Lecture 6

Assembler

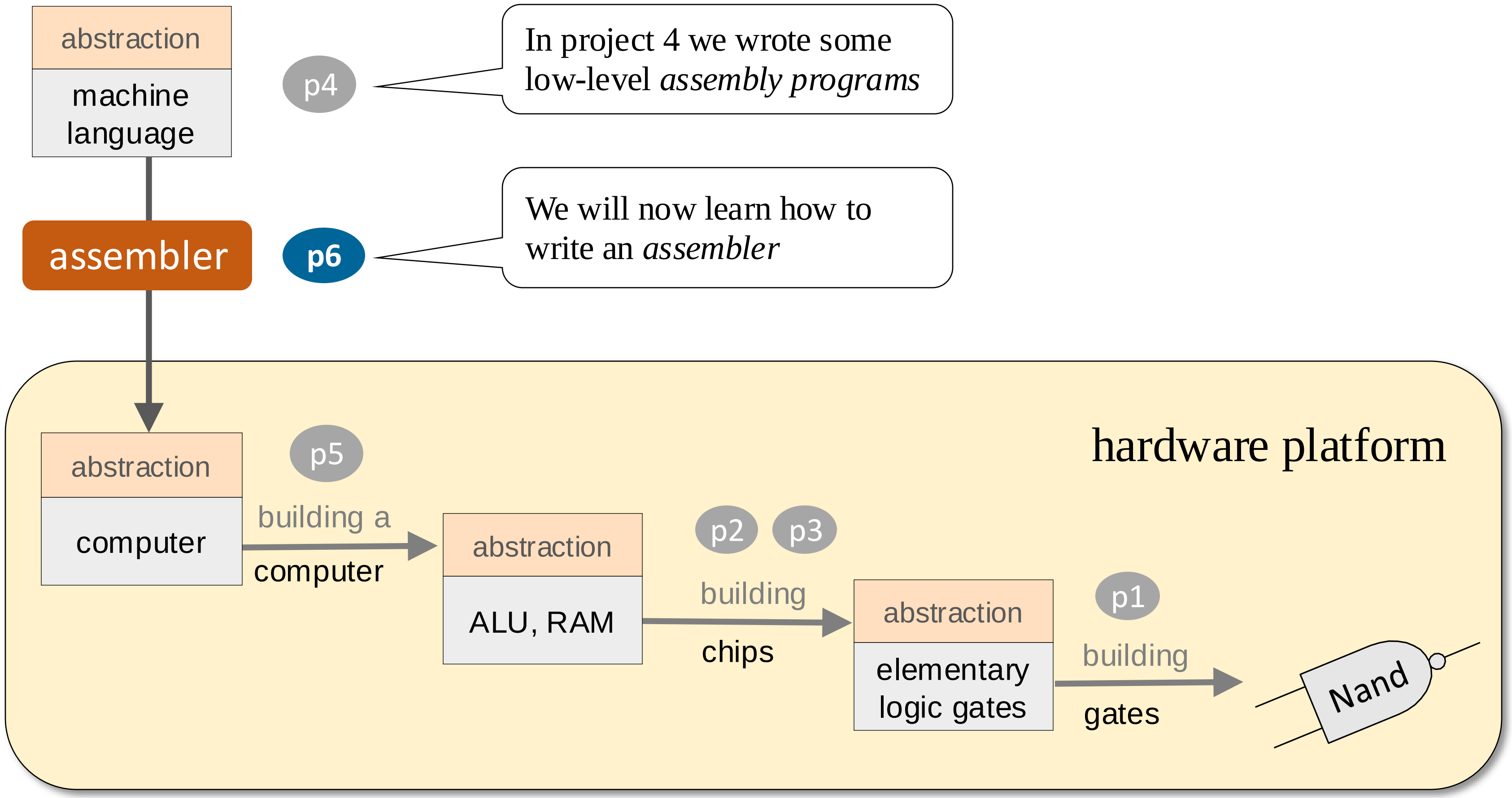
These slides support chapter 6 of the book

The Elements of Computing Systems

By Noam Nisan and Shimon Schocken

MIT Press, 2021

Nand to Tetris Roadmap: Hardware



Program translation

Assembly 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
   ...
```

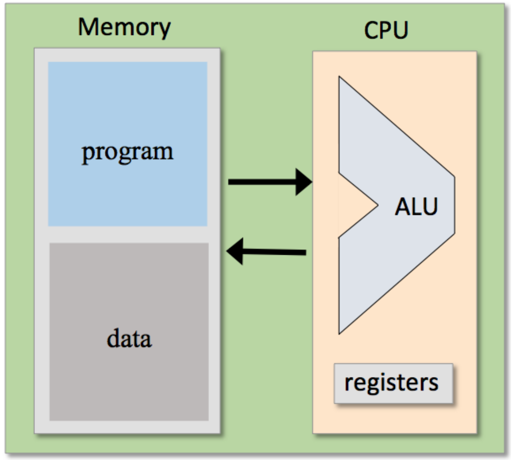
assembler

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



The assembler is…

- The first step in a typical hierarchy of translators (assembler, VM translator, compiler)
- A program that introduces basic software engineering techniques used by every translator:
  - Files handling
  - Parsing
  - Code generation
  - Symbol tables

Lecture plan

- Overview
- Translating Hack code:
  - A-instructions
  - C-instructions
- Translating programs
- Handling symbols
- Assembler architecture
- Assembler API
- Project 6
- Some history

Translating A-instructions

Symbolic syntax:

`@`*xxx*

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

Binary syntax:

`0`*vvvvvvvvvvvvvvv*

Where:

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

v v v … v is a binary value

Example:

`@7`

translate

`0000000000000111`

<u>Implementation</u>

If *xxx* is a decimal value: Translate the value into its 16-bit representation;

If *xxx* is a symbol: Later.

Lecture plan

- Overview
- Translating Hack code:
  - A-instructions
  - C-instructions
- Translating programs
- Handling symbols
- Assembler architecture
- Assembler API
- Project 6
- Some history

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

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

Binary:

Example: D=D+1;JLE → 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 (*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: A = -1 ➡ Binary: `1110111010100000`

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

Chapter 6: Assembler

- Overview
- Translating instructions
- Translating programs
- Handling symbols
- Assembler architecture
- Assembler API
- Project 6
- Some history

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
...
```

<u>Need to Handle</u>

- White space
- Instructions
- Symbols

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
...
```

<u>Need to Handle</u>

- White space
- Instructions
- Symbols

Normally, programs have `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)

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 — Ignore it
- Instructions
- Symbols (later)

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
...
```

no white space

Translate

Binary code

<u>Need to Handle</u>

- White space
- Instructions
- Symbols (later)

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 — Translate, one by one
- Symbols (later) — As shown earlier in the lecture

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
...
```

Need to Handle

- White space
- Instructions
- Symbols

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
    ...
```

Original program, with symbols

Translate

Binary code

Need to Handle

- White space
- Instructions
- 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
    ...
```

Predefined symbol used in this code example: `R0`

<u>Symbols</u>

- Predefined symbols
- Label symbols
- Variable symbols

In the Hack language:

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

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
    ...
```

Symbols

- Predefined symbols
- Label symbols
- Variable symbols

In the Hack language:

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*, and complete the translation.

Examples:

@R0	➡	0000000000000000
@R12	➡	0000000000001100
@SCREEN	➡	0100000000000000

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
    ...
```

Symbols

- Predefined symbols
- Label symbols
- Variable symbols

Label symbols in this code example: `LOOP`, `STOP`

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
    ...
```

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*

Label symbols in this code example: `LOOP`, `STOP`

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
    ...
```

Symbols

- Predefined symbols
- Label symbols
- Variable symbols

variable symbols in this code example: `i`, `sum`

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*
- Hack convention: 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 to 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

Created by the assembler, used during the program translation

Initialization:
Creates the symbol table and adds the predefined symbols to the table

First pass: Counts lines and adds the label symbols to the table

Second pass:
- Generates binary code; in the process:
- Adds the variable symbols to the table

(details, soon)

Lecture plan

- Overview
- Translating instructions
- Translating programs
- Handling symbols

- Assembler
- Assembler API
- Project 6
- Some history

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
...
```

Input (*Prog*`.asm`): a text file containing a sequence of lines, each being a string representing a comment, an A-instruction, a C-instruction, or a label declaration

