

## Chapter 6

# Assembler

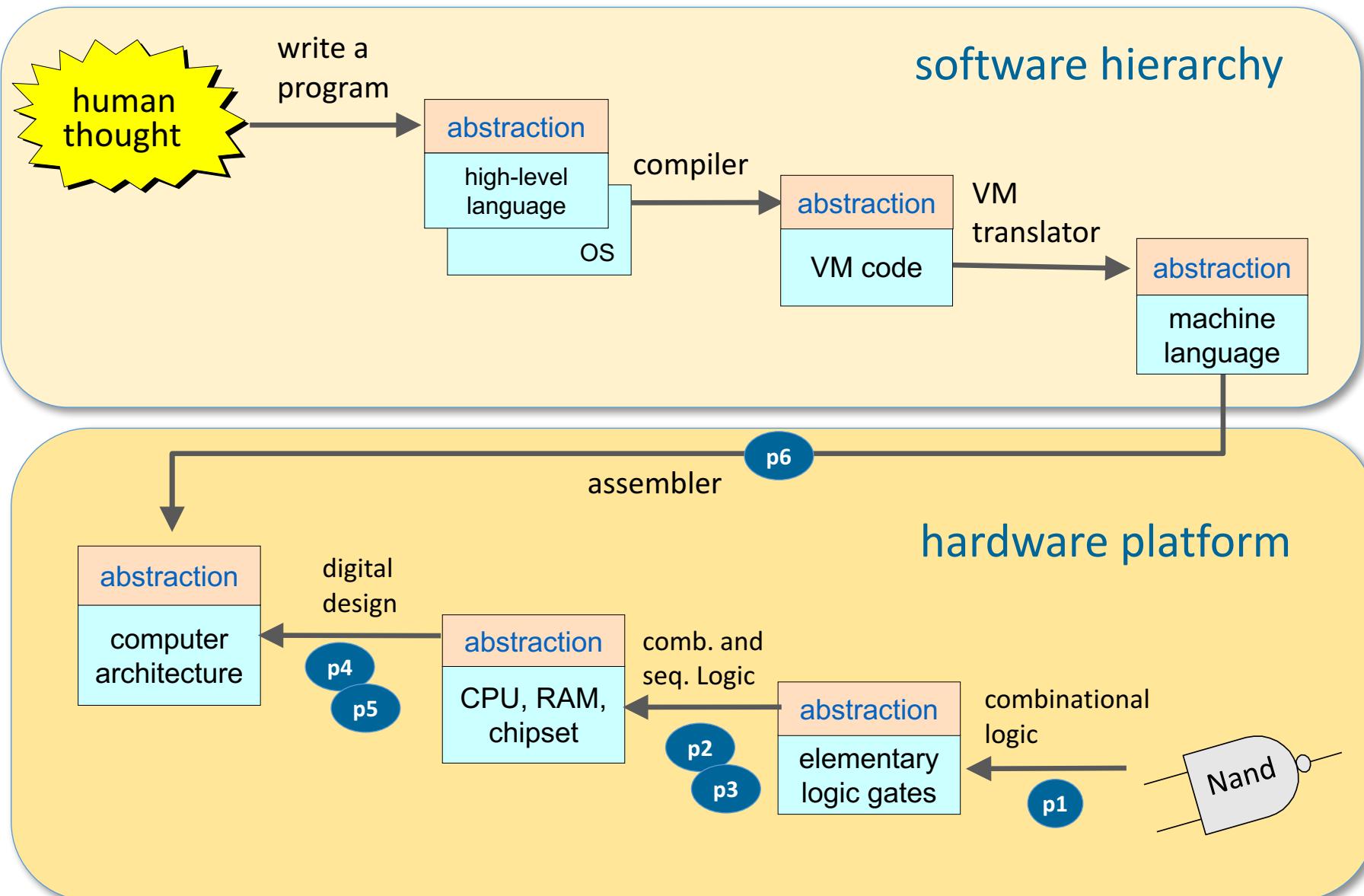
These slides support chapter 6 of the book

