



University of
Nottingham

UK | CHINA | MALAYSIA

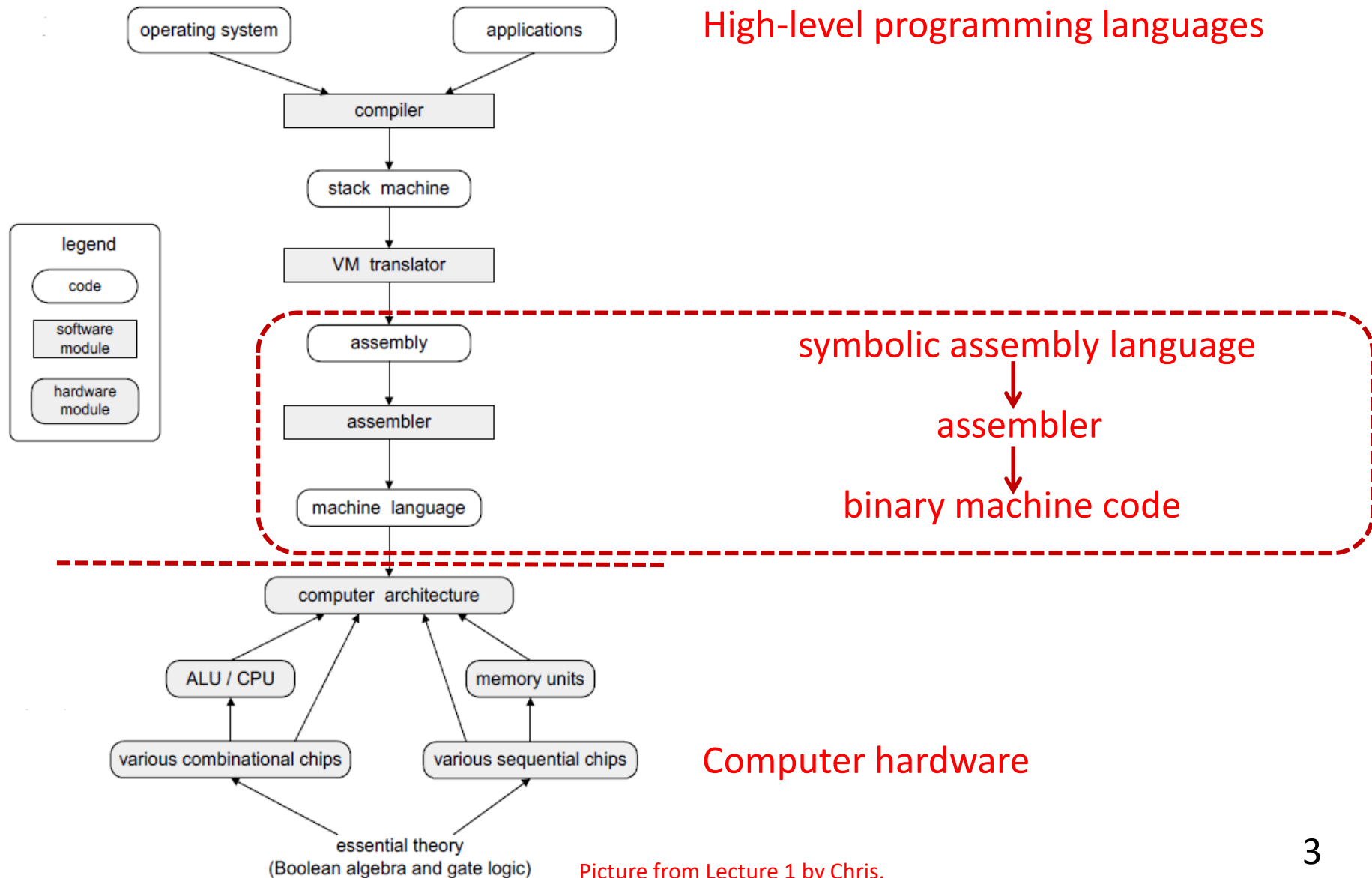
Assembler

Dr. Ren Jianfeng

Outlines

- Introduction to assembler
- Translate Hack assembly program
 - Translate program without symbols
 - Translate program with symbols
- Develop an assembler

Overview of computer system



The translator's challenge (overview)

Hack assembly code (source language)

```
// 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
...
```

Assembler



What are the rules of
the game?

