

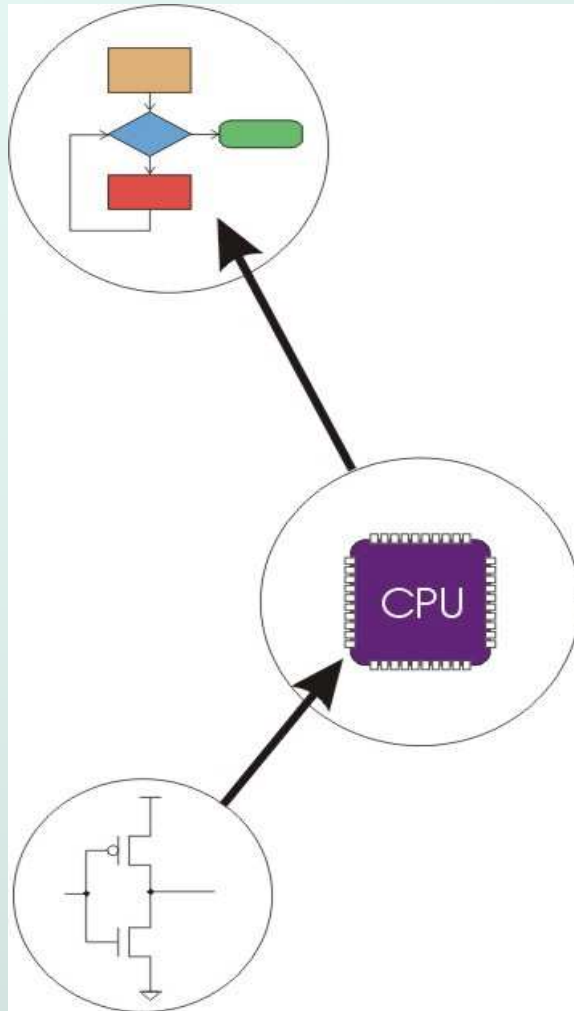
# Chapter 7

## Assembly Language

Original slides from Gregory Byrd, North  
Carolina State University

Modified by C. Wilcox, Y. Malaiya  
Colorado State University

# Computing Layers



Problems

Algorithms

Language

Instruction Set Architecture

Microarchitecture

Circuits

Devices



# Human-Readable Machine Language

- Computers like ones and zeros...

0001110010000110

- Humans like symbols...

**ADD R6,R2,R6** ; *increment index reg.*

- **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
- additional operations for allocating storage and initializing data

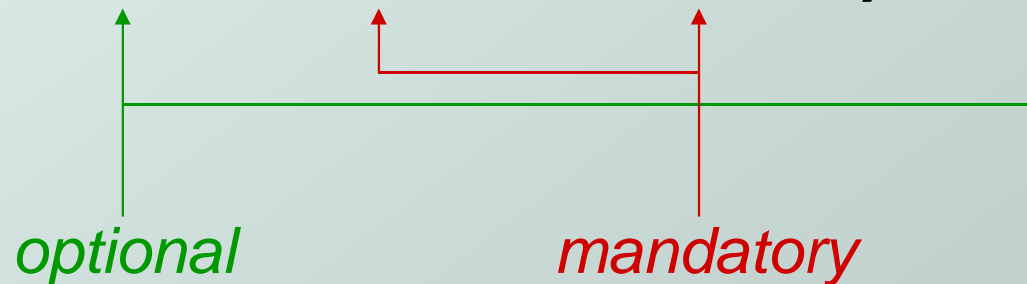
# An Assembly Language Program

```
;
; Program to multiply a number by six
;
    .ORIG    x3050
    LD      R1, SIX          ; R1 has constant
    LD      R2, NUMBER      ; R2 has variable
    AND     R3, R3, #0      ; R3 has product
;
; The inner loop
;
AGAIN  ADD   R3, R3, R2     ; R3 += R2
       ADD   R1, R1, #-1   ; R1 is loop counter
       BRp  AGAIN         ; conditional branch
;
       HALT
;
NUMBER .BLKW  1           ; variable
SIX    .FILL  x0006      ; constant
;
    .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 and case are ignored.
- Comments (beginning with “;”) are also ignored.
- An instruction has the following format:

**LABEL OPCODE OPERANDS ; COMMENTS**



# Opcodes and Operands

## ● Opcodes

- reserved symbols that correspond to LC-3 instructions
- listed in Appendix A
  - example: `ADD, AND, LD, LDR, ...`

## ● Operands

- registers -- specified by Rn, n is the register number
- numbers -- indicated by # (decimal) or x (hex)
- label -- symbolic name of memory location
- separated by comma
- number, order, and type correspond to instruction format
  - example:

