Chapter 4

Machine Language

These slides support chapter 4 of the book

*The Elements of Computing Systems*

By Noam Nisan and Shimon Schocken

MIT Press
Machine Language: lecture plan

Machine languages

- Basic elements
- The Hack computer and machine language
- The Hack language specification
- Input / Output
- Hack programming
- Project 4 overview
Computers are flexible

Same hardware can run many different software programs
Universality

Same **hardware** can run many different **software** programs

**Theory**

Alan Turing: Universal Turing Machine

**Practice**

John Von Neumann: Stored Program Computer
Stored program concept

Computer System

Memory

program

data

CPU

ALU

registers

input

output
Stored program concept

Computer System

Memory

\[
\begin{align*}
0 & : 0101110011100110 \\
1 & : 1011000101010100 \\
2 & : 111000101111100 \\
\cdots & : \cdots \\
n & : 1100101010010101 \\
n+1 & : 1100100101100111 \\
n+2 & : 0011001010101011 \\
\cdots & : \cdots
\end{align*}
\]

CPU

ALU

registers

output

instructions

input

data
Machine language

Computer System

Memory

<table>
<thead>
<tr>
<th>instructions</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>output</td>
</tr>
</tbody>
</table>

- ALU
- CPU
- registers

- Memory:
  - 0: 0101110011100110
  - 1: 1011000101010100
  - 2: 110001011111100
  - ... (n, n+1, n+2, ...)
Machine language

Handling instructions:
- 1011 means “addition”
- 0001010100 means “operate on memory address 340”
- Next we have to execute the instruction at address 2
Compilation

high-level program

```
while (n < 100) {
    sum += arr[i];
    n++
}
```

machine language

```
0101111100111100
1010101010101010
1101011010101010
1001101010010101
1101010010101010
1110010100100100
0011001010010101
1100100111000100
1100011001100101
0010111001010101
...
```

load and execute
Mnemonics

Instruction:

```
add R3 R2
1011000011000010
```

**Interpretation 1:**

- The symbolic form \texttt{add R3 R2} doesn’t really exist
- It is just a convenient mnemonic that can be used to present machine language instructions to humans
Mnemonics

Instruction:

```
1011000011000010
```

```
add  R3  R2
```

Sample instruction

**Interpretation 2:**

- Allow humans to write symbolic machine language instructions, using *assembly language*
- Use an *assembler* program to translate the symbolic code into binary form.
Symbols

Instruction:

```
1011000110000001
```

```
add  1  Mem[129]
```

Assembly:

```
add 1, Mem[129]
```

```
add 1, index
```

Friendlier syntax: we assume that index stands for Mem[129]

The assembler will resolve the symbol index into a specific address.
Machine Language: lecture plan

- Machine languages
- Basic elements
  - The Hack computer and machine language
  - The Hack language specification
  - Input / Output
  - Hack programming
  - Project 4 overview
Machine language

• Specification of the hardware/software interface:
  □ What supported operations?
  □ What do they operate on?
  □ How is the program controlled?

• Usually in close correspondence to the hardware architecture
  □ But not necessarily so

• Cost-performance tradeoffs:
  □ Silicon area
  □ Time to complete instruction.
Machine operations

- Usually correspond to the operations that the hardware is designed to support:
  - Arithmetic operations: add, subtract, …
  - Logical operations: and, or, …
  - Flow control: “goto instruction $n$”
    - “if (condition) then goto instruction $n$”

- Differences between machine languages:
  - Instruction set richness (division? bulk copy? …)
  - Data types (word width, floating point…).
How does the language allow us to specify on which data the instruction should operate?
Memory hierarchy

- Accessing a memory location is expensive:
  - Need to supply a long address
  - Getting the memory contents into the CPU takes time

- Solution: memory hierarchy:
Registers
Registers

- The CPU typically contains a few, easily accessed, registers
- Their number and functions are a central part of the machine language

Data registers:
add R1, R2
Registers

- The CPU typically contains a few, easily accessed, *registers*
- Their number and functions are a central part of the machine language

Data registers:

```plaintext
add R1, R2
```
Registers

- The CPU typically contains a few, easily accessed, *registers*
- Their number and functions are a central part of the machine language

Data registers:
add R1, R2

Address registers:
store R1, @A
Registers

- The CPU typically contains a few, easily accessed, *registers*
- Their number and functions are a central part of the machine language

**Data registers:**

add R1, R2

**Address registers:**

store R1, @A
Addressing modes

**Register**

\[
\text{add } R1, R2 \quad \text{// } R2 \leftarrow R2 + R1
\]

**Direct**

\[
\text{add } R1, M[200] \quad \text{// } \text{Mem}[200] \leftarrow \text{Mem}[200] + R1
\]

**Indirect**

\[
\text{add } R1, @A \quad \text{// } \text{Mem}[A] \leftarrow \text{Mem}[A] + R1
\]

**Immediate**

\[
\text{add } 73, R1 \quad \text{// } R1 \leftarrow R1 + 73
\]
Input / Output

• Many types of input and output devices:
  - Keyboard, mouse, camera, sensors, printers, screen, sound…

• The CPU needs some agreed-upon protocol to talk to each of them
  - Software drivers realize these protocols

• One general method of interaction uses *memory mapping*:
  - Memory location 12345 holds the direction of the last movement of the mouse
  - Memory location 45678 tells the printer to print single-side or double side
  - Etc.
Flow control

How does the language allow us to decide, and specify, which instruction to process next?
Flow control

• Usually the CPU executes machine instructions in sequence

• Sometimes we need to “jump” unconditionally to another location, e.g. in order to implement a loop:

Example:

101: \texttt{load R1,0}
102: \texttt{add 1, R1}
103: ...
...
// do something with R1 value
...
156: \texttt{jmp 102 \-comment{\texttt{// goto 102}}}

Symbolic version:

\begin{verbatim}
load R1,0
LOOP:
  add 1, R1
  ...
  \comment{\texttt{// do something with R1 value}}
  ...
  \texttt{jmp LOOP \comment{\texttt{// goto loop}}}
\end{verbatim}
Flow control

• Usually the CPU executes machine instructions in sequence

• Sometimes we need to “jump” unconditionally to another location, e.g. in order to implement a loop

• Sometimes we need to jump only if some condition is met:

Example:

```assembly
jgt R1, 0, CONT // if R1>0 jump to CONT
sub R1, 0, R1  // R1 ← (0 - R1)
CONT:
...
// Do something with positive R1
```
Machine Language: lecture plan

- Machine languages
- Basic elements
  - The Hack computer and machine language
    - The Hack language specification
    - Input / Output
    - Hack programming
    - Project 4 overview
Hack computer: hardware

A 16-bit machine consisting of:

- Data memory (RAM): a sequence of 16-bit registers:
  \[ \text{RAM}[0], \text{RAM}[1], \text{RAM}[2], \ldots \]

