Chapter 6

Assembler

These slides support chapter 6 of the book

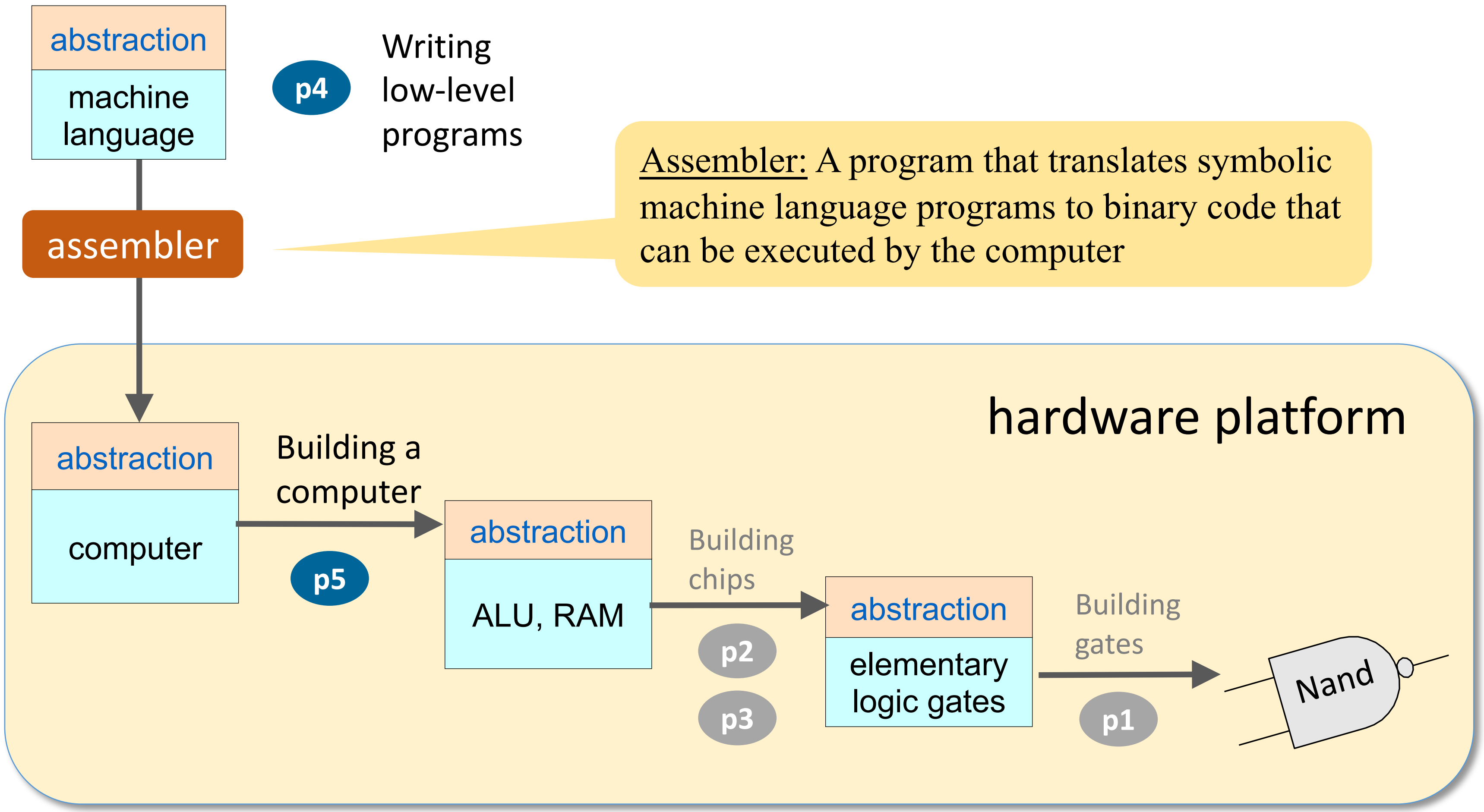
The Elements of Computing Systems

(1st and 2nd editions)

By Noam Nisan and Shimon Schocken

MIT Press

Nand to Tetris Roadmap (Part I: Hardware)



The assembler

Symbolic low-level program

```
// Program: Sum1ToN (R0 represents N)
// Computes R1 = 1 + 2 + 3 + ... + R0
// Usage: put a value >= 1 in R0
   // i = 1
   @i
   M=1
   // sum = 0
   @sum
   M=0
(LOOP)
   // if (i > R0) goto STOP
   @i
   D=M
   @R0
   D=D-M
   @STOP
   D;JGT
   // sum = sum + i
   @sum
   D=M
   @i
   D=D+M
   @sum
   M=D
   // i = i + 1
   @i
   M=M+1
   // goto LOOP
   @LOOP
   0;JMP
   ...
```