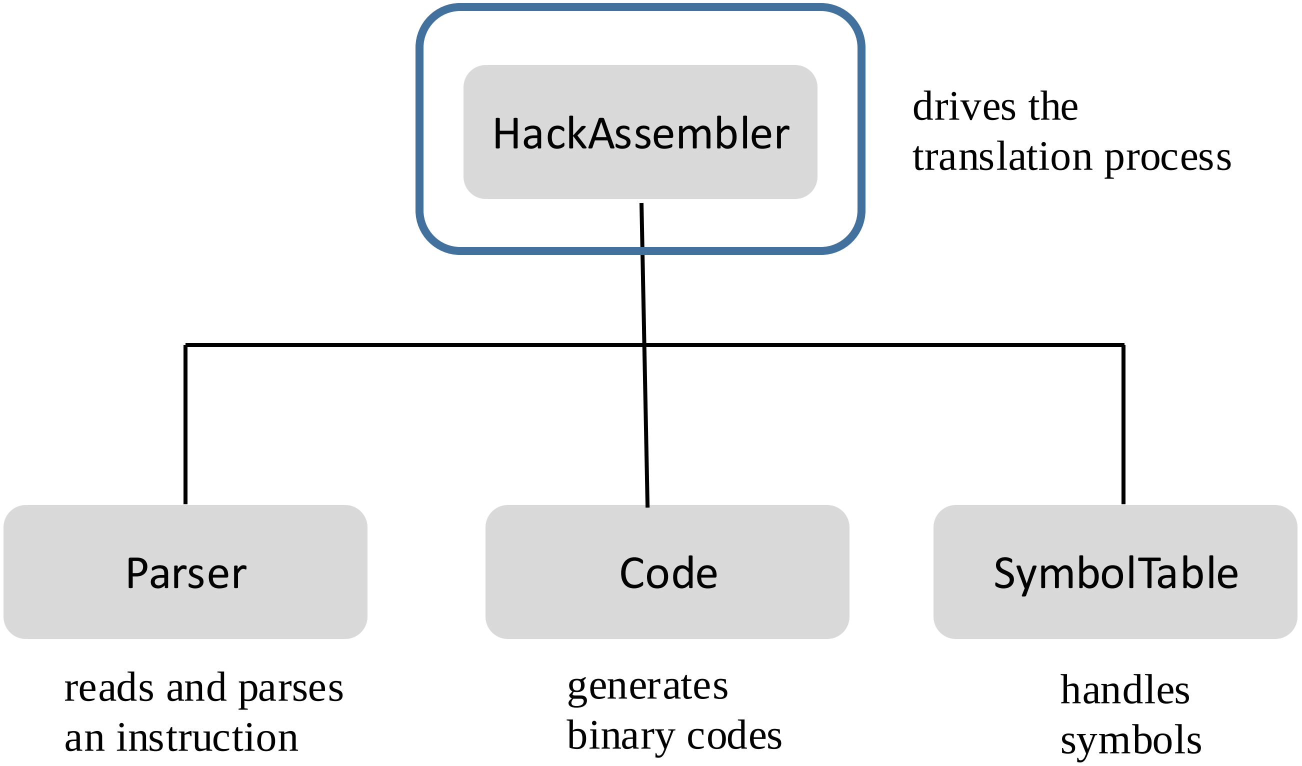
Output (*Prog*`.hack`): a text file containing a sequence of lines, each being a string of sixteen `0` and `1` characters

Usage: (e.g., Java implementation)

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

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

Assembler: Architecture



<u>Proposed architecture</u>

- Four software modules
- Can be realized in any programming language

HackAssembler

<u>Initialize</u>

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

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

<u>First pass</u>

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

<u>Second pass</u> (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 to the table

    Translates the *symbol* into its binary value

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

    Translates each of the three fields into its binary value

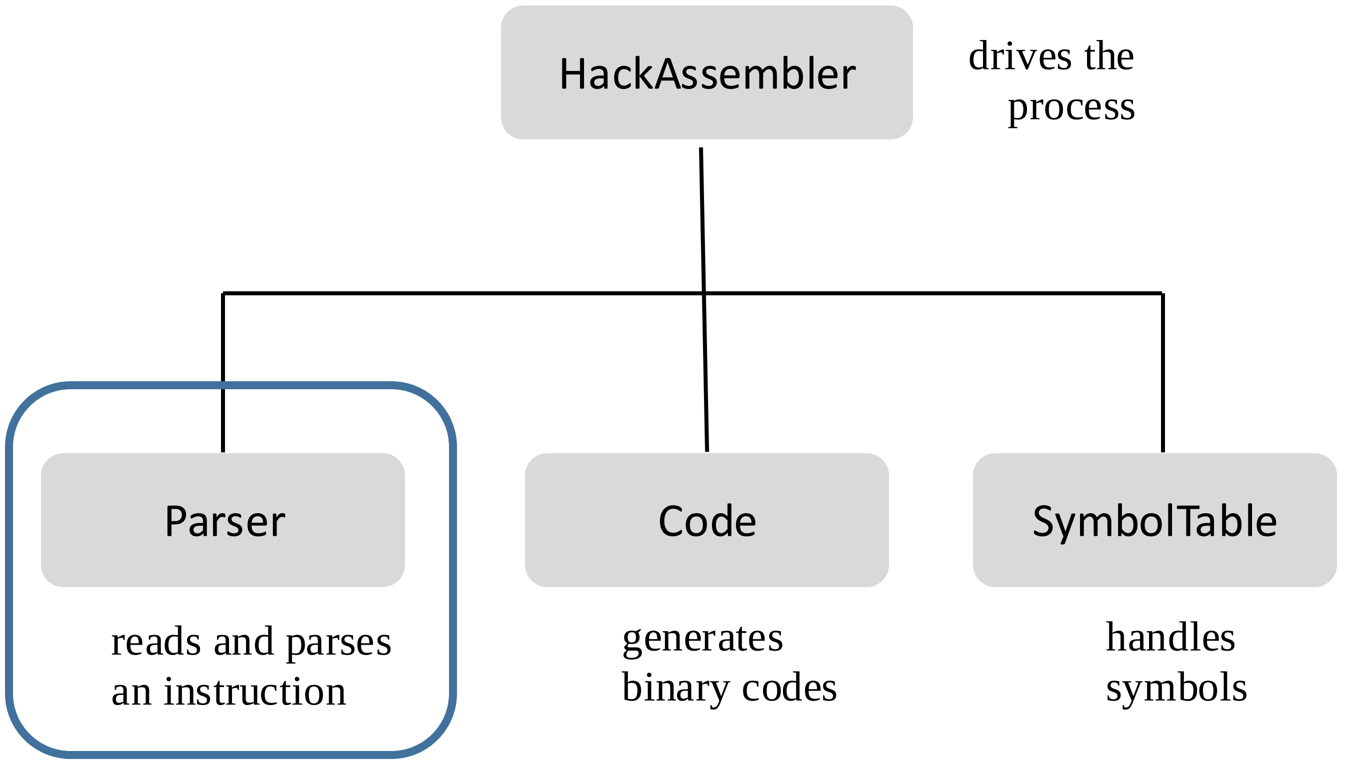
  Assembles the binary values into a string of sixteen `0`'s and `1`'s

  Writes the string to the output file.

Using the services of:

- `Parser`
- `Code`
- `SymbolTable`

Assembler API



`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 type of the current instruction, as a constant:

  `A_INSTRUCTION` for `@`*xxx*

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

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

Examples:

Current instruction	`instructionType()` returns:
`@17`	`A_INSTRUCTION`
`@sum`	`A_INSTRUCTION`
`D=0`	`C_INSTRUCTION`
`(END)`	`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 (boolean)

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

- Parsing the *current instruction*:

  `instructionType()`: Returns the instruction type

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

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

Examples:

Current instruction	`symbol()` returns:
`@sum`	`"sum"`
`(LOOP)`	`"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 (boolean)

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

- 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)

