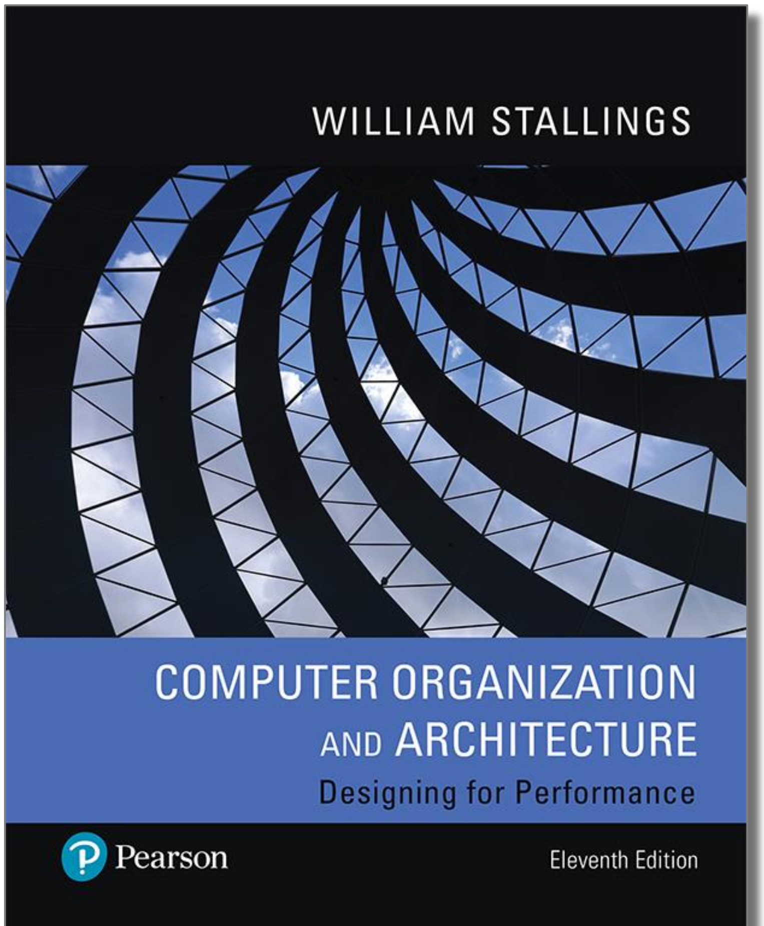
Computer Organization and Architecture
Designing for Performance

11th Edition



Chapter 15

Assembly Language and Related Topics

Assembler

A program that translates assembly language into machine code.

Assembly Language

A symbolic representation of the machine language of a specific processor, augmented by additional types of statements that facilitate program writing and that provide instructions to the assembler.

Compiler

A program that converts another program from some source language (or programming language) to machine language (object code). Some compilers output assembly language which is then converted to machine language by a separate assembler. A compiler is distinguished from an assembler by the fact that each input statement does not, in general, correspond to a single machine instruction or fixed sequence of instructions. A compiler may support such features as automatic allocation of variables, arbitrary arithmetic expressions, control structures such as FOR and WHILE loops, variable scope, input/output operations, higher-order functions and portability of source code.

Executable Code

The machine code generated by a source code language processor such as an assembler or compiler.
This is software in a form that can be run in the computer.

Instruction Set

The collection of all possible instructions for a particular computer; that is, the collection of machine language instructions that a particular processor understands.

Linker

A utility program that combines one or more files containing object code from separately compiled program modules into a single file containing loadable or executable code.

Loader

A program routine that copies an executable program into memory for execution.

Machine Language, or Machine Code

The binary representation of a computer program which is actually read and interpreted by the computer. A program in machine code consists of a sequence of machine instructions (possibly interspersed with data). Instructions are binary strings which may be either all the same size (e.g., one 32-bit word for many modern RISC microprocessors) or of different sizes.

Object Code

The machine language representation of programming source code. Object code is created by a compiler or assembler and is then turned into executable code by the linker.

Table 15.1

Key Terms For This Chapter

(Table can be found on page 508 in the textbook.)

