Assembler

Usage and Copyright Notice:

Copyright 2005 © Noam Nisan and Shimon Schocken


We provide both PPT and PDF versions.

The book web site, www.idc.ac.il/tecs, features 13 such presentations, one for each book chapter. Each presentation is designed to support about 3 hours of classroom or self-study instruction.

You are welcome to use or edit this presentation as you see fit for instructional and non-commercial purposes.

If you use our materials, we will appreciate it if you will include in them a reference to the book’s web site.

If you have any questions or comments, you can reach us at tecs.ta@gmail.com
Where we are at:

**Hardware hierarchy**
- Machine Language
  - Abstract interface
  - Chapters 4-5
- Gate Logic
  - Abstract interface
  - Chapters 1-3
- Chips & Logic Gates
- Electrical Engineering
- Physics

**Compiler**
- Virtual Machine
- Abstract interface
- Chapters 7-8
- Assembler
- Chapter 6

**Software hierarchy**
- Abstract design
- Chapters 9, 12
  - Abstract interface
  - Chapters 10-11

**Computer Architecture**
- Abstract interface
- Chapters 4-5

**Gate Logic**
- Abstract interface
- Chapters 1-3

**Hardware Platform**
- Abstract interface
- Chapters 4-5

**Machine Language**
- Abstract interface
- Chapters 4-5

**Assembler**
- Chapter 6

**Virtual Machine**
- Abstract interface
- Chapters 7-8

**Virtual Machine Translator**
- Abstract interface
- Chapters 7-8

**Human Thought**
- Abstract design
- Chapters 9, 12
Why care about assemblers?

Because …

- Assemblers employ nifty programming tricks
- Assemblers are the first rung up the software hierarchy ladder
- An assembler is a translator of a simple language
- Writing an assembler = good practice for writing compilers.
Program translation

**Source code**

```plaintext
// Computes sum = 1 + ... + 100
00 i=1
01 sum=0
02 LOOP:
03 IF i=101 GOTO END
04 sum=sum+i
05 i=i+1
06 GOTO LOOP
07 END:
08 GOTO END
```

**Target code**

```
00 10001010110011001000111011001101
01 0001110110111001000111011001100
02 00100110110011001000111011001100
03 10001110110011001000111011001101
04 00011101100111001000111011001100
05 10001110110011001010111011001110
06 00011101110011001000111011001100
07 10001101100111001010111011001101
08 00011101100110010010111011001100
09 10001110110011001000111011001101
10 00111010110011001000111011001100
11 10001110110011001000111011001100
```

The program translation challenge

- Parse the source program, using the syntax rules of the source language
- Re-express the program’s semantics using the syntax rules of the target language

**Assembler = simple translator**

- Translates each assembly command into one or more machine instructions
- Handles symbols (i, sum, LOOP, END, ...)

---

Symbol resolution

In low level languages, symbols are used to code:
- Variable names
- Destinations of goto commands (labels)
- Special memory locations

The assembly process:
- First pass: construct a symbol table
- Second pass: translate the program, using the symbol table for symbols resolution.

<table>
<thead>
<tr>
<th>Code with Symbols</th>
<th>Symbol table</th>
<th>Code with Symbols Resolved</th>
</tr>
</thead>
<tbody>
<tr>
<td>// Computes sum=1+...+100</td>
<td>i 1024</td>
<td>00 M[1024]=1 // (M=memory)</td>
</tr>
<tr>
<td>00 i=1</td>
<td>sum 1025</td>
<td>01 M[1025]=0</td>
</tr>
<tr>
<td>01 sum=0</td>
<td>LOOP 2</td>
<td>02 if M[1024]=101 goto 6</td>
</tr>
<tr>
<td>02 IF i=101 GOTO END</td>
<td></td>
<td>04 M[1024]=M[1024]+1</td>
</tr>
<tr>
<td>03 sum=sum+i</td>
<td></td>
<td>05 goto 2</td>
</tr>
<tr>
<td>04 i=i+1</td>
<td></td>
<td>06 goto 6</td>
</tr>
<tr>
<td>05 GOTO LOOP END</td>
<td></td>
<td>(assuming that each symbolic command is translated into one word in memory)</td>
</tr>
<tr>
<td>06 GOTO END</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
This example is based on some simplifying assumptions:

- Largest possible program is 1024 commands long
- Each command fits into one memory location
- Each variable fits into one memory location

Every one of these assumptions can be relaxed easily.
The Hack assembly language

Assembly program (Prog.asm)

// Adds 1 + ... + 100
@i
M=1  // i=1
@sum
M=0  // sum=0

(LOOP)
@i
D=M  // D=i
@100
D=D-A  // D=i-100
@END
D;JGT // if (i-100)>0 goto END
@i
D=M  // D=i
@sum
M=D+M  // sum=sum+i
@i
M=M+1  // i=i+1
@LOOP
0;JMP // goto LOOP

(END)
@END
0;JMP // infinite loop

Assembly program =
a stream of text lines, each being one of the following:

- Instruction:
  - A-instruction or
  - C-instruction

- Symbol declaration:
  - (symbol)

- Comment or white space:
  // comment.
Handling A-instructions

Symbolic: \( \text{@value} \)  // Where value is either a non-negative decimal number
// or a symbol referring to such number.

\[
\text{value} \ (v = 0 \text{ or } 1)
\]

Binary:

\[
\begin{array}{cccccccc}
0 & v & v & v & v & v & v & v & v & v
\end{array}
\]

Translation to binary:
- If \( \text{value} \) is a number: \text{simple}
- If \( \text{value} \) is a symbol: \text{later}. 
## Handling C-instruction

### 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:

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

### Translation to Binary:

**Simple!**

<table>
<thead>
<tr>
<th>comp</th>
<th>Mnemonic</th>
<th>Destination (where to store the computed value)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>null</td>
<td>The value is not stored anywhere</td>
</tr>
<tr>
<td>0 1 0</td>
<td>d</td>
<td>D register</td>
</tr>
<tr>
<td>0 1 0</td>
<td>M</td>
<td>Memory[A] (memory register addressed by A)</td>
</tr>
<tr>
<td>1 1 1</td>
<td>AMD</td>
<td>A register and D register</td>
</tr>
<tr>
<td>1 1 1</td>
<td>AMD</td>
<td>A register, Memory[A], and D register</td>
</tr>
</tbody>
</table>

### Table:

<table>
<thead>
<tr>
<th>j1 (out &lt; 0)</th>
<th>j2 (out = 0)</th>
<th>j3 (out &gt; 0)</th>
<th>Mnemonic</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>null</td>
<td>No jump</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>JGT</td>
<td>If out &gt; 0 jump</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>JEQ</td>
<td>If out = 0 jump</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>JGE</td>
<td>If out &gt;= 0 jump</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>JLT</td>
<td>If out &lt; 0 jump</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>JNE</td>
<td>If out != 0 jump</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>JLE</td>
<td>If out &lt;= 0 jump</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>JMP</td>
<td>Jump</td>
</tr>
</tbody>
</table>
The overall assembly logic

For each (real) command

- Parse the command, i.e. break it into its constituent fields
- Replace each symbolic reference (if any) with the corresponding memory address (a binary number)
- For each field, generate the corresponding binary code
- Assemble the binary codes into a complete machine instruction.

Assembly program (Prog.asm)

// Adds 1 + ... + 100
@i
M=1   // i=1
@sum
M=0   // sum=0
(LOOP)
  @i
  D=M   // D=i
  @100
  D=D-A  // D=i-100
  @END
  D;JGT   // if (i-100)>0 goto END
  @i
  D=M   // D=i
  @sum
  M=D+M  // sum=sum+i
  @i
  M=M+1  // i=i+1
  @LOOP
  0;JMP  // goto LOOP
(END)
  @END
  0;JMP  // infinite loop
Symbols handling (in the Hack language)

