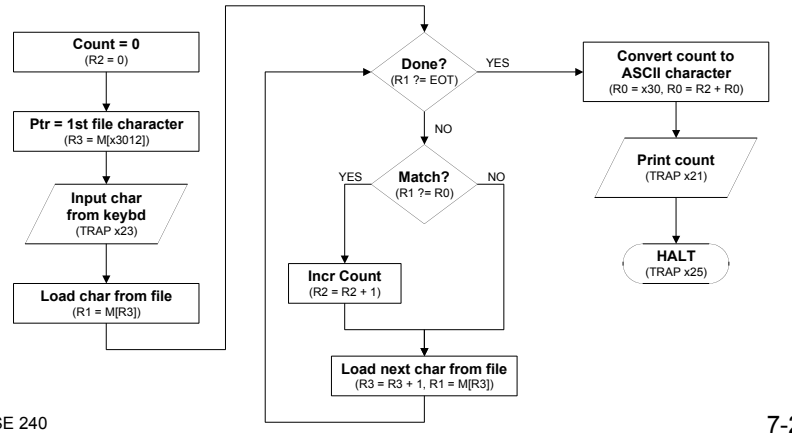
Chapter 7
Assembly Language

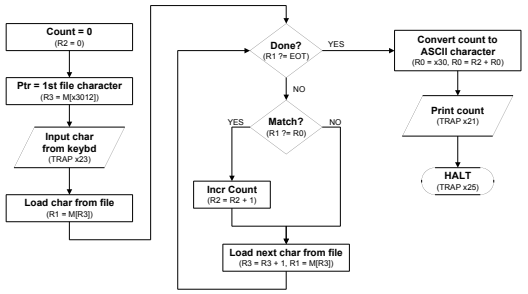
Revisited: Counting Characters (From Ch 5 & 6)

Count the occurrences of a character in a file

Remember this?



Revisited: Counting Characters (From Ch 5 & 6)



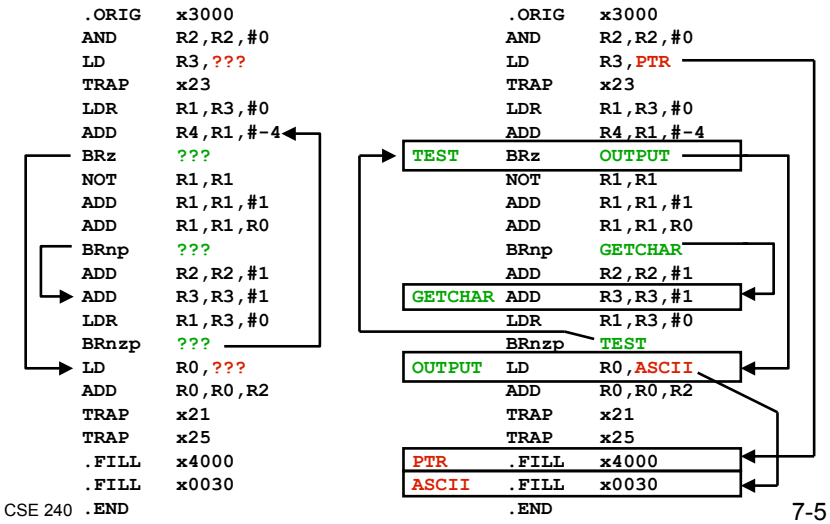

```
R2 ← 0 (Count)
R3 ← M[x3012] (Ptr)
Input to R0 (TRAP x23)
R1 ← M[R3]
R4 ← R1 – 4 (EOT)
BRz x????
R1 ← NOT R1
R1 ← R1 + 1
R1 ← R1 + R0
BRnp x????
R2 ← R2 + 1
R3 ← R3 + 1
R1 ← M[R3]
BRnzp x????
R0 ← M[x3013]
R0 ← R0 + R2
Print R0 (TRAP x21)
HALT (TRAP x25)
x3012: x4000
x3013: x0030
```