Figure 15.1
Programming the Statement n = *i* + *j*+ *k*

Address	Contents			
	Opcode		Operand	
101	0010	0010	1100	1001
102	0001	0010	1100	1010
103	0001	0010	1100	1011
104	0011	0010	1100	1100
201	0000	0000	0000	0010
202	0000	0000	0000	0011
203	0000	0000	0000	0100
204	0000	0000	0000	0000

(a) Binary program

Address	Contents
101	22C9
102	12CA
103	12CB
104	32CC
201	0002
202	0003
203	0004
204	0000

(b) Hexadecimal program

Address	Instruction	
101	LDA	201
102	ADD	202
103	ADD	203
104	STA	204
201	DAT	0002
202	DAT	0003
203	DAT	0004
204	DAT	0000

(c) Symbolic program

Label	Operation	Operand
FORMUL	LDA	I
	ADD	J
	ADD	K
	STA	N
I	DATA	2
J	DATA	3
K	DATA	4
N	DATA	0

(d) Assembly program

Motivation for Assembly Language Programming

- Assembly language is a programming language that is one step away from machine language
- Typically each assembly language instruction is translated into one machine instruction by the assembler
- Assembly language is hardware dependent, with a different assembly language for each type of processor
- Assembly language instructions can make reference to specific registers in the processor, include all of the opcodes of the processor, and reflect the bit length of the various registers of the processor and operands of the machine language
  - Therefore, an assembly language programmer must understand the computer's architecture

Assembly Language Programming (1 of 2)

Disadvantages

- The disadvantages of using an assembly language rather than an HLL include:
  - Development time
  - Reliability and security
  - Debugging and verifying
  - Maintainability
  - Portability
  - System code can use intrinsic functions instead of assembly
  - Application code can use intrinsic functions or vector classes instead of assembly
  - Compilers have been improved a lot in recent years

Assembly Language Programming (2 of 2)

Advantages

- Advantages to the occasional use of assembly language include:
  - Debugging and verifying
  - Making compilers
  - Embedded systems
  - Hardware drivers and system code
  - Accessing instructions that are not accessible from high-level language
  - Self-modifying code
  - Optimizing code for size
  - Optimizing code for speed
  - Function libraries
  - Making function libraries compatible with multiple compilers and operating systems

Assembly Language vs. Machine Language

- The terms *assembly language* and *machine language* are sometimes, erroneously, used synonymously
- Machine language:
  - Consists of instructions directly executable by the processor
  - Each machine language instruction is a binary string containing an opcode, operand references, and perhaps other bits related to execution, such as flags
  - For convenience, instead of writing an instruction as a bit string, it can be written symbolically, with names for opcodes and registers
- Assembly language:
  - Makes much greater use of symbolic names, including assigning names to specific main memory locations and specific instruction locations
  - Also includes statements that are not directly executable but serve as instructions to the assembler that produces machine code from an assembly language program

Figure 15.2
Assembly-Language Statement Structure

label:	**mnemonic**	**operand(s)**	**;comment**
optional	opcode name or directive name or macro name	zero or more	optional

Pearson

Statements (1 of 3)

Label

- If a label is present, the assembler defines the label as equivalent to the address into which the first byte of the object code generated for that instruction will be loaded
- The programmer may subsequently use the label as an address or as data in another instruction’s address field
- The assembler replaces the label with the assigned value when creating an object program
- Labels are most frequently used in branch instructions
- Reasons for using a label:
  - Makes a program location easier to find and remember
  - Can easily be moved to correct a program
  - Programmer does not have to calculate relative or absolute memory addresses, but just uses labels as needed

Statements (2 of 3)

