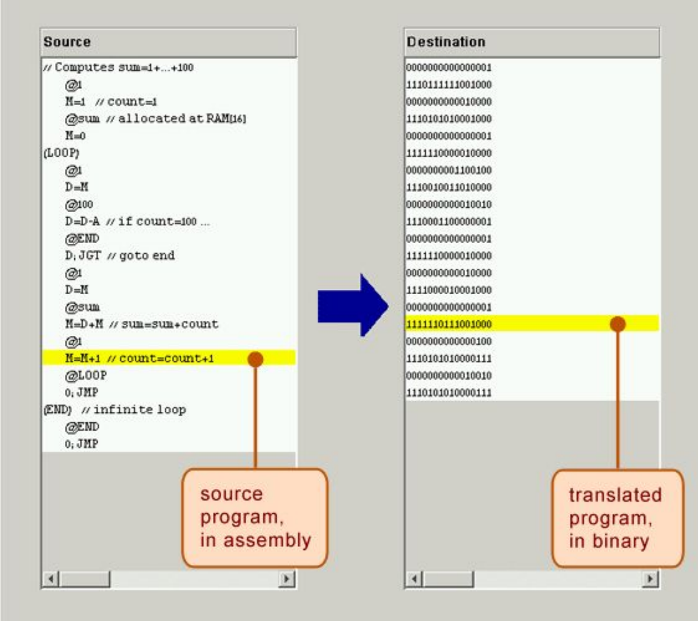
HACK Assembler:
Symbol and Instruction Lookup Table

Week 12

Assembler Overview

Assemblers are programs that will take **assembly code** and translate it into the corresponding **machine instructions** (binary)



Source

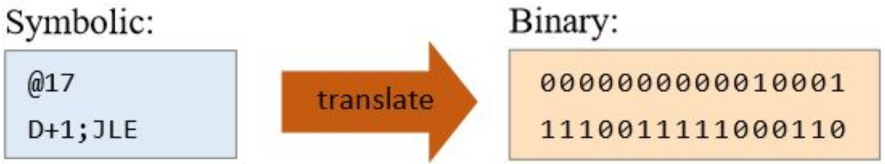
```
// Computes sum=1+...+100
    @1
    M=1  // count=1
    @sum // allocated at RAM[16]
    M=0
(LOOP)
    @1
    D=M
    @100
    D=D-A // if count=100 ...
    @END
    D;JGT // goto end
    @1
    D=M
    @sum
    M=D+M // sum=sum+count
    @1
    M=M+1 // count=count+1
    @LOOP
    0;JMP
(END)  // infinite loop
    @END
    0;JMP
```

Destination

```
0000000000000001
1110111111001000
0000000000010000
1110101010001000
0000000000000001
1111110000010000
0000000001100100
1110010011010000
0000000000010010
1110001100000001
0000000000000001
1111110000010000
0000000000010000
1111000010001000
0000000000000001
1111110111001000
0000000000000100
1110101010000111
0000000000010010
1110101010000111
```