Hack binary code (target language)

```
00000000000010000
1110111111001000
00000000000010001
1110101010001000
00000000000010000
11111100000010000
00000000000000000
1111010011010000
00000000000010010
11100011000000001
00000000000010000
11111100000010000
00000000000010001
1111000010001000
00000000000010000
1111110111001000
00000000000000100
1110101010000111
...
```

The translator's challenge (overview)

Hack assembly code (source language)

```
// 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
...
```

Assembler



Based on the syntax
rules of:

- The source language
- The target language

Hack binary code (target language)

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1111000010001000
0000000000010000
1111110111001000
00000000000000100
1110101010000111
...
```

Hack language specification: A-instruction

Symbolic syntax:

@ value

Where *value* is either

- a non-negative decimal constant or
- a symbol referring to such a constant

Examples:

@ 21

@foo

Binary syntax:

0 valueInBinary

Example:

0000000000010101

Hack language specification: C-instruction

Symbolic syntax:

dest = *comp* ; *jump*

Binary syntax:

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

opcode

not used

comp bits

dest bits

jump bits

<i>comp</i>		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						

<i>dest</i>	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

<i>jump</i>	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

Hack language specification: symbols

Pre-defined symbols:

<u>symbol</u>	<u>value</u>	<u>symbol</u>	<u>value</u>
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*

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

(LOOP) //label
@i // if i>RAM[0] goto STOP
D=M
@R0 //built-in symbols
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
...
```


The Hack language: a translator's perspective

Assembly program

```
// Compute RAM[1] = 1+2+ ... +n
// Usage: put a number (n) in RAM[0]
  @R0
  D=M
  @n
  M=D // n = R0
  @i
  M=1 // i = 1
  @sum
  M=0 // sum = 0
(LOOP)
  @i
  D=M // D = i
  @n
  D=D-M // D = i - n
  @STOP
  D;JGT // if i > n goto STOP
  @i
  D=M // D = i
  @sum
  M=D+M // sum = sum + i
  @i
  M=M+1 // i = i + 1
  @LOOP
  0;JMP // goto to LOOP
(STOP)
  @sum
  D=M // D = sum
  @R1
  M=D // RAM[1] = sum
(END)
  @END
  0;JMP // end
```

Assembly program elements:

- White space
 - Empty lines / indentation
 - Line comments
 - In-line comments
- Instructions
 - A-instructions
 - C-instructions
- Symbols
 - Predefined symbols
 - Variables
 - Labels

Outlines

- Introduction to assembler
- Translate Hack assembly program
 - Translate program without symbols
 - Translate program with symbols
- Develop an assembler

Handling programs without symbols

Assembly program (without symbols)

```
0      // Computes RAM[1] = 1 + ... +
1      RAM[0]
2      @16
3      M=1 // i = 1
4      @17
5      M=0 // sum = 0
6
7      @16 // if i>RAM[0] goto STOP
8      D=M
9      @0
10     D=D-M
11     @18
12     D;JGT
13     @16 // sum += i
14     D=M
15     @17
16     M=D+M
17     @16 // i++
18     M=M+1
19     @4 // goto LOOP
20     0;JMP
21     @17
22     D=M
23     @1
24     M=D // RAM[1] = the sum
25     @22
26     0;JMP
```

Assembler
for symbol-less
Hack programs

Challenges:

Handling...

- White space
- Instructions

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 programs without symbols

Assembly program (without symbols)

```
0  @16
1  M=1
2  @17
3  M=0
4  @16
5  D=M
6  @0
7  D=D-M
8  @18
9  D;JGT
10 @16
11 D=M
12 @17
13 M=D+M
14 @16
15 M=M+1
16 @4
17 0;JMP
18 @17
19 D=M
20 @1
21 M=D
22 @22
23 0;JMP
```

Assembler
for symbol-less
Hack programs

Challenges:

Handling...

