Address Space

- In 32-bit protected mode, a task or program can address a linear address space of up to 4 GBytes
  - Extended Physical Addressing allows a total of 64 GBytes of physical memory to be addressed
- Real-address mode programs, on the other hand, can only address a range of 1 MByte
- If the processor is in protected mode and running multiple programs in virtual-8086 mode, each program has its own 1-MByte memory area

Real Address Mode

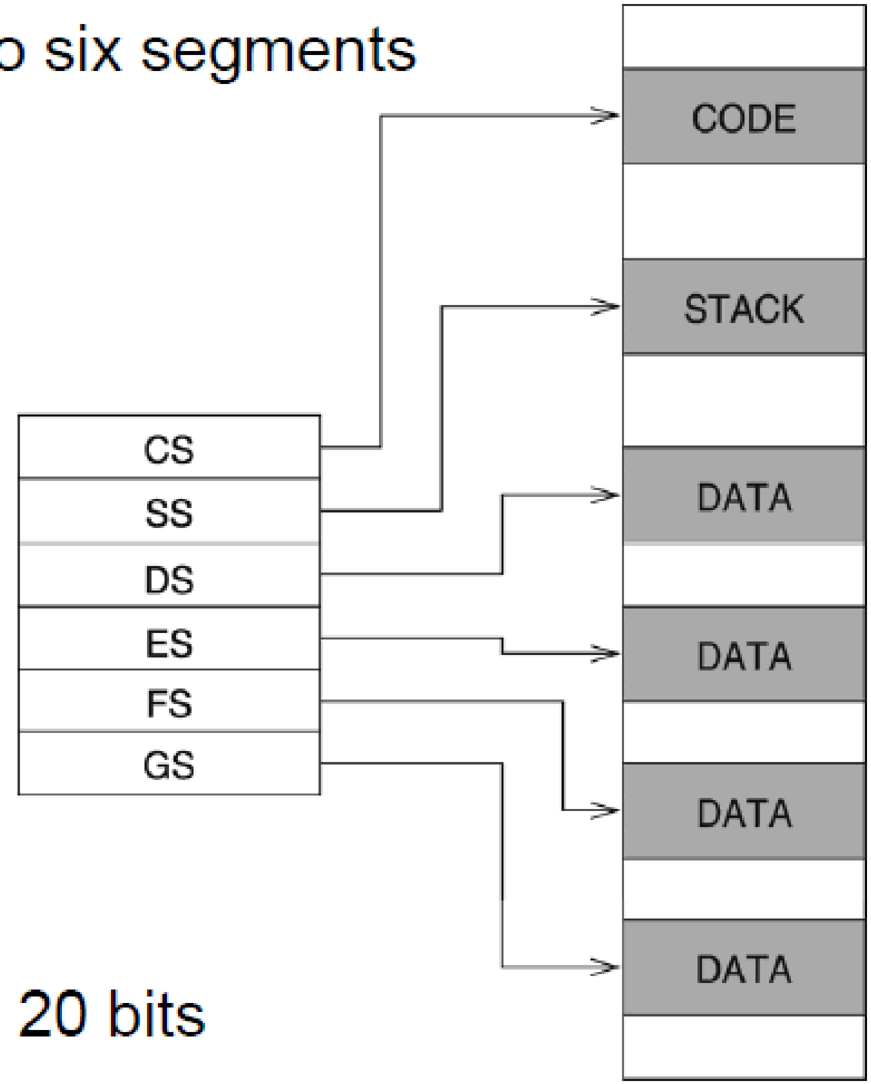
Program Segments and Segment Registers: Segment registers are used to hold base addresses for the program code, data and stack.