Mnemonic

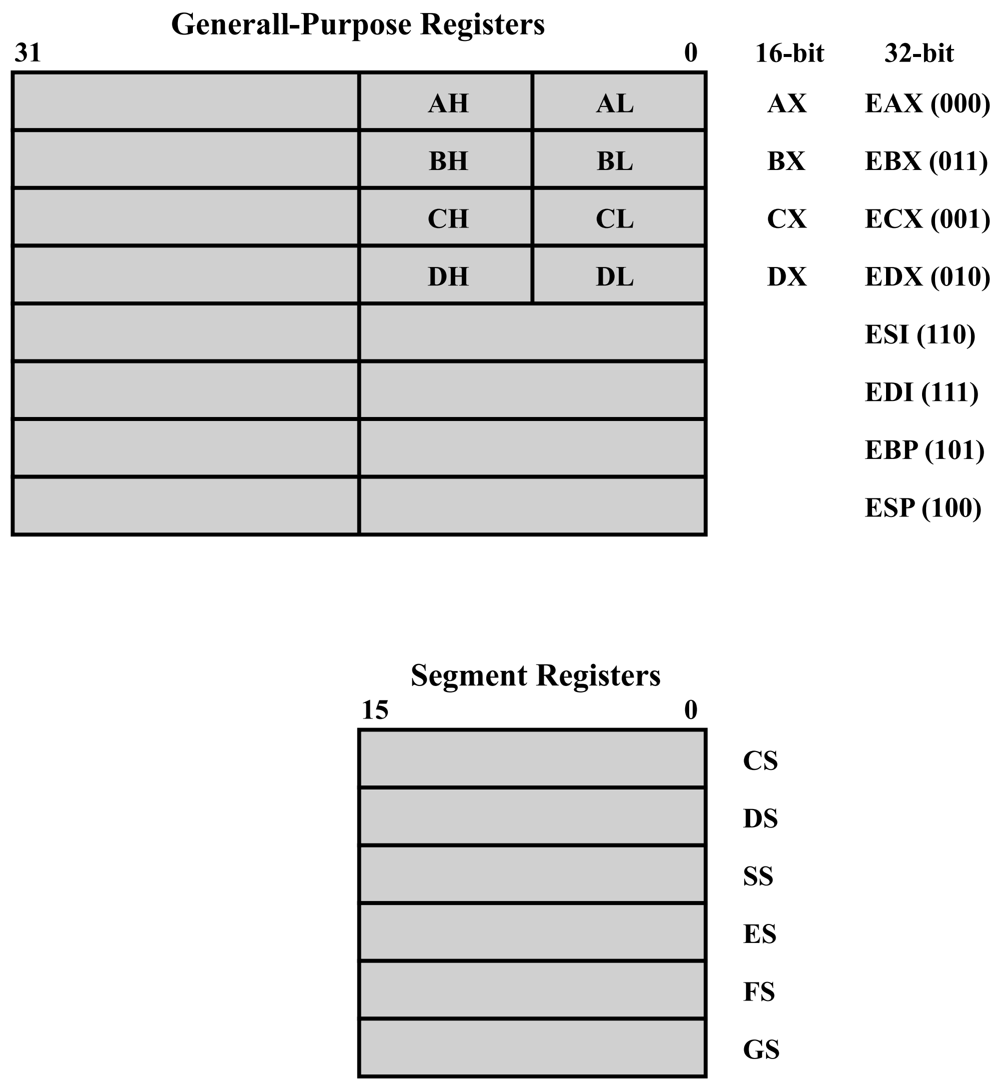
- The mnemonic is the name of the operation or function of the assembly language statement
- In the case of a machine instruction, a mnemonic is the symbolic name associated with a particular opcode

Statements (3 of 3)

Operands

- An assembly language statement includes zero or more operands
- Each operand identifies an immediate value, a register value, or a memory location
- Typically the assembly language provides conventions for distinguishing among the three types of operand references, as well as conventions for indicating addressing mode

Figure 15.3
Intel x86 Program Execution Registers



Statements (1 of 2)

Comment

- All assembly languages allow the placement of comments in the program
- A comment can either occur at the right-hand end of an assembly statement or can occupy and entire test line
- The comment begins with a special character that signals to the assembler that the rest of the line is a comment and is to be ignored by the assembler
- Typically, assembly languages for the x86 architecture use a semicolon (;) for the special character

Statements (2 of 2)

Pseudo-instructions

- Pseudo-instructions are statements which, though not real x86 machine instructions, are used in the instruction field anyway because that's the most convenient place to put them
- Pseudo-instructions are not directly translated into machine language instructions
- Instead, directives are instructions to the assembler to perform specified actions during the assembly process
- Examples include:
  - Define constants
  - Designate areas of memory for data storage
  - Initialize areas of memory
  - Place tables or other fixed data in memory
  - Allow references to other programs

Table 15.2
Some NASM Assembly-Language Directives

(a) Letters for RESx and Dx Directives

Unit	Letter
byte	B
word (2 bytes)	W
double word (4 bytes)	D
quad word (8 bytes)	Q
ten bytes	T

