
print("blt4 R" + str(Rx))

elif(fetch[0:6] == "110000"): #beq4 Rx
Rx = int(fetch[6:8],2)
print("beq R" + str(Rx))

elif(fetch[0:6] == "111110"): #andi5 imm
imm= int(fetch[6:8],2)
print("and5i R" + str(imm))
```

```
            elif(fetch[0:6] == "101111"): #srl5 imm
            imm=int(fetch[6:8])
            print("srl5 " + str(imm))

            elif(fetch[0:4] == "0110"): #xor Rx, Ry
            Rx =int(fetch[4:6],2)
            Ry =int(fetch[6:8],2)
            print("xor R" + str(Rx) + ",R" + str(Ry))

            else:
            print("Instruction set not supported")
            print("----------------")


#def assembler(I,Nlines):
#def simulator(I,Nsteps,debug_mode,Memory):
def main():
            instr_file = open("P1_Instruction.txt","r")
            data_file = open("project2_group8_p1_bin.txt" ,"r")
            data_file2 = open("project2_group8_p2_bin.txt", "r")
            #we need a file for the data set
            #Nsteps = 3  #How many cycles to run before output
            Nlines = 0   #How may instrs total in input.txt
            Instructions = [] #all instructions will be stored here
            Memory = []
            print( " ECE 366 Group 8")
   #print( " 1 = simulator")
            print( " 2 = disassembler")
            #print( " 3 = assembler")

            mode= int(input( "Please enter the mode of Program: "))
            print( "Mode selected: ",end=" ")
            modedis= int(input( "Please enter the which program 1 or 2:  "))
            print( "Mode selected: ", end=" ")

            #if(mode== 1):
            #print("Simulator")
            #elif(mode== 2):
            # print( "disassembler")
            #disassembler(Instructions,Nlines)
            #elif(mode== 3):
            # print( "assembler")
            #assemble(Instructions, Nlines)


            #for line in instr_file: # Reading in the instructions
            #            if (line == "/n" or line[0] == '#'):  #empty lines, comments ignored
            #            line = line.replace("\n," ")
            #            Instructions.append(line)        #Copy all instruction into a list
            #Nline+=1

            #for line in data_file: # Read in data P1_Machine.txt
            #   if(line== "\n" or line[0] =='#'):
   #        continue
            #  Memory.append(line)
            #Nlines+=1
            if(mode == 1): #Check whether to use disassembler of assembler or simulator
```

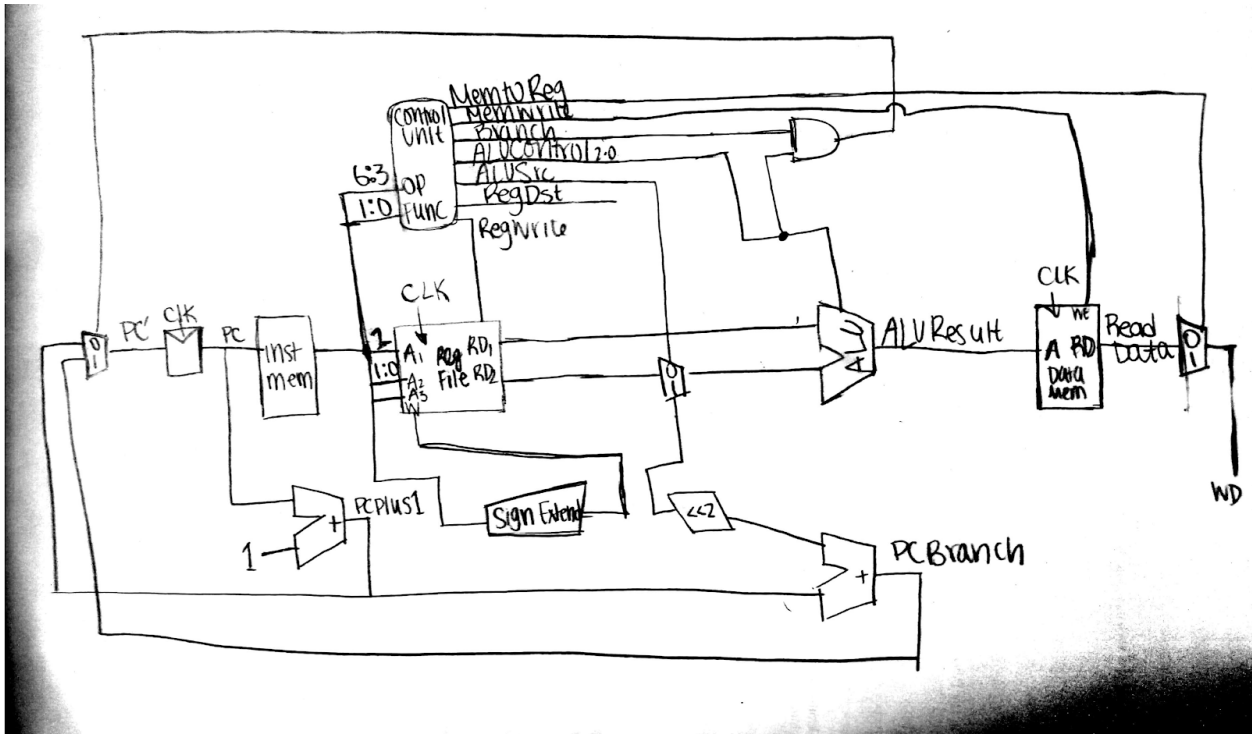
```
            #simulator(Instructions,Nsteps,debug_mode,Memory)
            print("assembler")
            elif(mode== 2):
            if (modedis == 1):
            for line in data_file: # Read in data P1_Machine.txt
            if(line== "\n" or line[0] =='#'):
            continue
            Memory.append(line)
            Nlines+=1
            elif(modedis == 2):
            for line in data_file2: # Read in data P1_Machine.txt
            if(line== "\n" or line[0] =='#'):
            continue
            Memory.append(line)
            Nlines+=1
            else:
            print("That is not one of the options")

            disassembler(Memory,Nlines)
            print("disassembler is being done")
            elif(mode== 3):
            #assembler(Instructions,Nlines)
            print("assembler is being done")
            else:
            print("Error. Unrecognized mode. Exiting")
            exit()

            #instr_file.close()
            data_file2.close()
            data_file.close()

if __name__ == "__main__":
            main()
```

Part D) Hardware implementation:

1. ALU schematic. A hierarchical sketch of your Arithmetic Logic Unit which implements whatever computation that your ISA instructions use (See textbook ch 5.2.4).



2. CPU Datapath design. A schematic including your register file, ALU, PC logic, and memory components (see textbook ch 7.3.1).



3. Control logic design. Decoder truth-table indicating how each control signal is generated from an instruction (see textbook ch 7.3.2).

INST	OP	RegWrt	ALUctr	Branch	Memwrt	MemtoReg
Load	1 000	1	1	0	1	1
Store	1 011	0	1	0	0	0
shl	0 001	1	0	0	0	0
sll	0 100	1	0	0	0	0
BezDec	0 010	0	0	1	0	0
xori	0 110	1	0	0	0	0
andi	0 111	1	0	0	0	0
jump	1 010	0	0	1	0	1
sub	1 101	1	0	0	0	0
loadi	0 011	1	1	0	1	1
blt4	0 000	0	0	0	0	0
beq4	1 100	0	0	0	0	0

andi5	1 111	1	0	0	0	0
srl5	1 011	1	0	0	0	0