A program can access up to six Segments:

- The code segment store code of the program. The code segment register (CS) holds the base address for all executable instructions in the program
- The data segment holds the base address for variables. This segment stores data for the program
- The extra segment is an extra data segment (often used for shared data)
- The stack segment holds the base address for the stack. The segment store subroutine return addresses, local variables, parameters and interrupts

Real Address Mode (1)

- A program can access up to six segments at any time
  - Code segment
  - Stack segment
  - Data segment
  - Extra segments (up to 3)
- Each segment is 64 KB
- Logical address
  - Segment = 16 bits
  - Offset = 16 bits
- Linear (physical) address = 20 bits



Slide Credit: Dr. Aiman El-Maleh & Dr. Kip Irvine

Real Address Mode (2)

Linear address = Segment × 10 (hex) + Offset

Example:

segment = A1F0 (hex)

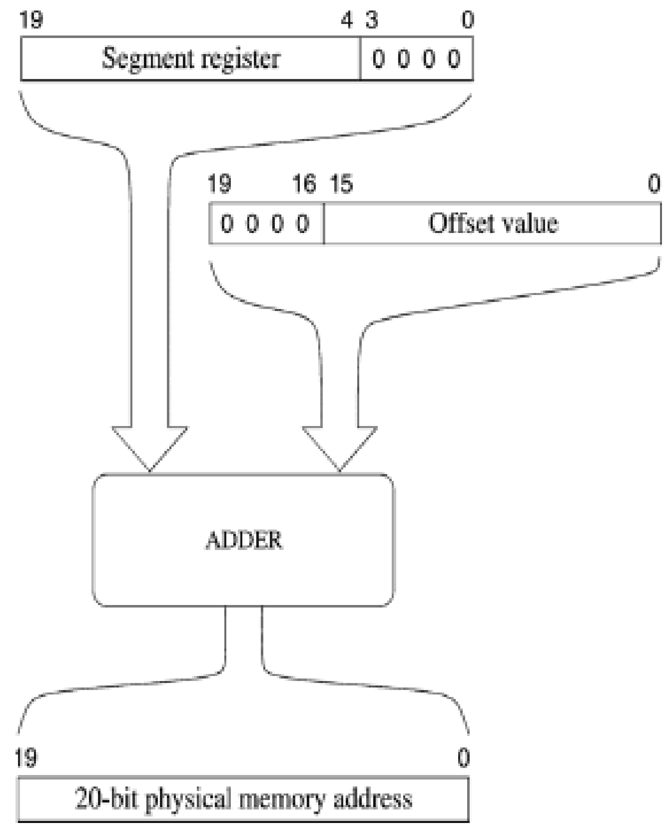
offset = 04C0 (hex)

logical address = A1F0:04C0 (hex)

what is the linear address?

Solution:

```
  A1F00   (add 0 to segment in hex)
+  04C0   (offset in hex)
---------
  A23C0   (20-bit linear address in hex)
```



Slide Credit: Dr. Aiman El-Maleh & Dr. Kip Irvine

Real Address Mode (3)

