ELEMENTS OF COMPUTING SYSTEMS-2

Project By:

Dinesh S	CB.EN.U4AIE19025
Harish K	CB.EN.U4AIE19029
Kabilan N	CB.EN.U4AIE19033
Sivamaran M.A.C	CB.EN.U4AIE19061
Roshan Tushar S	CB.EN.U4AIE19071

ACKNOWLEDGEMENT :-

I like to thank all those who have helped us with the project of designing an ASSEMBLER and coding a ASSEMBLY PROGRAM and creating an ARCHITECTURE under the subject “ELEMENTS OF COMPUTING SYSTEMS - 2” in our branch CSE-AI. Specially I would like to thank the faculties Dr.Govind & prof.Vijay Krishna Menon who have helped us with all the necessary knowledge and improve our project to yield the best possible result. I would thank my teammates and we gave it our best which gave us a positive result. This project helped un increasing our conceptual, technical and inner knowledge of the subject.

ABSTRACT:

CPU architecture defines the basic instruction set, as well as the exception and memory models that are relied upon by the operating system and hypervisor. The CPU microarchitecture determines how an implementation meets the architectural contract. This project marks the phase when hardware meets the software. The project is separated into three parts. In part 1 we familiarize with low level language by successfully computing the factorial of a number in hack language provided in nand2netris course. In part 2 we have built an assembler in java and python to convert the assembly program obtained from part 1 to machine codes. In part 3 we design and implement a CPU architecture, ensuring a connection between selection bits and data buses during execution

TABLE OF CONTENTS:-

ALGORITHM:-

For Example: If we are going to find 4!..

Which is 1*2*3*4

Step1:Start

Step2:We take the first two elements.

Step3: We add the first element ,second element times.

Step4: Next we replace the first two elements with the sum calculated in step3

Step5: We repeat the process from step2 to step4 until the last element is n.

Step6: End

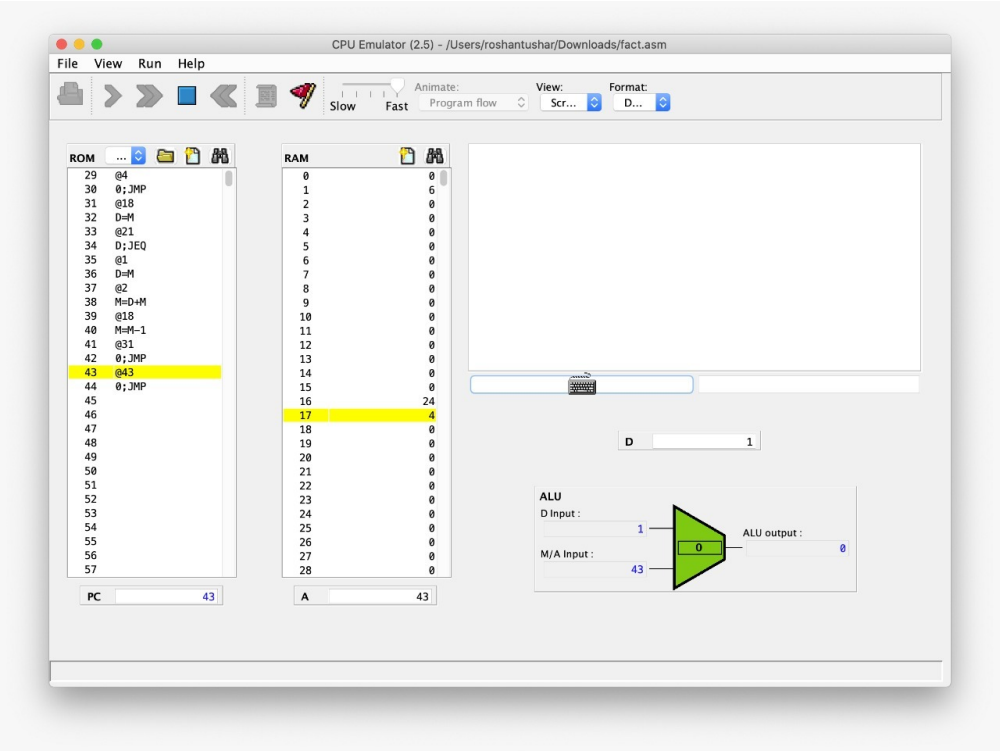
```
>> 1*2*3*4

ans =

     24

>> 1+1

ans =

      2

>> 2+2+2

ans =

      6

>> 6+6+6+6

ans =

     24
```

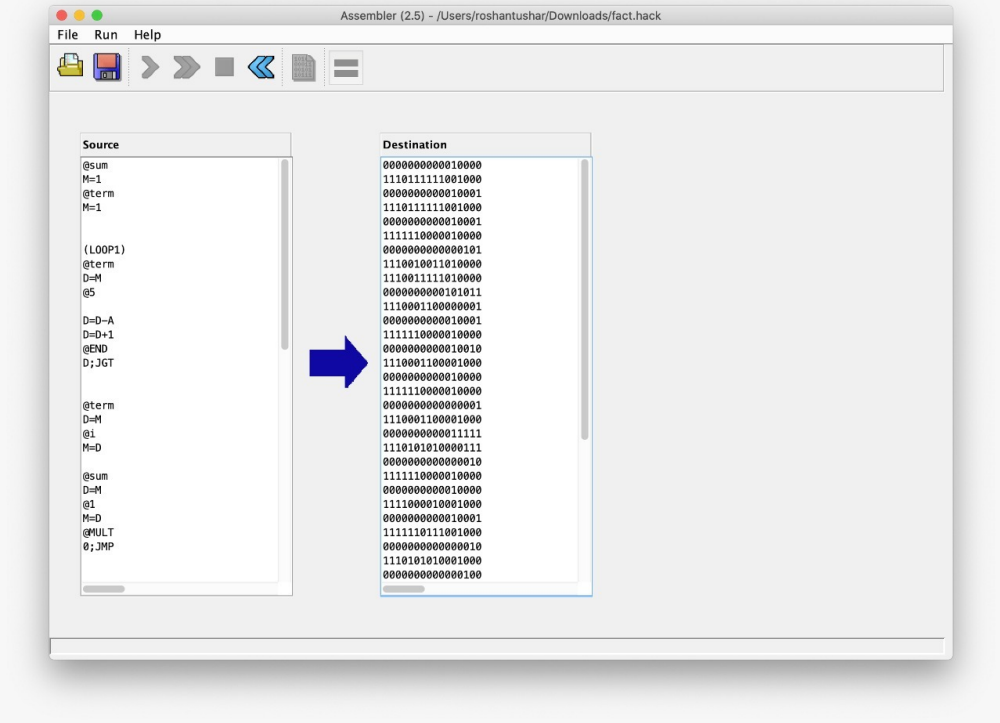
Similarly for 5!, 6!.....It continues

ASM CODE:-

```
@sum
M=1
@term
M=1

(LOOP1)
@term
D=M
@5

D=D-A
D=D+1
@END
D;JGT


@term
D=M
@i
M=D

@sum
D=M
@1
M=D
@MULT
0;JMP
```

```
(LOOP2)
@2
D=M
@sum
M=M+D
@term
M=M+1
@2
M=0

@LOOP1
0;JMP


(MULT)
   @i
   D=M
   @LOOP2
   D;JEQ //if i=0, return
   @1
   D=M
   @2
   M=M+D //R2 = R2 + R1
   @i
   M=M-1 //Decrement i
   @MULT
   0;JMP

(END)
@END
0;JMP
```


CPU Emulator (2.5) - /Users/roshantushar/Downloads/fact.asm
File View Run Help
Slow Fast
Animate:
Program flow
View:
Format:
ROM
29 @4
30 0;JMP
31 @18
32 D=M
33 @21
34 D;JEQ
35 @1
36 D=M
37 @2
38 M=D+M
39 @18
40 M=M-1
41 @31
42 0;JMP
43 @43
44 0;JMP
45
46
47
48
49
50
51
52
53
54
55
56
57
RAM
0 0
1 6
2 0
3 0
4 0
5 0
6 0
7 0
8 0
9 0
10 0
11 0
12 0
13 0
14 0
15 0
16 24
17 4
18 0
19 0
20 0
21 0
22 0
23 0
24 0
25 0
26 0
27 0
28 0
PC 43
A 43
D 1
ALU
D Input :
1
M/A Input :
43
0
ALU output :
0