- Instruction memory (ROM): a sequence of 16-bit registers:
  \[ \text{ROM}[0], \text{ROM}[1], \text{ROM}[2], \ldots \]

- Central Processing Unit (CPU): performs 16-bit instructions

- Instruction bus / data bus / address buses.
Hack computer: software

Hack machine language:
- 16-bit A-instructions
- 16-bit C-instructions

Hack program = sequence of instructions written in the Hack machine language
Hack computer: control

Control:
- The ROM is loaded with a Hack program
- The reset button is pushed
- The program starts running
The Hack machine language recognizes three 16-bit registers:

- **D**: used to store data
- **A**: used to store data / address the memory
- **M**: represents the currently addressed memory register: \( M = \text{RAM}[A] \)
The A-instruction

Syntax:

@value

Where value is either:

- a non-negative decimal constant or
- a symbol referring to such a constant (later)

Semantics:

- Sets the A register to value
- Side effects:
  - RAM[A] becomes the selected RAM register
  - ROM[A] becomes the selected ROM register

Example:

// Sets A to 17
@17
The C-instruction

Syntax: \[ \text{dest} = \text{comp} ; \text{jump} \] (both \text{dest} and \text{jump} are optional)

where:

\[ \text{comp} = 0, 1, -1, D, A, !D, !A, -D, -A, D+1, A+1, D-1, A-1, D+A, D-A, A-D, D\&A, D\mid A \]
\[ M, !M, -M, M+1, M-1, D+M, D-M, M-D, D\&M, D\mid M \]

\[ \text{dest} = \text{null}, M, D, MD, A, AM, AD, AMD \] (\( M \) refers to \( \text{RAM}[A] \))

\[ \text{jump} = \text{null}, \text{JGT}, \text{JEQ}, \text{JGE}, \text{JLT}, \text{JNE}, \text{JLE}, \text{JMP} \]

Semantics:

• Computes the value of \text{comp}
• Stores the result in \text{dest}
• If the Boolean expression \((\text{comp} \, \text{jump} \, 0)\) is true, jumps to execute the instruction at \( \text{ROM}[A] \)
The C-instruction

Syntax: \[ \textit{dest} = \textit{comp} ; \textit{jump} \] (both \textit{dest} and \textit{jump} are optional)

where:

\[
\]

\[
\textit{dest} = \text{null, M, D, MD, A, AM, AD, AMD}
\]

\[
\textit{jump} = \text{null, JGT, JEQ, JGE, JLT, JNE, JLE, JMP}
\]

Semantics:
- Computes the value of \textit{comp}
- Stores the result in \textit{dest}
- If the Boolean expression \((\text{comp} \text{ jump} 0)\) is true, jumps to execute the instruction at \text{ROM}[A]

Example:

// Sets the D register to -1
D=-1
The C-instruction

Syntax:  
\[ \text{dest} = \text{comp} \ ; \ \text{jump} \]  
(both \text{dest} and \text{jump} are optional)

where:

\( \text{comp} = \) 
\[ 0, 1, -1, D, A, \neg D, \neg A, -D, -A, D+1, D-1, A-1, D+A, D-A, A-D, D\&A, D|A \]
\[ \text{M, } \neg \text{M, } -\text{M, } \text{M+1, M-1, D+M, D-M, M-D, D\&M, D|M} \]

\( \text{dest} = \) 
\[ \text{null, M, D, MD, A, AM, AD, AMD} \]  
(M refers to RAM[A])

\( \text{jump} = \) 
\[ \text{null, JGT, JEQ, JGE, JLT, JNE, JLE, JMP} \]

Semantics:
• Computes the value of \text{comp}
• Stores the result in \text{dest}
• If the Boolean expression \((\text{comp \ jump 0})\) is true, 
jumps to execute the instruction at ROM[A]

Example:

// Sets RAM[300] to the value of the D register plus 1
@300  // A = 300
M=D+1  // RAM[300] = D + 1
The C-instruction

Syntax: $dest = comp ; jump$ (both $dest$ and $jump$ are optional)

where:


$dest = \{ \text{null, M, D, MD, A, AM, AD, AMD} \}$ (M refers to RAM[A])

$jump = \{ \text{null, JGT, JEQ, JGE, JLT, JNE, JLE, JMP} \}$

Semantics:

• Computes the value of $comp$
• Stores the result in $dest$
• If the Boolean expression ($comp \ jump \ 0$) is true, jumps to execute the instruction at ROM[A]

Example:

```c
// If (D-1 == 0) jumps to execute the instruction stored in ROM[56]
@56  // A = 56
D-1;JEQ  // if (D-1 == 0) goto to instruction ROM[A]
```
Machine Language: lecture plan

- Machine languages
- Basic elements
- The Hack computer and machine language

The Hack language specification
- Input / Output
- Hack programming
- Project 4 overview
Hack machine language

Two ways to express the same semantics:

Symbolic:

@17  
D+1; JLE

Binary:

\[ \begin{align*}
0000000000010001 \\
111001111100110
\end{align*} \]

translate load & execute
A-instruction specification

**Semantics:** Sets the A register to \textit{value}

**Symbolic syntax:**

- \( @\textit{value} \)

Where \textit{value} is either:

- a non-negative decimal constant \( \leq 65535 (=2^{15}-1) \) or
- a symbol referring to a constant (later)

**Binary syntax:**

- \( 0\textit{value} \)

Where \textit{value} is a 15-bit binary constant

**Example:**

- \( @\texttt{21} \)
- \texttt{0000000000010101} \\

Sets A to 21
C-instruction specification

Symbolic syntax: \[ \text{dest} = \text{comp} ; \text{jump} \]

Binary syntax:

- **opcode**
- not used
- **comp** bits
- **dest** bits
- **jump** bits
C-instruction specification

Symbolic syntax: \[ dest = \textit{comp} ; \textit{jump} \]

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

| \textit{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 |

\[a==0\] \[a==1\]
C-instruction specification

Symbolic syntax:
\[ dest = \text{comp} ; \text{jump} \]

Binary syntax:
\[
\begin{array}{cccccccccccc}
1 & 1 & 1 & a & c1 & c2 & c3 & c4 & c5 & c6 & d1 & d2 & d3 & j1 & j2 & j3
\end{array}
\]

<table>
<thead>
<tr>
<th>( dest )</th>
<th>( d1 )</th>
<th>( d2 )</th>
<th>( d3 )</th>
<th>effect: the value is stored in:</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>The value is not stored</td>
</tr>
<tr>
<td>M</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>RAM[A]</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>D register</td>
</tr>
<tr>
<td>MD</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>RAM[A] and D register</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A register</td>
</tr>
<tr>
<td>AM</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>A register and RAM[A]</td>
</tr>
<tr>
<td>AD</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>A register and D register</td>
</tr>
<tr>
<td>AMD</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>A register, RAM[A], and D register</td>
</tr>
</tbody>
</table>
C-instruction specification

