LC-3
Assembly Language

(Textbook Chapter 7)
Assembly and assembler

• Machine language - binary

• Assembly language - symbolic

• Assembler is a program that turns symbols into machine instructions.
  – ISA-specific: close correspondence between symbols and instruction set
    • mnemonics for opcodes
    • labels for memory locations
Syntax of LC-3 assembly: Language elements

- Instructions (we have seen most of them)
- Comments
- Labels
- Declarations
- Assembler directives and trap codes
  Whitespaces (between symbols) and case are ignored.
Instructions

- One instruction or declaration per line

```
LABEL OPCODE OPERANDS ; COMMENTS
```

- `LABEL` optional
- `OPCODE` mandatory
- `OPERANDS` optional
- `; COMMENTS` mandatory
Opcodes and Operands

• Opcodes
  - reserved symbols that correspond to LC-3 instructions
  - listed in Appendix A (ex: ADD, AND, …)

• Operands
  - Registers: $R_n$, where $n$ is the register number
  - Immediate numbers: (decimal), (hex), or (binary)
  - Labels: symbolic names of memory locations
  - Operands are separated by spaces, tabs, or commas
  - Their number, order, and type correspond to the instruction format
Data types

LC-3 has 2 basic data types
  • Integer
  • Character

Both are 16 bits wide (a word), though a character is only 8 bits in size.
Comments

- Anything on a line after a semicolon is a comment
- Comments are ignored by assembler
- Used by humans to document/understand programs
- Tips for useful comments:
  • avoid restating the obvious, as “decrement R1”
  • provide additional insight, as in “accumulate product in R6”
  • use comments to separate pieces of program
Labels

• Placed at beginning of line
• Assign a symbolic name to their line (its address)
• Symbolic names used to identify memory locations. Two kinds:
  – Location of target of a branch or jump
  – Location of a variable for loading and storing
• Can be 1-20 characters in size
Assembler directives

- **Directives or pseudo-ops** give information to the assembler.
- Not executed by the program
- All directives start with a period ‘.’

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ORIG</td>
<td>Where to start in placing things in memory</td>
</tr>
<tr>
<td>.FILL</td>
<td>Declare a memory location (variable)</td>
</tr>
<tr>
<td>.BLKW</td>
<td>Declare a group of memory locations (array)</td>
</tr>
<tr>
<td>.STRINGZ</td>
<td>Declare a group of characters in memory (string)</td>
</tr>
<tr>
<td>.END</td>
<td>Tells assembly where your program source ends</td>
</tr>
</tbody>
</table>
.ORIG

• Tells simulator where to put your code in memory (starting location)
• Only one .ORIG allowed per program module
• PC is set to this address at start up
• Similar to the main() function in C

• Example: the standard convention is

  .orig  x3000
• Declaration and initialization of variables
• One declaration per line
• Always declaring words
• Examples:

flag .FILL x0001
counter .FILL x0002
letter .FILL x0041
letters .FILL x4241
In C

\texttt{type \ varname;}

Where type is
\texttt{int} (integer)
\texttt{char} (character)
\texttt{float} (floating-point) \textbf{In LC-3}
\texttt{varname \ .FILL value}

- value is required (initialize)
- type is only 16-bit integer
.BLKW

- Reserves (and initializes) a sequence of contiguous memory locations (arrays)
- Examples:

;set aside 3 locations
.BLKW 3

;set aside 1 location and label it Bob
.BLKW 1

; set aside 7 locations,
; label them, and init them all to 4
Num .BLKW 7 #4
.STRINGZ

- Declare a string of characters
- Automatically terminated with x0000
- Example:

  hello .STRINGZ "Hello World!"
. END

• Tells the assembler where your program ends
• Only one .END allowed in your program module
• That’s where the assembler stops assembling, NOT where the execution stops!
TRAP
(System Calls)

Very tedious and dangerous for a programmer to deal with I/O.

This is why we like to have an OS.

Need an instruction to get its attention.

Use the TRAP instruction and a trap vector.
## Trap Service Routines

The LC-3 assembler provides “pseudo-instructions” for each trap code, so you don’t have to remember them.

<table>
<thead>
<tr>
<th>Trap Vector</th>
<th>Assembler Name</th>
<th>Usage &amp; Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>x20</td>
<td>GETC</td>
<td>Read a character from console into R0, not echoed.</td>
</tr>
<tr>
<td>x21</td>
<td>OUT</td>
<td>Write the character in R0[7:0] to console.</td>
</tr>
<tr>
<td>x22</td>
<td>PUTS</td>
<td>Write string of characters to console. Start with character at address contained in R0. Stops when 0x0000 is encountered.</td>
</tr>
<tr>
<td>x23</td>
<td>IN</td>
<td>Print a prompt to console and read in a single character into R0. Character is echoed.</td>
</tr>
<tr>
<td>x24</td>
<td>PUTSP</td>
<td>Write a string of characters to console, 2 characters per address location. Start with characters at address in R0. First [7:0] and then [15:0]. Stops when 0x0000 is encountered.</td>
</tr>
<tr>
<td>x25</td>
<td>HALT</td>
<td>Halt execution and print message to console.</td>
</tr>
</tbody>
</table>
To print a character

