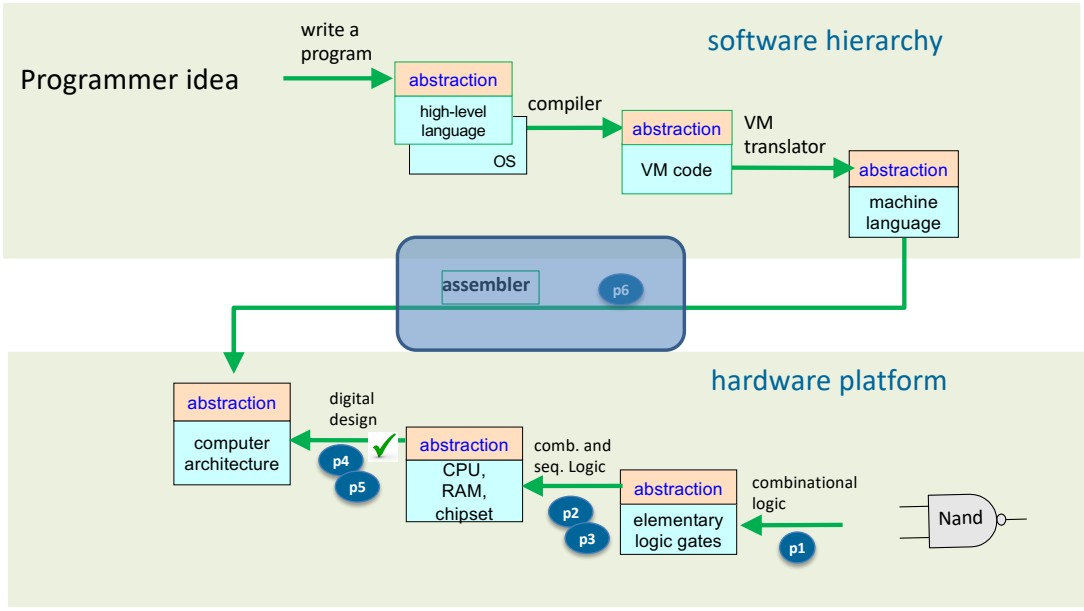
CSCE 312: Computer Organization

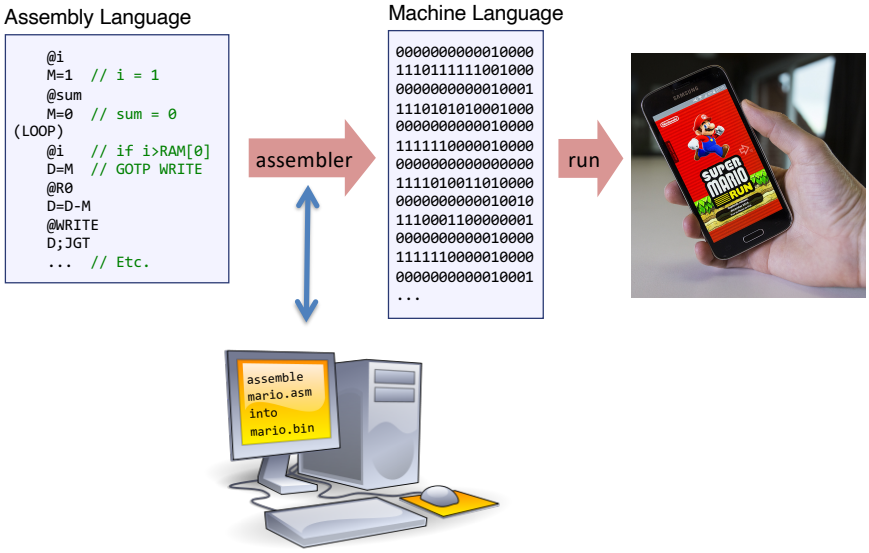
David Kebo Houngninou

Assembler Basic Concepts

The Big Picture



Assembly process



Why care about assemblers?