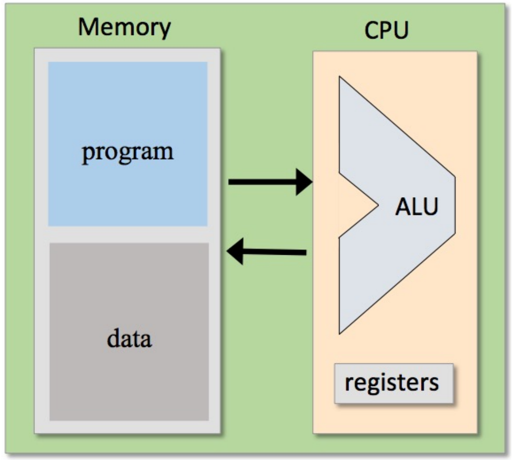
assembler

Binary code

```
0101111100111100
1010101010101010
1100000010101010
1011000010000001
0101111100111100
1010101010101010
1100000010101010
0101111100111100
1010101010101010
1100000010101010
1011000010000001
0101111100111100
1010101010101010
1100000010101010
0101111100111100
1010101010101010
1100000010101010
1011000010000001
0101111100111100
1010101010101010
1100000010101010
...
```

load and execute

Computer



Why write an assembler?

- Because it is the "linchpin" that connects the hardware platform and the software hierarchy that sits on top of it
- Because it provides a simple example of key software engineering techniques (parsing, code generation, symbol tables, ...)

Translating A-instructions

Symbolic syntax:

`@xxx`

translate

Binary syntax:

`0vvvvvvvvvvvvvvv`

Where *xxx* is a non-negative decimal value, or a symbol bound to such a value

Where:

`0` is the A-instruction op-code, and

v v v … v is the value in binary

Example:

`@17`

becomes

`0000000000010001`

Implementation

Simple: Translate the decimal value into its 16-bit representation.

Translating C-instructions

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

Binary syntax: `1 1 1` *a c c c c c c d d d j j j*

comp (a==0)	*comp* (a==1)	*c*	*c*	*c*	*c*	*c*	*c*
0		1	0	1	0	1	0
1		1	1	1	1	1	1
-1		1	1	1	0	1	0
D		0	0	1	1	0	0
A	M	1	1	0	0	0	0
!D		0	0	1	1	0	1
!A	!M	1	1	0	0	0	1
-D		0	0	1	1	1	1
-A	-M	1	1	0	0	1	1
D+1		0	1	1	1	1	1
A+1	M+1	1	1	0	1	1	1
D-1		0	0	1	1	1	0
A-1	M-1	1	1	0	0	1	0
D+A	D+M	0	0	0	0	1	0
D-A	D-M	0	1	0	0	1	1
A-D	M-D	0	0	0	1	1	1
D&A	D&M	0	0	0	0	0	0
D|A	D|M	0	1	0	1	0	1

a == `0` *a* == `1`

dest	*d*	*d*	*d*	effect: the value is stored in:
null	0	0	0	the value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register
DM	0	1	1	D register and RAM[A]
A	1	0	0	A register
AM	1	0	1	A register and RAM[A]
AD	1	1	0	A register and D register
ADM	1	1	1	A register, D register, and RAM[A]

jump	*j*	*j*	*j*	effect:
null	0	0	0	no jump
JGT	0	0	1	if *comp* > 0 jump
JEQ	0	1	0	if *comp* = 0 jump
JGE	0	1	1	if *comp* ≥ 0 jump
JLT	1	0	0	if *comp* < 0 jump
JNE	1	0	1	if *comp* ≠ 0 jump
JLE	1	1	0	if *comp* ≤ 0 jump
JMP	1	1	1	Unconditional jump

Implementation: Simple: Translate each field of the symbolic instruction (*dest*, *comp*, *jump*) into its binary code, and assemble the codes into a 16-bit instruction.

Translating C-instructions

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

Binary syntax: `1 1 1` *a c c c c c c d d d j j j*

comp (a==0)	*comp* (a==1)	*c*	*c*	*c*	*c*	*c*	*c*
0		1	0	1	0	1	0
1		1	1	1	1	1	1
-1		1	1	1	0	1	0
D		0	0	1	1	0	0
A	M	1	1	0	0	0	0
!D		0	0	1	1	0	1
!A	!M	1	1	0	0	0	1
-D		0	0	1	1	1	1
-A	-M	1	1	0	0	1	1
D+1		0	1	1	1	1	1
A+1	M+1	1	1	0	1	1	1
D-1		0	0	1	1	1	0
A-1	M-1	1	1	0	0	1	0
D+A	D+M	0	0	0	0	1	0
D-A	D-M	0	1	0	0	1	1
A-D	M-D	0	0	0	1	1	1
D&A	D&M	0	0	0	0	0	0
D|A	D|M	0	1	0	1	0	1

a==0 *a*==1

dest	*d*	*d*	*d*	effect: the value is stored in:
null	0	0	0	the value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register
DM	0	1	1	D register and RAM[A]
A	1	0	0	A register
AM	1	0	1	A register and RAM[A]
AD	1	1	0	A register and D register
ADM	1	1	1	A register, D register, and RAM[A]