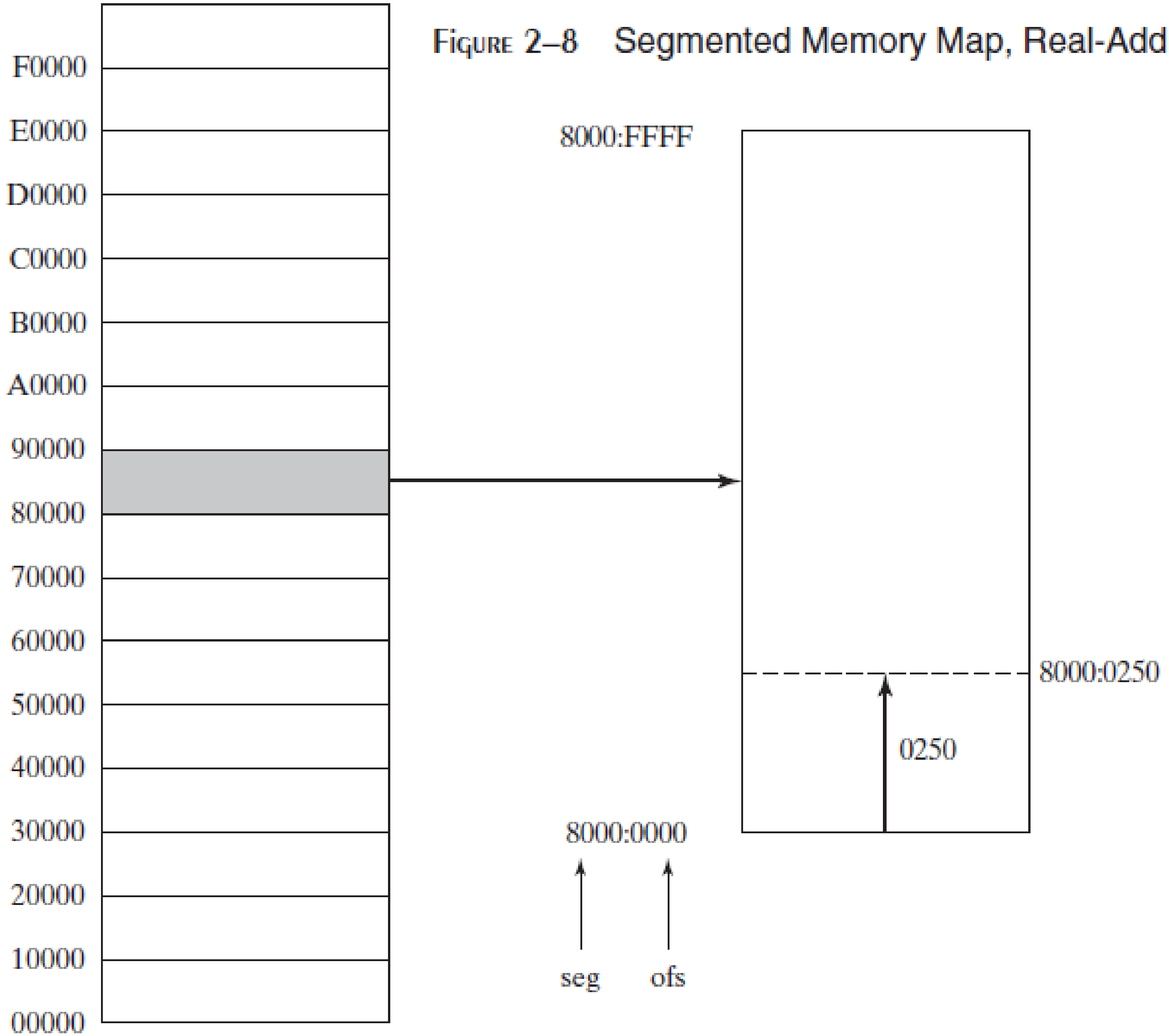
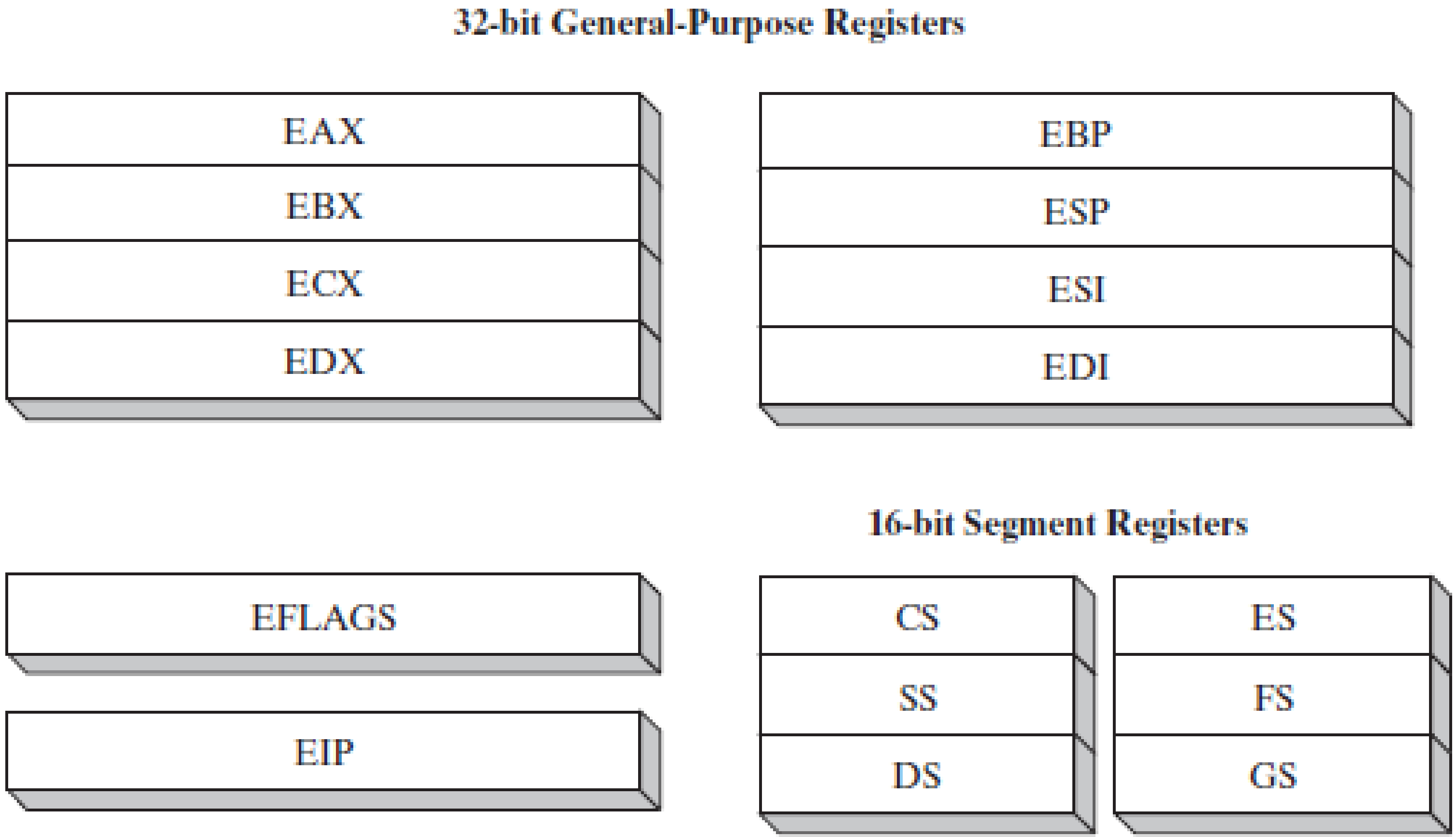