*The Elements of Computing Systems*

By Noam Nisan and Shimon Schocken

MIT Press

# Nand to Tetris: the big picture



# Assembly process

Assembly Language

```
@i  
M=1 // i = 1  
@sum  
M=0 // sum = 0  
(LOOP)  
@i // if i>RAM[0]  
D=M // GOTP WRITE  
@R0  
D=D-M  
@WRITE  
D;JGT  
... // Etc.
```

Machine Language

```
000000000010000  
1110111111001000  
0000000000010001  
1110101010001000  
0000000000010000  
1111110000010000  
0000000000000000  
1111010011010000  
0000000000010010  
1110001100000001  
0000000000010000  
1111110000010000  
0000000000010001  
...
```

assembler

run



# Assembler: lecture plan

---



The assembly process



The Hack assembly language

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

# 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

(source language)

```
000000000010000
1110111111001000
0000000000010001
1110101010001000
000000000010000
1111110000010000
0000000000000000
1111010011010000
000000000010010
1110001100000001
000000000010000
1111110000010000
000000000010001
1111000010001000
000000000010000
1111110111001000
0000000000001000
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

## Hack binary code

(source language)

```
000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
000000000010010
1110001100000001
000000000010000
1111110000010000
0000000000010001
1111000010001000
0000000000010000
1111110111001000
0000000000000001
1110101010000111
...
```

Based on the syntax  
rules of:

- The source language
- The target language

# Hack language specification: A-instruction

---

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:

`000000000010101`

# Hack language specification: C-instruction

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

Binary syntax: 1 1 1 *a* *c<sub>1</sub>* *c<sub>2</sub>* *c<sub>3</sub>* *c<sub>4</sub>* *c<sub>5</sub>* *c<sub>6</sub>* *d<sub>1</sub>* *d<sub>2</sub>* *d<sub>3</sub>* *j<sub>1</sub>* *j<sub>2</sub>* *j<sub>3</sub>*

<i>comp</i>		<i>c<sub>1</sub></i>	<i>c<sub>2</sub></i>	<i>c<sub>3</sub></i>	<i>c<sub>4</sub></i>	<i>c<sub>5</sub></i>	<i>c<sub>6</sub></i>
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>	<i>d<sub>1</sub></i>	<i>d<sub>2</sub></i>	<i>d<sub>3</sub></i>	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>	<i>j<sub>1</sub></i>	<i>j<sub>2</sub></i>	<i>j<sub>3</sub></i>	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*

# The Hack language: a translator's perspective

## 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
```

## Assembly program elements:

- White space
  - Empty lines / indentation
  - Line comments
  - In-line comments
- Instructions
  - A-instructions
  - C-instructions
- Symbols
  - References
  - Label declarations

# The Hack language: a translator's perspective

## 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

## Hack machine code

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1111000010001000
0000000000010000
1111110111001000
0000000000000001
1110101010000111
0000000000010001
1111110000010000
0000000000000001
1110001100001000
00000000000010110
1110101010000111
```

## Challenges:

### Handling...

- White space
- Instructions
- Symbols

# Symbols

---

## Program with symbols

```
// 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
```

### Challenges:

Handling...

- White space
- Instructions
- Symbols

### Simplifying assumption:

Let's deal with symbols later.

# Handling programs without symbols

## Assembly program (without symbols)

```
// Computes RAM[1] = 1 + ... + RAM[0]
@16
M=1    // i = 1
@17
M=0    // sum = 0

@16    // if i>RAM[0] goto STOP
D=M
@0
D=D-M
@18
D;JGT
@16    // sum += i
D=M
@17
M=D+M
@16    // i++
M=M+1
@4     // goto LOOP
0;JMP
@17
D=M
@1
M=D    // RAM[1] = the sum
@22
0;JMP
```

Assembler  
for symbol-less  
Hack programs

### Challenges:

Handling...

- White space
- Instructions

## Hack machine code

```
000000000010000
1110111111001000
000000000010001
1110101010001000
000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000001000
1110001100000001
000000000010000
1111110000010000
000000000010001
1111000010001000
000000000010000
1111110111001000
000000000000100
1110101010000111
000000000010001
1111110000010000
0000000000000001
1110001100001000
000000000010110
1110101010000111
```

# Handling white space

Assembly program (without symbols)

```
// Computes RAM[1] = 1 + ... + RAM[0]
@16
M=1    // i = 1
@17
M=0    // sum = 0

