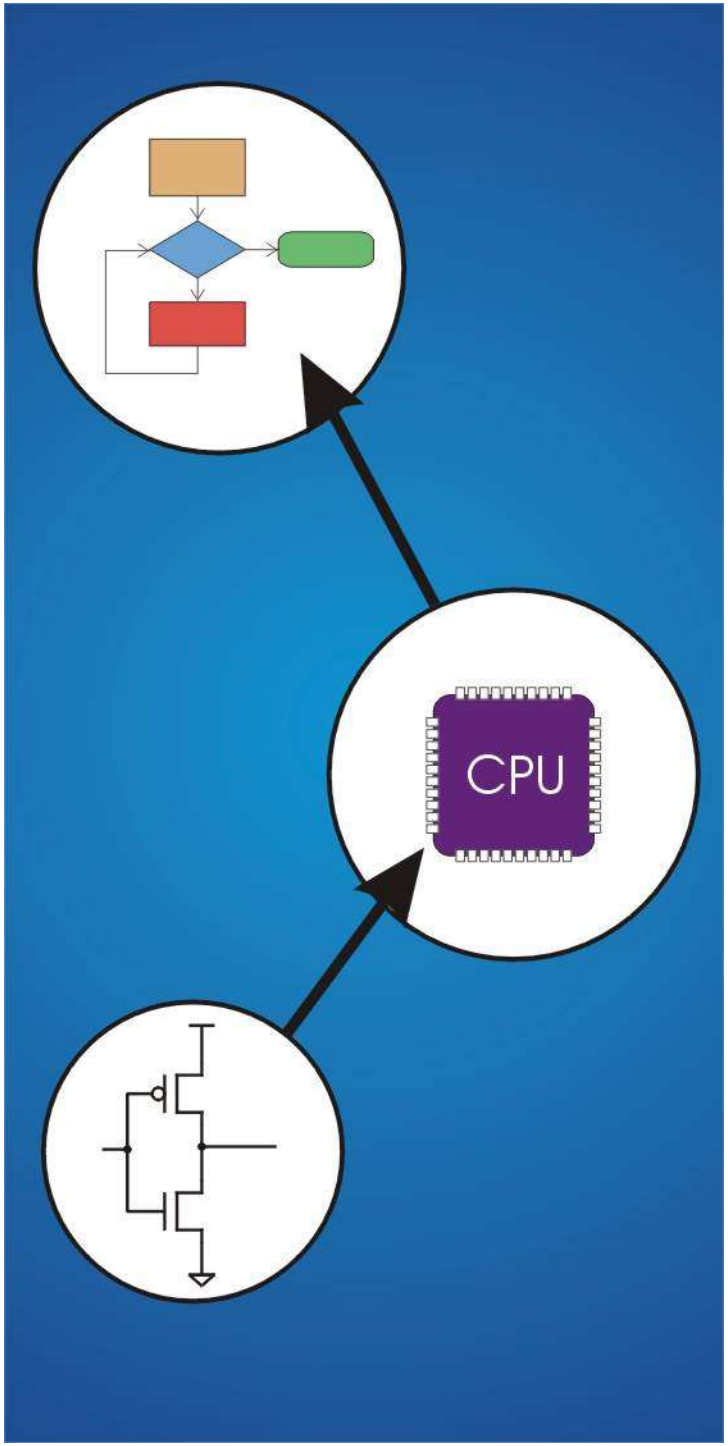


Assembly Language

Computer Architecture & Organization (CSE2003)

(Fall 2019-20)

Prof. Anand Motwani
Faculty, SCSE,
email-id:
anand.motwani@vitbhopal.ac.in

Contents

- Assembly Language
- Assembly Language Programming

Session Objectives

- Understand Assembly Language programming.