Assembler (2.5) - /Users/roshantushar/Downloads/fact.hack
File Run Help
Source
@sum
M=1
@term
M=1
(LOOP1)
@term
D=M
@5
D=D-A
D=D+1
@END
D;JGT
@term
D=M
@i
M=D
@sum
D=M
@1
M=D
@MULT
0;JMP
Destination
0000000000010000
1110111111001000
0000000000010001
1110111111001000
0000000000010001
1111110000010000
0000000000000101
1110010011010000
1110011111010000
0000000000101011
1110001100000001
0000000000010001
1111110000010000
0000000000010010
1110001100001000
0000000000010000
1111110000010000
0000000000000001
1110001100001000
0000000000011111
1110101010000111
0000000000000010
1111110000010000
0000000000010000
1111000010001000
0000000000010001
1111110111001000
0000000000000010
1110101010001000
0000000000000100

ASSEMBLER:-

Python code:-

```
#C INSTRUCTIONS


DEST = {
    'null' : '000',
    'M' : '001',
    'D' : '010',
    'MD' : '011',
    'A' : '100',
    'AM' : '101',
    'AD' : '110',
    'AMD': '111',
}


COMP = {
    '0': '0101010',
    '1': '0111111',
    '-1':'0111010',
    'D': '0001100',
    'A': '0110000',
    '!D': '0001101',
    '!A': '0110001',
    '-D': '0001111',
    '-A': '0110011',
    'D+1':'0011111',
    'A+1': '0110111',
    'D-1': '0001110',
    'A-1': '0110010',
    'D+A': '0000010',
    'D-A': '0010011',
    'A-D': '0000111',
```

```
    'D&A': '0000000',
    'D|A': '0010101',
    'M':   '1110000',
    '!M':  '1110001',
    '-M':  '1110011',
    'M+1': '1110111',
    'M-1': '1110010',
    'D+M': '1000010',
    'D-M': '1010011',
    'M-D': '1000111',
    'D&M': '1000000',
    'D|M': '1010101'
}

JUMP = {
    'null' : '000',
    'JGT' : '001',
    'JEQ' : '010',
    'JGE' : '011',
    'JLT' : '100',
    'JNE' : '101',
    'JLE' : '110',
    'JMP' : '111'
}

class Assembler(object):

    def __init__(self):
        #PRE DEFINED REGISTERS
        self.__pre_define_sym_table = {
            'R0': 0, 'R1': 1, 'R2': 2, 'R3': 3, 'R4': 4, 'R5': 5,
            'R6': 6, 'R7': 7, 'R8': 8, 'R9': 9, 'R10': 10, 'R11': 11,
```

```
            'R12': 12, 'R13': 13, 'R14': 14, 'R15': 15,
            'SP': 0, 'LCL': 1, 'ARG': 2, 'THIS': 3, 'THAT': 4,
            'SCREEN': 0x4000, 'KBD': 0x6000}
        self.define()
    #FINDING THE BINARY OF THE GIVEN DECIMAL VALUE
    def binary(self, value, padding=0):
        return bin(value)[2:].rjust(padding, '0')

    def define(self):
        self.sym_count = 16
        self.sym_table = self.__pre_define_sym_table
        self.label_table = {}
        self.instructions = []
        self.machine_code = []

    #READING OUR INPUT FILE0
    def read(self, path):
        self.define()
        with open(path) as x:
            self.instructions = x.readlines()
        for index, instruction in enumerate(self.instructions):
            self.instructions[index]                                    =
instruction.strip('\n').replace(' ', '').split('//')[0]

        # Remove empty line
        self.instructions = list(filter(None, self.instructions))

    def parse(self):
        self.machine_code = []
        # Truly parsing
        for pc, instruction in enumerate(self.instructions):
            if not instruction:
                continue
```

```
            if instruction.startswith('@'):
                # A-Instruction
                value = instruction[1:]
                value = int(value)
                value = self.binary(value, 15)
                self.machine_code.append('0' + value)

            elif instruction.startswith('(') and
instruction.endswith(')'):
                # Label
                pass

            else:
                # C-Instruction
                if '=' in instruction:
                    dest, comp = instruction.split('=')
                    dest = DEST[dest]
                    comp = COMP[comp]
                    machine = '111' + comp + dest + '000'
                # JUMP
                elif ';' in instruction:
                    comp, jump = instruction.split(';')
                    comp = COMP[comp]
                    jump = JUMP[jump]
                    machine = '111' + comp + '000' + jump
                self.machine_code.append(machine)


asm = Assembler()
asm.read('ADD@2.asm')
asm.parse()
output={}
```

```
for i in range(len(asm.instructions)):
    output[asm.instructions[i]]= asm.machine_code[i]
print(output)
```

Java code:-

```
package assembler;


import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintStream;
import java.util.ArrayList;


public class Assembler {


      String filename;// file name
      String Line[];// count line
      int count = 0;// intially zero
      int v = 1, l = 1, ii = 0;// parameters to store variables and
loops
      ArrayList<String>  v1  =  new  ArrayList<String>(26568);//  list
containing label name
      ArrayList<Integer> p1 = new ArrayList<Integer>(26568);// list
containing the labels position
      ArrayList<String>  l1  =  new  ArrayList<String>(26568);//  list
containing variables.


      Assembler(String s) {// constructor to initialise file name
            filename = s;
            Line = new String[60];
      }
```

```
	public void read() throws IOException {// read each line in asm
code
		BufferedReader	br	=	new	BufferedReader(new
FileReader(filename));

		String nextLine = br.readLine();

		int i = 0;
		while (nextLine != null) {// until next line is null
			// System.out.println(nextLine);
			Line[i] = nextLine;

			nextLine = br.readLine();

			i++;
		}
		br.close();
		// System.out.println(count + " " + i);
		count = i;// to store the number of lines.
	}

	public void assmble() {// decide whether c or a instruction

		for (ii = 0; ii < count; ii++) {
			String s = Line[ii];
			if (s.charAt(s.length() - 1) == ')') {
				// System.out.println(ii);
				l1.add(s.substring(1, s.length() - 1));
				p1.add(ii);
				System.out.println(l1 + " " + p1);
			}
		}
```

```
            for (ii = 0; ii < count; ii++) {

                  if (Line[ii].charAt(0) == '@' || Line[ii].charAt(0)
== '(') {
                        aIns(Line[ii]);

                  } else if (Line[ii].charAt(0) != '(')
                        cIns(Line[ii]);

            }

      }

      public void aIns(String s) {// if a instruction

            s = s.substring(1);
            s = trim(s);

            if (Character.isDigit(s.charAt(0))) {
                  int n = Integer.parseInt(s);

                  s              =              String.format("%016d",
Integer.parseInt(Integer.toBinaryString(n)));
                  System.out.println(s);
            } else if (s.charAt(0) == 'R') {// R[]
                  s = s.substring(1);
                  int n = Integer.parseInt(s);

                  s              =              String.format("%016d",
Integer.parseInt(Integer.toBinaryString(n)));
            } else {

                  // System.out.println(s);
                  switch (s) {
```

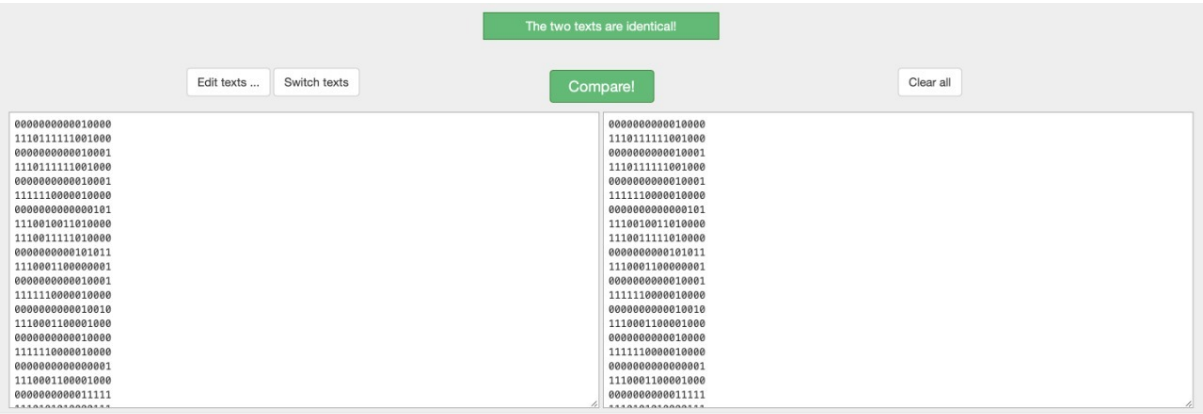
```
                case "SCREEN":
                      s = String.format("%016d", 16384);
                      break;
                case "KBD":
                      s = String.format("%016d", 24576);
                      break;
                case "SP":
                      s = String.format("%016d", 0);
                      break;
                case "LCL":
                      s = String.format("%016d", 1);
                      break;
                case "ARG":
                      s = String.format("%016d", 2);
                      break;
                case "THIS":
                      s = String.format("%016d", 3);
                      break;
                case "THAT":
                      s = String.format("%016d", 4);
                      break;

                default:
                      if (s.charAt(s.length() - 1) == ')') {

                      }
                      // System.out.println(s);
                      else {

                            if (l1.contains(s)) {

                                  s     =     String.format("%016d",
Integer.parseInt(Integer.toBinaryString(p1.get(l1.indexOf(s)))));
```

```
                        // System.out.println((s));
                    }

                    else if (v1.contains(s))

                    {

                        s = String.format("%016d",
Integer.parseInt(Integer.toBinaryString(v1.indexOf(s) + 16)));
                        System.out.println(s);
                    }

                    else

                    {
                        v1.add(s);
                        // System.out.print(s + " ");
                        s = String.format("%016d",
Integer.parseInt(Integer.toBinaryString(v1.indexOf(s) + 16)));
                        System.out.println(s);
                    }
                    // System.out.println(l1 );
                    // System.out.println(p1);
                    // System.out.println(v1 );

                }
            }
        }
    }

    public String trim(String s) {// to trim the string
        return s.trim();
    }
```

