

Assembly Language for x86 Processors

7th Edition , Global Edition

Kip Irvine

Chapter 1: Basic Concepts

Slides prepared by the author

Revision date: 1/15/2014

Modified by: Liang, 2016 Spring

Chapter Overview

- Welcome to Assembly Language
- Virtual Machine Concept
- Data Representation
- Boolean Operations

Welcome to Assembly Language

- Questions to Ask
- Assembly Language Applications

Questions to Ask

- Why am I learning Assembly Language?
- What background should I have?
- What is an assembler?
- What hardware/software do I need?
- What types of programs will I create?
- What do I get with this book?
- What will I learn?

Welcome to Assembly Language (*cont*)

- How does assembly language (AL) relate to machine language?
- How do C++ and Java relate to AL?
- Is AL portable?
- Why learn AL?

Assembly Language Applications

- Some representative types of applications:
 - Business application for single platform
 - Hardware device driver
 - Business application for multiple platforms
 - Embedded systems & computer games

(see next panel)

Comparing ASM to High-Level Languages

Type of Application	High-Level Languages	Assembly Language
Business application software, written for single platform, medium to large size.	Formal structures make it easy to organize and maintain large sections of code.	Minimal formal structure, so one must be imposed by programmers who have varying levels of experience. This leads to difficulties maintaining existing code.
Hardware device driver.	Language may not provide for direct hardware access. Even if it does, awkward coding techniques must often be used, resulting in maintenance difficulties.	Hardware access is straightforward and simple. Easy to maintain when programs are short and well documented.
Business application written for multiple platforms (different operating systems).	Usually very portable. The source code can be recompiled on each target operating system with minimal changes.	Must be recoded separately for each platform, often using an assembler with a different syntax. Difficult to maintain.
Embedded systems and computer games requiring direct hardware access.	Produces too much executable code, and may not run efficiently.	Ideal, because the executable code is small and runs quickly.

What's Next

- Welcome to Assembly Language
- **Virtual Machine Concept**
- Data Representation
- Boolean Operations

Virtual Machine Concept

- Virtual Machines
- Specific Machine Levels

Virtual Machines

- Tanenbaum: Virtual machine concept
- 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
- 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
```

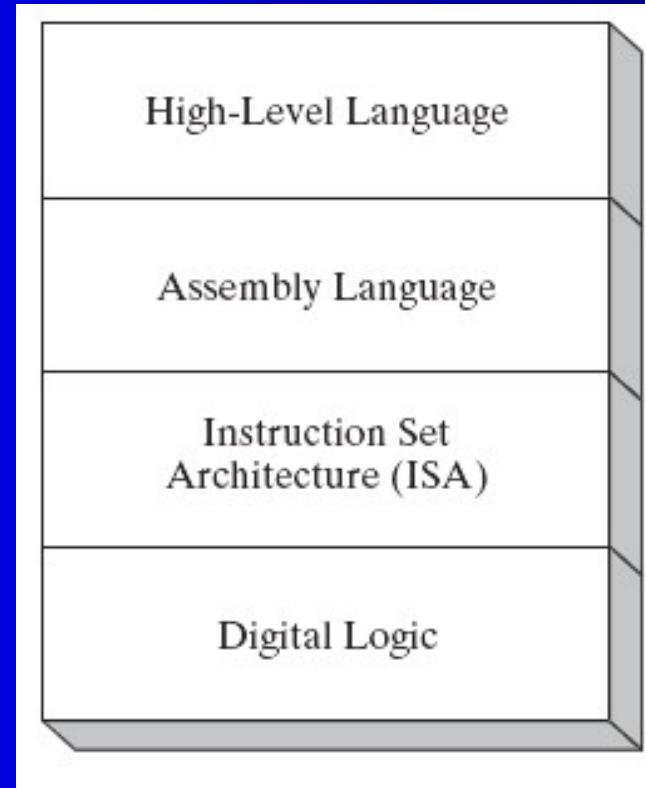
Specific Machine Levels

Level 4

Level 3

Level 2

Level 1



High-Level Language

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

```
a = b + 1;
```

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)
- Example: $a = b + 1$

```
mov eax, b  
add eax, 1  
mov a, eax
```

Instruction Set Architecture (ISA)

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