Figure 2–8 Segmented Memory Map, Real-Address Mode.

Basic Program Execution Registers

- *Registers* are high-speed storage locations directly inside the CPU, designed to be accessed at much higher speed than conventional memory
- There are:
  - eight general-purpose registers
  - six segment registers
  - a processor status flags register (EFLAGS), and
  - an instruction pointer (EIP)

Figure 2–5 Basic Program Execution Registers.

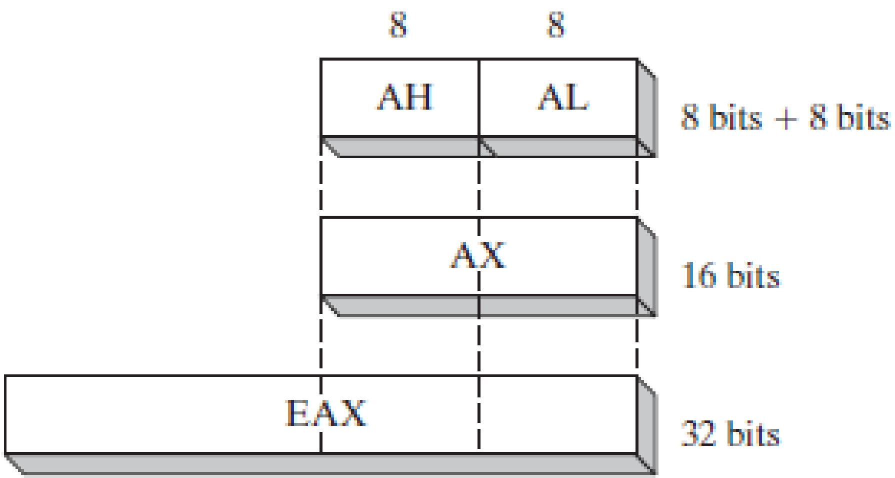


General-Purpose Registers

The general-purpose registers are primarily used for arithmetic and data movement.

- As shown in Figure 2–6, the lower 16 bits of the EAX register can be referenced by the name AX
- Portions of some registers can be addressed as 8-bit values
  - For example, the AX register, has an 8-bit upper half named AH and an 8-bit lower half named AL

Figure 2–6 General-Purpose Registers.



