International Institute of Information Technology
Bangalore

COMPUTX

Nand2Tetris

Name of the student: Aakash Khot
(IMT2020512)

May 18, 2022

Introduction

This report will include the project 4,5 and 6 of nand2tetris.
All the codes of these projects is on the given **Github link:-**https://github.com/aakash2khot/Nand2Tetris
I have taken the codes of project 1,2 and 3 from the given github link as suggested:-https://github.com/ComputX-research-group/Nand2Tetris

Project 4:-

0.1 Code for Fill.asm:-

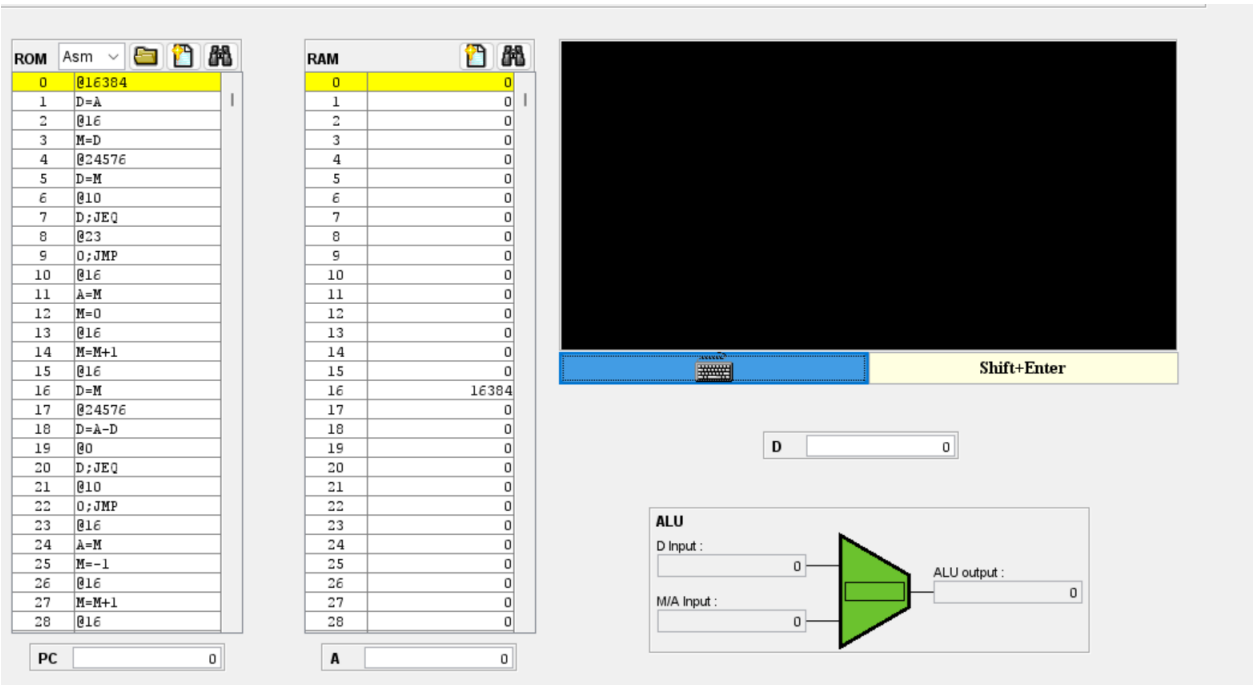
```
(START)
@SCREEN
D=A
@st
M=D // st-> screen
@KBD
D=M // input from keyboard
@WHITE
D;JEQ // if no input
      //then white
@BLACK
0;JMP  //else black

(WHITE)
@st
A=M
M=0 //starting from 0
@st
M=M+1
@st
D=M
@KBD
D=A-D
@START
D;JEQ //equal to 0 -> end
@WHITE
0;JMP //else continue

(BLACK)
@st
A=M
M=-1
@st
M=M+1
@st
D=M
@KBD
D=A-D
@START
D;JEQ //equal to 0->end
@BLACK
0;JMP //else continue
```

Fill.asm runs on an infinite loop that listens to keyboard. What it exactly does is, if a key is pressed (any key), the screen area will get black. So it takes each pixel of screen and blackens it out.