- Assemblers are the first rung up the software hierarchy ladder
- An assembler is a translator of a simple language
- Writing an assembler = low-impact practice for writing compilers.

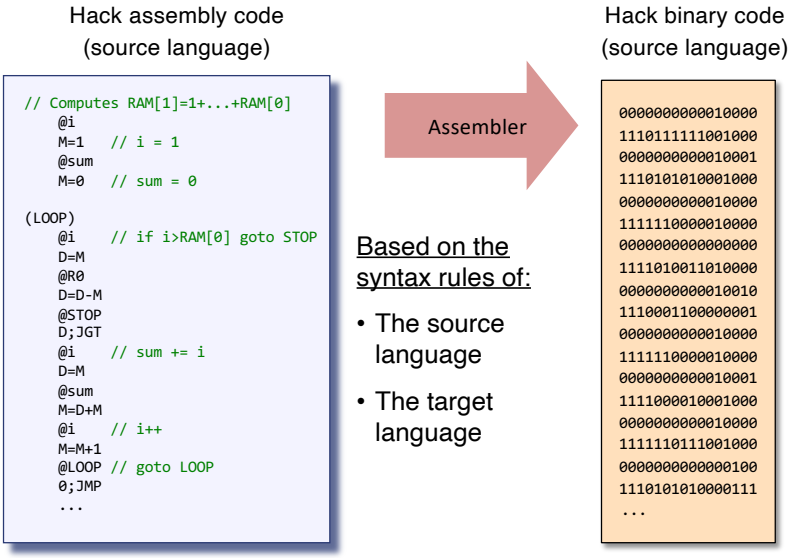
Assembler: lecture plan

- The assembly process
- The Hack assembly language
- The assembly process: instructions
- The assembly process: symbols
- Developing an assembler
- Project 6 overview

Example

```
// Write a program that adds the content of two memory
// locations 0 and 1 and stores the result in the memory
// location 2. RAM[2] = RAM[1] + RAM[0]
```


The translator's challenge



Based on the syntax rules of:

- The source language
- The target language

Assembly example

Assembler = simple translator

- Translates each assembly command into binary machine instructions
- Handles symbols (e.g. i, sum, LOOP, …)
- Handles Whitespace and Comments

```
@ sum
D = M
@ foo
M = D
```


Basic Assembler Logic

Repeat

- Read the next assembly language command
- Break it into its different fields
- Lookup the binary code for every field
- Combine these codes into a single machine language command
- Output the machine language command
- Until EOF reached

```
A: @<number or variable>
C: dest = comp ; jump
```

@4 D = 0 M = D	000 0000000 000 100 111 0101010 010 000 111 1001100 001 000

Hack language specification: symbols

Pre-defined symbols:

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

Label declaration: (label)

Variable declaration: @variableName

Translating A-instructions

Symbolic syntax:

```
@value
```

Examples:

```
@21
```

```
@foo
```

Where value is either
- a non-negative decimal constant or
- a symbol referring to such a constant (later)

Binary syntax:

```
0valueInBinary
```

Example:

```
0000000000001010
1
```

Translation to binary:

- If value is a decimal constant, generate the equivalent binary constant
- If value is a symbol, later.

Translating 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
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

Example:

Symbolic: `MD=D+1`

Binary: `1110011111011000`

The overall assembly logic

Assembly program

```
@16
M=1
@17
M=0
@16
D=M
@0
D=D-M
@18
D;JGT
@16
D=M
@17
M=D+M
@16
M=M+1
@4
0;JMP
@17
D=M
@1
M=D
@22
0;JMP
```

For each instruction

- Parse the instruction:
  break it into its underlying fields
- A-instruction:
  translate the decimal value into a binary value
- C-instruction:
  for each field in the instruction, generate the corresponding binary code;
- Assemble the translated binary codes into a complete 16-bit machine instruction
- Write the 16-bit instruction to the output file.