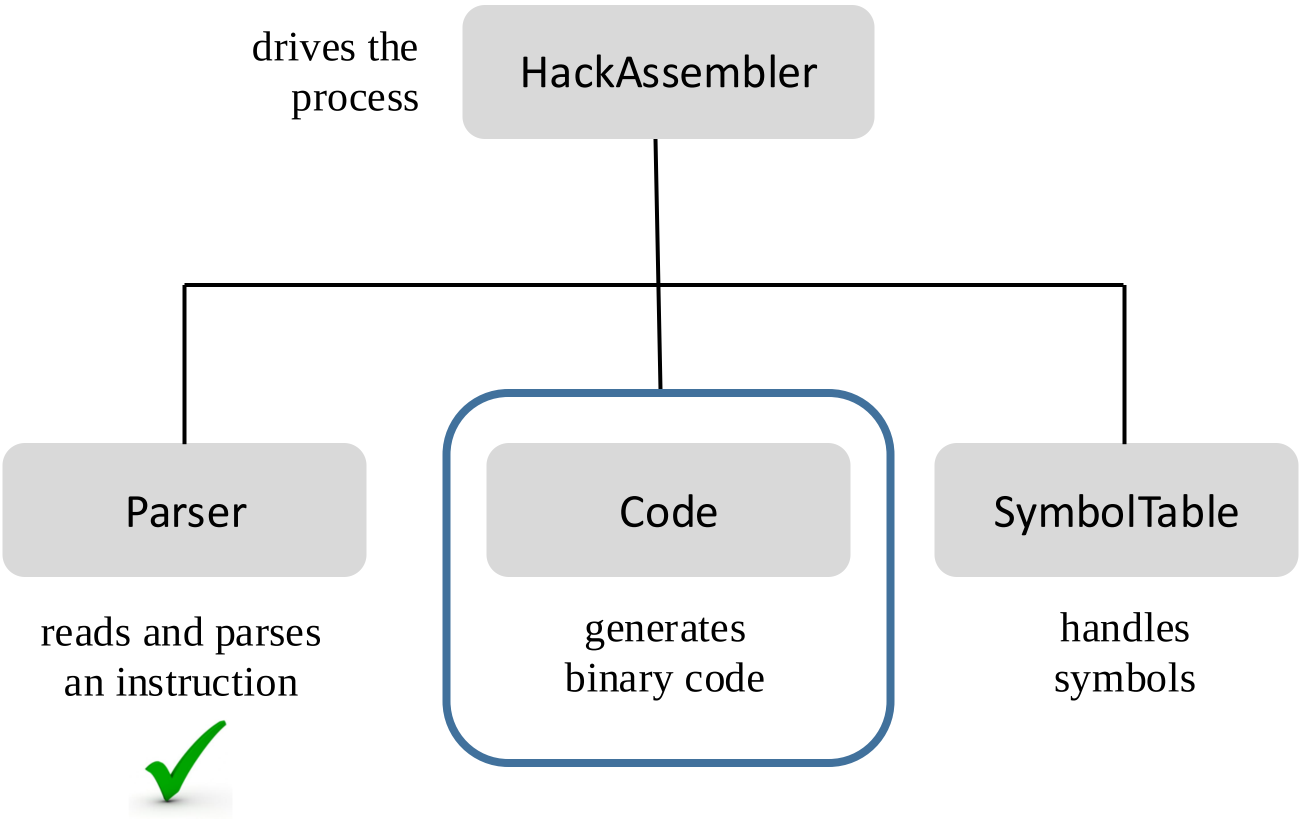
  `jump()`: Returns the instruction's *jump* field (string)

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

current instruction

Examples:			
`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`

Implementation



Code API

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

<u>Routines:</u>

`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:

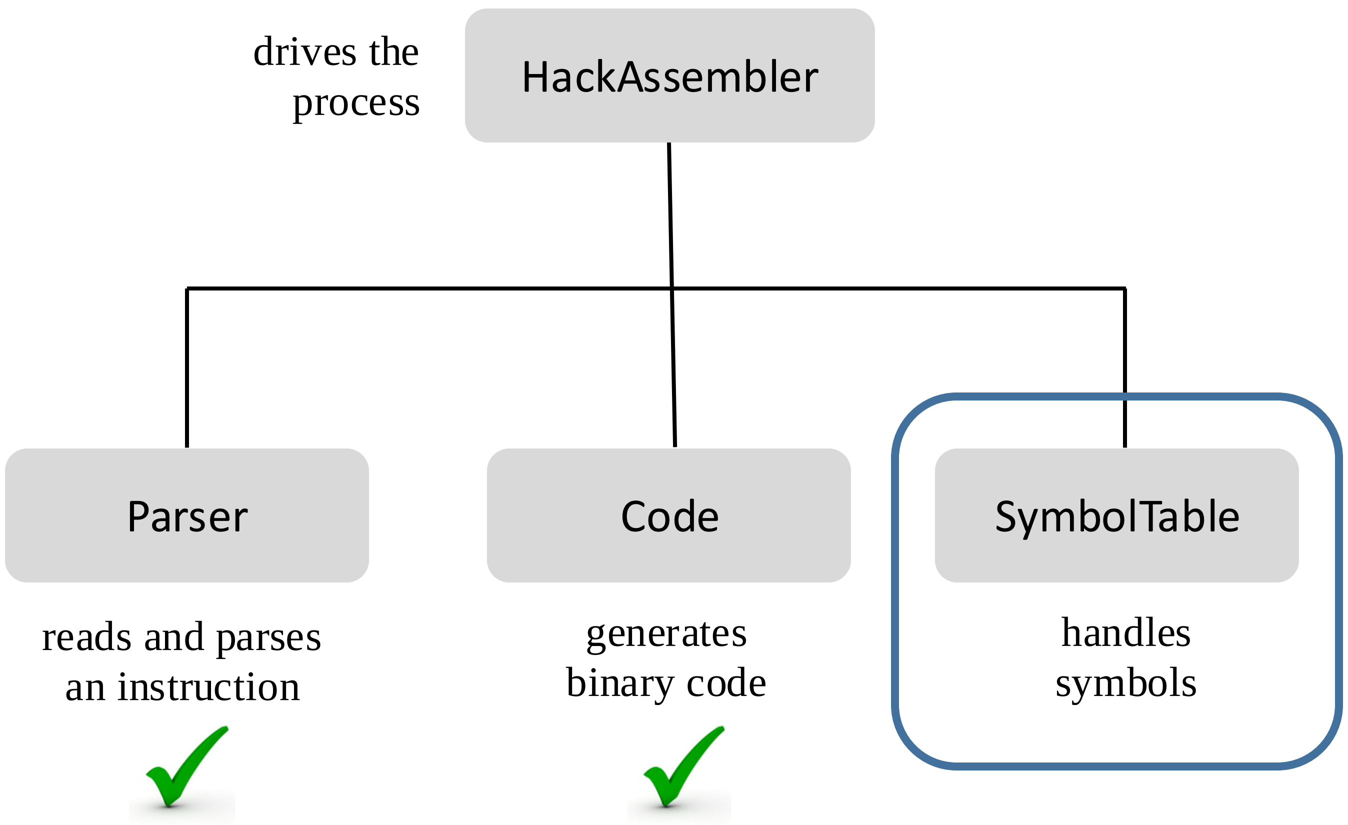
`dest("DM")` returns `"011"`

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

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

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

Implementation



`SymbolTable` API

Routines

Constructor / initializer: Creates and initializes a `SymbolTable`

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

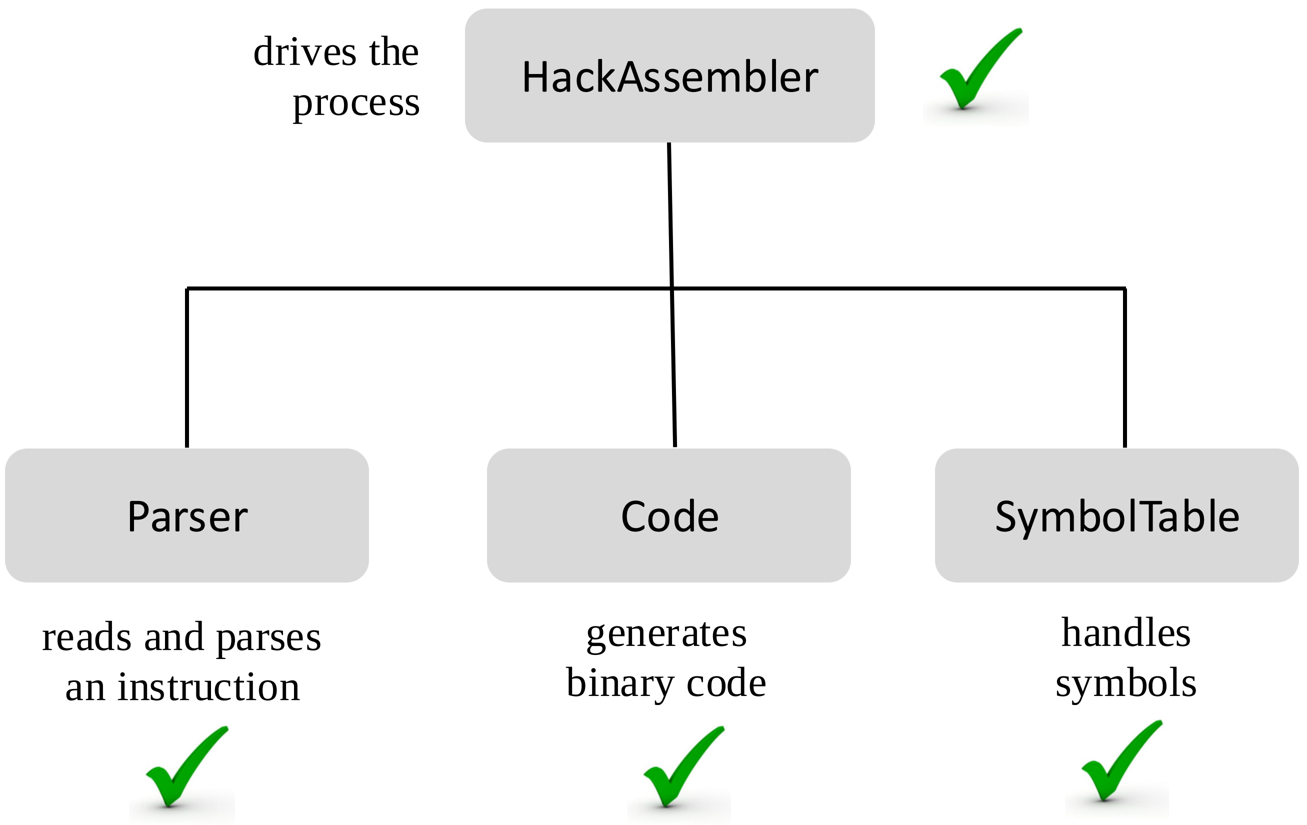
`boolean contains(String symbol):` Checks if `symbol` exists in the table