The screen is full black as long as the key is pressed, the moment you take off your hand from key, the screen will again change to its initial color that is white, so again colors is white every pixel.

As from code, START is the loop running on, KBD is to see the input from keyboard, WHITE function is to whiten the screen and similarly BLACK function to blacken the screen.

The figure given below is a screenshot taken when the key is pressed, and you will observe that when no key is pressed, the screen will be white.



0.2 Code for Mult.asm:-

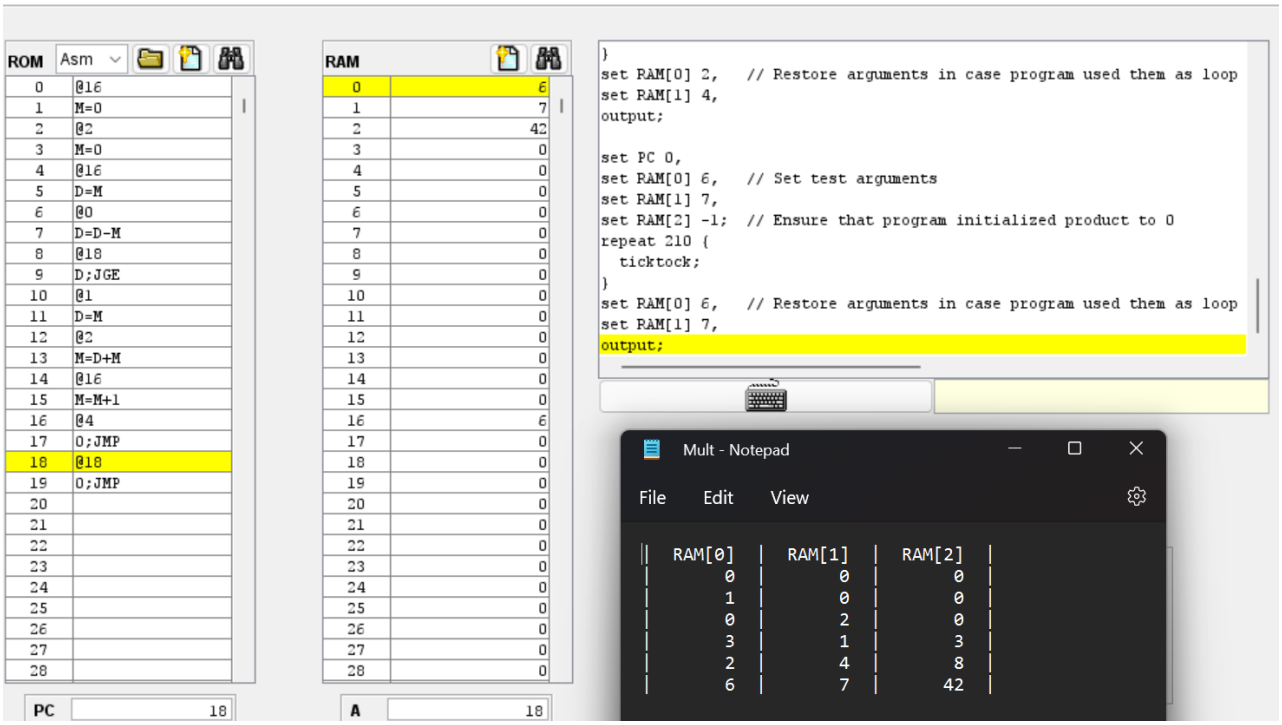
```
// i=0
@i
M=0
// RAM[2]=0
@2
M=0

(ADD)
@i
D=M // D reg -> i
@0
D=D-M // D reg -> i-RAM[0]
@END
D;JGE // Jump if D>=0 to END

@1
D=M // D-> RAM[1]
@2
M=M+D// R2-> R2+R1
@i
M=M+1
@ADD
0;JMP

(END)
@END
0;JMP// If END is called,
//calculation is over so you can stop
```

Mult.asm program can be used to multiply two numbers. So it first we will put any two values in register R0 and R1 i.e. RAM[0] and RAM[1]. This program will multiple the two numbers in present in these register and store it in R2 which is RAM[2].

