Because learning changes everything.®

From Bits and Gates to C and Beyond

Assembly Language

Chapter 7

Human-Friendly Programming

Computers need binary instruction encodings...

```
0001110010000110
```

Humans prefer symbolic languages...

```
a = b + c
```

High-level languages allow us to write programs in clear, precise language that is more like English or math. Requires a program (compiler) to translage from symbolic language to machine instructions.

Examples: C, Python, Fortran, Java, ...

We will introduce C in Chapters 11 to 19.

Assembly Language: Human-Friendly ISA Programming

Assembly Language is a low-level symbolic language, just a short step above machine instructions.

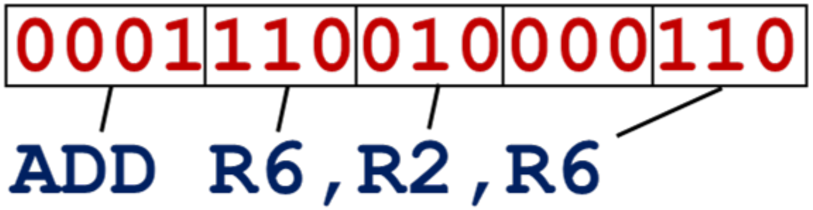
- Don't have to remember opcodes (ADD = 0001, NOT = 1001, ...).
- Give symbolic names to memory locations -- don't have to do binary arithmetic to calculate offsets.
- Like machine instructions, allows programmer explicit, instruction-level specification of program.

Disadvantage:

Not portable. Every ISA has its own assembly language.
Program written for one platform does not run on another.

Assembly Language

Very similar format to instructions -- replace bit fields with symbols.



0001 110 010 000 110

ADD R6,R2,R6

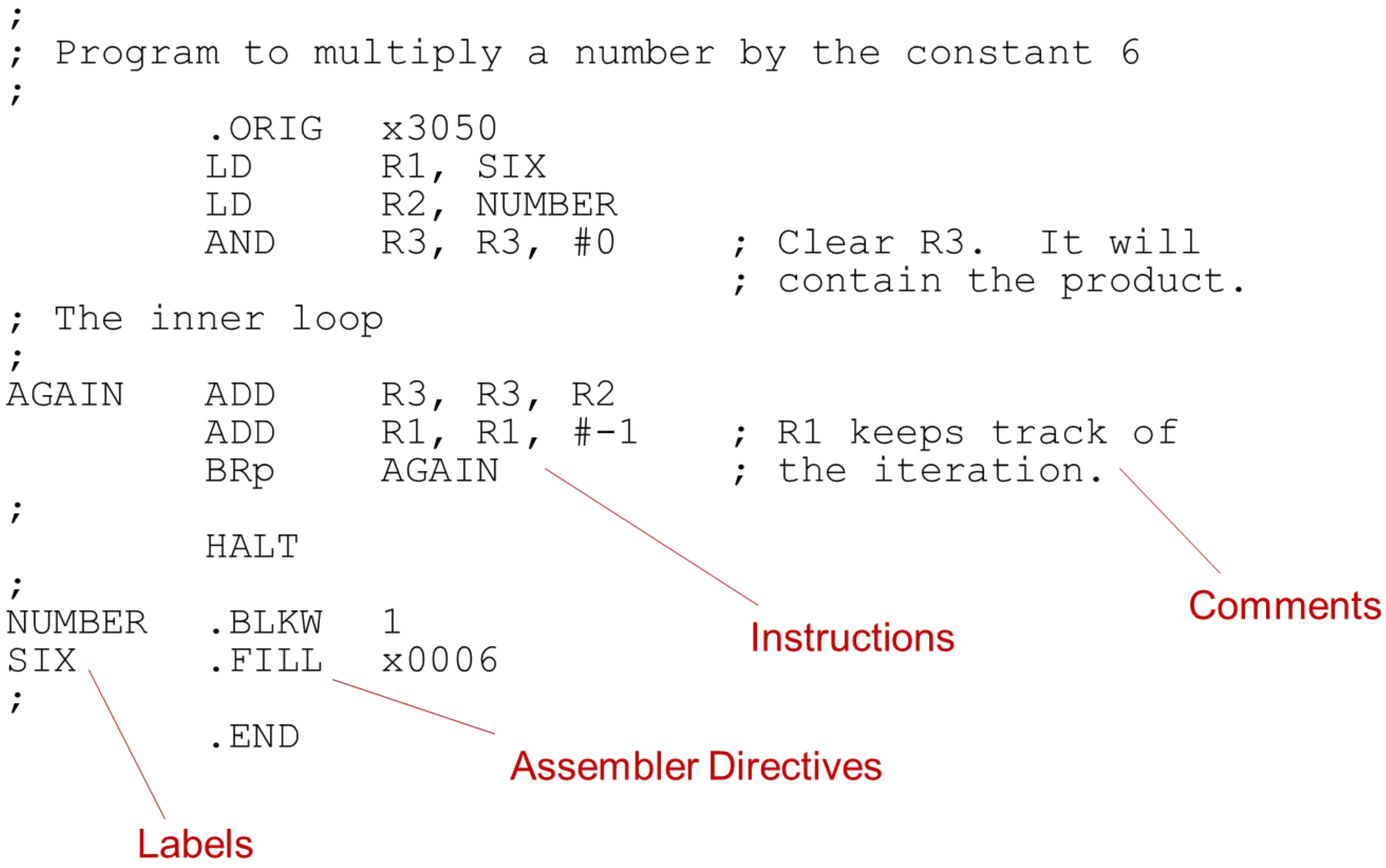
For the most part, one line of assembly language = one instruction.