Symbolic syntax:  
\[ dest = \text{comp} ; \text{jump} \]

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

<table>
<thead>
<tr>
<th>jump</th>
<th>j1</th>
<th>j2</th>
<th>j3</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>no jump</td>
</tr>
<tr>
<td>JGT</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>if out&gt;0 jump</td>
</tr>
<tr>
<td>JEQ</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>if out=0 jump</td>
</tr>
<tr>
<td>JGE</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>if out≥0 jump</td>
</tr>
<tr>
<td>JLT</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>if out&lt;0 jump</td>
</tr>
<tr>
<td>JNE</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>if out≠0 jump</td>
</tr>
<tr>
<td>JLE</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>if out≤0 jump</td>
</tr>
<tr>
<td>JMP</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>unconditional jump</td>
</tr>
</tbody>
</table>
C-instruction specification

Symbolic syntax: \[ dest = \texttt{comp} ; \texttt{jump} \]

Binary syntax: \[111a \texttt{c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3}\]

<table>
<thead>
<tr>
<th>( \texttt{comp} )</th>
<th>( c1 \ c2 \ c3 \ c4 \ c5 \ c6 )</th>
<th>( \texttt{dest} )</th>
<th>( d1 \ d2 \ d3 )</th>
<th>effect: the value is stored in:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 0 1 0 1 0</td>
<td>null</td>
<td>0 0 0</td>
<td>The value is not stored</td>
</tr>
<tr>
<td>1</td>
<td>1 1 1 0 1 0</td>
<td>M</td>
<td>0 0 1</td>
<td>( \text{RAM}[A] )</td>
</tr>
<tr>
<td>-1</td>
<td>1 1 1 0 1 0</td>
<td>D</td>
<td>0 1 0</td>
<td>( D ) register</td>
</tr>
<tr>
<td>D</td>
<td>0 0 1 1 0 0</td>
<td>MD</td>
<td>0 1 1</td>
<td>( \text{RAM}[A] ) and ( D ) register</td>
</tr>
<tr>
<td>A</td>
<td>M</td>
<td>1 1 0 0 0 0</td>
<td>A</td>
<td>1 0 0</td>
</tr>
<tr>
<td>!D</td>
<td>0 0 1 1 0 1</td>
<td>AM</td>
<td>1 0 1</td>
<td>( A ) register and ( \text{RAM}[A] )</td>
</tr>
<tr>
<td>!A</td>
<td>!M</td>
<td>1 1 0 0 0 1</td>
<td>AD</td>
<td>1 1 0</td>
</tr>
<tr>
<td>-D</td>
<td>0 0 1 1 1 1</td>
<td>AMD</td>
<td>1 1 1</td>
<td>( A ) register, ( \text{RAM}[A] ), and ( D ) register</td>
</tr>
<tr>
<td>-A</td>
<td>-M</td>
<td>1 1 0 0 1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D+1</td>
<td>0 1 1 1 1 1</td>
<td>jump</td>
<td>j1 j2 j3</td>
<td>effect:</td>
</tr>
<tr>
<td>A+1</td>
<td>M+1</td>
<td>1 1 0 1 1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D-1</td>
<td>0 0 1 1 1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A-1</td>
<td>M-1</td>
<td>1 1 0 0 1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D+A</td>
<td>D+M</td>
<td>0 0 0 0 1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D-A</td>
<td>D-M</td>
<td>0 1 0 0 1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A-D</td>
<td>M-D</td>
<td>0 0 0 1 1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D&amp;A</td>
<td>D&amp;M</td>
<td>0 0 0 0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>M</td>
<td>0 1 0 1 0 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( a=0 \ a=1 \)

Examples:

Symbolic: \[ \text{MD}=D+1 \]

Binary: \[1110011111011000\]
C-instruction specification

Symbolic syntax: \( dest = \text{comp} ; \text{jump} \)

Binary syntax: \( 1 \ 1 \ 1 \ \text{a} \ \text{c1} \ \text{c2} \ \text{c3} \ \text{c4} \ \text{c5} \ \text{c6} \ \text{d1} \ \text{d2} \ \text{d3} \ \text{j1} \ \text{j2} \ \text{j3} \)

<table>
<thead>
<tr>
<th>comp</th>
<th>c1 c2 c3 c4 c5 c6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 0 1 0 1 0</td>
</tr>
<tr>
<td>1</td>
<td>1 1 1 1 1 1</td>
</tr>
<tr>
<td>-1</td>
<td>1 1 1 0 1 0</td>
</tr>
<tr>
<td>D</td>
<td>0 0 1 1 0 0</td>
</tr>
<tr>
<td>A</td>
<td>M 1 1 0 0 0</td>
</tr>
<tr>
<td>!D</td>
<td>0 0 1 1 0 1</td>
</tr>
<tr>
<td>!A</td>
<td>!M 1 1 0 0 1</td>
</tr>
<tr>
<td>-D</td>
<td>0 0 1 1 1 1</td>
</tr>
<tr>
<td>-A</td>
<td>-M 1 1 0 0 1</td>
</tr>
<tr>
<td>A+1</td>
<td>M+1 1 1 1 1</td>
</tr>
<tr>
<td>D+1</td>
<td>0 1 1 1 1 1</td>
</tr>
<tr>
<td>A-1</td>
<td>M-1 1 1 0 1 0</td>
</tr>
<tr>
<td>D-1</td>
<td>0 0 1 1 1 0</td>
</tr>
<tr>
<td>A-1</td>
<td>M-1 1 1 0 1 0</td>
</tr>
<tr>
<td>D+1</td>
<td>0 1 1 1 1 1</td>
</tr>
<tr>
<td>D+1</td>
<td>0 0 0 0 0 1</td>
</tr>
<tr>
<td>D-1</td>
<td>0 1 0 0 0 1</td>
</tr>
<tr>
<td>A-1</td>
<td>M-D 0 0 0 1 1</td>
</tr>
<tr>
<td>D&amp;A</td>
<td>D&amp;M 0 0 0 0 0</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dest</th>
<th>d1 d2 d3</th>
<th>effect: the value is stored in:</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>0 0 0</td>
<td>The value is not stored</td>
</tr>
<tr>
<td>M</td>
<td>0 0 1</td>
<td>RAM[A]</td>
</tr>
<tr>
<td>D</td>
<td>0 1 0</td>
<td>D register</td>
</tr>
<tr>
<td>MD</td>
<td>0 1 1</td>
<td>RAM[A] and D register</td>
</tr>
<tr>
<td>A</td>
<td>1 0 0</td>
<td>A register</td>
</tr>
<tr>
<td>AM</td>
<td>1 0 1</td>
<td>A register and RAM[A]</td>
</tr>
<tr>
<td>AD</td>
<td>1 1 0</td>
<td>A register and D register</td>
</tr>
<tr>
<td>AMD</td>
<td>1 1 1</td>
<td>A register, RAM[A], and D register</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>jump</th>
<th>j1 j2 j3</th>
<th>effect:</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>0 0 0</td>
<td>no jump</td>
</tr>
<tr>
<td>JGT</td>
<td>0 0 1</td>
<td>if out &gt; 0 jump</td>
</tr>
<tr>
<td>JEQ</td>
<td>0 1 0</td>
<td>if out = 0 jump</td>
</tr>
<tr>
<td>JGE</td>
<td>0 1 1</td>
<td>if out ≥ 0 jump</td>
</tr>
<tr>
<td>JLT</td>
<td>1 0 0</td>
<td>if out &lt; 0 jump</td>
</tr>
<tr>
<td>JNE</td>
<td>1 0 1</td>
<td>if out ≠ 0 jump</td>
</tr>
<tr>
<td>JLE</td>
<td>1 1 0</td>
<td>if out ≤ 0 jump</td>
</tr>
<tr>
<td>JMP</td>
<td>1 1 1</td>
<td>Unconditional jump</td>
</tr>
</tbody>
</table>