Assembly language

- **Assembly language is a low-level programming language for a computer or other programmable device specific to a particular computer architecture in contrast to most high-level programming languages.**

- **Each family of processors has its own set of instructions for handling various operations such as getting input from keyboard, displaying information on screen and performing various other jobs. These set of instructions are called 'machine language instructions'.**
- **the low-level assembly language is designed for a specific family of processors that represents various instructions in symbolic code and a more understandable form.**

Few advantages of using assembly language are −

- **It requires less memory and execution time;**
- **It allows hardware-specific complex jobs in an easier way;**
- **It is suitable for time-critical jobs;**
- **It is most suitable for writing interrupt service routines and other memory resident programs.**

Assembly Language Statements

Assembly language programs consist of three types of statements −

- Executable instructions or instructions,
- Assembler directives or pseudo-ops, and
- Macros.
- The executable instructions or simply instructions tell the processor what to do. Each instruction consists of an operation code (opcode). Each executable instruction generates one machine language instruction.
- The assembler directives or pseudo-ops tell the assembler about the various aspects of the assembly process. These are non-executable and do not generate machine language instructions.

LC-3

- Little Computer 3, or LC-3, is a type of computer educational programming language, an assembly language, which is a type of low-level programming language.
- The language is less complex than x86 assembly but has many features similar to those in more complex languages.
- The LC-3 specifies a word size of 16 bits for its registers and uses a 16-bit addressable memory with a 2^{16}-location address space.
- Instructions are 16 bits wide and have 4-bit opcodes.
- The LC-3 instruction set implements fifteen types of instructions, with a sixteenth opcode reserved for later use.
- The calling convention for C functions on the LC-3 is similar to that implemented by other systems, such as the x86 ISA.

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 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:

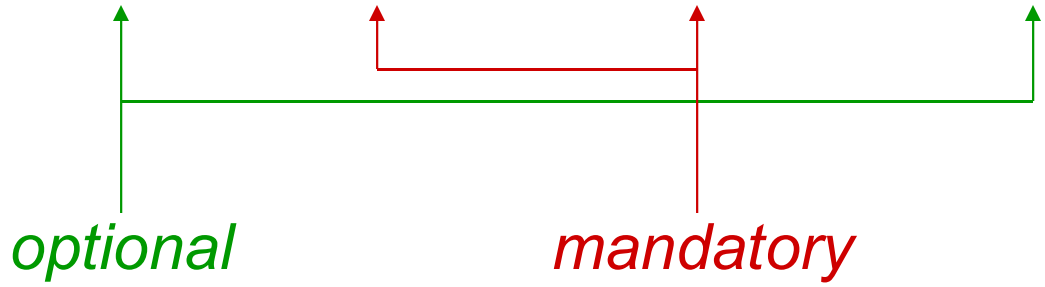
- **an instruction**
- **an assember directive (or pseudo-op)**
- **a comment**

Whitespace (between symbols) and case are ignored.

Comments (beginning with ";") are also ignored.

An instruction has the following format:

```
LABEL OPCODE OPERANDS ; COMMENTS
```



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
- 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
`.BLKW`	n	allocate n words of storage
`.FILL`	n	allocate one word, initialize with value n
`.STRINGZ`	n-character string	allocate n+1 locations, initialize w/characters and 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.

```
; LC−3 P rog ram t h a t d i s p l a y s 2
 ; ” H ell o World ! ” t o t h e c o n s o l e

        .ORIG x3000
LEA R0 , HW        ; l o a d a d d r e s s o f s t r i n g
PUTS               ; o u t p u t s t r i n g t o c o n s o l e
HALT               ; end p r o g ram
HW .STRINGZ ” H ell o World ! ”
 .END
```