Assembly Language: Opcode + Operands

```
R2 ← 0 (Count)
R3 ← M[x3012] (Ptr)
Input to R0 (TRAP x23)
R1 ← M[R3]
R4 ← R1 – 4 (EOT)
BRz x????
R1 ← NOT R1
R1 ← R1 + 1
R1 ← R1 + R0
BRnp x????
R2 ← R2 + 1
R3 ← R3 + 1
R1 ← M[R3]
BRnzp x????
R0 ← M[x3013]
R0 ← R0 + R2
Print R0 (TRAP x21)
HALT (TRAP x25)
x3012: x4000
x3013: x0030
```

```
.ORIG   x3000
AND     R2,R2,#0
LD      R3,???
TRAP    x23
LDR     R1,R3,#0
ADD     R4,R1,#-4
BRz     ????
NOT     R1,R1
ADD     R1,R1,#1
ADD     R1,R1,R0
BRnp    ???
ADD     R2,R2,#1
ADD     R3,R3,#1
LDR     R1,R3,#0
BRnzp   ???
LD      R0,???
ADD     R0,R0,R2
TRAP    x21
TRAP    x25
.FILL   x4000
.FILL   x0030
.END
```

Introducing Labels for PC-Relative Locations



```
        .ORIG   x3000                .ORIG   x3000
        AND     R2,R2,#0             AND     R2,R2,#0
        LD      R3,???               LD      R3,PTR
        TRAP    x23                  TRAP    x23
        LDR     R1,R3,#0             LDR     R1,R3,#0
        ADD     R4,R1,#-4            ADD     R4,R1,#-4
        BRz     ???          TEST    BRz     OUTPUT
        NOT     R1,R1                NOT     R1,R1
        ADD     R1,R1,#1             ADD     R1,R1,#1
        ADD     R1,R1,R0             ADD     R1,R1,R0
        BRnp    ???                  BRnp    GETCHAR
        ADD     R2,R2,#1             ADD     R2,R2,#1
        ADD     R3,R3,#1     GETCHAR ADD     R3,R3,#1
        LDR     R1,R3,#0             LDR     R1,R3,#0
        BRnzp   ???                  BRnzp   TEST
        LD      R0,???       OUTPUT  LD      R0,ASCII
        ADD     R0,R0,R2             ADD     R0,R0,R2
        TRAP    x21                  TRAP    x21
        TRAP    x25                  TRAP    x25
        .FILL   x4000        PTR     .FILL   x4000
        .FILL   x0030        ASCII   .FILL   x0030
        .END                         .END
```


Assembly: Human-Readable Machine Language

Computers like ones and zeros…

```
0001110010000110
```

Humans like mnemonics …

```
ADD     R6, R2, R6    ; increment index reg.
Opcode  Dest Src1 Src2  Comment
```

Assembler

- A program that turns mnemonics into machine instructions
- ISA-specific
- Mnemonics for opcodes
- One assembly instruction translates to one machine instruction
- 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
        BRp    AGAIN          ; the iteration.
        HALT
;
NUMBER: .BLKW 1
SIX:    .FILL  x0006
;
        .END
```


LC-3 Assembly Language Syntax

Each line of a program is one of the following:

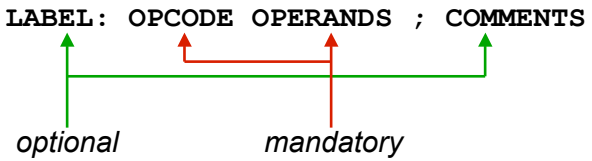
- An instruction
- An assembler directive (or pseudo-op)
- A comment

Whitespace (between symbols) and case are ignored

Comments (beginning with ";") are also ignored

Labels for instructions can be followed by ":"

An instruction has the following format:



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 R0, R1, …, R7
- Numbers -- indicated by # (decimal) or x (hex) or b (binary)
  - Examples: "#10" is "xA" is "b1010"
- Label -- symbolic name of memory location
- Separated by comma
- Number, order, and type correspond to instruction format
  - 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

Opcode	*Operand*	*Meaning*
`.ORIG`	address	starting address of program
`.END`		end of program
`.FILL`	value	allocate one word, initialize with value
`.BLKW`	number	allocate multiple words of storage, value unspecified
`.STRINGZ`	n-character string	allocate n+1 locations, initialize w/characters and null terminator

Muti-Word Assembler Directives

.BLKW #4 shorthand for:

- .FILL x0
- .FILL x0
- .FILL x0
- .FILL x0

.STRINGZ "Hello" shorthand for:

- .FILL x48 ; 'H'
- .FILL x65 ; 'e'
- .FILL x6C ; 'l'
- .FILL x6C ; 'l'
- .FILL x6F ; 'o'
- .FILL x0 ; NULL terminator

Trap Codes

LC-3 assembler provides "pseudo-instructions" for each trap code, so you don't have to remember them

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

Style Guidelines

Improve the readability of your programs

- Formatting: start labels, opcode, operands in same column
- Use comments to explain what each register does
- Give explanatory comment for most instructions
- Use meaningful symbolic names
- Provide comments between program sections
- Each line must fit on the page -- no wraparound or truncations
  - ➢ Long statements split in aesthetically pleasing manner

Use structured programming constructs

- From chapter 6
- Don't be overly clever (may make it harder to change later)

High-level programming style is similar

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)

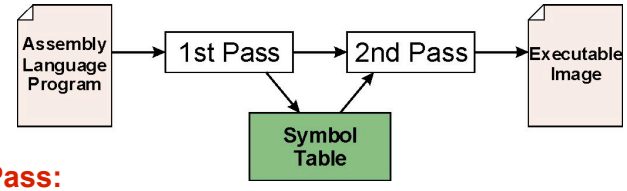
```
;
; Test character for match.  If a match, increment count.
;
        NOT     R1, R1
        ADD     R1, R1, #1  ; R1 = -R1
        ADD     R1, R1, R0  ; R1 == R0?
        BRnp    GETCHA      ; If no match, do not increment
        ADD     R2, R2, #1
;
; Get next character from file.
;
GETCHA: ADD     R3, R3, #1  ; Point to next character.
        LDR     R1, R3, #0  ; R1 gets next char to test
        BRnzp   TEST
;
; Output the count.
;
OUTPUT: LD      R0, ASCII   ; Load the ASCII template
        ADD     R0, R0, R2  ; Covert 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

