sFile = ['LDRL r2,1','LDRL r0,4','NOP','STRI r0,[r2]','LDRI r3,[r2]', /

'INC r3','ADDL r3,r3,2','NOP','DEC r3','BNE -2','DEC r3','STOP'] # Source program for testing

# Simple CPU instruction interpreter. Direct instruction interpretation. 30 September 2022. V1.0

# Class 0: no operand NOP

# Class 1: literal BEQ 3

# Class 2: register INC r1

# Class 3: register,literal LDRL **r1**,5

# Class 4: register,register, MOV **r1**,r2

# Class 5: register,register,literal ADDL **r1**,r2,5

# Class 6: register,register,register ADD **r1**,r2,r3

# Class 7: register,[register] LDRI r1,[r2]

codes = {'NOP':[0],'STOP':[0],'BEQ':[1],'BNE':[1],'BRA':[1],'INC':[2],'DEC':[2],'CMPL':[3], /

'LDRL':[3],'MOV':[4],'CMP':[4],'SUBL':[5],'ADDL':[5],'ANDL':[5],'ADD':[6],'SUB':[6], /

'AND':[6],'LDRI':[7],'STRI':[7]}

reg1 = {'r0':0,'r1':1,'r2':2,'r3':3} # Legal registers

reg2 = {'[r0]':0,'[r1]':1,'[r2]':2,'[r3]':3} # Legal pointer registers

r = [0] \* 4 # Four registers

r[0],r[1],r[2],r[3] = 1,2,3,4 # Preset registers for testing

m = [0] \* 8 # Eight memory locations

pc = 0 # Program counter initialize to zero

go = 1 # go is the run control (1 to run)

z = 0 # z is the zero flag. Set/cleared by SUB, DEC, CMP

while go == 1: # REPEAT execute fetch and execute loop

thisLine = sFile[pc] # Get current instruction

pc = pc + 1 # Increment pc

pcOld = pc # Remember pc value for this cycle

temp = thisLine.replace(',',' ') # Remove commas: ADD r1,r2,r3 to ADD r1 r2 r3

tokens = temp.split(' ') # Tokenize: ADD r1 r2 r3 to ['ADD','r1','r2','r3']

mnemonic = tokens[0] # Extract first token, the mnemonic

opClass = codes[mnemonic][0] # Extract instruction class

# Process the current instruction and analyze it

rD,rDval,rS1,rS1val,rS2,rS2val,lit,rPnt,rPntV = 0,0,0,0,0,0,0,0,0 # Clear all parameters

if opClass in [0]: pass # If class 0, nothing to be done (simple op-code only)

if opClass in [2,3,4,5,6,7,8]: # Look for ops with destination register rD

rD = reg1[tokens[1]] # Get token 1 and use it to get register number as rD

rDval = r[rD] # Get contents of register rD

if opClass in [4,5,6]: # Look at instructions with first source register rS1

rS1 = reg1[tokens[2]] # Get rS1 register number and then contents

rS1val = r[rS1]

if opClass in [6]: # If class 6, it’s got three registers. Extract rS2

rS2 = reg1[tokens[3]] # Get rS2 and rS2val

rS2val = r[rS2]

if opClass in [1,3,5,8]: # The literal is the last element in instructions

lit = int(tokens[-1]) # Get the literal

if opClass in [7]: # Class 7 involves register indirect addressing.

rPnt = reg2[tokens[2]] # Get the pointer (register) and value of the pointer

rPntV = r[rPnt] # Get the register number

if mnemonic == 'STOP': # Now execute instructions. If STOP, clear go and exit

go = 0

print('Program terminated')

elif mnemonic == 'NOP': pass # NOP does nothing. Just drop to end of loop

elif mnemonic == 'INC': r[rD] = rDval + 1 # Increment: (add 1 to destination register)

elif mnemonic == 'DEC': # Decrement: subtract 1 from register and update z bit

z = 0

r[rD] = rDval - 1

if r[rD] == 0: z = 1

elif mnemonic == 'BRA': # Unconditional branch

pc = pc + lit - 1

elif mnemonic == 'BEQ': # Conditional branch on zero

if z == 1: pc = pc + lit - 1

elif mnemonic == 'BNE': # Conditional branch on not zero

if z == 0: pc = pc + lit - 1

elif mnemonic == 'ADD': r[rD]=rS1val+rS2val # Add

elif mnemonic == 'ADDL': r[rD] = rS1val+lit # Add literal

elif mnemonic == 'SUB': # Subtract and set/clear z

r[rD] = rS1val - rS2val

z = 0

if r[rD] == 0: z = 1

elif mnemonic == 'SUBL': # Subtract literal

r[rD] = rS1val - lit

z = 0

if r[rD] == 0: z = 1

elif mnemonic == 'CMPL': # Compare literal

diff = rDval - lit

z = 0

if rDval == lit: z = 1

elif mnemonic == 'CMP': # Compare

diff = rDval - rS1val

z = 0

if rDval == lit: z = 1

elif mnemonic == 'MOV': r[rD] = rS1val # Move, load and store operations

elif mnemonic == 'LDRL': r[rD] = lit

elif mnemonic == 'LDRI': r[rD] = m[rPntV]

elif mnemonic == 'STRI': m[rPntV] = rDval

regs = " ".join("%02x" % b for b in r) # Format memory locations hex

mem = " ".join("%02x" % b for b in m) # Format registers hex

print('pc =','{:<3}'.format(pcOld), "{:<14}".format(thisLine),'Regs =', regs, 'Mem =',mem, 'z =', z)

x = input('>>> ')