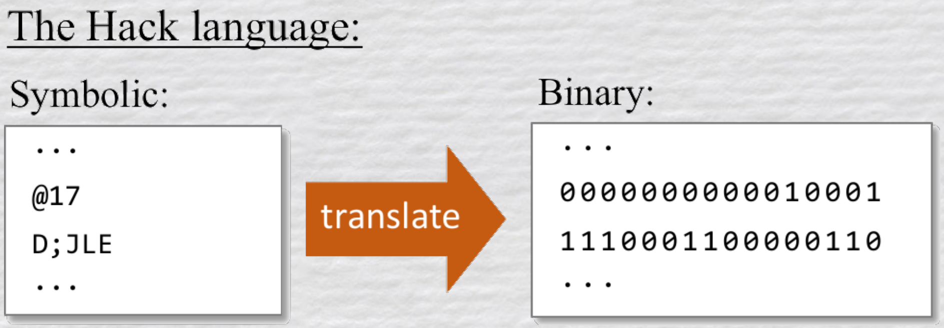
Computer Organization

HACK Disassembly

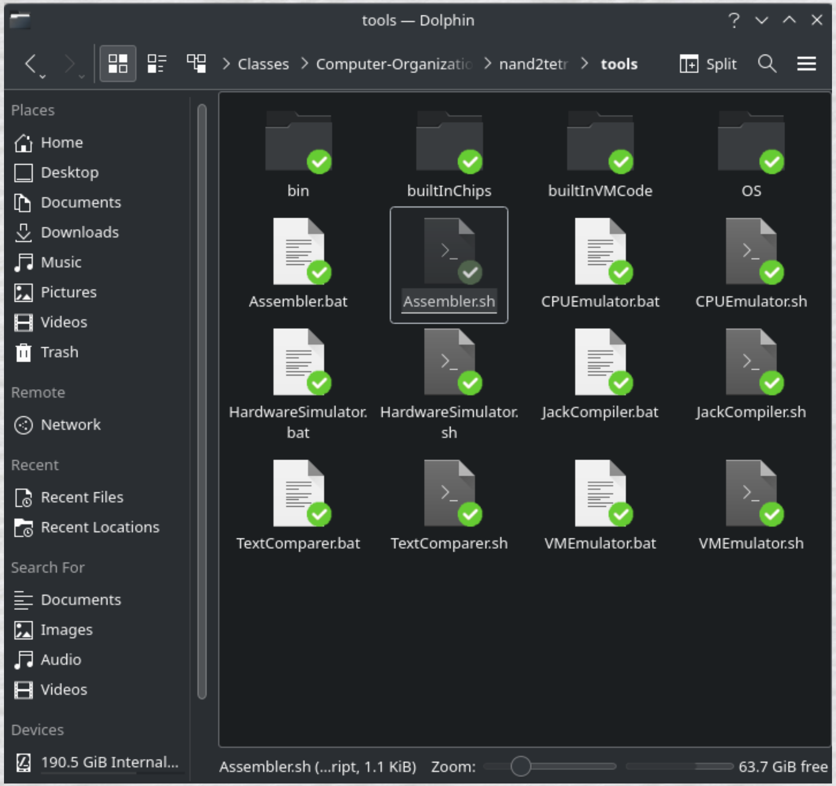
Assembly

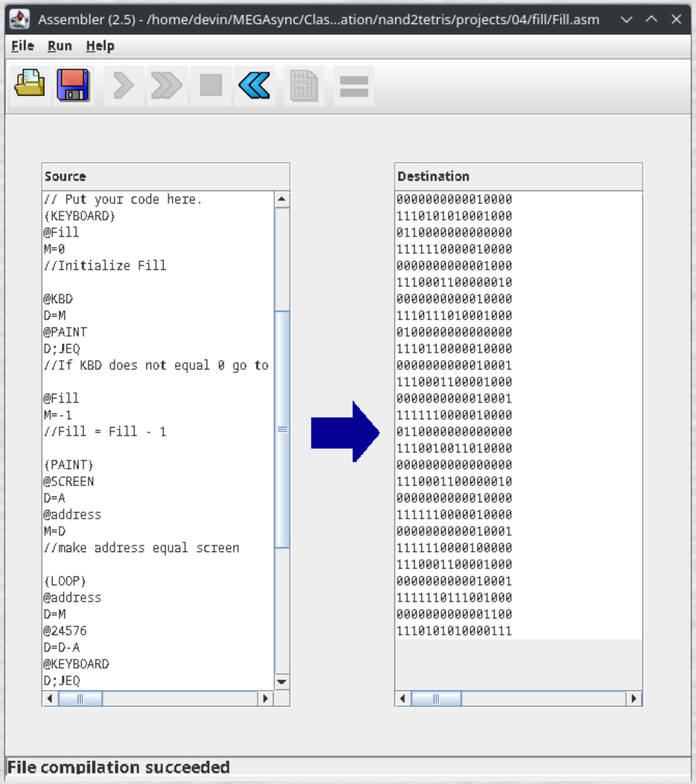
- What we have done so far in the course with the Hack assembly language is simply assembling it to Binary code

- This takes our symbolic assembly language and translates it to Binary



Nand2Tetris - Assembler





Disassembly

- Since we have a static assembler method, we can easily reverse engineer it and derive a disassembler in any high-level, programming language we want

<u>The Hack language:</u>

Binary:

```
...
0000000000010001
1110001100000110
...
```

translate

Symbolic:

```
...
@17
D;JLE
...
```

Identifying the Assembly Process

- The first step we need to take is to identify how we get to our Binary format
- This means we need to understand how the assembler generates the Binary strings that the machine reads
- We can do that by breaking down the assembly process itself...

The Binary Strings

- Each Binary string is very specific to a symbolic Hack instruction.
  - **There is absolutely no ambiguity here**

- We know that ever Bit in the resulting strings is specific to some aspect of an instruction type
  - We also know that we only have 2 types of instructions...

The Instruction Types

