ECE 366
Project 3

A)
1. The name of the architecture is ctz. We have made this specific ISA design to be able to calculate R = 6^P % Q and to be able to find the 16-bit pattern/patterns out of 100 that has the most bits similar to pattern T.
2.

PC	Functionality	Instruction	Coding	Example	
PC += 1	RX = M[RY]	load RX, RY	000 RX RY	load R0, R1	10000001
PC += 1	M[RY] = RX	store RX, RY	001 RX RY	store R2, R3	00011011
PC += 1	RX = RX + RY	add RX, RY	010 RX RY	add R0, R1	00100001
PC += RX	PC = PC + RX	jump RX	110 10 RX	jump R0	11101000
If RX = RY then PC += 2, else PC += 1		beq RX, RY	011 RX RY	beq R0, R2	10110010
PC += 1	If RX < RY then R0 = 1, else R0 = 0	slt RX, RY	100 RX RY	slt R1, R2	11000110
PC += 1	RX = imm, imm = [-2:1]	init RX, imm	101 RX ii	init R0, -2	01010011
PC += 1	RX = RX >> 1	slr RX	110 00 RX	slr R0	01100000
PC += 1	RX = RX & 1	and RX	110 01 RX	and R1	01100101
PC += 1	RX = ~ RX	not RX	110 11 RX	not R2	11101110
PC += 1	RX = RX XOR RY	xor RX, RY	111 RX RY	xor R0, R1	01110001

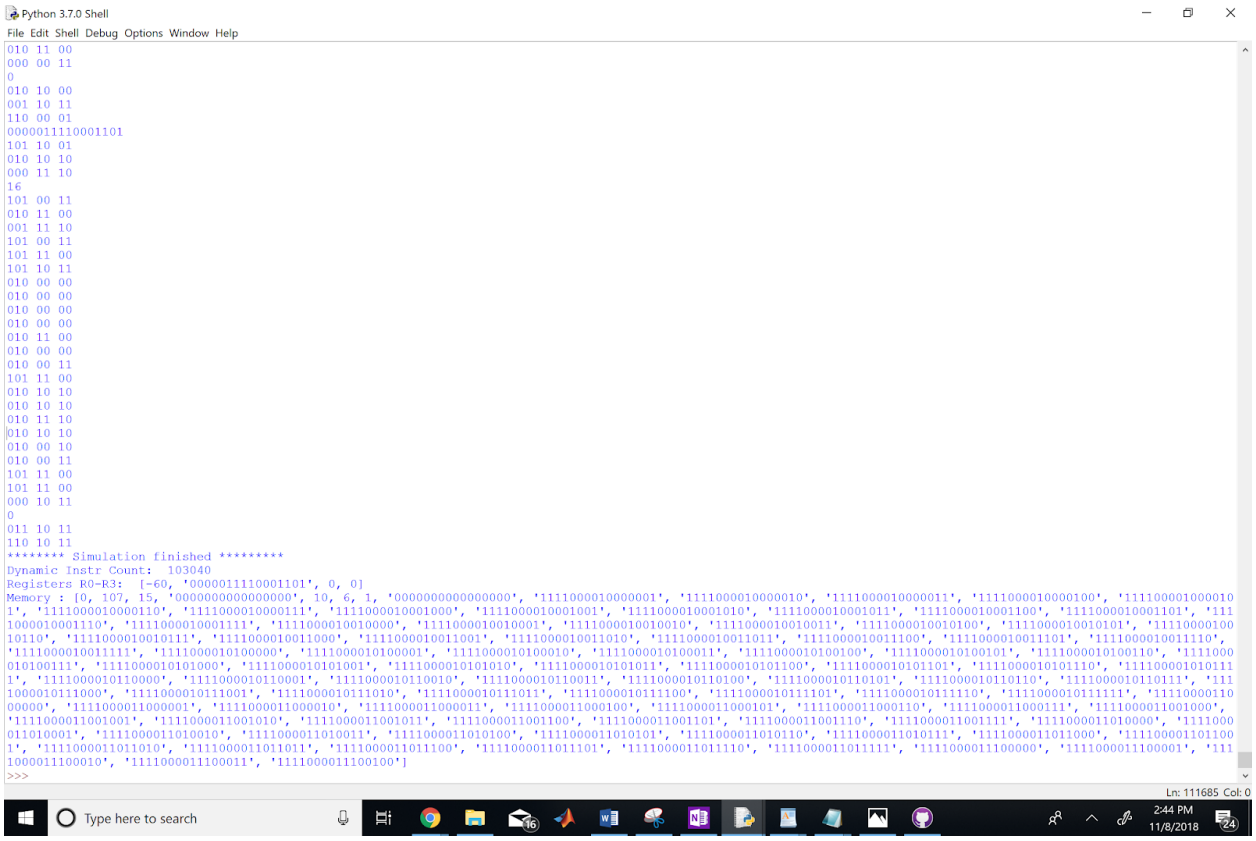
3. The registers that are supported arefrom R0-R3.
4. The only special design that we have is in beq. When the 2 things being compared is indeed equal to each other, pc would be increased by 2 and increased by 1 otherwise. We changed jump to jump by a certain value in the register provided instead of using immediate values which leads to bigger jumps.
5. Load and store works the same. The load instruction takes in M[RY] and loads it into RX. The store instruction takes the value of the register RX, and stores it into M[RY].
Load : 000 RX RY
Store : 001 RX RY
Both RX and RY are limited to 2 bits which gives it a range of registers R0-R3.