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

Example: `D=D+1;JLE` → Binary: `1110011111010110`

Translating C-instructions

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

Binary syntax: `1 1 1` *a c c c c c c d d d j j j*

comp		*c*	*c*	*c*	*c*	*c*	*c*
0		1	0	1	0	1	0
1		1	1	1	1	1	1
-1		1	1	1	0	1	0
D		0	0	1	1	0	0
A	M	1	1	0	0	0	0
!D		0	0	1	1	0	1
!A	!M	1	1	0	0	0	1
-D		0	0	1	1	1	1
-A	-M	1	1	0	0	1	1
D+1		0	1	1	1	1	1
A+1	M+1	1	1	0	1	1	1
D-1		0	0	1	1	1	0
A-1	M-1	1	1	0	0	1	0
D+A	D+M	0	0	0	0	1	0
D-A	D-M	0	1	0	0	1	1
A-D	M-D	0	0	0	1	1	1
D&A	D&M	0	0	0	0	0	0
D|A	D|M	0	1	0	1	0	1
a==0	*a*==1						

dest	*d*	*d*	*d*	effect: the value is stored in:
null	0	0	0	the value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register
DM	0	1	1	D register and RAM[A]
A	1	0	0	A register
AM	1	0	1	A register and RAM[A]
AD	1	1	0	A register and D register
ADM	1	1	1	A register, D register, and RAM[A]

jump	*j*	*j*	*j*	effect:
null	0	0	0	no jump
JGT	0	0	1	if *comp* > 0 jump
JEQ	0	1	0	if *comp* = 0 jump
JGE	0	1	1	if *comp* ≥ 0 jump
JLT	1	0	0	if *comp* < 0 jump
JNE	1	0	1	if *comp* ≠ 0 jump
JLE	1	1	0	if *comp* ≤ 0 jump
JMP	1	1	1	Unconditional jump

Example: `A=-1` → Binary: `1110111010100000`

Program translation

Symbolic code

```
// Computes R1=1 + ... + R0
    // i = 1
    @i
    M=1
    // sum = 0
    @sum
    M=0
(LOOP)
    // if i>R0 goto STOP
    @i
    D=M
    @R0
    D=D-M
    @STOP
    D;JGT
    // sum += i
    @i
    D=M
    @sum
    M=D+M
    // i++
    @i
    M=M+1
    @LOOP
    0;JMP
(STOP)
    @sum
    D=M
    ...
```

Translate

Binary code

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1111000010001000
0000000000010000
1111110111001000
0000000000000100
1110101010000111
0000000000010001
1111110000010000
...
```

Need to handle:

- White space
- Instructions
- Symbols

We'll start with programs that have no symbols, and handle symbols later

Program translation

Symbolic code

```
// Computes R1=1 + ... + R0
    // i = 1
    @16
    M=1
    // sum = 0
    @17
    M=0

    // if i>R0 goto STOP
    @16
    D=M
    @0
    D=D-M
    @18
    D;JGT
    // sum += i
    @16
    D=M
    @17
    M=D+M
    // i++
    @16
    M=M+1
    @4
    0;JMP
    @17
    D=M
    ...
```

no symbols

Translate

Binary code

Need to handle:

- White space
- Instructions
- Symbols (later)

Ignore it

White space:

- Empty lines,
- Comments,
- Indentation

Program translation

Symbolic code

```
@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
...
```

Translate

Binary code

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1111000010001000
0000000000010000
1111110111001000
0000000000000100
1110101010000111
0000000000010001
1111110000010000
...
```

<u>Need to handle:</u>

- White space
- Instructions
- Symbols (later)

Translate, one by one

Program translation

Symbolic code

```
// Computes R1=1 + ... + R0
    // i = 1
    @i
    M=1
    // sum = 0
    @sum
    M=0
(LOOP)
    // if i>R0 goto STOP
    @i
    D=M
    @R0
    D=D-M
    @STOP
    D;JGT
    // sum += i
    @i
    D=M
    @sum
    M=D+M
    // i++
    @i
    M=M+1
    @LOOP
    0;JMP
(STOP)
    @sum
    D=M
    ...
```

Translate

Binary code

Need to handle:

- White space
- Instructions
- Symbols

Original program, with symbols

Handling symbols

Symbolic code

```
// Computes R1=1 + ... + R0
    // i = 1
    @i
    M=1
    // sum = 0
    @sum
    M=0
(LOOP)
    // if i>R0 goto STOP
    @i
    D=M
    @R0
    D=D-M
    @STOP
    D;JGT
    // sum += i
    @i
    D=M
    @sum
    M=D+M
    // i++
    @i
    M=M+1
    @LOOP
    0;JMP
(STOP)
    @sum
    D=M
    ...
```

Original program, with symbols

Symbols

- Predefined symbols
- Label symbols
- Variable symbols

Handling symbols

Symbolic code

```
// Computes R1=1 + ... + R0
    // i = 1
    @i
    M=1
    // sum = 0
    @sum
    M=0
(LOOP)
    // if i>R0 goto STOP
    @i
    D=M
    @R0
    D=D-M
    @STOP
    D;JGT
    // sum += i
    @i
    D=M
    @sum
    M=D+M
    // i++
    @i
    M=M+1
    @LOOP
    0;JMP
(STOP)
    @sum
    D=M
    ...
```

The Hack language features
23 *predefined symbols*:

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

Translating @*preDefinedSymbol* :

