Part A: ISA and Q&A

- **Introduction:** name of the architecture, overall philosophy, specific goals strived for and achieved.
  - **Name of Architecture:** Hardcoded ISA
  - **Overall Philosophy:** Maximize the register use to make prpg easier and more accessible (hardcode to a register only if there is other information needed from the info to customize)
  - **Specific Goal:** None

- **Instruction list/ table**

PC	Opcode	Instruction	Formats	Example
+4	0000	sw	0000 Rx[0-15] (take value from R0 always)	saves R0 to M[8+Rx] Sw R1 R0 = 00000001 Final binary value = 00000001

+4	0001	lw	0001 Rx[0-15]
			(always writes to R2)
				R2 = M[8 + Rx]
				
				Lw R1
				R1 = 00000001
				Final binary value = 00010001

+4	0010	addi	0010 Rxi[0-7](will only addi by 1 or 0)	Rx = Rx + imm(1bit)
				
				Addi R7, 1
				R7 = 00000111
				Final binary value = 00001000

+4	10XX	bne	10Rxiiii [0-15](will be hardcoded to always be a specific register R7)	R7 =? Rx If yes = loop Else break Will always jump to previous instructions Bne R1, seeding_loop R1 = 00000001 To get to seeding loop = 6 jumps necessary = 0110 Final binary value = 10010110
+4	1100	Add with carry	1100RxRy	Rx = Rx + Ry Add Rx, Ry Add R2, R0

				R2 = 00000010
				R0 = 00000000
				Rx = ((R6 << 8) + Rx) + Ry
				R6 = 8 MSB
				Final binary value = 11001000

+4	0101	init	0101imm R0 = imm Imm: ● 0000 = 251 ● 0001 = 118 ● 0010 = 79 ● 0011 = 5	int(imm) Init (251) Final binary value R0= 01010000

+4	0110	AOS (average of sums)	0110Rx
			(R0 = where we save our answer
			Rx is where read the value from)
				If R6 !=? 0
				If true -> temp =(Rx<<8) + Rx
				If false -> temp = Rx
				Output : temp = temp >>4
				
				Rx = 1111 1111
				R6 = 0000 0001<< 8
				R6 = 100000000
				Rx = R6 + Rx, Rx = 111111111
				R0 = Rx >> 4

+4	0100	setR	0100Rxi	setR Rx, imm(1bit) setR R6, 0 R6 = 0 Final binary value = 01001100

+4	0011	MAD mult&drop middle	0011RxRy(multiplies two registers and concatenates the value so that Rx holds only the 4MSB and $LSB of the result)	R1 = 10000001
				R0 = 01000000
				Rx = [0100]0000100[0000]
				Rx = 01000000