Program that converts 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. Begin with 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-blank line in the program:
   a) If line contains a label, put label/LC pair into symbol table
   b) Increment LC
      - NOTE: If statement is `.BLKW` or `.STRINGZ`, increment LC by the number of words allocated
      - A line with only a comment is considered "blank"
3. Stop when `.END` statement is reached

Second Pass: Generating Machine Code

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 errors:

- Improper number or type of arguments
  - 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 from instruction
  - Can't use PC-relative addressing mode

Assembly Process Example: First Pass

```
                .ORIG   x3000
x3000           AND     R2,R2,#0
x3001           LD      R3,PTR
x3002           TRAP    x23
x3003           LDR     R1,R3,#0
x3004           ADD     R4,R1,#-4
x3005 TEST      BRz     OUTPUT
x3006           NOT     R1,R1
X3007           ADD     R1,R1,#1
x3008           ADD     R1,R1,R0
x3009           BRnp    GETCHAR
x300A           ADD     R2,R2,#1
x300B GETCHAR   ADD     R3,R3,#1
x300C           LDR     R1,R3,#0
x300D           BRnzp   TEST
x300E OUTPUT    LD      R0,ASCII
x300F           ADD     R0,R0,R2
x3010           TRAP    x21
x3011           TRAP    x25
x3012 ASCII     .FILL   x0030
x3013 PTR       .FILL   x4000
                .END
```

Symbol	Address
TEST	x3005
GETCHAR	x300B
OUTPUT	x300E
ASCII	x3012
PTR	x3013

Assembly Process Example: Second Pass

```
                .ORIG   x3000
x3000           AND     R2,R2,#0
x3001           LD      R3,PTR
x3002           TRAP    x23
x3003           LDR     R1,R3,#0
x3004           ADD     R4,R1,#-4
x3005 TEST      BRz     OUTPUT
x3006           NOT     R1,R1
x3007           ADD     R1,R1,#1
x3008           ADD     R1,R1,R0
x3009           BRnp    GETCHAR
x300A           ADD     R2,R2,#1
x300B GETCHAR   ADD     R3,R3,#1
x300C           LDR     R1,R3,#0
x300D           BRnzp   TEST
x300E OUTPUT    LD      R0,ASCII
x300F           ADD     R0,R0,R2
x3010           TRAP    x21
x3011           TRAP    x25
x3012 ASCII     .FILL   x0030
x3013 PTR       .FILL   x4000
                .END
```

```
0101 010 010 1 00000
0010 011 000010001
1111 0000 00100011
.
.
```

Symbol	Address
TEST	x3005
GETCHAR	x300B
OUTPUT	x300E
ASCII	x3012
PTR	x3013

LC-3 Assembler

Generates two different output files

Object file (.obj)

- Binary representation of the program

Symbol file (.sym)

- Includes names of labels (also known as symbols)
- Used by simulator to make code easier to read
- A text file of symbol mappings

Object File Format

LC-3 object file contains

- Starting address (location where program must be loaded), followed by…
- Machine instructions
- (Real-world object file formats can be more complicated)

LC-3 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
        .
        .
        .
```

Using 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

- Load multiple object files into memory, then start executing at a desired address
- System routines, such as keyboard input, are loaded with OS
  - OS code starts at 0x0200
  - 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