`int getAddress(String symbol):` Returns the `address` 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

`HackAssembler`: Drives the translation process



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 (main program):

No proposed design; Implement as you see fit.

Chapter 6: Assembler

- Overview
- Translating instructions
- Translating programs
- Handling symbols
- Assembler architecture
- Assembler API
- Project 6
- Some history

Project

Contract

Develop a program that translates symbolic Hack programs into binary Hack instructions;

The source assembly program (input) is read from a text file named *Prog*`.asm`

The generated binary code (output) is written to a text file named *Prog*`.hack`

Assumption: *Prog*`.asm` is error-free.

Usage (e.g. Java implementation):

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

Project

Prog.asm

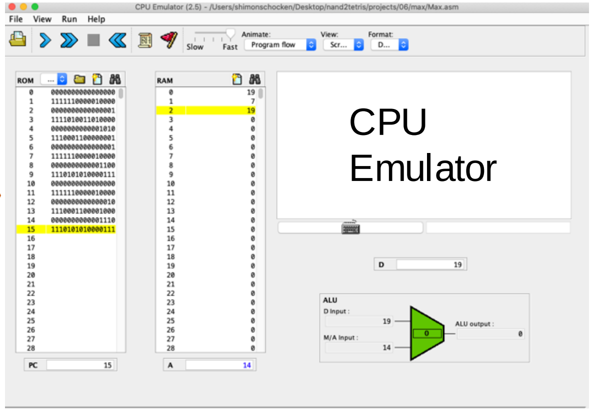
```
// Computes R1 = 1 + ... + R0
    // i = 1
    @i
    M=1
    // sum = 0
    @sum
    M=0
(LOOP)
    // if i > R0 goto STOP
    @i
    D=M
    @R0
    D=D-M
    @STOP
    D;JGT
    ...
```

Your assembler

Prog.hack

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

Load / Run



Staged development plan

1. Develop a basic assembler that translates Hack assembly programs containing no symbols
2. Develop an ability to handle symbols
3. Morph your basic assembler into an assembler that translates any Hack assembly program.

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 your basic assembler)

Testing: Add

Add.asm

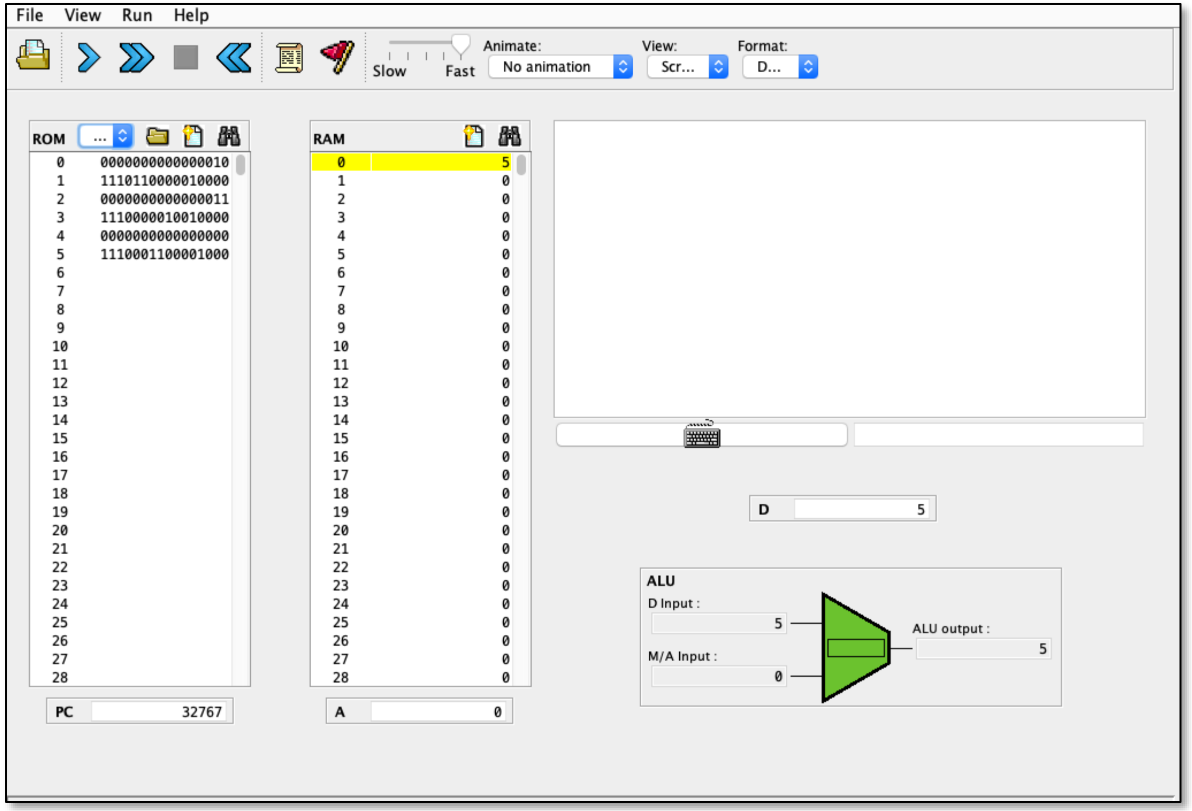
```
// Computes RAM[0] = 2  + 3
@2
D=A
@3
D=D+A
@0
M=D
```

Testing: `Add`

`Add.asm`

```
// Computes RAM[0] = 2  + 3
@2
D=A
@3
D=D+A
@0
M=D
```

Testing on the CPU emulator:



Techincal note

When loading a binary *Prog*`.hack` file into the CPU emulator, the emulator may present the code symbolically, for readability (depending on the emulator's version).

To inspect the binary code, select the "binary" display option.

1. Translate `Add.asm` using your assembler
2. Load into the CPU emulator the translated `Add.hack`
3. Run the code, inspect `R0`.