Conditions: R0 >= 0, R1 >= 0, and R0*R1 < 32768.

The ADD function will be used to add the two numbers. So what we will do is, lets say one number is x and other is y, then will will be adding x, y times so that the result is multiplication of both.

This given figure shows the running of test file and getting the output file correct.



Project 5:-

0.3 Code for CPU.hdl:-

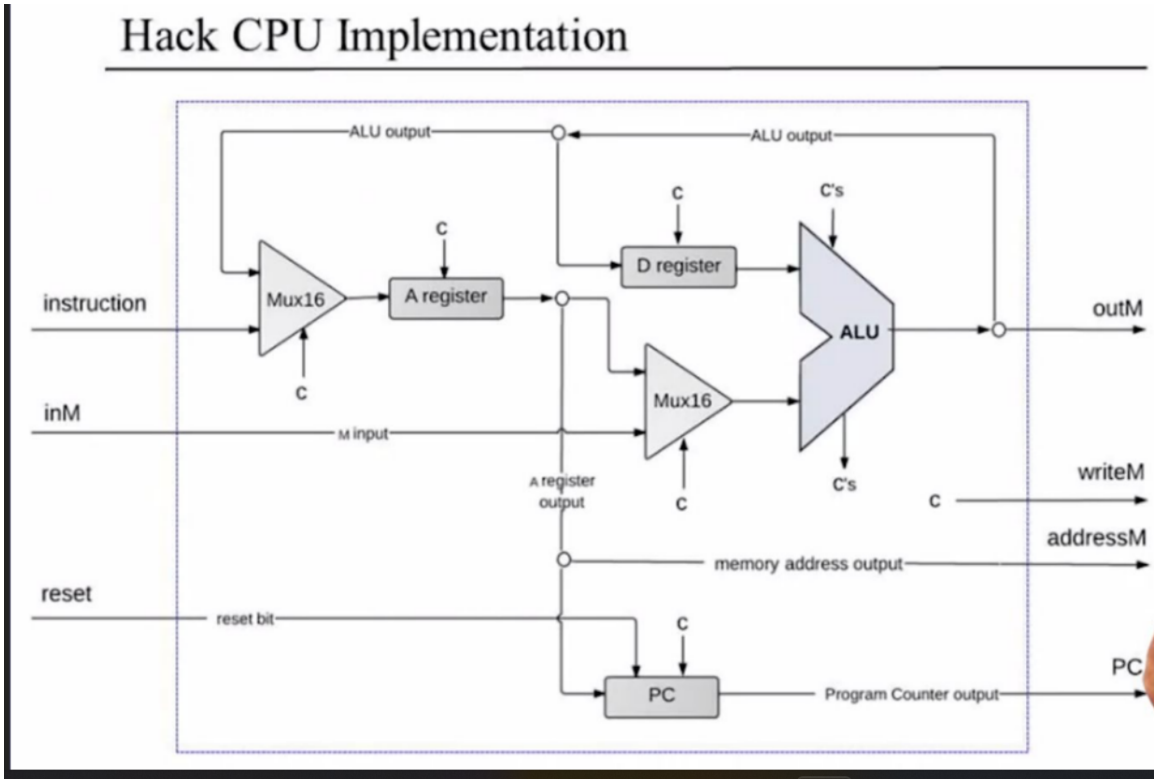
```
1
2  CHIP CPU {
3
4  IN   inM[16],
5       instruction[16],
6       reset;
7
8
9
10 OUT outM[16],
11      writeM,
12      addressM[15],
13      pc[15];
14
15 PARTS:
16 // Put your code here:
17 // If instruction[15]  is 0 then A instruction else C instruction
18 Mux16(a=instruction,b=ALUoutput,sel=instruction[15],out=Areginput);
19 // storing not of instruction[15] for other use
20 Not(in=instruction[15],out=Areg);
21
22 // If Areg is 1, load of A register will always be 1
```

```
// If Areg is 0, then load of A will depend on d1 ( as its C instruction)
Or(a=Areg,b=instruction[5],out=Aregload);
// A register block
ARegister(in=Areginput,load=Aregload,out=Aregoutput,out[0..14]=addressM);
// AddressM
//Register(in=Areginput,load=Aregload,out[0..14]=addressM);
Mux16(a=Aregoutput,b=inM,sel=instruction[12],out=AM);

// Inputs for ALU
// Here if intruction[15] is 0 then c1c2c3c4c5c6 will be