source program, in assembly

translated program, in binary



Symbolic:

```
@17
D+1;JLE
```

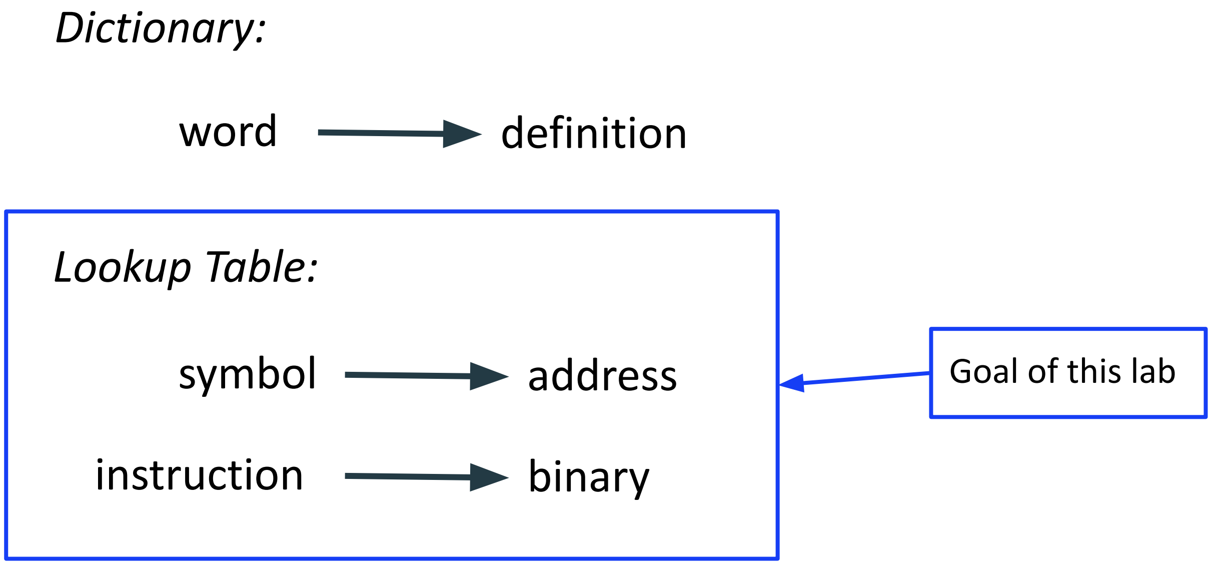
Binary:

```
0000000000010001
1110011111000110
```

How do translations work?

Assemblers use lookup tables to translate symbols and instructions

- lookup tables act like a dictionary



Dictionary:

word → definition

Lookup Table:

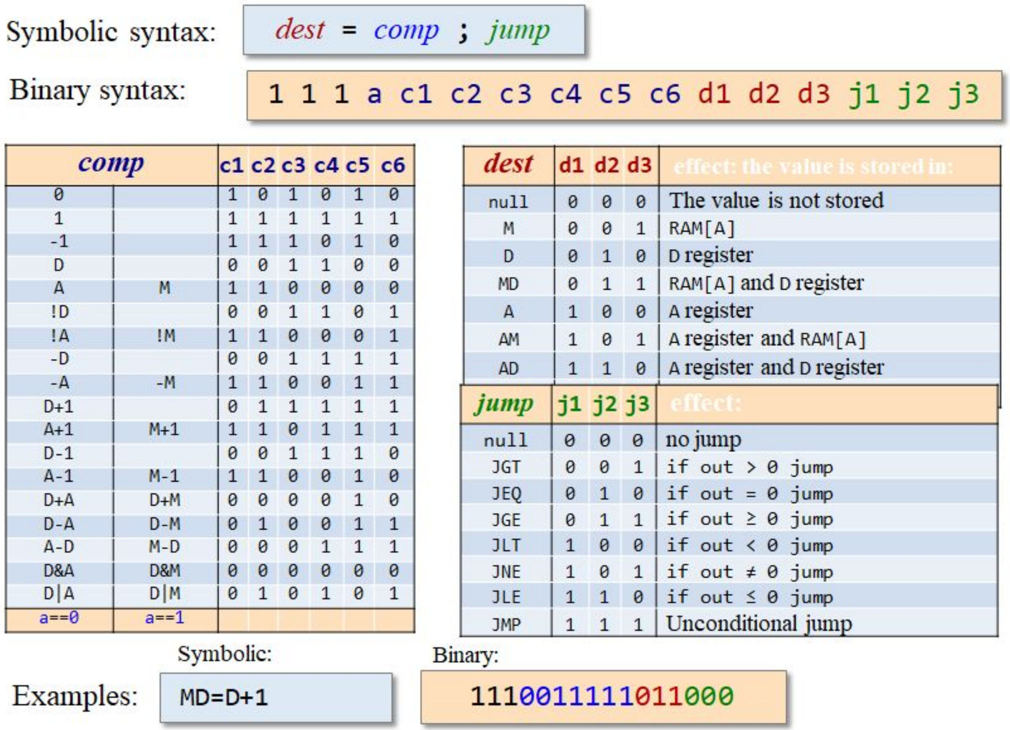
symbol → address

instruction → binary

Goal of this lab

Recap: HACK Assembly Instructions

C instructions



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

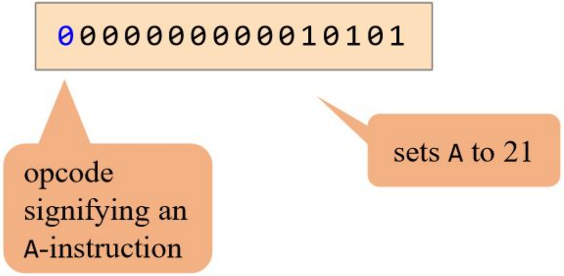
jump	j1	j2	j3	effect:
null	0	0	0	no jump
JGT	0	0	1	if out > 0 jump
JEQ	0	1	0	if out = 0 jump
JGE	0	1	1	if out ≥ 0 jump
JLT	1	0	0	if out < 0 jump
JNE	1	0	1	if out ≠ 0 jump
JLE	1	1	0	if out ≤ 0 jump
JMP	1	1	1	Unconditional jump

	Symbolic:	Binary:
Examples:	MD=D+1	1110011111011000

A instructions

Symbolic syntax: @value

Binary Syntax:



0000000000010101

opcode signifying an A-instruction

sets A to 21

Symbols

Symbols are words that refer to some **memory location**

Can be:

- ROM
- data memory

Whenever A instruction `@wordNOTnumber` is encountered, needs to be replaced with the appropriate address

This symbol can be the following:

Labels

Labels are references to Instruction ROM addresses

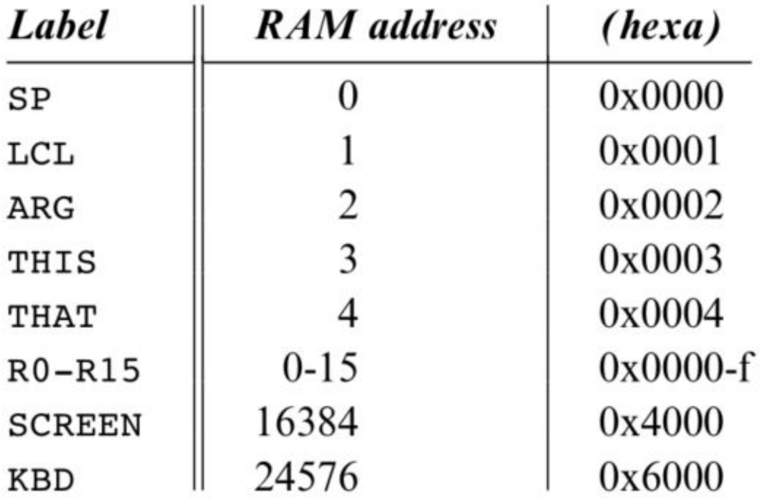
```
(loop)
   // do some things
   @loop
   0;JMP
```

Variables

Variables are references to data memory addresses

Implementing Symbol Lookup Table

Predefined Symbols



Label	*RAM address*	*(hexa)*
SP	0	0x0000
LCL	1	0x0001
ARG	2	0x0002
THIS	3	0x0003
THAT	4	0x0004
R0-R15	0-15	0x0000-f
SCREEN	16384	0x4000
KBD	24576	0x6000

Simplest way to implement a lookup table is using a **hash table** or **hash map**

Hash Table Review

- takes O(1) time to access values
- maps key to value
  - (key, value)

For our implementation, we're mapping *strings* (symbols) to *integers* (addresses)

```
symbolTable = HashTable(string, integer)
```

Steps to Create Symbol Table

1. Populate symbol table with predefined values

Populated first since already known

2. Populate symbol table with labels

3. Populate symbol table with variables

Both labels and variables are specified by @symbol

- *Labels* populated first with a scan of the entire assembly file for **(label)** to identify the instruction line number corresponding to the label
- *Variables* are then the symbols that aren't classified as labels. Data Memory Register addresses are assigned to variable symbols starting with RAM[16]