(b) Directives

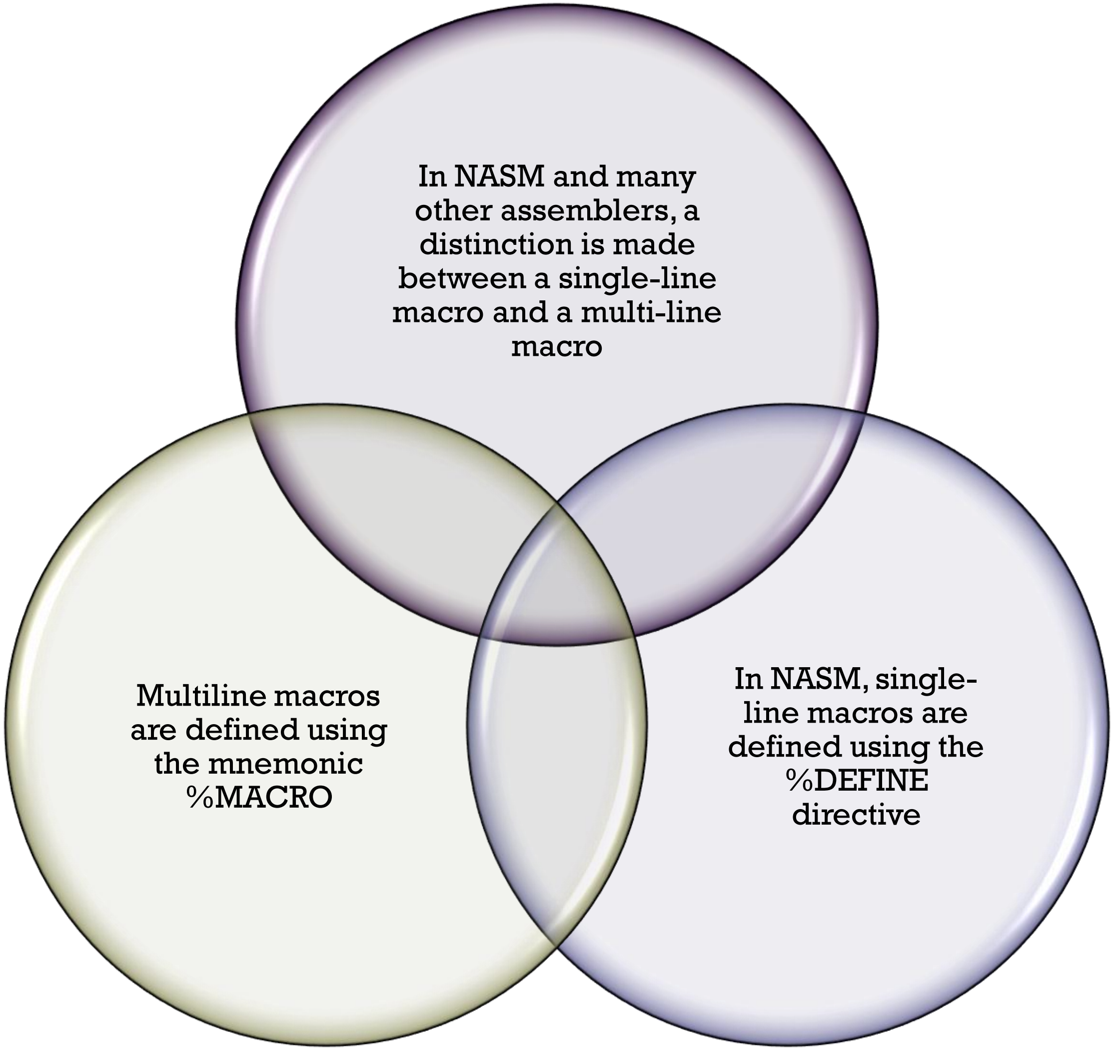
Name	Description	Example
DB, DW, DD, DQ, DT	Initialize locations	`L6 DD 1A92H` `;doubleword at L6 initialized to 1A92H`
RESB, RESW, RESD, RESQ, REST	Reserve uninitialized locations	`BUFFER RESB 64` `;reserve 64 bytes starting at BUFFER`
INCBIN	Include binary file in output	`INCBIN "file.dat" ; include this file`
EQU	Define a symbol to a given constant value	`MSGLEN EQU 25` `;the constant MSGLEN equals decimal 25`
TIMES	Repeat instruction multiple times	`ZEROBUF TIMES 64 DB 0` `;initialize 64-byte buffer to all zeros`

(Table can be found on page 515 in the textbook)

Macro Definitions (1 of 2)