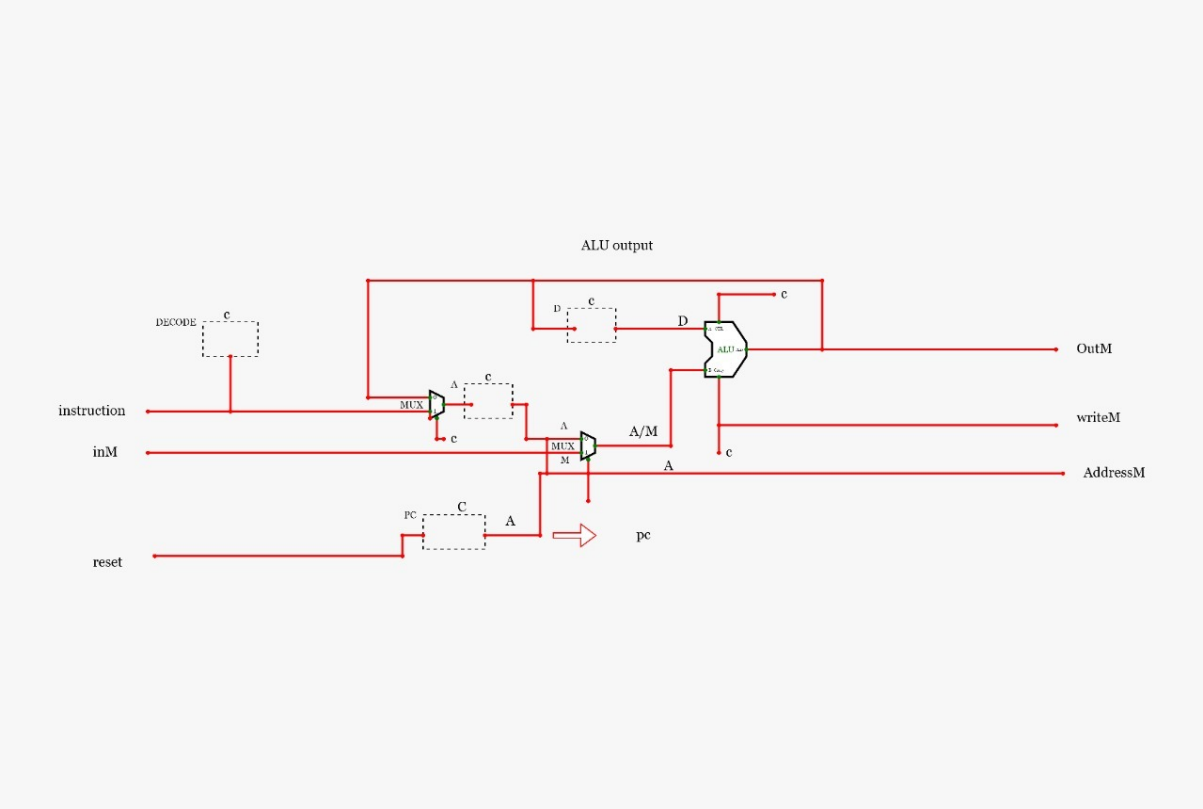
```
     public void cIns(String s) {// if c instruction
          String dest[] = { "0", "M", "D", "MD", "A", "AM", "AD",
"AMD" };
          String jump[] = { "0", "JGT", "JEQ", "JGE", "JLT", "JNE",
"JLE", "JMP" };
          String comp1[] = { "0", "1", "-1", "D", "A", "!D", "!A",
"-D", "-A", "D+1", "A+1", "D-1", "A-1", "D+A", "D-A",
                    "A-D", "D&A", "D|A" };
          String comp2[] = { "0", "1", "-1", "D", "M", "!D", "!M",
"-D", "-M", "D+1", "M+1", "D-1", "M-1", "D+M", "D-M",
                    "M-D", "D&M", "D|M" };
          // String comp2[] = { "", "", "", "", "M", "", "!M", "",
"-M", "", "M+1", "",
          // "M-1", "D+M", "D-M", "M-D", "D&M",
          // "D|M" };
          String comp3[] = { "101010", "111111", "111010", "001100",
"110000", "001101", "110001", "001111", "110011",
                    "011111",   "110111",   "001110",   "110010",
"000010", "010011", "000111", "000000", "010101" };

          String d1 = "0", j1 = "0", c1 = "0", a = "0";
          // String d2 = "", j2 = "", c2 = "";
          if (s.contains("=")) {
               d1 = s.substring(0, s.indexOf("="));
               c1 = s.substring(s.indexOf("=") + 1);
               if (c1.contains("M"))
                    a = "1";

               // System.out.println(d1 + " " + c1 + "<>" + s);
          } else if (s.contains(";")) {
               c1 = s.substring(0, s.indexOf(";"));
               j1 = s.substring(s.indexOf(";") + 1);
               // System.out.println(c1 + " " + j1 + "<>" + s);
```

```
            }

            d1 = Integer.toBinaryString(findEle(d1, dest));
            d1   =   String.format("%03d",   Integer.parseInt(d1));//
destination

            j1 = Integer.toBinaryString(findEle(j1, jump));

            j1 = String.format("%03d", Integer.parseInt(j1));// jump

            if (c1.contains("A"))// compute instruction

                  c1 = comp3[findEle(c1, comp1)];

            else {

                  c1 = comp3[findEle(c1, comp2)];
            }

            // System.out.println("111" + "A-" + a + "C" + c1 + "D" +
d1 + "J" + j1 + " for
            // " + s);
            System.out.println("111" + a + c1 + d1 + j1);
      }

      public int findEle(String d1, String a[]) {// to find the element
            int flag = 0;
            for (int i = 0; i < a.length; i++) {
                  if (a[i].equals(d1))
                        flag = i;
            }
            return flag;
      }
```

```
    public void write() throws FileNotFoundException {// to write the answer in a new file
        PrintStream ps = new PrintStream(new FileOutputStream("machineCode.txt"));

        ps.println();
        ps.close();
    }

    public static void main(String[] args) throws IOException {// main function to execute the follow
        Assembler obj = new Assembler("array.txt");// pa

        obj.write();
        obj.read();
        obj.assmble();
    }

}
```



ARCHITECTURE:-



CPU is going to be composed of two main ingredients two main components one of them is well you often called the arithmetic logic unit it's actually a piece of hardware that actually is able to add numbers subtract numbers maybe do logical operations .so on the second element is there are going to be a bunch of registers a bunch of places where we can store a data that we're going to use for the rest of the computation.

The CPU is built of the memory itself as we said has two parts the part that stores the program in the start that probe stores the data to try to understand how all these things work together it's best to actually consider is a flow of data what kind of information needs to pass within the computers.

so basically there are I would say three types of information that usually pass throughout the system and one of them is a data when we have numbers that need to be added the numbers needs to

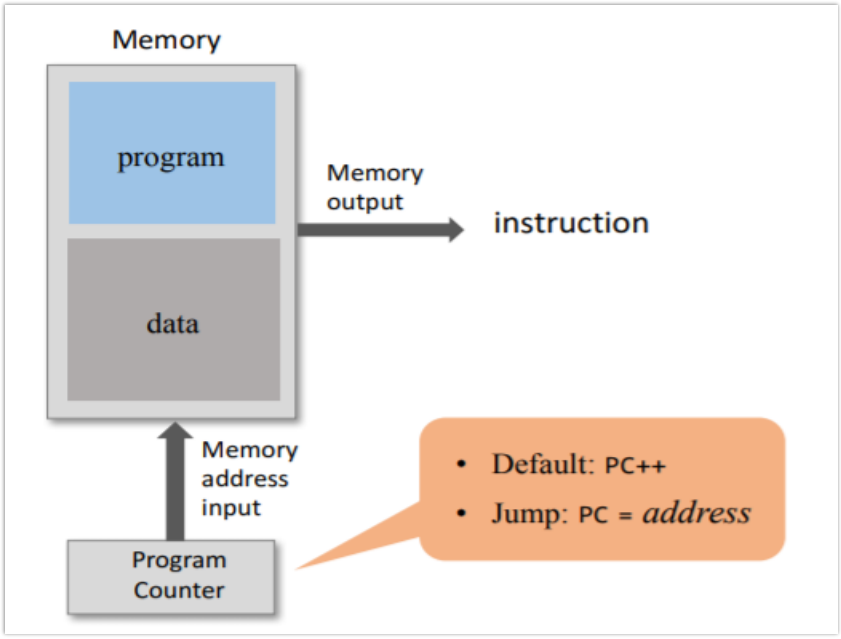
be moved from somewhat from one place to another from the data a memory to the registers to the actually arithmetic logic unit is going to do something with them.