@16    // if i>RAM[0] goto STOP
D=M
@0
D=D-M
@18
D;JGT
@16    // sum += i
D=M
@17
M=D+M
@16    // i++
M=M+1
@4     // goto LOOP
0;JMP
@17
D=M
@1
M=D    // RAM[1] = the sum
@22
0;JMP
```

Assembler  
for symbol-less  
Hack programs

Challenges:

Handling...

- White space
- Instructions

Handling white  
space:

Ignore it!

Hack machine code

```
000000000010000
1110111111001000
000000000010001
1110101010001000
000000000010000
1111110000010000
0000000000000000
1111010011010000
000000000010010
1110001100000001
000000000010000
1111110000010000
000000000010001
1111000010001000
000000000010000
1111110000010000
000000000010001
1111000010001000
000000000010000
1111110111001000
000000000000100
1110101010000111
000000000010001
1111110000010000
0000000000000001
1110001100001000
000000000010110
1110101010000111
```

# Handling instructions

Assembly program (without symbols)

```
@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
```

Assembler  
for symbol-less  
Hack programs

Challenges:

Handling...

- ✓ White space
- Instructions

Hack machine code

```
000000000010000  
1110111111001000  
000000000010001  
1110101010001000  
000000000010000  
1111110000010000  
0000000000000000  
1111010011010000  
0000000000010010  
1110001100000001  
0000000000010000  
1111110000010000  
0000000000010001  
1111000010001000  
0000000000010000  
1111110111001000  
000000000000100  
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 (later)

Binary syntax:

`0 valueInBinary`

Example:

`000000000010101`

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 c<sub>1</sub> c<sub>2</sub> c<sub>3</sub> c<sub>4</sub> c<sub>5</sub> c<sub>6</sub> d<sub>1</sub> d<sub>2</sub> d<sub>3</sub> j<sub>1</sub> j<sub>2</sub> j<sub>3</sub>

<i>comp</i>		c <sub>1</sub>	c <sub>2</sub>	c <sub>3</sub>	c <sub>4</sub>	c <sub>5</sub>	c <sub>6</sub>
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>	d <sub>1</sub>	d <sub>2</sub>	d <sub>3</sub>	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>	j <sub>1</sub>	j <sub>2</sub>	j <sub>3</sub>	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:

Example:

MD=D+1

Binary:



# 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:

Example:

MD=D+1

Binary:

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

# 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:

Example:

MD=D+1

Binary:

1 1 1

# 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:

Example:

MD=D+1

Binary:

1 1 1

# 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 $\geq$ 0 jump
JLT	1	0	0	if out < 0 jump
JNE	1	0	1	if out $\neq$ 0 jump
JLE	1	1	0	if out $\leq$ 0 jump
JMP	1	1	1	Unconditional jump

Symbolic:

Example:

MD=D+1

Binary:

1 1 1

# 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 $\geq$ 0 jump
JLT	1	0	0	if out < 0 jump
JNE	1	0	1	if out $\neq$ 0 jump
JLE	1	1	0	if out $\leq$ 0 jump
JMP	1	1	1	Unconditional jump

Symbolic:

Example:

MD=D+1

Binary:

1 1 1

# 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 $\geq$ 0 jump
JLT	1	0	0	if out < 0 jump
JNE	1	0	1	if out $\neq$ 0 jump
JLE	1	1	0	if out $\leq$ 0 jump
JMP	1	1	1	Unconditional jump

Symbolic:

Example:

MD=D+1

Binary:

1110011111

# 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

Symbolic:

Example:

MD=D+1

Binary:

1110011111

# 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

Symbolic:

Example:

MD=D+1

Binary:

1110011111

# 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

Symbolic:

Example:

MD=D+1

Binary:

1110011111

# 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

Symbolic:

Example:

MD=D+1

Binary:

1110011111011

# 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:

Example:

MD=D+1

Binary:

1110011111011

# 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:

Example:

MD=D+1

Binary:

1110011111011

# 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:

Example:

MD=D+1

Binary:

1110011111011

# 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:

Example:

MD=D+1

Binary:

1110011111011000

# Translating C-instructions

Symbolic syntax:

*dest* = *comp* ; *jump*

Binary syntax:

1 1 1 a c<sub>1</sub> c<sub>2</sub> c<sub>3</sub> c<sub>4</sub> c<sub>5</sub> c<sub>6</sub> d<sub>1</sub> d<sub>2</sub> d<sub>3</sub> j<sub>1</sub> j<sub>2</sub> j<sub>3</sub>

<i>comp</i>		c <sub>1</sub>	c <sub>2</sub>	c <sub>3</sub>	c <sub>4</sub>	c <sub>5</sub>	c <sub>6</sub>
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>	d <sub>1</sub>	d <sub>2</sub>	d <sub>3</sub>	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>	j <sub>1</sub>	j <sub>2</sub>	j <sub>3</sub>	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:

Example:

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
```

Resulting code:

Disclaimer

The source code  
contains no symbols

Hack machine code

```
000000000010000  
1110111111001000  
000000000010001  
1110101010001000  
000000000010000  
1111110000010000  
0000000000000000  
1111010011010000  
000000000010010  
1110001100000001  
000000000010000  
1111110000010000  
000000000010001  
1111000010001000  
000000000010000  
1111110111001000  
000000000000100  
1110101010000111  
000000000010001  
1111110000010000  
0000000000000001  
1110001100001000  
000000000010110  
1110101010000111
```

# Assembler: lecture plan

---

-  Assembler logic (basic)
-  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

## Hack machine code

```
000000000010000
1110111111001000
000000000010001
1110101010001000
000000000010000
1111110000010000
0000000000000000
1111010011010000
000000000010010
1110001100000001
000000000010000
1111110000010000
000000000010001
1111000010001000
000000000010000
1111110111001000
0000000000000001
1110101010000111
000000000010001
1111110000010000
0000000000000001
1110001100001000
000000000010110
1110101010000111
```

## Challenges:

Handling...

- ✓ White space
- ✓ Instructions
- Symbols

# 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	symbol	value
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.

# Handling symbols that denote labels

## Assembly program

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

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

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

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

## Assembly program

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

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

### Initialization:

Add the pre-defined symbols

### First pass:

Add the label symbols

# 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
```

Assembler

## Hack machine code

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1111000010001000
0000000000010000
1111110111001000
0000000000000001
1110101010000111
0000000000010001
1111110000010000
0000000000000001
1110001100001000
00000000000010110
1110101010000111
```

## Challenges:

Handling...

- ✓ White space
- ✓ Instructions
- ✓ Symbols

# Assembler: lecture plan

---

- ✓ Assembler logic (basic)
- ✓ The Hack assembly language
- ✓ The assembly process: instructions
- ✓ The assembly process: symbols
- Developing an assembler
  - Project 6 overview

## Sub-tasks that need to be done

---

- Reading and parsing commands
- Converting mnemonics → code
- Handling symbols

# Reading and Parsing Commands

---

No need to understand  
the *meaning* of anything

# Reading and Parsing Commands

---

- 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

# Reading and Parsing Commands

---

- Start reading a file with a given name
- 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

---

- Start reading a file with a given name
- Move to the next command in the file
- Get the fields of the current command
  - Type of current command (A-Command, C-Command, or Label)
  - Easy access to the fields:

D=M+1; JGT

@sum

D

M | + | 1

J | G | T

s | u | m