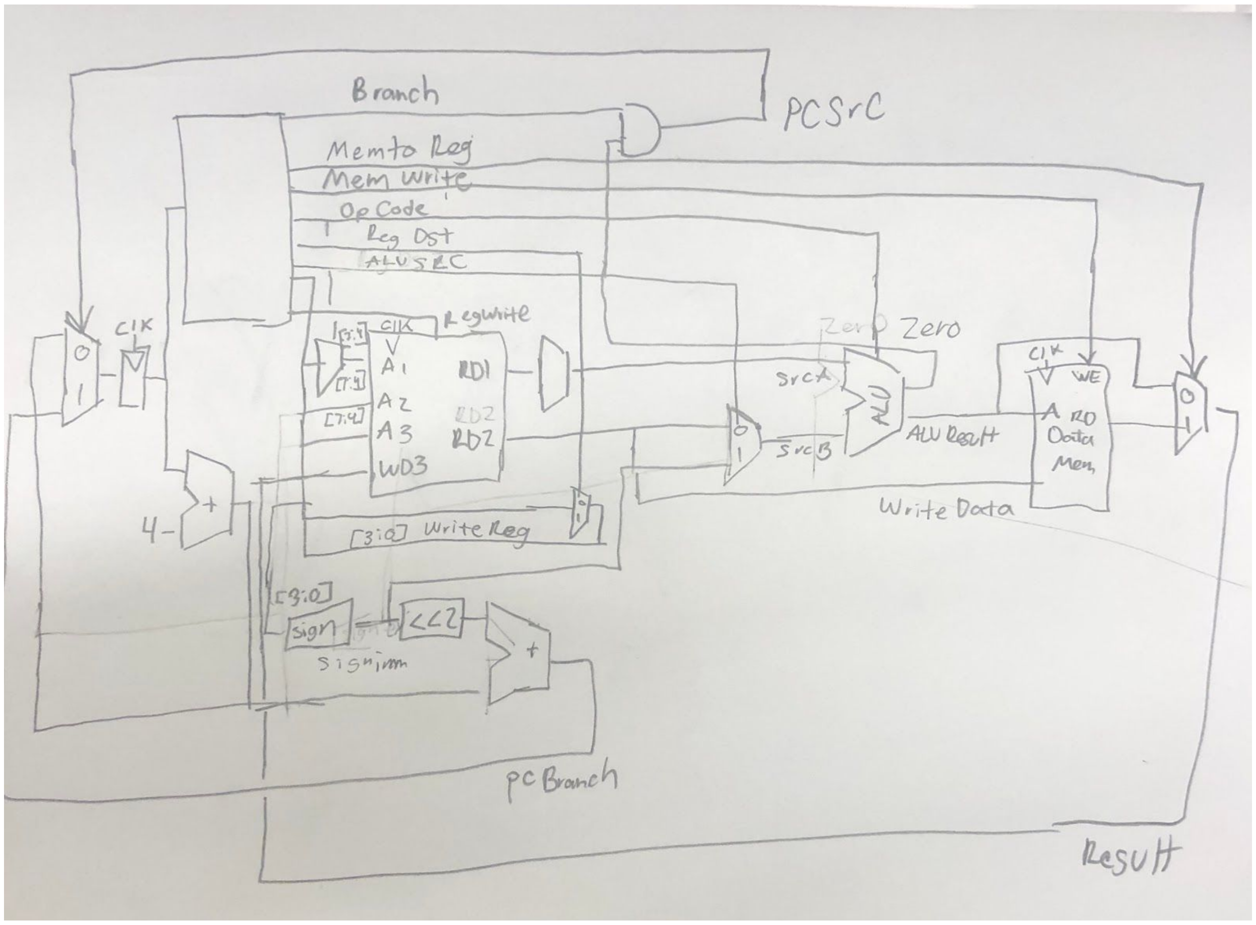
+4	0111	sll	0111Rx(shift left R3)	Sll Rx Sll R5 R5 = R3 << 1 Final binary value = 01110101

+4	1110	Count	1110Rx	R0 = # of 1’s in (Rx)
				
				Count R2
				R0 = # of 1’s in R2
				Final binary value = 11100010

+4		exit	11111111	Signifies end of the program

Part B: Hardware Implementations

1. **CPU Datapath Design**



*** Mux feeding into A1 works with opcode and either sets it to a register specific register or allows it to put in a register value depending on the function the op code represents (further explanation at table below)**

2. **Control Logic Design**

   Key:
   - Register Write = RW
   - Register Destination = RD
   - ALUSource = ALUS
   - Branch = B
   - Memory Write = MW