Adding Predefined Symbols to Table

```
symbolTable = HashTable(string, integer)

symbolTable[“SP”] = 0
symbolTable[“LCL”] = 1
symbolTable[“ARG”] = 2
symbolTable[“THIS”] = 3
symbolTable[“THAT”] = 4

for r in (0, 1, 2, ... , 15)
    symbolTable[“R” + string(r)] = r

symbolTable[“SCREEN”] = 16384
symbolTable[“KBD”] = 24576
```

Adding Labels to Table

```
file = openFile(assemblyFile)
pc = 0 // refer to location in file

for line in file:

    clean = removeCommentsAndWhitespace(line)

    if ( isLabel(clean) ):
        label = removeParenthesis(clean)

        if ( NOT symbolTable.contains(label) ):
            symbolTable[label] = pc

    if ( isInstruction(clean) ):
        pc = pc + 1
```

Adding Variables to Table

```
file = openFile(assemblyFile)
nextAddress = 16 // variable assigned starting at addr 16
for line in file:

    clean = removeCommentsAndWhitespace(line)

    if ( isValidAInstruction(clean) ):
        AInstructionVal = clean.Strip(“@”)

        // the value is not a number and not in the symbol
        // table already. (this means it is not a label)
        if ( isNotNumber(AInstructionVal)
            AND NOT symbolTable.contains(AInstructionVal) ):

            symbolTable[AInstructionVal] = nextAddress
            nextAddress = nextAddress + 1
```

Using Symbol Lookup Table

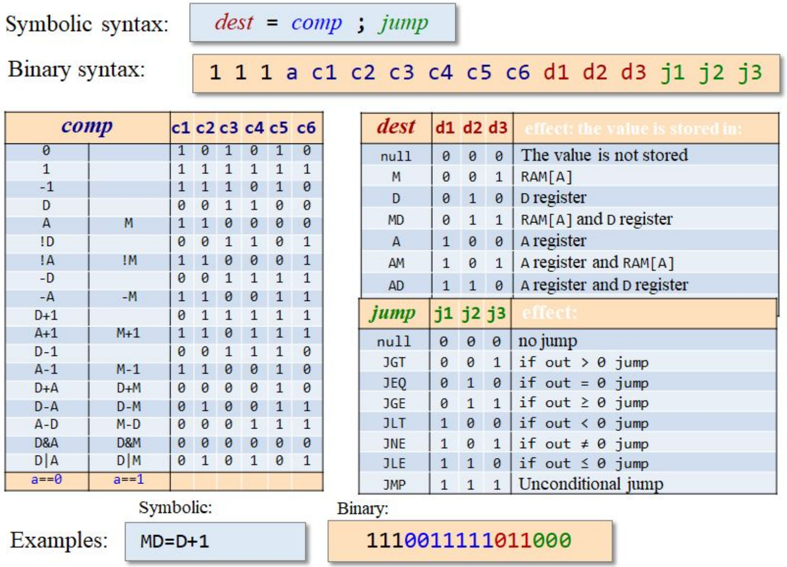
```
instruction = “@R5”

addr = symbolTable[ instruction.Strip(“@”) ]

bin = “0”+ to15BitBinary(addr) // instruction in binary
```

With this, the lookup table for A instructions is complete!

Implementing Instruction Lookup Table



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

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

	Symbolic:	Binary:
Examples:	MD=D+1	1110011111011000

Now need to make a lookup table for C instructions

How to implement this?