The second type of information that we need to control is what's called addresses what instruction are we actually executing now what piece of data within the memory do we need to access now these are in addresses. And of course there's going to be a very going to need to be a big bunch of wires that actually do all the control that actually tell each part of the system what to do at this particular point and this is called the control.

Basic CPU loop Repeat:-

•Fetch an instruction from the program memory
• Execute the instruction.

Fetching



Put the location of the next instruction in the Memory address input

Get the instruction code by reading the contents at that Memory location

Executing

• The instruction code specifies “what to do”

Which arithmetic or logical instruction to execute

Which memory address to access (for read / write)

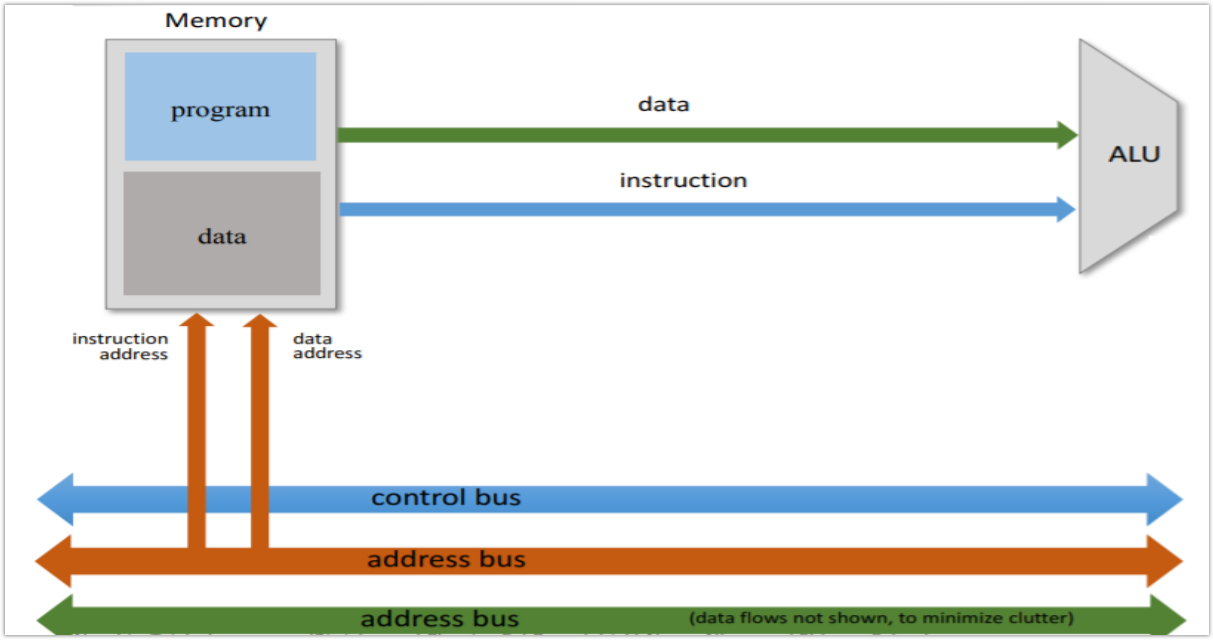
If / where to jump q …

• Executing the instruction involves:

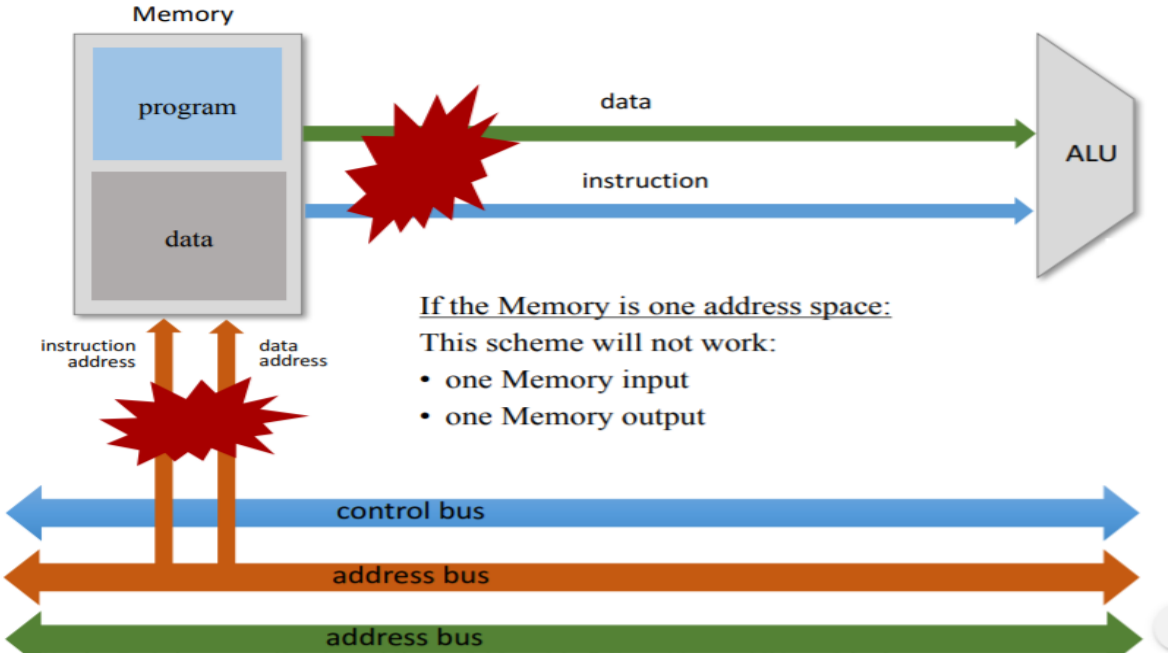
accessing registers

and / or:

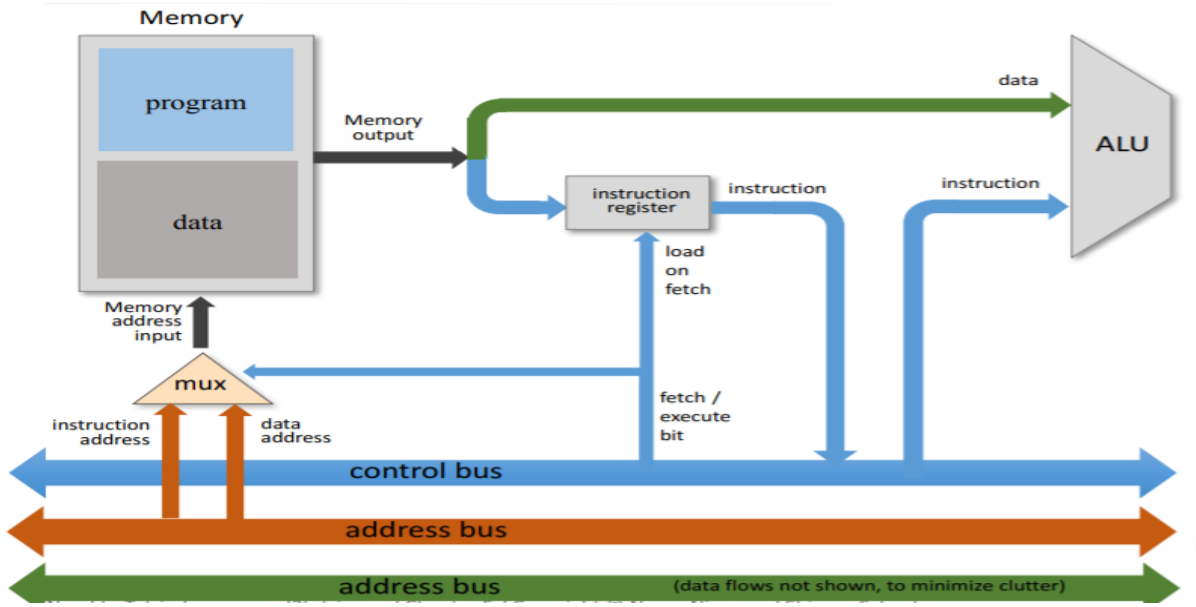
accessing the data memory.