Testing: Max

Max.asm

```
// Computes RAM[2] =
// max(RAM[0],RAM[1])
   @R0
   D=M
   @R1
   D=D-M
   @OUTPUT_RAM0
   D;JGT
   // Output RAM[1]
   @R1
   D=M
   @R2
   M=D
   @END
   0;JMP
(OUTPUT_RAM0)
   @R0
   D=M
   @R2
   M=D
(END)
   @END
   0;JMP
```

with symbols

MaxL.asm

```
// Computes RAM[2] =
// max(RAM[0],RAM[1])
   @0
   D=M
   @1
   D=D-M
   @12
   D;JGT
   // Output RAM[1]
   @1
   D=M
   @2
   M=D
   @16
   0;JMP

   @0
   D=M
   @2
   M=D

   @16
   0;JMP
```

without symbols

Same test program, without symbols, for unit-testing the basic assembler

Testing: `Max`

`Max.asm`

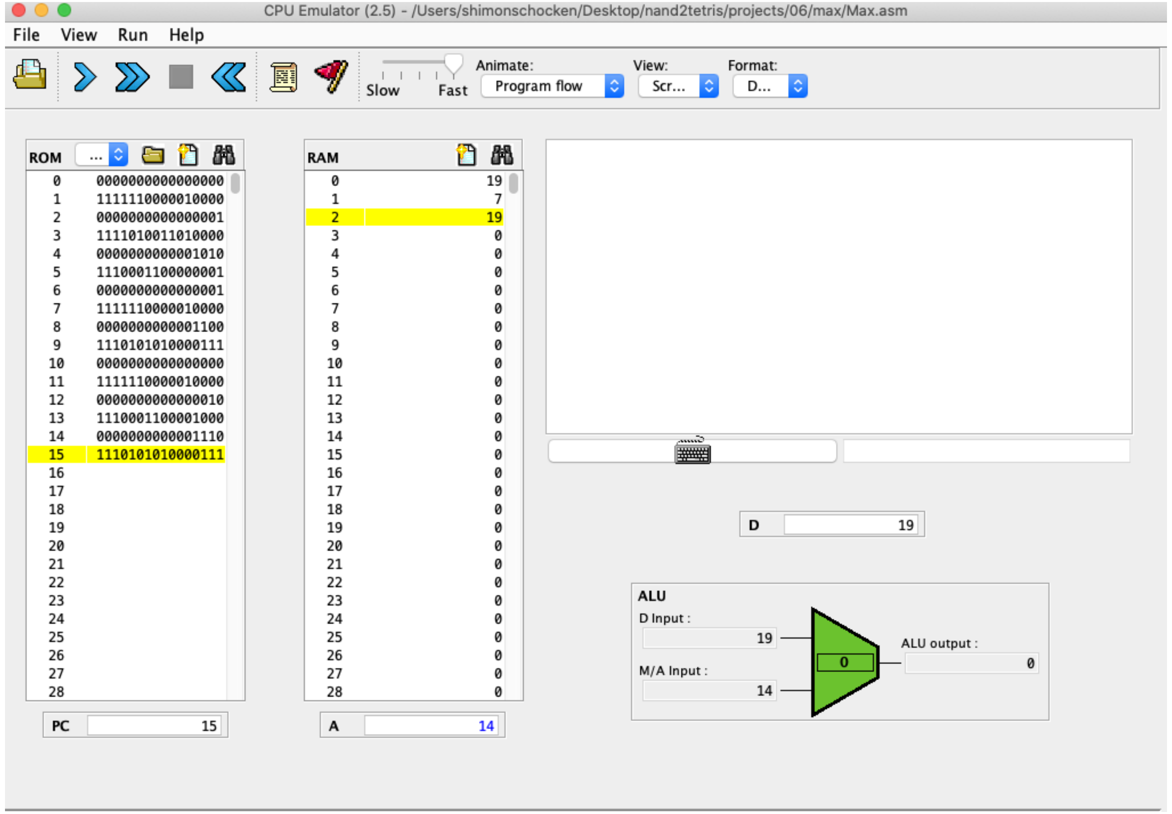
```
// Computes RAM[2] =
// max(RAM[0],RAM[1])
   @R0
   D=M
   @R1
   D=D-M
   @OUTPUT_RAM0
   D;JGT

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

(OUTPUT_RAM0)
   @R0
   D=M
   @R2
   M=D

(END)
   @END
   0;JMP
```

Testing on the CPU emulator:



1. Translate `Max.asm`
2. Load `Max.hack`
3. Put test values in `R0` and `R1`, run the code, inspect `R2`.

Testing: `Rect`

`Rect.asm`

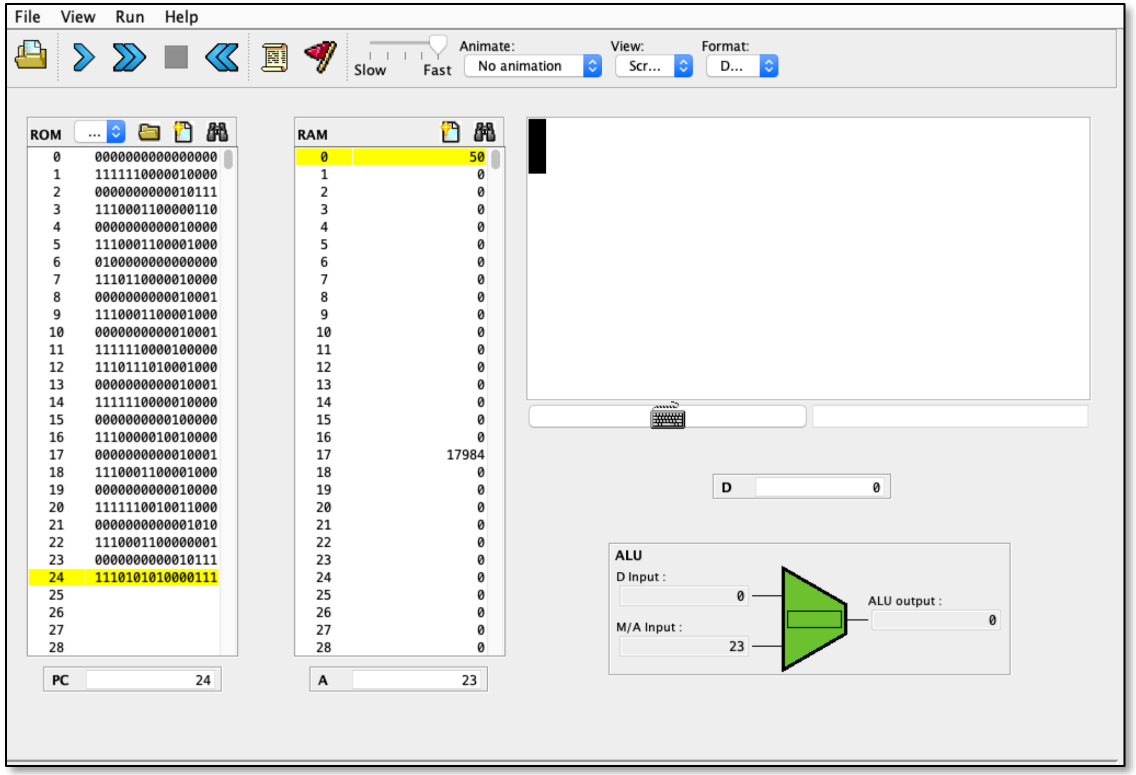
```
// Draws a rectangle,
// 16 pixels wide,
// R0 pixels high,
// at the screen's top-left.

   @R0
   D=M
   @n
   M=D
   @i
   M=0

   @SCREEN
   D=A
   @address
   M=D

(LOOP)
   @i
   D=M
   @n
   D=D-M
   @END
   D;JGT
   ...
```

Testing on the CPU emulator:

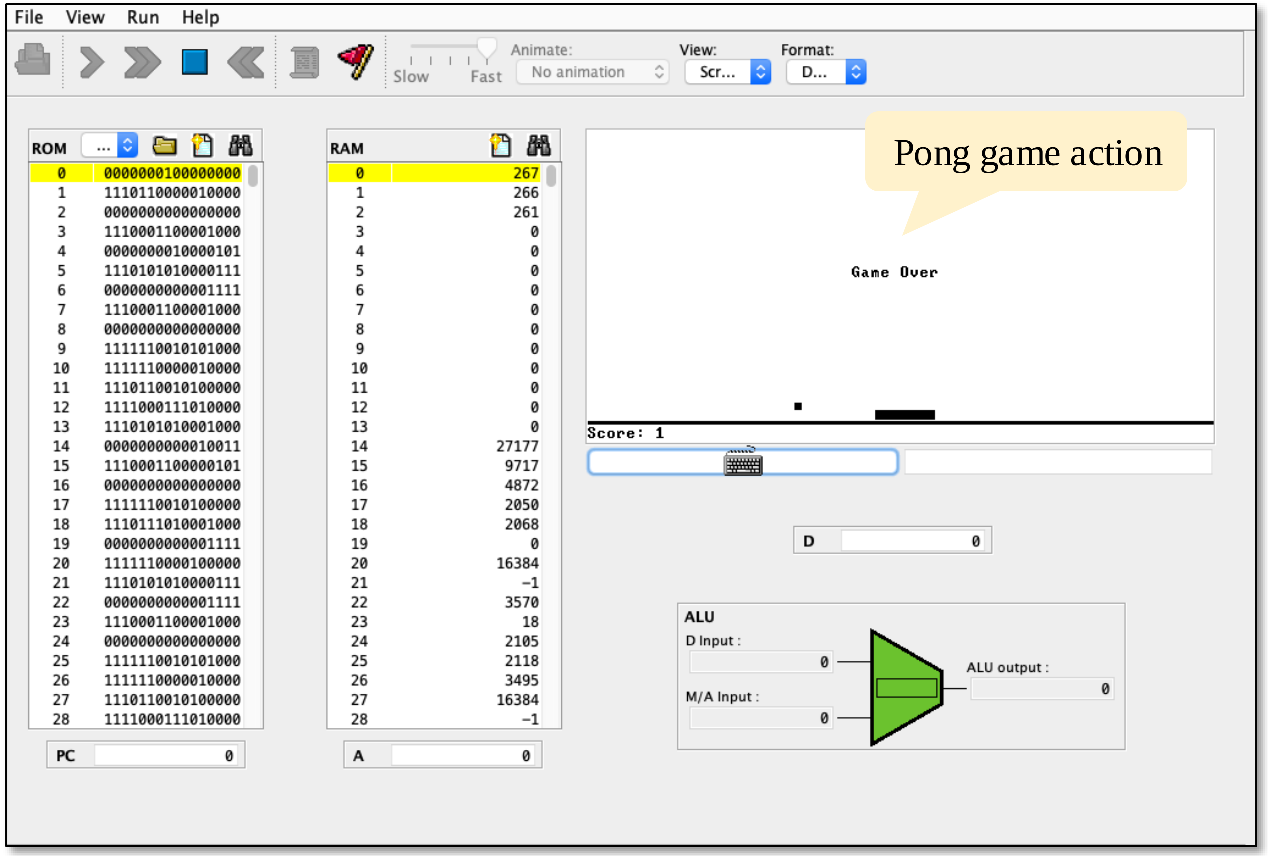


1. Translate `Rect.asm`
2. Load `Rect.hack`
3. Put a non-negative value in `R0`, run the code, inspect the screen.

Testing: Pong

Pong.asm

```
// Pong game
@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
...
```



Translate `Pong.asm`, load `Pong.hack`, and play the game:

Set the speed slider to "fast", and run the code;
Control the game's paddle using the left- and right-arrow keys.

Testing: Pong

Pong.asm

```
// Pong game
@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
...
```

28,374 instructions

Background

The source Pong program was written in the high-level Jack language;

The computer's operating system is also written in Jack;

The Pong code + the OS code were compiled by the Jack compiler, creating a single file named Pong.asm;

This file contains many compiler-generated labels and symbols.

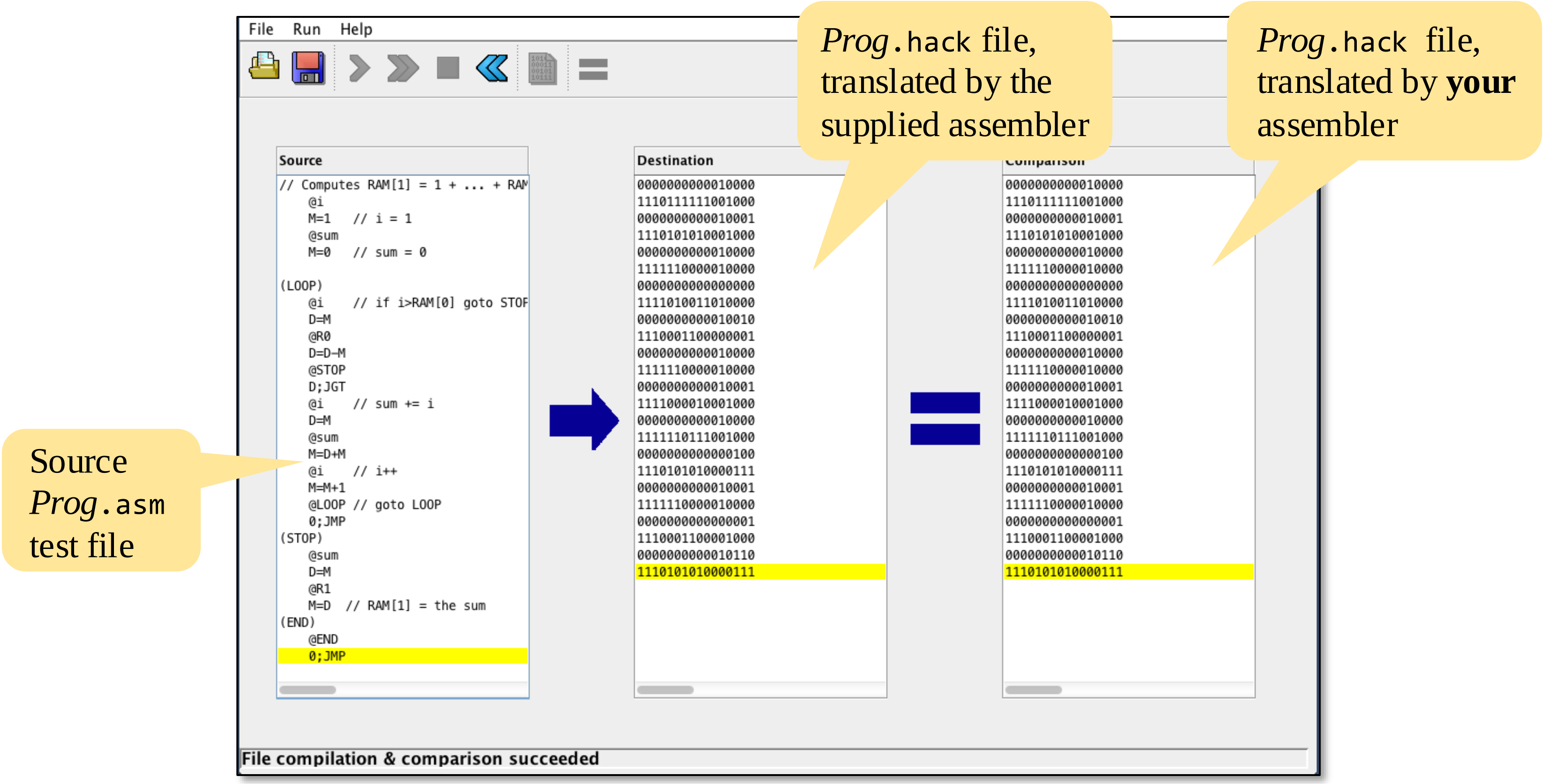
Testing option II: Using the hardware simulator

(Not necessarily recommended, but possible)

1. Use your assembler to translate *Prog*`.asm`, generating the executable file *Prog*`.hack`
2. Put the *Prog*`.hack` file in your project 5 folder
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

(Not necessarily recommended, but possible)



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 code comparison feedback messages.

Chapter 6: Assembler

- Overview
- Translating instructions
- Translating programs
- Handling symbols
- Assembler architecture
- Assembler API
- Project 6
- Some history

Industrial revolution (1760 – 1840)

Before 1800: Labor intensive economy / slavery

Case in point: The textile industry

Picking cotton

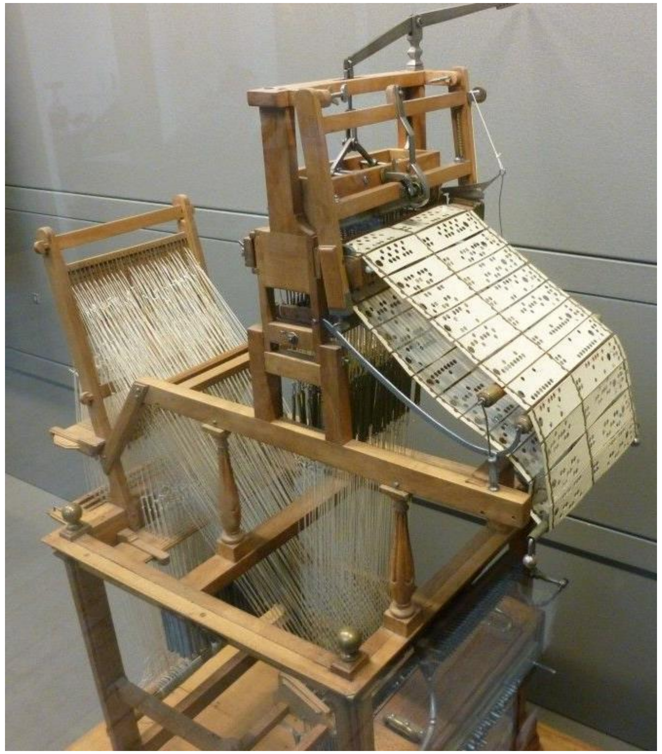
cleaning

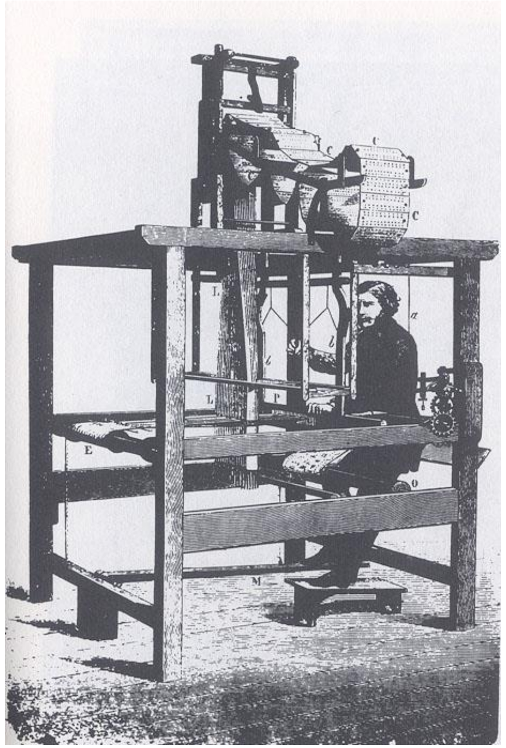
spinning

dyeing

weaving

Industrial revolution (1760 – 1840)



