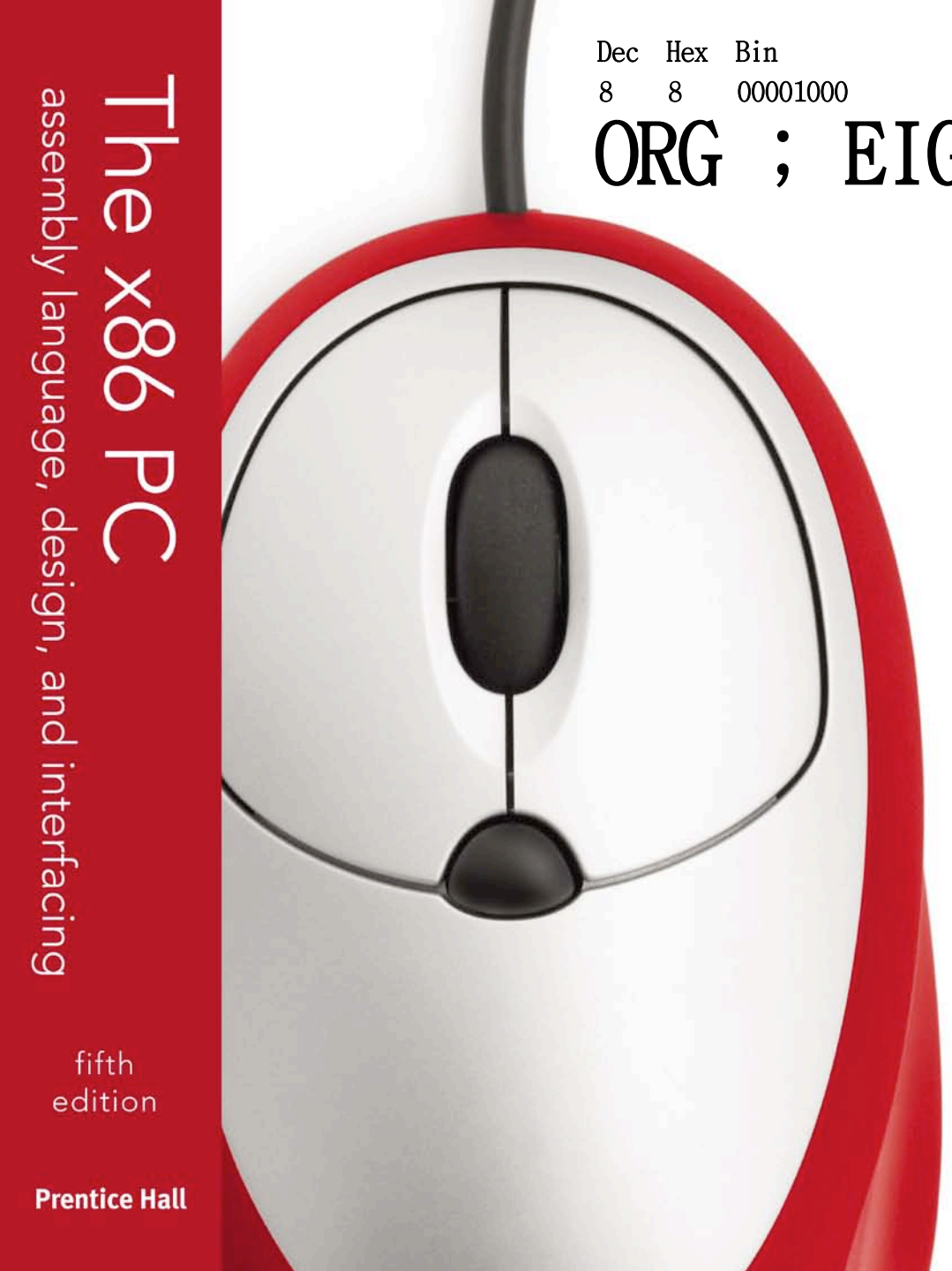


Dec Hex Bin
8 8 00001000

ORG ; EIGHT

32-Bit Programming for x86

The x86 PC

assembly language, design, and interfacing

fifth edition

MUHAMMAD ALI MAZIDI
JANICE GILLISPIE MAZIDI
DANNY CAUSEY

OBJECTIVES

this chapter enables the student to:

- Discuss the major differences between the 16-bit and 32-bit CPUs.
- List the 32-bit registers of the x86 CPU.
- Diagram the register sizes available in the 32-bit CPUs.
- Explain the difference in register usage between the 16-bit and the 32-bit systems.
- Discuss how the increased register size of 32-bit systems relates to an increased memory range.