32-Bit	16-Bit	8-Bit (High)	8-Bit (Low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

The remaining general-purpose registers can only be accessed using 32-bit or 16-bit names, as shown in the following table:

32-Bit	16-Bit
ESI	SI
EDI	DI
EBP	BP
ESP	SP

Specialized Uses of general-purpose registers

Some general-purpose registers have specialized uses:

- EAX (*accumulator*) - favored for arithmetic operations.
  - It is automatically used by multiplication and division instructions
- EBX (Base) - Holds base address for procedures and variables
- ECX - The CPU automatically uses ECX as a counter for looping operations
- EDX (Data) - Used in multiplication and division operations

Default Operands:

Dividend	Divisor	Quotient	Remainder
AX	*r/m8*	AL	AH
DX:AX	*r/m16*	AX	DX
EDX:EAX	*r/m32*	EAX	EDX

Index Registers

Index Registers contain the offsets for data and instructions.
Offset- distance (in bytes) from the base address of the segment.

- ESP (*extended stack pointer* register) contains the offset for the top of the stack to addresses data on the stack (a system memory structure)

- ESI and EDI (*extended source index* and *extended destination index)* points to the source and destination string respectively in the string move instructions

- EBP is used to reference function parameters and local variables on the stack

Instruction Pointer Register

- The EIP, or *instruction pointer*, register contains the address of the next instruction to be executed.
- Certain machine instructions manipulate EIP, causing the program to branch to a new location.

EFLAGS Register:

The EFLAGS register consists of individual binary bits that control the operation of the CPU or reflect the outcome of some CPU operation.

- A flag is *set* when it equals 1; it is *clear* (or reset) when it equals 0.

- Programs can set individual bits in the EFLAGS register to control the CPU's operation

- For example: Interrupt when arithmetic overflow is detected

x	x	x	x	O	D	I	T	S	Z	x	A	x	P	x	C

O = Overflow

D = Direction

I = Interrupt

T = Trap

x = undefined

S = Sign

Z = Zero

A = Auxiliary Carry

P = Parity

C = Carry

Flags

There are two types of flags: control flags (which determine how instructions are carried out) and status flags (which report on the results of operations.

Control flags include:

- ***Direction*** Flag (DF) - affects the direction of block data transfers (like long character string). 1 = up; 0 - down.
- ***Interrupt*** Flag (IF) - determines whether interrupts can occur (whether hardware devices like the keyboard, disk drives, and system clock can get the CPU's attention to get their needs attended to.
- ***Trap*** Flag (TF) - determines whether the CPU is halted after every instruction. Used for debugging purposes.

Status Flags

The Status flags reflect the outcomes of arithmetic and logical operations performed by the CPU.

- Status Flags include:
  - *Carry* Flag (CF) - set when the result of **unsigned** arithmetic is too large to fit in the destination. 1 = carry; 0 = no carry.
  - *Overflow* Flag (OF) - set when the result of **signed** arithmetic is too large to fit in the destination. 1 = overflow; 0 = no overflow.
  - *Sign* Flag (SF) - set when an arithmetic or logical operation generates a negative result. 1 = negative; 0 = positive.
  - *Zero* Flag (ZF) - set when an arithmetic or logical operation generates a result of zero. Used primarily in jump and loop operations. 1 =zero; 0 = not zero.
  - *Auxiliary Carry* Flag - set when an operation causes a carry from bit 3 to 4 or borrow (frombit 4 to 3). 1 = carry, 0 = no carry.
  - ***Parity-*** is set if the least-significant byte in the result contains an even number of 1 bits. It used to verify memory integrity.

Integer Constants

An *integer constant* (or integer literal) is made up of an optional leading sign, one or more digits, and an optional suffix character (called a *radix*) indicating the number's base:

```
[{+|−}] digits [radix]
```

Radix may be one of the following (uppercase or lowercase):

h	Hexadecimal	r	Encoded real
q/o	Octal	t	Decimal *(alternate)*
d	Decimal	y	Binary *(alternate)*
b	Binary		

If no radix is given, the integer constant is assumed to be decimal. Here are some examples using different radixes:

26	Decimal	42o	Octal
26d	Decimal	1Ah	Hexadecimal
11010011b	Binary	0A3h	Hexadecimal
42q	Octal		

Integer Expressions

- An *integer expression* is a mathematical expression involving integer values and arithmetic operators
- The integer expression must evaluate to an integer, which can be stored in 32 bits (0 through FFFFFFFFh)

Table 3-1 Arithmetic Operators.