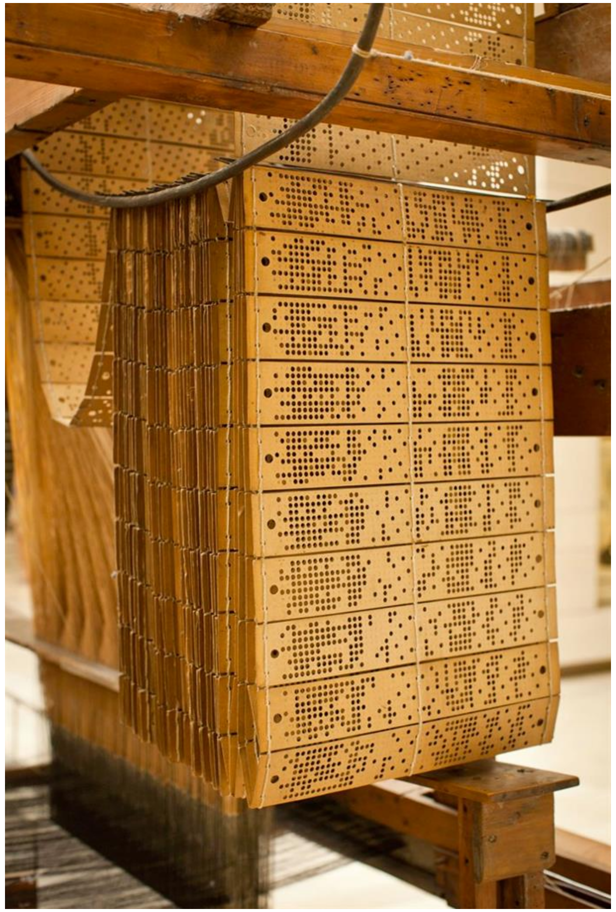


Jacquard loom (1801)

Weaving instructions programmed by punched cards;

The punched cards controlled the loom’s hardware.

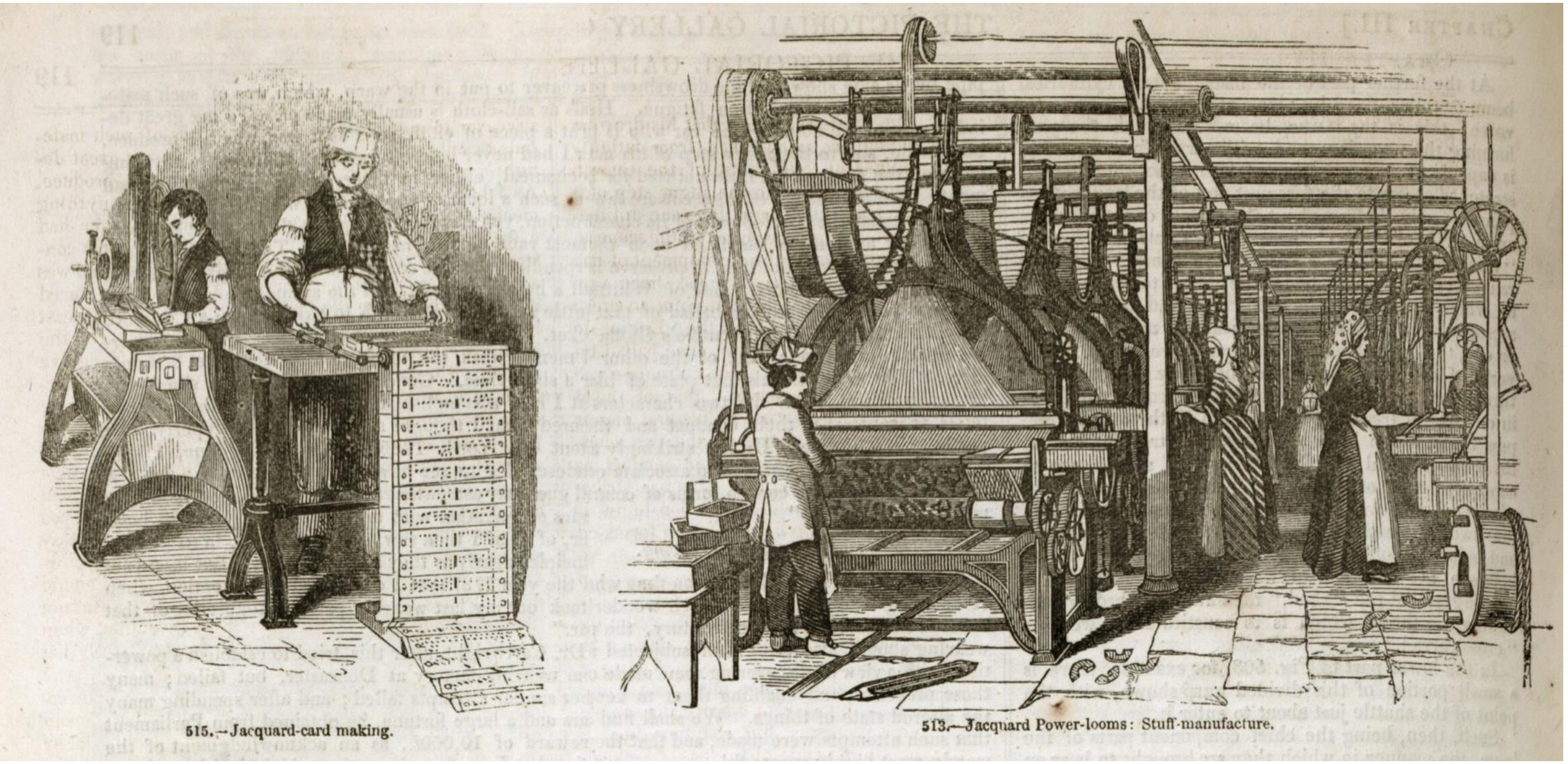
Industrial revolution (1760 – 1840)



Weaving instructions programmed by punched cards;

The punched cards controlled the loom's hardware.

Industrial revolution (1760 – 1840)



Weaving instructions programmed by punched cards;

The punched cards controlled the loom’s hardware.

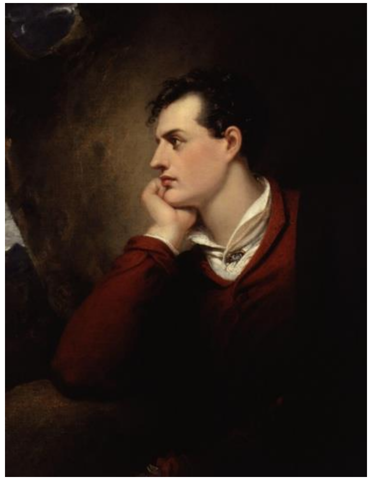
Industrial revolution (1760 – 1840)

Weaving instructions programmed by punched cards;

The punched cards controlled the loom’s hardware.

