COAL Chapter 1,2,3

Week 2 and 3

Nouman M Durrani

Assembly Language

Assembly language consists of statements written with short mnemonics such as ADD, MOV, SUB, and CALL.

Assembly language has a *one-to-one* relationship with machine language:

- Each assembly language instruction corresponds to a single machine-language instruction.

- High-level languages such as C++ and Java have a *one-to-many* relationship with assembly language and machine language.

- A single statement in C++ expands into multiple assembly language or machine instructions.

- The following C++ code carries out two arithmetic operations and assigns the result to a variable. Assume X and Y are integers:

```
int Y;
int X = (Y + 4) * 3;
```

- Following is the equivalent translation to assembly language.

- The translation requires multiple statements because assembly language works at a detailed level:

```
mov eax,Y ; move Y to the EAX register
add eax,4 ; add 4 to the EAX register
mov ebx,3 ; move 3 to the EBX register
imul ebx ; multiply EAX by EBX
mov X,eax ; move EAX to X
```

Registers are named storage locations in the CPU that hold intermediate results of operations

What Are Assemblers and Linkers?

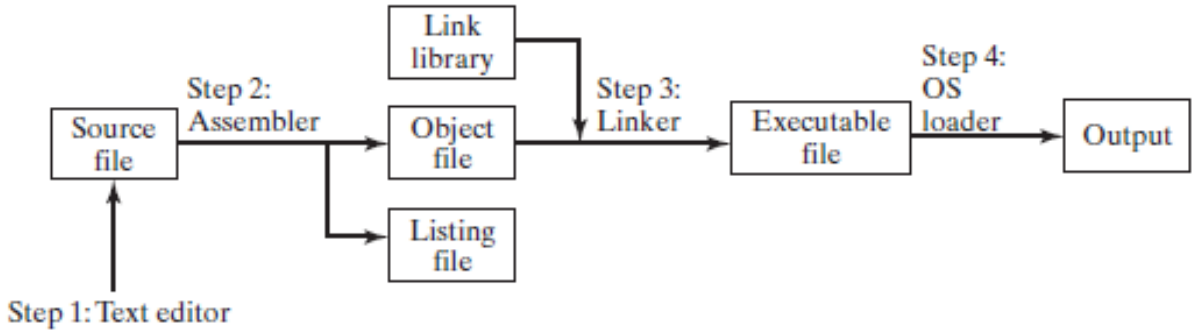
- **Assembler** is a utility program that converts source code programs from assembly language into an object file, a machine language translation of the program. Optionally a Listing file is also produced.

- The **linker** reads the object file and checks to see if the program contains any calls to procedures in a link library. The **linker** copies any required procedures from the link library, combines them with the object file, and produces the *executable file*.

- **OS Loader:** A program that loads executable files into memory, and branches the CPU to the program's starting address, (may initialize some registers (e.g. IP)) and the program begins to execute.

- **Debugger** is a utility program, that lets you step through a program while it's running and examine registers and memory

FIGURE 3–7 Assemble-Link-Execute cycle.



Listing File

- A *listing file* contains:
  - a copy of the program's source code,
  - with line numbers,
  - the numeric address of each instruction,
  - the machine code bytes of each instruction (in hexadecimal), and
  - a symbol table.

  The symbol table contains the names of all program identifiers, segments, and related information.

FIGURE 3–8 Excerpt from the AddTwo source listing file.

```
 1:    ; AddTwo.asm - adds two 32-bit integers.
 2:    ; Chapter 3 example
 3:
 4:    .386
 5:    .model flat,stdcall
 6:    .stack 4096
 7:    ExitProcess PROTO,dwExitCode:DWORD
 8:
 9:     00000000                        .code
10:     00000000                        main PROC
11:     00000000  B8 00000005              mov  eax,5
12:     00000005  83 C0 06                 add  eax,6
13:
14:                                        invoke ExitProcess,0
15:     00000008  6A 00                    push   +000000000h
16:     0000000A  E8 00000000 E            call   ExitProcess
17:     0000000F                        main ENDP
18:                                     END main
```

Assembly Language for x86 Processors

- *Assembly Language for x86 Processors* focuses on programming microprocessors compatible with the Intel IA-32 and AMD x86 processors running under Microsoft Windows.

- Assembly language bears the closest resemblance to native machine language.

- It provides direct access to computer hardware, requiring you to understand much about your computer's architecture and operating system.

Is Assembly Language Portable?

- A language whose source programs can be compiled and run on a wide variety of computer systems is said to be *portable.*

- A C++ program, for example, should compile and run on just about any computer, unless it makes specific references to library functions that exist under a single operating system.

- A major feature of the Java language is that compiled programs run on nearly any computer system.

- Assembly language is not portable because it is designed for a specific processor family.

- There are a number of different assembly languages widely used today, each based on a processor family.

  - Some well-known processor families are Motorola 68x00, x86, SUN Sparc, Vax, and IBM-370.

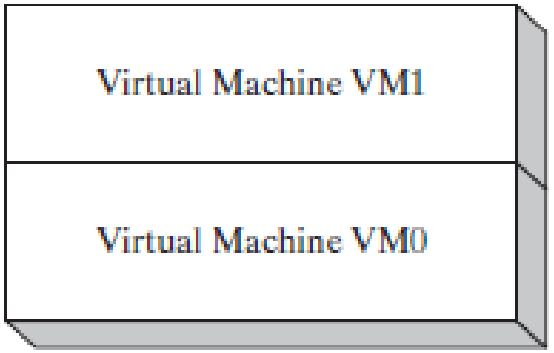
- The instructions in assembly language may directly match the computer's architecture or they may be translated during execution by a program inside the processor known as a *microcode interpreter*.