Some additional features for allocating memory, initializing memory locations, service calls.

Numerical values specified in hexdecimal (x30AB) or decimal (#10).

Access the text alternative for slide images.

Example 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
```

Instructions
Comments
Assembler Directives
Labels

Access the text alternative for slide images.

Assembly Language Syntax

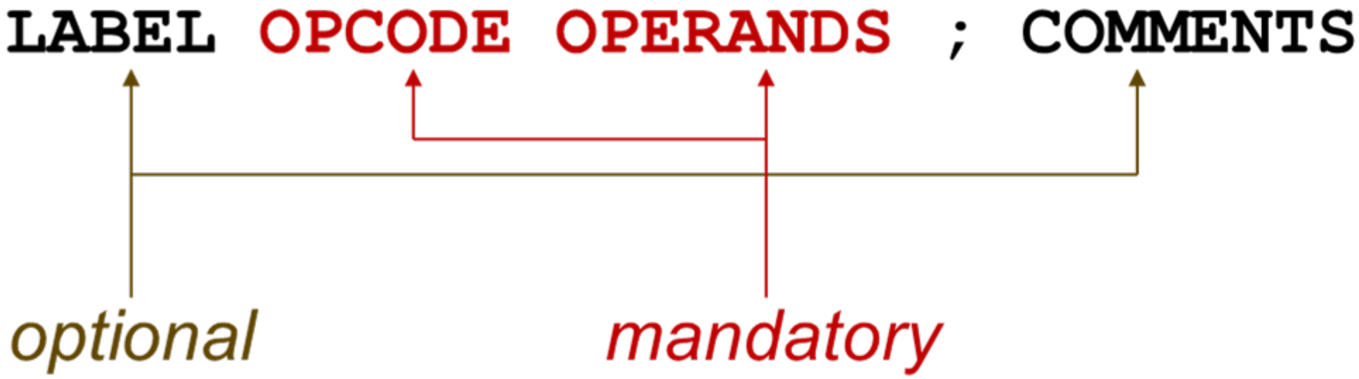
Each line of a program is one of the following:

- 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:



Access the text alternative for slide images.

Mandatory: Opcode and Operands

Opcodes

Reserved symbols that correspond to LC-3 instructions.

Listed in Appendix A and Figure 5.3.

- For example: `ADD`, `AND`, `LD`, `LDR`, …

reserved means that it cannot be used as a label

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 (whitespace ignored).
- Number, order, and type correspond to instruction format.

```
ADD R1,R1,R3    ; DR, SR1, SR2
ADD R1,R1,#3    ; DR, SR1, Imm5
LD  R6,NUMBER   ; DR, address (converted to PCoffset)
BRz LOOP        ; nzp becomes part of opcode, address
```

Optional: Label and Comment

Label

- Placed at the beginning of the line.
- Assigns a symbolic name to the address corresponding to that line.