OBJECTIVES (*cont*)

this chapter enables the student to:

- Diagram how the “little endian” storage convention of x86 machines stores doubleword-sized operands.
- Code programs for the 32-bit CPU using extended registers and new directives.
- Code arithmetic statements using the extended registers of the 32-bit CPUs.
- Code Assembly language within C programs by using in-line coding.

8.1: 32-BIT PROGRAMMING IN x86
register size

- 386 & higher CPU register size was extended to 32 bits, with all register names changed to reflect this.
  - AX has become EAX, BX is now EBX, etc.
    - 386 & higher CPUs contain registers AL, AH, AX, and EAX, with 8, 8, 16, and 32 bits, respectively.

8.1: 32-BIT PROGRAMMING IN x86 registers

- There are a two new segment registers: **FS** & **GS** and several control registers: **CR0**, **CR1**, **CR2**, **CR3**.

Table 8-1: Registers of the 32-bit x86 by Category

Category	Bits	Register Names
General	32	EAX, EBX, ECX, EDX
	16	AX, BX, CX, DX
	8	AH, AL, BH, BL, CH, CL, DH, DL
Pointer	32	ESP (extended SP), EBP (extended BP)
	16	SP (stack pointer), BP (base pointer)
Index	32	ESI (extended SI), EDI (extended DI)
	16	SI (source index), DI (destination index)
Segment	16	CS (code segment), DS (data segment)
		SS (stack segment), ES (extra segment)
		FS (extra segment), GS (extra segment)
Instruction	32	EIP (extended instruction pointer)
Flag	32	EFT (extended flag register)
Control	32	CR0, CR1, CR2, CR3

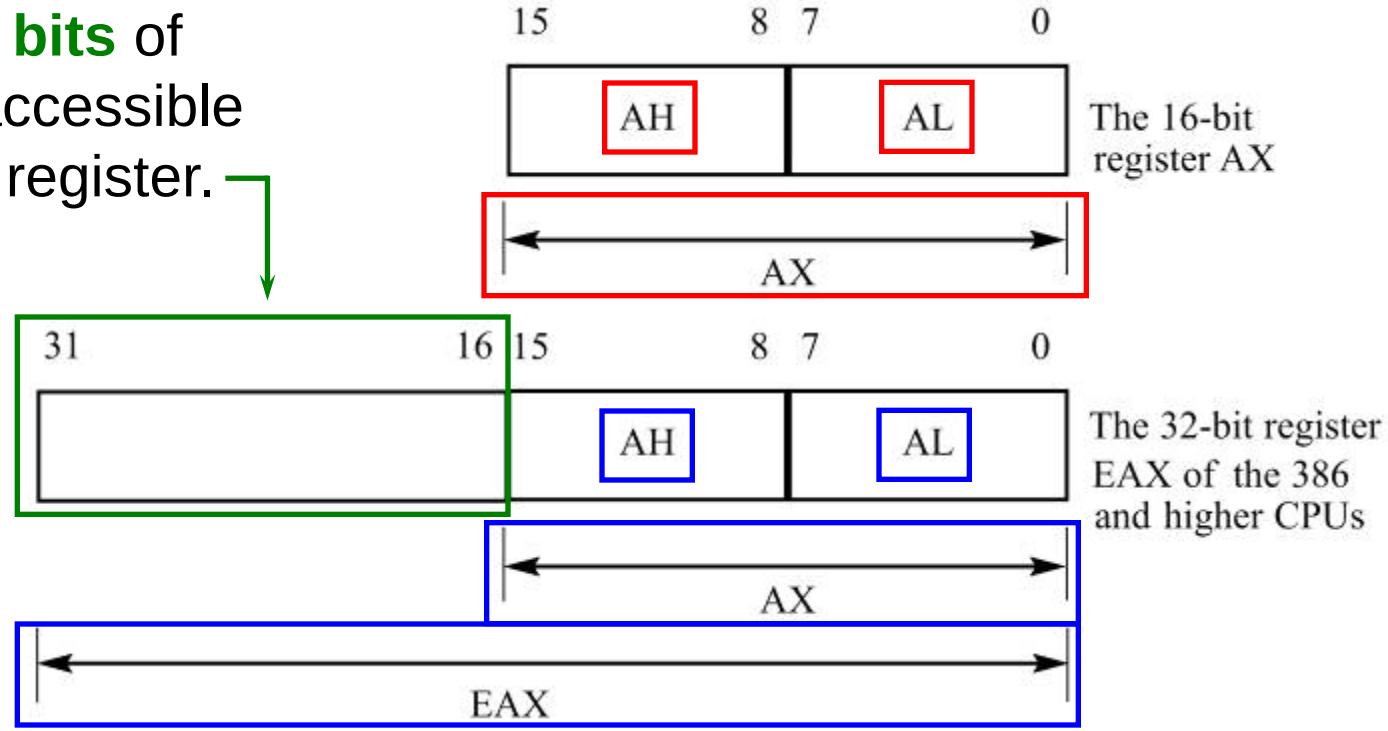
Note: Only bit 0 of CR0 is available in real mode. All other control registers are available in protected mode only.