B)
1. The most significant advantages of our ISA design is the jump and beq instructions. With these 2 instructions being the way we have it, we are able to move and jump to different parts of the code very quickly. By having jump take a register, we were able to remove the limitation of having to jump only by 3 bits for example. Since jump takes in a register instead of an immediate value, we need to keep track of more values in memory since we have to load and store the values we are jumping by and the value that we are moving.

2. Working towards a goal of a low DIC unfortunately a luxury that we had on this project it was a struggle even to get anything work. The hardware on the other hand was as simple as it possibly could be without being made specifically for this application. The ISA was designed with the belief that the ISA could be used for many applications, which in this case lead to a very high DIC. In order to optimize the DIC we would have needed to adjust the ISA such that its commands were less universal and more specific to this particular project. Doing that would change the design of the hardware completely and would likely make it more complicated in a hardware aspect, but much faster. We miscalculated the "cost" versus the value of time, time is money and our ISA takes a long time for basic arithmetic.
3. We have learned that when given only a certain number of bits to work with, it can be very confusing and tedious when dealing with certain tasks. It was horrible to only have 4 registers to work with since we have to keep loading and storing the values we used and make sure that none of them is overwritten. If one of them is overwritten, the whole thing will be wrong. It also taught us to work in small parts of code at a time instead of working on all of it. By working in parts, we are able to find where the problem occurs and fix it right away. The advice that we would give to someone taking this project in the future is that they should always keep track of the registers and values they hold. With the limit of 4 registers, the values are easily lost which leads to the wrong outputs. Another piece of advice would be to set clear expectations for group members at the beginning of the process and create our own "due dates" to keep group members in check and to think about making a happy medium between hardware complexity and DIC rather than one extreme or the other.
The project holds a lot of value for a job interview. It taught us micro managing and creativity. Creativity in a way that we had to figure out how to make our ISA design work for multiple programs with a limit of bits. Micro managing because of how we need to keep track of certain values to ensure the right output. This project taught us a very hard lesson in time management, group management and to pay attention to the big picture when developing something such as paying attention to DIC and hardware not just one or the other but both. So, that a project or product can be produced cheaply and user friendly.

C)

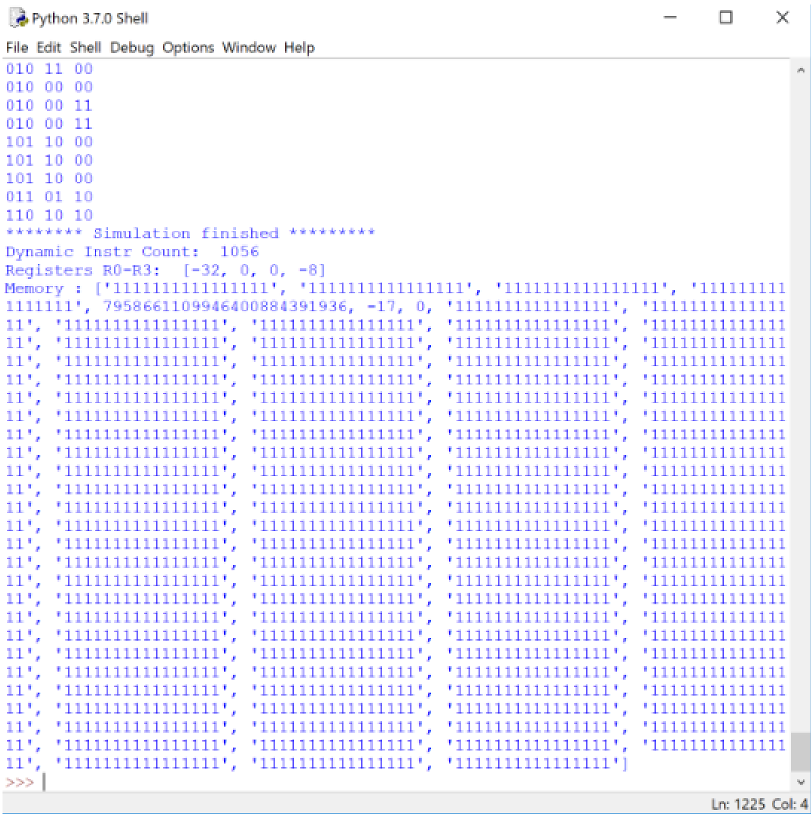
Pattern A



Pattern A unfortunately was not successful due to poor coding of our ISA. The simulator seems to work for a majority of the entire program, but debugging the ISA programs was extremely difficult. When using the simulator on simpler code the simulator performed its expected actions. Pattern B was even less successful. The coding of the ISA again was the problem here. Time was spent perfecting the simulator rather than the coding of the programs and this caused major issues in the end.

Pattern C and D:

These patterns never were created due to the lack of success with the ISA programs. Part two of the project is nearly functional, however part 1 is severely lacking and has no functionality. #P01 is the second test program created aside from the earliest samples and it works great with the simulator it is designed to take 6 to the power of 32.



A test program for part 2 unfortunately was not able to be completed due to lack of time. However it had some major functionality in part 2 it had trouble when going through all 1600 bits and keeping things straight again this was due to poor debugging.