Replace *preDefinedSymbol* with its *value*

Example: @R15 → 0000000000001111

Handling symbols

Symbolic code

```
       // Computes R1=1 + ... + R0
           // i = 1
 0         @i
 1         M=1
           // sum = 0
 2         @sum
 3         M=0
       (LOOP)
           // if i>R0 goto STOP
 4         @i
 5         D=M
 6         @R0
 7         D=D-M
 8         @STOP
 9         D;JGT
           // sum += i
10         @i
11         D=M
12         @sum
13         M=D+M
           // i++
14         @i
15         M=M+1
16         @LOOP
17         0;JMP
       (STOP)
18         @sum
19         D=M
...        ...
```

Label symbols

- Used to label destinations of goto instructions
- Declared by the pseudo-instruction (*label*)
- The (*label*) directive defines the symbol *label* to refer to the memory location holding the next instruction in the program,
- Which corresponds to the instruction's *line number*

Example:

symbol	*value*
LOOP	4
STOP	18

Translating @*labelSymbol* :

Replace *labelSymbol* with its *value*

Example: `@LOOP` ➡ `0000000000000100`

Handling symbols

Symbolic code

```
// Computes R1=1 + ... + R0
    // i = 1
    @i
    M=1
    // sum = 0
    @sum
    M=0
(LOOP)
    // if i>R0 goto STOP
    @i
    D=M
    @R0
    D=D-M
    @STOP
    D;JGT
    // sum += i
    @i
    D=M
    @sum
    M=D+M
    // i++
    @i
    M=M+1
    @LOOP
    0;JMP
(STOP)
    @sum
    D=M
    ...
```

Variable symbols

- Any symbol *xxx* which is neither predefined, nor defined elsewhere using an (*xxx*) label declaration, is treated as a *variable*
- Each variable is bound to a running memory address, starting at 16

Example:

symbol	*value*
i	16
sum	17

Translating @*variableSymbol* :

1. If *variableSymbol* is seen for the first time, bind to it a *value*, from 16 onward
   Else, it has a *value*
2. Replace *variableSymbol* with its *value*.

Example: `@sum` → `0000000000010001`

Handling symbols

Symbolic code

```
// Computes R1=1 + ... + R0
    // i = 1
    @i
    M=1
    // sum = 0
    @sum
    M=0
(LOOP)
    // if i>R0 goto STOP
    @i
    D=M
    @R0
    D=D-M
    @STOP
    D;JGT
    // sum += i
    @i
    D=M
    @sum
    M=D+M
    // i++
    @i
    M=M+1
    @LOOP
    0;JMP
(STOP)
    @sum
    D=M
    ...
```

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
i	16
sum	17

A data structure that the assembler creates and uses during the program translation

Contains the
predefined symbols,
label symbols,
variable symbols,
And their bindings.

Handling symbols

Symbolic code

```
// Computes R1=1 + ... + R0
    // i = 1
    @i
    M=1
    // sum = 0
    @sum
    M=0
(LOOP)
    // if i>R0 goto STOP
    @i
    D=M
    @R0
    D=D-M
    @STOP
    D;JGT
    // sum += i
    @i
    D=M
    @sum
    M=D+M
    // i++
    @i
    M=M+1
    @LOOP
    0;JMP
(STOP)
    @sum
    D=M
    ...
```

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

A data structure that the assembler creates and uses during the program translation

Initialization:
Creates the table and adds all the predefined symbols

Handling symbols

Symbolic code

```
      // Computes R1=1 + ... + R0
          // i = 1
 0        @i
 1        M=1
          // sum = 0
 2        @sum
 3        M=0
      (LOOP)
          // if i>R0 goto STOP
 4        @i
 5        D=M
 6        @R0
 7        D=D-M
 8        @STOP
 9        D;JGT
          // sum += i
10        @i
11        D=M
12        @sum
13        M=D+M
          // i++
14        @i
15        M=M+1
16        @LOOP
17        0;JMP
      (STOP)
18        @sum
19        D=M
...       ...
```

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

A data structure that the assembler creates and uses during the program translation

Initialization: Creates the table and adds all the predefined symbols

First pass: Counts lines and adds the label symbols

Handling symbols

Symbolic code

```
// Computes R1=1 + ... + R0
    // i = 1
    @i
    M=1
    // sum = 0
    @sum
    M=0
(LOOP)
    // if i>R0 goto STOP
    @i
    D=M
    @R0
    D=D-M
    @STOP
    D;JGT
    // sum += i
    @i
    D=M
    @sum
    M=D+M
    // i++
    @i
    M=M+1
    @LOOP
    0;JMP
(STOP)
    @sum
    D=M
    ...
```

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
i	16
sum	17

A data structure that the assembler creates and uses during the program translation

Initialization:
Creates the table and adds all the predefined symbols

First pass: Counts lines and adds the label symbols

Second pass: Generates binary code;
In the process, adds the variable symbols

(details, soon)

Assembler: Usage

```
// Computes R1=1 + ... + R0
    // i = 1
    @i
    M=1
    // sum = 0
    @sum
    M=0
(LOOP)
    // if i>R0 goto STOP
    @i
    D=M
    @R0
    D=D-M
    ...
```