Examples: Symbolic: \( M=1 \)  
Binary: \( 1110111111001000 \)
C-instruction specification

Symbolic syntax: \[ dest = \text{comp} ; \text{jump} \]

Binary syntax: \[ \text{1 1 1  a  c1  c2  c3  c4  c5  c6  d1  d2  d3  j1  j2  j3} \]

<table>
<thead>
<tr>
<th>comp</th>
<th>c1 c2 c3 c4 c5 c6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 0 1 0 1 0</td>
</tr>
<tr>
<td>1</td>
<td>1 1 1 1 1 1</td>
</tr>
<tr>
<td>-1</td>
<td>1 1 1 0 1 0</td>
</tr>
<tr>
<td>D</td>
<td>0 0 1 1 0 0</td>
</tr>
<tr>
<td>A</td>
<td>M 1 1 0 0 0</td>
</tr>
<tr>
<td>!D</td>
<td>0 0 1 1 0 1</td>
</tr>
<tr>
<td>!A</td>
<td>!M 1 1 0 0 0</td>
</tr>
<tr>
<td>-D</td>
<td>0 0 1 1 1 1</td>
</tr>
<tr>
<td>-A</td>
<td>-M 1 1 0 0 1</td>
</tr>
<tr>
<td>D+1</td>
<td>0 1 1 1 1 1</td>
</tr>
<tr>
<td>A+1</td>
<td>M+1 1 1 0 1 1</td>
</tr>
<tr>
<td>D-1</td>
<td>0 0 1 1 1 0</td>
</tr>
<tr>
<td>A-1</td>
<td>M-1 1 1 0 1 0</td>
</tr>
<tr>
<td>D+A</td>
<td>D+M 0 0 0 1 0</td>
</tr>
<tr>
<td>D-A</td>
<td>D-M 0 1 0 0 1</td>
</tr>
<tr>
<td>A-D</td>
<td>M-D 0 0 0 1 1</td>
</tr>
<tr>
<td>D&amp;A</td>
<td>D&amp;M 0 0 0 0 0</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
</tr>
<tr>
<td>a==0</td>
<td>a==1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dest</th>
<th>d1 d2 d3</th>
<th>effect: the value is stored in:</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>0 0 0</td>
<td>The value is not stored</td>
</tr>
<tr>
<td>M</td>
<td>0 0 1</td>
<td>RAM[A]</td>
</tr>
<tr>
<td>D</td>
<td>0 1 0</td>
<td>D register</td>
</tr>
<tr>
<td>MD</td>
<td>0 1 1</td>
<td>RAM[A] and D register</td>
</tr>
<tr>
<td>A</td>
<td>1 0 0</td>
<td>A register</td>
</tr>
<tr>
<td>AM</td>
<td>1 0 1</td>
<td>A register and RAM[A]</td>
</tr>
<tr>
<td>AD</td>
<td>1 1 0</td>
<td>A register and D register</td>
</tr>
<tr>
<td>AMD</td>
<td>1 1 1</td>
<td>A register, RAM[A], and D register</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>jump</th>
<th>j1 j2 j3</th>
<th>effect:</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>0 0 0</td>
<td>no jump</td>
</tr>
<tr>
<td>JGT</td>
<td>0 0 1</td>
<td>if out &gt; 0 jump</td>
</tr>
<tr>
<td>JEQ</td>
<td>0 1 0</td>
<td>if out = 0 jump</td>
</tr>
<tr>
<td>JGE</td>
<td>0 1 1</td>
<td>if out ≥ 0 jump</td>
</tr>
<tr>
<td>JLT</td>
<td>1 0 0</td>
<td>if out &lt; 0 jump</td>
</tr>
<tr>
<td>JNE</td>
<td>1 0 1</td>
<td>if out ≠ 0 jump</td>
</tr>
<tr>
<td>JLE</td>
<td>1 1 0</td>
<td>if out ≤ 0 jump</td>
</tr>
<tr>
<td>JMP</td>
<td>1 1 1</td>
<td>Unconditional jump</td>
</tr>
</tbody>
</table>

Examples:
- Symbolic: \[ D+1;\text{JLE} \]
- Binary: \[ 1110011111000110 \]
Hack program

Symbolic code

// Computes RAM[1] = 1+...+RAM[0]
// Usage: put a number in RAM[0]
@16 // RAM[16] represents i
M=1 // i = 1
@17 // RAM[17] represents sum
M=0 // sum = 0

@16
D=M
@0
D=D-
M
@17 // if i>RAM[0] goto 17
D;JGT

@16
D=M
@17
M=M+1 // i++
@4 // goto 4 (loop)
0;JMP

@17
D=M
@1
M=D // RAM[1] = sum
@21 // program’s end
0;JMP // infinite loop

Observations:

- Hack program: a sequence of Hack instructions
- White space is permitted
- Comments are welcome
- There are better ways to write symbolic Hack programs; stay tuned.

No need to understand ... we’ll review the code later in the lecture.
### Hack programs: symbolic and binary

#### Symbolic code

```pseudo
// Computes RAM[1] = 1+...+RAM[0]
// Usage: put a number in RAM[0]
@16 // RAM[16] represents i
M=1 // i = 1
@17 // RAM[17] represents sum
M=0 // sum = 0

@16
D=M
@0
D=D-M
@17 // if i>RAM[0] goto 17
D;JGT

@16
D=M
@17
M=D+M // sum += i
@16
M=M+1 // i++
@4 // goto 4 (loop)
0;JMP

@17
D=M
@1
M=D // RAM[1] = sum
@21 // program's end
0;JMP // infinite loop
```

#### Binary code

```
0000000000001000
111011111010000
000000000010001
111010101000100
000000000010000
111111000010000
000000000000000
111101011100000
000000000010001
111000110000001
000000000010000
111111000010000
000000000010000
111010101000011
000000000000010
111010101000011
000000000000010
111111000010000
000000000010000
111111000010000
000000000010000
```

---

[slide 49]
Machine Language: lecture plan

- Machine languages
- Basic elements
- The Hack computer and machine language
- The Hack language specification

Input / Output
  - Hack programming
  - Project 4 overview
Input / output

I/O handling (high-level):
Software libraries enabling text, graphics, audio, video, etc.