```
a1 04 30 40 00 ; mov eax, b  
83 c0 01          ; add eax, 1  
a3 00 30 40 00 ; mov a, eax
```

Digital Logic

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

What's Next

- Welcome to Assembly Language
- Virtual Machine Concept
- **Data Representation**
- Boolean Operations

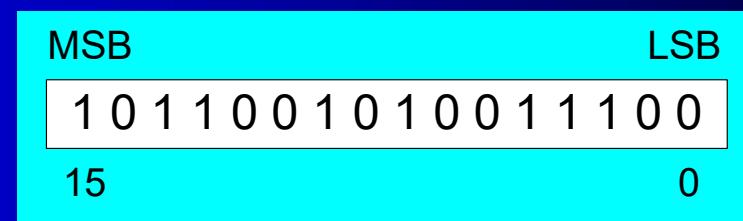
Data Representation

- Binary Numbers
- Hexadecimal Integers
- Unsigned Integer Storage Sizes
- Signed Integers
- Character Storage

Binary Numbers

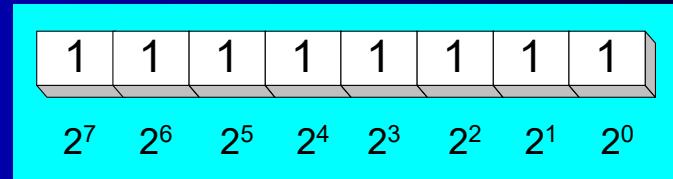
- Digits are 1 and 0
 - 1 = true
 - 0 = false
- MSB – most significant bit
- LSB – least significant bit

- Bit numbering:



Binary Numbers

- Each digit (bit) is either 1 or 0
- Each bit represents a power of 2:



Every binary number is a sum of powers of 2

Table 1-3 Binary Bit Position Values.

2^n	Decimal Value	2^n	Decimal Value
2^0	1	2^8	256
2^1	2	2^9	512
2^2	4	2^{10}	1024
2^3	8	2^{11}	2048
2^4	16	2^{12}	4096
2^5	32	2^{13}	8192
2^6	64	2^{14}	16384
2^7	128	2^{15}	32768

Translating Binary to Decimal

Weighted positional notation shows how to calculate the decimal value of each binary bit:

$$dec = (D_{n-1} \times 2^{n-1}) + (D_{n-2} \times 2^{n-2}) + \dots + (D_1 \times 2^1) + (D_0 \times 2^0)$$

D = binary digit

1	1	1	1	1	1	1	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

0 0 0 1 0 1 1 0 _b

$$\begin{aligned} &= 0 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 \\ &\quad + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\ &= 0 + 0 + 0 + 16 + 0 + 4 + 2 + 0 \\ &= 22 \end{aligned}$$

Translating Unsigned Decimal to Binary

- Repeatedly divide the decimal integer by 2. Each remainder is a binary digit in the translated value:

Division	Quotient	Remainder
$37 / 2$	18	1
$18 / 2$	9	0
$9 / 2$	4	1
$4 / 2$	2	0
$2 / 2$	1	0
$1 / 2$	0	1

$$37 = 2 * 18 + 1$$

$$18 = 2 * 9 + 0$$

$$9 = 2 * 4 + 1$$

$$4 = 2 * 2 + 0$$

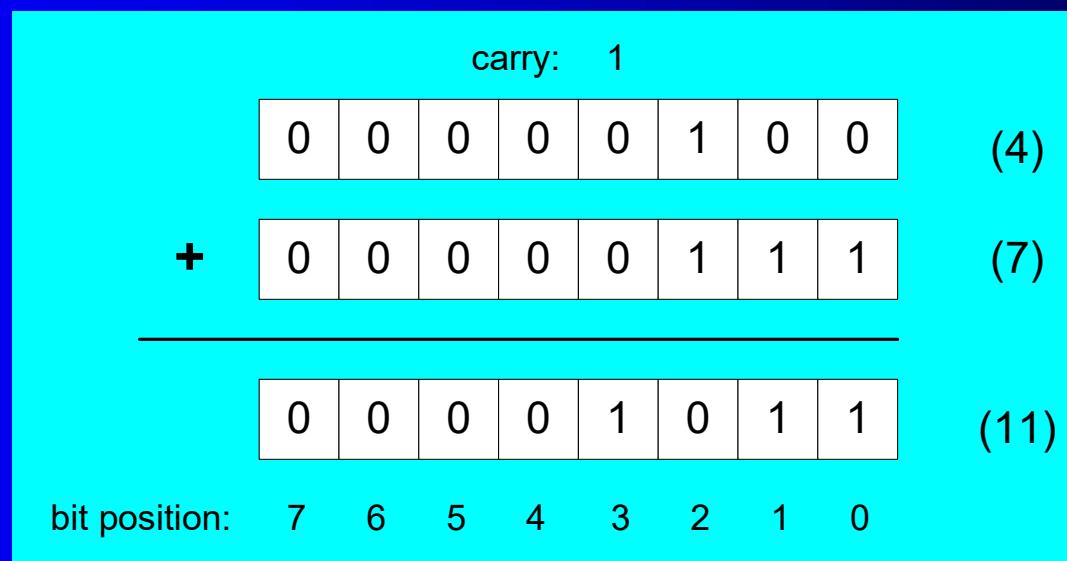
$$2 = 2 * 1 + 0$$

$$1 = 2 * 0 + 1$$

$$37 = 100101$$

Binary Addition

- Starting with the LSB, add each pair of digits, include the carry if present.



Hexadecimal Integers

Binary values are represented in hexadecimal.

Table 1-5 Binary, Decimal, and Hexadecimal Equivalents.

Binary	Decimal	Hexadecimal	Binary	Decimal	Hexadecimal
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	10	A
0011	3	3	1011	11	B
0100	4	4	1100	12	C
0101	5	5	1101	13	D
0110	6	6	1110	14	E
0111	7	7	1111	15	F

Translating Binary to Hexadecimal

- Each hexadecimal digit corresponds to 4 binary bits.
- Example: Translate the binary integer
000101101010011110010100 to hexadecimal:

1	6	A	7	9	4
0001	0110	1010	0111	1001	0100

Converting Hexadecimal to Decimal

- Multiply each digit by its corresponding power of 16:

$$\text{dec} = (D_3 \times 16^3) + (D_2 \times 16^2) + (D_1 \times 16^1) + (D_0 \times 16^0)$$

- Hex 1234 equals $(1 \times 16^3) + (2 \times 16^2) + (3 \times 16^1) + (4 \times 16^0)$, or decimal 4,660.
- Hex 3BA4 equals $(3 \times 16^3) + (11 \times 16^2) + (10 \times 16^1) + (4 \times 16^0)$, or decimal 15,268.

