Chapter 7
Assembly Language
Human-Readable Representation of Machine Language

Computers like ones and zeros…

0001110010000110

Humans like symbols…

ADD R6,R2,R6 ; increment index reg.

Assembler is a program that turns symbolic representation of machine instructions (assembly program) into actual binary representation of machine instructions.

• ISA-specific
• Mnemonics for opcodes
• Labels for memory locations
• Additional operations for allocating storage and initializing data
An Assembly Language Program

; Program to multiply a number by the constant 6

.ORIG x3050
LD R1, SIX
LD R2, NUMBER
AND R3, R3, #0 ; Clear R3. It will contain the product.

; The inner loop

AGAIN ADD R3, R3, R2
ADD R1, R1, #-1 ; R1 keeps track of the iteration.
BRp AGAIN ;

HALT

; NUMBER .BLKW 1
SIX .FILL x0006

.END
LC-3 Assembly Language Syntax

Each line of a program is one of the following:

- an instruction
- an assembler directive (or pseudo-op)
- a comment

Whitespace (between symbols) and case are ignored. Comments (beginning with “;”) are also ignored.

An instruction has the following format:

```
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, LD, LDR, …

Operands

• registers -- specified by Rn, where n is the register number
• numbers -- indicated by # (decimal) or x (hex)
• label -- symbolic name of memory location
• operands separated by comma
• number, order, and type of operands specific to each instruction type
  ➢ ex:
    ADD R1,R1,R3
    ADD R1,R1,#3
    LD R6,NUMBER
    BRz LOOP
Labels and Comments

Label
- placed at the beginning of the line
- assigns a symbolic name to the address corresponding to line
  ex:
  ```
  LOOP ADD R1,R1,#-1
  BRp LOOP
  ```

Comment
- anything after a semicolon is a comment
- 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
Assembler Directives

Pseudo-operations

- do not refer to operations executed by program
- used by assembler
- look like instruction, but “opcode” starts with dot

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operand</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ORIG</td>
<td>address</td>
<td>starting address of program</td>
</tr>
<tr>
<td>.END</td>
<td></td>
<td>end of program</td>
</tr>
<tr>
<td>.BLKW</td>
<td>n</td>
<td>allocate n words of storage</td>
</tr>
<tr>
<td>.FILL</td>
<td>#n or xn</td>
<td>allocate one word, initialize with value n in decimal (#) or in hex (x)</td>
</tr>
<tr>
<td>.STRINGZ</td>
<td>n-character string</td>
<td>allocate n+1 locations, initialize w/characters and null terminator</td>
</tr>
</tbody>
</table>
## Trap Codes

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

<table>
<thead>
<tr>
<th>Code</th>
<th>Equivalent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HALT</td>
<td>TRAP x25</td>
<td>Halt execution and print message to console.</td>
</tr>
<tr>
<td>IN</td>
<td>TRAP x23</td>
<td>Print prompt on console, read (and echo) one character from keybd. Character stored in R0[7:0].</td>
</tr>
<tr>
<td>OUT</td>
<td>TRAP x21</td>
<td>Write one character (in R0[7:0]) to console.</td>
</tr>
<tr>
<td>GETC</td>
<td>TRAP x20</td>
<td>Read one character from keyboard. Character stored in R0[7:0].</td>
</tr>
<tr>
<td>PUTS</td>
<td>TRAP x22</td>
<td>Write null-terminated string to console. Address of string is in R0.</td>
</tr>
</tbody>
</table>
Style Guidelines

Use the following style guidelines to improve the readability and understandability of your programs:

1. Provide a program header, with author’s name, date, etc., and purpose of program.

2. Start labels, opcode, operands, and comments in same column for each line. (Unless entire line is a comment.)

3. Use comments to explain what each register does.

4. Give explanatory comment for most instructions.

5. Use meaningful symbolic names.
   - Mixed upper and lower case for readability.
   - ASCIItoBinary, InputRoutine, SaveR1

6. Provide comments between program sections.

7. Each line must fit on the page -- no wraparound or truncations.
   - Long statements split in aesthetically pleasing manner.
Sample Program

Count the occurrences of a character in a file.

Remember this?

- **Count = 0** (R2 = 0)
- **Ptr = 1st file character** (R3 = M[x3012])
- **Input char from keybd** (TRAP x23)
- **Load char from file** (R1 = M[R3])
