COMP1003 Computer Organization

Lecture 10 From Machine Language to Assembly Language

```
                .ORIG   x3000
3000            AND     R2, R2, x0      ; clear R2
3001            LDI     R1, Input       ; load word into R1
3002    Count   BRz     Report          ; if 0, done counting
3003            BRp     Shift           ; if >0, skip ADD
3004            ADD     R2, R2, x1      ; increment count
3005    Shift   ADD     R1, R1, R1      ; shift left 1 bit
3006            BRnzp   Count           ; go back up
3007    Report  AND     R3, R2, x1      ; LSB 1 or 0?
3008            STI     R3, Output      ; store results
3009            TRAP    x25             ; halt program
300A    Input   .FILL   x3200           ; address of input
300B    Output  .FILL   x3201           ; address of output
```

United International College

Machine Language
vs
Assembly Language

- Computers like zeros and ones:
  - 0001110010000110
- Humans like symbols:
  - ADD R6,R2,R6 ;
  - meaning: R6 ← R2 + R6
- Assembler is a program that turns symbols into binary machine instructions.

An Assembly Language Program

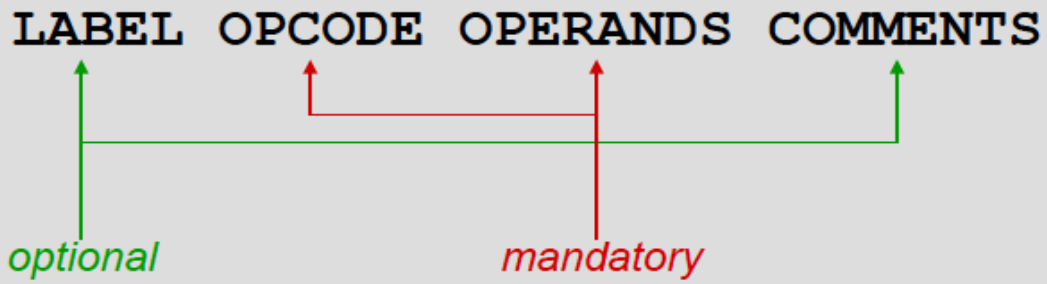
```
;
; Program to multiply a number by the constant 6
;
        .ORIG  x3050
        LD     R1, SIX
        LD     R2, NUMBER
        AND    R3, R3, #0     ; Clear R3.  It will
                              ; contain the product.
; The inner loop
;
AGAIN   ADD    R3, R3, R2
        ADD    R1, R1, #-1    ; 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.

LC-3 Assembly Language Instruction

- An instruction has the following format:



- Example

```
LOOP    ADD  R1,R1,#-1;  loop start
        BRp  LOOP;
```

Opcodes and Operands

- Opcodes
  - reserved symbols that correspond to LC-3 instructions
  - listed in Appendix A
  - example: ADD, AND, LD, LDR, …
- Operands
  - Registers: Rn
  - Numbers: # (Dec) or x (Hex)
  - Label: symbolic name of memory location
  - Number, order and type correspond to instruction format

Example

```
ADD R1,R1,R3
ADD R1,R1,#3
LD    R6, NUMBER
BRz  LOOP
```

Labels

- placed at the beginning of the line
- assigns a symbolic name to the address corresponding to line
- example:

```
LOOP ADD R1,R1,#-1
     Brp LOOP
```

Comments

- 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

Question

- What is the difference between HALT and .END ?

Trap Codes

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

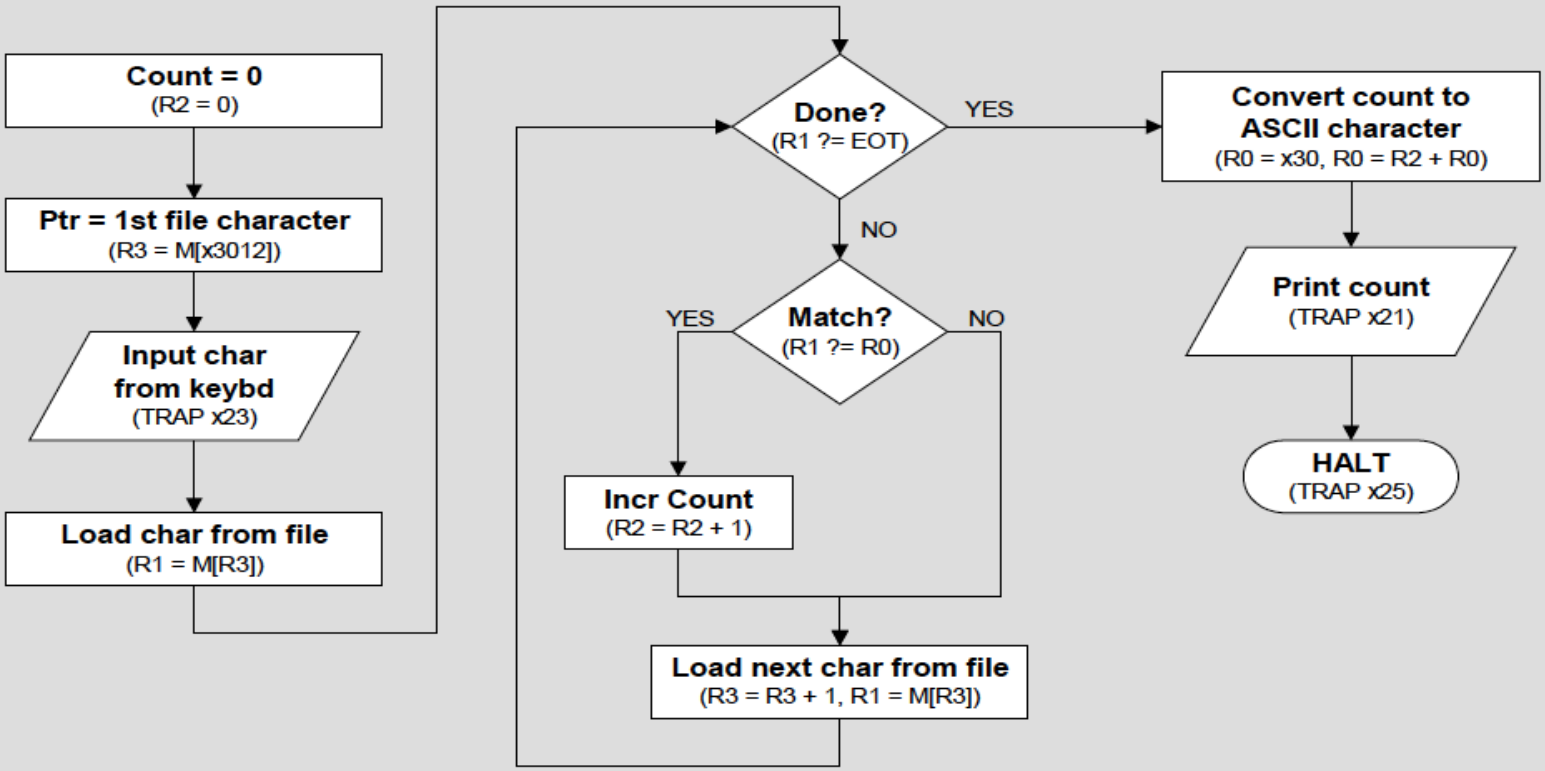
Code	*Equivalent*	*Description*
HALT	TRAP x25	Halt execution and print message to console.
IN	TRAP x23	Print prompt on console, read (and echo) one character from keybd. Character stored in R0[7:0].
OUT	TRAP x21	Write one character (in R0[7:0]) to console.
GETC	TRAP x20	Read one character from keyboard.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

Style Guidelines

0. Provide a program header, with author's name, date, etc., and purpose of program.
1. Start labels, opcode, operands, and comments in same column for each line. (Unless entire line is a comment.)
2. Use comments to explain what each register does.
3. Give explanatory comment for most instructions.
4. Use meaningful symbolic names.
   - Mixed upper and lower case for readability.
   - ASCIItoBinary, InputRoutine, SaveR1
5. Provide comments between program sections.
6. 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.



Code (1 of 2)

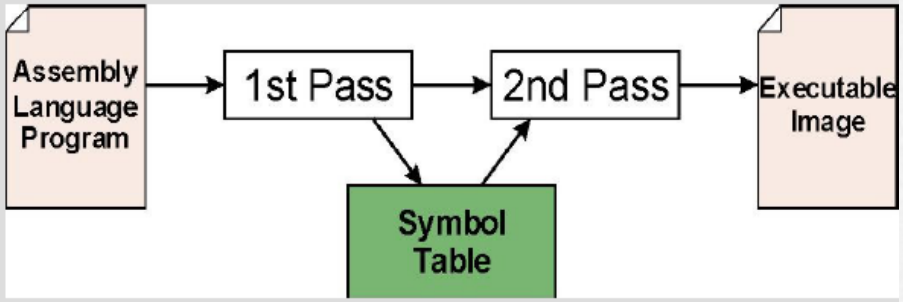
```
;
; Program to count occurrences of a character in a file.
; Character to be input from the keyboard.
; Result to be displayed on the monitor.
; Program only works if no more than 9 occurrences are found.
;
;
; Initialization
;
        .ORIG   x3000
        AND     R2, R2, #0      ; R2 is counter, 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
```

Code (2 of 2)