Powers of 16

Used when calculating hexadecimal values up to 8 digits long:

16^n	Decimal Value	16^n	Decimal Value
16^0	1	16^4	65,536
16^1	16	16^5	1,048,576
16^2	256	16^6	16,777,216
16^3	4096	16^7	268,435,456

Converting Decimal to Hexadecimal

Division	Quotient	Remainder
$422 / 16$	26	6
$26 / 16$	1	A
$1 / 16$	0	1

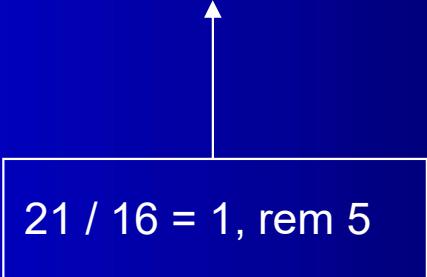
decimal 422 = 1A6 hexadecimal

Hexadecimal Addition

- Divide the sum of two digits by the number base (16). The quotient becomes the carry value, and the remainder is the sum digit.

36	28	28	6A
42	45	58	4B
78 6D 80 B5			

$$21 / 16 = 1, \text{ rem } 5$$



Important skill: Programmers frequently add and subtract the addresses of variables and instructions.

Hexadecimal Subtraction

- When a borrow is required from the digit to the left, add 16 (decimal) to the current digit's value:

$$\begin{array}{r} 16 + 5 = 21 \\ \downarrow \\ \begin{array}{r} C6 \quad 75 \\ A2 \quad 47 \\ \hline 24 \quad 2E \end{array} \end{array}$$

Practice: The address of **var1** is 00400020. The address of the next variable after var1 is 0040006A. How many bytes are used by var1?

Integer Storage Sizes

Standard sizes:

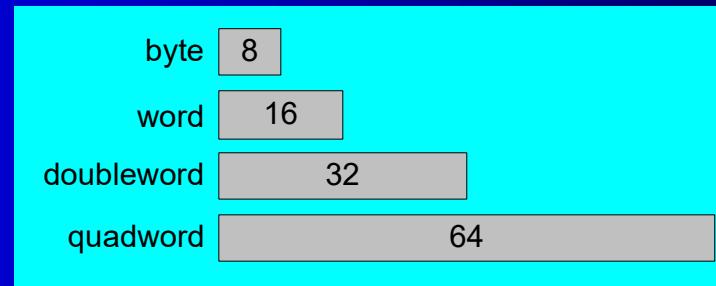


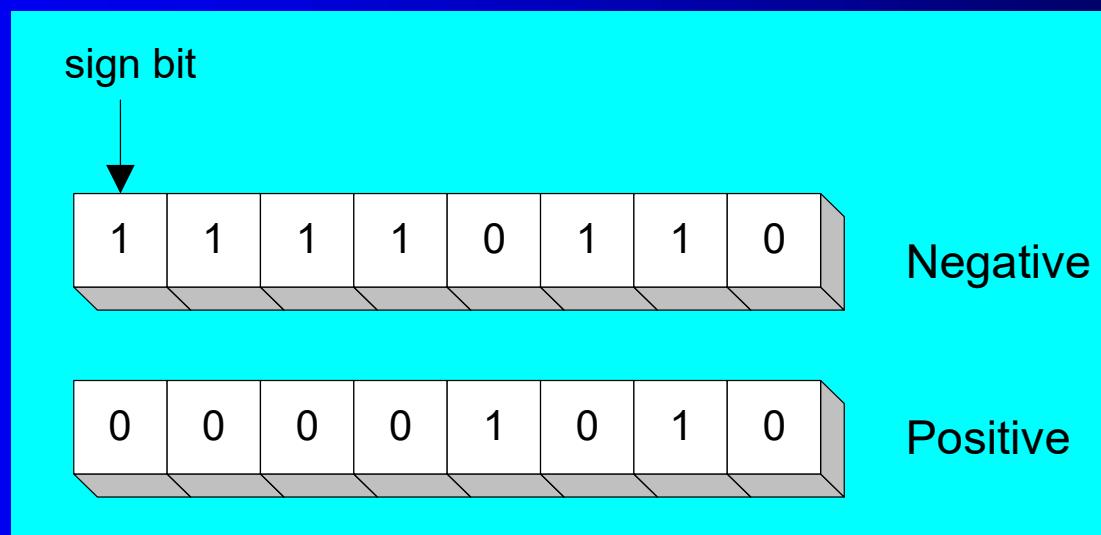
Table 1-4 Ranges of Unsigned Integers.

Storage Type	Range (low–high)	Powers of 2
Unsigned byte	0 to 255	0 to $(2^8 - 1)$
Unsigned word	0 to 65,535	0 to $(2^{16} - 1)$
Unsigned doubleword	0 to 4,294,967,295	0 to $(2^{32} - 1)$
Unsigned quadword	0 to 18,446,744,073,709,551,615	0 to $(2^{64} - 1)$

What is the largest unsigned integer that may be stored in 20 bits?

Signed Integers

The highest bit indicates the sign. 1 = negative, 0 = positive



If the highest digit of a hexadecimal integer is > 7, the value is negative. Examples: 8A, C5, A2, 9D

Forming the Two's Complement

- Negative numbers are stored in two's complement notation
- Represents the additive Inverse

What we want?

Find the negative value

How to find?

Find the value which add original value will equal to zero

	Two's Complement	Decimal
Starting value	0000 0010	2
Step 1 : reverse the bits	1111 1101	-3
Step 2 : add 1 to the value from Step 1	1111 1101 +0000 0001	-3+1
Sum : two's complement representation	1111 1110	-2

Note that

$$\begin{cases} 00000010 + 11111101 = 11111111 \\ 00000011 + 11111100 = 11111111 \end{cases}$$

Sum of every number and its reverse is 1111 1111
So we need plus 0000 0001
This is the reason of step 2.

Binary Subtraction

- When subtracting $A - B$, convert B to its two's complement
- Add A to $(-B)$

$$\begin{array}{r} 00001100 \\ - 00000011 \\ \hline \end{array} \quad \xrightarrow{\hspace{1cm}} \quad \begin{array}{r} 00001100 \\ 11111101 \\ \hline 00001001 \end{array}$$

Practice: Subtract 0101 from 1001.

Learn How To Do the Following:

- Form the two's complement of a hexadecimal integer
- Convert signed binary to decimal
- Convert signed decimal to binary
- Convert signed decimal to hexadecimal