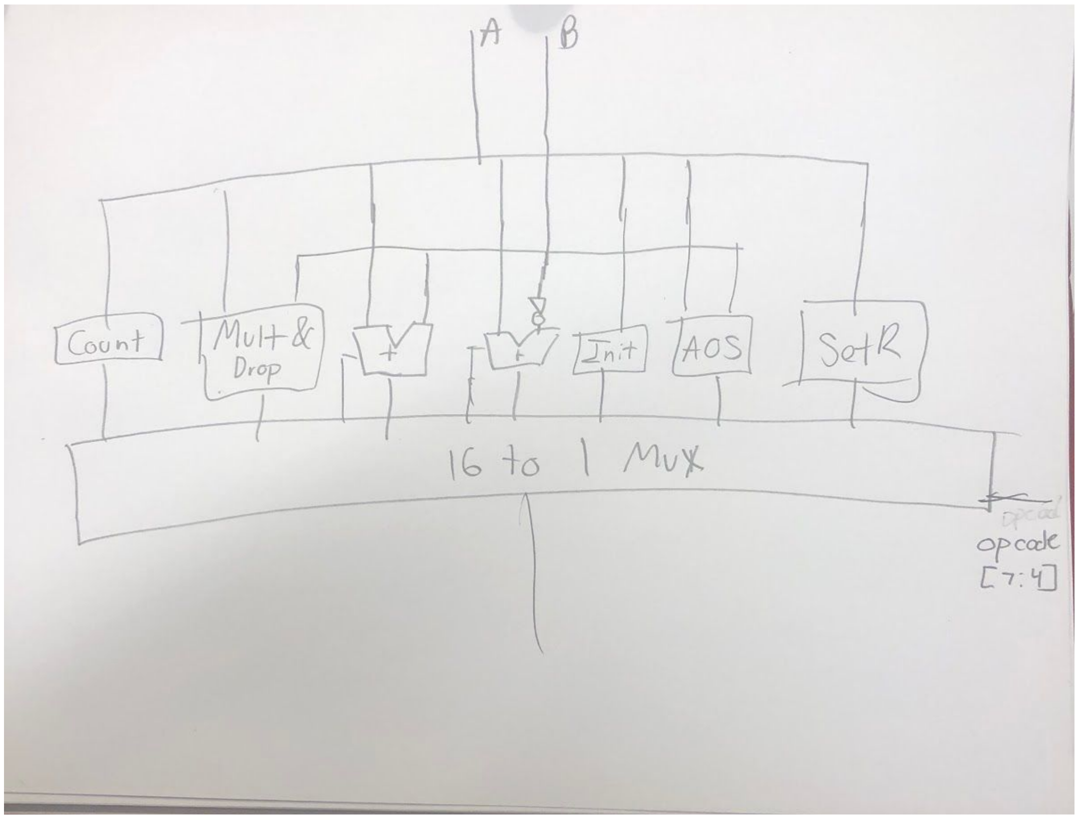
- Memory to Register = MR
- ALU Control = ALUO
- A1, A2, A3 = specific registers

Instr	Opcode	RW	RD	ALUS	B	MW	MR	ALUO	A1	A2	A3
sw	0000	0	X	1	0	1	X	101	reg[0000]	X	0
lw	0001	1	0	1	0	0	1	101	0	0	reg[0010]
addi	0010	1	0	1	0	0	0	101	X	X	X
bne	10XX	X	X	0	1	0	X	0XX	X	reg[0111]	X
add	1100	1	1	0	0	0	0	101	X	X	0110
init	0101	1	0	1	0	0	0	101	0	0	reg[0000]
AOS	0110	1	1	1	0	0	0	001	X	reg0110]	reg[0000]
setR	0100	1	1	1	0	0	0	000	X	0	X
MAD	0011	1	1	0	0	0	0	110	X	X	X
sll	1100	0	1	X	0	0	0	XXX	0011	0	X
count	1110	1	1	X	0	0	0	111	X	0	X

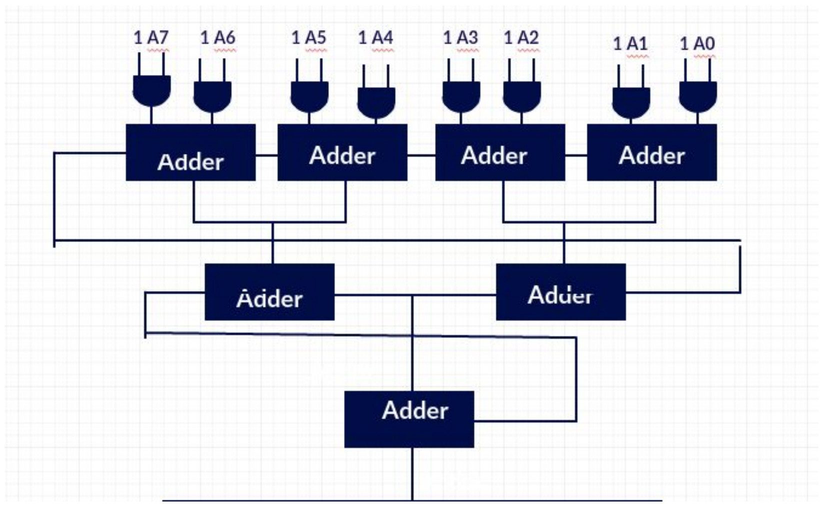
***Reg[bin] -> this area of the the registers is set specifically to the register at the binary value otherwise it can any register which is represented in our code by Rx**

