COMPUTER ARCHITECTURE ASSIGNMENT-1

INTRODUCTION:

In our project, we have written a program to find the roots of a quadratic equation.We have implemented this using the IAS instruction set architecture. We have added two new instructions that we designed: the DISC instruction and the SQRT instruction. The program prints the roots of the quadratic equation if they are real. The high-level language is written in the C programming language, and the assembler and the processor are implemented using the Python programming language.

QUADRATIC ROOTS:

ax^2 + bx + c= 0. We are finding the roots of the quadratic equation by first calculating the discriminant. If it is positive, we proceed to calculate the roots. Otherwise, we exit from the program.

C PROGRAM:

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void roots(int a, int b, int c)
{
    if (a == 0) return;
    int discrim = b*b - 4*a*c;
    float sqrt_discrim = sqrt(discrim);
    //x1 and x2 are roots of our quadratic
    if (discrim >= 0)
    printf("x1= %f \n x2= %f ", (-b + sqrt_discrim)/(2*a) , (-b - sqrt_discrim)/(2*a));
}

void main()
{
    int a,b,c;  //ax^2 +bx +c is the quadratic
    scanf("%d%d%d",&a,&b,&c);
    roots(a,b,c);
}
```

ASSEMBLY PROGRAM:

```
M={ a, b, c, a_2, dis, 4, 2, root_d, x1, x2  }

DISC M[0],M[1],M[2] //these are represented by 4 bits //496
STOR M[4] //497
JUMP + M[500,0:19] //498
HALT //499
SQRT M[4] STOR M[7] //500
LOAD MQ,M[0] //501
MUL M[6] LOAD MQ //502
STOR M[3] //503
//x1
LOAD -M[1] ADD M[7] //504
DIV M[3] LOAD MQ //505
STOR M[8] //506
//x2
LOAD -M[1] SUB M[7] //507
DIV M[3] LOAD MQ //508
STOR M[9] //509
EXIT //510
```

DISC instruction: We use this instruction to calculate the discriminant of a quadratic equation. We send the memory locations at which a, b, c are stored and extract them for use. The result is stored in AC.(Assumption: -15<= a , b , c <=15)

SQRT instruction: This instruction calculates the square root of a value stored in memory M at an address specified by the binary representation of the address. The result is stored in the accumulator AC.

ASSEMBLER:

An assembler converts assembly language code into machine code. Assemblers bridge the gap between high-level programming languages and hardware.

In our assembler we have three functions:

Function 1 is opcode, which returns the opcode of the given instruction as described in the IAS architecture.

```python
def opcode(instruction):
    if(instruction == 'LOAD -'):
        return '00000010'
    elif(instruction == 'LOAD MQ,M(X)'):
        return '00001001'
    elif instruction=="LOAD MQ":
        return '00001010'
    elif instruction == 'DISC':
        return '01010101'
    elif(instruction == 'JUMP'):
        return '00001111'
    elif(instruction == 'SQRT'):
        return '00110011'
    elif(instruction == 'STOR'):
        return '00100001'
    elif(instruction == 'MUL'):
        return '00001011'
    elif(instruction == 'SUB'):
        return '00000110'
    elif(instruction == 'ADD'):
        return '00000101'
    elif(instruction == 'HALT'):
        return '00000000'
    elif(instruction == 'DIV'):
        return '00001100'
    else:
        return None
```

Function 2 is convert_binary(), whose task is to convert a decimal integer into binary and then return it as a string.

```python
def convert_binary(x, my_opcode):
    x=int(x)
    binary_num=bin(x)
    binary_lst=list(binary_num[2:])
    n= len(binary_lst)

    if my_opcode=="01010101":
        for i in range(4,n,-1):
            binary_lst.insert(0,'0')
    else:
        for i in range(12,n,-1):
            binary_lst.insert(0,'0')

    return "".join(binary_lst)
```

Function 3 is mach_lang_prog() , it takes input of an empty string, mach_lang, in which the final machine language code is stored. The input is taken one line at a time. The components are stored in a list. The components are then compared, and the respective opcode and address are returned in binary.