```
String dest(); String comp(); String jump(); String label();
```

# 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: overview

---

No need to worry about  
how the mnemonic fields  
were obtained

# 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

<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

# 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

<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

# 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

<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						

# 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	—	—	<ul style="list-style-type: none"> <li>Reads the next command from the input, and makes it the current command.</li> <li>Takes care of whitespace, if necessary.</li> <li>Should be called only if <code>hasMoreCommands()</code> is true.</li> <li>Initially there is no current command.</li> </ul>
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	<ul style="list-style-type: none"> <li>Returns the symbol or decimal xxx of the current command @xxx or (xxx).</li> <li>Should be called only when <code>commandType()</code> is A_COMMAND OR L_COMMAND.</li> </ul>
dest	—	string	<ul style="list-style-type: none"> <li>Returns the dest mnemonic in the current C-command (8 possibilities).</li> <li>Should be called only when <code>commandType()</code> is C_COMMAND.</li> </ul>
comp	—	string	<ul style="list-style-type: none"> <li>Returns the comp mnemonic in the current C-command (28 possibilities).</li> <li>Should be called only when <code>commandType()</code> is C_COMMAND.</li> </ul>
jump	—	string	<ul style="list-style-type: none"> <li>Returns the jump mnemonic in the current C-command (8 possibilities).</li> <li>Should be called only when <code>commandType()</code> is C_COMMAND.</li> </ul>

## Code module: proposed API

---

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: 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            MaxL.asm

Rectangle.asm    RectangleL.asm

Pong.asm          PongL.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 RAM[0]

// 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

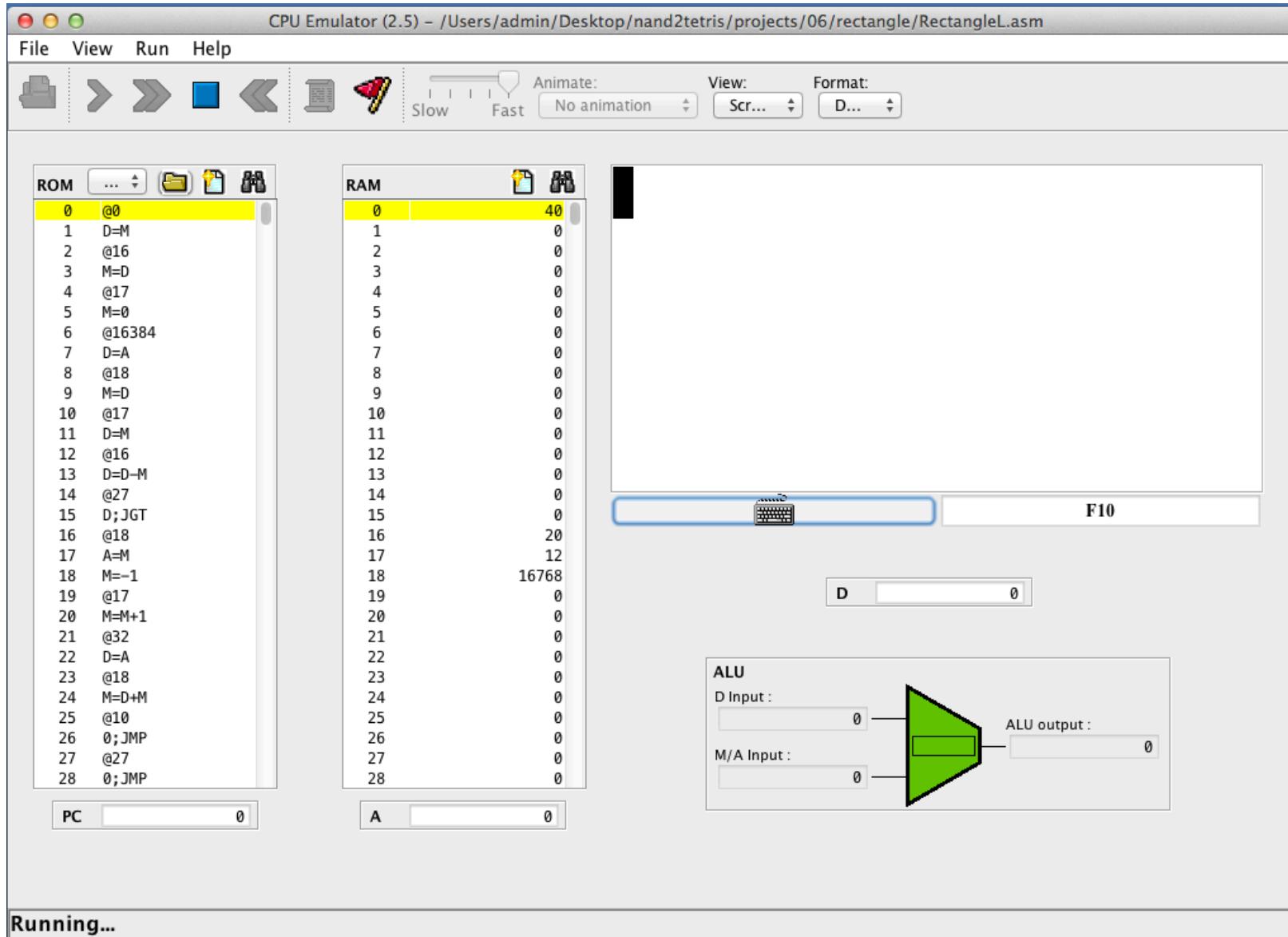
@0
D=M          // D = RAM[0]
@1
D=D-M       // D = RAM[0] - RAM[1]
@12
D;JGT        // if D>0 goto output RAM[0]

// 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 screen

(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

---



# 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

The screenshot shows the Assembler 2.5 application window. The menu bar includes File, Run, and Help. The toolbar contains icons for opening files, saving, running, and comparing. The main window has three panes: Source, Destination, and Comparison.

**Source:**