- Looking back at Chapter 4, we can verify that we have 2 instruction types
  - A Instructions
  - C Instructions

- The first step is verifying what instruction type we are dealing with by analyzing the Op-Code (the MSB)

A Instruction

Symbolic syntax:

`@`*value*

Where *value* is either:

a non-negative decimal constant $\leq 65535\ (=2^{16}-1)$

or a symbol bound to a constant

<u>Example:</u>

Symbolic:

`@6`

Binary syntax:

`0`$vvvvvvvvvvvvvvv$

Where:

`0` is the A instruction op-code

$v\ v\ v\ \ldots\ v$ is the 15-bit binary representation of the constant

Binary:

`0000000000000110`

A Instruction – cont.

- We can easily tell what we need to do with an A Instruction since it's simply an Assignment Instruction
- If the Op-Code is 0, convert the remaining 15 Bits to a Decimal Value
  - Once we do that, we simply prepend '@' to the value
- C Instructions require a bit more work...

C Instruction

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

comp is mandatory.
If *dest* is empty, the = is omitted; If *jump* is empty, the ; is omitted

Binary syntax:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	*a*	*c*	*c*	*c*	*c*	*c*	*c*	*d*	*d*	*d*	*j*	*j*	*j*

C instruction op-code

not used

comp bits

dest bits

jump bits

C Instruction – cont.