//001100 which in turn means D reg is a=0
// For c1
And(a=instruction[11],b=instruction[15],out=zx);
// For c2
And(a=instruction[10],b=instruction[15],out=nx);
 // For c3
Or(a=instruction[9],b=Areg,out=zy);
// For c4
Or(a=instruction[8],b=Areg,out=ny);
// For c5
And(a=instruction[7],b=instruction[15],out=f);
// For c6
And(a=instruction[6],b=instruction[15],out=no);

ALU(x=Dregoutput,y=AM,zx=zx,nx=nx,zy=zy,ny=ny,f=f,
no=no,out=outM,out=ALUoutput,zr=zr,ng=ng);

// If C instruction then writeM will depend on d3
And(a=instruction[15],b=instruction[3],out=writeM);

// d2 will be used for load of D register
And(a=instruction[15],b=instruction[4],out=Dregload);
// D register block
DRegister(in=ALUoutput,load=Dregload,out=Dregoutput);

// If Jump
// If either zr or ng is 1 then its not positive(np=1) else positive(p=1)
Or(a=zr,b=ng,out=np);
Not(in=np,out=p);

// For j3
And(a=instruction[0],b=p,out=j3);
// For j2
And(a=instruction[1],b=zr,out=j2);
// For j1
And(a=instruction[2],b=ng,out=j1);

// So if either of j1,j2,j3 is 1 then we have to jump
 // If not then only pc->increase by 1
```

```
73  Or(a=j1,b=j2,out=temp);
74  Or(a=temp,b=j3,out=j);
75  And(a=j,b=instruction[15],out=jump);
76
77  // This goes to PC with reset
78  PC(in=Aregoutput,load=jump,reset=reset,inc=true
79  ,out[0..14]=pc);
80
81  }
```