Virtual Machine Concept

- Virtual Machines:
  - An effective way to explain how a computer's hardware and software are related is called the *virtual machine concept.*
- Specific Machine Levels

Virtual Machines

- Virtual machine as a software program that emulates the functions of some other physical or virtual computer.
- Programming Language analogy:
  - Each computer has a native machine language (language L0) that runs directly on its hardware
  - A more human-friendly language is usually constructed above machine language, called Language L1
- The virtual machine **VM1**, can execute commands written in language L1.
- The virtual machine **VM0** can execute commands written in language L0



- Programs written in L1 can run two different ways:
  - Interpretation – L0 program interprets and executes L1 instructions one by one
  - Translation – L1 program is completely translated into an L0 program, which then runs on the computer hardware

Translating Languages

English: Display the sum of A times B plus C.

C++: cout << (A * B + C);

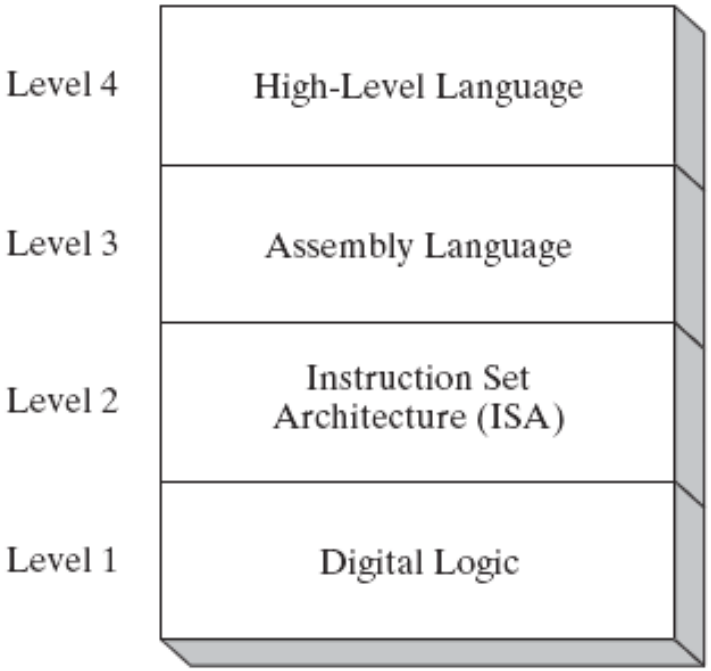
Assembly Language:

```
mov eax,A
mul B
add eax,C
call WriteInt
```


Intel Machine Language:

```
A1 00000000
F7 25 00000004
03 05 00000008
E8 00500000
```

Specific Machine Levels



(descriptions of individual levels follow . . .)

High-Level Language

- Level 4
- Application-oriented languages
  - C++, Java, Pascal, Visual Basic . . .
- Programs compile into assembly language (Level 3)

- The Java programming language is based on the virtual machine concept.
- A program written in the Java language is translated by a Java compiler into *Java byte code* - a low-level language quickly executed at runtime by a program known as a *Java virtual machine (JVM)*.
- The JVM has been implemented on many different computer systems, making Java programs relatively system independent.

Assembly Language

- Level 3
- Instruction mnemonics that have a one-to-one correspondence to machine language
- Programs are translated into Instruction Set Architecture Level - machine language (Level 2)

- The instructions in assembly language may directly match the computer's architecture or they may be translated during execution by a program inside the processor known as a *microcode interpreter*.

Instruction Set Architecture (ISA)

- Level 2
- Also known as conventional machine language
- Executed by Level 1 (Digital Logic)

Digital Logic

- Level 1
- CPU, constructed from digital logic gates
- System bus
- Memory
- Implemented using bipolar transistors