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.**

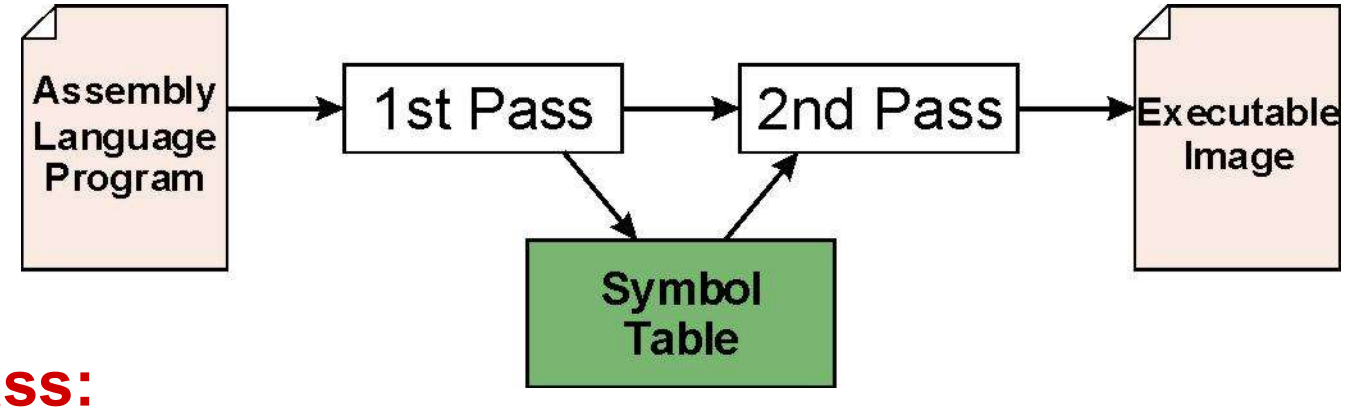
A subroutine for the function f(n) = 2n+3.

```
1           LDI R0, N     ; Argument N is now in R0
2           JSR F         ; Jump to subroutine F.
3           STI R1, FN
4           HALT
5  N        .FILL 3120    ; Address where n is located
6  FN       .FILL 3121    ; Address where fn will be stored.
7                          ; Subroutine F begins
8  F        AND R1, R1, x0 ; Clear R1
9           ADD R1, R0, x0 ; R1 ← R0
10          ADD R1, R1, R1 ; R1 ← R1 + R1
11          ADD R1, R1, x3 ; R1 ← R1 + 3.  Result is in R1
12          RET            ; Return from subroutine
13          END
```

- **Additional Material on Next Slides**

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).

Symbol	Address

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 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**