- White space
 - Ignore it
- Instructions

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
```

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

Binary syntax:

0 *valueInBinary*

Example:

00000000000010101

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

<i>comp</i>		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						

<i>dest</i>	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

<i>jump</i>	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:

Example:

MD = D+1

1 1 1 0 0 1 1 1 1 1 0 1 1 0 0 0

Exercise: translate assembly code

- Translate the following assembly code to binary code:

@10

D=A

@R10

M=D

(END)

@END

0;JMP

Binary syntax:

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

<i>comp</i>		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						

<i>dest</i>	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

<i>jump</i>	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

Exercise: translate assembly code

- Translate the following assembly code to binary code:

@10 00000000000001010

D=A 1110110000010000

@R10 00000000000001010

M=D 1110001100001000

(END)

@END 0000000000000100

0;JMP 1110101010000111

Thinking & self-study

- D register appears in 'dest' or 'comp', the binary codes are different.
- **Label declaration such as '(END)' does not account as one instruction!**
- Why 7 bits for 'comp', whereas only 3 bits for 'dest' and 'jump'?
- @value operation often requires **conversion from decimal number to binary number**. Practice it by yourself.
- Manual translation from assembly code to binary code is tedious. Practice it by yourself.

The overall assembler logic

Assembly program (without symbols)

```
0  @16
1  M=1
2  @17
3  M=0
4  @16
5  D=M
6  @0
7  D=D-M
8  @18
9  D;JGT
10 @16
11 D=M
12 @17
13 M=D+M
14 @16
15 M=M+1
16 @4
17 0;JMP
18 @17
19 D=M
20 @1
21 M=D
22 @22
23 0;JMP
```

For each instruction

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

Outlines

- Introduction to assembler
- Translate Hack assembly program
 - Translate program without symbols
 - Translate program with symbols
- Develop an assembler

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*:

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

Translating @preDefinedSymbol :

Replace *preDefinedSymbol* with its value.

Examples

Symbolic:

@R0

Binary:

00000000000000000000 21

Handling label symbols

Assembly program

```
0 // Computes RAM[1] = 1 + ... + RAM[0]
1 @i
2 M=1 // i = 1
3 @sum
4 M=0 // sum = 0
5 (LOOP)
6 @i // if i>RAM[0] goto STOP
7 D=M
8 @R0
9 D=D-M
10 @STOP
11 D;JGT
12 @i // sum += i
13 D=M
14 @sum
15 M=D+M
16 @i // i++
17 M=M+1
18 @LOOP // goto LOOP
19 0;JMP
20 (STOP)
21 @sum
22 D=M
23 @R1
24 M=D // RAM[1] = the sum
25 (END)
26 @END
27 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.

Examples

Symbolic:

@LOOP

Binary:

0000000000000000100 22

Handling variable symbols

Assembly program

```
0 // Computes RAM[1] = 1 + ... + RAM[0]
1 @i
2 M=1 // i = 1
3 @sum
4 M=0 // sum = 0
5
6 (LOOP)
7 @i // if i>RAM[0] goto STOP
8 D=M
9 @R0
10 D=D-M
11 @STOP
12 D;JGT
13 @i // sum += i
14 D=M
15 @sum
16 M=D+M
17 @i // i++
18 M=M+1
19 @LOOP // goto LOOP
20 0;JMP
21 (STOP)
22 @sum
23 D=M
24 @R1
25 M=D // RAM[1] = the sum
26 (END)
27 @END
28 0;JMP
```

Variable symbols

- A symbol, not pre-defined, nor defined elsewhere as a label, then it is a *variable*.
- Each variable is assigned a unique memory address, starting at **16**.

<u>symbol</u>	<u>value</u>
---------------	--------------

i	16
sum	17

Translating @variableSymbol :

- First time see it, assign a unique memory address.
- Replace *variableSymbol* with this address.

Examples

Symbolic:

@i

Binary:

00000000000010000

23

Symbol table

Assembly program

```
0  // Computes RAM[1] = 1 + ... + RAM[0]
1  @i
2  M=1  // i = 1
3  @sum
4  M=0  // sum = 0
5
6  (LOOP)
7  @i  // if i>RAM[0] goto STOP
8  D=M
9  @R0
10 D=D-M
11 @STOP
12 D;JGT
13 @i  // sum += i
14 D=M
15 @sum
16 M=D+M
17 @i  // i++
18 M=M+1
19 @LOOP // goto LOOP
20 0;JMP
21 (STOP)
22 @sum
23 D=M
24 @R1
25 M=D  // RAM[1] = the sum
26 (END)
27 @END
28 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: handle predefined symbols.
 - Construct an empty symbol table.
 - Add the pre-defined symbols to the symbol table.
- First pass: handle label symbols.
 - 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: handle variable symbols and instructions

The assembly process - second pass

- Set n to 16
- Scan the entire program again; for each instruction:
 - If it is *@symbol*, look up *symbol* in symbol table;
 - ❑ If (*symbol*, *value*) is found, use *value* to complete instruction's translation;
 - ❑ If not found:
 - ✓ Add (*symbol*, n) to the symbol table,
 - ✓ Use n to complete instruction's translation,
 - ✓ $n++$
 - If it is a C-instruction, complete instruction's translation
 - Write translated instruction to output file.

Outlines

- Introduction to assembler
- Translate Hack assembly program
 - Translate program without symbols
 - Translate program with symbols
- Develop an assembler

Develop an assembler

- **Reading** and **parsing** commands.
- **Converting** mnemonics to codes.
- **Handling** symbols.

Assumption: The
symbolic code is
error-free.

Read and parse commands

- Start reading a file with a given name.
 - Constructor for a **Parser** object that accepts a string specifying a file name.
 - Handle reading text files.
- Move to the next command in the file
 - Are we finished? `boolean hasMoreCommands()`.
 - Get the next command: `void advance()`.
 - Read one line at a time.
 - Skip whitespace including comments.
- Get the fields of the current command.
 - ... more on next slide

Read and parse commands

- Get the fields of the current command.
 - Type of current command (A-Command, C-Command, or Label)
 - Handle fields as strings:

D=M+1; JGT

D	M	+	1	J	G	T
---	---	---	---	---	---	---

@sum

s	u	m
---	---	---

String dest(); String comp(); String jump();

String variable();

Translate mnemonic to code

Symbolic syntax:

dest = *comp* ; *jump*

Binary syntax:

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

<i>comp</i>		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
ID		0	0	1	1	0	1
IA	IM	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						

<i>dest</i>	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

<i>jump</i>	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

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;
```

- Parser
 - Parse the command line in symbolic syntax.
- Coder
 - Generate binary code according to binary syntax.

The symbol table

- Create a new empty table,
- Add all the **pre-defined symbols** to the table,
- While reading input, add labels and variables to table.
 - **Labels:** for “(xxx)” command, add the symbol xxx and the address of the next machine language command.
 - ❑ Maintain this running address,
 - ❑ Do it in the first pass.
 - **Variables:** for “@xxx” command, if xxx not a number, nor in the table, add symbol xxx and the next free address to table.
- For “@xxx” command, if xxx not a number, consult the table to replace symbol xxx with its address.

Overall logic

- Initialization
 - Of *Parser*,
 - Of *Symbol Table*, e.g. add built-in symbols.
- First pass: read all commands, pay attention to *labels* and update the *Symbol Table*,
- Second pass: restart reading and translating commands, in a main loop, pay attention to *variables*.
 - 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

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	—	—	<ul style="list-style-type: none"> • 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 <i>dest = comp ; jump</i> L_COMMAND for (xxx) where xxx is a symbol.
symbol	—	string	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • Returns the <i>dest</i> mnemonic in the current C-command (8 possibilities). • Should be called only when commandType() is C_COMMAND.
comp	—	string	<ul style="list-style-type: none"> • Returns the <i>comp</i> mnemonic in the current C-command (28 possibilities). • Should be called only when commandType() is C_COMMAND.
jump	—	string	<ul style="list-style-type: none"> • Returns the jump mnemonic in the current C-command (8 possibilities). • Should be called only when commandType() is C_COMMAND.

Code module

Routine	Arguments	Returns	Function
dest	mnemonic (string)	3 bits	Returns the binary code of the <i>dest</i> mnemonic.
comp	mnemonic (string)	7 bits	Returns the binary code of the <i>comp</i> mnemonic.
jump	mnemonic (string)	3 bits	Returns the binary code of the <i>jump</i> mnemonic.

SymbolTable module

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.

Summary

- Introduction to assembler
 - Translate assembly language (symbolic code) to machine language (binary code).
- Assembly process
 - White space, including comments. (Simply ignore it)
 - Instructions. (Parser and Code)
 - Symbols. (SymbolTable)
- Develop an assembler
 - Initialization: Parser and SymbolTable (Predefined symbols)
 - First pass: handle labels.
 - Second pass: handle variables and instructions.

Q & A

Acknowledgement

- This set of lecture notes are based on the lecture notes provided by Noam Nisam / Shimon Schocken.
- You may find more information on:
www.nand2tetris.org.