- A macro definition is similar to a subroutine in several ways
  - A subroutine is a section of a program that is written once, and can be used multiple times by calling the subroutine from any point in the program
  - When a program is compiled or assembled, the subroutine is loaded only once
  - A call to the subroutine transfers control to the subroutine and a return instruction in the subroutine returns control to the point of the call
- Similarly, a macro definition is a section of code that the programmer writes once, and then can use many times
  - The main difference is that when the assembler encounters a macro call, it replaces the macro call with the macro itself
  - This process is call *macro expansion*
- Macros are handled by the assembler at assembly time
- Macros provide the same advantage as subroutines in terms of modular programming, but without the runtime overhead of a subroutine call and return
  - The tradeoff is that the macro approach uses more space in the object code

Macro Definitions (2 of 2)



Directives

- A directive is a command embedded in the assembly source code that is recognized and acted upon by the assembler
- NASM includes the following directives:
- **BITS**
  - Specifies whether NASM should generate code designed to run on a processor operating in 16-bit mode, 32-bit mode, or 64-bit mode
- **DEFAULT**
  - Can change some assembler defaults, such as whether to use relative or absolute addressing
- **SECTION or SEGMENT**
  - Changes that section of the output file the source code will be assembled into
- **EXTERN**
  - Used to declare a symbol which is not defined anywhere in the module being assembled, but is assumed to be defined in some other module and needs to be referred to by this one
- **GLOBAL**
  - Is the other end of EXTERN: if one module declares a symbol as EXTERN and refers to it, then in order to prevent linker errors, some other module must actually define the symbol and declare it as GLOBAL
- **COMMON**
  - Used to declare common variables
- **CPU**
  - Restricts assembly to those instructions that are available on the specified CPU
- **FLOAT**
  - Allows the programmer to change some of the default settings to options other than those used in IEEE 754
- **[WARNING]**
  - Used to enable or disable classes of warnings

System Calls

- The assembler makes use of the x86 INT instruction to make system calls
- There are six registers that store the arguments of the system call used
  - EBX
  - ECX
  - EDX
  - ESI
  - EDI
  - EDP
- These registers take the consecutive arguments, starting with the EBX register
- If there are more than six arguments, then the memory location of the first argument is stored in the EBX register

Assembly Programs – "Hello World"

```
;nasm 2.13.02
global _start           ; tells the linker entry point
section .data
   txt: db 'Hello world!', 10 ; initialize a 'Hello world!' and a
                              ; newline character "10" in address `txt`
   txt_len: equ $ - txt ; $ means the current address
                        ;`txt` is the length of the 'Hello world!' string

section .text
_start:
   mov eax, 4         ; The system call for write (sys_write)
   mov ebx, 1         ; File descriptor 1 - standard output
   mov ecx, txt       ; pointer of the string in ecx
   mov edx, txt_len   ; txt_len is the length of the string
   int 0x80           ; interrupt code 80 for system calls

   mov eax, 1         ; The system call for exit (sys_exit)
   mov ebx, 0         ; Exit with return code of 0 (no error)
   int 0x80;
```

Assembly Programs – Sum of [1 – 6]

```
;nasm 2.13.02
global _start           ; tells the linker entry point

section .text
_start:
   mov ebx, 0        ; initialize ebx with zero
   mov ecx, 6        ; the last number to add in the sequence

L:
   add  ebx, ecx     ; ebx = ebx + ecx
   dec  ecx          ; ecx -= 1
   cmp  ecx, 0       ; compare ecx & 0
   jg   L            ; jump back to L, if ecx > 0

   mov  eax, 1       ; The system call for exit (sys_exit)
   int  0x80         ; interrupt with ebx exit status = result
```

Figure 15.4 Assembly Programs for Greatest Common Divisor

(a) Compiled program	(b) Written directly in assembly language

<pre>gcd: mov   ebx,eax
       mov   eax,edx
       test   ebx,ebx
       jne   L1
       test   edx,edx
       jne   L1
       mov   eax,1
       ret
L1:   test   eax,eax
       jne   L2
       mov   eax,ebx
       ret
L2:   test   ebx,ebx
       je   L5
L3;   cmp   ebx,eax
       je   L5
       jae   L4
       sub   eax,ebx
       jmp   L3
L4:   sub   ebx,eax
       jmp   L3
L5:   ret</pre>
	<pre>gcd: neg eax
	       je L3
	L1: neg eax
	       xchg eax,edx
	L2: sub eax,edx
	       jg L2
	       jne L1
	L3: add eax,edx
	       jne L4
	       inc eax
	L4: ret</pre>

Copyright