```
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 symbolic name to the address of line

● example:      `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 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

<i>Opcode</i>	<i>Operand</i>	<i>Meaning</i>
<b>.ORIG</b>	<b>address</b>	starting address of program
<b>.END</b>		end of program
<b>.BLKW</b>	<b>n</b>	allocate n words of storage
<b>.FILL</b>	<b>n</b>	allocate one word, initialize with value n
<b>.STRINGZ</b>	<b>n-character string</b>	allocate n+1 locations, initialize w/chars and null terminator



# Trap Codes

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

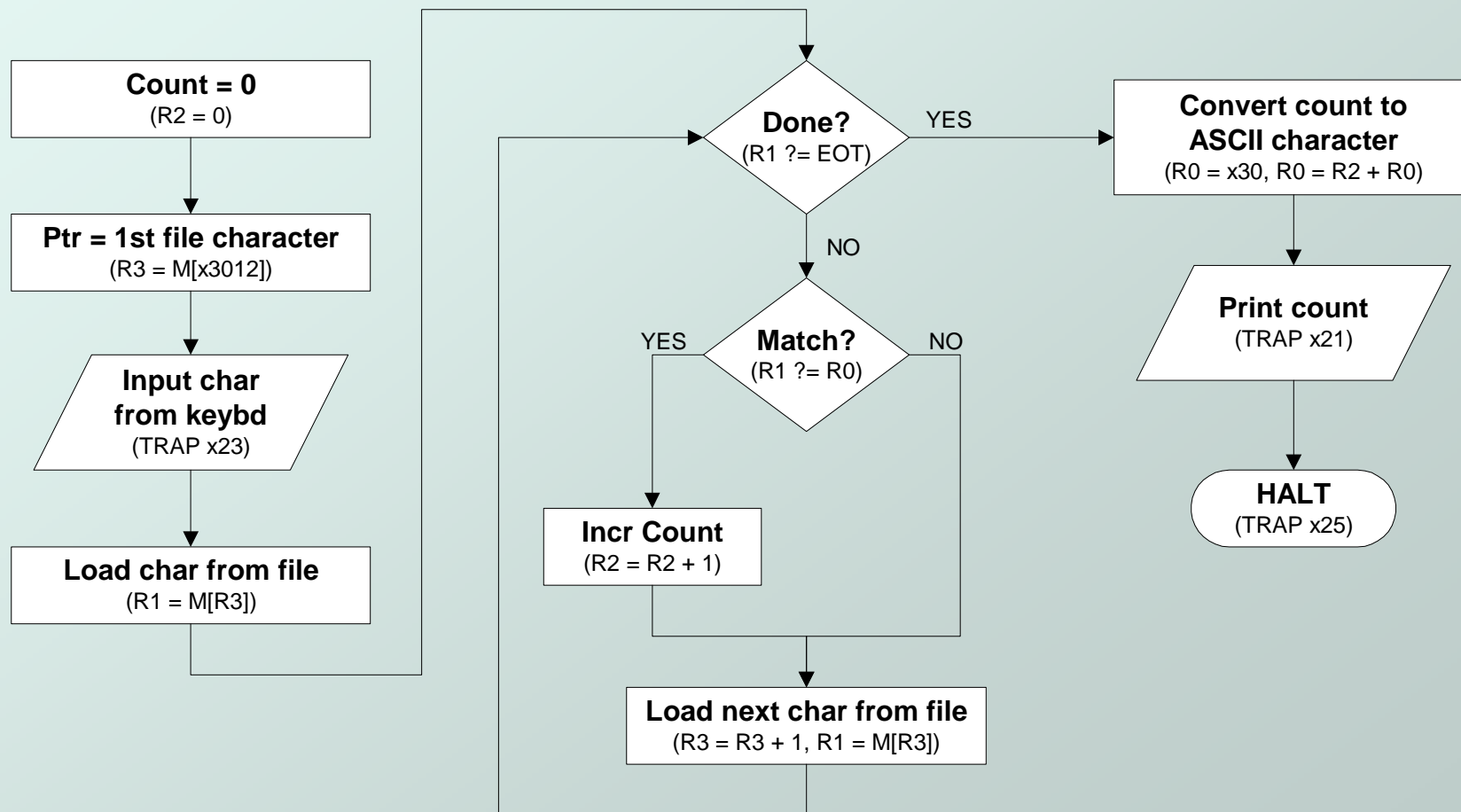
<i>Code</i>	<i>Equivalent</i>	<i>Description</i>
<b>HALT</b>	<b>TRAP x25</b>	Halt execution and print to console.
<b>IN</b>	<b>TRAP x23</b>	Print prompt on console, read character (in R0[7:0]) from keyboard.
<b>OUT</b>	<b>TRAP x21</b>	Write one character (in R0[7:0]) to console.
<b>GETC</b>	<b>TRAP x20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