The overall assembly logic

Assembly program

```
@16
M=1
@17
M=0
@16
D=M
@0
D=D-M
@18
D;JGT
@16
D=M
@17
M=D+M
@16
M=M+1
@4
0;JMP
@17
D=M
@1
M=D
@22
0;JMP
```

Hack machine code

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1111000010001000
0000000000010000
1111110111001000
0000000000000100
1110101010000111
0000000000010001
1111110000010000
0000000000000001
1110001100001000
0000000000010110
1110101010000111
```

Disclaimer

The source code contains no symbols

Building an Assembler

Assembler: lecture plan

- The assembly process
- The Hack assembly language
- The assembly process: instructions
- The assembly process: symbols
- Developing an assembler
- Project 6 overview

Hack Assembler

Assembly program

```
// Computes RAM[1] = 1 + ... + RAM[0]
    @i
    M=1   // i = 1
    @sum
    M=0   // sum = 0

(LOOP)
    @i    // if i>RAM[0] goto STOP
    D=M
    @R0
    D=D-M
    @STOP
    D;JGT
    @i    // sum += i
    D=M
    @sum
    M=D+M
    @i    // i++
    M=M+1
    @LOOP // goto LOOP
    0;JMP
(STOP)
    @sum
    D=M
    @R1
    M=D  // RAM[1] = the sum
(END)
    @END
    0;JMP
```

Assembler

Challenges:

Handling…

- ✓ White space
- ✓ Instructions
- Symbols

Hack machine code

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1111000010001000
0000000000010000
1111110111001000
0000000000000100
1110101010000111
0000000000010001
1111110000010000
0000000000000001
1110001100001000
0000000000010110
1110101010000111
```

Handling symbols

Assembly program

```
// Computes RAM[1] = 1 + ... + RAM[0]
    @i
    M=1   // i = 1
    @sum
    M=0   // sum = 0

(LOOP)
    @i    // if i>RAM[0] goto STOP
    D=M
    @R0
    D=D-M
    @STOP
    D;JGT
    @i    // sum += i
    D=M
    @sum
    M=D+M
    @i    // i++
    M=M+1
    @LOOP // goto LOOP
    0;JMP
(STOP)
    @sum
    D=M
    @R1
    M=D  // RAM[1] = the sum
(END)
    @END
    0;JMP
```

Pre-defined symbols:
represent special memory locations

label symbols:
represent destinations of goto instructions

variable symbols:
represent memory locations where the programmer wants to maintain values

Handling pre-defined symbols

Assembly program

```
// Computes RAM[1] = 1 + ... + RAM[0]
    @i
    M=1   // i = 1
    @sum
    M=0   // sum = 0

(LOOP)
    @i    // if i>RAM[0] goto STOP
    D=M
    @R0
    D=D-M
    @STOP
    D;JGT
    @i    // sum += i
    D=M
    @sum
    M=D+M
    @i    // i++
    M=M+1
    @LOOP // goto LOOP
    0;JMP
(STOP)
    @sum
    D=M
    @R1
    M=D  // RAM[1] = the sum
(END)
    @END
    0;JMP
```

The Hack language specification describes 23 pre-defined 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

Translating @preDefinedSymbol :

Replace preDefinedSymbol with its value.

Handling symbols that denote labels

Assembly program

```
0   // Computes RAM[1] = 1 + ... + RAM[0]
1       @i
2       M=1   // i = 1
3       @sum
        M=0   // sum = 0

    (LOOP)
4       @i    // if i>RAM[0] goto STOP
5       D=M
6       @R0
7       D=D-M
8       @STOP
9       D;JGT
10      @i    // sum += i
11      D=M
12      @sum
13      M=D+M
14      @i    // i++
15      M=M+1
16      @LOOP // goto LOOP
17      0;JMP
    (STOP)
18      @sum
19      D=M
20      @R1
21      M=D  // RAM[1] = the sum
    (END)
22      @END
23      0;JMP
```