3. ALU Schematic

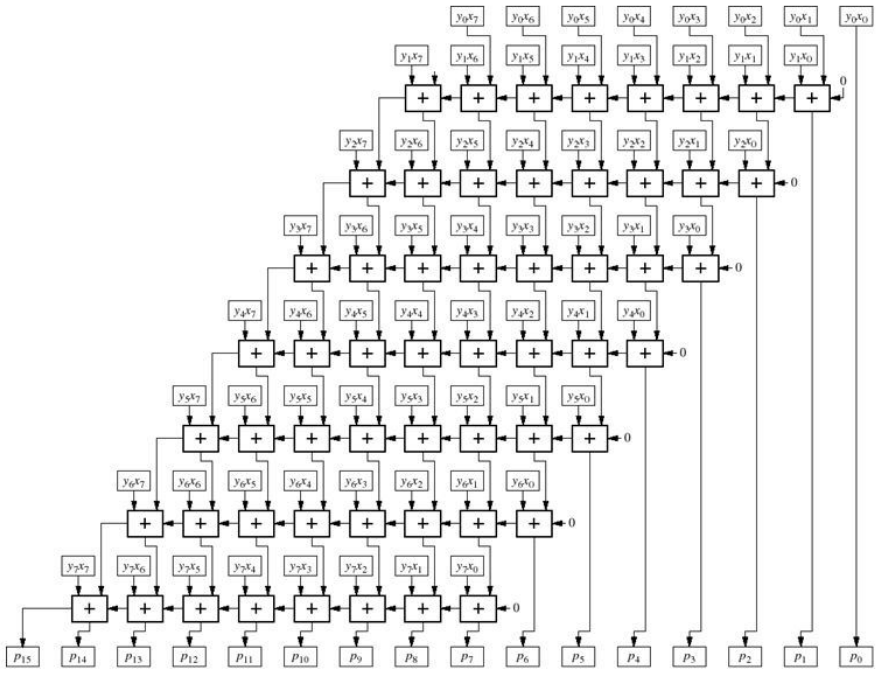
Overall ALU Design


A
B
Count
Mult & Drop
+
+
Init
AOS
SetR
16 to 1 Mux
opcode [7:4]