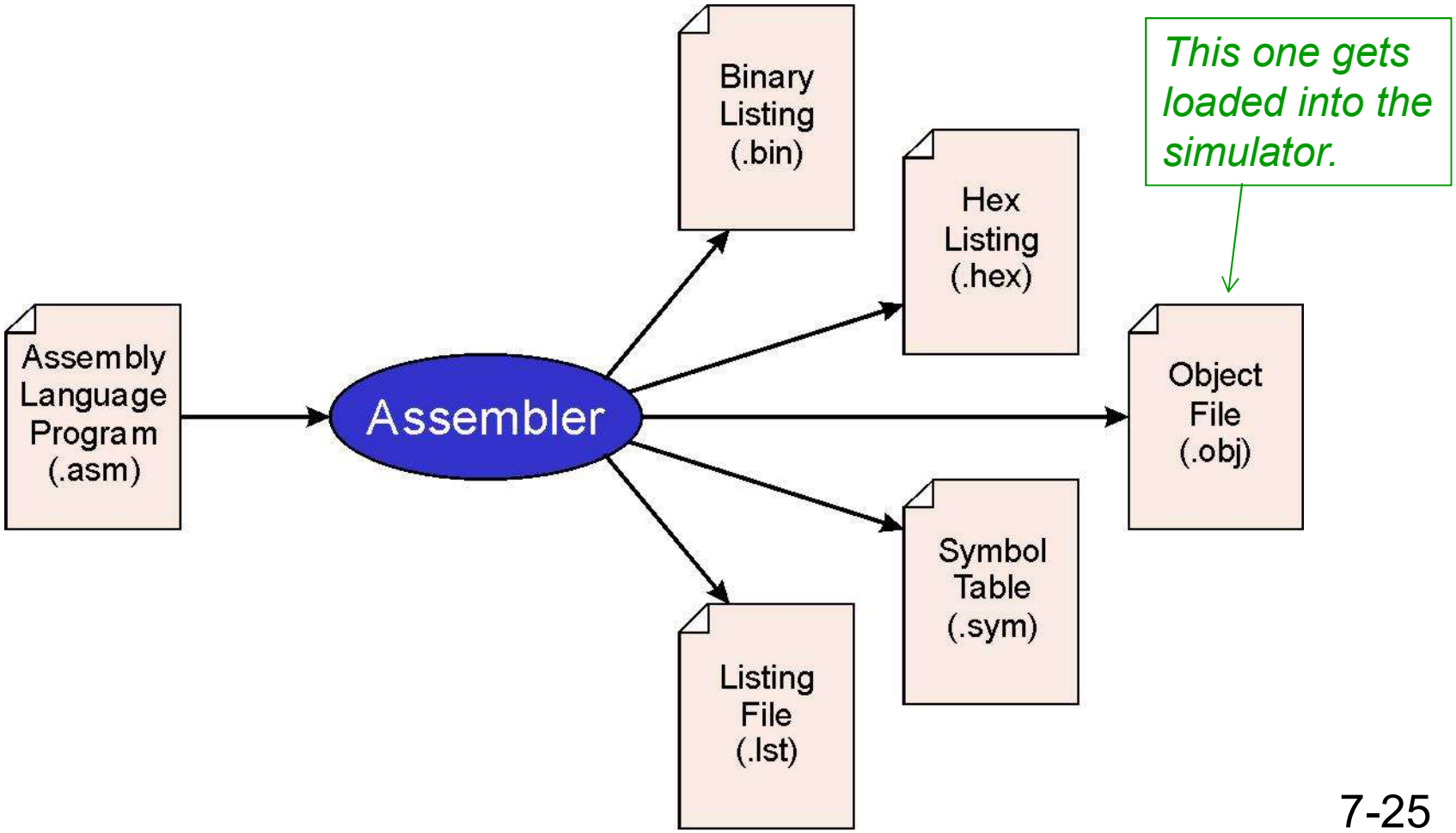
Practice

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.



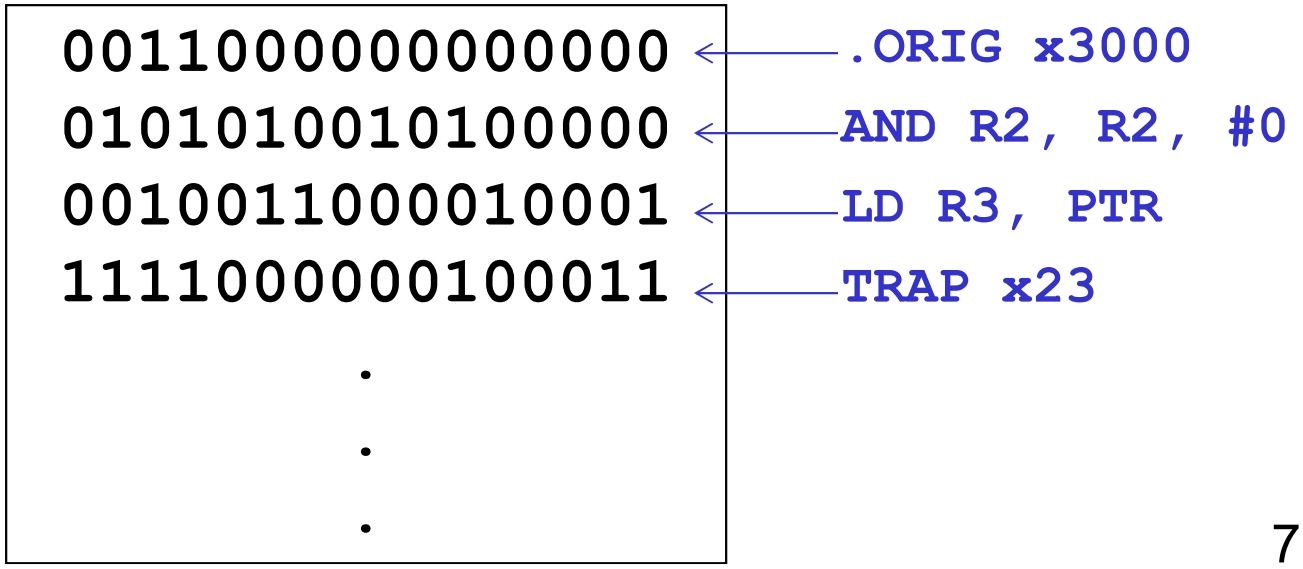
Object File Format

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 this:



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**