Project 4

MACHINE LANGUAGE

Machine language is used to tell computer what to do. We can use binary instructions to tell the computer what information to use, where to get it & what operations to perform on them. We can use a symbolic representation of machine language to understand and program better.

An assembler is used to convert symbolic representation to machine language.

3 Machine operations that we need

-> Arithmetic operation — add, substract etc...

-> Logical operations :- And, Or ... etc.

-> Flow Control :- goto instruction, if conditions etc ...

But there are few problems.

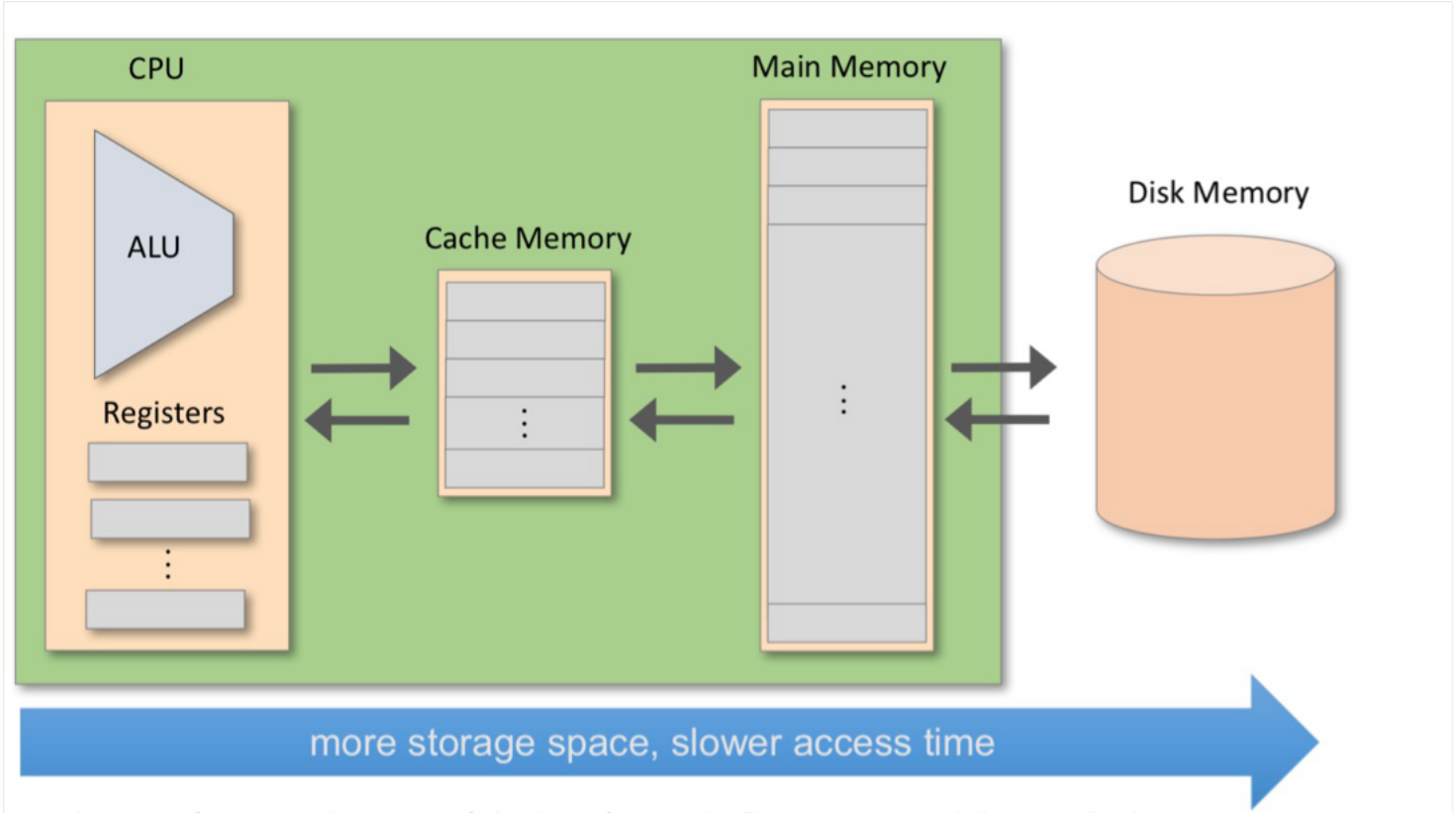
-> Accessing information can sometimes take longer than the operation itself

-> As we get more data, the cost to store & retrieve data increases.

Cost is both in terms of computational power & hardware economic cost.