Assembler

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

<u>Input</u> (*Prog*`.asm`): a text file containing a sequence of lines, each being a comment, an A instruction, or a C-instruction

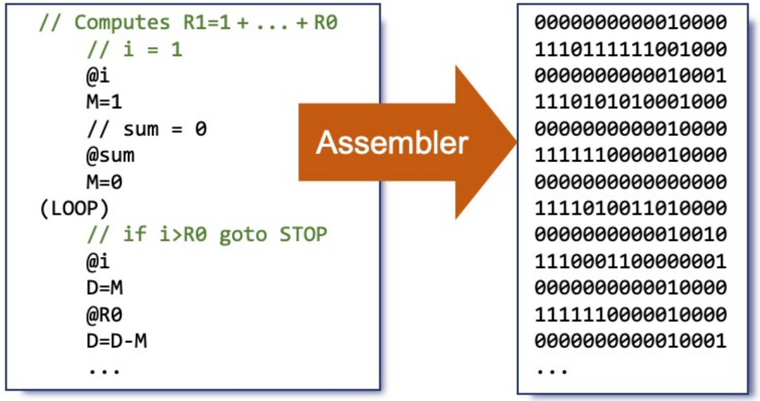
<u>Output</u> (*Prog*`.hack`): a text file containing a sequence of lines, each being a string of sixteen `0` and `1` characters

<u>Usage</u>: (if the assembler is implemented in Java)

`$ java HackAssembler` *Prog*`.asm`

Action: Creates a *Prog*`.hack` file, containing the translated Hack program

Assembler: Algorithm



Initialize:

Opens the input file (*Prog*`.asm`),
and gets ready to process it

Constructs a symbol table,
and adds to it all the predefined symbols

First pass:

Reads the program lines, one by one,
focusing only on `(`*label*`)` declarations.
Adds the found labels to the symbol table

Second pass (main loop):

(starts again from the beginning of the file)

While there are more lines to process:

Gets the next instruction, and parses it

If the instruction is `@`*symbol*

If *symbol* is not in the symbol table, adds it

Translates the *symbol* to its binary value

If the instruction is *dest*`=`*comp*`;`*jump*

Translates each of the three fields into its binary value

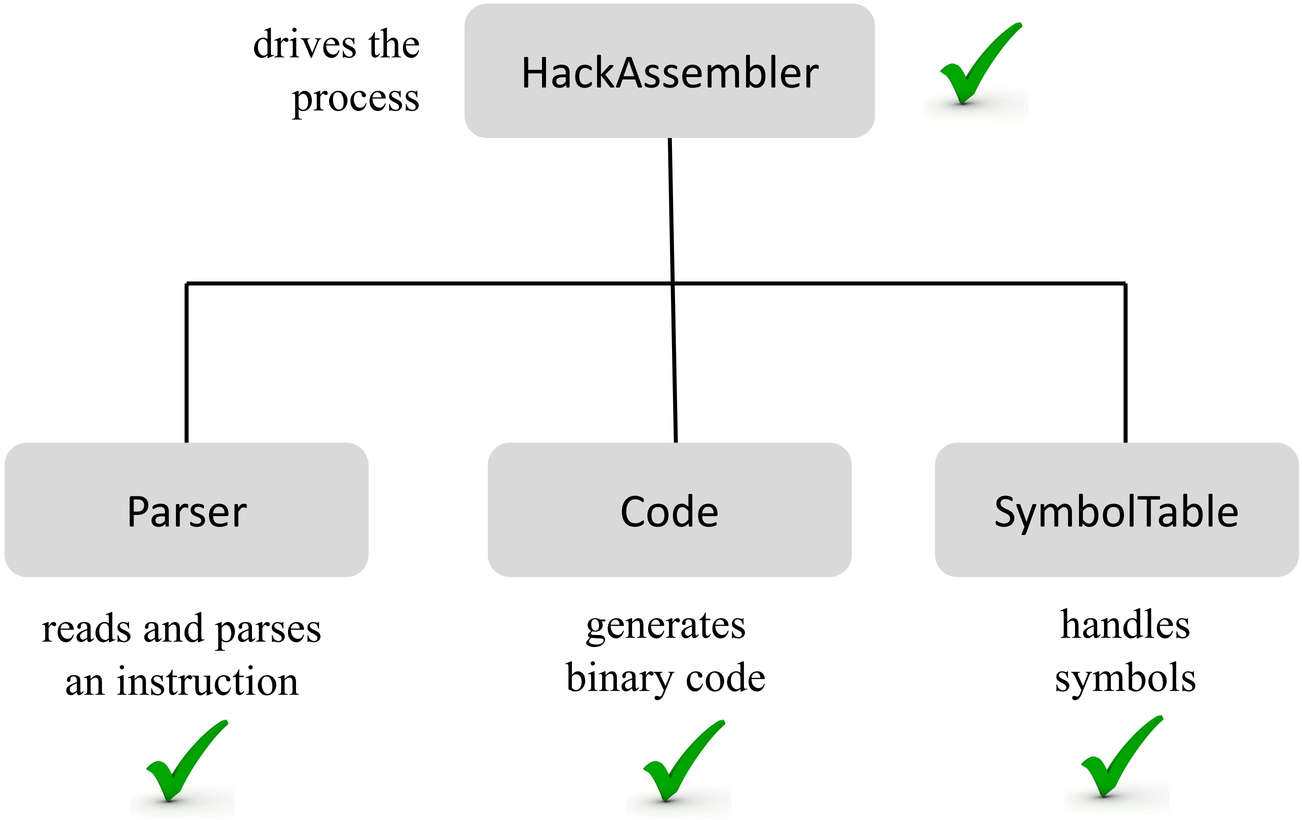
Assembles the binary values into a string of sixteen `0`'s and `1`'s

Writes the string to the output file.