- Convert signed hexadecimal to decimal

Ranges of Signed Integers

The highest bit is reserved for the sign. This limits the range:

Storage Type	Range (low–high)	Powers of 2
Signed byte	-128 to +127	-2^7 to $(2^7 - 1)$
Signed word	-32,768 to +32,767	-2^{15} to $(2^{15} - 1)$
Signed doubleword	-2,147,483,648 to 2,147,483,647	-2^{31} to $(2^{31} - 1)$
Signed quadword	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	-2^{63} to $(2^{63} - 1)$

Practice: What is the largest positive value that may be stored in 20 bits?

Signed integers in 16 bits

- Positive integers:
from 0 ~ 32767
- Negative integers:
from -1 ~ -32768

Decimal	Binary
+32767	0111 1111 1111 1111
+32766	0111 1111 1111 1110
...	...
+2	0000 0000 0000 0010
+1	0000 0000 0000 0001
0	0000 0000 0000 0000
-1	1111 1111 1111 1111
-2	1111 1111 1111 1110
...	...
-32766	1000 0000 0000 0010
-32767	1000 0000 0000 0001
-32768	1000 0000 0000 0000

Character Storage

- Character sets
 - Standard ASCII (0 – 127)
 - Extended ASCII (0 – 255)
 - ANSI (0 – 255)
 - Unicode (0 – 65,535)
- Null-terminated String
 - Array of characters followed by a *null byte*
- Using the ASCII table
 - back inside cover of book
 - N : 4E₁₆ , n: 6E₁₆
 - Only one bit different for N and n.

Numeric Data Representation

- pure binary
 - can be calculated directly
- ASCII binary
 - string of digits: "01010101"
- ASCII decimal
 - string of digits: "65"
- ASCII hexadecimal
 - string of digits: "9C"

next: Boolean Operations

What's Next

- Welcome to Assembly Language
- Virtual Machine Concept
- Data Representation
- **Boolean Operations**

Boolean Operations

- Boolean Algebra
 - NOT
 - AND
 - OR
- Operator Precedence
- Truth Tables

Boolean Algebra

- Based on **symbolic logic**, designed by George Boole
- Boolean expressions created from:
 - NOT, AND, OR

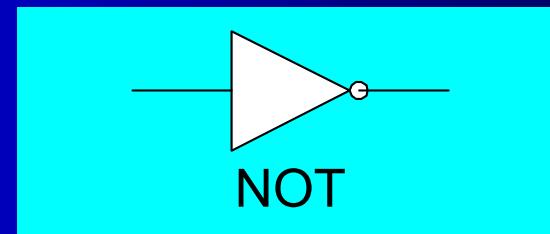
Expression	Description
$\neg X$	NOT X
$X \wedge Y$	X AND Y
$X \vee Y$	X OR Y
$\neg X \vee Y$	(NOT X) OR Y
$\neg(X \wedge Y)$	NOT (X AND Y)
$X \wedge \neg Y$	X AND (NOT Y)

NOT

- Inverts (reverses) a boolean value
- Truth table for Boolean NOT operator:

X	$\neg X$
F	T
T	F

Digital gate diagram for NOT:

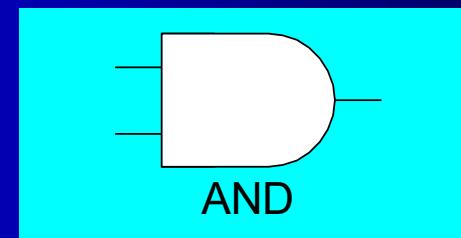


AND

- Truth table for Boolean AND operator:

X	Y	$X \wedge Y$
F	F	F
F	T	F
T	F	F
T	T	T

Digital gate diagram for AND:

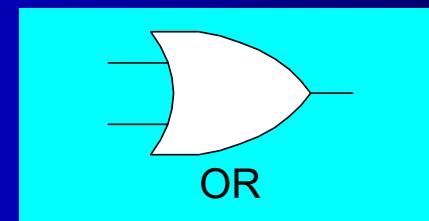


OR

- Truth table for Boolean OR operator:

X	Y	$X \vee Y$
F	F	F
F	T	T
T	F	T
T	T	T

Digital gate diagram for OR:



Operator Precedence

- Examples showing the order of operations:

Expression	Order of Operations
$\neg X \vee Y$	NOT, then OR
$\neg(X \vee Y)$	OR, then NOT
$X \vee (Y \wedge Z)$	AND, then OR

Truth Tables (1 of 3)

- A Boolean function has one or more Boolean inputs, and returns a single Boolean output.
- A truth table shows all the inputs and outputs of a Boolean function

Example: $\neg X \vee Y$

X	$\neg X$	Y	$\neg X \vee Y$
F	T	F	T
F	T	T	T
T	F	F	F
T	F	T	T

Truth Tables (2 of 3)