Solution: Memory Hierarchy.

We keep the faster & more expensive memory near the CPU & cheaper and slower memory away from CPU with low memory capacity



There are few addressing modes or the way we can perform operations on the stored data. They are listed below

Addressing Modes

- Register
  - Add R1, R2 // R2 ← R2 + R1
- Direct
  - Add R1, M[200] // Mem[200] ← Mem[200] + R1
- Indirect
  - Add R1, @A // Mem[A] ← Mem[A] + R1
- Immediate
  - Add 73, R1 // R1 ← R1 + 73

Flow Control

We may need to jump from one code to another. eg. if we want to implement a loop, we can program it in such a way that unless a condition is met we keep jumping to a specific line.

Example:

```
    jgt R1, 0, CONT    // if R1>0 jump to CONT
    sub R1, 0, R1      // R1 ← (0 - R1)
CONT:
     ...
    // Do something with positive R1
```

A 16-bit machine consisting of:

- Data memory (RAM): a sequence of 16-bit registers:
  RAM[0], RAM[1], RAM[2],…
- Instruction memory (ROM): a sequence of 16-bit registers:
  ROM[0], ROM[1], ROM[2],…
- Central Processing Unit (CPU): performs 16-bit instructions
- Instruction bus / data bus / address buses.

The hack machine language (the one we will be building) consists of 3 16-bit registers:

A — used to store data or address of memory
M — represents currently addressed memory register
D — used to store data

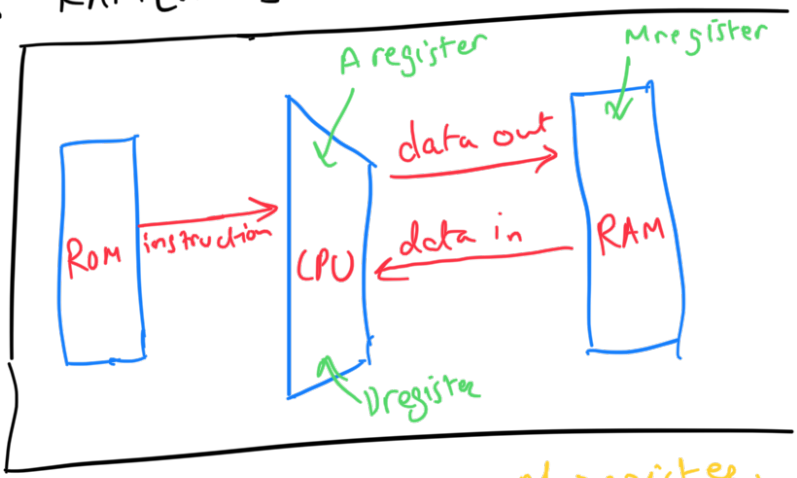
↑ I think of it as a variable in high-level languages

A INSTRUCTION

We can use this syntax -> @value, to select any register
Now A register becomes RAM[value]

Ex->
We want to take 250th register from RAM & set it to 100

```
@250
M=100
```



↳ first we select the 250th register from RAM. This now becomes M register. So we can say M=100, to change its value.

To change value in a register, one must first use A instruction to select it.

C INSTRUCTION

The C-instruction

dest = *comp* ; *jump* (both *dest* and *jump* are optional)

where:

comp =

```
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
           M,     !M,     -M,     M+1,        M-1, D+M, D-M, M-D, D&M, D|M
```

dest = `null, M, D, MD, A, AM, AD, AMD`

M refers to RAM[A]

jump = `null, JGT, JEQ, JGE, JLT, JNE, JLE, JMP`

if (*comp jump* 0) jump to execute the instruction in ROM[A]

Semantics:

- Compute the value of *comp*
- Stores the result in *dest*;
- If the Boolean expression (*comp jump* 0) is true, jumps to execute the instruction stored in ROM[A].

Example:

```
// If (D-1==0) jump to execute the instruction stored in ROM[56]
@56      // A=56
D-1;JEQ  // if (D-1 == 0) goto 56
```

C instruction is all about computation & it has 3 parts:
destination (dest), Computation (comp) & jump

optional

The above image contains all possible operations.

→ We compute the given instruction
→ Store it in the defined M register
→ Perform jump if the keyword is present.