Assembler implementation options

- Manual
- Program-based

`HackAssembler`: Drives the translation process



HackAssembler

Initialize:

Opens the input file (*Prog*.asm) and gets ready to process it

Constructs a symbol table, and adds to it all the predefined symbols

First pass:

Reads the program lines, one by one
focusing only on (*label*) declarations.
Adds the found labels to the symbol table

Second pass (main loop):

(starts again from the beginning of the file)

While there are more lines to process:

  Gets the next instruction, and parses it

  If the instruction is @*symbol*

    If *symbol* is not in the symbol table, adds it

    Translates the *symbol* into its binary value

  If the instruction is *dest*=*comp*;*jump*

    Translates each of the three fields into its binary value

  Assembles the binary values into a string of sixteen 0's and 1's

  Writes the string to the output file.

The HackAssembler implements the assembly algorithm, using the services of:

- Parser
- Code
- SymbolTable

`Parser` API

Routines:

- Constructor / initializer: Creates a `Parser` and opens the source text file
- Getting the current instruction:

  `hasMoreLines()`: Checks if there is more work to do (boolean)

  `advance()`: Gets the next instruction and makes it the *current instruction* (string)

- Parsing the *current instruction*:

  `instructionType()`: Returns the current instruction type (constant):

  `A_INSTRUCTION` for `@`*xxx*, where *xxx* is either a decimal number or a symbol

  `C_INSTRUCTION` for *dest* `=` *comp* `;` *jump*

  `L_INSTRUCTION` for `(` *label* `)`

Examples:

current instruction	
`@17`	`instructionType()` returns `A_INSTRUCTION`
`@sum`	`instructionType()` returns `A_INSTRUCTION`
`D=0`	`instructionType()` returns `C_INSTRUCTION`
`(END)`	`instructionType()` returns `L_INSTRUCTION`

Parser API

Routines:

- Constructor / initializer: Creates a `Parser` and opens the source text file
- Getting the current instruction:

  `hasMoreLines()`: Checks if there is more work to do

  `advance()`: Gets the next instruction and makes it the *current instruction*
- Parsing the *current instruction*:

  `instructionType()`: Returns the instruction type

  `symbol()`: Returns the instruction's *symbol* (string)

Used if the current instruction is `@`*symbol* or `(`*symbol*`)`

Examples:

current instruction	
`@sum`	`symbol()` returns `"sum"`
`(LOOP)`	`symbol()` returns `"LOOP"`

`Parser` API

Routines:

- Constructor / initializer: Creates a `Parser` and opens the source text file
- Getting the current instruction:

  `hasMoreLines()`: Checks if there is more work to do

  `advance()`: Gets the next instruction and makes it the *current instruction*

- Parsing the *current instruction*:

  `instructionType()`: Returns the instruction type

  `symbol()`: Returns the instruction's *symbol* (string)

  `dest()`: Returns the instruction's *dest* field (string)

  `comp()`: Returns the instruction's *comp* field (string)

  `jump()`: Returns the instruction's *jump* field (string)

Used if the current instruction is *dest* = *comp* ; *jump*

Examples:

current instruction			
`D=D+1;JLE`	`dest()` returns `"D"`	`comp()` returns `"D+1"`	`jump()` returns `"JLE"`
`M=-1`	`dest()` returns `"M"`	`comp()` returns `"-1"`	`jump()` returns `null`

Code API

Deals only with C-instructions: *dest* = *comp* ; *jump*

Routines:

`dest`(string): Returns the binary representation of the parsed *dest* field (string)

`comp`(string): Returns the binary representation of the parsed *comp* field (string)

`jump`(string): Returns the binary representation of the parsed *jump* field (string)

According to the language specification:

comp (a==0)	*comp* (a==1)	c	c	c	c	c	c
0		1	0	1	0	1	0
1		1	1	1	1	1	1
-1		1	1	1	0	1	0
D		0	0	1	1	0	0
A	M	1	1	0	0	0	0
!D		0	0	1	1	0	1
!A	!M	1	1	0	0	0	1
-D		0	0	1	1	1	1
-A	-M	1	1	0	0	1	1
D+1		0	1	1	1	1	1
A+1	M+1	1	1	0	1	1	1
D-1		0	0	1	1	1	0
A-1	M-1	1	1	0	0	1	0
D+A	D+M	0	0	0	0	1	0
D-A	D-M	0	1	0	0	1	1
A-D	M-D	0	0	0	1	1	1
D&A	D&M	0	0	0	0	0	0
D|A	D|M	0	1	0	1	0	1

dest	d	d	d
null	0	0	0
M	0	0	1
D	0	1	0
DM	0	1	1
A	1	0	0
AM	1	0	1
AD	1	1	0
ADM	1	1	1

jump	j	j	j
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

Examples:

`dest("DM")` returns `"011"`

`comp("A+1")` returns `"0110111"`

`comp("D&M")` returns `"1000000"`

`jump("JNE")` returns `"101"`

`SymbolTable` API

Routines

Constructor / initializer: Creates and initializes a `SymbolTable`

`addEntry(symbol` (string), **`address`** (int)**`):`** Adds `<symbol`, `address>` to the table (void)

`contains(symbol` (string)**`):`** Checks if `symbol` exists in the table (boolean)

`getAddress(symbol` (string)**`):`** Returns the `address` (int) associated with `symbol`

Symbol table:
(example)

symbol	*address*
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
i	16
sum	17

Assembler API (detailed)

Parser module:

Routine	*Arguments*	*Returns*	*Function*
Constructor / initializer	Input file or stream	—	Opens the input file/stream and gets ready to parse it.
`hasMoreLines`	—	boolean	Are there more lines in the input?
`advance`	—	—	Skips over whitespace and comments, if necessary. Reads the next instruction from the input, and makes it the current instruction. This method should be called only if `hasMoreLines` is true. Initially there is no current instruction.
`instructionType`	—	`A_INSTRUCTION`, `C_INSTRUCTION`, `L_INSTRUCTION` (constants)	Returns the type of the current instruction: `A_INSTRUCTION` for @*xxx*, where *xxx* is either a decimal number or a symbol. `C_INSTRUCTION` for *dest*=*comp*;*jump* `L_INSTRUCTION` for (*xxx*), where *xxx* is a symbol.
`symbol`	—	string	If the current instruction is (*xxx*), returns the symbol *xxx*. If the current instruction is @*xxx*, returns the symbol or decimal *xxx* (as a string). Should be called only if `instructionType` is `A_INSTRUCTION` or `L_INSTRUCTION`.
`dest`	—	string	Returns the symbolic *dest* part of the current *C*-instruction (8 possibilities). Should be called only if `instructionType` is `C_INSTRUCTION`.
`comp`	—	string	Returns the symbolic *comp* part of the current *C*-instruction (28 possibilities). Should be called only if `instructionType` is `C_INSTRUCTION`.
`jump`	—	string	Returns the symbolic *jump* part of the current *C*-instruction (8 possibilities). Should be called only if `instructionType` is `C_INSTRUCTION`.

Assembler API (detailed)

Code module:

Routine	Arguments	Returns	Function
`dest`	string	3 bits, as a string	Returns the binary code of the *dest* mnemonic.
`comp`	string	7 bits, as a string	Returns the binary code of the *comp* mnemonic.
`jump`	string	3 bits, as a string	Returns the binary code of the *jump* mnemonic.

SymbolTable module:

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

HackAssembler module:

We propose no API; Implement as you see fit.

Developing a Hack Assembler

Contract

- Develop a program that translates symbolic Hack programs into binary Hack instructions
- The source program (input) is supplied as a text file named *Prog*`.asm`
- The generated code (output) is written into a text file named *Prog*`.hack`
- Assumption: *Prog*`.asm` is error-free

Usage (if the assembler is implemented in Java):

```
$ java HackAssembler Prog.asm
```

Staged development plan

1. Develop a basic assembler that translates programs that have no symbols
2. Develop an ability to handle symbols
3. Morph the basic assembler into an assembler that translates any program

Testing

Prog.asm

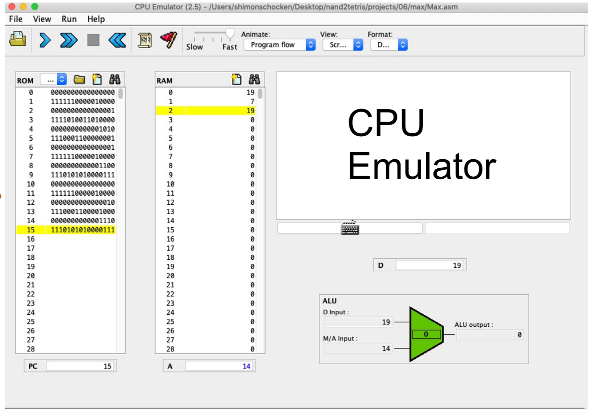
```
// Computes R1 = 1 + ... + R0
    // i = 1
    @i
    M=1
    // sum = 0
    @sum
    M=0
(LOOP)
    // if i > R0 goto STOP
    @i
    D=M
    @R0
    D=D-M
    @STOP
    D;JGT
    ...
```

Your assembler

Prog.hack

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

Load / Run



Or use a supplied test script that loads `Prog.hack` into the CPU emulator and tests it using pre-defined testing scenarios

Test programs

- `Add.asm`
- `Max.asm`
- `Rect.asm`
- `Pong.asm`

(with symbols)