```
// Computes RAM[1] = 1 + ... + RAM
@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
```

**Destination:**

0000000000010000
1110111111001000
0000000000100001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000001001
1111000010001000
0000000000010000
1111110111001000
0000000000000100
1110101010000111
0000000000010001
1111110000010000
0000000000000001
1110001100001000
0000000000000000
1110101010000110
1110101010000111

**Comparison:**

0000000000010000
1110111111001000
0000000000100001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000001001
1111000010001000
0000000000010000
1111110111001000
0000000000000100
1110101010000111
0000000000010001
1111110000010000
0000000000000001
1110001100001000
0000000000000000
1110101010000110
1110101010000111

**Annotations:**

- A callout bubble points to the Destination pane with the text: "Xxx.hack file, translated by the supplied assembler".
- A callout bubble points to the Comparison pane with the text: "Xxx.hack file, translated by your assembler".
- An orange arrow points from the Source pane to the Destination pane.
- An orange callout bubble points to the Source pane with the text: "Source Xxx.asm file".
- A blue double-equals sign icon is positioned between the Destination and Comparison panes.

**Status Bar:**

File compilation & comparison succeeded

# Project 6 Resources

The screenshot shows the 'Project 6 Resources' page from the [From NAND to Tetris](http://www.nand2tetris.org) website. The page has a dark background with orange highlights for navigation. At the top left, it says 'From NAND to Tetris' and 'Building a Modern Computer From First Principles'. Below that is the website address 'www.nand2tetris.org'. On the right side, there's a small illustration of a character walking towards a Tetris-like stack of colored blocks.

**Project 6: The Assembler**

**Background**

Low-level machine programs are rarely written by humans. Typically, they are generated by compilers. Yet humans can inspect the translated code and learn important lessons about how to write their high-level programs better, in a way that avoids low-level pitfalls and exploits the underlying hardware better. One of the key players in this translation process is the *assembler* -- a program designed to translate code written in a symbolic machine language into code written in binary machine language.

This project marks an exciting landmark in our *Nand to Tetris* odyssey: it deals with building the first rung up the software hierarchy, which will eventually end up in the construction of a compiler for a Java-like high-level language. But, first things first.

**Objective**

Write an Assembler program that translates programs written in the symbolic Hack assembly language into binary code that can execute on the Hack hardware platform built in the previous projects.

**Contract**

There are three ways to describe the desired behavior of your assembler: (i) When loaded into Prog.asm file containing a valid Hack assembly language program should be translated into

All the necessary project 6 files are available in:  
nand2tetris / projects / 06

# More resources

---

- Supplied Assembler
- Supplied CPU emulator
- Assembler Tutorial
- Proposed Assembler API
- nand2tetris Q&A forum

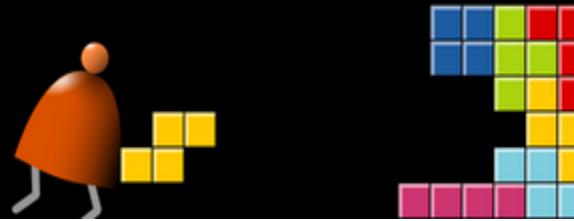


All available in: [www.nand2tetris.org](http://www.nand2tetris.org)

# Assembler: lecture plan

---

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



## Chapter 6

# Assembler

These slides support chapter 6 of the book

*The Elements of Computing Systems*

By Noam Nisan and Shimon Schocken

MIT Press