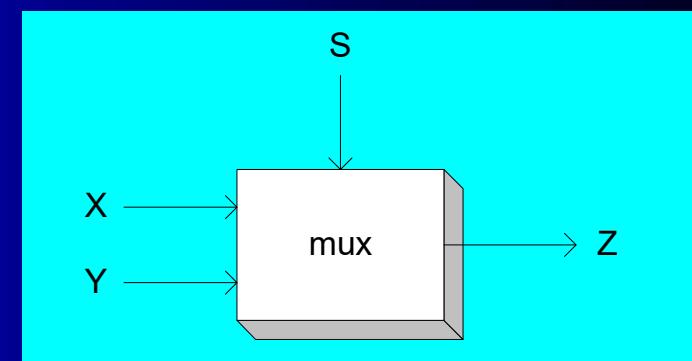
- Example: $X \wedge \neg Y$

X	Y	$\neg Y$	$X \wedge \neg Y$
F	F	T	F
F	T	F	F
T	F	T	T
T	T	F	F

Truth Tables (3 of 3)

- Example: $(Y \wedge S) \vee (X \wedge \neg S)$

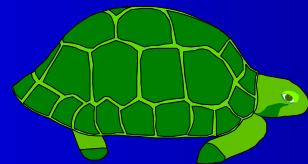
X	Y	S	$Y \wedge S$	$\neg S$	$X \wedge \neg S$	$(Y \wedge S) \vee (X \wedge \neg S)$
F	F	F	F	T	F	F
F	T	F	F	T	F	F
T	F	F	F	T	T	T
T	T	F	F	T	T	T
F	F	T	F	F	F	F
F	T	T	T	F	F	T
T	F	T	F	F	F	F
T	T	T	T	F	F	T



Two-input multiplexer

Summary

- Assembly language helps you learn how software is constructed at the lowest levels
- Assembly language has a one-to-one relationship with machine language
- Each layer in a computer's architecture is an abstraction of a machine
 - layers can be hardware or software
 - Boolean expressions are essential to the design of computer hardware and software



54 68 65 20 45 6E 64

What do these numbers represent?

Assembly Language for x86 Processors

7th Edition, Global Edition

Kip Irvine

Chapter 2: x86 Processor Architecture

Slides prepared by the author

Revision date: 1/15/2014

Modified by: Liang, 2016 Spring

Chapter Overview

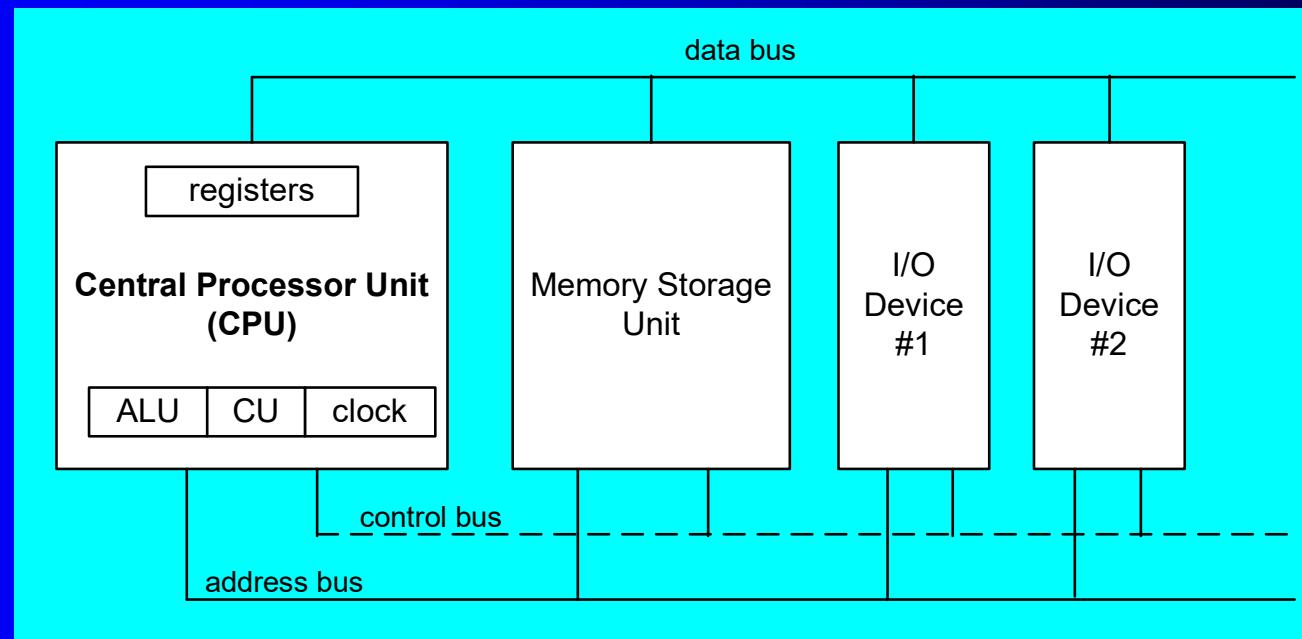
- General Concepts
- IA-32 Processor Architecture
 - Modes of operation
 - Basic execution environment
- IA-32 Memory Management
 - Protected Mode
- 64-bit Processors
- Components of an IA-32 Microcomputer
- Input-Output System

General Concepts

- Basic microcomputer design
- Instruction execution cycle
- Reading from memory
- How program runs

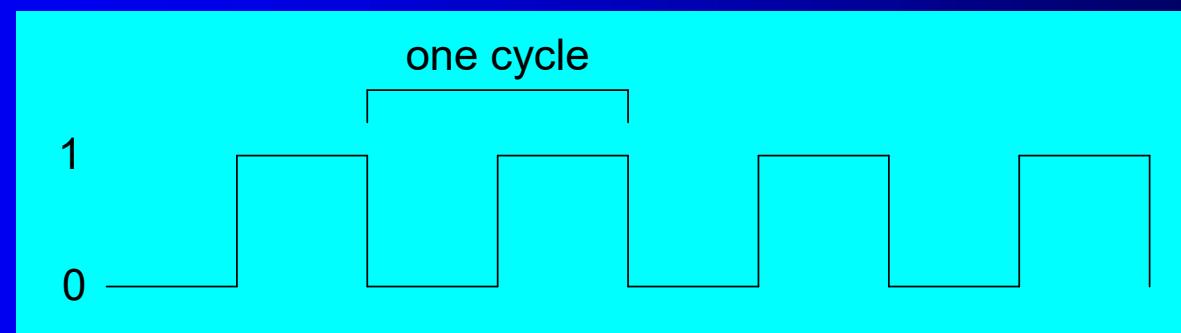
Basic Microcomputer Design

- clock synchronizes CPU operations
- control unit (CU) coordinates sequence of execution steps
- ALU performs arithmetic and bitwise processing



Clock

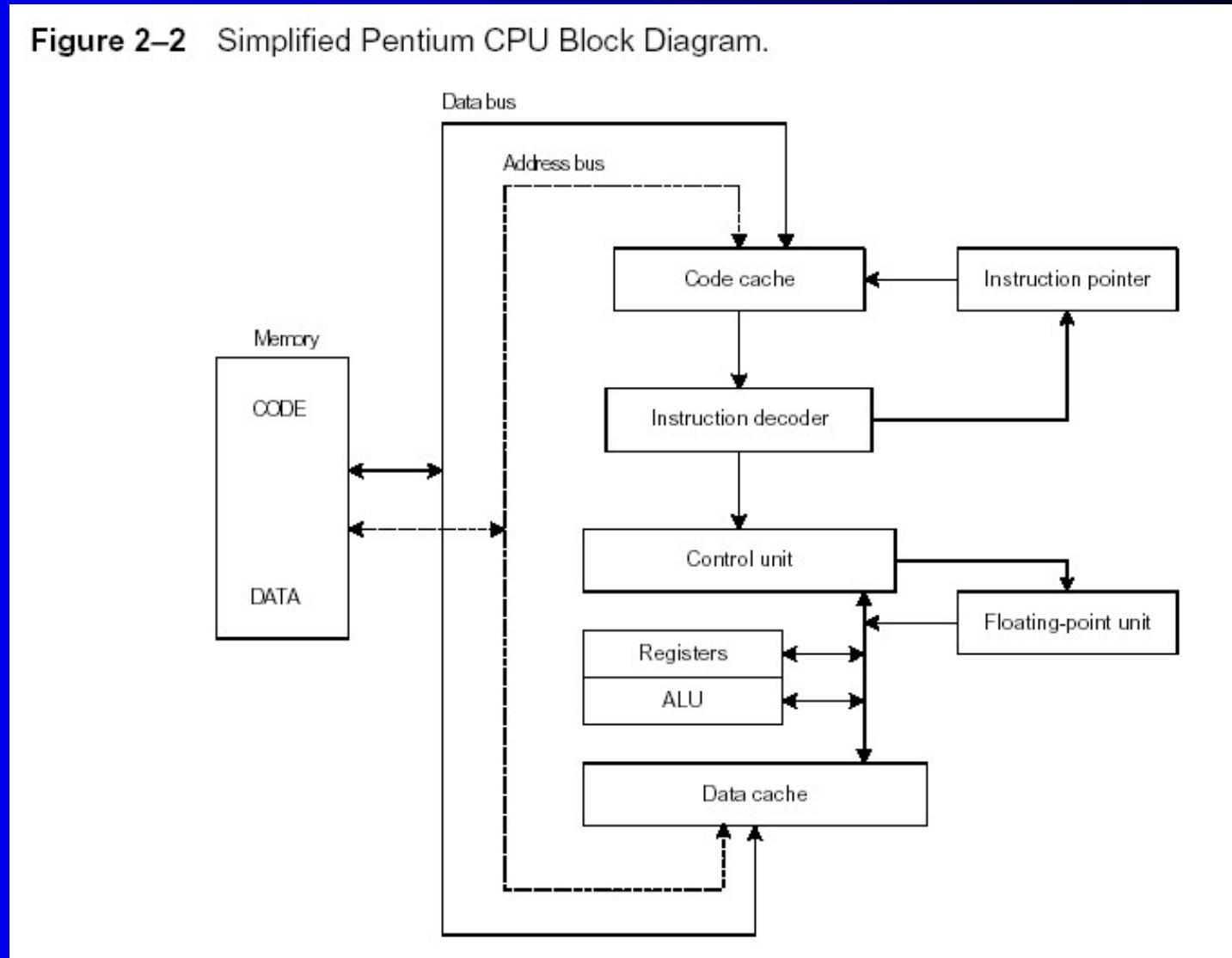
- synchronizes all CPU and BUS operations
- machine (clock) cycle measures time of a single operation
- clock is used to trigger events



Instruction Execution Cycle

- Fetch
- Decode
- Fetch operands
- Execute
- Store output

Figure 2–2 Simplified Pentium CPU Block Diagram.



Reading from Memory

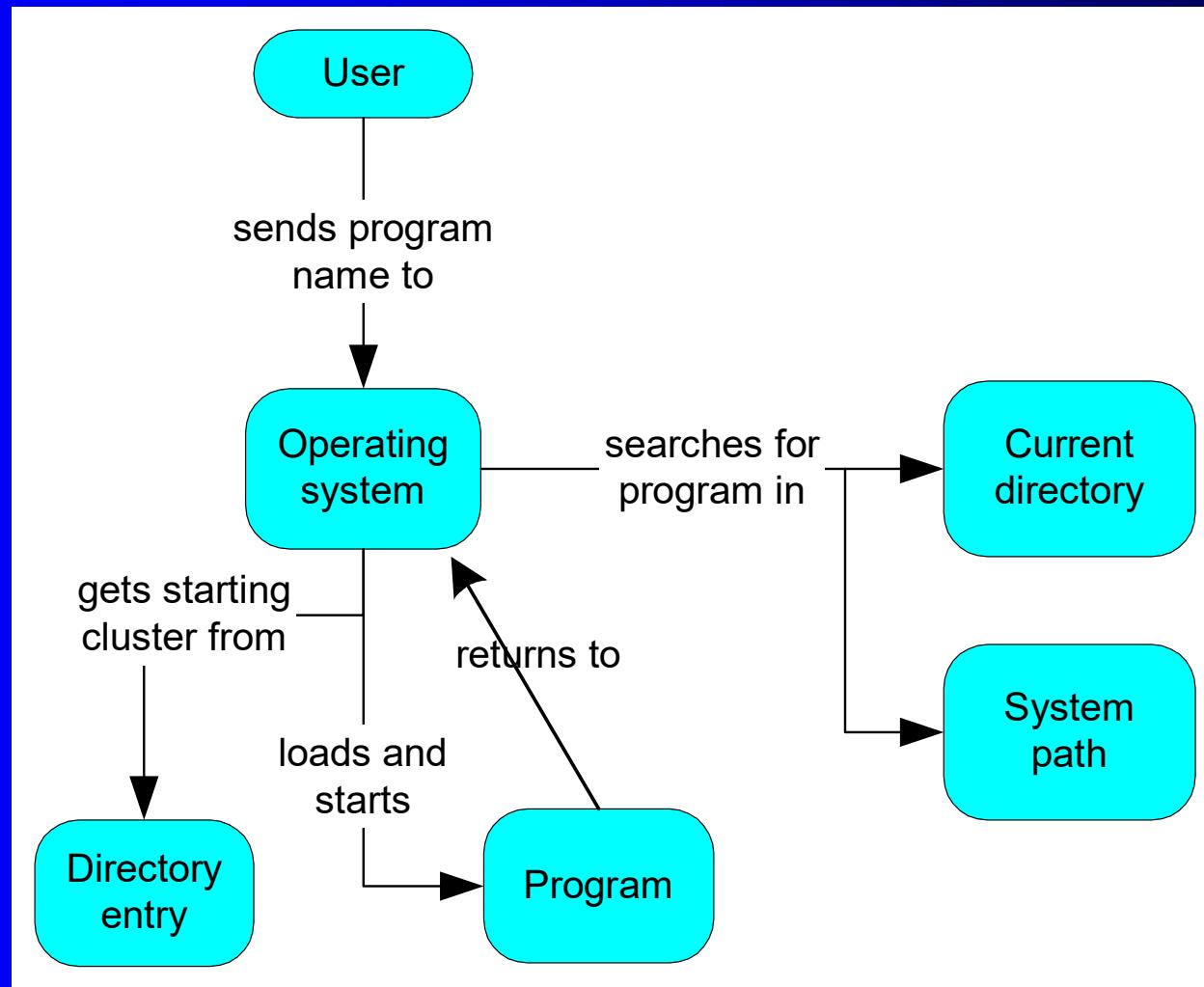
Multiple machine cycles are required when reading from memory, because it responds much more slowly than the CPU. The steps are:

1. Place the address of the value you want to read on the address bus.
2. Assert (changing 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

Cache Memory

- High-speed expensive static RAM both inside and outside the CPU.
 - Level-1 cache: inside the CPU
 - Level-2 cache: outside the CPU
- Cache hit: when data to be read is already in cache memory
- Cache miss: when data to be read is not in cache memory.

How a Program Runs



What's Next

- General Concepts
- IA-32 Processor Architecture
- IA-32 Memory Management
- 64-Bit Processors
- Components of an IA-32 Microcomputer
- Input-Output System

IA-32 Processor Architecture

- Modes of operation
- Basic execution environment

Modes of Operation

- Protected mode
 - native mode (Windows, Linux)