**Program example**

```plaintext
// Adds 1 + ... + 100
@i
M=1  // i=1
@sum
M=0  // sum=0

(LOOP)
    @i
    D=M  // D=i
    @100
    D=D-A  // D=i-100
    @END
    D;JGT  // if (i-100)>0 goto END
    @i
    D=M  // D=i
    @sum
    M=M+M  // sum=sum+i
    @i
    M=M+1  // i=i+1
    @LOOP
    0;JMP  // goto LOOP

(END)
    @END
    0;JMP  // infinite loop
```

- **Predefined symbols:** (don’t appear in this example)
  
<table>
<thead>
<tr>
<th>Label</th>
<th>RAM address</th>
</tr>
</thead>
<tbody>
<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>R0-R15</td>
<td>0-15</td>
</tr>
<tr>
<td>SCREEN</td>
<td>16384</td>
</tr>
<tr>
<td>KBD</td>
<td>24576</td>
</tr>
</tbody>
</table>

- **Label symbols:** The pseudo-command `(label)` declares that the user-defined symbol `label` should refer to the memory location holding the next command in the program.

- **Variable symbols:** If `label` appears in a `@label` command, and `label` is neither predefined nor defined elsewhere in the program using the `(label)` pseudo command, then `label` is treated as a variable.

Design decision: variables are mapped to consecutive memory locations starting at RAM address 16.
Example

**Assembly code (Prog.asm)**

```assembly
// Adds 1 + ... + 100

@i
M=1 // i=1
@sum
M=0 // sum=0

(LOOP)

@i
D=M // D=i
@100
D=D-A // D=i-100
@END
D;JGT // if (i-100)>0 goto END

@i
D=M // D=i
@sum
M=D+M // sum=sum+i
@i
M=M+1 // i=i+1
@LOOP
0;JMP // goto LOOP

(END)

@END
0;JMP // infinite loop
```

**Binary code (Prog.hack)**

```
0000 0000 0001 0000 1110 1111 1100 1000
0000 0000 0001 0001 1110 1010 1000 1000

0000 0000 0001 0000 1111 1100 0001 0000
1111 1100 0001 0000
0000 0000 0110 0100 1110 0100 1101 0000
0000 0000 0001 0010
1110 0011 0000 0001
0000 0000 0001 0000
1111 1100 0001 0000
0000 0000 0001 0001 0000
1111 1100 0001 0001
0000 0000 0001 1000 1000
1111 1101 1100 1000
0000 0000 0001 0010
1110 1010 1000 0111
```

Proposed implementation

An assembler program can be implemented as follows.

Software modules:

- **Parser**: Unpacks each command into its underlying fields
- **Code**: Translates each field into its corresponding binary value
- **SymbolTable**: Manages the symbol table
- **Main**: Initializes I/O files and drives the show.