D)

1. Program 1 (By Patrick)

Init r1, 1

Not r2

Not r2

Not r2

Not r2

Not r2

Not r2

Loop:

Add r1, r1

Init r2, 0

```
Add r2, r1
Add r2, r1
Add r2, r1
Init r3, 1        #used to move from memory to memory
Load r0, r3       #r0 = M[1] r0=Q
Add r3, r3        #r3 = 2
Store r2, r3      #M[2] = r2
Init r1, 0
Add r1, r0        #r1 is also Q now

Mod:
Init r3, 1
Add r3, r3        #r3 = 2
Add r3, r3        #r3 = 4
Add r3, r3        #r3 = 8
Init r0, 1
Add r0, r0
Add r3,r0         #r3= 10
Init r0,1
Add r3, r0        #r3=11
Not r3
Not r3
Slt r2, r1        #r2<r1 then r0 = 1
Init r1, 0
Beq r1, r0
Jump r3
Init r1, 1
```

Program 2 (By Cameron)

```
#load 000 rR rR
#store 001 rR rR
#add 010 rR rR
#beq 011 rR rR #we need to make this more clear.
#slt 100 rR rR
#init 101 rR ii
#slr 110 00 rR
#and 110 01 rR
```

```
#jump 110 10 rR
#xor 111 rR     RR
# This is part two of project 2 searching for best matching     # score and count.
# M[8] = beginning of Pattern_Array
# M[7] = temp score # 2
# M[6] = temp score holder
# M[5] = best_matching_count
# M[4] = best_mathing_Score
# M[3] = target
# M[2] = 16 a counter for each compare
# M[1] = the address of beginning of pattern
# M[0] = be used as a counter for number of patterns
        init r0, 1
        init r1, 1
        init r2, 1
        init r3, 1
        add r3, r2 #r3 = 2
        add r2, r2 #r2 = 2
        add r3, r3 # r3 = 4
        add r0, r3 #r0 = 5
        init r2, -1
        store r2, r0 #stores -1 into match count so first iteration when M[6]==M[4] (they both
begin at zero)
        init r2, -1
        init r0, 0
        add r0, r3 #r0 = 4
        add r3, r3      #r3 = 8
        add r3, r2 # stores 7
        store r3, r1    #M[1] = 7; the address of beginning of pattern
        init r2, 1
        add r3, r2
        add r3, r3      # r3 = 16

        add r2, r2
        store r3, r2    #M[2] = 16 a counter for each compare
        add r3, r3      # r3 = 32
        init r1, 0
        add r1, r3 # r1= 32
        add r3, r3 # r3 = 64
```

```
        init r2, 0
        add r2, r3 #r2 = 64
        add r3, r1 # r3 = 96
        add r3, r0 # r3 = 100
        init r0, 0
        store r3, r0 # saves 100 into M[0] to be used as a counter
#Big loop:
        init r0, 1
        init r1, 1
        add r1, r1
        add r0, r0 #2
        add r0, r0 #4
        add r0, r0 #8
        add r0, r0 #16
        store r0, r1 #sets bit counter back to 16
        init r0, 1
        init r1, 1
        add r1, r1 #2
        add r1, r1 #4
        add r0, r0 #2
        add r0, r1 #6
        load r2, r1 #r2 = M[4] best score
        load r3, r0 #r3 = M[6] temp best score
#if they match add to match count
        init r1, 1
        init r0, -1
        add r1, r1      #2
        add r1, r1 #4
        add r1, r1 #8
        add r1, r1 #16
        add r0, r0 #-2
        add r1, r0 #14
        beq r2, r3
        jump r1 #to end of incrementing match count (14)
        init r0, 1
        init r1, 1
        add r0, r0 #2
        add r0, r0 #4
        add r0, r1 #5
```

```
        load r2, r0 # loads match count
        add r2, r1 # increments match count
        store r2, r0 # stores match count
        add r1, r1 #2
        add r1, r1 #4
        add r0, r1 #r0=9
        add r0, r0 #18
        jump r0 #to where M[6] is switched to 0 (18)
# if they are not equal are is the temp score larger
# if so write new value into best score
        slt r2, r3 # if r2<r3 r0=1
        init r1, 1
        init r2, 1
        add r2, r2 #2
        add r2, r2 #4
        add r2, r2 #8
        add r2, r1 #9
        beq r0, r1 # skips jump to store new value in best
        jump r2 #jumps to set temp counter to 0 (9)
        init r0, 1
        add r0, r0
        add r0, r0
        store r3, r0 # if r2<r3 then M[4] = r3 (r0=1)
        init r1, 1
        add r0, r1
        init r1, 0
        store r1, r0 #if new best score found match=0
        init r0, 1
        init r1, 0
        add r0, r0 #r0=2
        add r1, r0 #r1=2
        add r0, r0 #r0=4
        add r0, r1 #r0=6
        init r1, 0
        store r1, r0 #sets temp counter (M[6]) to 0
        init r0, 0 # where it actually begins to decrement
        init r1, -1
        load r3, r0
        add r3, r1
```

```
        store r3, r0 #storing decremented counter M[0]
        init r1, 1
        load r3, r1 #loads the value for loading the pattern array
        add r3, r1
        store r3, r1
        load r2, r3 #loads the value of the pattern array
        add r1, r1 #make r1 = 2
        init r0, 1
        add r1, r0
        load r1, r1 #load the target word
        xor r1, r2 #the number of 0s in r1 is = to # of bits match
        not r1 #NOT's r1 to that the 0's become 1's
#loop: DONT TOUCH r1
        init r3, 0
        init r0, -1
        init r2, 0
        add r0, r0 #-2
        add r0, r0 #-4
        add r2, r0 #r2=-4
        add r0, r0 #-8
        add r0, r0 #-16
        add r0, r0 #-32
        add r3, r0 #-32
        add r0, r3 #-64
        add r3, r0 #-96
        add r3, r2 #-100
        init r2, 1
        init r2, 1
        init r0, 0
        add r2, r2
        load r2, r2
        beq r2, r0 # if M[2] = goes to jump at mem
        beq r0, r0 # jumps next line if above statment is not true
        jump r3 #jumps to line *big loop* in code for new pattern
        init r3, 1
        init r2, 0
        add r2, r1 # r2 = 16bit NXOR of target & pattern
        and r2  #if the first bit matches result =1
        init r0, 0
```

```
add r3, r3 #r3 = 2
add r0, r3
add r3, r3 #r3 = 4
add r3, r0 # r3 = 6
load r0, r3       #loads the existing value for best score temp
add r2, r0        #adds the result of and to the best temp score counter
store r2, r3      #saves r2 back to best temp score
slr r1
init r2, 1
add r2, r2 #r2 = 2
load r3, r2       #loads the counter (from 16 down)
init r0, -1
add r3, r0
store r3, r2 # saves the counter back in after -1
init r0, -1
init r3, 0
init r2, -1 #literally because odd numbers suck
add r0, r0 #-2
add r0, r0 #-4
add r0, r0 #-8
add r0, r0 #-16
add r3, r0 #-16
add r0, r0 #-32
add r0, r3 #-48
init r3, 0
add r2, r2 # r2 = -2   -48 from here
add r2, r2 # r2 = -4
add r3, r2 # r3 = -4
add r2, r2 # r2 = -8
add r0, r2 # r0 = -56
add r0, r3 # r0 = -60
init r3, 0 # a filler to make the jump work easier.
init r3, 0
load r2, r3
beq r2, r3 #if M[0] == 0 skip the jump & end
jump r0 #back to *loop*
jump r3 #end the program
```