This is the code for working of CPU. The Hack CPU (Central Processing unit), consisting of an ALU, two registers named A and D, and a program counter named PC. The CPU is designed to fetch and execute instructions written in the Hack machine language.
In particular, functions as follows: Executes the inputted instruction according to the Hack machine language specification. The D and A in the language specification refer to CPU-resident registers, while M refers to the external memory location addressed by A, i.e. to Memory[A]. The inM input holds the value of this location. If the current instruction needs to write a value to M, the value is placed in outM, the address of the target location is placed in the addressM output, and the writeM control bit is asserted. (When writeM==0, any value may appear in outM).
The outM and writeM outputs are combinational: they are affected instantaneously by the execution of the current instruction. The addressM and pc outputs are clocked: although they are affected by the execution of the current instruction, they commit to their new values only in the next time step. If reset==1 then the CPU jumps to address 0 (i.e. pc is set to 0 in next time step) rather than to the address resulting from executing the current instruction.

The figure below shows the block diagram of CPU.



0.4 Code for Memory.hdl:-

```
1
2  CHIP Memory {
3  IN in[16], load, address[15];
4  OUT out[16];
5
6  PARTS:
7  // Put your code here:
8  // Demux for load
9  DMux4Way(in=load, sel=address[13..14], a=RAMload1, b=RAMload2, c=Screenload,
10 Or(a=RAMload1, b=RAMload2, out=RAMload);
11 RAM16K(in=in, load=RAMload, address=address[0..13], out=RAMoutput);
12 Screen(in=in, load=Screenload, address=address[0..12], out=Screenoutput);
13 Keyboard(out=Keyboardoutput);
14
15 // Output from all to one -> Mux 4 to 16
16 Mux4Way16(a=RAMoutput, b=RAMoutput, c=Screenoutput, d=Keyboardoutput, sel=add
17 }
```

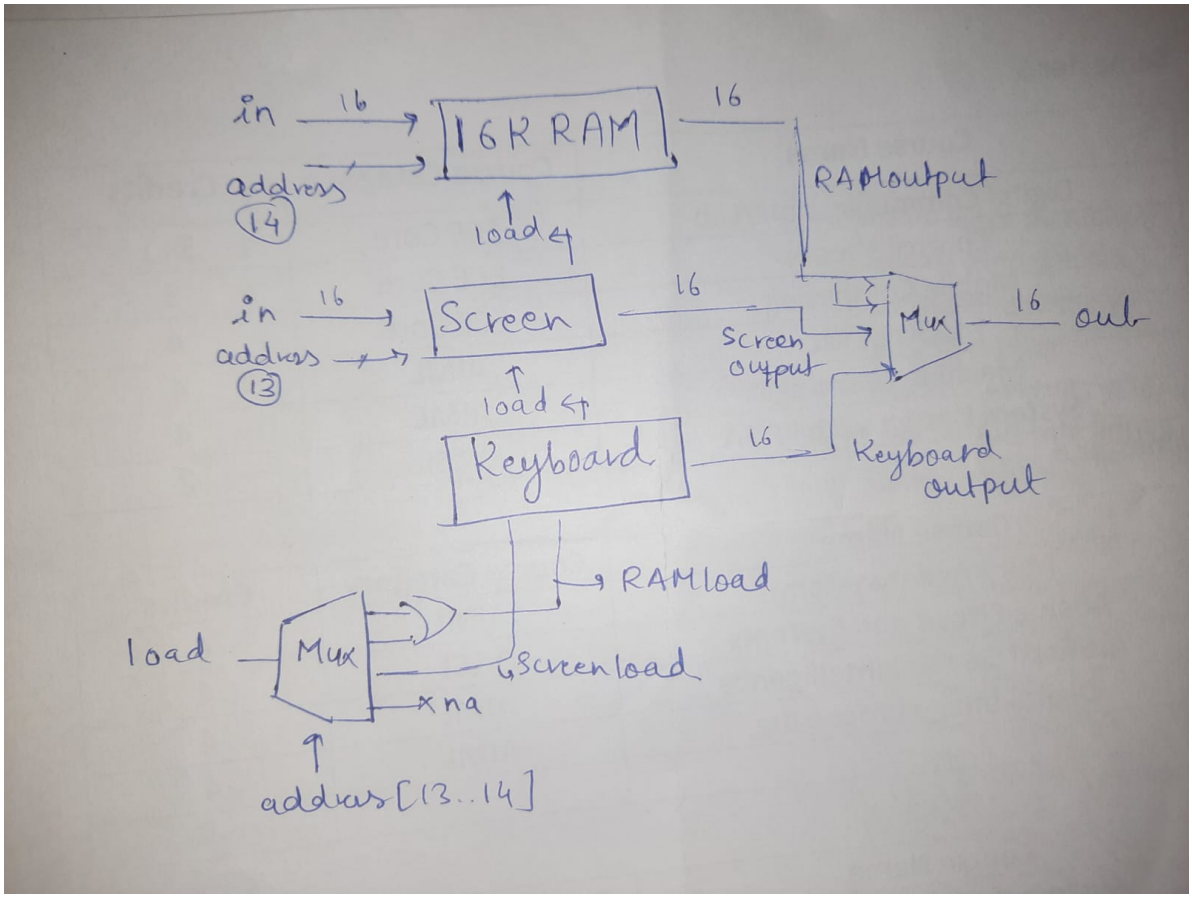
This is the code for working of Memory.