Label symbols

- Used to label destinations of goto commands
- Declared by the pseudo-command (XXX)
- This directive defines the symbol XXX to refer to the memory location holding the next instruction in the program

symbol	value
LOOP	4
STOP	18
END	22

Translating @labelSymbol :

Replace labelSymbol with its value

Handling symbols that denote variables

Assembly program

```
    // Computes RAM[1] = 1 + ... + RAM[0]
        @i
        M=1   // i = 1
        @sum
        M=0   // sum = 0

    (LOOP)
        @i    // if i>RAM[0] goto STOP
        D=M
        @R0
        D=D-M
        @STOP
        D;JGT
        @i    // sum += i
        D=M
        @sum
        M=D+M
        @i    // i++
        M=M+1
        @LOOP // goto LOOP
        0;JMP
    (STOP)
        @sum
        D=M
        @R1
        M=D  // RAM[1] = the sum
    (END)
        @END
        0;JMP
```

Variable symbols

- Any symbol XXX appearing in an assembly program which is not pre-defined and is not defined elsewhere using the (XXX) directive is treated as a variable
- Each variable is assigned a unique memory address, starting at 16

symbol	value
i	16
sum	17

Translating `@variableSymbol`

- If seen for the first time, assign a unique memory address
- Replace variableSymbol with this address

Symbol table

Assembly program

```
// Computes RAM[1] = 1 + ... + RAM[0]
    @i
    M=1   // i = 1
    @sum
    M=0   // sum = 0

(LOOP)
    @i    // if i>RAM[0] goto STOP
    D=M
    @R0
    D=D-M
    @STOP
    D;JGT
    @i    // sum += i
    D=M
    @sum
    M=D+M
    @i    // i++
    M=M+1
    @LOOP // goto LOOP
    0;JMP
(STOP)
    @sum
    D=M
    @R1
    M=D  // RAM[1] = the sum
(END)
    @END
    0;JMP
```

Symbol table

symbol	value
R0	0
R1	1
R2	2
...	...
R15	15
SCREEN	16384
KBD	24576
SP	0
LCL	1
ARG	2
THIS	3
THAT	4
LOOP	4
STOP	18
END	22
i	16
sum	17

Initialization: Add the pre-defined symbols

First pass: Add the label symbols

Second pass: Add the var. symbols

Usage:

To resolve a symbol, look up its value in the symbol table

The assembly process

Initialization:

- Construct an empty symbol table
- Add the pre-defined symbols to the symbol table

First pass:

Scan the entire program;
For each "instruction" of the form `(xxx)`:

- Add the pair (xxx, *address*) to the symbol table, where *address* is the number of the instruction following `(xxx)`

Second pass:

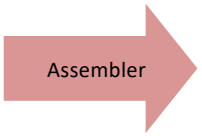
Set *n* to 16
Scan the entire program again; for each instruction:

- If the instruction is *@symbol*, look up *symbol* in the symbol table;
  - If (*symbol, value*) is found, use *value* to complete the instruction's translation;
  - If not found:
    - Add (*symbol*, *n*) to the symbol table,
    - Use *n* to complete the instruction's translation,
    - *n*++
- If the instruction is a C-instruction, complete the instruction's translation
- Write the translated instruction to the output file.

Hack Assembler

Assembly program

```
// Computes RAM[1] = 1 + ... + RAM[0]
    @i
    M=1   // i = 1
    @sum
    M=0   // sum = 0

(LOOP)
    @i    // if i>RAM[0] goto STOP
    D=M
    @R0
    D=D-M
    @STOP
    D;JGT
    @i    // sum += i
    D=M
    @sum
    M=D+M
    @i    // i++
    M=M+1
    @LOOP // goto LOOP
    0;JMP
(STOP)
    @sum
    D=M
    @R1
    M=D  // RAM[1] = the sum
(END)
    @END
    0;JMP
```