I/O handling (low-level):
Bits manipulation.
Memory mapped output

A designated memory area, dedicated to manage a display unit

The physical display is continuously *refreshed* from the memory map, many times per second

Output is effected by writing code that manipulates the screen memory map.
Memory mapped output

To set pixel \((row, col)\) on/off:

1. \(word = \text{RAM}[16384 + 32 \times row + col/16]\)
2. Set the \((col \% 16)th\) bit of \(word\) to 0 or 1
3. \(\text{RAM}[i] = word\)
Memory mapped output
The physical keyboard is associated with a *keyboard memory map*. 
Memory mapped input

When a key is pressed on the keyboard, the key’s *scan code* appears in the *keyboard memory map*.
When no key is pressed, the resulting code is 0.
### The Hack character set

<table>
<thead>
<tr>
<th>key</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>(space)</td>
<td>32</td>
</tr>
<tr>
<td>!</td>
<td>33</td>
</tr>
<tr>
<td>“</td>
<td>34</td>
</tr>
<tr>
<td>#</td>
<td>35</td>
</tr>
<tr>
<td>$</td>
<td>36</td>
</tr>
<tr>
<td>%</td>
<td>37</td>
</tr>
<tr>
<td>&amp;</td>
<td>38</td>
</tr>
<tr>
<td>‘</td>
<td>39</td>
</tr>
<tr>
<td>(</td>
<td>40</td>
</tr>
<tr>
<td>)</td>
<td>41</td>
</tr>
<tr>
<td>*</td>
<td>42</td>
</tr>
<tr>
<td>+</td>
<td>43</td>
</tr>
<tr>
<td>,</td>
<td>44</td>
</tr>
<tr>
<td>-</td>
<td>45</td>
</tr>
<tr>
<td>.</td>
<td>46</td>
</tr>
<tr>
<td>/</td>
<td>47</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>key</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>48</td>
</tr>
<tr>
<td>1</td>
<td>49</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>9</td>
<td>57</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Z</td>
<td>90</td>
</tr>
<tr>
<td>:</td>
<td>58</td>
</tr>
<tr>
<td>;</td>
<td>59</td>
</tr>
<tr>
<td>&lt;</td>
<td>60</td>
</tr>
<tr>
<td>=</td>
<td>61</td>
</tr>
<tr>
<td>&gt;</td>
<td>62</td>
</tr>
<tr>
<td>?</td>
<td>63</td>
</tr>
<tr>
<td>@</td>
<td>64</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>key</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>65</td>
</tr>
<tr>
<td>B</td>
<td>66</td>
</tr>
<tr>
<td>C</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Z</td>
<td>90</td>
</tr>
<tr>
<td>[</td>
<td>91</td>
</tr>
<tr>
<td>]</td>
<td>93</td>
</tr>
<tr>
<td>^</td>
<td>94</td>
</tr>
<tr>
<td>_</td>
<td>95</td>
</tr>
<tr>
<td>\</td>
<td>96</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>key</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>97</td>
</tr>
<tr>
<td>b</td>
<td>98</td>
</tr>
<tr>
<td>c</td>
<td>99</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>z</td>
<td>122</td>
</tr>
<tr>
<td>{</td>
<td>123</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>125</td>
</tr>
<tr>
<td>~</td>
<td>126</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>key</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>newline</td>
<td>128</td>
</tr>
<tr>
<td>backspace</td>
<td>129</td>
</tr>
<tr>
<td>left arrow</td>
<td>130</td>
</tr>
<tr>
<td>up arrow</td>
<td>131</td>
</tr>
<tr>
<td>right arrow</td>
<td>132</td>
</tr>
<tr>
<td>down arrow</td>
<td>133</td>
</tr>
<tr>
<td>home</td>
<td>134</td>
</tr>
<tr>
<td>end</td>
<td>135</td>
</tr>
<tr>
<td>Page up</td>
<td>136</td>
</tr>
<tr>
<td>Page down</td>
<td>137</td>
</tr>
<tr>
<td>insert</td>
<td>138</td>
</tr>
<tr>
<td>delete</td>
<td>139</td>
</tr>
<tr>
<td>esc</td>
<td>140</td>
</tr>
<tr>
<td>f1</td>
<td>141</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>f12</td>
<td>152</td>
</tr>
</tbody>
</table>
Handling the keyboard

To check which key is currently pressed:
• Probe the contents of the Keyboard chip
• In the Hack computer: probe the contents of RAM[24576].
Handling the keyboard
Machine Language: lecture plan

- Machine languages
- Basic elements
- The Hack computer and machine language
- The Hack language specification
- Input / Output
  - Hack programming
  - Project 4 overview
Machine Language: lecture plan

- Machine languages
- Basic elements
- The Hack computer and machine language
- The Hack language specification
- Input / Output
  - Hack programming
    - Part 1: registers and memory
      - Part 2: branching, variables, iteration
      - Part 3: pointers, input/output
  - Project 4 overview
Hack assembly language (overview)

A-instruction:

```
@value     // A = value
```

where value is either a constant or a symbol referring to such a constant

C-instruction:

```
dest = comp ; jump
```

(both dest and jump are optional)

where:

```
      M,     !M,   -M,   M+1,    M-1, D+M, D-M, M-D, D&M, D|M
```

```
dest = null, M, D, MD, A, AM, AD, AMD
      (M refers to RAM[A])
```

```
jump = null, JGT, JEQ, JGE, JLT, JNE, JLE, JMP
```

Semantics:

- Computes the value of comp
- Stores the result in dest
- If the Boolean expression (comp jump 0) is true, jumps to execute the instruction at ROM[A]
Hack assembler

Assembly program

```
// Program: Flip.asm
// flips the values of
// RAM[0] and RAM[1]
@R1
D=M
@temp
M=D   // temp = R1
@R0
D=M
@R1
M=D   // R1 = R0
@temp
D=M
@R0
M=D   // R0 = temp
(END)
@END
0;JMP
```

Binary code

```
0000000000000001 1111110000010000 0000000000100000
1110001100001000 0000000000000000
1111110000010000 0000000000000001
1110001100001000 0000000000100000
1111110000100000 0000000001000000
1110001100000100 0000000000000000
1111110000010000 0000000000000000
1110101010000111
```

We’ll develop a Hack assembler later in the course.
CPU Emulator

Assembly program

```assembly
// Program: Flip.asm
// flips the values of
// RAM[0] and RAM[1]
@R1
D=M
@temp
M=D // temp = R1
@R0
D=M
@R1
M=D // R1 = R0
@temp
D=M
@R0
M=D // R0 = temp
(END)
@end
0;JMP
```