Industrial revolution (1760 – 1840)

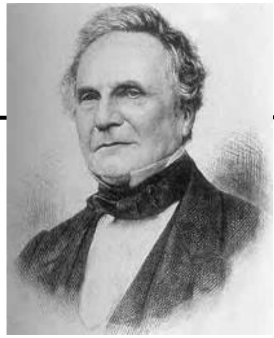
“Luddites” (technology opponents) attacking looms



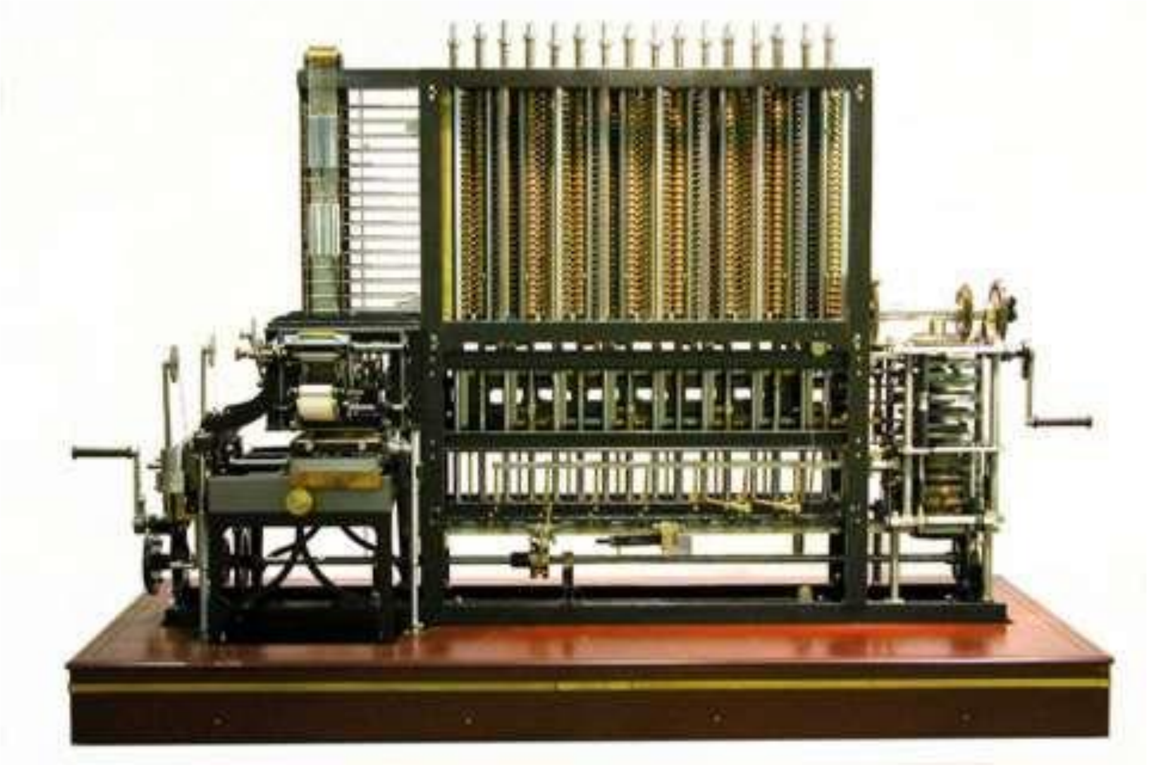
“… While these outrages must be admitted to exist to an alarming extent, it cannot be denied that they have arisen from circumstances of the most unparalleled distress. The perseverance of these miserable men … tends to prove that nothing but absolute want could have driven a large and once honest and industrious body of the people into the commission of excesses so hazardous to themselves, their families, and the community.

The rejected workmen, in the blindness of their ignorance, instead of rejoicing at these improvements in arts so beneficial to mankind, conceived themselves to be sacrificed to improvements in mechanism” (Lord Byron, 1812)

The Analytic Engine 1837



Charles
Babbage
1791-1871



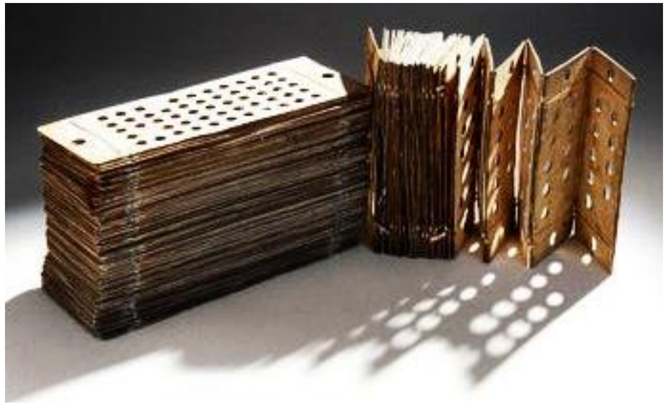
An early mechanical computer,

Designed to tabulate data collected in a UK national census

Inspired by Jacquard’s loom;

Featured a simple programming model with conditional branching;

Software = punched cards.



Ada Lovelace (1815 – 1852)

Ada's insight:

If you want to code instructions, don't start by punching cards (low-level programming);

Instead, use a *symbolic* language for expressing instructions (high-level programming)

Write and test your program *on paper*, using symbolic instructions (debugging)

Only when convinced that the program is error-free, translate the symbolic instructions into punched cards (compiling)

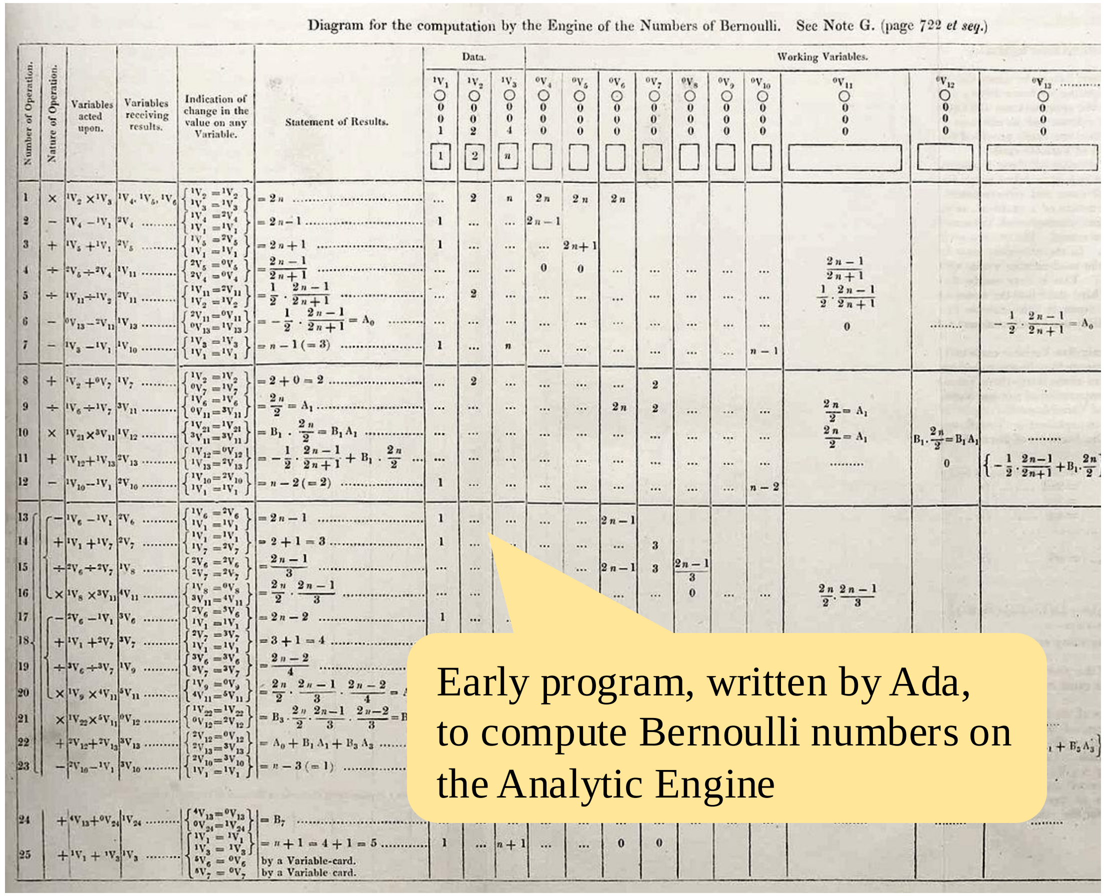
Invention of programming

Gifted mathematician and writer

Worked closely with Babbage on early computers

Byron's daughter...

Ada Lovelace (1815 – 1852)



Early program, written by Ada, to compute Bernoulli numbers on the Analytic Engine

- Often described as "the first programmer"
- The programming language Ada is named after her.

Ada Lovelace (1815 – 1852)

"The Analytical Engine might act upon other things besides numbers...

Suppose, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptations...

If so, the engine might compose elaborate and scientific pieces of music of any degree of complexity."

(1840!)