- Real-address mode
 - native MS-DOS
- System management mode
 - power management, system security, diagnostics

- Virtual-8086 mode
 - hybrid of Protected
 - each program has its own 8086 computer

Basic Execution Environment

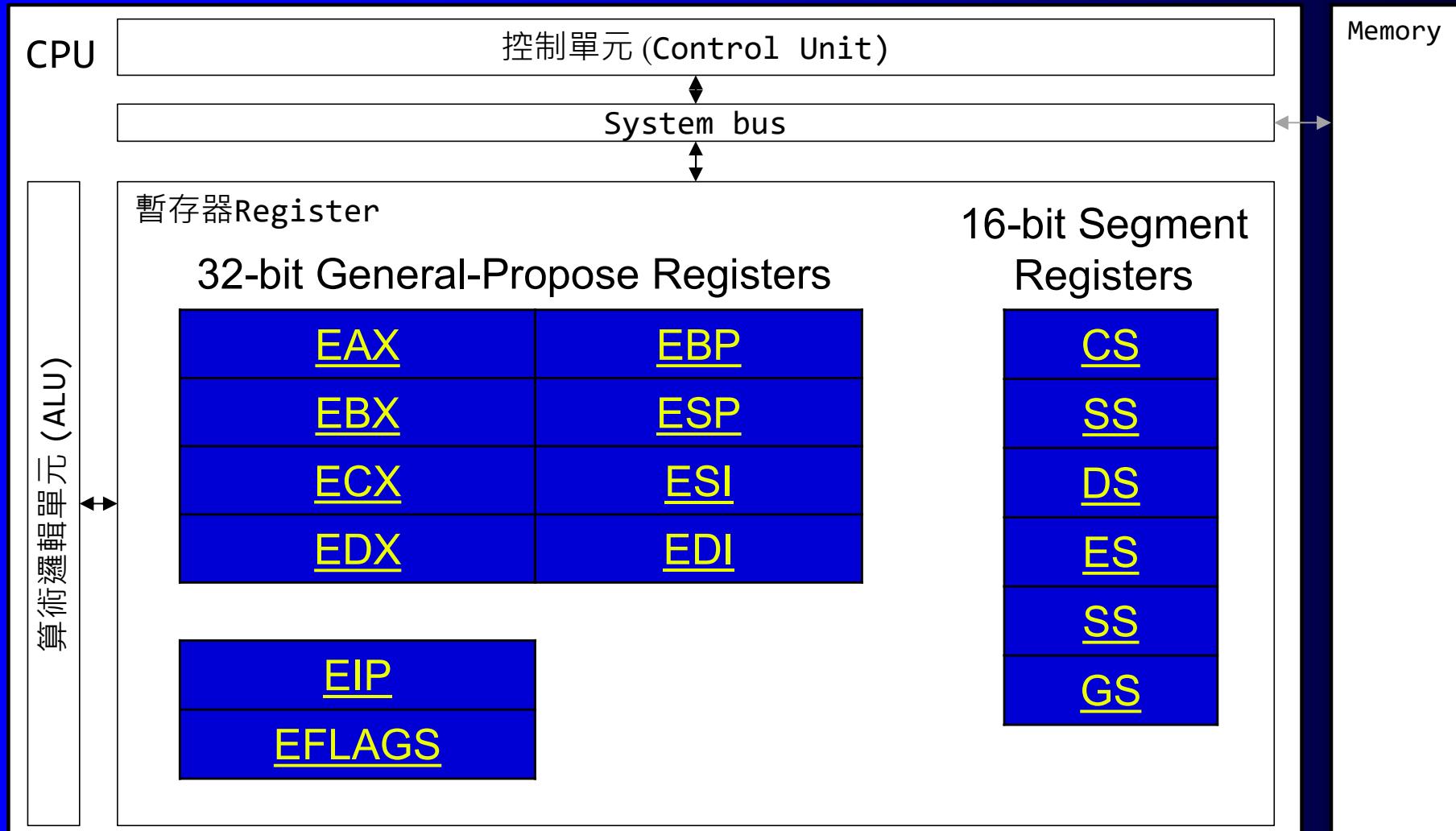
- Addressable memory
- General-purpose registers
 - Index and base registers
 - Specialized register uses
 - Status flags
- Floating-point, MMX, XMM registers

Addressable Memory

- Protected mode
 - 4 GB
 - 32-bit address
- Real-address and Virtual-8086 modes
 - 1 MB space
 - 20-bit address

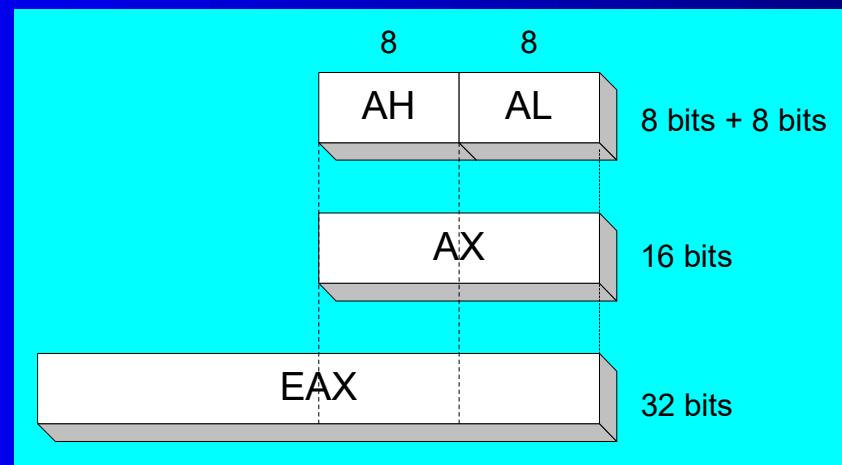
General-Purpose Registers

Named storage locations inside the CPU, optimized for speed.



Accessing Parts of Registers

- Use 8-bit name, 16-bit name, or 32-bit name
- Applies to EAX, EBX, ECX, and EDX



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

Index and Base Registers

- Some registers have only a 16-bit name for their lower half:

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

Some Specialized Register Uses (1 of 2)

- General-Purpose
 - EAX – accumulator
 - ECX – loop counter
 - ESP – stack pointer
 - ESI, EDI – index registers
 - EBP – extended frame pointer (stack)
- Segment
 - CS – code segment
 - DS – data segment
 - SS – stack segment
 - ES, FS, GS - additional segments

Some Specialized Register Uses (2 of 2)

- EIP – instruction pointer
- EFLAGS
 - status and control flags
 - each flag is a single binary bit

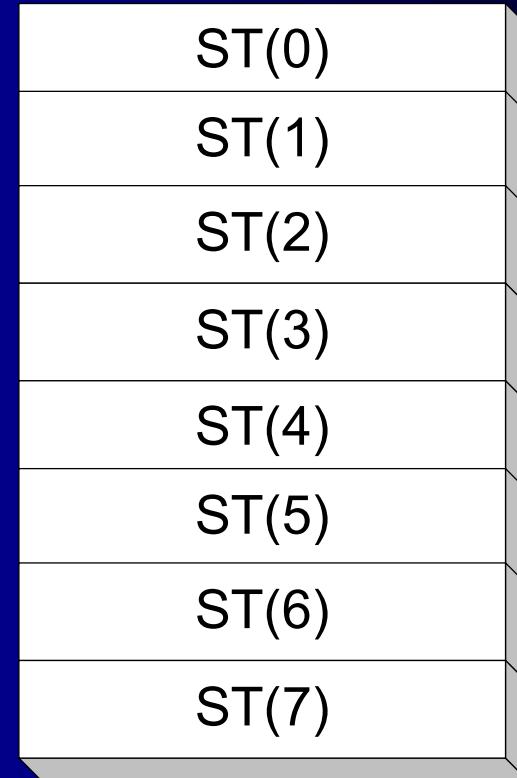
Status Flags

- Carry (bit 0)
 - unsigned arithmetic out of range
- Overflow (bit 11)
 - signed arithmetic out of range
- Sign (bit 7)
 - result is negative
- Zero (bit 6)
 - result is zero
- Auxiliary Carry (bit 4)
 - carry from bit 3 to bit 4
- Parity (bit 2)
 - sum of 1 bits is an even number



Floating-Point, MMX, XMM Registers

- Eight 80-bit floating-point data registers
 - ST(0), ST(1), . . . , ST(7)
 - arranged in a stack
 - used for all floating-point arithmetic
- Eight 64-bit MMX registers
- Eight 128-bit XMM registers for single-instruction multiple-data (SIMD) operations



What's Next

- General Concepts
- IA-32 Processor Architecture
- **IA-32 Memory Management**
- 64-Bit Processors
- Components of an IA-32 Microcomputer
- Input-Output System

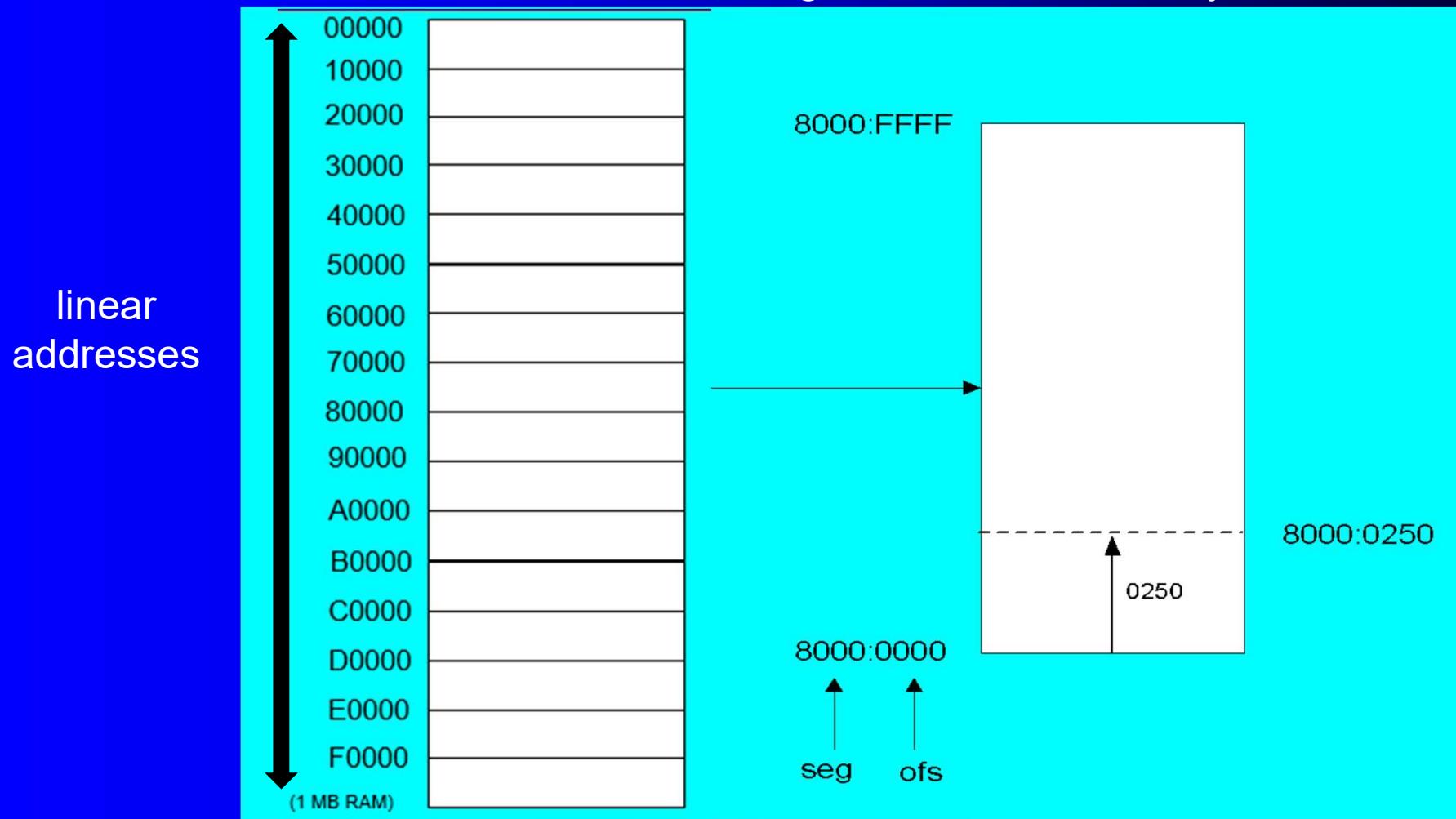
IA-32 Memory Management

- Real-Address mode
- Protected mode

Real-address mode

Real-address and Virtual-8086 modes

- 1 MB space
- 20-bit address
- Single tasking
- Programs can access any area of memory



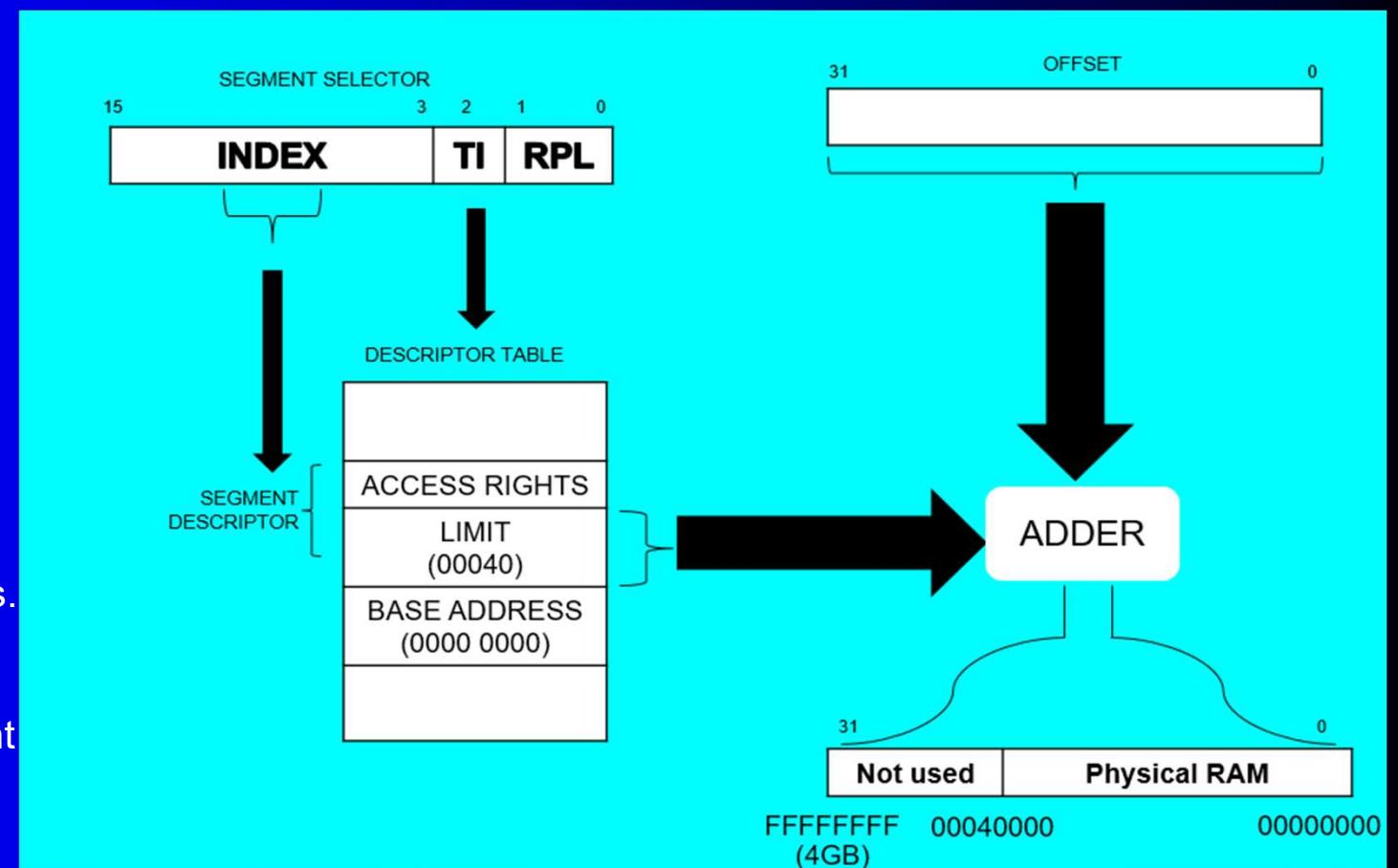
Protected Mode

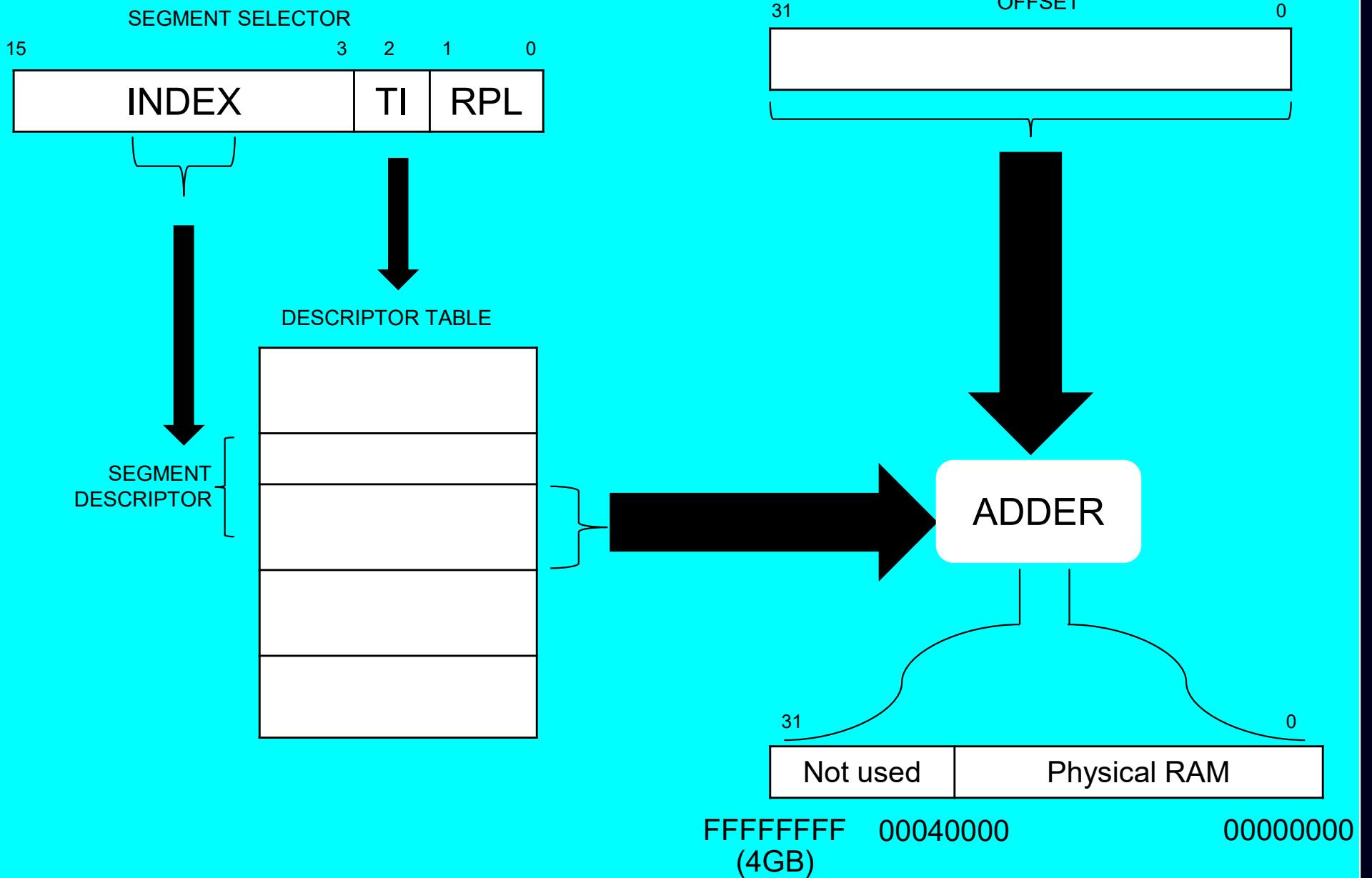
- Designed for multitasking
- Each program assigned a memory partition which is protected from other programs

Protected Mode use segment selector's

1. INDEX is used to get segment descriptor
2. TI is used to decide which kind descriptor table
3. RPL is access right.

Descriptor Table transforms Logical address into linear address. And it include base address、limit and access right. This access right is used to prevent program to assign other program's Memory address.