```
LOOP   ADD    R1,R1,#-1   ; LOOP is address of ADD
       BRp    LOOP
```

Comment

A semicolon, and anything after it on the same line, 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 and empty lines to separate pieces of program.

Assembler Directive

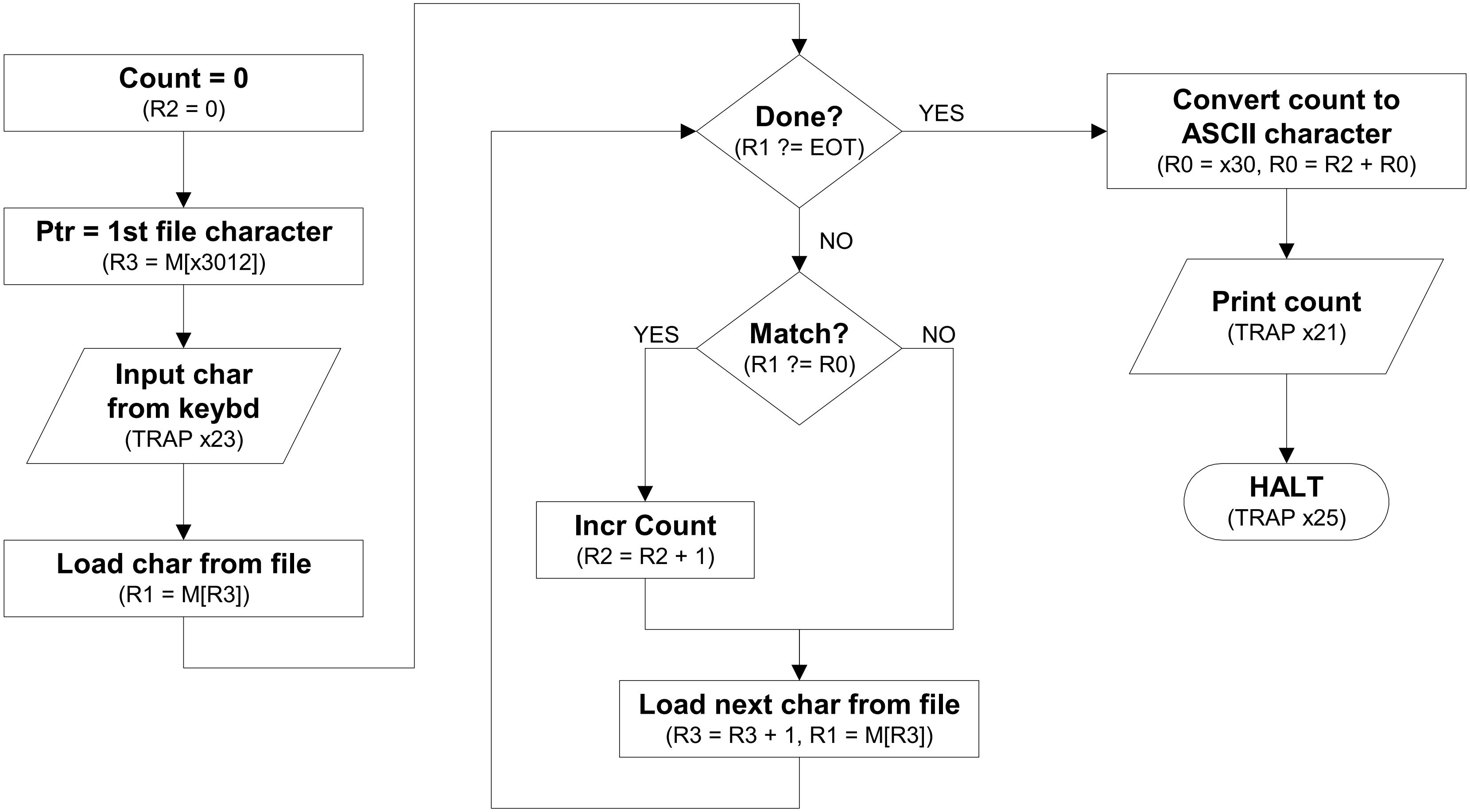
Pseudo-operation

- Does not refer to an actual instruction to be executed.
- Tells the assembler to do something.
- Looks like an instruction, except "opcode" starts with a 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**

Sample Program: Counting Occurrences in a File

Once again, we show the program that counts the number of times (up to nine) a user-specified character appears in a file.



Access the text alternative for slide images.

Assembly Language Program 1