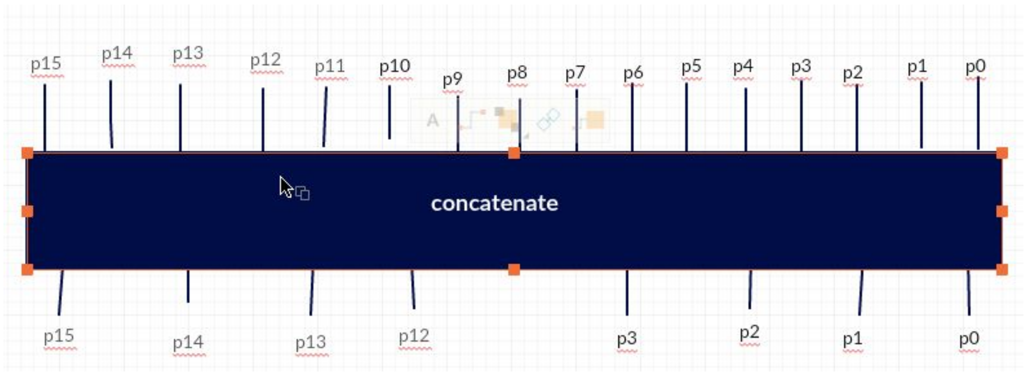
Count


1 A7
1 A6
1 A5
1 A4
1 A3
1 A2
1 A1
1 A0
Adder
Adder
Adder
Adder
Adder
Adder
Adder

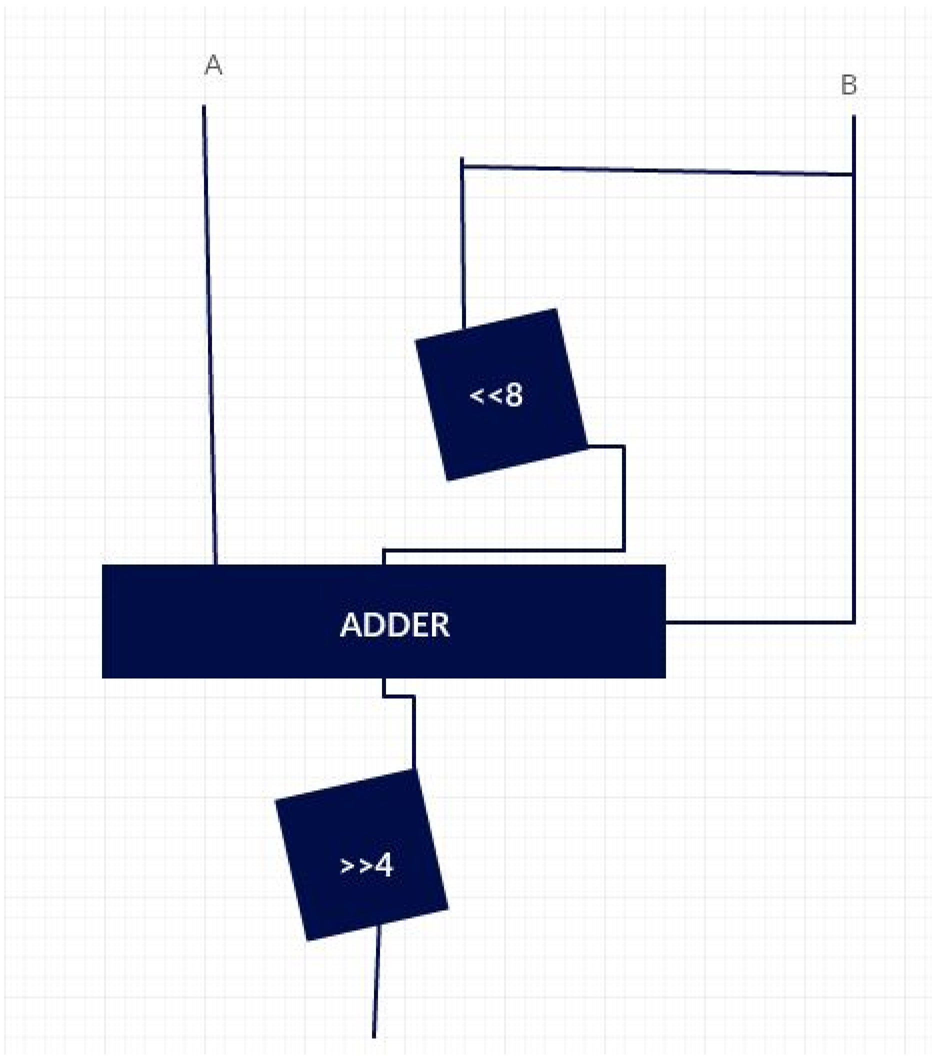
Mult



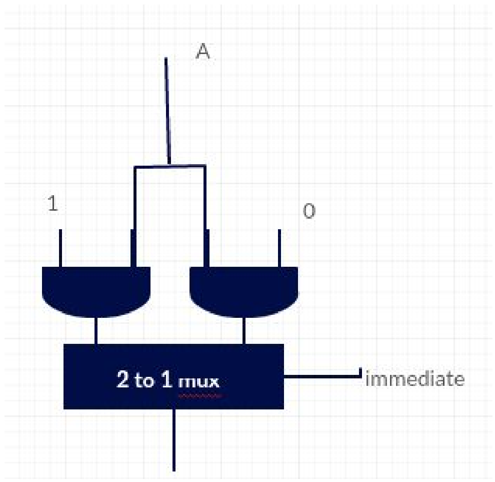
y_0x_7
y_0x_6
y_0x_5
y_0x_4
y_0x_3
y_0x_2
y_0x_1
y_0x_0
y_1x_7
y_1x_6
y_1x_5
y_1x_4
y_1x_3
y_1x_2
y_1x_1
y_1x_0
y_2x_7
y_2x_6
y_2x_5
y_2x_4
y_2x_3
y_2x_2
y_2x_1
y_2x_0
y_3x_7
y_3x_6
y_3x_5
y_3x_4
y_3x_3
y_3x_2
y_3x_1
y_3x_0
y_4x_7
y_4x_6
y_4x_5
y_4x_4
y_4x_3
y_4x_2
y_4x_1
y_4x_0
y_5x_7
y_5x_6
y_5x_5
y_5x_4
y_5x_3
y_5x_2
y_5x_1
y_5x_0
y_6x_7
y_6x_6
y_6x_5
y_6x_4
y_6x_3
y_6x_2
y_6x_1
y_6x_0
y_7x_7
y_7x_6
y_7x_5
y_7x_4
y_7x_3
y_7x_2
y_7x_1
y_7x_0
+
0
p_{15}
p_{14}
p_{13}
p_{12}
p_{11}
p_{10}
p_9
p_8
p_7
p_6
p_5
p_4
p_3
p_2
p_1
p_0