Hack machine code

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1111000010001000
0000000000010000
1111110111001000
0000000000000100
1110101010000111
0000000000010001
1111110000010000
0000000000000001
1110001100001000
0000000000010110
1110101010000111
```

Challenges:

Handling…

- ✓ White space
- ✓ Instructions
- ✓ Symbols

Assembler: lecture plan

- The assembly process
- The Hack assembly language
- The assembly process: instructions
- The assembly process: symbols
- Developing an assembler
- Project 6 overview

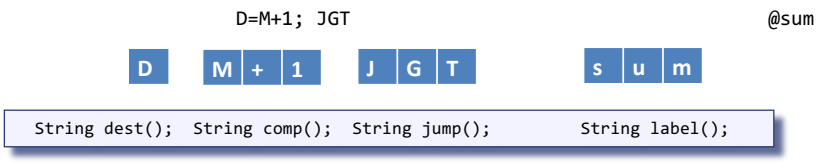
Reading and Parsing Commands

1. Start reading a file with a given name
   - E.g. Constructor for a **Parser** object that accepts a string specifying a file name.
   - Need to know how to read text files

2. Move to the next command in the file
   - Are we finished? `boolean hasMoreCommands()`
   - Get the next command: `void advance()`
   - Need to read one line at a time
   - Need to skip whitespace including comments

Reading and Parsing Commands

3. Get the fields of the current command
   - Type of current command (A-Command, C-Command, or Label)
   - Easy access to the fields:



Translating Mnemonic to Code: overview

Symbolic syntax: `dest = comp ; jump`

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

Translating Mnemonic to Code: destination

Symbolic syntax: `dest = comp ; jump`

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

dest	d1	d2	d3
null	0	0	0
M	0	0	1
D	0	1	0
MD	0	1	1
A	1	0	0
AM	1	0	1
AD	1	1	0
AMD	1	1	1

Translating Mnemonic to Code: jump

Symbolic syntax: dest = comp ; jump

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

jump	j1	j2	j3
null	0	0	0
JGT	0	0	1
JEQ	0	1	0
JGE	0	1	1
JLT	1	0	0
JNE	1	0	1
JLE	1	1	0
JMP	1	1	1

Translating Mnemonic to Code: computation

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						

Recap: Parsing + Translating

Symbolic syntax: `dest = comp ; jump`

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

```
// Assume that current command is
//      D = M+1; JGT

String c=parser.comp();  // "M+1"
String d=parser.dest();  // "D"
String j=parser.jump();  // "JGT"

String cc = Code.comp(c);  // "1110111"
String dd = Code.dest(d);  // "010"
String jj = Code.jump(j);  // "001"