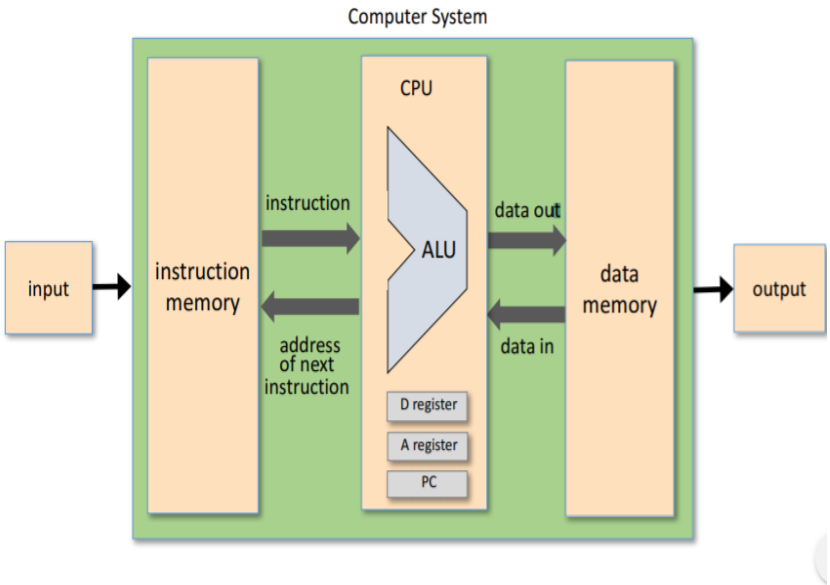
Fetch & Execute clash



Solution: multiplex, using an instruction register

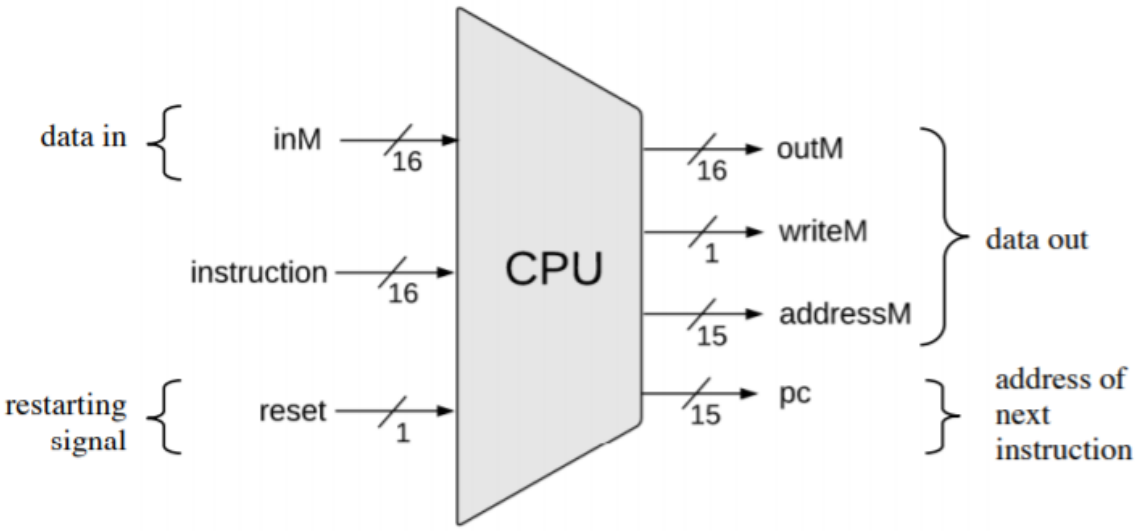


Hack computer



CPU is the center piece of every architecture. All calculations takes place in the here and about which instruction should be fetched and executed. It is a 16 bit processer. It is connected to both instruction memory and data memory.

Hack CPU Interface

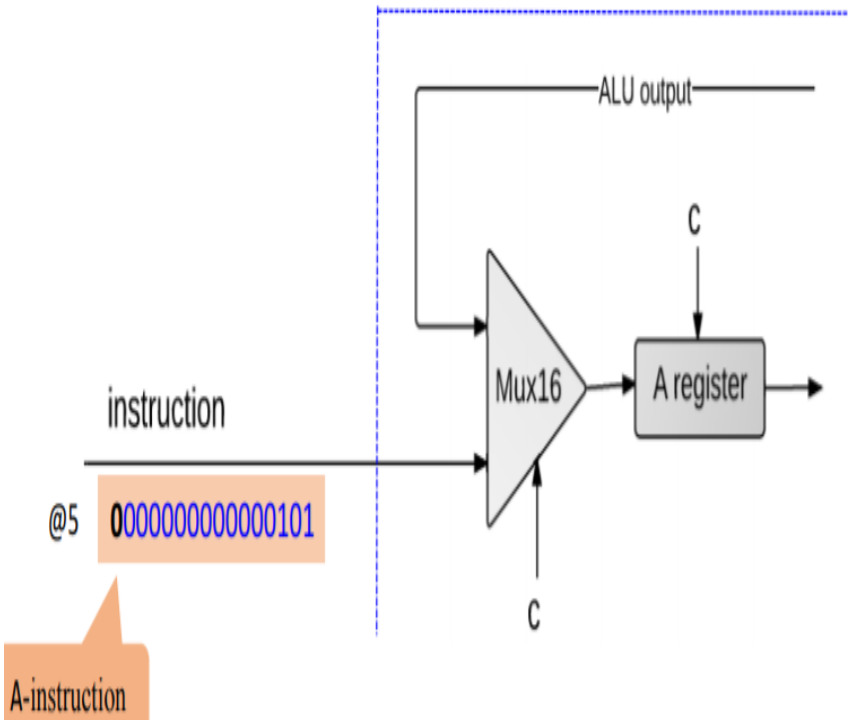


Data registers: These registers give the CPU short-term memory services. For example, when calculating the value of (a - b) · c, we must first compute and remember the value of (a - b). Although this result can be temporarily stored in some memory location, a better solution is to store it locally inside the CPU—in a data register.

Addressing registers: The CPU has to continuously access the memory in order to read data and write data. In every one of these operations, we must specify which individual memory word has to be accessed, namely, supply an address. In some cases this address appears as part of the current instruction, while in others it depends on the execution of a previous instruction. In the latter case, the address should be stored in a register whose contents can be later treated as a memory address—an addressing register.

Program counter register: When executing a program, the CPU must always keep track of the address of the next instruction that must be fetched from the instruction memory. This address is kept in a special register called program counter, or PC. The contents of the PC are then used as the address for fetching instructions from the instruction memory. Thus, in the process of executing the current instruction, the CPU updates the PC in one of two ways. If the current instruction contains no goto directive, the PC is incremented to point to the next instruction in the program. If the current instruction includes a goto n directive that should be executed, the CPU loads n into the PC.

CPU operation: handling A-instructions



CPU handling of an A-instruction:

• Decodes the instruction into:

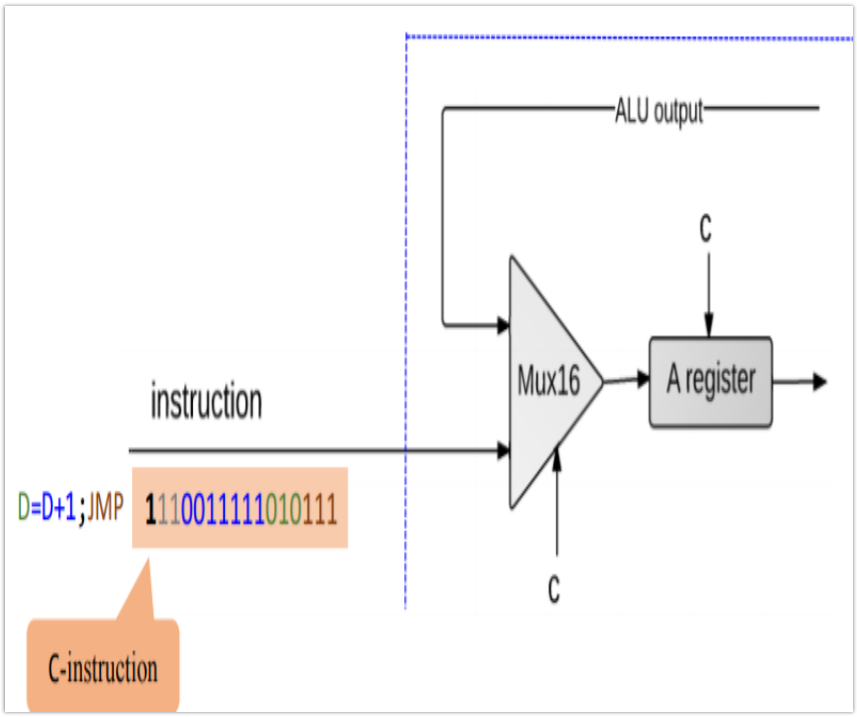
op-code

15-bit value

• Stores the value in the A-register

• Outputs the value (not shown in this diagram).