8.1: 32-BIT PROGRAMMING IN x86
register function

- In the 16-bit, register **AX** is accessible either as **AL**, **AH** *or* **AX**, while in the 32-bit, register **EAX** can be accessed only as **AL**, **AH**, **AX** *or* **EAX**.

The **upper 16 bits** of **EAX** are *not* accessible as a separate register.

The same rule applies to EBX, ECX, and EDX.



8.1: 32-BIT PROGRAMMING IN x86
32-bit general registers as pointers

- A major change from 16- to 32-bit is the ability of general registers to be used as pointers.
  - Such as EAX, ECX, and EDX
    - AX, CX, and DX could *not* be used as pointers.
- Starting with the 386, these instructions are legal:

```
MOV     AX,[ ECX]
ADD     SI,[ EDX]
OR      EBX,[ EAX] +20
```

8.1: 32-BIT PROGRAMMING IN x86
32-bit general registers as pointers

When **EAX**, **ECX**, or **EDX** are used as offset addresses, **DS** is the default segment register.

SS is the default segment register for **ESP** and EBP.

CS is the default for EIP & DS for all other registers.

Table 8-2: Addressing Modes for 32-bit Programming

Addressing Mode	Operand	Default Segment
Register	register	none
Immediate	data	none
Direct	[offset]	DS
Register indirect	[BX]	DS
	[SI]	DS
	[DI]	DS
	[EAX]	DS
	[EBX]	DS
	[ECX]	DS
	[EDX]	DS
	[ESI]	DS
	[EDI]	DS
Based relative	[BX]+disp	DS
	[BX]+disp	SS
	[EAX]+disp	DS
	[EBX]+disp	DS
	[ECX]+disp	DS
	[EDX]+disp	DS
	[EBP]+disp	SS
Indexed relative	[DI]+disp	DS
	[SI]+disp	DS
	[EDI]+disp	DS
	[ESI]+disp	DS
Based indexed relative	[R1][R2]+disp where RI and R2 are any of the above	If BP is used, the segment is SS; otherwise; DS is the segment

Note: In based indexed relative addressing, disp is optional.

8.1: 32-BIT PROGRAMMING IN x86
accessing 32-bit registers

- The Assembly language directive ".386" is used to access 32-bit registers of 386 and higher CPUs.
  - The ".386" directive means the program cannot run on 8086/286.
  - Additional assembler directives indicate the type of microprocessor supported by (MASM):

MASM	Meaning
.86	will run on any x86 CPU (default)
.386	will run on any 386 and higher CPU; allows use of new 386 instructions

8.1: 32-BIT PROGRAMMING IN x86
accessing 32-bit registers

- Program 8-1 demonstrates the ".386" directive, 386 32-bit instructions & simplified segment definition.
  - Register EAX adds/subtracts values of various size to demonstrate 32-bit programming of the x86.

```
MOV     AX,@DATA
MOV     DS,AX
SUB     EAX,EAX
ADD     EAX,100000   ;EAX = 186A0H
ADD     EAX,200000   ;EAX = 186A0H + 30D40 H = 493E4H
ADD     EAX,40000    ;EAX = 493E4H + 9C40H = 53020H
SUB     EAX,80000    ;EAX = 53020H - 13880H = 3F7A0H
SUB     EAX,35000    ;EAX = 3F7A0H - 88B8H = 36EE8H
SUB     EAX,250      ;EAX = 36338H - FAH = 36DEEH (224750)
MOV     RESULT,EAX
```

See the entire program listing on page 220 of your textbook.