String out = "111" + cc + dd + jj;
```


The Symbol Table

Symbol	Address
loop	73
sum	12

No need to worry about what these symbols mean

The Symbol Table

Symbol	Address
loop	73
sum	12

- Create an new empty table
- Add a (*symbol* , *address*) pair to the table
- Does the table contain a given symbol?
- What is the address associated with a given symbol?

Using the Symbol Table

- Create a new empty table
- Add all the pre-defined symbols to the table
- While reading the input, add labels and new variables to the table
- Whenever you see a "@*xxx*" command, where *xxx* is not a number,
  consult the table to replace the symbol *xxx* with its address.

Using the Symbol Table: adding symbols

While reading the input, add labels and new variables to the table

- Labels: when you see a "(*xxx*)" command, add the symbol *xxx* and the address of the next machine language command
  - Comment 1: this requires maintaining this running address
  - Comment 2: this may need to be done in a first pass

- Variables: when you see an "@*xxx*" command, where *xxx* is not a number and not already in the table, add the symbol *xxx* and the next free address for variable allocation

Overall logic

- Initialization
  - Of Parser
  - Of Symbol Table
- First Pass: Read all commands, only paying attention to labels and updating the symbol table
- Restart reading and translating commands
- Main Loop:
  - Get the next Assembly Language Command and parse it
  - For A-commands: Translate symbols to binary addresses
  - For C-commands: get code for each part and put them together
  - Output the resulting machine language command

Parser module: proposed API

Routine	Arguments	Returns	Function
Constructor / initializer	Input file or stream	—	Opens the input file/stream and gets ready to parse it.
`hasMoreCommands`	—	boolean	Are there more lines in the input?
`advance`	—	—	• Reads the next command from the input, and makes it the current command. • Takes care of whitespace, if necessary. • Should be called only if `hasMoreCommands()` is true. • Initially there is no current command.
`commandType`	—	`A_COMMAND`, `C_COMMAND`, `L_COMMAND`	Returns the type of the current command: `A_COMMAND` for @*xxx* where *xxx* is either a symbol or a decimal number `C_COMMAND` for *dest* = *comp* ; *jump* `L_COMMAND` for (*xxx*) where *xxx* is a symbol.
`symbol`	—	string	• Returns the symbol or decimal *xxx* of the current command @*xxx* or (*xxx*). • Should be called only when `commandType()` is `A_COMMAND` or `L_COMMAND`.
`dest`	—	string	• Returns the *dest* mnemonic in the current *C*-command (8 possibilities). • Should be called only when `commandType()` is `C_COMMAND`.
`comp`	—	string	• Returns the *comp* mnemonic in the current *C*-command (28 possibilities). • Should be called only when `commandType()` is `C_COMMAND`.
`jump`	—	string	• Returns the jump mnemonic in the current *C*-command (8 possibilities). • Should be called only when `commandType()` is `C_COMMAND`.

Code module: proposed API

Routine	Arguments	Returns	Function
`dest`	mnemonic (string)	3 bits	Returns the binary code of the *dest* mnemonic.
`comp`	mnemonic (string)	7 bits	Returns the binary code of the *comp* mnemonic.
`jump`	mnemonic (string)	3 bits	Returns the binary code of the *jump* mnemonic.

SymbolTable module: proposed API

Routine	Arguments	Returns	Function
Constructor	—	—	Creates a new empty symbol table.
`addEntry`	symbol (string), address (int)	—	Adds the pair (symbol, address) to the table.
`contains`	symbol (string)	boolean	Does the symbol table contain the given symbol?
`getAddress`	symbol (string)	integer	Returns the address associated with the symbol.

Assembler: lecture plan

- The assembly process
- The Hack assembly language
- The assembly process: instructions
- The assembly process: symbols
- Developing an assembler
- Project 6 overview

Developing a Hack Assembler

Contract

Develop an *assembler* that translates Hack assembly programs into executable Hack binary code

The source program is supplied in a text file named Xxx.asm

The generated code is written into a text file named Xxx.hack

Assumption: Xxx.asm is error-free

Usage

prompt > java HackAssembler Xxx.asm

This command should create a new Xxx.hack file that can be executed as-is on the Hack computer.

Proposed design

The assembler can be implemented in any high-level language

Proposed software design

- Parser: unpacks each instruction into its underlying fields
- Code: translates each field into its corresponding binary value
- SymbolTable: manages the symbol table
- Main: initializes I/O files and drives the process.

Proposed Implementation

Staged development

- Develop a basic assembler that can translate assembly programs without symbols
- Develop an ability to handle symbols
- Morph the basic assembler into an assembler that can translate any assembly program

Supplied test programs

- Add.asm
- Max.asm
- Rectangle.asm
- Pong.asm

Test program: Add

Add.asm

```
// Computes RAM[0] = 2 + 3

@2
D=A
@3
D=D+A
@0
M=D
```

Basic test of handling:

- White space
- Instructions

Test program: Max

Max.asm

```
// Computes RAM[2] = max(RAM[0],RAM[1])

   @R0
   D=M            // D = RAM[0]
   @R1
   D=D-M          // D = RAM[0] - RAM[1]
   @OUTPUT_RAM0
   D;JGT          // if D>0 goto output R

   // Output RAM[1]
   @R1
   D=M
   @R2
   M=D            // RAM[2] = RAM[1]
   @END
   0;JMP

(OUTPUT_RAM0)
   @R0
   D=M
   @R2
   M=D            // RAM[2] = RAM[0]