Operator	Name	Precedence Level
( )	Parentheses	1
+, −	Unary plus, minus	2
*, /	Multiply, divide	3
MOD	Modulus	3
+, −	Add, subtract	4

Precedence refers to the implied order of operations when an expression contains two or more operators. The order of operations is shown for the following expressions:

```
4 + 5 * 2            Multiply, add
12 - 1 MOD 5         Modulus, subtract
-5 + 2               Unary minus, add
(4 + 2) * 6          Add, multiply
```

The following are examples of valid expressions and their values:

Expression	Value
16 / 5	3
$-(3 + 4) * (6 - 1)$	-35
$-3 + 4 * 6 - 1$	20
25 mod 3	1

3.1.3 Real Number Constants

Real number constants are represented as decimal reals or encoded (hexadecimal) reals. A *decimal real* contains an optional sign followed by an integer, a decimal point, an optional integer that expresses a fraction, and an optional exponent:

```
[sign]integer.[integer][exponent]
```

Following are the syntax for the sign and exponent:

```
sign        {+,-}
exponent    E[{+,-}]integer
```

Following are examples of valid real number constants:

```
2.
+3.0
-44.2E+05
26.E5
```

At least one digit and a decimal point are required.

```
rVal1       REAL4 -1.2
rVal2       REAL8  3.2E-260
rVal3       REAL10 4.6E+4096
ShortArray REAL4  20 DUP(0.0)
```

Table 3-4 Standard Real Number Types.

Data Type	Significant Digits	Approximate Range
Short real	6	1.18×10^{-38} to 3.40×10^{38}
Long real	15	2.23×10^{-308} to 1.79×10^{308}
Extended-precision real	19	3.37×10^{-4932} to 1.18×10^{4932}

3.1.4 Character Constants

A *character constant* is a single character enclosed in single or double quotes. MASM stores the value in memory as the character's binary ASCII code. Examples are

```
'A'
"d"
```

3.1.5 String Constants

A *string constant* is a sequence of characters (including spaces) enclosed in single or double quotes:

```
'ABC'
'X'
"Good night, Gracie"
'4096'
```

Embedded quotes are permitted when used in the manner shown by the following examples:

```
"This isn't a test"
'Say "Good night," Gracie'
```

3.1.6 Reserved Words

Reserved words have special meaning in MASM and can only be used in their correct context. There are different types of reserved words:

- Instruction mnemonics, such as MOV, ADD, and MUL
- Register names
- Directives, which tell MASM how to assemble programs
- Attributes, which provide size and usage information for variables and operands. Examples are BYTE and WORD
- Operators, used in constant expressions
- Predefined symbols, such as @data, which return constant integer values at assembly time

3.1.7 Identifiers

An *identifier* is a programmer-chosen name. It might identify a variable, a constant, a procedure, or a code label. Keep the following in mind when creating identifiers:

- They may contain between 1 and 247 characters.
- They are not case sensitive.
- The first character must be a letter (A..Z, a..z), underscore (_), @ , ?, or $. Subsequent characters may also be digits.
- An identifier cannot be the same as an assembler reserved word.

The @ symbol is used extensively by the assembler as a prefix for predefined symbols, so avoid it in your own identifiers. Make identifier names descriptive and easy to understand. Here are some valid identifiers:

```
var1        Count       $first
_main       MAX         open_file
myFile      xVal        _12345
```

3.1.8 Directives

A *directive* is a command embedded in the source code that is recognized and acted upon by the assembler. Directives do not execute at runtime. Directives can define variables, macros, and procedures. They can assign names to memory segments and perform many other housekeeping tasks related to the assembler. In MASM, directives are case insensitive. For example, it recognizes **.data**, **.DATA**, and **.Data** as equivalent.