next: Data Representation

Basic Microcomputer Design

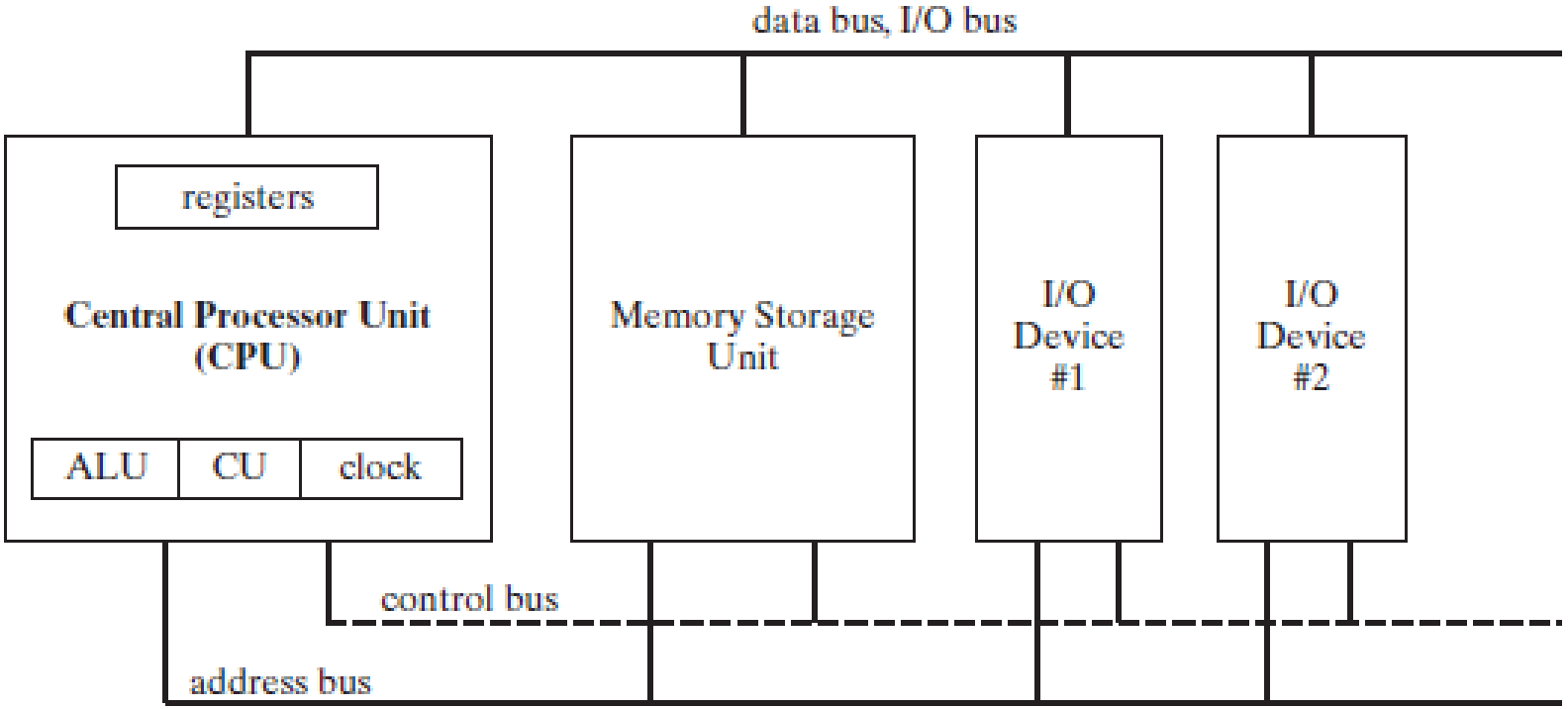
- The *central processor unit* (CPU), where calculations and logic operations take place, contains a limited number of storage locations named *registers*, a high-frequency clock, a control unit, and an arithmetic logic unit.

- The *memory storage unit* is where instructions and data are held while a computer program is running.

- The storage unit receives requests for data from the CPU, transfers data from random access memory (RAM) to the CPU, and transfers data from the CPU into memory.

- All processing of data takes place within the CPU, so programs residing in memory must be copied into the CPU before they can execute.

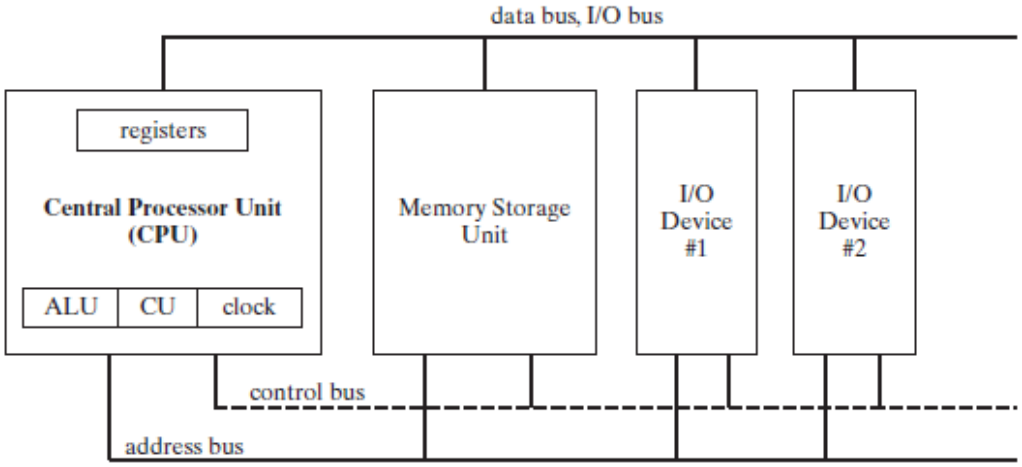
FIGURE 2–1 Block Diagram of a Microcomputer.



BUSES

- A *bus* is a group of parallel wires that transfer data from one part of the computer to another.
- A computer system usually contains four bus types: data, I/O, control, and address.
- The *data bus* transfers instructions and data between the CPU and memory.
- The I/O bus transfers data between the CPU and the system input/output devices.
- The *control bus* uses binary signals to synchronize actions of all devices attached to the system bus.
- The *address bus* holds the addresses of instructions and data when the currently executing instruction transfers data between the CPU and memory.

FIGURE 2–1 Block Diagram of a Microcomputer.



Clock Cycles

- A machine instruction requires at least one clock cycle to execute.
  - Few require in excess of 50 clocks (the multiply instruction on the 8088 processor, for example).
- Instructions requiring memory access often have empty clock cycles called *wait states.*
  - Because of the differences in the speeds of the CPU, the system bus, and memory circuits.

Instruction Execution Cycle

- The CPU go through a predefined sequence of steps to execute a machine instruction, called the *instruction execution cycle*.
- The instruction pointer (IP) register holds the address of the instruction we want to execute.

Instruction Execution Cycle

Here are the steps to execute it:

1. First, the CPU has to **fetch the instruction** from an area of memory called the *instruction queue*. It then increments the instruction pointer.

2. Next, the CPU **decodes** the instruction by looking at its binary bit pattern.

- This bit pattern might reveal that the instruction has operands (input values).