```
;
; Test character for match.  If a match, increment count.
;
          NOT     R1, R1
          ADD     R1, R1, R0    ; If match, R1 = xFFFF
          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   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
;
; Storage for pointer and ASCII template
;
ASCII     .FILL   x0030
PTR       .FILL   x4000
.END
```

Assembly Process

from (.asm) to (.obj)



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

- Find the .ORIG instruction which tells is the address of the first instruction
- Initialize LC (location counter)
- For each non-empty line, if it contains a label, add label and LC to symbol table, increment LC.
- If statement is .BLKW or STRINGZ, increment LC by the number of words allocated
- Stop when .END is reached

Example

- Construct the symbol table for the program in

Figure 7.1 (page 179).

Symbol	Address

Second Pass: Generating Machine Language

- For each executable assembly language statement, generate the corresponding machine language instruction.

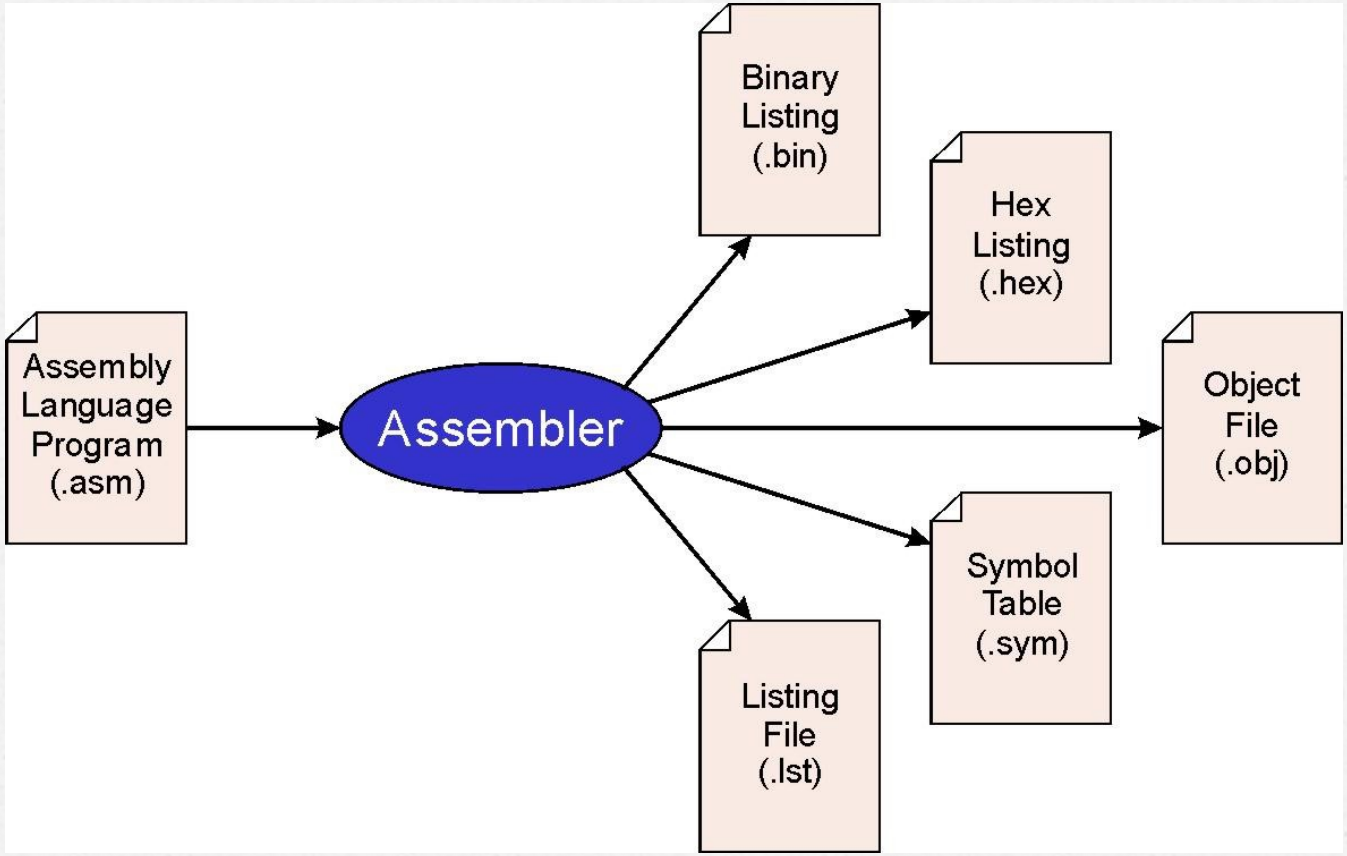
- If operand is a label, look up the address from the symbol table.

Example

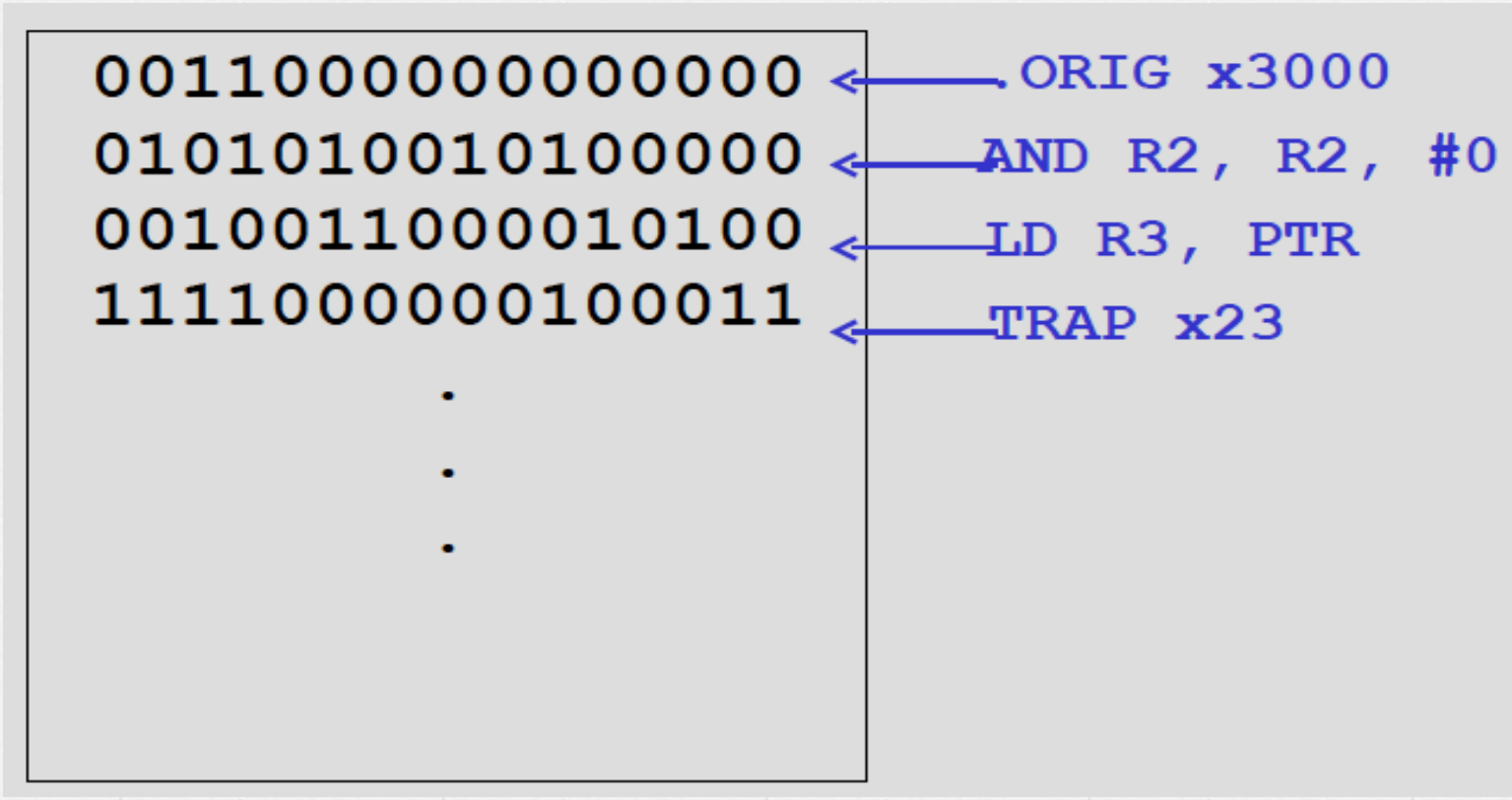
- Using the symbol table constructed earlier, translate these statements into LC-3 machine language.

Statement	Machine Language
`LD R1, SIX`	
`LD R2, NUMBER`	
`AND R3, R3, #0`	
`AGAIN ADD R3,R3,R2`	

LC-3 Assembler



Object File


0011000000000000
0101010010100000
0010011000010100
1111000000100011
.
.
.
ORIG x3000
AND R2, R2, #0
LD R3, PTR
TRAP x23

Multiple Object Files

- LC3 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 x1000
    - by convention, user code should be loaded between x3000 and xCFFF
  - each object file includes a starting address
  - be careful not to load overlapping object file!

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

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

What's next: Calling Sub-routines