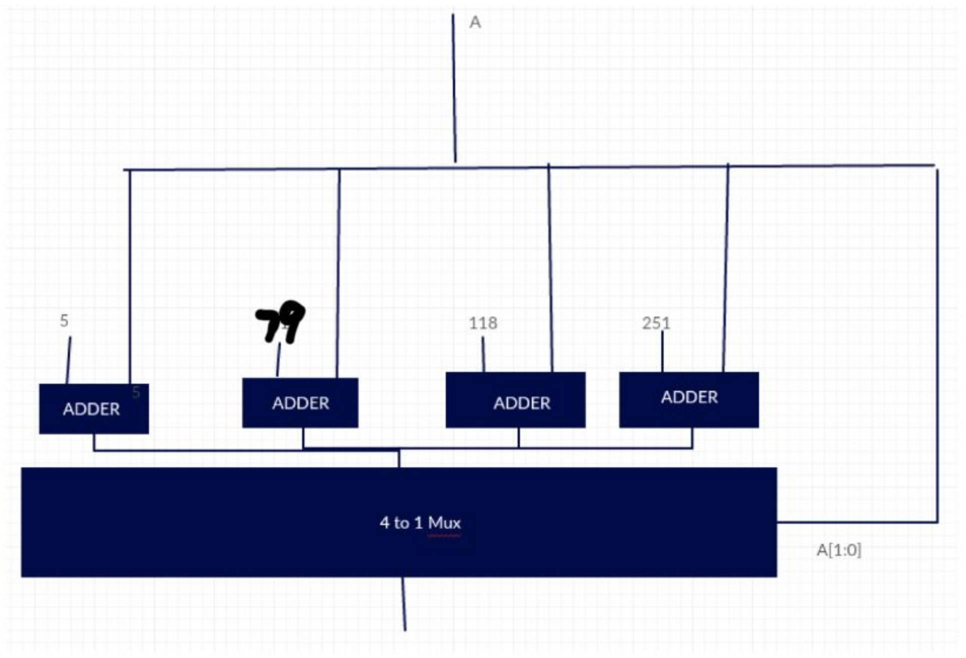
p15
p14
p13
p12
p11
p10
p9
p8
p7
p6
p5
p4
p3
p2
p1
p0
concatenate
p15
p14
p13
p12
p3
p2
p1
p0