# Style Guidelines

- Use the following style guidelines to improve 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?**



# Char Count in Assembly Language (1 of 3)

```
;  
; Program to count occurrences of a char in a file.  
; Character to be input from the keyboard.  
; Result to be displayed on the monitor.  
; Program only works if <= 9 occurrences are found.  
;  
; Initialization  
;  
        .ORIG      x3000  
        AND        R2, R2, #0    ; R2 is counter  
        LD         R3, PTR      ; R3 is pointer to chars  
        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  
        BRZ        OUTPUT      ; If done, prepare output
```

# Char Count in Assembly Language (2 of 3)

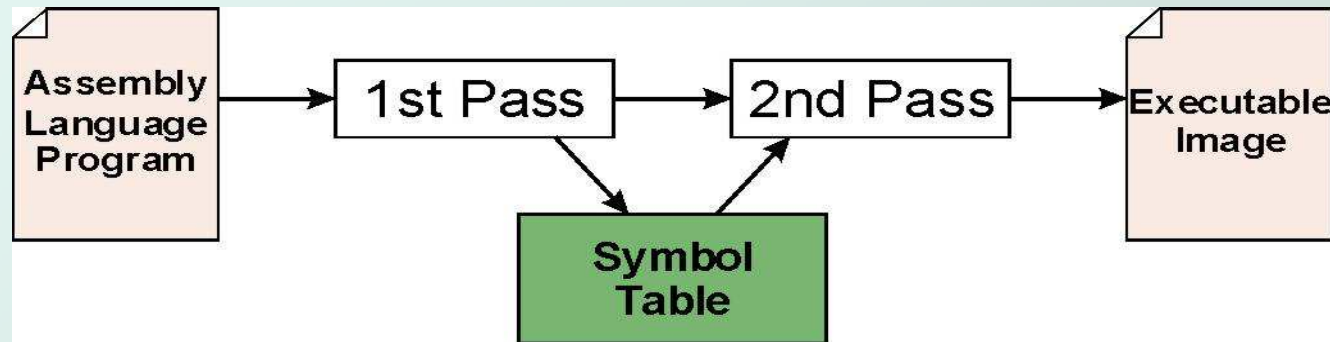
```
;  
; Test character for match, if so increment count.  
;  
        NOT    R1, R1  
        ADD    R1, R1, R0 ; If match, R1 = xFFFF  
        NOT    R1, R1     ; If match, R1 = x0000  
        BRnp  GETCHAR    ; No match, no increment  
        ADD    R2, R2, #1  
  
;  
; Get next character from file.  
;  
GETCHAR ADD    R3, R3, #1 ; Point to next character.  
        LDR    R1, R3, #0 ; R1 gets next char to test  
        BRnzp TEST  
  