The complete address space of the Hack computer's memory, including RAM and memory-mapped I/O. The chip facilitates read and write operations, as follows:
Read: out(t) = Memory[address(t)](t)
Write: if load(t-1) then Memory[address(t-1)](t) = in(t-1)

In words: the chip always outputs the value stored at the memory location specified by address. If load==1, the in value is loaded into the memory location specified by address. This value becomes available through the out output from the next time step onward.

Address space rules: Only the upper 16K+8K+1 words of the Memory chip are used. Access to address>0x6000 is invalid. Access to any address in the range 0x4000-0x5FFF results in accessing the screen memory map. Access to address 0x6000 results in accessing the keyboard memory map.

The figure below shows the block diagram of Memory.



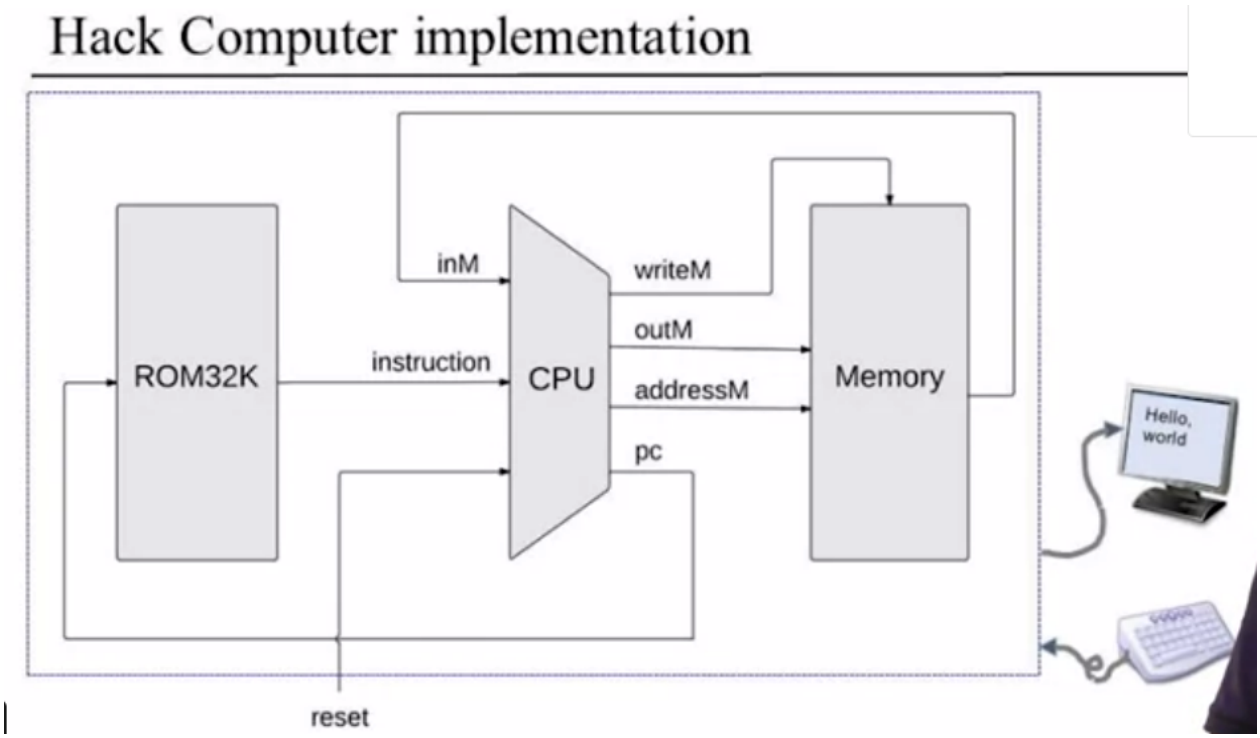
0.5 Code for Computer.hdl:-

```
CHIP Computer {

IN reset;

PARTS:
// Put your code here:
CPU(instruction=instruction, reset=reset, inM=inM, outM=outM, writeM=writeM, addre
Memory(in=outM, load=writeM, address=addressM, out=inM);
ROM32K(address=pc, out=instruction);
}
```