CPU operation: handling C-instructions



CPU handling of a C-instruction:

- Decodes the instruction bits into:

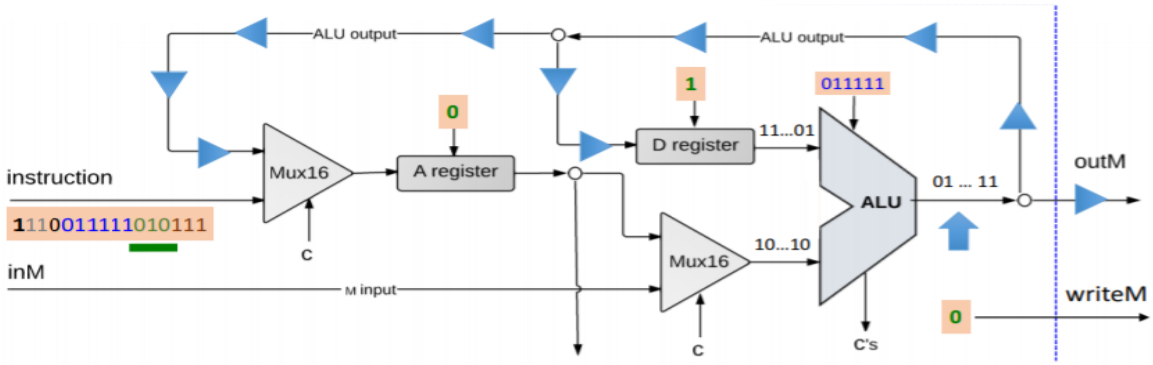
  Op-code

  ALU control bits

  Destination load bits

  Jump bits

- Routes these bits to their chip-part destinations
- The chip-parts (most notably, the ALU) execute the instruction.



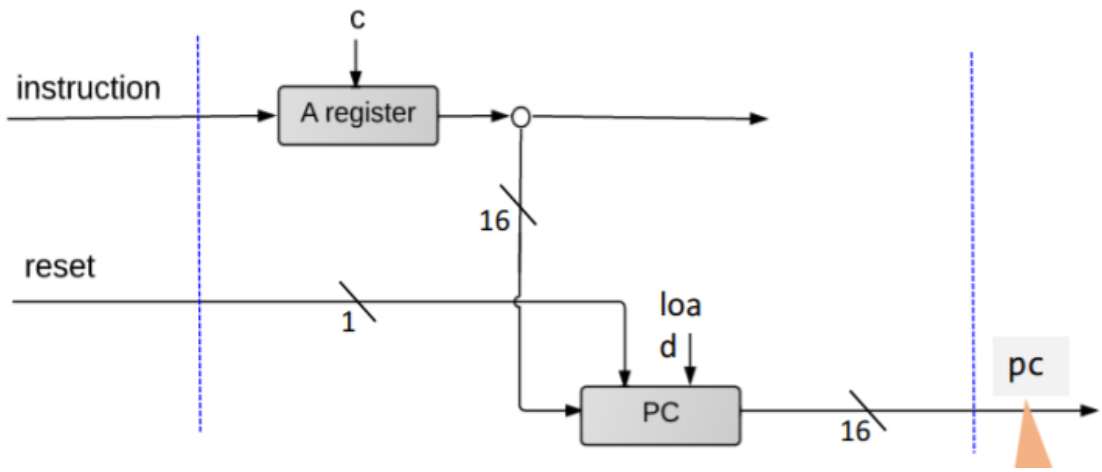
ALU data output:

• Result of ALU calculation

• Fed simultaneously to: D-register, A-register, data memory 01 … 11 0 1 0

• Which destination actually commits to the ALU output is determined by the instruction's destination bits.

CPU operation: control



PC operation (abstraction)

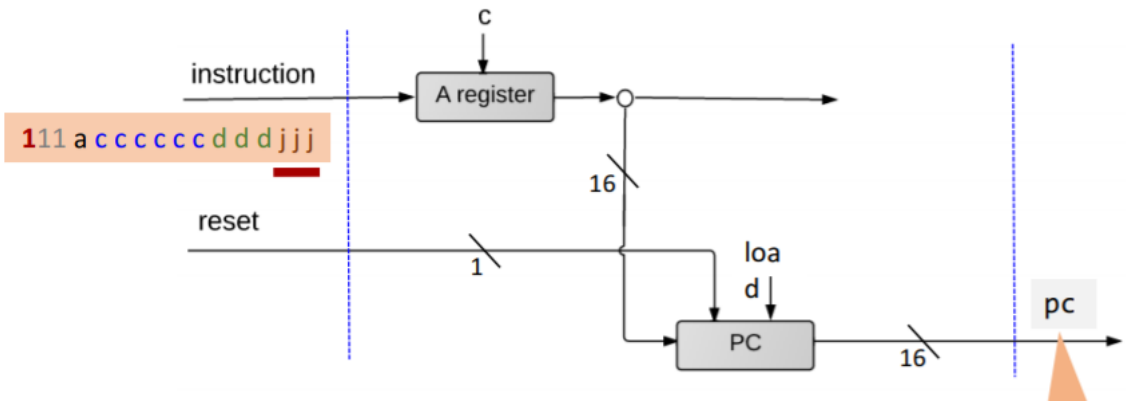
Emits the address of the next instruction:

restart: PC = 0

no jump: PC++

goto: PC=A

conditional goto: if (condition) PC=A else PC++



PC operation (implementation)

if (reset==1) PC=0

else

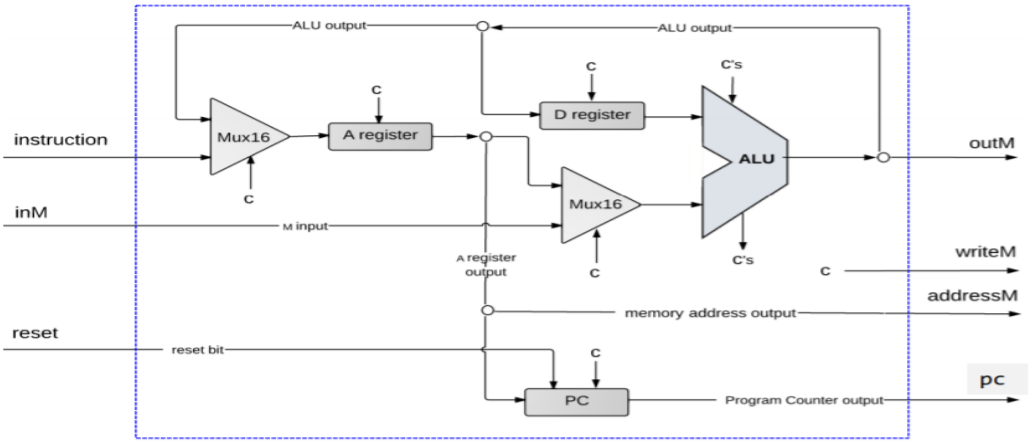
// in the course of handling the current instruction:

load = f (jump bits, ALU control outputs)