P01 (By Cameron)

```
#P01
```

```
init r0, 1
init r1, 1
add r0, r0
add r0, r0
add r1, r1
add r0, r1#6
add r1, r1#4
add r1, r1#8
add r1, r1#16
add r1, r1#32
init r2, -1
add r1, r2
store r1, r0 #stores 32 into M[6]
init r0, 1
init r1, 1
add r0, r0
add r0, r0
add r1, r1
add r0, r1 #r0 = 6
add r1, r1
store r0, r1 #M[4] = 6

init r1, 1
add r1, r1
add r1, r1
load r0, r1 #loads M[4]
add r0, r0 # M[4]*2
init r2, 0
add r2, r0
add r0, r0 # M[4]*4
add r0, r2 # M[4]*6
store r0, r1 #M[4]=M[4] *6
init r0, -1
init r2, 1
add r2, r2#2
add r2, r2#4
init r3, 1
add r3, r3
```

```
add r2, r3#6
load r1, r2
add r1, r0 #decrement power by 1
store r1, r2 #store decremented power back
init r0, -2
init r3, 0
add r0, r0#-4
add r0, r0#-8
add r3, r0
add r0, r0#-16
add r0, r3#-24
add r0, r3#-32
init r2, 0
init r2, 0
init r2, 0
beq r1, r2
jump r0
jump r2
```

2)

Program 1

```
101 01 01
110 11 10
110 11 10
110 11 10
110 11 10
110 11 10
110 11 10
010 01 01
101 10 00
010 10 01
010 10 01
010 10 01
101 11 01
000 00 11
010 11 11
001 10 11
101 01 00
010 01 00
```

101 11 01
010 11 11
010 11 11
010 11 11
101 00 01
010 00 00
010 11 00
101 00 01
010 11 00
110 11 11
110 11 11
100 10 01
101 01 00
011 01 00
110 10 11
101 01 01
000 01 01
110 11 01
010 10 01
101 00 01
010 00 00
001 10 00
110 11 01
110 11 11
110 10 11
110 11 11
110 11 11
110 11 11
101 01 00
000 11 01
010 01 10
101 00 01
110 11 00
010 11 00
101 10 01
010 10 10
010 10 10
010 10 10
010 10 10

010 10 10
010 10 10
100 11 00
101 11 00
011 11 00
110 10 10
101 00 01
110 10 00
101 00 01
010 00 00
001 10 00
101 00 00
110 10 00

Program 2
101 00 01
101 01 01
101 10 01
101 11 01
010 11 10
010 10 10
010 11 11
010 00 11
101 10 11
001 10 00
101 10 11
101 00 00
010 00 11
010 11 11
010 11 10
001 11 01
101 10 01
010 11 10
010 11 11
010 10 10
001 11 10
010 11 11
101 01 00
010 01 11

010 11 11
101 10 00
010 10 11
010 11 01
010 11 00
101 00 00
001 11 00
101 00 01
101 01 01
010 01 01
010 00 00
010 00 00
010 00 00
010 00 00
001 00 01
101 00 01
101 01 01
010 01 01
010 01 01
010 00 00
010 00 01
000 10 01
000 11 00
101 01 01
101 00 11
010 01 01
010 01 01
010 01 01
010 01 01
010 00 00
010 01 00
011 10 11
110 10 01
101 00 01
101 01 01
010 00 00
010 00 00
010 00 01
000 10 00

