Assembly Language
Symbolic notation for machine language programs
• Thinly overlays machine language
• Much easier for programmers
• Requires a translator (assembler)

Most assemblers provide facilities beyond just symbolic representation of machine language:
• Labels instead of addresses
• High-level constants (e.g., strings)
• Macro languages

Example

```assembly
@SCREEN // initialize current to
D=A // screen address
@current
M=D

(LFILL)
@current // set current register of
A=M // screen to all black
D=1
@current // increment screen location
M=M+1
@24576 // load end address
D=A
@current
D=D-M // if not equal, loop again
@LFILL
D;JNE

(END)
@END
0;JMP
```

Hack Machine Language: A-instructions

Symbolic: @value // Where value is either a non-negative decimal number or a symbol referring to such number.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Non-negative decimal number or symbol</td>
</tr>
</tbody>
</table>

```binary
0 v v v v v v v v v v v v v v v
```

Source: www.nand2tetris.org

Hack Machine Language: C-instructions

Symbolic: dest=jump;comp 0 Either the dest or jump fields may be empty.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dest</td>
<td>Destination address</td>
</tr>
<tr>
<td>comp</td>
<td>Comparison operation</td>
</tr>
<tr>
<td>jump</td>
<td>Jump target address</td>
</tr>
</tbody>
</table>

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

Source: www.nand2tetris.org

Labels

Not part of machine language, just assembly:

(NAME) in program fixes NAME as an alias for next instruction location in ROM.

E.g.,

```
D=0
(LOOP)
D=D+1
@LOOP
0;JMP
```

LOOP becomes an alias for ROM address 1, assuming this is the whole program.

Source: www.nand2tetris.org
Variables

Using any symbol in an A-instruction not declared somewhere using (NAME) defines a variable.

You are declaring a variable, which automatically gets a unique location in RAM (starting at address 16).

E.g.,

```asm
@i // loads address of i into A
M=0 // sets i to zero
(LOOP)
@i
M=M+1
@LOOP
0;JMP
```

becomes an alias for RAM address 16 if it is the first variable declared in program.

Assembler Construction

2-passes:

- First pass finds all labels (“L-instructions”) and identifies their corresponding ROM addresses
  - In first pass, keep a “program counter” of instructions
  - Program counter tells us where labels point to
  - Without this step, have to do various kludges to make everything fit (e.g. “NOP”, errata)

- Second pass translates A and C instructions into machine codes, resolves all symbols

Symbol Resolution

Assembler maintains a symbol table:

- Stores reserved symbols (R0-R15, SCREEN, etc.) with their fixed addresses (initialize)
- Stores labels with ROM addresses (first pass)

In second pass:

- Replace reserved symbols, labels with stored addresses
- Lookup/store variable addresses
  - If not previously seen, store new unique address
  - Replace variable with stored RAM address

Assembly Process: 0 - Initialization

Create symbol table & populate with predefined symbols.

Symbol Table Example

```
@SCREEN // initialize current to D=A // screen address
@current
M=0

(LFILL)
@current // set current register of
A=M // screen to all black
D=1
M=0
@current // increment screen location
M=M+1
@24576 // load end address
D=A
@current
D=D-M // if not equal, loop again
@END
0;JMP
```

Assembly Process: 1 – First Pass

- Walk through code, ignoring whitespace and comments
- Identify A and C instructions, keep count for now
- Identify label declarations (“L-instructions”)
  - Add to symbol table with current instruction count (address of next instruction)

Symbol Table

```
Symbol Table
R0 > 0
R1 > 1
...
R15 > 15
SCREEN > 16844
KBD > 24576
SP > 0
LCL > 1
ARG > 2
THIS > 3
THAT > 4
LFILL > 4
END > 16
current > 16
```
First Pass Example

@SCREEN // initialize current to
D=A
// screen address
@current
M=D

(LFILL)
@current
// set current register of
A=M
D=1
M=D
@current
// increment screen location
M=M+1
@24576
D=A
@current
D=D-M // if not equal, loop again
@LFILL
D;JNE
(END)
@END
0;JMP