if (load == 1) PC=A // jump

else PC++ // next instruction

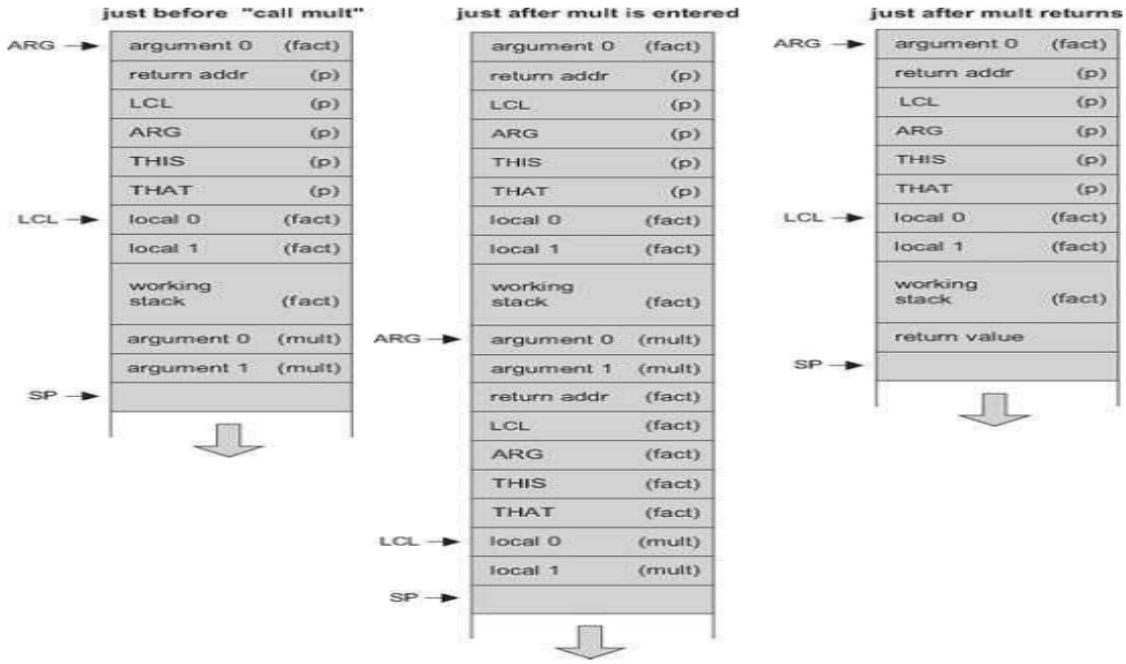
Hack CPU Implementation



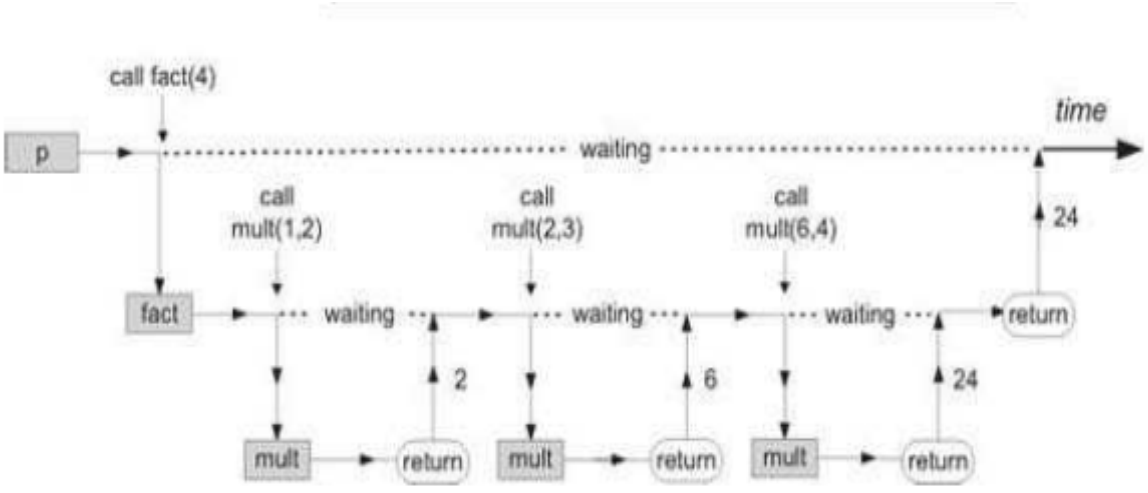
Reset:-

Pushing reset causes the program to start running.

VM Implementation



Execution



The factorial of a positive number n can be computed by the iterative formula n! = 1 ·2·... ·(n - 1) · n. This algorithm is implemented in the above figure.

Let us focus on the call mult command highlighted in the fact function code from above figure. The above figure shows three stack states related to this call, illustrating the function calling protocol in action.

If we ignore the middle stack instance in the above figure, we observe that fact has set up some arguments and called mult to operate on them (left stack instance).

When mult returns (right stack instance), the arguments of the called function have been replaced with the function's return value. In other words, when the dust clears from the function call, the calling function has received the service that it has requested, and processing resumes as if nothing happened: The drama of mult's processing (middle stack instance) has left no trace whatsoever on the stack, except for the return value.

The life cycle of function calls. An arbitrary function p calls function fact, which then calls mult several times. Vertical arrows depict transfer of control from one function to another. At any given point in time, only one function is running, while all the functions up the calling chain are waiting for it to return. When a function returns, the function that called it resumes its execution.

Global stack dynamics corresponding to above figure, focusing on the call mult event. The pointers SP, ARG, and LCL are not part of the VM abstraction and are used by the VM implementation to map the stack on the host RAM.

MULTIPLICATION VM CODE:-

```
function mult 2     //2 local variables
    push constant 0 //result=0
    pop local 0
    push argument 1 //j=y
    pop local 1
label loop
    push constant 0 //if j==0 goto end
    push local 1
    eq
    if-goto end
    push local 0    //resut=result+x
    push argument 0
    add
    pop local 0
    push local 1    //j=j-1
    push constant 1
    sub
    pop local 1
    goto loop
label end
    push local 0    //return result
    return
```

FACTORIAL VM CODE:-

```
function fact 2          //2 local variables
    //Returns the factorial of a given argument
    push constant 1
    pop local 0          //result=1
    push constant 1
    pop local 1          //j=1
label loop
    push constant 1
    push local 1
    add
    pop local 1          //j=j+1
    push local 1
    push argument 0
    gt
    if-goto end          //if j>n goto end
    push local 0
    push local 1
    call mult 2          //2 arguments were pushed
    pop local 0          //result=mult(result,j)
    goto loop
label end
    push local 0
    return
```

FACTORIAL:- (JACK)

```
//factorial computation
class Compute
{
function int factorial(int n)
{
if (n = 0)
{
return 1;
}
else
{
 return n * Math.factorial(n - 1);
}
}
```

CHIPS:-

ALU:-

```
CHIP ALU {
   IN
      x[16], y[16],  // 16-bit inputs
      zx, // zero the x input
      nx, // negate the x input
      zy, // zero the y input
      ny, // negate the y input
      f,  // compute  out = x + y (if 1) or out = x & y (if 0)
      no; // negate the out output
  OUT
      out[16], // 16-bit output
      zr, // 1 if (out==0), 0 otherwise
      ng; // 1 if (out<0),  0 otherwise
```

```
    PARTS:

    // if (zx==1) set x = 0
    Mux16(a=x,b=false,sel=zx,out=zxout);

    // if (zy==1) set y = 0
    Mux16(a=y,b=false,sel=zy,out=zyout);

    // if (nx==1) set x = ~x
    // if (ny==1) set y = ~y
    Not16(in=zxout,out=notx);
    Not16(in=zyout,out=noty);
    Mux16(a=zxout,b=notx,sel=nx,out=nxout);
    Mux16(a=zyout,b=noty,sel=ny,out=nyout);
    // if (f==1)  set out = x + y
    // if (f==0)  set out = x & y
    Add16(a=nxout,b=nyout,out=addout);
    And16(a=nxout,b=nyout,out=andout);
    Mux16(a=andout,b=addout,sel=f,out=fout);

    // if (no==1) set out = ~out
    // 1 if (out<0),  0 otherwise
    Not16(in=fout,out=nfout);
    Mux16(a=fout,b=nfout,sel=no,out=out,out[0..7]=zr1,out[8..15]=zr2,out[15]=ng);
    // 1 if (out==0), 0 otherwise
    Or8Way(in=zr1,out=or1);
    Or8Way(in=zr2,out=or2);
    Or(a=or1,b=or2,out=or3);
    Not(in=or3,out=zr);
}
```

AND:-

```
CHIP And {
    IN a, b;
    OUT out;

    PARTS:
    Nand(a=a,b=b,out=nandout);
    Not(in=nandout,out=out);
}
```

16 bit AND :-

```
CHIP And16 {
    IN a[16], b[16];
    OUT out[16];

    PARTS:
    And(a=a[0],b=b[0],out=out[0]);
    And(a=a[1],b=b[1],out=out[1]);
    And(a=a[2],b=b[2],out=out[2]);
    And(a=a[3],b=b[3],out=out[3]);
    And(a=a[4],b=b[4],out=out[4]);
    And(a=a[5],b=b[5],out=out[5]);
    And(a=a[6],b=b[6],out=out[6]);
    And(a=a[7],b=b[7],out=out[7]);
    And(a=a[8],b=b[8],out=out[8]);
    And(a=a[9],b=b[9],out=out[9]);
    And(a=a[10],b=b[10],out=out[10]);
    And(a=a[11],b=b[11],out=out[11]);
    And(a=a[12],b=b[12],out=out[12]);
    And(a=a[13],b=b[13],out=out[13]);
    And(a=a[14],b=b[14],out=out[14]);
```

```
    And(a=a[15],b=b[15],out=out[15]);
}
```

FULL ADDER:-

```
CHIP FullAdder {
    IN a, b, c;  // 1-bit inputs
    OUT sum,     // Right bit of a + b + c
        carry;   // Left bit of a + b + c
PARTS:
    HalfAdder(a=a,b=b,sum=sum1,carry=carry1);
    HalfAdder(a=c,b=sum1,sum=sum,carry=carry2);
    Or(a=carry1,b=carry2,out=carry);
}
```