Protected Mode

- 4 GB addressable RAM
 - (00000000 to FFFFFFFFh)
- Each program assigned a memory partition which is protected from other programs
- Designed for multitasking
- Supported by Linux & MS-Windows

What's Next

- General Concepts
- IA-32 Processor Architecture
- IA-32 Memory Management
- **64-Bit Processors**
- Components of an IA-32 Microcomputer
- Input-Output System

64-Bit Processors

- 64-Bit Operation Modes
 - Compatibility mode – can run existing 16-bit and 32-bit applications (Windows supports only 32-bit apps in this mode)
 - 64-bit mode – Windows 64 uses this
- Basic Execution Environment
 - addresses can be 64 bits (48 bits, in practice)
 - 16 64-bit general purpose registers
 - 64-bit instruction pointer named RIP

64-Bit General Purpose Registers

- 32-bit general purpose registers:
 - EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D, R9D, R10D, R11D, R12D, R13D, R14D, R15D
- 64-bit general purpose registers:
 - RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8, R9, R10, R11, R12, R13, R14, R15

What's Next

- General Concepts
- IA-32 Processor Architecture
- IA-32 Memory Management
- 64-Bit Processors
- **Components of an IA-32 Microcomputer**
- Input-Output System

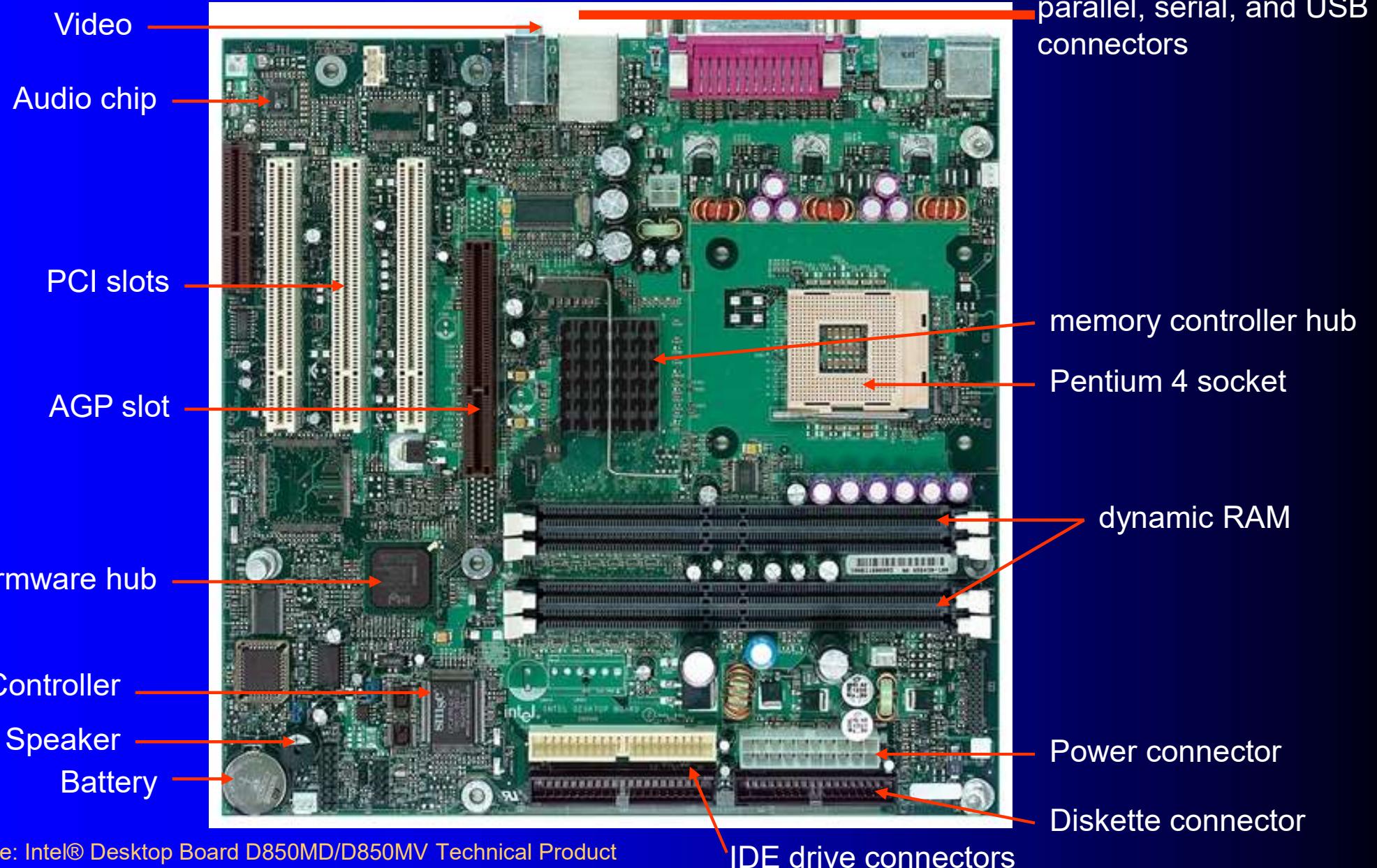
Components of an IA-32 Microcomputer

- Motherboard
- Video output
- Memory
- Input-output ports

Motherboard

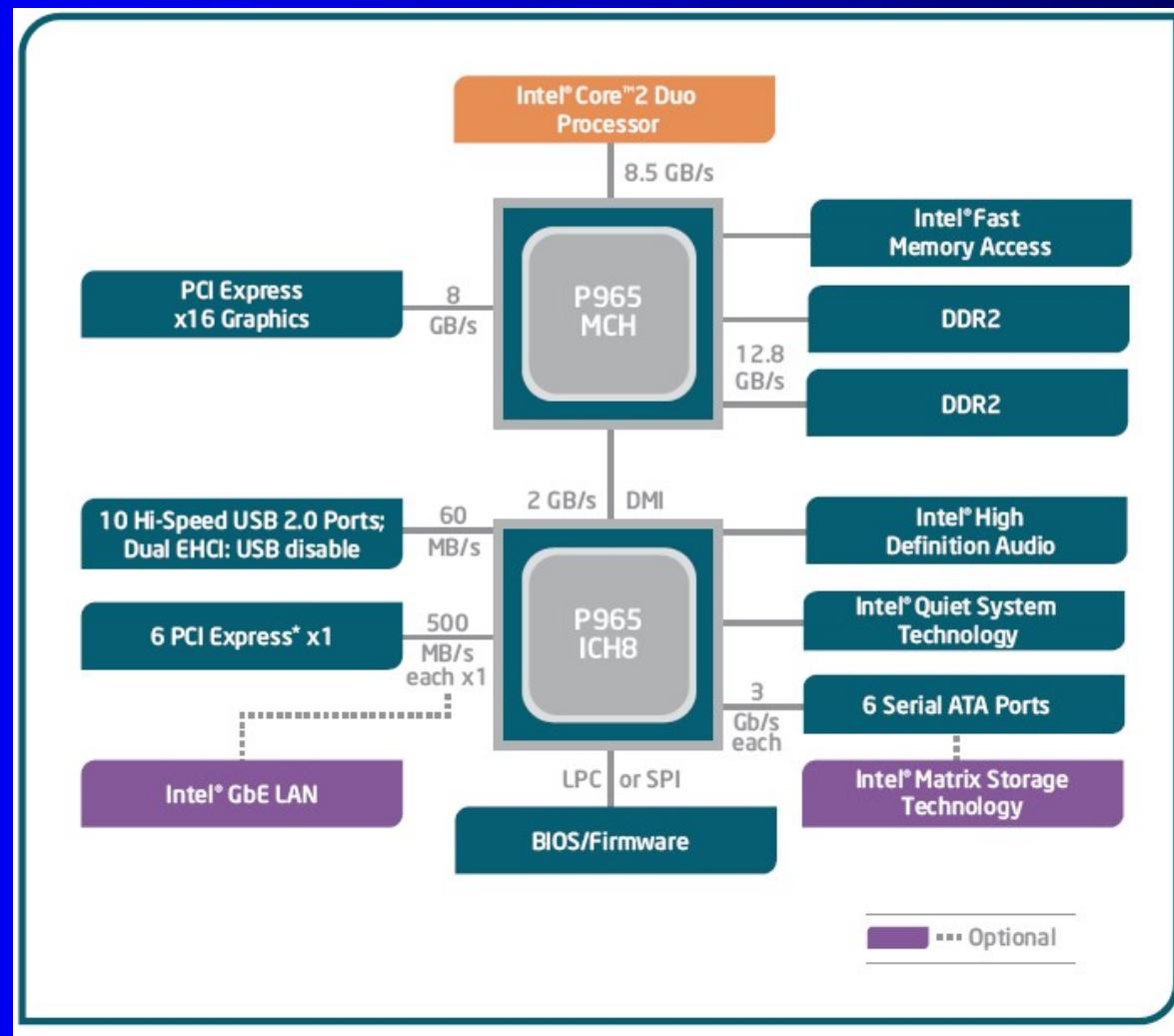
- CPU socket
- External cache memory slots
- Main memory slots
- BIOS chips
- Sound synthesizer chip (optional)
- Video controller chip (optional)
- IDE, parallel, serial, USB, video, keyboard, joystick, network, and mouse connectors
- PCI bus connectors (expansion cards)

Intel D850MD Motherboard



Source: Intel® Desktop Board D850MD/D850MV Technical Product Specification

Intel 965 Express Chipset



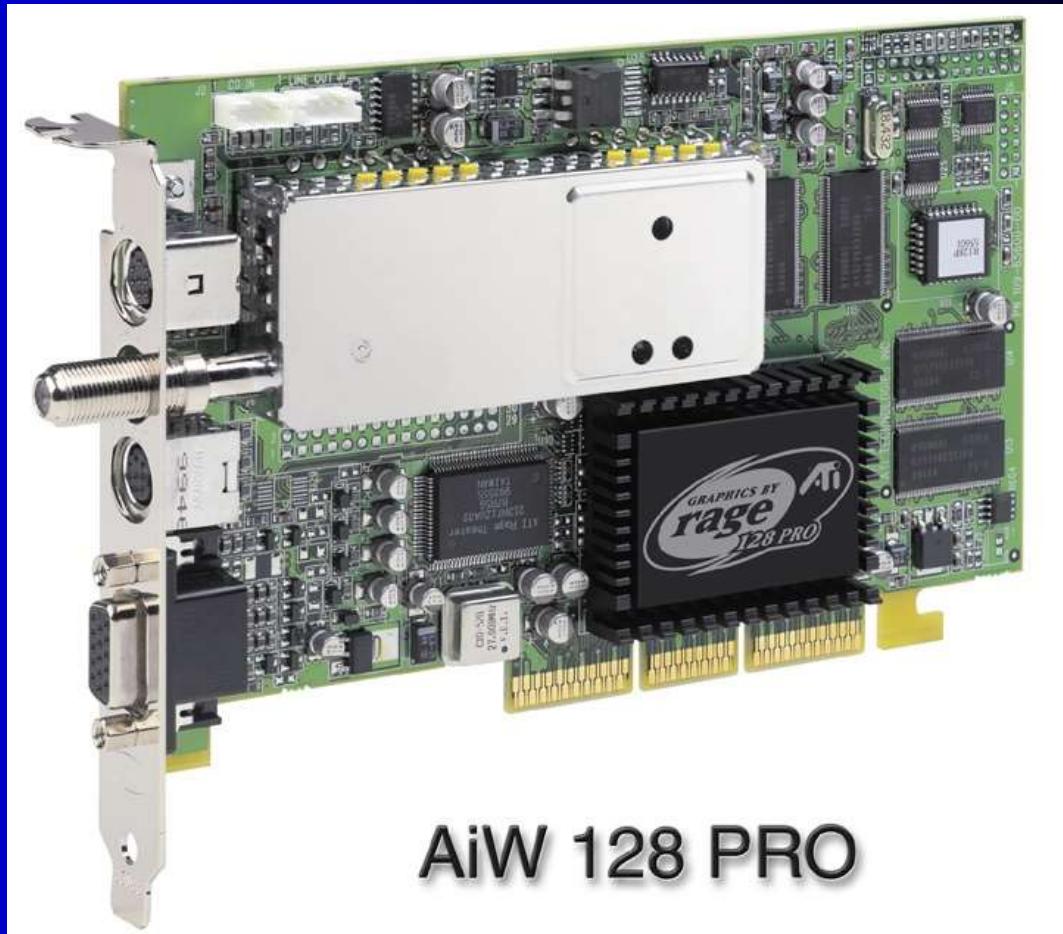
Video Output

- Video controller
 - on motherboard, or on expansion card
 - AGP (accelerated graphics port technology)*
- Video memory (VRAM)
- Video CRT Display
 - uses raster scanning
 - horizontal retrace
 - vertical retrace
- Direct digital LCD monitors
 - no raster scanning required

* This link may change over time.

Sample Video Controller (ATI Corp.)

- 128-bit 3D graphics performance powered by RAGE™ 128 PRO
- 3D graphics performance
- Intelligent TV-Tuner with Digital VCR
- TV-ON-DEMAND™
- Interactive Program Guide
- Still image and MPEG-2 motion video capture
- Video editing
- Hardware DVD video playback
- Video output to TV or VCR



Memory

- ROM
 - read-only memory
- EPROM
 - erasable programmable read-only memory
- Dynamic RAM (DRAM)
 - inexpensive; must be refreshed constantly
- Static RAM (SRAM)
 - expensive; used for cache memory; no refresh required
- Video RAM (VRAM)
 - dual ported; optimized for constant video refresh
- CMOS RAM
 - complimentary metal-oxide semiconductor
 - system setup information