The following example helps to show the difference between directives and instructions. The DWORD directive tells the assembler to reserve space in the program for a doubleword variable. The MOV instruction, on the other hand, executes at runtime, copying the contents of **myVar** to the EAX register:

```
myVar  DWORD 26                              ; DWORD directive
mov    eax,myVar                             ; MOV instruction
```

Defining Segments One important function of assembler directives is to define program sections, or *segments*. The .DATA directive identifies the area of a program containing variables:

```
.data
```

The .CODE directive identifies the area of a program containing executable instructions:

```
.code
```

The .STACK directive identifies the area of a program holding the runtime stack, setting its size:

```
.stack 100h
```

Directives

- Commands that are recognized and acted upon by the assembler
  - Not part of the Intel instruction set
  - Used to declare code, data areas, select memory model, declare procedures, etc.
  - not case sensitive
- Different assemblers have different directives
  - NASM not the same as MASM, for example

```
myVar   DWORD 26            ; DWORD directive, set aside
                            ; enough space for double word
Mov     eax, myVar          ; MOV instruction
```

3.1.9 Instructions

An *instruction* is a statement that becomes executable when a program is assembled. Instructions are translated by the assembler into machine language bytes, which are loaded and executed by the CPU at runtime. An instruction contains four basic parts:

- Label (optional)
- Instruction mnemonic (required)
- Operand(s) (usually required)
- Comment (optional)

This is the basic syntax:

```
[label:] mnemonic [operands] [;comment]
```

We use the Intel IA-32 instruction set

Label

A *label* is an identifier that acts as a place marker for instructions and data. A label placed just before an instruction implies the instruction's address. Similarly, a label placed just before a variable implies the variable's address.

Data Labels A data label identifies the location of a variable, providing a convenient way to reference the variable in code. The following, for example, defines a variable named count:

```
count  DWORD 100
```

The assembler assigns a numeric address to each label. It is possible to define multiple data items following a label. In the following example, array defines the location of the first number (1024). The other numbers following in memory immediately afterward:

```
array DWORD 1024, 2048
      DWORD 4096, 8192
```

Code Labels A label in the code area of a program (where instructions are located) must end with a colon (:) character. Code labels are used as targets of jumping and looping instructions. For example, the following JMP (jump) instruction transfers control to the location marked by the label named **target**, creating a loop:

```
target:
    mov   ax,bx
    ...
    jmp   target
```

3.1.10 The NOP (No Operation) Instruction

The safest (and the most useless) instruction you can write is called NOP (no operation). It takes up 1 byte of program storage and doesn't do any work. It is sometimes used by compilers and assemblers to align code to even-address boundaries. In the following example, the first MOV instruction generates three machine code bytes. The NOP instruction aligns the address of the third instruction to a doubleword boundary (even multiple of 4):

```
00000000  66 8B C3    mov ax,bx
00000003  90          nop                  ; align next instruction
00000004  8B D1       mov edx,ecx
```

x86 processors are designed to load code and data more quickly from even doubleword addresses.

Labels

- Act as place markers
  - marks the address (offset) of code and data
- Follow identifer rules
- Data label
  - must be unique
  - example: **myArray** (not followed by colon)
  - count DWORD 100
- Code label
  - target of jump and loop instructions
  - example: **L1:** (followed by colon)

Instruction Format Examples

- No operands
  - stc ; set Carry flag
- One operand
  - inc eax ; register
  - inc myByte ; memory
- Two operands
  - add ebx, ecx ; register, register
  - sub myByte, 25 ; memory, constant
  - add eax, 36 * 25 ; register, constant-expression

```
TITLE Add and Subtract              (AddSub.asm)

; This program adds and subtracts 32-bit integers.

INCLUDE Irvine32.inc

 .code
 main PROC

     mov   eax,10000h           ; EAX = 10000h
     add   eax,40000h           ; EAX = 50000h
     sub   eax,20000h           ; EAX = 30000h
     call  DumpRegs             ; display registers

     exit
 main ENDP
 END main
```

```
TITLE Add and Subtract              (AddSub.asm)
```

The TITLE directive marks the entire line as a comment. You can put anything you want on this line.

```
; This program adds and subtracts 32-bit integers.
```

All text to the right of a semicolon is ignored by the assembler, so we use it for comments.

```
INCLUDE Irvine32.inc
```