3. If operands are involved, the CPU **fetches the operands** from registers and memory.

- Sometimes, this involves address calculations.

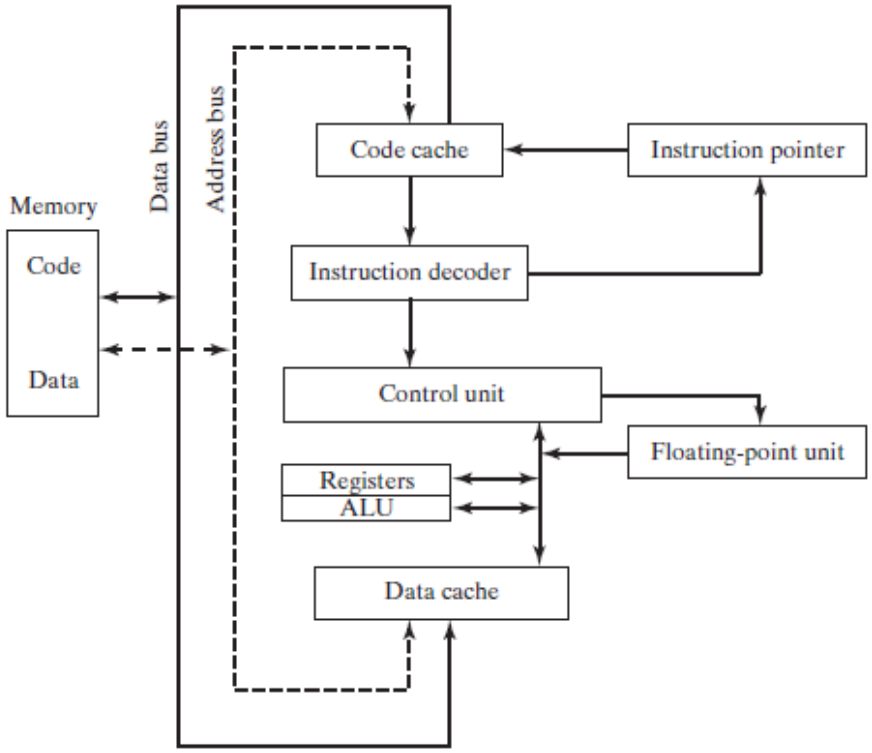
4. Next, the CPU **executes** the instruction, using any operand values it fetched during the earlier step. It also updates a few status flags, such as Zero, Carry, and Overflow.

5. Finally, if an output operand was part of the instruction, the CPU **stores the result** of its execution in the operand.

Instruction Execution Cycle

- An *operand* is a value that is either an input or an output to an operation.
- For example, the expression Z = X + Y has two input operands (X and Y) and a single output operand (Z).
- In order to read program instructions from memory, an address is placed on the address bus.
- Next, the memory controller places the requested code on the data bus, making the code available inside the code cache.

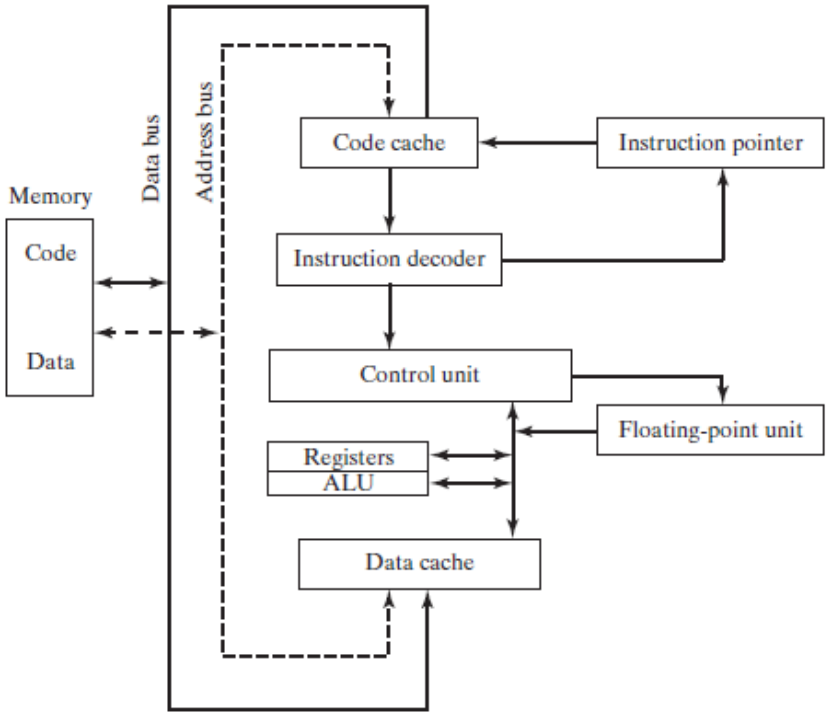
FIGURE 2–2 Simplified CPU block diagram.



Instruction Execution Cycle

- The instruction pointer's value determines which instruction will be executed next.
- The instruction is analyzed by the instruction decoder, causing the appropriate digital signals to be sent to the control unit, which coordinates the ALU and floating-point unit.
- Control bus carries signals that use the system clock to coordinate the transfer of data between the different CPU components.

FIGURE 2–2 Simplified CPU block diagram.



Reading from Memory

As a rule, computers read memory much more slowly than they access internal registers.

Reading a single value from memory involves four separate steps:

1. Place the address of the value you want to read on the address bus.
2. Assert (change the value of) the processor's RD *(read)* pin.
3. Wait one clock cycle for the memory chips to respond.
4. Copy the data from the data bus into the destination operand.