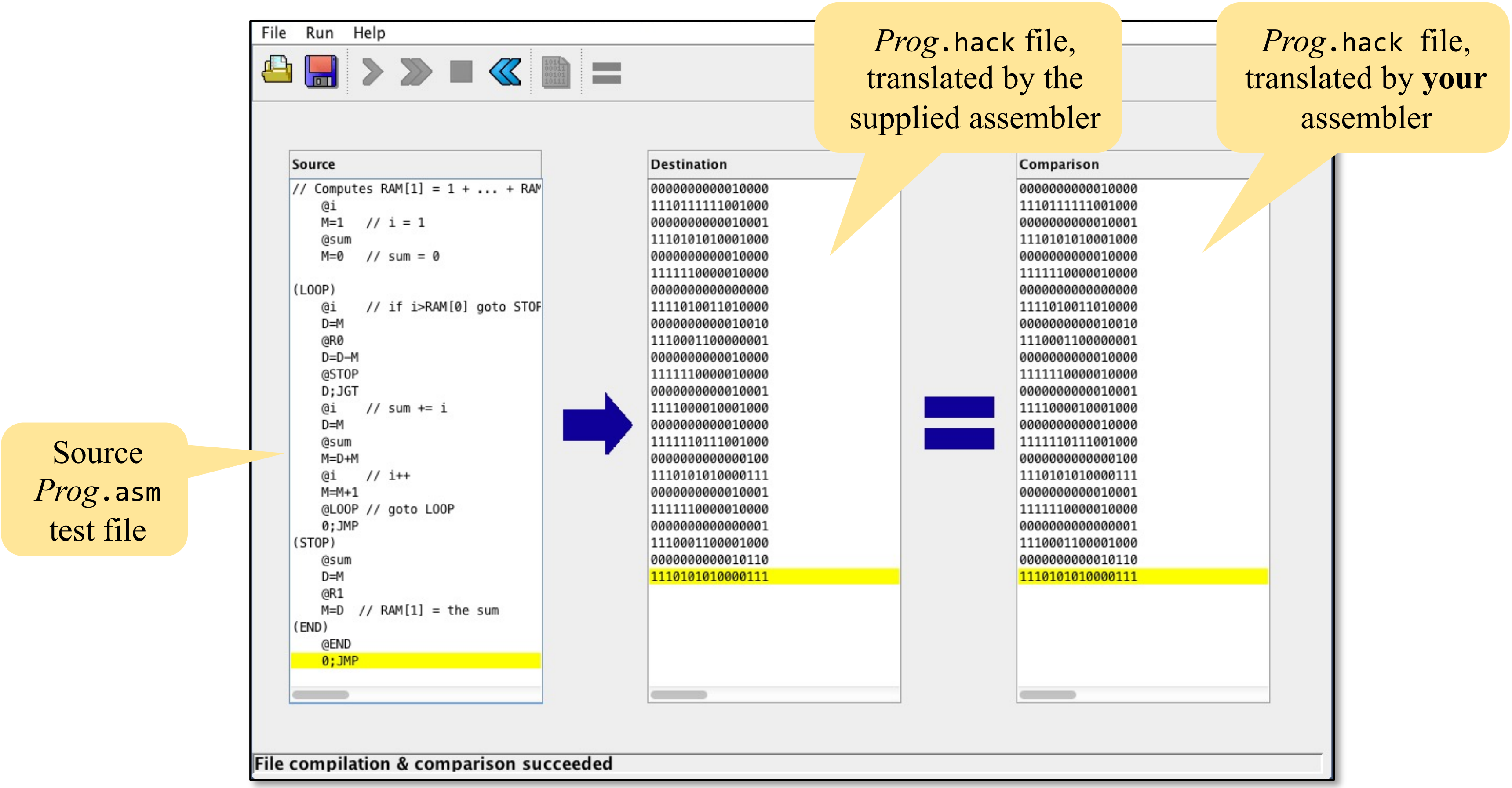
- `MaxL.asm`
- `RectL.asm`
- `PongL.asm`

(same programs, without symbols, for unit-testing the basic assembler)

Testing option II: Using the hardware simulator

1. Use your assembler to translate *Prog*`.asm`, generating the executable file *Prog*`.hack`
2. Put the *Prog*`.hack` file in a folder containing the chips that you developed in project 5: `Computer.hdl`, `CPU.hdl`, and `Memory.hdl`
3. Load `Computer.hdl` into the Hardware Simulator
4. Load *Prog*`.hack` into the `ROM32K` chip-part
5. Run the clock to execute the program.

Testing option III: Using the supplied assembler



1. Use your assembler to translate *Prog*.asm, generating the executable file *Prog*.hack
2. Load *Prog*.asm into the supplied assembler, and load *Prog*.hack as a compare file
3. Translate *Prog*.hack, and inspect the comparison feedback messages.

Project 6

Guidelines: www.nand2tetris.org (projects section)

Files: `nand2tetris/projects/06` (on your PC)

Tools

- The programming language in which you develop your assembler
- CPU emulator (for testing the translated programs)
- Assembler (if you plan to use it)

Guides

- CPU emulator tutorial
- Assembler tutorial