;  
; Output the count.  
;  
OUTPUT LD     R0, ASCII   ; Load the ASCII template  
        ADD    R0, R0, R2 ; Covert binary to ASCII  
        OUT    ; ASCII code 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 7.2 (slide 12)

Symbol	Address

# Second Pass: Generating Machine Language

- For each executable assembly language statement, generate the 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`  
`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

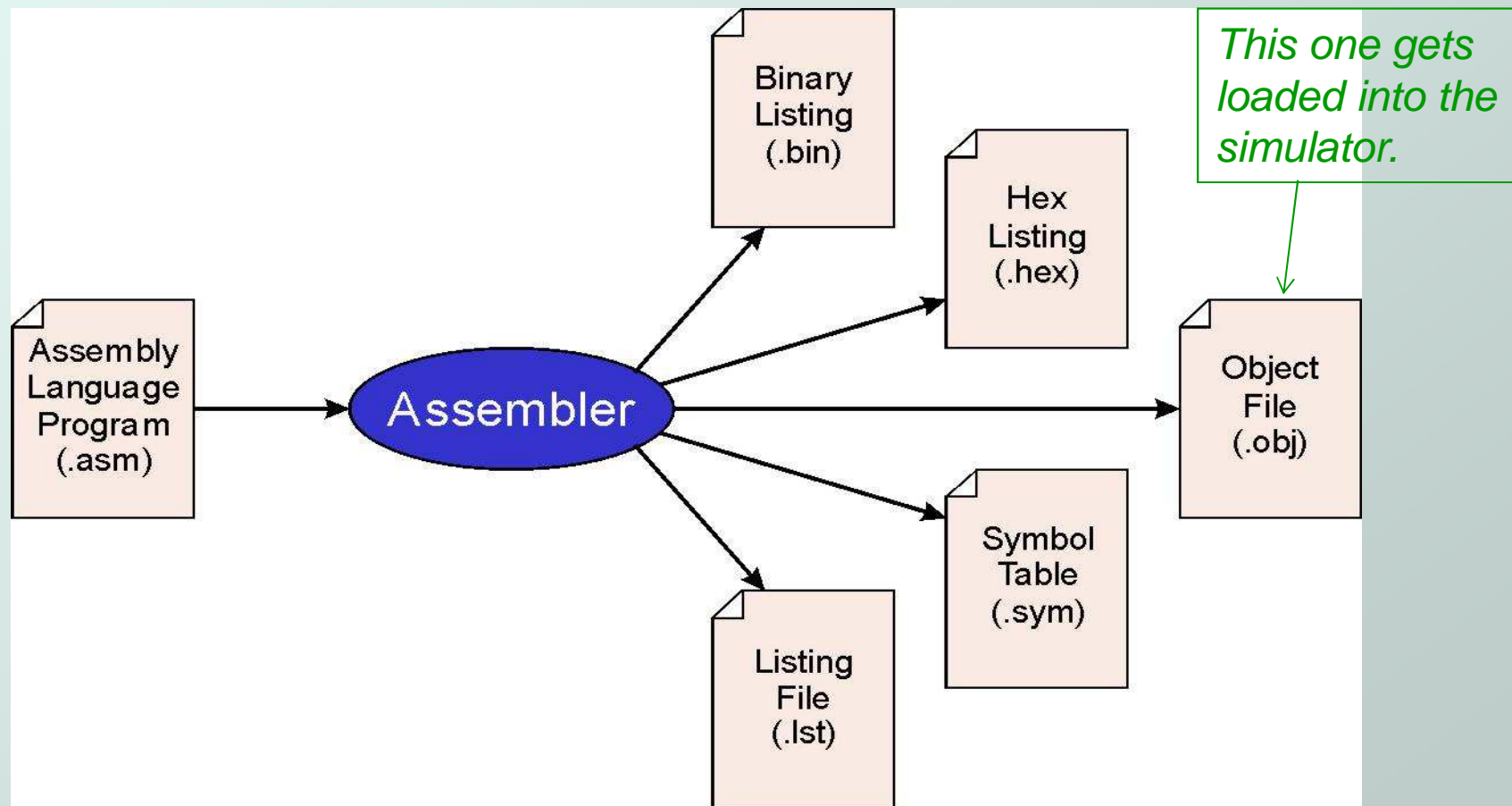
# Practice

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

Statement	Machine Language
LD R3, PTR	
ADD R4, R1, #-4	
LDR R1, R3, #0	
BRnp GETCHAR	

# LC-3 Assembler

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



# 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:

001100000000000000	←	.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 at a desired address.
  - system routines, such as keyboard input, are loaded automatically
    - loaded into “system memory,” below x3000
    - user code 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 searches symbol tables of other modules to resolve symbols and generate all code before loading