Operations in their Binary form

A instruction → Symbolic → @ value
Binary → 0 value
↑ indicates it is a A operation

C instruction → Symbolic → dest = comp ; jump

1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3

- 1 1 1: opcode (if 0 it's A else C)
- useless
- a c1 c2 c3 c4 c5 c6: comp bits
- d1 d2 d3: destination
- j1 j2 j3: jump

The C-instruction: symbolic and binary syntax

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						

All C operations

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

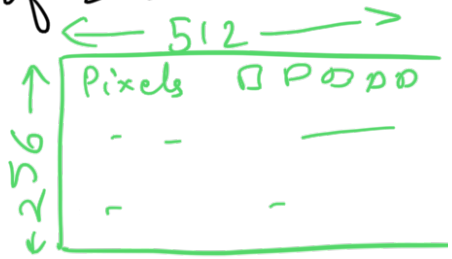
jump	j1	j2	j3	effect:
null	0	0	0	no jump
JGT	0	0	1	if out > 0 jump
JEQ	0	1	0	if out = 0 jump
JGE	0	1	1	if out ≥ 0 jump
JLT	1	0	0	if out < 0 jump
JNE	1	0	1	if out ≠ 0 jump
JLE	1	1	0	if out ≤ 0 jump
JMP	1	1	1	Unconditional jump

I/O → INPUT & OUTPUT

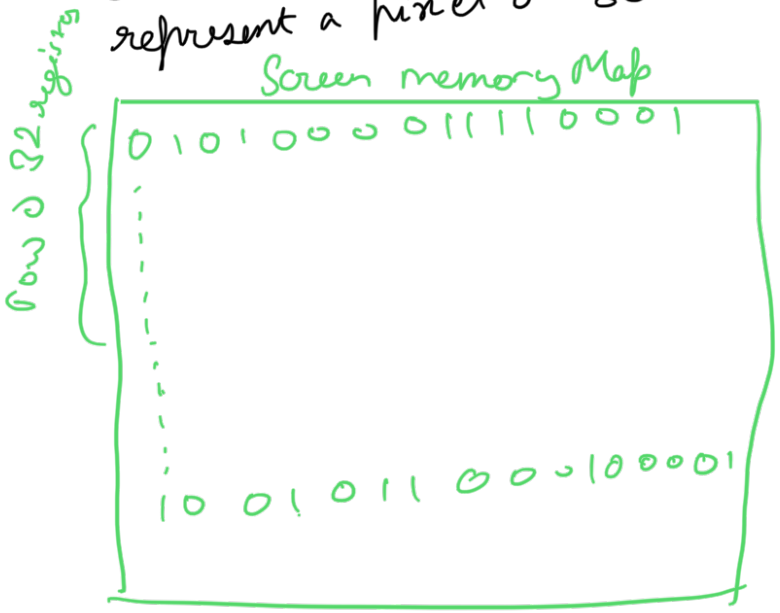
There are input & output devices which we need to connect to our hack computer. To interpret these devices we must first store their instructions. These instructions are stored in registers in RAM.

Display → Pixel data is stored in Screen Memory Map in RAM & display is refreshed at regular intervals to update based on this screen memory map.

In this computer, we have a display of 256 × 512.



each bit in our screen memory map represent a pixel on screen.



As we have a 16 Bit computer, we need 512/16 = 32 registers to represent a row of pixels. Thus we need 32 × 256 = 8192 registers.

Turning a pixel on or off

To access a particular pixel we first need to access the register this pixel resides on. As 32 registers are need in a row, 32 × row will give us which group of registers we are talking about.

need to select one out of these 32 ... 16 bits/pixels. We can get

So near ... registers. As each register ... a particular register → Col val // 16. → We need the absolute val.

So → our register is (32×row) + (col // 16)

Now we need to find the particular bit in that register. To do that we can → Col % 16, this will give us the reminder which is between 0 & 15

Keyboard

Keyboard just requires 1 register for it keyboard memory map. This is because we can represent all the required characters just using 16 bits.

If the keyboard is not pressed the defautt value is 0. if some other key is pressed than this table will be followed.

The Hack character set

Key	Code
0	48
1	49
...	...
9	57

Key	Code
A	65
B	66
...	...
Z	90

When no key is pressed, the resulting code is 0

Key	Code
(space)	32
!	33
"	34
#	35
$	36
%	37
&	38
'	39
(	40
)	41
*	42
+	43
,	44
-	45
.	46
/	47

Key	Code
:	58
;	59
<	60
=	61
>	62
?	63
@	64

Key	Code
[	91
/	92
]	93
^	94
_	95

Key	Code
newline	128
backspace	129
left arrow	130
up arrow	131
right arrow	132
down arrow	133
home	134
end	135
Page up	136
Page down	137
insert	138
delete	139
esc	140
f1	141
...	...
f12	152