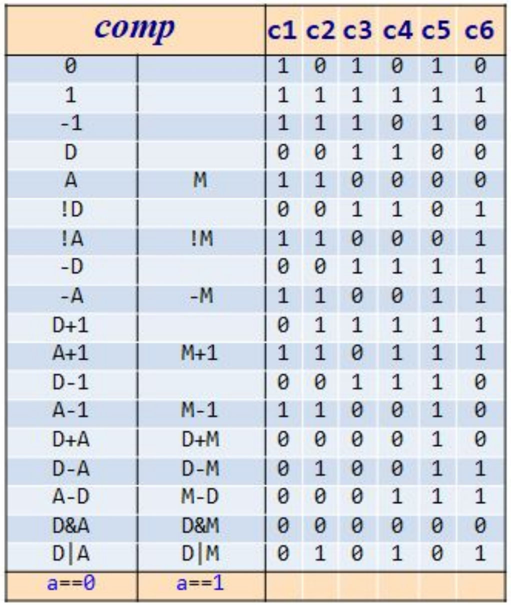
Create a lookup table for each portion

```
compTable = HashTable(string, integer)

destTable = HashTable(string, integer)

jumpTable = HashTable(string, integer)
```

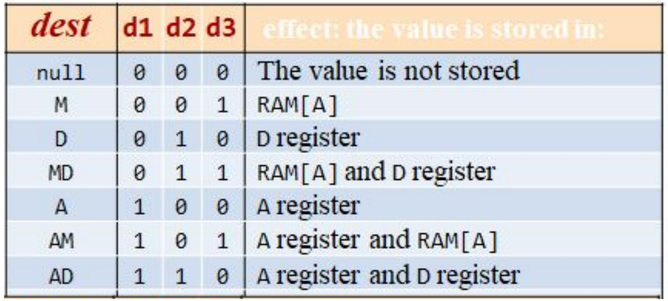
Implementing Instruction Lookup Table



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						

```
compTable["0"] = "0101010";
compTable["1"] = "0111111";
compTable["-1"] = "0111010";
... // many entries omitted because it gets long
compTable["A+1"] = "0110111";
compTable["D-1"] = "0001110";
compTable["A-1"] = "0110010";
compTable["D+A"] = "0000010";
... // many entries omitted because it gets long
compTable["D&M"] = "1000000";
compTable["D|M"] = "1010101";
```

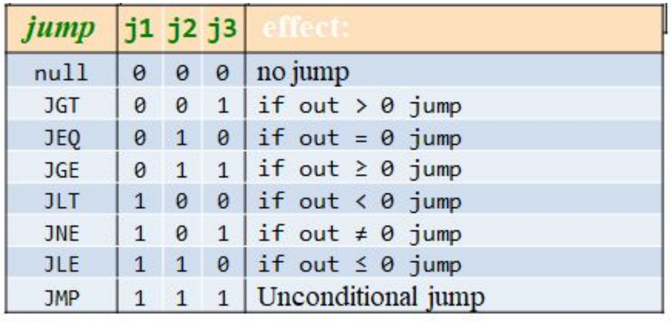
Implementing Instruction Lookup Table



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

```
destTable[“M”] = “001”
destTable[“D] = “010”
destTable[“MD”] = “011”
... // finish off destination
```

Implementing Instruction Lookup Table



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

```
jumpTable[“JGT”] = “001”
jumpTable[“JEQ”] = “010”
jumpTable[“JGE”] = “011”
... // finish off jump
```

Implementing Instruction Lookup Table

```
compTable = HashTable(string, integer)
destTable = HashTable(string, integer)
jumpTable = HashTable(string, integer)

compTable["0"] = "0101010";
compTable["1"] = "0111111";
compTable["-1"] = "0111010";
... // many entries omitted because it gets long

destTable[“M”] = “001”
destTable[“D] = “010”
destTable[“MD”] = “011”
... // finish off destination

jumpTable[“JGT”] = “001”
jumpTable[“JEQ”] = “010”
jumpTable[“JGE”] = “011”
... // finish off jump
```

Using Instruction Lookup Table

```
tokens = deconstructCInstruction(instruction)
prefix = “111”
dest  = “000”
jump = “000”

comp = compTable[ tokens.comp ]

if (tokens.dest != NULL):
    dest = destTable[ tokens.dest ]

if (token.jump != NULL):
    jump = jumpTable[ tokens.jump ]

binary = prefix + comp + dest + jump
```

Note:
Imagine that tokens is a class which stores each separate part of the C instruction

The lookup table for C instructions is now complete!