HALF ADDER:-

```
CHIP HalfAdder {
    IN a, b;    // 1-bit inputs
    OUT sum,    // Right bit of a + b
        carry;  // Left bit of a + b

    PARTS:
    And(a=a,b=b,out=carry);
    Xor(a=a,b=b,out=sum);
}
```

MUX:-

```
CHIP Mux {
    IN a, b, sel;
    OUT out;

    PARTS:
    Not(in=sel,out=notsel);
```

```
    And(a=a,b=notsel,out=anotsel);
    And(a=b,b=sel,out=bsel);
    Or(a=anotsel,b=bsel,out=out);
}
```

NOT:-

```
CHIP Not {
    IN in;
    OUT out;

    PARTS:
    Nand(a=in,b=in,out=out);
}
```

16 bit NOT:-

```
CHIP Not16 {
    IN in[16];
    OUT out[16];

    PARTS:
    Not(in=in[0],out=out[0]);
    Not(in=in[1],out=out[1]);
    Not(in=in[2],out=out[2]);
    Not(in=in[3],out=out[3]);
    Not(in=in[4],out=out[4]);
    Not(in=in[5],out=out[5]);
    Not(in=in[6],out=out[6]);
    Not(in=in[7],out=out[7]);
    Not(in=in[8],out=out[8]);
    Not(in=in[9],out=out[9]);
    Not(in=in[10],out=out[10]);
    Not(in=in[11],out=out[11]);
```

```
    Not(in=in[12],out=out[12]);
    Not(in=in[13],out=out[13]);
    Not(in=in[14],out=out[14]);
    Not(in=in[15],out=out[15]);
}
```

OR:-

```
CHIP Or {
    IN a, b;
    OUT out;




  PARTS:
    Not(in=a,out=nota);
    Not(in=b,out=notb);
    Nand(a=nota,b=notb,out=out);
}
```

16 bit OR:-

```
CHIP Or16 {
    IN a[16], b[16];
    OUT out[16];

    PARTS:
    Or(a=a[0],b=b[0],out=out[0]);
    Or(a=a[1],b=b[1],out=out[1]);
    Or(a=a[2],b=b[2],out=out[2]);
    Or(a=a[3],b=b[3],out=out[3]);
    Or(a=a[4],b=b[4],out=out[4]);
    Or(a=a[5],b=b[5],out=out[5]);
```

```
    Or(a=a[6],b=b[6],out=out[6]);
    Or(a=a[7],b=b[7],out=out[7]);
    Or(a=a[8],b=b[8],out=out[8]);
    Or(a=a[9],b=b[9],out=out[9]);
    Or(a=a[10],b=b[10],out=out[10]);
    Or(a=a[11],b=b[11],out=out[11]);
    Or(a=a[12],b=b[12],out=out[12]);
    Or(a=a[13],b=b[13],out=out[13]);
    Or(a=a[14],b=b[14],out=out[14]);
    Or(a=a[15],b=b[15],out=out[15]);
}
```

PC:-

```
CHIP PC {
    IN in[16],load,inc,reset;
    OUT out[16];

    PARTS:
    Inc16(in=out5,out=out1);
    Mux16(a=out5,b=out1,sel=inc,out=out2);
    Mux16(a=out2,b=in,sel=load,out=out3);
    Mux16(a=out3,b=false,sel=reset,out=out4);
    Register(in=out4,load=true,out=out5,out=out);

}
```

RAM 512:-

```
CHIP RAM512 {
    IN in[16], load, address[9];
    OUT out[16];
```

```
   PARTS:
DMux8Way(in=load,sel=address[6..8],a=load0,b=load1,c=load2,d=load3,e=load4,f=load5,g=load6,h=
load7);

   RAM64(in=in,load=load0,address=address[0..5],out=out0);

   RAM64(in=in,load=load1,address=address[0..5],out=out1);

   RAM64(in=in,load=load2,address=address[0..5],out=out2);

   RAM64(in=in,load=load3,address=address[0..5],out=out3);

   RAM64(in=in,load=load4,address=address[0..5],out=out4);

   RAM64(in=in,load=load5,address=address[0..5],out=out5);

   RAM64(in=in,load=load6,address=address[0..5],out=out6);

   RAM64(in=in,load=load7,address=address[0..5],out=out7);

Mux8Way16(a=out0,b=out1,c=out2,d=out3,e=out4,f=out5,g=out6,h=out7,sel=address[6..8],out=out
);

}
```

REGISTER:-

```
CHIP Register {

   IN in[16], load;

   OUT out[16];


   PARTS:

   Bit(in=in[0],load=load,out=out[0]);

   Bit(in=in[1],load=load,out=out[1]);

   Bit(in=in[2],load=load,out=out[2]);

   Bit(in=in[3],load=load,out=out[3]);

   Bit(in=in[4],load=load,out=out[4]);

   Bit(in=in[5],load=load,out=out[5]);

   Bit(in=in[6],load=load,out=out[6]);

   Bit(in=in[7],load=load,out=out[7]);

   Bit(in=in[8],load=load,out=out[8]);

   Bit(in=in[9],load=load,out=out[9]);

   Bit(in=in[10],load=load,out=out[10]);

   Bit(in=in[11],load=load,out=out[11]);
```

```
    Bit(in=in[12],load=load,out=out[12]);
    Bit(in=in[13],load=load,out=out[13]);
    Bit(in=in[14],load=load,out=out[14]);
    Bit(in=in[15],load=load,out=out[15]);
}
```

MUX4WAY16:-

```
CHIP Mux4Way16 {
    IN a[16], b[16], c[16], d[16], sel[2];
    OUT out[16];

    PARTS:
    Mux16(a=a,b=b,sel=sel[0],out=out1);
    Mux16(a=c,b=d,sel=sel[0],out=out2);
    Mux16(a=out1,b=out2,sel=sel[1],out=out);
}
```

COMPUTER:-

```
CHIP Computer {

    IN reset;

    PARTS:

CPU(instruction=instruction,reset=reset,inM=outMemo,outM=CPUoutM,writeM=wM,addressM=adM,pc=PC);
    Memory(in=CPUoutM,load=wM,address=adM,out=outMemo);
    ROM32K(address=PC,out=instruction);
}
```

CPU:-

```
CHIP CPU {
```

```
IN  inM[16],         // M value input  (M = contents of RAM[A])
    instruction[16], // Instruction for execution
    reset;           // Signals whether to re-start the current
                     // program (reset=1) or continue executing
                     // the current program (reset=0).

OUT outM[16],        // M value output
    writeM,          // Write into M
    addressM[15],    // Address in data memory of M
    pc[15];          // address of next instruction

PARTS:
Mux16(a=instruction,b=ALUout,sel=instruction[15],out=Ain);

Not(in=instruction[15],out=notinstruction);

//RegisterA
//when instruction[15] = 0, it is @value means A should load value
Or(a=notinstruction,b=instruction[5],out=loadA);//d1
ARegister(in=Ain,load=loadA,out=Aout,out[0..14]=addressM);

Mux16(a=Aout,b=inM,sel=instruction[12],out=AMout);

//Prepare for ALU, if it is not an instruction, just return D
And(a=instruction[11],b=instruction[15],out=zx);// zx to zero the x input
And(a=instruction[10],b=instruction[15],out=nx);// nx negate the x input
Or(a=instruction[9],b=notinstruction,out=zy);//zy to zero the y input
Or(a=instruction[8],b=notinstruction,out=ny);//ny to negate the y input
And(a=instruction[7],b=instruction[15],out=f);//f:1 for Add ,0 for And
And(a=instruction[6],b=instruction[15],out=no);//no to negate the output
```

```
    //Feed the computed zx,nx,zy,ny,f,no to the ALU
    ALU(x=Dout,y=AMout,zx=zx,nx=nx,zy=zy,ny=ny,f=f,no=no,out=outM,out=ALUout,zr=zero,ng=neg);

    //when it is an instruction, write M
    And(a=instruction[15],b=instruction[3],out=writeM);//d3

    //RegisterD,when it is an instruction, load D
    And(a=instruction[15],b=instruction[4],out=loadD);//d2
    DRegister(in=ALUout,load=loadD,out=Dout);

    //Prepare for jump
    //get positive
    Or(a=zero,b=neg,out=notpos);
    Not(in=notpos,out=pos);

    And(a=instruction[0],b=pos,out=j3);//j3
    And(a=instruction[1],b=zero,out=j2);//j2
    And(a=instruction[2],b=neg,out=j1);//j1

    Or(a=j1,b=j2,out=j12);
    Or(a=j12,b=j3,out=j123);

    And(a=j123,b=instruction[15],out=jump);

    //when jump,load Aout
    PC(in=Aout,load=jump,reset=reset,inc=true,out[0..14]=pc);
}
```