Proposed implementation stages

**Stage I**: Build a basic assembler for programs with no symbols

**Stage II**: Extend the basic assembler with symbol handling capabilities.
### Parser module

**Parser**: Encapsulates access to the input code. Reads an assembly language command, parses it, and provides convenient access to the command's components (fields and symbols). In addition, removes all white space and comments.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Arguments</th>
<th>Returns</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor / initializer</td>
<td>Input file / stream</td>
<td>--</td>
<td>Opens the input file/stream and gets ready to parse it.</td>
</tr>
<tr>
<td>hasMoreCommands</td>
<td>--</td>
<td>Boolean</td>
<td>Are there more commands in the input?</td>
</tr>
<tr>
<td>advance</td>
<td>--</td>
<td>--</td>
<td>Reads the next command from the input and makes it the current command. Should be called only if hasMoreCommands() is true. Initially there is no current command.</td>
</tr>
<tr>
<td>commandType</td>
<td>--</td>
<td>A_COMMAND, C_COMMAND, L_COMMAND</td>
<td>Returns the type of the current command:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>* A_COMMAND for @XXX where XXX is either a symbol or a decimal number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>* C_COMMAND for dest=comp;jump</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>* L_COMMAND (actually, pseudo-command) for (XXX) where XXX is a symbol.</td>
</tr>
</tbody>
</table>
### Parser module (cont.)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>symbol</td>
<td>string</td>
<td>Returns the symbol or decimal ( \text{xxx} ) of the current command ( @\text{xxx} ) or ( (\text{xxx}) ). Should be called only when <code>commandType()</code> is <code>A_COMMAND</code> or <code>L_COMMAND</code>.</td>
</tr>
<tr>
<td>dest</td>
<td>string</td>
<td>Returns the dest mnemonic in the current <code>C-command</code> (3 possibilities). Should be called only when <code>commandType()</code> is <code>C_COMMAND</code>.</td>
</tr>
<tr>
<td>comp</td>
<td>string</td>
<td>Returns the comp mnemonic in the current <code>C-command</code> (28 possibilities). Should be called only when <code>commandType()</code> is <code>C_COMMAND</code>.</td>
</tr>
<tr>
<td>jump</td>
<td>string</td>
<td>Returns the jump mnemonic in the current <code>C-command</code> (3 possibilities). Should be called only when <code>commandType()</code> is <code>C_COMMAND</code>.</td>
</tr>
</tbody>
</table>
## Code module

<table>
<thead>
<tr>
<th>Routine</th>
<th>Arguments</th>
<th>Returns</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dest</code></td>
<td>mnemonic (string)</td>
<td>3 bits</td>
<td>Returns the binary code of the <code>dest</code> mnemonic.</td>
</tr>
<tr>
<td><code>comp</code></td>
<td>mnemonic (string)</td>
<td>7 bits</td>
<td>Returns the binary code of the <code>comp</code> mnemonic.</td>
</tr>
<tr>
<td><code>jump</code></td>
<td>mnemonic (string)</td>
<td>3 bits</td>
<td>Returns the binary code of the <code>jump</code> mnemonic.</td>
</tr>
</tbody>
</table>
Building the final assembler

- **Initialization**: create the symbol table and initialize it with the pre-defined symbols.

- **First pass**: march through the program without generating any code. For each label declaration of the form "(label)", add the pair <label, n> to the symbol table.

- **Second pass**: march again through the program, and translate each line:
  - If the line is a C-instruction, simple.
  - If the line is "@label" where label is a number, simple.
  - If the line is "@label" and label is a symbol, look it up in the symbol table and proceed as follows:
    - If the symbol is found, replace it with its numeric meaning and complete the command’s translation.
    - If the symbol is not found, then it must represent a new variable: add the pair <label, n> to the symbol table, where n is the next available RAM address, and complete the command’s translation.

(The allocated RAM addresses are running, starting at address 16).
## Symbol table

**SymbolTable**: A symbol table that keeps a correspondence between symbolic labels and numeric addresses.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Arguments</th>
<th>Returns</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor</td>
<td>--</td>
<td>--</td>
<td>Creates a new empty symbol table.</td>
</tr>
<tr>
<td>addEntry</td>
<td>symbol (string),</td>
<td>--</td>
<td>Adds the pair (symbol, address) to the table.</td>
</tr>
<tr>
<td></td>
<td>address (int)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>contains</td>
<td>symbol (string)</td>
<td>Boolean</td>
<td>Does the symbol table contain the given symbol?</td>
</tr>
<tr>
<td>GetAddress</td>
<td>symbol (string)</td>
<td>int</td>
<td>Returns the address associated with the symbol.</td>
</tr>
</tbody>
</table>
Perspective

- Simple machine language, simple assembler
- Most assemblers are not stand-alone, but rather encapsulated in a translator of a higher order
- Low-level C programming (e.g. for real-time systems) may involve some assembly programming (e.g. for optimization)
- Macro assemblers:

```assembly
// Computes sum=1+...+100
00 i=1
01 sum=0
LOOP:
02 IF i=101 GOTO END
03 sum=sum+i
04 i=i+1
05 GOTO LOOP
END:
06 GOTO END
```