SetR



Init



Result would be an 8 bit value

```
def disassemble(instructions, debugMode):
        branch = 0                      # branch counter
        MI = 0                          #memory instruction counter
        line = 0                        #keeps track of what line of of instruction from the txt file is
                                        being run

        finished = False                #is the program finished?
        reg = [0]*15                    #declare register array all initialized to 0
        mem = [0]*49                    #declares mem array all initialized to 49
        fetch = instructions[line]
        while(not finished):
        fetch = instructions[line]
        if(fetch[0:4] == "1111"):
                finished = True
                display(mem, reg, line, MI, branch)
                elif(fetch[0:4] == "0000"):     #SW-> Mem[8+value in reg[x]] = reg[0]
                Rx = reg[int(fetch[4:8],2)]
                mem[8+Rx] = reg[0]
                line += 1
                MI += 1
                if(debugMode):
                        print("IC = " + str(line).zfill(2) +":        sw $" + str(int(fetch[4:8],2)) + " =>  "
+ "mem[8+" + str(reg[int(fetch[4:8],2)]) + "]" +  " = $0")
        elif(fetch[0:4] == "0001"):        #LW-> reg[2] = mem[8+value in R[x]]
        reg[2] = mem[8+reg[int(fetch[4:8],2)]]
        line += 1
        MI +=1
        if(debugMode):
                print("IC = " + str(line).zfill(2) +":        lw $" + str(int(fetch[4:8],2)) + " =>  "+ "$2 = "
+  "mem[8+" + str(reg[int(fetch[4:8],2)]) + "]")
        elif(fetch[0:4] == "0010"):        #addi -> R[x] = R[x] + imm(either 1 or 0)
        Rx  = int(fetch[4:7],2)
        if(fetch[7:8] == "1"):
                imm = 1
        else:
                imm = 0
        reg[Rx] = reg[Rx] + imm
        line += 1
        if(debugMode):
                print("IC = " + str(line).zfill(2) +":        addi $"+ str(Rx) +", "+ str(imm) + " =>
"+"$"+ str(Rx) + " = $" + str(Rx) + " + " + str(imm))
        elif(fetch[0:2] == "10"):          #bne -> if (reg[x] != reg[7]) =True always loop to previous
                                                                   instructions(imm)
                                           If False go to the next instruction
        valx  = reg[int(fetch[2:4],2)]
        imm = int(fetch[4:8],2)
        if(reg[7] != valx):
                line += 1
                line  = line - imm
                if(debugMode):
                print("IC = " + str(line).zfill(2) +":        bne $"+ str(int(fetch[2:4],2)) +", LOOP(-"+
str(imm) + ")" + " =>  ($7 != $"+ str(int(fetch[2:4],2)) + ") = " + "True")
        else:
                line += 1
                if(debugMode):
                print("IC = " + str(line).zfill(2) +":        bne $"+ str(int(fetch[2:4],2)) +", LOOP(-"+
str(imm) + ")" + " =>  ($7 != $"+ str(int(fetch[2:4],2)) + ") = " + "False")
        branch += 1
        elif(fetch[0:4] == "1100"):        #add  -> R[x] = R[x] +R[y]
        Rx  = reg[int(fetch[4:6],2)]
        Rx = (reg[6]<<8) + Rx
        Ry = reg[int(fetch[6:8],2)]
        tempAns = Rx + Ry
```

```
          reg[6] = int(tempAns>>8)
          reg[int(fetch[4:6],2)] = tempAns & 255
          line += 1
          if(debugMode):
                    print("IC = " + str(line).zfill(2) +":        add $"+ str(int(fetch[4:6],2))+", " +
str(int(fetch[6:8],2)) + "   =>  $" + str(int(fetch[4:6],2)) + " = $" + str(int(fetch[4:6],2)) +" + $"
+str(int(fetch[6:8],2)) )
          elif(fetch[0:4] == "0101"):        #init = r[0] = a specific value
          line += 1
          if(fetch[4:8] == "0000"):
                    reg[0]  = 251
                    if(debugMode):
                    print("IC = " + str(line).zfill(2) +":        init  0"+ "  =>  " + "$0 = " + str(reg[0]))
          elif (fetch[4:8] == "0001"):
                    reg[0] = 118
                    if(debugMode):
                    print("IC = " + str(line).zfill(2) +":        init  1" + "  =>  " + "$0 = " + str(reg[0]))
          elif(fetch[4:8] == "0010"):
                    reg[0]  = 79
                    if(debugMode):