```python
def mach_lang_prog(mach_lang):
    print("type the commands:")
    zeroes="00000000000000000000"
    while(1):
        ins_list=list(input("").split())
        if (ins_list[0] == "EXIT"): #EXIT
            break
        else:
            #special for jump
            if(ins_list[0] == "JUMP"): #JUMP +
                op=opcode("JUMP")
                binary=convert_binary(ins_list[2][2:5], op)
                address=op+binary+zeroes
            #lhs only
            elif(len(ins_list)==2):
                if(ins_list[0]=="DISC"): #DISC
                    op=opcode("DISC")
                    mem_a=ins_list[1][2] #m[0] ka 0
                    mem_b=ins_list[1][7] #1
                    mem_c=ins_list[1][12] #2
                    binary=convert_binary(mem_a,op)+convert_binary(mem_b,op)+convert_binary(mem_c,op)
                    address=op+binary+zeroes
                elif(ins_list[0]=="LOAD" and ins_list[1][0:4]=="MQ,M"): #LOAD MQ,MX
                    op=opcode("LOAD MQ,M(X)")
                    binary=convert_binary(ins_list[1][5],op)
                    address=op+binary+zeroes
                elif(ins_list[0]=="LOAD" and ins_list[1][0:2]=="MQ"): #LOAD MQ
                    op=opcode("LOAD MQ")
                    binary=convert_binary(ins_list[1][5],op)
                    address=op+binary+zeroes
                else:
                    op=opcode(ins_list[0]) #STOR M[x]
                    str1=ins_list[1] #should this be 2?
                    binary=convert_binary(str1[2],op)
                    address=op+binary+zeroes
```

```python
            #lhs and rhs
            elif(len(ins_list)==4):
                if(ins_list[1][0]=="-"): #LOAD-M[x]
                    op=opcode("LOAD -")
                    binary=convert_binary(ins_list[1][3],op)
                    address=op+binary
                    op=opcode(ins_list[2])
                    binary=convert_binary(ins_list[3][2],op)
                    address=address+op+binary
                elif (ins_list[2]=="LOAD" and ins_list[3]=="MQ"): ## lhs , load mq
                    op=opcode(ins_list[0])
                    binary=convert_binary(ins_list[1][2],op)
                    address=op+binary
                    op= opcode("LOAD MQ")
                    binary= "000000000000"
                    address=address+op+binary
                elif (ins_list[1][1]=="-"):  #LOAD -Mx
                    op= opcode("LOAD -")
                    binary = convert_binary(ins_list[1][3],op)
                    address= op+binary
                    op=opcode(ins_list[2])
                    binary=convert_binary(ins_list[3][2],op)
                    address=address+op+binary
                else:
                    op=opcode(ins_list[0])
                    binary=convert_binary(ins_list[1][2],op)
                    address=op+binary
                    op=opcode(ins_list[2])
                    binary=convert_binary(ins_list[3][2],op)
                    address=address+op+binary
        mach_lang.append(address)
    print("The machine language program")
    for i in mach_lang:
        print(i, end="")
    print("\n")
    return mach_lang
```

PROCESSOR:

The processor executes instructions according to a defined process. The code body calls the function "mach_lang_prog" from the assembler, which returns the machine language program. It then executes the instructions provided in assembly language one by one using the machine language code. If the instruction is "EXIT", the program terminates; otherwise, it continues to execute until it encounters an EXIT instruction.

Function 1 is my_operation() , which executes the instruction provided to it.

```python
import IMT2023020_IMT2023034_IMT2023059_assembler

MAR=""
IR=""
PC= 0
IBR=""
AC= 0
MQ= 0
MBR= 0
ext= 0
a, b, c, a_2, dis, root_d, x1, x2=0,0,0,0,0,0,0,0
M = [ a, b, c, a_2, dis, 4, 2, root_d, x1, x2 ]
#equation is ax^2 +bx +c
#a_2 stores the value of 2a needed for denominator
#dis stores the value of discriminant
#x1 and x2 are roots

def my_operation(op, address):
    global PC
    global MAR
    global IR
    global IBR
    global MBR
    global AC
    global MQ


    if op=="01010101": #DISC M[x],M[x],M[x]
        bin_a=address[0:4]
        bin_b=address[4:8]
        bin_c=address[8:12]
        a=M[int(bin_a,2)]
        b=M[int(bin_b,2)]
        c=M[int(bin_c,2)]
        discriminant= (b**2) - (4*a*c)
        AC = discriminant
        print(f"AC={AC}")
        print("\n")
```

```python
elif op=="00000010": #LOAD -M[x]
    MBR= M[int(address,2)]
    AC= -MBR
    print(f"AC={AC}")
    print("\n")
elif op=="00100001": #STOR M[x]
    MBR=AC
    M[int(address,2)] = MBR
    print(f"AC={AC}")
    print("\n")
elif op=="00110011": #SQRT M[x]
    MBR= M[int(address,2)]
    AC=MBR
    AC=AC**0.5
    print(f"AC={AC}")
    print("\n")
elif op=="00001111": #JUMP
    if AC>=0:
        PC = int(address,2) - 1
elif op == '00000101': #ADD M[x]
    MBR = M[int(address, 2)]
    AC = AC + MBR
    print(f'AC = {AC}')
    print('\n')
elif op == '00000110': #SUB M[x]
    MBR = M[int(address,2)]
    AC = AC-MBR
    print(f"AC = {AC}")
    print("\n")
elif op == "00001100": #DIV M[x]
    MBR = M[int(address,2)]
    MQ = AC / MBR
    AC = AC % MBR
    print(f'AC ={AC}')
    print(f'MQ ={MQ}')
    print("\n")
elif op == '00001011': #MUL M[x]
    MBR = M[int(address,2)]
    AC = MBR
    MQ = AC * MQ
    print(f'AC = {AC}')
    print(f'MQ = {MQ}')
    print("\n")
```