HACK PROGRAMMING

ENDING A PROGRAM – Infinite loop

just add an infinite loop to end the program

ADDs 2 Num

```
@0      // Register 0
D=M     // D is now the value stored in register 0
@1      // Register 1
D=D+M   // D is now sum of 2 numbers
@2      // Register 2
M=D     // Register 2 stores the sum of 2 numbers
@6
0;JMP   // Infinite loop
```

BUILT IN SYMBOLS

Hack assembly language has some built in symbols

Built-in symbols

The Hack assembly language features *built-in symbols*:

symbol	value
R0	0
R1	1
R2	2
...	...
R15	15
SCREEN	16384
KBD	24576

symbol	value
SP	0
LCL	1
ARG	2
THIS	3
THAT	4

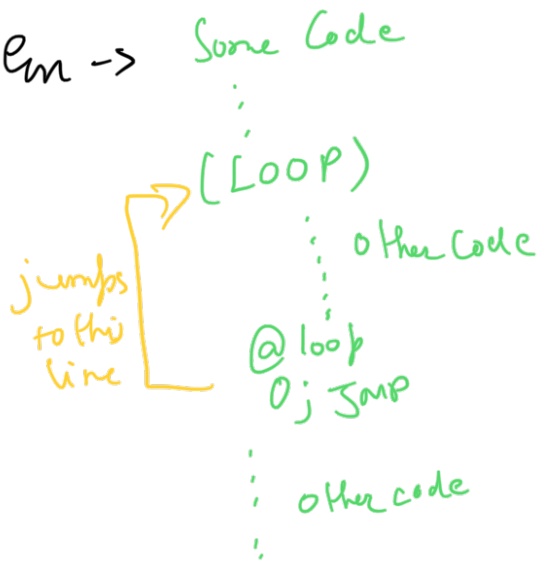
- `R0`, `R1` ,…, `R15` : "virtual registers", can be used as variables
- `SCREEN` and `KBD` : base addresses of I/O memory maps
- Remaining symbols: used in the implementation of the Hack *virtual machine*, discussed in chapters 7-8.

LABELS

You can use labels to jump to specific parts of code. It also improve readability.

u can name it anything { @LOOP → referencing a labled
(LOOP) → declaring a labled

Eg →



VARIABLES

There are 16 registers with their own symbolic representation → R0 ... R15. Variables are stored from R16 onwards.

Here how to declare & use a variable → @temp

Variables

Variable usage example:

```
// Program: Flip.asm
// flips the values of
// RAM[0] and RAM[1]

// temp = R1
// R1 = R0
// R0 = temp
   @R1
   D=M
   @temp
   M=D     // temp = R1

   @R0
   D=M
   @R1
   M=D     // R1 = R0

   @temp
   D=M
   @R0
   M=D     // R0 = temp

(END)
   @END
   0;JMP
```

symbol used for the first time

symbol used again

resolving symbols

Symbol resolution rules:

- A reference to a symbol that has no corresponding label declaration is treated as a reference to a variable
- If the reference *@symbol* occurs in the program for first time, *symbol* is allocated to address 16 onward (say *n*), and the generated code is *@n*
- All subsequencet *@symbol* commands are translated into *@n*

In other words: variables are allocated to `RAM[16]` onward.

Memory

0	@1
1	D=M
2	@16 // @temp
3	M=D
4	@0
5	D=M
6	@1
7	M=D
8	@16 // @temp
9	D=M
10	@0
11	M=D
12	@12
13	0;JMP
14	
15	
	⋮
32767	

POINTERS

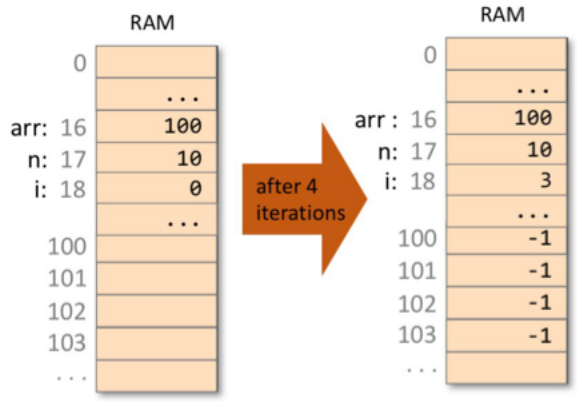
If we want to use a data structure like an array, we just need to store the address of first item in the array & its length. Variables that represents these addresses are called pointers. We can use pointers as shown below

Pointers

Example:

```
(LOOP)
   // if (i==n) goto END
   @i
   D=M
   @n
   D=D-M
   @END
   D;JEQ

   // RAM[arr+i] = -1
   @arr
   D=M
   @i
   A=D+M
   M=-1

   // i++
   @i
   M=M+1

   @LOOP
   0;JMP

(END)
   @END
   0;JMP
```

typical pointer manipulation



- Pointers: Variables that store memory addresses (like arr)
- Pointers in Hack: Whenever we have to access memory using a pointer, we need an instruction like A=*expression*
- Semantics: "set the address register to some value".