                    print("IC = " + str(line).zfill(2) +":        init  2" + "  =>  " + "$0 = " + str(reg[0]))
          elif (fetch[4:8] == "0011"):
                    reg[0] = 5
                    if(debugMode):
                    print("IC = " + str(line).zfill(2) +":        init  3" + "  =>  " + "$0 = " + str(reg[0]))
          elif(fetch[0:4] == "0110"):        #AOS - Average of sums
                                             Reads the sum and divides
          line += 1
          if(reg[6] != 0):
                    reg[0] = (reg[6]<<8) + reg[int(fetch[4:8],2)]
                    reg[0] = reg[0] >> 4
                    if(debugMode):
                    print("IC = " + str(line).zfill(2) +":        AOS  " + str(int(fetch[4:8],2)) + "   =>  " + "$6
!= 0, " + "$0 = ($6<<8) + $"+str(int(fetch[4:8],2))+", $0 = "+ str(reg[0]))
          else:
                    reg[0] = reg[int(fetch[4:8],2)]
                    reg[0] = reg[0] >> 4
                    if(debugMode):
                    print("IC = " + str(line).zfill(2) +":        AOS  " + str(int(fetch[4:8],2)) + "   =>  " + "$6
== 0, " + "$0 += $"+str(int(fetch[4:8],2))+", $0 = "+ str(reg[0]))
          elif(fetch[0:4] == "0100"):        #setR-> r[x] = imm(1 or 0)
          imm = int(fetch[7:8],2)
          reg[int(fetch[4:7],2)] = imm
```

```
        line += 1
        if(debugMode):
                print("IC = " + str(line).zfill(2) +":        setR  $" + str(int(fetch[4:7],2)) +", " +
str(imm) + "   =>   $" + str(int(fetch[4:7],2))+ " = " + str(imm) )
        elif(fetch[0:4] == "0011"):   #MAD = multiplies two registers and concatenates the value
                                          so that Rx holds only the 4 MSB and 4$LSB of the result
        valx  = reg[int(fetch[4:6],2)]
        valy = reg[int(fetch[6:8],2)]
        tempMSB= (valx * valy) >> 12
        tempLSB =(valx * valy) & 15
        reg[int(fetch[4:6],2)] =(tempMSB<<4) + tempLSB
        line += 1
        if(debugMode):
                print("IC = " + str(line).zfill(2) +":        MAD  $" + str(int(fetch[4:6],2)) +", $" +
str(int(fetch[6:8],2)) + "   =>   $" + str(int(fetch[4:6],2))+ " = " + str(reg[int(fetch[4:6],2)]))
        elif(fetch[0:4] == "0111"):       #SLL -> shift logical left by 1
        reg[int(fetch[4:8],2)] = reg[3] <<1
        line += 1
        if(debugMode):
                print("IC = " + str(line).zfill(2) +":        sll  $" + str(int(fetch[4:8],2)) + "   =>  " + "$
"+ str(int(fetch[4:8],2)) + "+= $3 << 1")
        elif(fetch[0:4] == "1110"):       #cont - > counts the number of 1's int R[x]
        Rx = reg[int(fetch[4:8],2)]
        temp=0
        temp  += Rx & 1
        temp  += (Rx & 2)>>1
        temp  += (Rx & 4)>>2
        temp  += (Rx & 8)>>3
        temp  += (Rx & 16)>>4
        temp  += (Rx & 32)>>5
        temp  += (Rx & 64)>>6
        temp  += (Rx & 128)>>7
        reg[int(fetch[4:8],2)] = temp
        line += 1
        if(debugMode):
                print("IC = " + str(line).zfill(2) +":        count  $" + str(int(fetch[4:8],2)) + "   =>   $" +
str(int(fetch[4:8],2)) +" = # of 1's in $" + str(int(fetch[4:8],2))+ ",  $" + str(int(fetch[4:8],2)) + " = " +
str(temp))
        else:
        print("This opcode is not valid: ", str(fetch[0:4]) )
        finished = True
        print("Done!!!!")
```

```
def main():
        inFile = open("Project3Hex.txt", "r")   #opens the file
        instructions = []                        #declares an array

        for line in inFile:
        if(line == "\n" or line[0] == '#'):
        continue
        line = line.replace('\n', '')
        line = format(int(line, 16), "08b")       #formats the number as 8 bits and uses 0 as filler
        instructions.append(line)
        inFile.close()
        debugMode = int(input("1: Debug Mode \n0: Normal Mode : "))
        disassemble(instructions, debugMode)
main()
```

Appendix

1. p3_g_15_report.pdf ✓
2. p3_g_15_sim.py ✓
3. p3_g_15_prpg_251.txt ✓
4. p3_g_x_prpg_sim_out_251.txt ✓
5. p3_g_15_prpg_118.txt ✓
6. p3_g_x_prpg_sim_out_118.txt ✓
7. p3_g_15_prpg_79.txt ✓
8. p3_g_x_prpg_sim_out_79.txt ✓

9. p3_g_15_prpg_5.txt ✓

10. p3_g_x_prpg_sim_out_5.txt ✓