010 10 01
001 10 00
010 01 01
010 01 01
010 00 01
010 00 00
110 10 00
100 10 11
101 01 01
101 10 01
010 10 10
010 10 10
010 10 10
010 10 01
011 00 01
110 10 10
101 00 01
010 00 00
010 00 00
001 11 00
101 01 01
010 00 01
101 01 00
001 01 00
101 00 01
101 01 00
010 00 00
010 01 00
010 00 00
010 00 01
101 01 00
001 01 00
101 00 00
101 01 11
000 11 00
010 11 01
001 11 00
101 01 01
000 11 01

010 11 01
001 11 01
000 10 11
010 01 01
101 00 01
010 01 00
000 01 01
111 01 10
110 11 01
101 11 00
101 00 11
101 10 00
010 00 00
010 00 00
010 10 00
010 00 00
010 00 00
010 00 00
010 11 00
010 00 11
010 11 00
010 11 10
101 10 01
101 10 01
101 00 00
010 10 10
000 10 10
011 10 00
011 00 00
110 10 11
101 11 01
101 10 00
010 10 01
110 01 10
101 00 00
010 11 11
010 00 11
010 11 11
010 11 00

000 00 11
010 10 00
001 10 11
110 00 01
101 10 01
010 10 10
000 11 10
101 00 11
010 11 00
001 11 10
101 00 11
101 11 00
101 10 11
010 00 00
010 00 00
010 00 00
010 00 00
010 11 00
010 00 00
010 00 11
101 11 00
010 10 10
010 10 10
010 11 10
010 10 10
010 00 10
010 00 11
101 11 00
101 11 00
000 10 11
011 10 11
110 10 00
110 10 11

P01:
101 01 11
101 11 10
101 10 01
010 11 11

010 11 11
010 11 11
010 11 01
010 10 10
010 10 10
101 01 01
010 10 01
001 11 10
101 00 01
101 01 01
010 00 00
010 00 00
010 01 01
010 00 01
010 01 01
010 01 01
010 01 01
010 01 01
101 10 11
010 01 10
001 01 00
101 00 01
101 01 01
010 00 00
010 00 00
010 01 01
010 00 01
010 01 01
001 00 01
101 01 01
010 01 01
010 01 01
000 00 01
010 00 00
101 10 00
010 10 00
010 00 00
010 00 10
001 00 01

```
101 00 11
101 10 01
010 10 10
010 10 10
101 11 01
010 11 11
010 10 11
000 01 10
010 01 00
001 01 10
101 00 10
101 11 00
010 00 00
010 00 00
010 11 00
010 00 00
010 00 11
010 00 11
101 10 00
101 10 00
101 10 00
011 01 10
110 10 00
110 10 10
```