@arr // stores the address of first value in register -> 100
D=M // D becomes 100
@i // 0, 1, 2, ... the counter
A=D+M // A becomes 100 then 101 then 102...
M=-1 // Put -1 in the register with address A

MULTIPLICATION

Performs multiplication btw values in R1 & R0 & stores it in R2

```
// Put your code here.

//Take value in R0 and store it in a variable caled value1
@R0
D=M
@value1
M=D

//Take value in R1 and store it in a variable caled value2
@R1
D=M
@value2
M=D

//SET R2 to 0
@R2
M=0

// IF value2 is 0 END THE PROGRAM
@value2
D=M
@END
D; JEQ

// Loop. Add value1 to r2 as long as value2 is above 0
(LOOP)

//Take value in R2, and value1, then add them and store in R2
@R2
D=M
@value1
D = D + M
@R2
M=D

//Reduce value2 by 1
@value2
M=M-1
D=M

//Loop as long as value2 is > 0
@LOOP
D; JGT

//END THE PROGRAM
(END)
@END
0; JMP
```

takes user input & store it in value 1 (variable)

takes user input & store it in value2 (variable)

initially multiplication is 0

If value 2 is 0 then end the program as the result will be 0. (Do same for Val 1)

(LOOP BEGINS)

Add value in R2 & value1 store it in R2.

Reduce value2 by 1. This will eventually end the loop

if value2 is >0 keep looping, else end.

Program Ends.

As we do not have any functionality for multiplication, we just need to add the value 1 to itself value 2 number of times.

This one is pretty simple & straightforward. Please give it a try.

FILL SCREEN

When user presses a key turn display black else keep it white.

```
// Put your code here.

(START)
@SCREEN
D=A
@R0
M=D

@temp
M=-1

(LOOP)
@24576
D=M
@SWITCH
D;JEQ

(CONT)
@temp
D=M
@R0
A=M
M=D

@R0
M = M+1
D=M
@R1
D = D-M
@START
D;JEQ

@LOOP
0;JMP

(SWITCH)
@temp
M=0
@CONT
0;JMP

(END)
@START
0;JMP
```

set value of screen register, address of where it starts & store it in R0

This will store if we should make screen white or Black. -1 is Black, 0 is white.

This stores the value of key pressed. Go to the Switch part of code if the value of D is 0.

Infinite loop

R0 stores the address of register. We take this address, go to that register & change its val to 0 or -1. To make the screen black or white

Increment val in R0 by 1 to access next register on screen

R1 stores the value of last register in screen memory map. We substract val in R0 & R1 to see which register we are at. if we are at last register, we restart everything. We do this to not get any errors.

This is to switch the value of temp to 0, if no keys are pressed on computer. This will make the screen white.

To keep updating the screen.

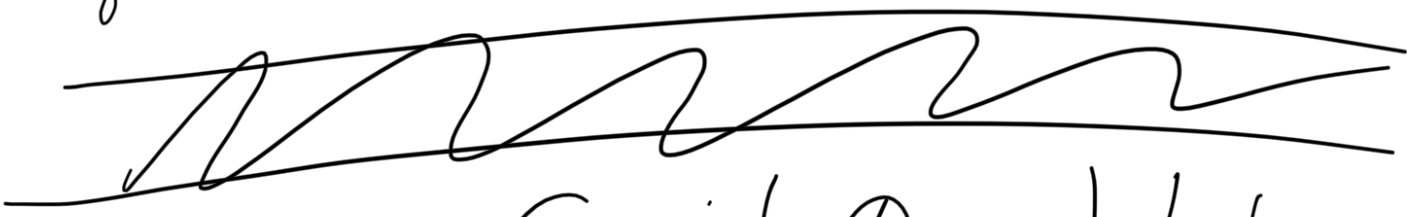
In the above code I forgot to add a line. The line is updated in the code in github.

after the line 22 just add ->

```
@24576
D=A
@R1
M=D
```

} One More than the last register in screen memory Map

I have used a temp variable to store the "state" of screen. It is either 0 (white) or -1 (Black) we change this temp to 0 if no key is pressed. Then we go through the loop & change the value of each register to temp.



VOILA!!!