(END)
   @END
   0;JMP
```

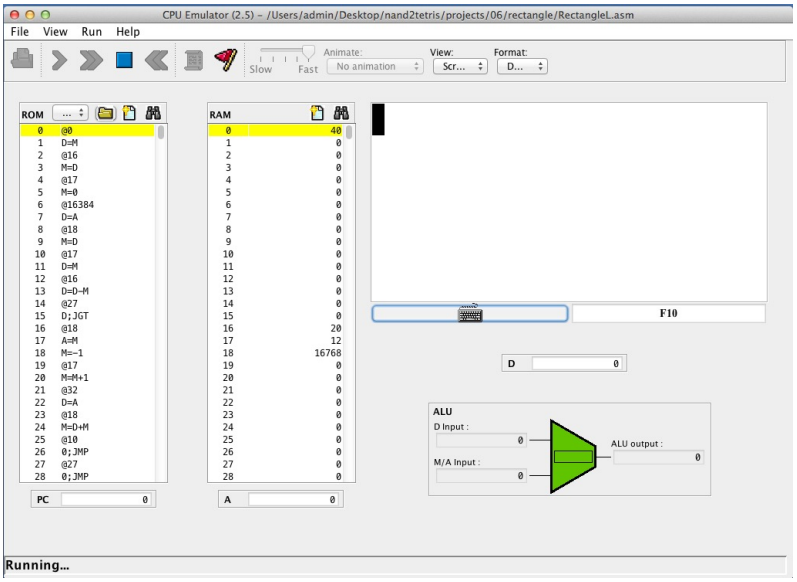
with labels

MaxL.asm

```
// Symbol-less version

   @0
   D=M            // D = RAM[0]
   @1
   D=D-M          // D = RAM[0] - RAM[1]
   @12
   D;JGT          // if D>0 goto output RAM[0]

   // Output RAM[1]
   @1
   D=M
   @2
   M=D            // RAM[2] = RAM[1]
   @16
   0;JMP

   @0
   D=M
   @2
   M=D            // RAM[2] = RAM[0]

   @16
   0;JMP
```

without labels

Test program: Rectangle



Test program: Rectangle

Rectangle.asm

```
// Rectangle.asm

   @R0
   D=M
   @n
   M=D  // n = RAM[0]

   @i
   M=0  // i = 0

   @SCREEN
   D=A
   @address
   M=D  // base address of the Hack scre

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

with symbols

RectangleL.asm

```
// Symbol-less version

   @0
   D=M
   @16
   M=D   // n = RAM[0]

   @17
   M=0   // i = 0

   @16384
   D=A
   @18
   M=D   // base address of the Hack screen

   @17
   D=M
   @16
   D=D-M
   @27
   D;JGT  // if i>n goto END
   ...
```

without symbols

Test program: Pong

Pong.asm

```
// Pong.asm
@256
D=A
@SP
M=D
@133
0;JMP
@R15
M=D
@SP
AM=M-1
D=M
A=A-1
D=M-D
M=0
@END_EQ
D;JNE
@SP
A=M-1
M=-1
(END_EQ)
@R15
A=M
0;JMP
@R15
M=D
...
```

Observations:

- Source code originally written in the Jack language
- The Hack code was generated by the Jack compiler and the Hack assembler
- The resulting code is 28,374 instructions long (includes the Jack OS)

Machine generated code:

- No white space
- “Strange” addresses
- “Strange” labels
- “Strange” pre-defined symbols

Testing options

Use your assembler to translate Xxx.asm, generating the executable file Xxx.hack

Hardware simulator:
load Xxx.hack into the Hack Computer chip, then execute it

CPU Emulator:
load Xxx.hack into the supplied CPUEmulator, then execute it

Assembler:
use the supplied Assembler to translate Xxx.asm;
Compare the resulting code to the binary code generated by your assembler.

Testing your assembler using the supplied assembler