Python Simulator code:

```
# Authors: Trung Le, Wenjing Rao, Cameron Sprouse
# This program is a simulator packed with both assembler and disassembler.
# Simulator has 2 modes:
#           Debug mode:  Execute program every # steps and
#                       output the state of each reg, and PC
#           Normal mode: Execute program all at once

def disassemble(I,Nlines):
    print("ECE366 Fall 2018 ISA Design: Disassembler")
    print("")
    #print(I)

    for i in range(Nlines):
```

```
fetch = I[i]
print(fetch)
fetch = fetch.replace(" ", "")
if(fetch[0:3]=="101"):   # init
    #fetch = fetch.replace(" ", "")
    Rx = int(fetch[3:5] ,2)
    imm = int(fetch[5:7] ,2)
    if (fetch[6:8] == "11"):
        imm = "-1"
    print("init R"+str(Rx) + ", " + str(imm))
elif(fetch[0:3]=="000"):   # load
   # fetch = fetch.replace(" ", "")
    Rx = int(fetch[3:5],2)
    Ry = int(fetch[5:7],2)
    print("load R" + str(Rx) +", R" + str(Ry) + "")
elif(fetch[0:3]=="001"):   # store
   # fetch = fetch.replace(" ", "")
    Rx = int(fetch[3:5],2)
    Ry = int(fetch[5:7],2)
    print("store R" + str(Rx) +", R" + str(Ry) + "")
elif(fetch[0:3]=="010"):   # add
  #  fetch = fetch.replace(" ", "")
    Rx = int(fetch[3:5],2)
    Ry = int(fetch[5:7],2)
    print("add R" + str(Rx) +", R" + str(Ry) )
elif(fetch[0:3]=="111"):   # xor
   # fetch = fetch.replace(" ", "")
    Rx = int(fetch[3:5],2)
    Ry = int(fetch[5:7],2)
    print("xor R" + str(Rx) +", R" + str(Ry) )
elif(fetch[0:3]=="100"):   # slt
   # fetch = fetch.replace(" ", "")
    Rx = int(fetch[3:5],2)
    Ry = int(fetch[5:7],2)
    print("slt R" + str(Rx) +", R" + str(Ry) )
elif(fetch[0:3]=="011"):   # beq
   # fetch = fetch.replace(" ", "")
    Rx = int(fetch[3:5],2)
    Ry = int(fetch[5:7],2)
```

```
            print("beq R" + str(Rx) +", R" + str(Ry) )
        elif(fetch[0:5]=="11010"):   # jump
          # fetch = fetch.replace(" ", "")
            Rx = int(fetch[5:7],2)
            print("jump R" + str(Rx) )
        elif(fetch[0:5]=="11000"):   # slr
          # fetch = fetch.replace(" ", "")
            Rx = int(fetch[5:7],2)
            print("slr R" + str(Rx) )
        elif(fetch[0:5]=="11001"):   # and
          # fetch = fetch.replace(" ", "")
            Rx = int(fetch[5:7],2)
            print("and R" + str(Rx) )
        elif(fetch[0:5]=="11011"):   # not
         #  fetch = fetch.replace(" ", "")
            Rx = int(fetch[5:7],2)
            print("not R" + str(Rx) )
        print()

def assemble(I,Nlines):
    print("ECE366 Fall 2018 ISA Design: Assembler")
    print("")

    for i in range(Nlines):
        fetch = I[i]
        print()
        #print(fetch)
        fetch = fetch.replace("R","")
        if (fetch[0:4] == "init"):
            fetch = fetch.replace("init ","")
            fetch = fetch.split(",")
            R = format(int(fetch[0]),"02b")
            imm = format(int(fetch[1]))
            if (imm == "-1"):
                imm = "11"
            elif (imm == "-2"):
                imm = "10"
            else:
                imm = format(int(fetch[1]),"02b")
```

```
    op = "101"
    print(op + " " + R + " " + imm)

elif (fetch[0:4] == "add "):
    fetch = fetch.replace("add ","")
    fetch = fetch.split(",")
    Rx = format(int(fetch[0]),"02b")
    Ry = format(int(fetch[1]),"02b")
    op = "010"
    print(op + " " + Rx + " " + Ry)

elif (fetch[0:4] == "xor "):
    fetch = fetch.replace("xor ","")
    fetch = fetch.split(",")
    Rx = format(int(fetch[0]),"02b")
    Ry = format(int(fetch[1]),"02b")
    op = "111"
    print(op + " " + Rx + " " + Ry)

elif (fetch[0:4] == "load"):
    fetch = fetch.replace("load ","")
    fetch = fetch.replace("(","")
    fetch = fetch.replace(")","")
    fetch = fetch.split(",")
    Rx = format(int(fetch[0]),"02b")
    Ry = format(int(fetch[1]),"02b")
    op = "000"
    print ( op + " " + Rx + " " + Ry)

elif (fetch[0:5] == "store"):
    fetch = fetch.replace("store ","")
    fetch = fetch.replace("(","")
    fetch = fetch.replace(")","")
    fetch = fetch.split(",")
    Rx = format(int(fetch[0]),"02b")
    Ry = format(int(fetch[1]),"02b")
    op = "001"
    print ( op + " " + Rx + " " + Ry)
elif (fetch[0:4] == "slt "):
```

```
    fetch = fetch.replace("slt ","")
    fetch = fetch.split(",")
    Rx = format(int(fetch[0]),"02b")
    Ry = format(int(fetch[1]),"02b")
    op = "100"
    print ( op + " " + Rx + " " + Ry)

elif (fetch[0:4] == "beq "):
    fetch = fetch.replace("beq ","")
    fetch = fetch.split(",")
    Rx = format(int(fetch[0]),"02b")
    Ry = format(int(fetch[1]),"02b")
    op = "011"
    print(op + " " + Rx + " " + Ry)

elif (fetch[0:4] == "jump"):
    fetch = fetch.replace("jump ","")
    fetch = fetch.split(",")
    Rx = format(int(fetch[0]),"02b")
    op = "110 10"
    print(op + " " + Rx)
elif(fetch[0:4] == "slr "):
    fetch = fetch.replace("slr ", "")
    fetch = fetch.split(",")
    Rx = format(int(fetch[0]), "02b")
    op = "110 00"
    print(op + " " + Rx)
elif(fetch[0:4] == "and "):
    fetch = fetch.replace("and ", "")
    fetch = fetch.split(",")
    Rx = format(int(fetch[0]), "02b")
    op = "110 01"
    print(op + " " + Rx)
elif(fetch[0:4] == "not "):
    fetch = fetch.replace("not ", "")
    fetch = fetch.split(",")
    Rx = format(int(fetch[0]), "02b")
    op = "110 11"
    print(op + " " + Rx)
```

```
def simulate(I,Nsteps): #NEEDS TO BE DONE IN MACHINE CODE!!!
    print("ECE366 Fall 2018 ISA Design: Simulator")
    print()
    PC = 0              # Program-counter
    DIC = 0
    Reg = [0,0,0,0]     # 4 registers, init to all 0
    Memory = [0 for i in range(110)] # data memory, 10 spaces all init to 0.

    Memory = ["1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
```

```
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
```