```
;
; 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
           TRAP     x23                 ; 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
;
; Test character for match.  If a match, increment count.
;
           NOT      R1, R1
           ADD      R1, R1, #1
           ADD      R1, R1, R0      ; Compute R0-R1 to compare
           BRnp     GETCHAR         ; If no match, do not increment count
           ADD      R2, R2, #1
```

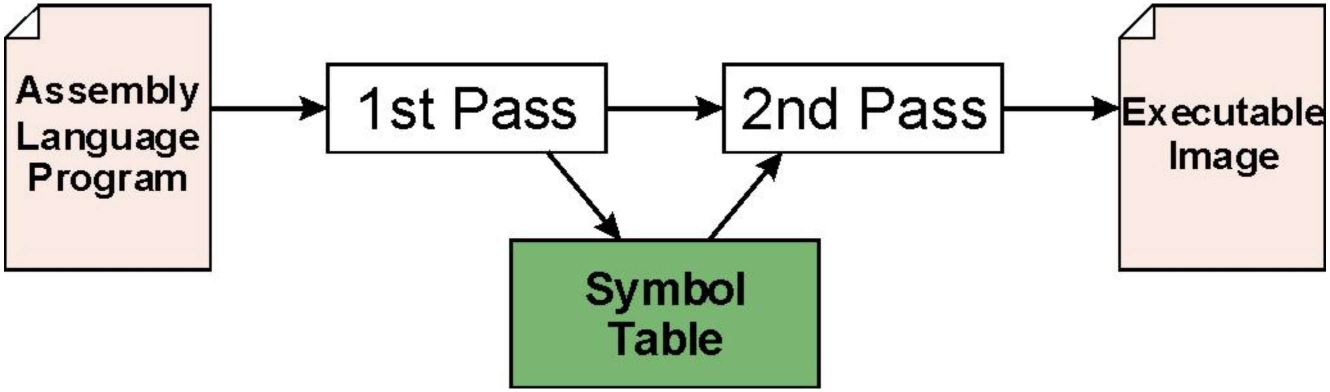
Assembly Language Program 2

```
;
; 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
           TRAP     x21            ; ASCII code in R0 is displayed.
           TRAP     x25            ; Halt machine

;
; Storage for pointer and ASCII template
;
ASCII      .FILL    x0030
PTR        .FILL    x4000
           .END
```

Assembly Process

The assembler is a program that translate an assembly language (.asm) file to a binary object (.obj) file that can be loaded into memory.



First Pass:

- Scan program file, check for syntax errors.
- Find all labels and calculate the corresponding addresses: the *symbol table*.

Second Pass:

- Convert instructions to machine language, using information from symbol table.

Access the text alternative for slide images.

First Pass: Construct 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:
    - If line contains a label, add label and LC to symbol table.
    - 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.

First Pass on Sample Program (Comments Removed)

```
--                 .ORIG    x3000
x3000              AND      R2, R2, #0
x3001              LD       R3, PTR
x3002              TRAP     x23
x3003              LDR      R1, R3, #0
x3004    TEST      ADD      R4, R1, #-4
x3005              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
```

Label	Address
TEST	x3004
GETCHAR	x300B
OUTPUT	x300E
ASCII	x3012
PTR	x3013

Second Pass: Convert to Machine Instructions

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:
   - If line contains an instruction, translate opcode and operands to binary machine instruction. For label, lookup address in symbol table and subtract (LC+1). Increment LC.
   - If line contains .FILL, convert value/label to binary. Increment LC.
   - If line contains .BLKW, create n copies of x0000 (or any arbitrary value). Increment LC by n.
   - If line contains .STRINGZ, convert each ASCII character to 16-bit binary value. Add null (x0000). Increment LC by n+1.
3. Stop when .END statement is reached.

Errors during Code Translation

While assembly language is being translated to machine instructions, several types of errors may be discovered.

- Immediate value too large -- can't fit in Imm5 field.
- Address out of range -- greater than LC+1+255
  or less than LC+1-256.
- Symbol not defined, not found in symbol table.

If error is detected, assembly process is stopped and an error message is printed for the user.

Beyond a Single Object File

Larger programs may be written by multiple programmers, or may use modules written by a third party. Each module is assembled independently, each creating its own object file and symbol table.

To execute, a program must have all of its modules combined into a single executable image.

Linking is the process to combine all of the necessary object files into a single executable.

External Symbols

In the assembly code we're writing, we may want to symbolically refer to information defined in a different module.

For example, suppose we don't know the starting address of the file in our counting program. The starting address and the file data could be defined in a different module.

We want to do this:

```
        PTR   .FILL  STARTofFILE
```

To tell the assembler that `STARTofFILE` will be defined in a different module, we could do something like this:

```
        .EXTERNAL   STARTofFILE
```

This tells the assembler that it's not an error that `STARTofFILE` is not defined. It will be up to the linker to find the symbol in a different module and fill in the information when creating the executable.

Mc
Graw
Hill
Education

Because learning changes everything.®

www.mheducation.com