```python
elif op=="00001001": #LOAD MQ,M[x]
    MBR = M[int(address,2)]
    MQ= MBR
    print(f"MQ = {MQ}")
    print("\n")
elif op=="00001010": #LOAD MQ
    AC=MQ
    print(f"MQ = {MQ}")
    print(f'AC = {AC}')
    print("\n")
elif op=="00000010": #LOAD -M[x]
    AC = -M[int(address,2)]
    print(f'AC = {AC}')
    print("\n")
```

Function 2 is decode_instruction() which receives the word and potrays the instruction cycle.

```python
def decode_instruction(word): #word is a string , stored at m[pc]
    global PC
    global MAR
    global IR
    global IBR
    global MBR
    global AC
    global MQ

    #assume pc set to address of instrucion
    MAR=str(PC)

    MBR=word #first the word goes to MBR from M[MAR]

    #transfer the word present in the MBR
    left_instruction = MBR[0:20]
    left_opcode = MBR[0:8]
    left_address = MBR[8:20]
    right_instruction = MBR[20:40]
    right_opcode = MBR[20:28]
    right_address = MBR[28:40]

    IR = left_opcode #From MBR , left address opcode goes to IR
    MAR = left_address #from MBR , left address goes to MAR
    IBR = right_instruction #from MBR right instruction goes to IBR

    my_operation(IR,MAR)#execute left instruction

    IR = IBR[0:8] #from IBR , load right opcode in IR
    MAR = IBR[8:20] #from IBR , load right address in MAR

    my_operation(IR, MAR) #execute right instruction

    PC= int(PC) + 1 #increment PC to point to next word's memory location
```

```python
print('''Enter the values of a , b , c corresponding to ax^2 + bx +c :
      note: a<=15, b<=15, c<=15''')
a_in= int(input("a:"))
b_in= int(input("b:"))
c_in= int(input("c:"))

M[0]=a_in
M[1]=b_in
M[2]=c_in
PC=496
i=0
mach_lang=[]
mach_lang = IMT2023020_IMT2023034_IMT2023059_assembler.mach_lang_prog(mach_lang)

i=0
while (1):
    if PC==510:
        break
    if mach_lang[i]=="0000000000000000000000000000000000000000" or PC==499: #HALT
        break

    decode_instruction(mach_lang[i])
    print(f"PC={PC}")
    print("\n\n")
    i+=1

if (PC==499):
    print("No real roots exist since d<0") # if d<0
else:
    print(f"The roots are {M[8]} and {M[9]}")  #roots are stored in these memory locations
```

RESULT:

```
Enter the values of a , b , c corresponding to ax^2 + bx +c :
      note: a<=15, b<=15, c<=15
a:1
b:6
c:9
type the commands:
DISC M[0],M[1],M[2]
STOR M[4]
JUMP + M[500,0:19]
HALT
SQRT M[4] STOR M[7]
LOAD MQ,M[0]
MUL M[6] LOAD MQ
STOR M[3]
LOAD -M[1] ADD M[7]
DIV M[3] LOAD MQ
STOR M[8]
LOAD -M[1] SUB M[7]
DIV M[3] LOAD MQ
STOR M[9]
EXIT
The machine language program
0101010100000000100100000000000000000000001000010000000001000000000000000000000000001111000111
110100000000000000000000000000001111000111110100000000000000000000000011001100000000010000100001
000000000111000010010000000000000000000000000000000000001011000000000011000010100000000000000
1000010000000000110000000000000000000000000010000000000001000011000000000000110000101000000000
0000001000010000000010000000000000000000000000000001000000000000100001100000000000011000010100
00000000000010000100000000100100000000000000000000
```

```
AC=0

PC=497

AC=0

PC=498

PC=500

PC=500

AC=0.0

AC=0.0

PC=501

MQ = 1

PC=502

AC = 2
MQ = 2
```

```
PC=503


AC=2


PC=504


AC=-6


PC=505


AC =0
MQ =-3.0


MQ = -3.0
AC = -3.0


PC=506


AC=-3.0


PC=507


AC=-6


PC=508
```

```
AC =0
MQ =-3.0


MQ = -3.0
AC = -3.0


PC=509



AC=-3.0


PC=510



The roots are -3.0 and -3.0
```