8.1: 32-BIT PROGRAMMING IN x86
accessing 32-bit registers

- In MASM, the CodeView utility allows monitoring of the execution of 16-bit and 32-bit programs.
  - Shown is a partial CodeView trace of Program 8-1.

```
File View Search Run Watch Options Language Calls Help | F8=Trace F5=Go

4833:0000 B83648 MOV AX,4836 EAX=00036DEE
4833:0003 8ED8 MOV DS,AX EBX=00000000
4833:0005 662BC0 SUB EAX,EAX ECX=00000000
4833:0008 6605A0860100 ADD EAX,000186A0 EDX=00000000
4833:000E 6605400D0300 ADD EAX,00030D40 ESP=00000200
4833:0014 6605409C0000 ADD EAX,00009C40 EBP=00000000
4833:001A 662D80380100 SUB EAX,00013880 ESI=00000000
4833:0020 662DB8880000 SUB EAX,000088B8 EDI=00000000
4833:0026 662DFA000000 SUB EAX,000000FA DS=...4836
 ES=....4823
```

Register EAX adds/subtracts values of various size to demonstrate 32-bit programming of the x86.

8.1: 32-BIT PROGRAMMING IN x86
little endian revisited

- x86 stores 32-bit data in memory, or loads 32-bit operands into registers with little endian convention.
  - Low byte to low address; high byte to high address.

```
MOV     AX,@DATA
MOV     DS,AX
SUB     EAX,EAX
ADD     EAX,100000    ;EAX =
ADD     EAX,200000    ;EAX =
ADD     EAX,40000     ;EAX =
SUB     EAX,80000     ;EAX =
SUB     EAX,35000     ;EAX =
SUB     EAX,250       ;EAX =
MOV     RESULT,EAX
```

An instruction such as "**MOV RESULT,EAX**" will store the data in this way:

OFFSET	CONTENTS
RESULT	d0-d7
RESULT+1	d8-d15
RESULT+2	d16-d23
RESULT+3	d24-d31

See the entire program listing on page 220 of your textbook.

8.1: 32-BIT PROGRAMMING IN x86
little endian revisited

- x86 stores 32-bit data in memory, or loads 32-bit operands into registers with little endian convention.
  - Low byte to low address; high byte to high address.

Example 8-1

Assuming that SI = 1298 and EAX = 41992F56H, show the contents of memory locations after the instruction "`MOV [SI],EAX`".

Solution (in hex):

DS:1298 = (56)
DS:1299 = (2F)
DS:129A = (99)
DS:129B = (41)

8.1: 32-BIT PROGRAMMING IN x86
adding 16-bit words with 32-bit registers

- Program 3-1B used 16-bit registers for adding several words of data.
  - The sum was accumulated in one register and another register was used to add up the carries.
    - Not necessary when using 32-bit registers.

```
TITLE     PROG3-1B (EXE) ADDING 5 WORDS
PAGE      60,132
          MOV   CX,COUNT          ;CX is the loop counter
          MOV   SI,OFFSET DATA    ;SI is the data pointer
          MOV   AX,00             ;AX will hold the sum
          MOV   BX,AX             ;BX will hold the carries
BACK:ADD  AX,[ SI]                ;add the next word to AX
```

See the entire program listing on page 94 of your textbook.

8.1: 32-BIT PROGRAMMING IN x86
adding 16-bit words with 32-bit registers

```
TITLE       REVISION OF PROGRAM 3-1B USING 32-BIT REGISTERS
PAGE  60,132
            .MODEL SMALL
            .386
            .STACK 200H
            .DATA
DATA1 DD    27345,28521,29533,30105,32375
SUM   DD    ?
COUNT EQU   5
            .CODE
BEGIN:      MOV   AX,@DATA
            MOV   DS,AX
            MOV   CX,COUNT            ;CX is loop counter
            MOV   SI,OFFSET DATA1     ;SI is data pointer
            SUB   EAX,EAX             ;EAX will hold sum
BACK:       ADD   EAX,DWORD PTR[ SI]  ;add next word to EAX
            ADD   SI,4                ;SI points to next dword
            DEC   CX                  ;decrement loop counter
            JNZ   BACK                ;continue adding
            MOV   SUM,EAX             ;store sum
            MOV   AH,4CH
            INT   21H
            END   BEGIN
```

Review Program 3-1B, then examine Program 8-2, a 32-bit version of the same program, written for 386 and higher CPUs.

See the program listing on page 222 of your textbook.

8.1: 32-BIT PROGRAMMING IN x86
adding multiword data in 32-bit

- In Program 3-2, two multiword numbers were added using 16-bit registers.
  - Each number could be as large as 8 bytes wide.
    - That program required a total of four iterations.

```
TITLE     PROG3-2 (EXE) MULTIWORD ADDITION
PAGE      60,132

BACK:MOV  AX,[ SI]   ;move the first operand to AX
     ADC  AX,[ DI]   ;add the second operand to AX
     MOV  [ BX] ,AX  ;store the sum
```

See the entire program listing on page 95 of your textbook.