- The C Instruction is far more involved since it will require using several look-up tables for our various Bit clusters
  - Comp Bits (a and c bits)
  - Dest Bits (d bits)
  - Jump Bits (j bits)

- We can simply use Data Structures like Python Dictionaries or C++ Maps to create these tables

Computation Bits

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

comp is mandatory.
If *dest* is empty, the = is omitted; If *jump* is empty, the ; is omitted

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Binary syntax:	1	1	1	*a*	*c*	*c*	*c*	*c*	*c*	*c*	*d*	*d*	*d*	*j*	*j*	*j*

$\underbrace{a\ c\ c\ c\ c\ c\ c}_{\textit{comp}\text{ bits}}$

- We read in the 6-Bit C String and associate that with the appropriate symbolic instruction

- We can combine the A and M instructions and select which we wish to use based on our A Bit

comp (*a*==0)	*comp* (*a*==1)	*c*	*c*	*c*	*c*	*c*	*c*
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

Destination Bits

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

comp is mandatory.
If *dest* is empty, the = is omitted; If *jump* is empty, the ; is omitted

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	*a*	*c*	*c*	*c*	*c*	*c*	*c*	*d*	*d*	*d*	*j*	*j*	*j*

Binary syntax: 1 1 1 *a c c c c c c* $\underbrace{d\ d\ d}_{dest\ \text{bits}}$ *j j j*

dest	*d*	*d*	*d*	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
DM	0	1	1	D register and RAM[A]
A	1	0	0	A register
AM	1	0	1	A register and RAM[A]
AD	1	1	0	A register and D register
ADM	1	1	1	A register, D register, and RAM[A]

Destination Bits

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

comp is mandatory.
If *dest* is empty, the = is omitted; If *jump* is empty, the ; is omitted

Binary syntax:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	*a*	*c*	*c*	*c*	*c*	*c*	*c*	*d*	*d*	*d*	*j*	*j*	*j*

jump bits

jump	*j*	*j*	*j*	effect:
null	0	0	0	no jump
JGT	0	0	1	if *comp* $>$ 0 jump
JEQ	0	1	0	if *comp* $=$ 0 jump
JGE	0	1	1	if *comp* $\geq$ 0 jump
JLT	1	0	0	if *comp* $<$ 0 jump
JNE	1	0	1	if *comp* $\neq$ 0 jump
JLE	1	1	0	if *comp* $\leq$ 0 jump
JMP	1	1	1	Unconditional jump

C Instruction – Format

- Once we have all the Bits accounted for, we simply need to format the instruction
- Recall that only the Comp Bits are required, so you may have varying formats due to the Dest Bits and Jump Bits
- The final format should look something like this:
- Dest=Comp;Jump

Instructions – Overview

A instruction

Symbolic: @*xxx* — (*xxx* is a decimal value ranging from 0 to 32767, or a symbol bound to such a decimal value)

Binary: `0` *vvvvvvvvvvvvvvv* — (*vv ... v* = 15-bit value of *xxx*)

C instruction

Symbolic: *dest* = *comp* ; *jump* — (*comp* is mandatory. If *dest* is empty, the = is omitted; If *jump* is empty, the ; is omitted)

Binary: `111`*acccccc ddd jjj*

Predefined symbols:

symbol	value
R0	0
R1	1
R2	2
...	...
R15	15
SP	0
LCL	1
ARG	2
THIS	3
THAT	4
SCREEN	16384
KBD	24576

comp (a == 0)	comp (a == 1)	c	c	c	c	c	c
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	d	d	d	Effect: store comp in:
null	0	0	0	the value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register (reg)
DM	0	1	1	RAM[A] and D reg
A	1	0	0	A reg
AM	1	0	1	A reg and RAM[A]
AD	1	1	0	A reg and D reg
ADM	1	1	1	A reg, D reg, and RAM[A]

jump	j	j	j	Effect:
null	0	0	0	no jump
JGT	0	0	1	if $comp > 0$ jump
JEQ	0	1	0	if $comp = 0$ jump
JGE	0	1	1	if $comp \geq 0$ jump
JLT	1	0	0	if $comp < 0$ jump
JNE	1	0	1	if $comp \neq 0$ jump
JLE	1	1	0	if $comp \leq 0$ jump
JMP	1	1	1	unconditional jump