Symbol Table

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RO</td>
<td>0</td>
</tr>
<tr>
<td>R1</td>
<td>1</td>
</tr>
<tr>
<td>R15</td>
<td>15</td>
</tr>
<tr>
<td>SCREEN</td>
<td>16384</td>
</tr>
<tr>
<td>KBD</td>
<td>24576</td>
</tr>
<tr>
<td>SP</td>
<td>0</td>
</tr>
<tr>
<td>LCL</td>
<td>1</td>
</tr>
<tr>
<td>ARG</td>
<td>2</td>
</tr>
<tr>
<td>THIS</td>
<td>3</td>
</tr>
<tr>
<td>THAT</td>
<td>4</td>
</tr>
<tr>
<td>LFILL</td>
<td>4</td>
</tr>
<tr>
<td>END</td>
<td>16</td>
</tr>
</tbody>
</table>

Assembly Process: 2 – Second Pass

- Walk through code, ignoring whitespace and comments, label declarations
- Translate A & C instructions (details to follow)

Hack Machine Language: A-instructions

Symbolic: $value // Where value is either a non-negative decimal number
// or a symbol referring to such number.

value (v = 0 or 1)

Binary: 0 v v v v v v v v v v v v v v

Source: www.nand2tetris.org

Assembly Process: 2 – Second Pass – A-instructions

- Initialize a counter to beginning of non-reserved RAM (e.g., mem = 16)
- Translate A-instructions:
  - If string after @ is a number, then val = number
  - Else
    - If string after @ is in symbol table, then lookup val in symbol table
    - If string after @ NOT in symbol table, then it is a new variable:
      - Store string -> mem in symbol table
      - Increment mem
      - val = mem
    - Output “0” + string representation of 15-bit binary representation of val

Hack Machine Language: C-instructions

Symbolic: dest=comp;jump
// Either the dest or jump fields may be empty.
// If dest is empty, the “=” is omitted.
// If jump is empty, the “;” is omitted.

Binary: 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0

Source: www.nand2tetris.org

Assembly Process: 2 – Second Pass – C-instructions

- Parse out dest, comp, and jump parts
- Translate dest, comp, and jump to corresponding bit fields
- Output “111” + a c1 c2 c3 c4 c5 c6 + d1 d2 d3 + j1 j2 j3

Source: www.nand2tetris.org
Hack Machine Language: C-instructions (comp)

<table>
<thead>
<tr>
<th>Hack code</th>
<th>Value in comp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101010</td>
<td>0</td>
</tr>
<tr>
<td>0111111</td>
<td>1</td>
</tr>
<tr>
<td>0111010</td>
<td>2</td>
</tr>
<tr>
<td>0000010</td>
<td>D</td>
</tr>
<tr>
<td>0110000</td>
<td>A</td>
</tr>
<tr>
<td>0110010</td>
<td>M</td>
</tr>
<tr>
<td>0001100</td>
<td>D</td>
</tr>
<tr>
<td>1110000</td>
<td>!M</td>
</tr>
<tr>
<td>1110001</td>
<td>M+1</td>
</tr>
<tr>
<td>1110011</td>
<td>M-</td>
</tr>
<tr>
<td>1110111</td>
<td>DIM</td>
</tr>
</tbody>
</table>

Easiest approach: lookup table

Hack Machine Language: C-instructions (jump)

<table>
<thead>
<tr>
<th>Cond.</th>
<th>C</th>
<th>J</th>
<th>Jmp</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>GT</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>EQ</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>LT</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>LE</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>JP</td>
</tr>
</tbody>
</table>

Easiest approach is again a lookup table.

Hack Machine Language: C-instructions (dest)

<table>
<thead>
<tr>
<th>Hack code</th>
<th>Memmem</th>
<th>Mem</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0 0 0 0</td>
<td>0 0 1</td>
<td>M</td>
<td>Mem[0]</td>
</tr>
<tr>
<td>00 1 0 0 0</td>
<td>0 1 0</td>
<td>D</td>
<td>Mem[0]</td>
</tr>
<tr>
<td>00 0 0 0 0</td>
<td>1 0 0</td>
<td>A</td>
<td>Mem[0]</td>
</tr>
<tr>
<td>00 0 0 0 0</td>
<td>1 0 1</td>
<td>AR</td>
<td>Mem[0]</td>
</tr>
<tr>
<td>11 0 0 0 0</td>
<td>1 0 1</td>
<td>AND</td>
<td>Mem[0]</td>
</tr>
</tbody>
</table>

Can use lookup table or translate one bit at a time.

Building Your Assembler

- Book provides a modular design
  - Easily translated into classes for OO language or modules for structured language
  - This existing design works: do you want to re-invent?
- Book suggests a 2-stage approach:
  - First translate programs with no symbols (.asm files without symbols provided for you to test on)
  - Then add in symbol support