-the simulator software translates from symbolic to binary as it loads-

CPU Emulator

- A software tool
- Convenient for debugging and executing symbolic Hack programs.
Registers and memory

D: data register
A: address / data register
M: the currently selected memory register: $M = \text{RAM}[A]$
Registers and memory

D: data register
A: address / data register
M: the currently selected memory register: \( M = \text{RAM}[A] \)

Typical operations:

```
// D=10
@10
D=A

// D++
D=D+1

// D=\text{RAM}[17]
@17
D=M

// \text{RAM}[17]=D
@17
M=D

// \text{RAM}[17]=10
@10
D=A
@17
M=D

// \text{RAM}[5] = \text{RAM}[3]
@3
D=M
@5
M=D
```
Program example: add two numbers

Hack assembly code

```asm
@0
D=M  // D = RAM[0]

@1
D=D+M  // D = D + RAM[1]

@2
```

Memory (ROM)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>@0</td>
<td>D=M</td>
<td>@1</td>
<td>D=D+M</td>
<td>@2</td>
<td>M=D</td>
</tr>
</tbody>
</table>

(white space ignored)

translate and load

symbolic view

32767
Program example: add two numbers

Hack assembly code:

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

@1
D=D+M // D = D + RAM[1]

@2
```

Memory (ROM):

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>1</td>
<td>1111110000010000</td>
</tr>
<tr>
<td>2</td>
<td>0000000000000001</td>
</tr>
<tr>
<td>3</td>
<td>1111000101010000</td>
</tr>
<tr>
<td>4</td>
<td>0000000000001000</td>
</tr>
<tr>
<td>5</td>
<td>1110001100001000</td>
</tr>
</tbody>
</table>

// Program: Add2.asm
// Usage: put values in RAM[0], RAM[1]

32767
Program example: add two numbers
Terminating a program

Hack assembly code

// Program: Add2.asm
// Usage: put values in RAM[0], RAM[1]

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

@1
D=D+M  // D = D + RAM[1]

@2
Terminating a program

Hack assembly code

// Program: Add2.asm
// Usage: put values in RAM[0], RAM[1]

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

@1
D=D+M // D = D + RAM[1]

@2

Memory (ROM)

0 @0
1 D=M
2 @1
3 D=D+M
4 @2
5 M=D
6 0
7 1
8 2
9 3
10 4
11 5
12 6
13 7
14 8
15 9

malicious code starts here...

Resulting from some attack on the computer
Terminating a program

Hack assembly code

```
// Program: Add2.asm
// Usage: put values in RAM[0], RAM[1]

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

@1
D=D+M
  // D = D + RAM[1]

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

@6
0;JMP
  // Jump to instruction number A (which happens to be 6)
  // 0: syntax convention for jmp instructions
```

Memory (ROM)

```
0 @0
1 D=M
2 @1
3 D=D+M
4 @2
5 M=D
6 @6
7 0;JMP
```

Best practice:

To terminate a program safely, end it with an infinite loop.
Built-in symbols

The Hack assembly language features *built-in symbols*:

<table>
<thead>
<tr>
<th>symbol</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>0</td>
</tr>
<tr>
<td>R1</td>
<td>1</td>
</tr>
<tr>
<td>R2</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>R15</td>
<td>15</td>
</tr>
</tbody>
</table>

Attention: Hack is case-sensitive! R5 and r5 are different symbols.

These symbols can be used to denote “virtual registers”

Example: suppose we wish to use \texttt{RAM[5]} to represent some variable, say \(x\), and we wish to let \(x=7\)

implementation:

```
// let RAM[5] = 7
@7
D=A
@5
M=D
```

better style:

```
// let RAM[5] = 7
@7
D=A
@R5
M=D
```
Built-in symbols

The Hack assembly language features *built-in symbols*:

<table>
<thead>
<tr>
<th>symbol</th>
<th>value</th>
<th>symbol</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>0</td>
<td>SP</td>
<td>0</td>
</tr>
<tr>
<td>R1</td>
<td>1</td>
<td>LCL</td>
<td>1</td>
</tr>
<tr>
<td>R2</td>
<td>2</td>
<td>ARG</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>THIS</td>
<td>3</td>
</tr>
<tr>
<td>R15</td>
<td>15</td>
<td>THAT</td>
<td>4</td>
</tr>
<tr>
<td>SCREEN</td>
<td>16384</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KBD</td>
<td>24576</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- R0, R1, ..., R15: “virtual registers”, can be used as variables
- SCREEN and KBD: base addresses of I/O memory maps
- Remaining symbols: used in the implementation of the Hack *virtual machine*, discussed in chapters 7-8.
Machine Language: lecture plan

- Machine languages
- Basic elements
- The Hack computer and machine language
- The Hack language specification
- Input / Output
  - Hack programming
    - Part 1: registers and memory
    - Part 2: branching, variables, iteration
      - Part 3: pointers, input/output
  - Project 4 overview
Branching

example:

// Program: Signum.asm
// Computes: if R0>0
//           R1=1
// else
//           R1=0
// Usage: put a value in RAM[0],
//        run and inspect RAM[1].

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

@8
D;JGT  // If R0>0 goto 8

@R1
M=0  // RAM[1]=0

@10
0;JMP  // goto end

@R1
M=1  // R1=1

@10
0;JMP
Branching

example:

```
// Program: Signum.asm
// Computes: if R0>0
//           R1=1
//           else
//              R1=0
// Usage: put a value in RAM[0], run and inspect RAM[1].

@R0
D=M    // D = RAM[0]
@8
D;JGT // If R0>0 goto 8

@R1
M=0    // RAM[1]=0
@10
0;JMP // goto end

@R1
M=1    // R1=1
@10
0;JMP
```

“Instead of imagining that our main task as programmers is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.”

– Donald Knuth
Branching

example:

// Program: Signum.asm
// Computes: if R0>0
//          R1=1
// else
//        R1=0
// Usage: put a value in RAM[0],
//        run and inspect RAM[1].

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

@POSITIVE
D;JGT // If R0>0 goto 8

@R1
M=0 // RAM[1]=0
@10
0;JMP // goto end

(POSITIVE)
@R1
M=1 // R1=1

(END)
@END
0;JMP
Labels

example:

```assembly
// Program: Signum.asm
// Computes: if R0>0
// R1=1
// else
// R1=0
// Usage: put a value in RAM[0], run and inspect RAM[1].