"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",
"1111111111111111",

```
"1111111111111111"]
    print("******** Simulation starts *********")
    finished = False
    while(not(finished)):
        fetch = I[PC]
        DIC += 1
        print(fetch)
        fetch = fetch.split(" ") # Delete all the spaces to make things simpler
        if (fetch[0] == "101"): #init
            #fetch = fetch.replace("init ","") dont need I think
            #fetch = fetch.split(",")
            Rx = int(fetch[1],2)
            if (int(fetch[2] == "11")):
                imm = -1
            elif(int(fetch[2] == "10")):
                imm = -2
            else:
                imm = int(fetch[2],2)
            Reg[Rx] = imm
            PC += 1
        elif (fetch[0] == "010"): #add
            #fetch = fetch.replace("add ","")
            #fetch = fetch.split(",")
            Rx = int(fetch[1],2)
            Ry = int(fetch[2],2)
            Reg[Rx] = int(Reg[Rx]) + int(Reg[Ry])
            PC += 1
        elif (fetch[0] == "111"):# xor
            #fetch = fetch.replace("xor ","")
            #fetch = fetch.split(",")
            Rx = int(fetch[1],2)
            Ry = int(fetch[2],2)
            Reg[Rx] = int(Reg[Rx]) ^ int(Reg[Ry])
            PC += 1
        elif (fetch[0] == "000"):# load
            #fetch = fetch.replace("load ","")
            #fetch = fetch.split(",")
            Rx = int(fetch[1],2)
            Ry = int(fetch[2],2)
```

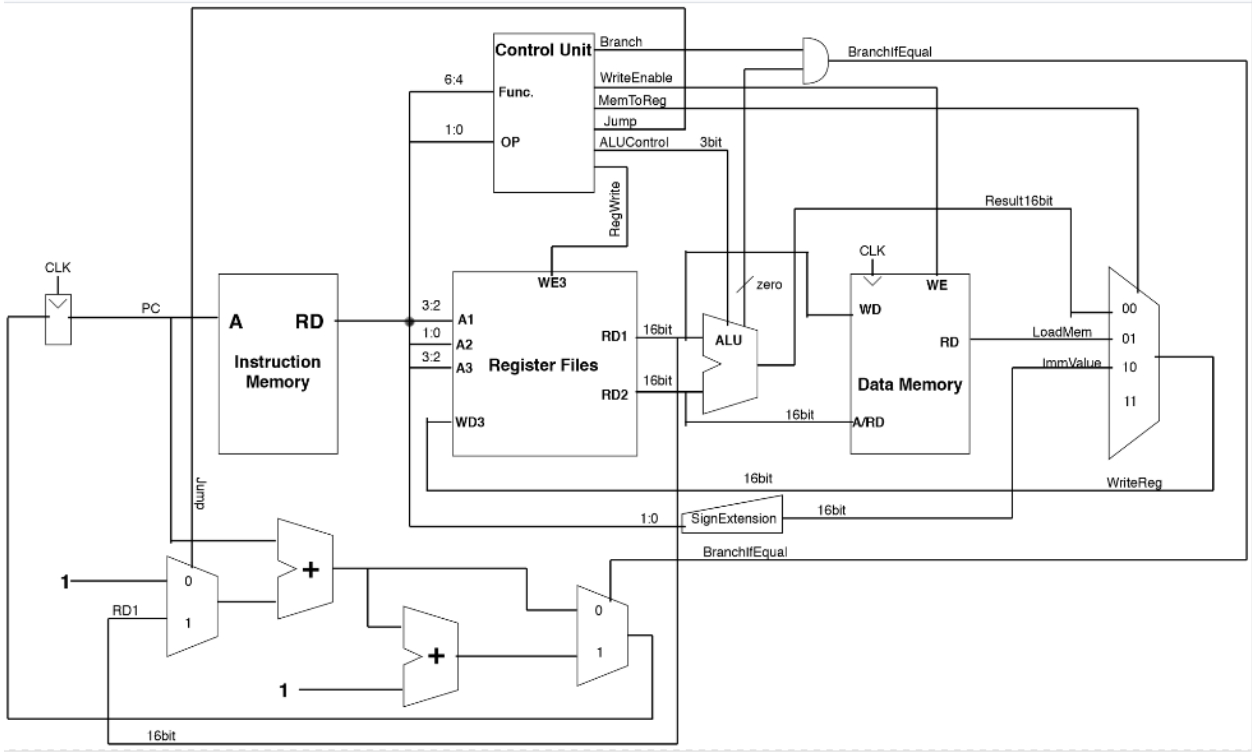
```
    Reg[Rx] = Memory[Reg[Ry]]
    print(Reg[Rx])
    PC += 1
elif (fetch[0] == "001"):# store
    #fetch = fetch.replace("store ","")
    #fetch = fetch.split(",")
    Rx = int(fetch[1],2)
    Ry = int(fetch[2],2)
    print (Reg[Rx])
    Memory[Reg[Ry]] = Reg[Rx]
    PC += 1
elif (fetch[0] == "100"):# slt
    #fetch = fetch.replace("slt ","")
    #fetch = fetch.split(",")
    Rx = int(fetch[1],2)
    Ry = int(fetch[2],2)
    if( int(Reg[Rx]) < int(Reg[Ry]) ):
        Reg[0] = 1
    else:
        Reg[0] = 0
    PC += 1
elif (fetch[0] == "011"):# beq
    #fetch = fetch.replace("beq ","")
    #fetch = fetch.split(",")
    Rx = int(fetch[1],2)
    Ry = int(fetch[2],2)

    if ( Reg[Rx] == Reg[Ry]):
        PC = PC + 2
    else:
        PC += 1
elif (fetch[0] == "110" and fetch[1] == "10" ):# jump
    #fetch = fetch.replace("jump ","")
    #fetch = fetch.split(",")
    Rx = int(fetch[2],2)
    if(Reg[Rx] == 0):
        finished = True
    else:
        PC = PC + Reg[Rx]
```

```
        elif (fetch[0] == "110" and fetch[1] == "00" ):# slr
            Rx = int(fetch[2],2)
            Reg[Rx] = format(int(Reg[Rx],2) >> 1, '016b')
            print(Reg[Rx])
            print(Reg[Ry])
            #finished = True
            PC = PC + 1
        elif (fetch[0] == "110" and fetch[1] == "01" ):# and
            Rx = int(fetch[2],2)
            Reg[Rx] = int(Reg[Rx]) & 1
            #finished = True
            PC = PC + 1
        elif (fetch[0] == "110" and fetch[1] == "11" ):# not
            Rx = int(fetch[2],2)
            Reg[Rx] = format(1111111111111111 - int(Reg[Rx]), '016')
            PC = PC + 1
        if ( (DIC % Nsteps) == 0):
            print("Registers R0-R3: ", Reg)
            print("Memory: ",Memory)
            print()

    print("******** Simulation finished *********")
    print("Dynamic Instr Count: ",DIC)
    print("Registers R0-R3: ",Reg)
    print("Memory :",Memory)


def main():
    input_file = open("Part1(actual one)NC.txt","r")#Part2ISACodeWOComments
Part2MachineCode
    debug_mode = False  # is machine in debug mode?
    Nsteps = 10        # How many cycle to run before output statistics
    Nlines = 0         # How many instrs total in input.txt
    Instruction = []    # all instructions will be stored here
    mode = 3           # 1 = Simulation
                       # 2 = disassembler
                       # 3 = assembler
    for line in input_file:
        if (line == "\n" or line[0] =='#'):             # empty lines,comments ignored
```