Each of these steps generally requires a single *clock cycle*,

Cache

- CPU designers figured out that computer memory creates a speed bottleneck because most programs have to access variables.
- To reduce the amount of time spent in reading and writing memory the most recently used instructions and data are stored in high-speed memory called *cache*.
- The idea is that a program is more likely to want to access the same memory and instructions repeatedly, so cache keeps these values where they can be accessed quickly.
- When the CPU begins to execute a program, it loads the next thousand instructions (for example) into cache, on the assumption that these instructions will be needed in the near future.
- If there happens to be a loop in that block of code, the same instructions will be in cache.
- When the processor is able to find its data in cache memory, we call that a *cache hit*.
- On the other hand, if the CPU tries to find something in cache and it's not there, we call that a *cache miss*.

X86 family Cache types

- Cache memory for the x86 family comes in two types.
  - *Level-1 cache* (or *primary cache*) is stored right on the CPU.
  - *Level-2 cache* (or *secondary cache*) is a little bit slower, and attached to the CPU by a high-speed data bus.

why cache memory is faster than conventional RAM?

- It's because cache memory is constructed from a special type of memory chip called *static RAM*.
  - It's expensive, but it does not have to be constantly refreshed in order to keep its contents.
- On the other hand, conventional memory, known as *dynamic RAM*, must be refreshed constantly.
  - It's much slower, but cheaper.

Loading and Executing a Program (1)

- The operating system (OS) searches for the program's filename in the current disk directory.
  - If it cannot find the name there, it searches a predetermined list of directories (called *paths*) for the filename.
  - If the OS fails to find the program filename, it issues an error message.
- If the program file is found, the OS retrieves basic information about the program's file from the disk directory, including the file size and its physical location on the disk drive.
- The OS determines the next available location in memory and loads the program file into memory.
  - It allocates a block of memory to the program and enters information about the program's size and location into a table (sometimes called a *descriptor table*).
  - Additionally, the OS adjust the values of pointers within the program so they contain addresses of program data.

Loading and Executing a Program (2)

- The OS begins execution of the program's first machine instruction (its entry point).
- As soon as the program begins running, it is called a *process*.
- The OS assigns the process an identification number (*process ID*), which is used to keep track of it while running.
- It is the OS's job to track the execution of the process and to respond to requests for system resources.
  - Examples of resources are memory, disk files, and input-output devices.
- When the process ends, it is removed from memory.

Mode of Operations

x86 processors have three primary modes of operation: protected mode, real-address mode, and system management mode. A sub-mode, named *virtual-8086*, is a special case of protected mode.

- Real-Address mode (original mode provided by 8086)
  - Only 1 MB of memory can be addressed, from 0 to FFFFF (hex)
  - Programs can access any part of main memory
  - MS-DOS runs in real-address mode
- This mode is available in Windows 98, and can be used to run an MS-DOS program that requires direct access to system memory and hardware devices.
- Programs running in real-address mode can cause the operating system to crash (stop responding to commands).
- Implements the programming environment of the Intel 8086 processor

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

Mode of Operations

- Protected mode (introduced with the 80386 processor)
  - Each program can address a maximum of 4 GB of memory
  - The operating system assigns memory to each running program
  - Programs are prevented from accessing each other's memory
  - Native mode used by Windows NT, 2000, XP, and Linux

- Protected mode is the native state of the processor, in which all instructions and features are available.

- Programs are given separate memory areas named *segments*, and the processor prevents programs from referencing memory outside their assigned segments.