@R0
D=M // D = RAM[0]
@POSITIVE
D;JGT // If R0>0 goto 8

@R1
M=0 // RAM[1]=0
@10
0;JMP // goto end

(POSITIVE)
@R1
M=1 // R1=1

(END)
@END
0;JMP
```

Memory

```
@0
D=M

@8  // @POSITIVE
@8
D;JGT

@1
M=0
@10  // @END

@10
M=1
@10 // @END

0;JMP

@1
M=1
@10 // @END

0;JMP
```

Label resolution rules:

- Label declarations generate no code
- Each reference to a label is replaced with a reference to the instruction number following that label’s declaration.
Labels

Implications:

- Instruction numbers no longer needed in symbolic programming
- The symbolic code becomes relocatable.

```
// Program: Signum.asm
// Computes: if R0>0
//           R1=1
// else
//           R1=0
// Usage: put a value in RAM[0],
//        run and inspect RAM[1].

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

@POSITIVE
D;JGT  // If R0>0 goto 8

@R1
M=0    // RAM[1]=0
@10    // goto end
0;JMP

@R1
M=1    // R1=1

(END)
@END
0;JMP
```

```
0
@0

1
D=M

2
@8  // @POSITIVE

3
D;JGT

4
@1

5
M=0

6
@10 // @END

7
0;JMP

8
@1

9
M=1

10
@10 // @END

11
0;JMP

32767
```
Variables

Variable usage example:

```
// Program: Flip.asm
// flips the values of
// RAM[0] and RAM[1]

// temp = R1
// R1 = R0
// R0 = temp

@R1
D=M
@temp
M=D  // temp = R1

@R0
D=M
@R1
M=D  // R1 = R0

@temp
D=M
@R0
M=D  // R0 = temp

(END)

@END
0;JMP
```
Variables

Variable usage example:

```
// Program: Flip.asm
// flips the values of RAM[0] and RAM[1]
// temp = R1
// R1 = R0
// R0 = temp

@R1
D=M
@temp
M=D   // temp = R1

@R0
D=M
@R1
M=D   // R1 = R0

@temp
D=M
@R0
M=D   // R0 = temp

(END)

@END
0;JMP
```

Symbol resolution rules:

- A reference to a symbol that has no corresponding label declaration is treated as a reference to a variable.
- If the reference `@symbol` occurs in the program for the first time, `symbol` is allocated to address 16 onward (say `n`), and the generated code is `@n`
- All subsequent `@symbol` commands are translated into `@n`

In other words: variables are allocated to RAM[16] onward.

Memory

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>@1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>D=M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>@16   // @temp</td>
<td>M=D</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>@0</td>
<td>D=M</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>@0</td>
<td>M=D</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>@16   // @temp</td>
<td>D=M</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>@0</td>
<td>D=M</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>@0</td>
<td>M=D</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>@12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0;JMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32767
Variables

Variable usage example:

// Program: Flip.asm
// flips the values of
// RAM[0] and RAM[1]

// temp = R1
// R1 = R0
// R0 = temp

@R1
D=M
@temp
M=D
// temp = R1

@R0
D=M
@R1
M=D
// R1 = R0

@temp
D=M
@R0
M=D
// R0 = temp

(END)
@END
0;JMP

Implications:

symbolic code is easy
to read and debug

Memory

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>@1</td>
<td>D=M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>@16 // @temp</td>
<td>M=D</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>@0</td>
<td></td>
<td>D=M</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>@1</td>
<td>M=D</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>@16 // @temp</td>
<td></td>
<td>D=M</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>@0</td>
<td></td>
<td>M=D</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>@12</td>
<td></td>
<td>0;JMP</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32767
Iterative processing

pseudo code

// Computes RAM[1] = 1+2+ ... +RAM[0]

n = R0
i = 1
sum = 0

LOOP:
    if i > n goto STOP
    sum = sum + i
    i = i + 1
    goto LOOP

STOP:
    R1 = sum
Iterative processing

pseudo code

// Computes RAM[1] = 1 + 2 + ... + RAM[0]

n = R0
i = 1
sum = 0

...
Iterative processing

pseudo code

```pseudo
def compute_ram1(n):
    i = 1
    sum = 0
    while i <= n:
        sum += i
        i += 1
    return sum

// Computes RAM[1] = 1+2+ ... +RAM[0]
```
Iterative processing

pseudo code

// Computes RAM[1] = 1+2+ ... +RAM[0]

n = R0
i = 1
sum = 0

LOOP:
    if i > n goto STOP
    sum = sum + i
    i = i + 1
    goto LOOP

STOP:
    R1 = sum

assembly program

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

assembly program

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

(STOP)
@sum
D=M
@R1
M=D  // RAM[1] = sum
(END)
@END
0;JMP
```

**iterations**

| 0  | 1  | 2  | 3  | 4  | ...
|----|----|----|----|----|----|
| RAM[0]| 3  | 3  | 6  | 10 | ...
| n  | 3  | 3  | 3  | 3  | ...
| i  | 1  | 2  | 3  | 4  | ...
| sum| 0  | 1  | 3  | 6  | ...

```

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

(STOP)
@sum
D=M
@R1
M=D  // RAM[1] = sum
(END)
@END
0;JMP
```
Writing assembly programs

assembly program

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

Best practice:

- **Design** the program using pseudo code
- **Write** the program in assembly language
- **Test** the program (on paper) using a variable-value trace table
Machine Language: lecture plan

- Machine languages
- Basic elements
- The Hack computer and machine language
- The Hack language specification
- Input / Output
  - Hack programming
    - Part 1: registers and memory
    - Part 2: branching, variables, iteration
    - Part 3: pointers, input/output
  - Project 4 overview
Pointers

Example:

```c
// for (i=0; i<n; i++) {
//      arr[i] = -1
// }
```

Observations:

- The array is implemented as a block of memory registers.
- In order to access these memory registers one after the other, we need a variable that holds the current address.
- Variables that represent addresses are called pointers.
- There is nothing special about pointer variables, except that their values are interpreted as addresses.
Pointers

Example:

```c
// for (i=0; i<n; i++) {
//      arr[i] = -1
// }

// Suppose that arr=100 and n=10

// Let arr = 100
@100
D=A
@arr
M=D

// Let n = 10
@10
D=A
@n
M=D

// Let i = 0
@0
M=0

// Loop code continues
// in next slide...
```
Pointers

Example:

(LOOP)
// if (i==n) goto END
@i
D=M
@n
D=D-M
@END
D;JEQ

// RAM[arr+i] = -1
@arr
D=M
@i
A=D+M
M=-1

...
Pointers

Example:

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

// RAM[arr+i] = -1
@arr
D=M
@i
A=D+M
M=-1

// i++
@i
M=M+1

@LOOP
0;JMP

(END)
@END
0;JMP
```

typical pointer manipulation

- Pointers: Variables that store memory addresses (like `arr`)
- Pointers in Hack: Whenever we have to access memory using a pointer, we need an instruction like `A = expression`
- Semantics: “set the address register to some value”.

Example:

<table>
<thead>
<tr>
<th>RAM</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>arr: 16</td>
<td>100</td>
</tr>
<tr>
<td>n: 17</td>
<td>10</td>
</tr>
<tr>
<td>i: 18</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>101</td>
</tr>
<tr>
<td>101</td>
<td>102</td>
</tr>
<tr>
<td>102</td>
<td>103</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

after 4 iterations

<table>
<thead>
<tr>
<th>RAM</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>arr: 16</td>
<td>100</td>
</tr>
<tr>
<td>n: 17</td>
<td>10</td>
</tr>
<tr>
<td>i: 18</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>101</td>
</tr>
<tr>
<td>101</td>
<td>102</td>
</tr>
<tr>
<td>102</td>
<td>103</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Hack language convention:

- `SCREEN`: base address of the screen memory map
Handling the screen (example)

Task: draw a filled rectangle at the upper left corner of the screen, 16 pixels wide and RAM[0] pixels long.
Handling the screen (example)

CPU Emulator
demo
Handling the screen (example)

Pseudo code

```c
// for (i=0; i<n; i++) {
//     draw 16 black pixels at the
//     beginning of row i
// }

addr = SCREEN
n = RAM[0]
i = 0

LOOP:
    if i > n goto END
    RAM[addr] = -1 // 1111111111111111
    // advances to the next row
    addr = addr + 32
    i = i + 1
    goto LOOP

END:
goto END
```

16 black pixels, corresponding to the first row of the rectangle

Physical screen

Screen memory map
Handling the screen (example)

Assembly code

```
// Program: Rectangle.asm
// Draws a filled rectangle at the
// screen's top left corner, with
// width of 16 pixels and height of
// RAM[0] pixels.
// Usage: put a non-negative number
// (rectangle's height) in RAM[0].

@SCREEN
D=A
@addr
M=D
// addr = 16384
// (screen's base address)
@0
D=M
@n
M=D
// n = RAM[0]
@i
M=0
// i = 0

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

@addr
A=M
M=-1 // RAM[addr]=1111111111111111

@i
M=M+1 // i = i + 1
@32
D=A
@addr
M=D+M // addr = addr + 32
@LOOP
0;JMP // goto LOOP

(END)
@END // program's end
0;JMP // infinite loop
```
Handling the screen (example)

Assembly code

```assembly
@SCREEN
D=A
@addr
M=D

// addr = 16384
// (screen’s base address)

@0
D=M
@n
M=D

// n = RAM[0]

@i
D=M
@n
D=D-M
@END
D;JGT // if i>n goto END

@addr
A=M
M=-1   // RAM[addr]=1111111111111111

@i
M=M+1   // i = i + 1
@32
D=A
@addr
M=D+M   // addr = addr + 32
@LOOP
0;JMP // goto LOOP

(END)

@END // program’s end
0;JMP // infinite loop
```

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

@addr
A=M
M=-1   // RAM[addr]=1111111111111111

@i
M=M+1   // i = i + 1
@32
D=A
@addr
M=D+M   // addr = addr + 32
@LOOP
0;JMP // goto LOOP
```

(continued)
Input

Hack RAM

- data memory (16K)
- screen memory map (8K)
- keyboard map

Hack language convention:

- `SCREEN`: base address of the screen memory map
- `KBD`: address of the keyboard memory map
Handling the keyboard

To check which key is currently pressed:

- Read the contents of $\text{RAM}[24576]$ (address $\text{KBD}$)
- If the register contains 0, no key is pressed
- Otherwise, the register contains the scan code of the currently pressed key.

Scan-code of ‘k’ = 75
High level code

```java
for (i=0; i<n; i++) {
    arr[i] = -1
}
```

Machine language

```
...  
@i  
M=0  
(LOOP)  
    @i  
    D=M  
    @n  
    D=D-M  
    @END  
    D;JEQ  
    @arr  
    D=M  
    @i  
    A=D+M  
    M=1  
    @i  
    M=M+1  
    @LOOP  
    0;JMP  
(END)  
    @END  
    0;JMP
```

Low-level programming is:

- Low level
- Profound
- Subtle
- Efficient (or not)
- Intellectually challenging.
Machine Language: lecture plan

- Machine languages
- Basic elements
- The Hack computer and machine language
- The Hack language specification
- Input / Output
- Hack programming
  - Part 1: registers and memory
  - Part 2: branching, variable, iteration
  - Part 3: pointers, input/output

Project 4 overview
In a Nutshell

Project objectives

Have a taste of:

• low-level programming
• Hack assembly language
• Hack hardware

Tasks

• Write a simple algebraic program
• Write a simple interactive program.
Mult: a program performing $R2 = R0 \times R1$
**Fill: a simple interactive program**
Fill: a simple interactive program
Fill: a simple interactive program

Implementation strategy

- Listen to the keyboard
- To blacken / clear the screen, write code that fills the entire screen memory map with either “white” or “black” pixels
  (Accessing the memory requires working with pointers)

Testing

- Select “no animation”
- Manual testing (no test scripts).
Program development process

1. Write / edit the program using a text editor
   - `Prog.asm`

2. Load the program into the CPU Emulator, and run it

3. Find and fix the errors

4. happy with results?
   - Yes
   - No
Best practice

Well-written low-level code is

• Short
• Efficient
• Elegant
• Self-describing

Technical tips

• Use symbolic variables and labels
• Use sensible variable and label names
• Variables: lower-case
• Labels: upper-case
• Use indentation
• Start with pseudo code.
Project 4 resources

Project 4: Machine Language Programming

<table>
<thead>
<tr>
<th>Background</th>
</tr>
</thead>
<tbody>
<tr>
<td>Each hardware platform is designed to execute a certain machine language, expressed using agreed-upon binary codes. Writing programs directly in binary code is possible, yet an unnecessary, tedious. Instead, we can write such programs in a low-level symbolic language, called assembly, and have them translated into binary code by a program called assembler. In this project you will write some low-level assembly programs, and will be forever thankful for high-level languages like C and Java. (Actually, assembly programming can be a lot of fun, if you are in the right mood; it's an excellent brain teaser, and it allows you to control the underlying machine directly and completely.)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>To get a taste of low-level programming in machine language, and to experience the thrill of writing programs in that environment. In the process of working on this project, you will become familiar with the notion of mapping a high-level language to machine-language - and you will appreciate visually how machine hardware executes software - on a platform. These lessons will be learned in the context of writing a simple multiplication program, as shown below.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>Mult.asm</td>
</tr>
</tbody>
</table>

All the necessary project 4 files are available in: nand2tetris / projects / 04
Machine Language: lecture plan

- Machine languages
- Basic elements
- The Hack computer and machine language
- The Hack language specification
- Input / Output
- Hack programming
  - Part 1: registers and memory
  - Part 2: branching, variable, iteration
  - Part 3: pointers, input/output
- Project 4 overview
Chapter 4

Machine Language

These slides support chapter 4 of the book

*The Elements of Computing Systems*

By Noam Nisan and Shimon Schocken

MIT Press