Assembly and Assembler

- Machine language - binary
- Assembly language - symbolic
- An assembler is a program that turns symbols into machine instruction
  - ISA-specific: close correspondence between symbols and instruction set
    - mnemonics for opcodes
    - labels for memory locations
  - ADD R6, R2, R6 ; increment index reg.
Elements of Assembly Language

- Instructions (we have seen most of them)
- Comments
- Labels
- Declarations
- Assembler directives and trap codes
- Ignored
  - White space (between symbols)

Assembly Instructions

- One instruction or declaration per line

```
LABEL OPCODE OPERANDS ; COMMENTS
```

`optional`  `mandatory`
Opcodes and Operands

- **Opcodes**
  - Reserved symbols that correspond to LC-3 instructions
  - Listed in Appendix A (ex: ADD, AND, ...)

- **Operands**
  - Can be registers: \( R_n \), where \( n \) is the register number
  - Can be immediate numbers
    - Prefix: \# (decimal), \( x \) (hex), or \( b \) (binary)
  - Can be labels
    - Symbolic names of memory locations
  - Separated by spaces, tabs, or commas
  - Number, order, and type correspond to the instruction format

Data Types

- LC-3 has two basic data types
  - Integer
  - Character

- Both are 16 bits wide (a word)
- Though a character is only 8 bits in size
- How does that work??
Comments In Code

- What is a comment?
  - Anything on a line after a semicolon is a comment
  - Comments are ignored by the assembler
  - Used by humans to document and understand programs

- Some tips for useful comments
  - Avoid restating the obvious
    * Bad: Decrement R1
  - Provide additional insight
    * Good: Accumulate the product
  - 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 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>

Assembler Directives: .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
- Example
  - .orig x3000
  - Typical address for LC-3 is x3000
Assembler Directives: **.FILL**

- 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`

Assembler Directives: **.FILL**

- **In C**
  - `type varname;`
  - Where `type` is one of these
    - `int` (integer)
    - `char` (character)
    - `float` (floating-point)
- **In LC-3**
  - `varname .FILL value`
  - Where...
    - `value` is required
    - `type` is 16-bit integer
Assembler Directives: .BLKW

- `set aside 3 unnamed spaces`:
  ```assembly```
  .BLKW 3
  ```assembly```

- `set aside 1 named word`:
  ```assembly```
  Bob .BLKW 1
  ```assembly```

- `set aside 7 labeled words, initialize them all to 4`:
  ```assembly```
  Num .BLKW 7 #4
  ```assembly```

Assembler Directives: .STRINGZ

- Declare a string of characters
- Stored contiguously in memory
- Automatically terminated with x0000
  - “Null-terminated”
- Example:
  ```assembly```
  hello .STRINGZ
  “Hello World!”
Assembler Directives: .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!

System Calls: TRAP

- A trap is an exception that interrupts normal processing to perform a system-level task
- Certain traps are pre-defined in a trap vector
- To call a trap
  - Use the TRAP instruction
  - Specifying the trap vector
- Very tedious and dangerous for a programmer to deal with I/O
Trap Service Routines

<table>
<thead>
<tr>
<th>Trap Vector</th>
<th>Pseudo-Instruction</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>PUTSF</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>

Trap Examples

To print a character
; the char must be in R0
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

```assembly
.orig 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
- The executable file is the pure binary machine code
- LC-3 uses a two-pass assembler
  - Status messages are shown when assembling
  - E.g.,
    - Starting Pass 1...
    - Pass 1 - 0 error(s)
    - Starting Pass 2...
    - Pass 2 - 0 error(s)
The Assembly Process

- The assembly process is...
  - Assembling
    - Pass 1
    - Pass 2
  - Linking
  - Loading
  - Running

The Assembly Process: Assembling

- First Pass
  - Scan program file
  - Find all labels and calculate the corresponding addresses
    - Generate the symbol table

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

- Find the `.orig` statement
  - Tells the address of the first instruction
  - Initialize the location counter (LC), which keeps track of the current instruction
- For each non-empty line in the program
  - If a line contains a label, add label plus LC to symbol table
  - Increment LC
    - For a `.BLKW` or `.STRINGZ`, increment LC by the amount of space allocated
  - Stop when `.END` statement is reached
  - A line with only a comment is considered an empty line

Example: Generating a Symbol Table

<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 R2, Zero</td>
</tr>
<tr>
<td>x3001</td>
<td>LD R0, M0</td>
</tr>
<tr>
<td>x3002</td>
<td>LD R1, M1</td>
</tr>
<tr>
<td></td>
<td>; begin multiply</td>
</tr>
<tr>
<td>x3003</td>
<td>Loop BRz Done</td>
</tr>
<tr>
<td>x3004</td>
<td>ADD R2, R2, R0</td>
</tr>
<tr>
<td>x3005</td>
<td>ADD R1, R1, # -1</td>
</tr>
<tr>
<td>x3006</td>
<td>BR Loop</td>
</tr>
<tr>
<td></td>
<td>; end multiply</td>
</tr>
<tr>
<td>x3007</td>
<td>Done ST R2, Result</td>
</tr>
<tr>
<td>x3008</td>
<td>HALT</td>
</tr>
<tr>
<td>x3009</td>
<td>Result .FILL x0000</td>
</tr>
<tr>
<td>x300A</td>
<td>Zero .FILL x0000</td>
</tr>
<tr>
<td>x300B</td>
<td>M0 .FILL x0007</td>
</tr>
<tr>
<td>x300C</td>
<td>M1 .FILL x0003</td>
</tr>
<tr>
<td></td>
<td>.END</td>
</tr>
</tbody>
</table>
Second Pass: Generating Machine Language

- For each executable assembly language statement, generate the corresponding machine language instruction
  - If the operand is a label, look up the address from the symbol table
- Potential problems
  - Improper number of type of arguments
    - E.g.: `NOT R1, #7`
    - `ADD R1, R2`
  - Immediate argument too large
    - E.g.: `ADD R1, R2, #1023`
  - Address (associated with label) more than 256 from instruction
    - Then, can’t use PC-relative addressing mode

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, you 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
  - It is possible to load overlapping object files
The Assembly Process: 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

The Assembly Process: 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 re-adjust branch targets and load/store addresses
And Finally: Running

- The loader makes the CPU jump to the first instruction
  - Specified by .ORIG
- The program executes
- When execution completes, control returns to the OS or simulator

The LC-3 Assembler

- The LC-3 assembler generates several different output files
Recommended exercises

- Ex 7.1 to 7.11
- Especially recommended: 7.12 to 7.16, and 7.18 to 7.25 (yes, all of them except 7.17)