; the char must be in R0[7:0]
TRAP x21
  or
OUT

To read in a character

; will go into R0[7:0],
; no echo.
TRAP x20
  or
GETC

To end the program

TRAP x25
  or
HALT
Simple LC-3 program

.ORG x3000
LD R2, Zero
LD R0, M0
LD R1, M1
Loop BRz Done
ADD R2, R2, R0
ADD R1, R1, -1
BR Loop
Done ST R2, Res
HALT
Res .FILL x0000
Zero .FILL x0000
M0 .FILL x0007
M1 .FILL x0003
.END

What does this program do?

What is in Res at the end?
The assembly process

• Convert assembly language file (.asm) into an executable file (.obj) for the LC-3 simulator.

  - First Pass:
    - scan program file
    - find all labels and calculate the corresponding addresses - the *symbol table*

  - Second Pass:
    - convert instructions to machine language, using information from symbol table
First Pass: The Symbol Table

1. Find the .ORIG statement, which tells us the address of the first instruction.
   • Initialize Location Counter (LC), which keeps track of the current instruction.

2. For each non-empty line in the program:
   a) If line contains a label, add label and LC to symbol table.
   b) Increment LC.
      - NOTE: If statement is .BLKW or .STRINGZ, increment LC by the number of words allocated.

3. Stop when .END statement is reached.

NOTE: A line with only a comment is considered an empty line.
## Practice: Symbol Table

Build the symbol table for the multiply program:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ORIG</td>
<td>x3000</td>
</tr>
<tr>
<td>x3000</td>
<td>LD</td>
</tr>
<tr>
<td>x3001</td>
<td>LD</td>
</tr>
<tr>
<td>x3002</td>
<td>LD</td>
</tr>
<tr>
<td>; begin multiply</td>
<td></td>
</tr>
<tr>
<td>x3003</td>
<td>Loop</td>
</tr>
<tr>
<td>x3004</td>
<td>ADD</td>
</tr>
<tr>
<td>x3005</td>
<td>ADD</td>
</tr>
<tr>
<td>x3006</td>
<td>BR</td>
</tr>
<tr>
<td>; end multiply</td>
<td></td>
</tr>
<tr>
<td>x3007</td>
<td>Done</td>
</tr>
<tr>
<td>x3008</td>
<td>HALT</td>
</tr>
<tr>
<td>x3009</td>
<td>Result</td>
</tr>
<tr>
<td>x300A</td>
<td>Zero</td>
</tr>
<tr>
<td>x300B</td>
<td>M0</td>
</tr>
<tr>
<td>x300C</td>
<td>M1</td>
</tr>
</tbody>
</table>
2nd Pass: Generating Machine Language

- For each executable assembly language statement, generate the corresponding machine language instruction.
  - If operand is a label, look up the address from the symbol table.

- Potential problems:
  - Improper number or type of arguments
    - ex: NOT R1,#7
    - ADD R1,R2
  - Immediate argument too large
    - ex: ADD R1,R2,#1023
  - Address (associated with label) more than 256 from instruction
    - can't use PC-relative addressing mode
The LC-3 Assembler

• Using "assemble" (Unix) or LC3Edit (Windows), generates several different output files.

Diagram:
- Assembly Language Program (.asm) → Assembler
- Bin Listing (.bin)
- Hex Listing (.hex)
- Symbol Table (.sym)
- Listing File (.lst)
- Object File (.obj)
Multiple Object Files

• An object file is not necessarily a complete program.
  - system-provided library routines
  - code blocks written by multiple developers

• For LC-3 simulator, can load multiple object files into memory, then start executing at a desired address.
  - system routines, such as keyboard input, are loaded automatically
    • loaded into “system memory,” below x3000
    • user code should be loaded between x3000 and xFDFF
  - each object file includes a starting address
  - be careful not to load overlapping object files
Linking

Linking is the process of resolving symbols between independent object files.

- Suppose we define a symbol in one module, and want to use it in another.
- The directive `.EXTERNAL` is used to tell the assembler that a symbol is defined in another module.
- The linker will search symbol tables of other modules to resolve symbols and complete code generation before loading.
Loading

• *Loading* is the process of copying an executable image into memory.
  - more sophisticated loaders are able to *relocate* images to fit into available memory
  - must readjust branch targets, load/store addresses
Running

- The loader makes the CPU jump to the first instruction → .ORIG.
- The program executes
- When execution completes, control returns to the OS or to the simulator
- Load again to run again with different data (in LC3 we must assemble again, since data is in program)
Recommended exercises:

- Ex 7.1 to 7.5, 7.7 to 7.9
- *Especially recommended*: 7.13 to 7.15, and 7.18 to 7.24 (yes, all of them except 7.16 and 7.17)