- **Done?** (R1 ?= EOT)
- **Match?** (R1 ?= R0)
- **Incr Count** (R2 = R2 + 1)
- **Load next char from file** (R3 = R3 + 1, R1 = M[R3])
- **Convert count to ASCII character** (R0 = x30, R0 = R2 + R0)
- **Print count** (TRAP x21)
- **HALT** (TRAP x25)
Char Count in Assembly Language (1 of 3)

; Program to count occurrences of a character in a file.
; Character to be input from the keyboard.
; Result to be displayed on the monitor.
; Program only works if no more than 9 occurrences are found.

; Initialization

.ORIG x3000
AND R2, R2, #0 ; R2 is counter, initially 0
LD R3, PTR ; R3 is pointer to characters
GETC ; R0 gets character input
LDR R1, R3, #0 ; R1 gets first character

; Test character for end of file

TEST ADD R4, R1, #-4 ; Test for EOT (ASCII x04)
BRz OUTPUT ; If done, prepare the output
Char Count in Assembly Language (2 of 3)

; Test character for match. If a match, increment count.
;
    NOT   R1, R1
    ADD   R1, R1, R0 ; If match, R1 = xFFFFFF
    NOT   R1, R1    ; If match, R1 = x0000
    BRnp  getchar    ; If no match, do not increment
    ADD   R2, R2, #1

; Get next character from file.
;
    getchar ADD   R3, R3, #1 ; Point to next character.
    LDR    R1, R3, #0 ; R1 gets next char to test
    BRnzp get char

; Output the count.
;
    output LD     R0, ASCII ; Load the ASCII template
    ADD     R0, R0, R2 ; Convert binary count to ASCII
    OUT     ; ASCII code in R0 is displayed.
    HALT ; Halt machine
Char Count in Assembly Language (3 of 3)

;  Storage for pointer and ASCII template

ASCII  .FILL  x0030
PTR    .FILL  x4000
.END
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; this is called the *symbol table*

**Second Pass:**
- convert instructions to machine language, using information from symbol table
First Pass: Constructing 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 that contains only a comment is considered an empty line.
Practice

Construct the symbol table for the program in Figure 7.1 (Slides 7-11 through 7-13).

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Second 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 operands
  - ex: NOT R1,#7
  - ADD R1,R2
  - ADD R3,R3,NUMBER

- Immediate argument too large
  - ex: ADD R1,R2,#1023

- Address (associated with label) more than 256 instructions away
  - can’t use PC-relative addressing mode
## Practice

Using the symbol table constructed earlier, translate these statements into LC-3 machine language.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Machine Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R3, PTR</td>
<td></td>
</tr>
<tr>
<td>ADD R4, R1, #4</td>
<td></td>
</tr>
<tr>
<td>LDR R1, R3, #0</td>
<td></td>
</tr>
<tr>
<td>BRnp GETCHAR</td>
<td></td>
</tr>
</tbody>
</table>
LC-3 Assembler

Using “assemble” (Unix) or LC3Edit (Windows), generates several different output files.

- Assembly Language Program (.asm)
- Binary Listing (.bin)
- Hex Listing (.hex)
- Symbol Table (.sym)
- Listing File (.lst)
- Object File (.obj)

This one gets loaded into the simulator.
Object File Format

LC-3 object file contains

- Starting address (location where program must be loaded), followed by...
- Machine instructions

Example

- Beginning of “count character” object file looks like this:

```
0011000000000000  ORIG x3000
0101010010100000  AND R2, R2, #0
0010011000010001  LD R3, PTR
1111000000100011  TRAP x23
.                  .
.                  .
```
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 and 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

**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
- some notation, such as `.EXTERNAL`, is used to tell assembler that a symbol is defined in another module
- linker will search symbol tables of other modules to resolve symbols and complete code generation before loading
• Assembly language
  • A human-readable representation of machine language
    • ISA specific
    • Mnemonic opcodes
    • Symbolic names for registers and memory locations (i.e., labels)
    • Additional features for allocating storage and initializing data
    • Comments
  • Assembly language program is “translated” into machine language program (object code) by two-pass assembler whereas instructions in machine language program are “interpreted” by hardware