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

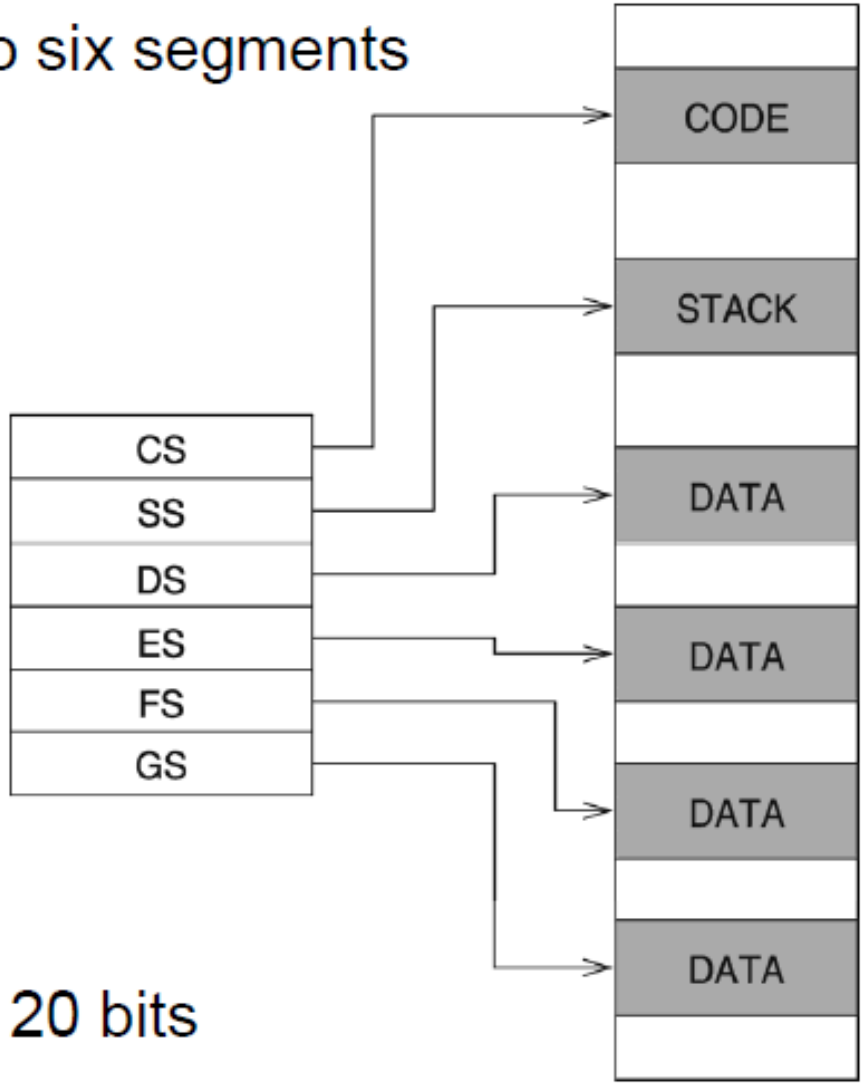
Mode of Operations

- Virtual 8086 mode
  - Processor runs in protected mode, and creates a virtual 8086 machine with 1 MB of address space for each running program
- In protected mode, the processor can directly execute real-address mode software such as MS-DOS programs in a safe multitasking environment.
- If an MS-DOS program crashes or attempts to write data into the system memory area, it will not affect other programs running at the same time.
- Windows XP can execute multiple separate virtual-8086 sessions at the same time.

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

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

Program Segments:

- The code segment holds the program instruction codes.
- The data segment stores data for the program.
- The extra segment is an extra data segment (often used for shared data).
- The stack segment is used to store interrupt and subroutine return addresses.

Real Address Mode (2)

Linear address = Segment × 10 (hex) + Offset

Example:

segment = A1F0 (hex)

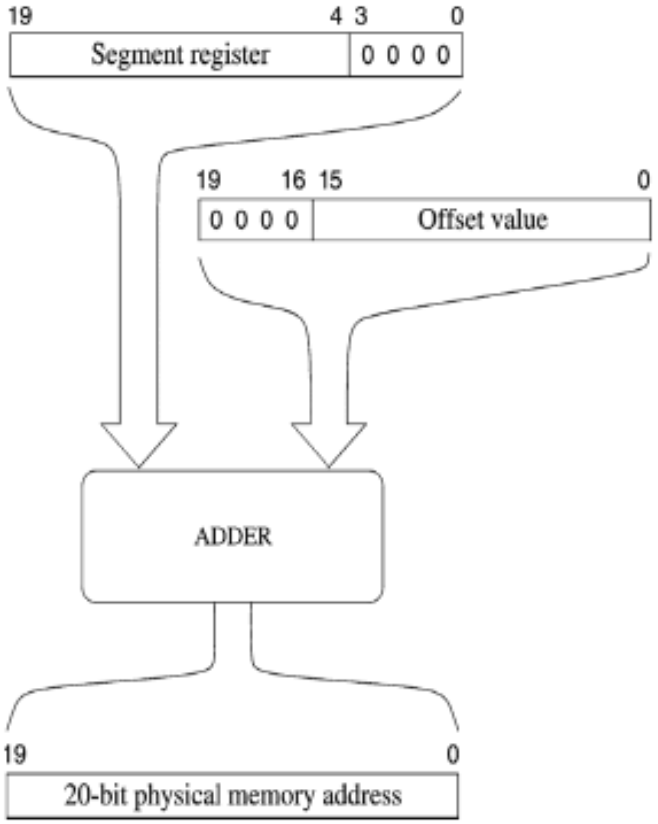
offset = 04C0 (hex)

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)