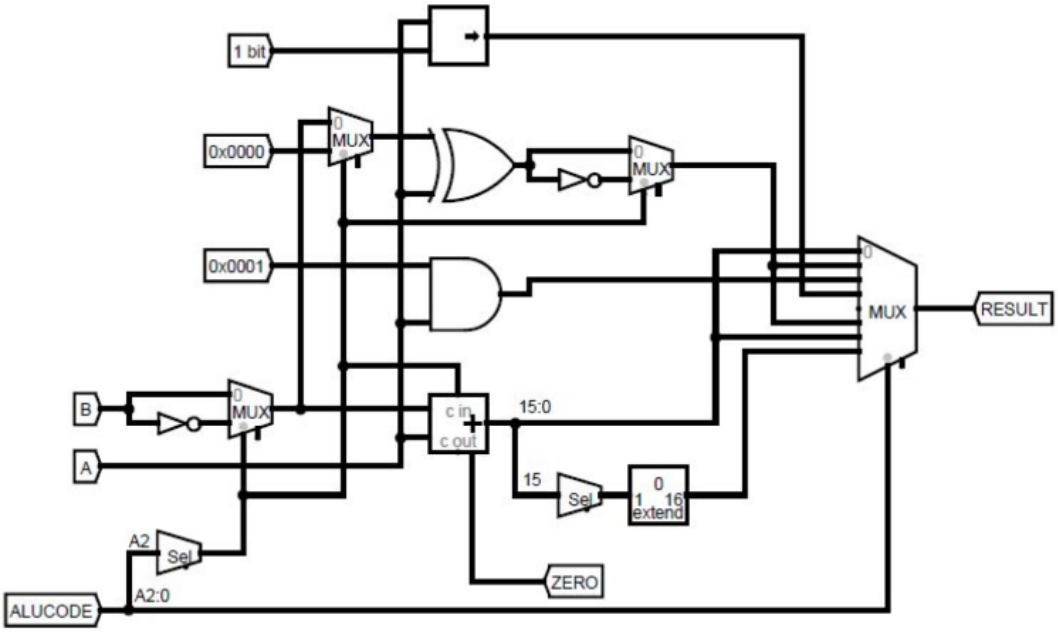
```
            continue
        line = line.replace("\n","")
        Instruction.append(line)                    # Copy all instruction into a list
        Nlines +=1

    if(mode == 1):   # Check wether to use disasembler or assembler or simulation
        simulate(Instruction,Nsteps)
    elif(mode == 2):
        disassemble(Instruction,Nlines)
    else:
        assemble(Instruction,Nlines)
```

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

CPU datapath



ALU Design



ALU DECODER TABLE:

NAME	FUNCTION	OP	ALUCODE
beq (Branch equal)	011	XX	110
slt (Set less than)	100	XX	111
not (Bitwise not)	110	11	101
add (Signed addition)	010	XX	000
xor (Bitwise xor)	111	01	001
slr (Logical shift right)	110	XX	011
and (Bitwise and with 1)	110	00	010

ISA CONTROL DECODE TABLE:

Instruction	Function	OP	RegWrite	Branch	Jump	WriteEnable	MemtoReg	ALU CONTROL 2:0
load	000	XX	1	0	0	0	01	XXX
store	001	XX	0	0	0	1	XX	XXX
beq	011	XX	0	1	0	0	XX	110
slt	100	XX	1	0	0	0	00	111
not	110	11	1	0	0	0	00	101
add	010	XX	1	0	0	0	00	000
and	110	01	1	0	0	0	00	010
xor	111	XX	1	0	0	0	00	001
slr	110	00	1	0	0	0	00	011
jump	110	10	0	0	1	0	XX	XXX
init	101	XX	1	0	0	0	10	XXX