The INCLUDE directive copies necessary definitions and setup information from a text file named *Irvine32.inc*, located in the assembler's INCLUDE directory. (The file is described in Chapter 5.)

```
INCLUDE IRVINE32.INC
.DATA
X WORD 20
Y WORD 10
Z WORD 0H
.CODE
MAIN PROC
```

```
MOV AX, X
ADD AX, Y
MOV Z, AX
CALL DUMPREGS
EXIT
MAIN ENDP
END MAIN
```

```
.code
```

The .code directive marks the beginning of the *code segment*, where all executable statements in a program are located.

```
main PROC
```

The PROC directive identifies the beginning of a procedure. The name chosen for the only procedure in our program is **main**.

```
mov   eax,10000h                      ; EAX = 10000h
```

The MOV instruction moves (copies) the integer 10000h to the EAX register. The first operand (EAX) is called the *destination operand*, and the second operand is called the *source operand*. The comment on the right side shows the expected new value in the EAX register.

```
add   eax,40000h                      ; EAX = 50000h
```

The ADD instruction adds 40000h to the EAX register. The comment shows the expected new value in EAX.

```
sub   eax,20000h                      ; EAX = 30000h
```

The SUB instruction subtracts 20000h from the EAX register.

```
call  DumpRegs                        ; display registers
```

The CALL statement calls a procedure that displays the current values of the CPU registers. This can be a useful way to verify that a program is working correctly.

```
        exit
    main ENDP
```

The **exit** statement (indirectly) calls a predefined MS-Windows function that halts the program. The ENDP directive marks the end of the **main** procedure. Note that **exit** is not a MASM keyword; instead, it's a macro command defined in the *Irvine32.inc* include file that provides a simple way to end a program.

```
    END main
```

The END directive marks the last line of the program to be assembled. It identifies the name of the program's *startup* procedure (the procedure that starts the program execution).

Program Output The following is a snapshot of the program's output, generated by the call to DumpRegs:

```
EAX=00030000  EBX=7FFDF000  ECX=00000101  EDX=FFFFFFFF
ESI=00000000  EDI=00000000  EBP=0012FFF0  ESP=0012FFC4
EIP=00401024  EFL=00000206  CF=0  SF=0  ZF=0  OF=0  AF=0  PF=1
```

3.4.1 Intrinsic Data Types

MASM defines *intrinsic data types*, each of which describes a set of values that can be assigned to variables and expressions of the given type. The essential characteristic of each type is its size in bits: 8, 16, 32, 48, 64, and 80. Other characteristics (such as signed, pointer, or floating-point) are optional and are mainly for the benefit of programmers who want to be reminded about the type of data held in the variable. A variable declared as DWORD, for example, logically holds an unsigned 32-bit integer. In fact, it could hold a signed 32-bit integer, a 32-bit single precision real, or a 32-bit pointer. The assembler is not case sensitive, so a directive such as DWORD can be written as **dword**, **Dword**, **dWord**, and so on.

Table 3-2 Intrinsic Data Types.

Type	Usage
BYTE	8-bit unsigned integer. B stands for byte
SBYTE	8-bit signed integer. S stands for signed
WORD	16-bit unsigned integer (can also be a Near pointer in real-address mode)
SWORD	16-bit signed integer
DWORD	32-bit unsigned integer (can also be a Near pointer in protected mode). D stands for double
SDWORD	32-bit signed integer. SD stands for signed double
FWORD	48-bit integer (Far pointer in protected mode)
QWORD	64-bit integer. Q stands for quad
TBYTE	80-bit (10-byte) integer. T stands for Ten-byte
REAL4	32-bit (4-byte) IEEE short real
REAL8	64-bit (8-byte) IEEE long real
REAL10	80-bit (10-byte) IEEE extended real

```
.DATA
PROMPTUSE BYTE "ENTER TWO
INTEGERS: ", 0
RESULTS BYTE "RESULT IS: ", 0
.CODE
 MAIN PROC
 MOV ESI, OFFSET ARRAY
 MOV ECX, INT_COUNT
L1:
 MOV EDX, OFFSET PROMPTUSER
 CALL WRITESTRING
 CALL READINT
 MOV [ESI],EAX
 ADD EAX, [ESI]
 ADD ESI, TYPE DWORD
 LOOP L1
```

```
 MOV EDX, OFFSET
RESULTS
 CALL WRITESTRING
CALL WRITEINT
EXIT
MAIN ENDP
END MAIN
```