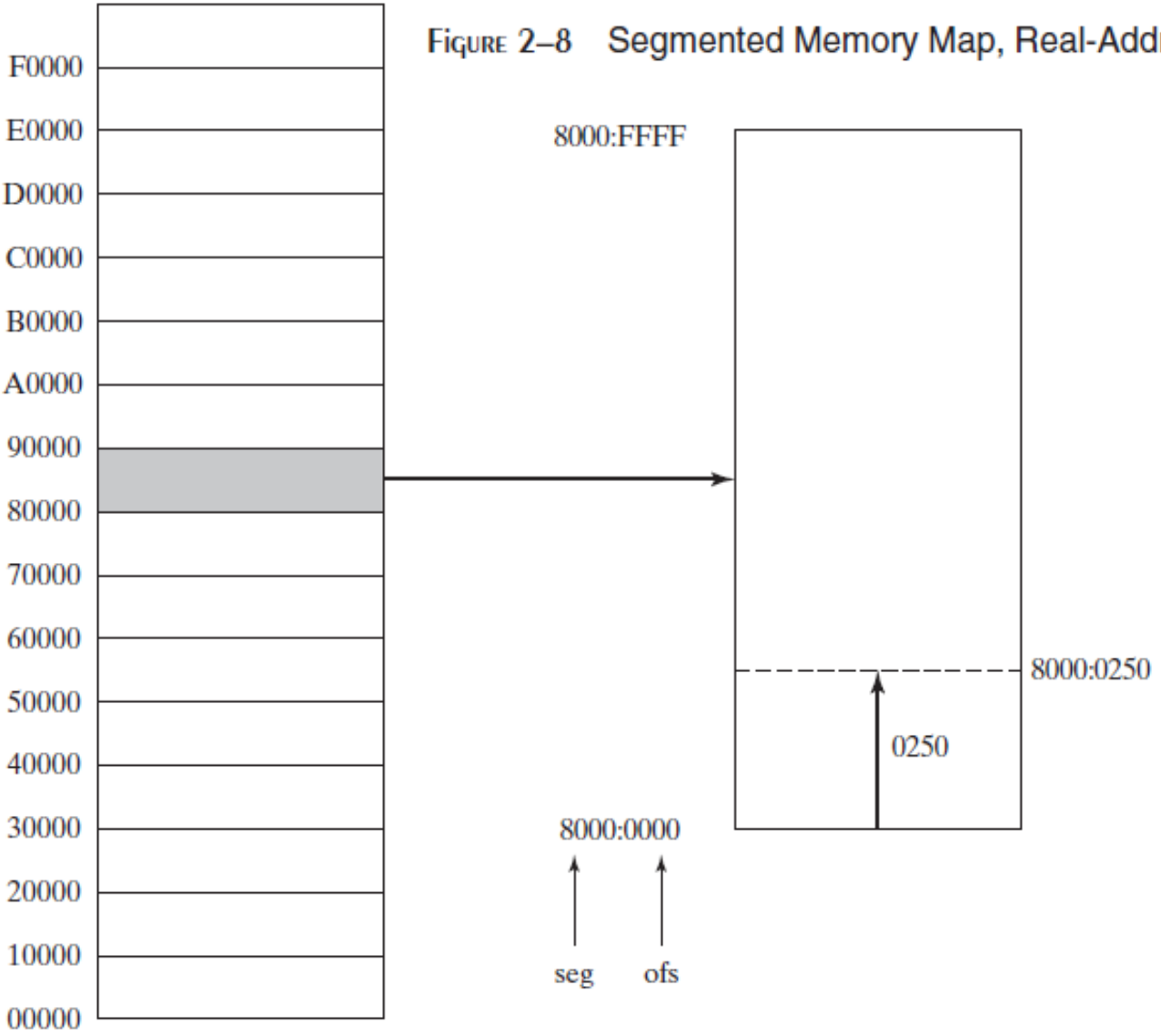


FIGURE 2–8 Segmented Memory Map, Real-Address Mode.

Address Space

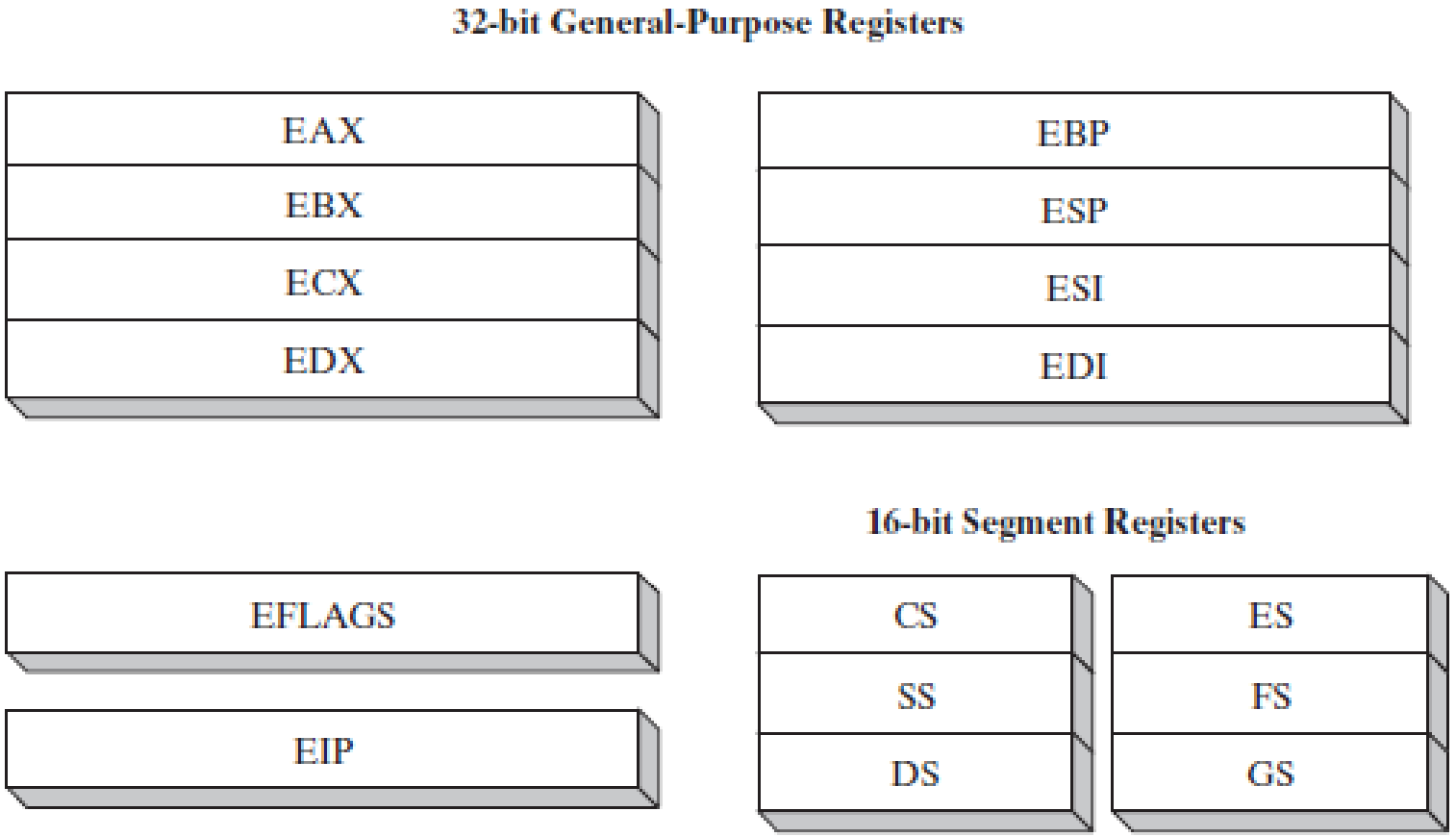
- In 32-bit protected mode, a task or program can address a linear address space of up to 4 GBytes.
  - Extended Physical Addressing allows a total of 64 GBytes of physical memory to be addressed.
- Real-address mode programs, on the other hand, can only address a range of 1 MByte.
- If the processor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