8.1: 32-BIT PROGRAMMING IN x86
adding multiword data in 32-bit

- Using the 32-bit registers of the 386/46 requires only two iterations, as shown in Program 8-3a.

```
TITLE   ADD TWO 8-BYTE NUMBER USING 32-BIT REGISTERS IN THE 386
PAGE    60,132

        MOV   SI,OFFSET DATA1    ;SI is pointer for operand1
        MOV   DI,OFFSET DATA2    ;DI is pointer for operand2
        MOV   BX,OFFSET DATA3    ;BX is pointer for the sum
        MOV   CX,02              ;CX is the loop counter
BACK:MOV   EAX,DWORD PTR [ SI]      ;move the operand to EAX
        ADC   EAX,DWORD PTR [ DI]      ;add the operand to EAX
        MOV   DWORD PTR [ BX] ,EAX     ;store the sum
```

However, this loop version of the program is very long &inefficient.

```
        LOOP  BACK               ;if not finished, continue adding
        MOV   AH,4CH
        INT   21H                ;go back to DOS
```

See the program listing on page 223 of your textbook.

8.1: 32-BIT PROGRAMMING IN x86
adding multiword data in 32-bit

- Due to penalties associated with instructions such as LOOP & Jcondition, in 386 and higher CPUs, it is better to use the nonloop version of this program:

```
TITLE   ADD TWO 8-BYTE NUMBER USING 32-BIT REGISTERS IN THE 386
PAGE 60,132                 ;(NO-LOOP VERSION)

BEGIN:MOV   AX,@DATA
      MOV   DS,AX
      MOV   EAX,DWORD PTR DATA1    ;move lower dword of DATA1 into EAX
      ADD   EAX,DWORD PTR DATA2    ;add lower dword of DATA2 to EAX
      MOV   EBX,DWORD PTR DATA1+4  ;move upper dword of DATA1 into EBX
      ADC   EBX,DWORD PTR DATA2+4  ;add upper dword of DATA2 to EBX
      MOV   DWORD PTR DATA3,EAX    ;store lower dword of result
      MOV   DWORD PTR DATA3+4,EBX  ;store upper dword of result
      MOV   AH,4CH
      INT   21H
      END   BEGIN
```

See the program listing on page 224 of your textbook.

8.1: 32-BIT PROGRAMMING IN x86
combining C with Assembly

- Although Assembly language is the fastest language available for a given CPU, it cannot be run on different CPUs.
  - A portable language is needed.
- Today, a large portion of programs written for all computers are in the C/C++ language.
  - A universal programming language that it can be run on any CPU architecture with little or no modification
    - It is simply recompiled for that CPU.
  - Combining C & Assembly takes advantage of C/C++'s portability and Assembly's speed.

8.1: 32-BIT PROGRAMMING IN x86
combining C with Assembly

- There are two ways to mix C/C++ and Assembly.
  - One is simply to insert the Assembly code in C programs.
    - Commonly referred to as *in-line assembly*.
  - The second method is to make the C/C++ language call an external Assembly language procedure.
- The following code demonstrates how to change the cursor position to row = 10 & column = 20.
  - Assembly instructions are prefaced with "asm“, a reserved word.
    - Microsoft uses the keyword "_asm".

Inserting x86 assembly code into Visual C++ programs - inlining

```
/* version 1: using keyword asm before each line of in-line code */
/* Microsoft uses keyword "_asm"  */
/* compiled in Visual C++ 2005 Express Edition - a free download
-- from Microsoft website*/

#include <iostream>
#include <windows.h>
#include <tchar.h>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
   int data1=0xFFFFFF;
   int data2=0xFFFFFF;
   int sum;
   _asm{
      mov eax,data1
      mov ebx,data2
      add eax,ebx
      mov sum,eax
   }
   cout<<sum;
   return 0;
}
```

Note that in Microsoft, not all interrupts may be supported in the latest versions of Visual C++.

Each line must end in a semicolon or newline, and any comments must be in the correct form for C.

The x86 PC

assembly language, design, and interfacing

fifth edition

Prentice Hall

Dec Hex Bin
8 8 00001000

ENDS ; EIGHT

The x86 PC

assembly language, design, and interfacing

fifth edition

MUHAMMAD ALI MAZIDI
JANICE GILLISPIE MAZIDI
DANNY CAUSEY