This is the code for working of Computer.

The HACK computer, including CPU, ROM and RAM. When reset is 0, the program stored in the computer's ROM executes. When reset is 1, the execution of the program restarts. Thus, to start a program's execution, reset must be pushed "up" (1) and "down" (0). From this point onward the user is at the mercy of the software. In particular, depending on the program's code, the screen may show some output and the user may be able to interact with the computer via the keyboard.

The figure below shows the block diagram of Computer.



Project 6:-

0.6 About:-

The code for Assembler.py is on the github link. If you run the program according to the commands given in readme, you will find new files of .hack which you can compare with the loaded files on Simulator.

Differnt types of functions used in program:-
1) Parser:-
Unpacks each instruction into underlying fields. It removes the unwanted parts like spaves and comments and gives you the type of intructions and depending upon the type it breaks the code into differnt fields like comp,dest etc.

2) Code:-
Translates each field into its corresponding binary values as per given in table below at last. According to different codes of comp, dest and jump like M, AMD and two differnt values of a , it maps all the values and return the given part required.

3) Symbol Table:-
Manages the symbol table. Here a special type of mapping is done according to symbols like KBD, SCREEN etc which are pre defined and we can add more symbols as we go through the program which we will be iterating.

4) Iterate:-

The first function here will iterate the file and tell type of Code and add symbols if required and use Parser function. The second function will be making a new file .hack and will be writing all the binary codes on it.This function will work accordingly to convert to binary codes and writing them on .hack file.

Symbolic syntax: *dest* = *comp* ; *jump*

Binary syntax: `1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3`

comp		c1	c2	c3	c4	c5	c6
0		1	0	1	0	1	0
1		1	1	1	1	1	1
-1		1	1	1	0	1	0
D		0	0	1	1	0	0
A	M	1	1	0	0	0	0
!D		0	0	1	1	0	1
!A	!M	1	1	0	0	0	1
-D		0	0	1	1	1	1
-A	-M	1	1	0	0	1	1
D+1		0	1	1	1	1	1
A+1	M+1	1	1	0	1	1	1
D-1		0	0	1	1	1	0
A-1	M-1	1	1	0	0	1	0
D+A	D+M	0	0	0	0	1	0
D-A	D-M	0	1	0	0	1	1
A-D	M-D	0	0	0	1	1	1
D&A	D&M	0	0	0	0	0	0
D|A	D|M	0	1	0	1	0	1
a=0	a=1						

dest	d1	d2	d3	effect: the value is stored in:
null	0	0	0	The value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register
MD	0	1	1	RAM[A] and D register
A	1	0	0	A register
AM	1	0	1	A register and RAM[A]
AD	1	1	0	A register and D register
AMD	1	1	1	A register, RAM[A], and D register

jump	j1	j2	j3	effect:
null	0	0	0	no jump
JGT	0	0	1	if out > 0 jump
JEQ	0	1	0	if out = 0 jump
JGE	0	1	1	if out ≥ 0 jump
JLT	1	0	0	if out < 0 jump
JNE	1	0	1	if out ≠ 0 jump
JLE	1	1	0	if out ≤ 0 jump
JMP	1	1	1	Unconditional jump