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.



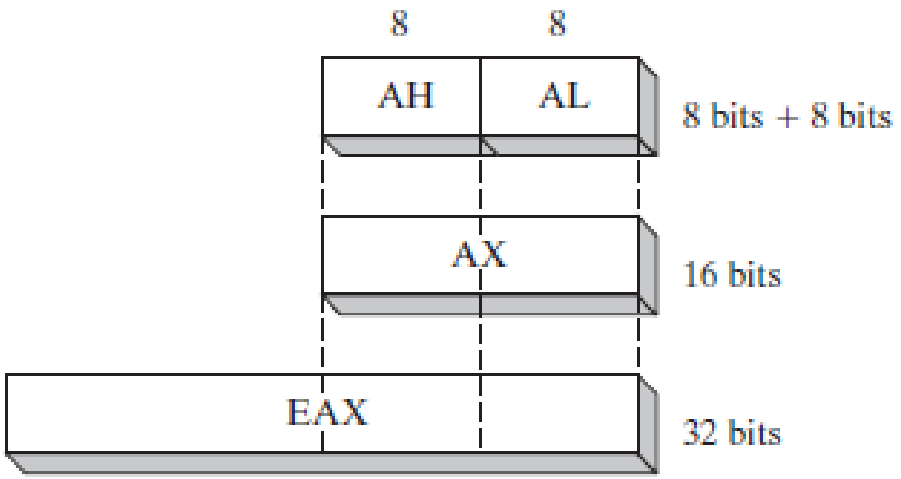
General-Purpose Registers:

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

Some general-purpose registers have specialized uses:

- EAX is automatically used by multiplication and division instructions. It is often called the *extended accumulator* register.

- The CPU automatically uses ECX as a loop counter.

- ESP (*extended stack pointer* register) addresses data on the stack (a system memory structure). It is rarely used for ordinary arithmetic or data transfer.

- ESI and EDI (*extended source index* and *extended destination index*) are used by high-speed memory transfer instructions.

- EBP is used by high-level languages to reference function parameters and local variables on the stack. It should not be used for ordinary arithmetic or data transfer except at an advanced level of programming. It is often called the *extended frame pointer* register.

- *Segment Registers:*
  - In real-address mode, 16-bit segment registers indicate base addresses of pre-assigned memory areas named *segments*.
  - In protected mode, segment registers hold pointers to segment descriptor tables (The descriptor describes the location, length and access rights of the memory segment).
  - Some segments hold program instructions (code), others hold variables (data), and another segment named the *stack segment* holds local function variables and function parameters.
- *Instruction Pointer :*
  - 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 (or just *Flags*) 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.
- *Control Flags:*
  - Control flags control the CPU's operation.
  - For example, they can cause the CPU to break after every instruction executes, interrupt when arithmetic overflow is detected, enter virtual-8086 mode, and enter protected mode.
- Programs can set individual bits in the EFLAGS register to control the CPU's operation.
- Examples are the *Direction* and *Interrupt* flags.

Status Flags

- The Status flags reflect the outcomes of arithmetic and logical operations performed by the CPU. They are the Overflow, Sign, Zero, Auxiliary Carry, Parity, and Carry flags.
  - The **Carry** flag (CF) is set when the result of an *unsigned* arithmetic operation is too large to fit into the destination.
  - The **Overflow** flag (OF) is set when the result of a *signed* arithmetic operation is too large or too small to fit into the destination.
  - The **Sign** flag (SF) is set when the result of an arithmetic or logical operation generates a negative result.

- The **Zero** flag (ZF) is set when the result of an arithmetic or logical operation generates a result of zero.
- The **Parity** flag (PF) is set if the least-significant byte in the result contains an even number of 1 bits. Otherwise, PF is clear. In general, it is used for error checking when there is a possibility that data might be altered or corrupted.

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]
```

Instructions

- An instruction is a statement that becomes executable when a program is assembled.
- Assembled into machine code by assembler
- Executed at runtime by the CPU
- We use the Intel IA-32 instruction set
- An instruction contains:
  - Label (optional)
  - Mnemonic (required)
  - Operand (depends on the instruction)
  - Comment (optional)
- Basic syntax
  - [label:] mnemonic [operands] [; comment]

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
```

A code label can share the same line with an instruction, or it can be on a line by itself:

```
L1: mov   ax,bx
L2:
```

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)

Mnemonics and Operands

- Instruction Mnemonics
  - memory aid
  - examples: MOV, ADD, SUB, MUL, INC, DEC
- Operands
  - constant 96
  - constant expression 2 + 4
  - register eax
  - memory (data label) count

Constants and constant expressions are often called immediate values

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.)

```
.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
```