- See: [Intel platform memory](#) (Intel technology brief: link address may change)

Input-Output Ports

- USB (universal serial bus)
 - intelligent high-speed connection to devices
 - up to 12 megabits/second
 - USB hub connects multiple devices
 - *enumeration*: computer queries devices
 - supports *hot* connections
- Parallel
 - short cable, high speed
 - common for printers
 - bidirectional, parallel data transfer
 - Intel 8255 controller chip

Input-Output Ports (cont)

- Serial
 - RS-232 serial port
 - one bit at a time
 - uses long cables and modems
 - 16550 UART (universal asynchronous receiver transmitter)
 - programmable in assembly language

Device Interfaces

- ATA host adapters
 - intelligent drive electronics (hard drive, CDROM)
- SATA (Serial ATA)
 - inexpensive, fast, bidirectional
- FireWire
 - high speed (800 MB/sec), many devices at once
- Bluetooth
 - small amounts of data, short distances, low power usage
- Wi-Fi (wireless Ethernet)
 - IEEE 802.11 standard, faster than Bluetooth

What's Next

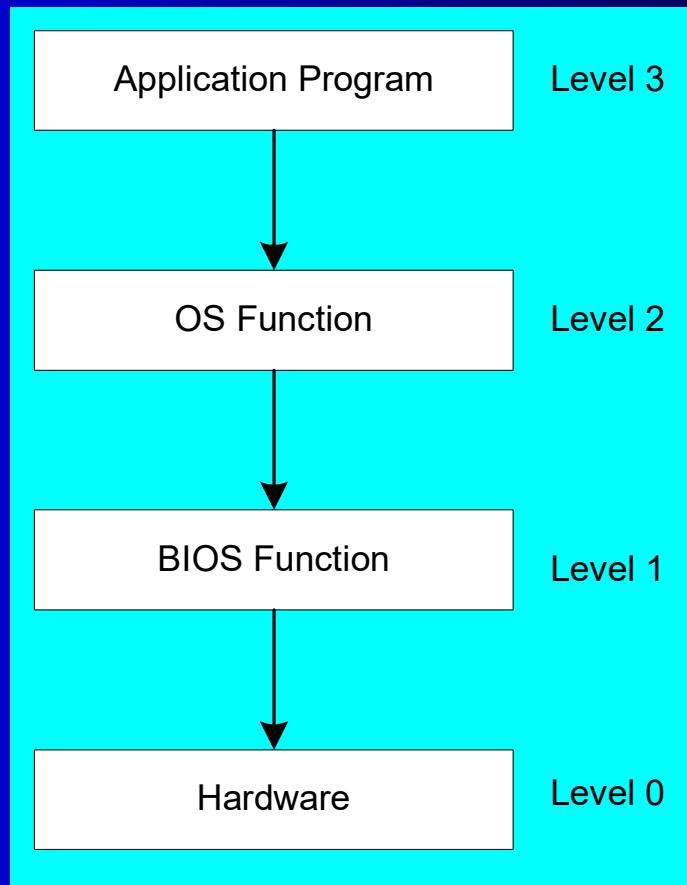
- General Concepts
- IA-32 Processor Architecture
- IA-32 Memory Management
- Components of an IA-32 Microcomputer
- **Input-Output System**

Levels of Input-Output

- Level 3: High-level language function
 - examples: C++, Java
 - portable, convenient, not always the fastest
- Level 2: Operating system
 - Application Programming Interface (API)
 - extended capabilities, lots of details to master
- Level 1: BIOS
 - drivers that communicate directly with devices
 - OS security may prevent application-level code from working at this level

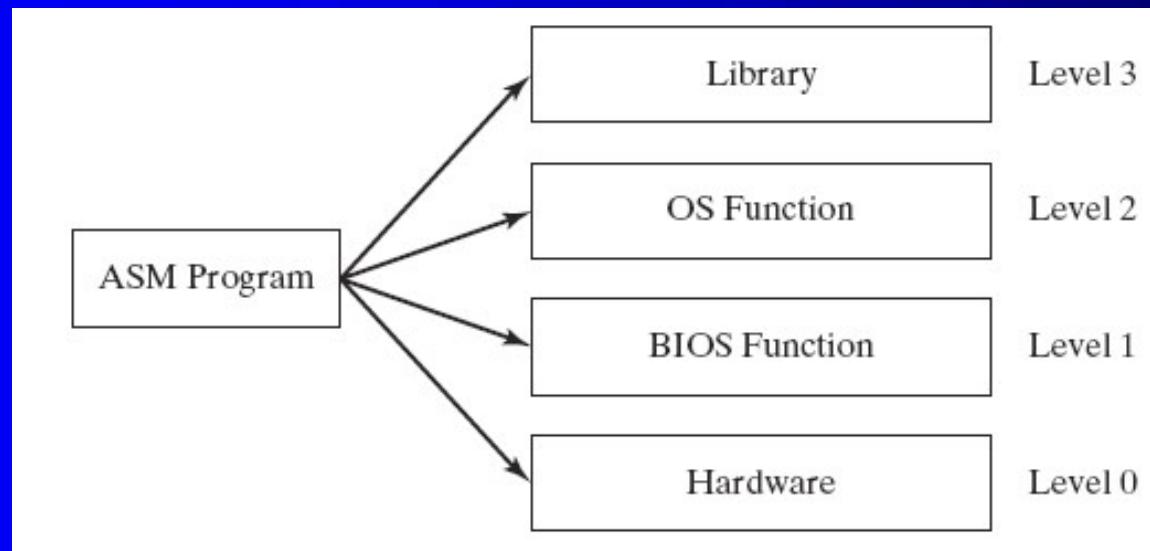
Displaying a String of Characters

When a HLL program displays a string of characters, the following steps take place:



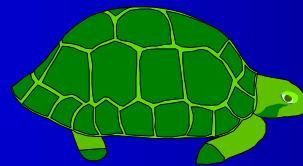
Programming levels

Assembly language programs can perform input-output at each of the following levels:



Summary

- Central Processing Unit (CPU)
- Arithmetic Logic Unit (ALU)
- Instruction execution cycle
- Multitasking
- Floating Point Unit (FPU)
- Complex Instruction Set
- Real mode and Protected mode
- Motherboard components
- Memory types
- Input/Output and access levels



42696E617279

What does this say?

Assembly Language for x86 Processors

7th Edition , Global Edition

Kip Irvine

Chapter 3: Assembly Language Fundamentals

Slides prepared by the author

Revision date: 1/15/2014

Modified by: Liang, 2016 Spring

(c) Pearson Education, 2015. All rights reserved. You may modify and copy this slide show for your personal use, or for use in the classroom, as long as this copyright statement, the author's name, and the title are not changed.

Chapter Overview

- Basic Elements of Assembly Language
- Example: Adding and Subtracting Integers
 - Suggested Coding Standards
- 64-Bit Programming
- Assembling, Linking, and Running Programs
- Defining Data
- Symbolic Constants

Basic Elements of Assembly Language

- Example
- Integer constants
- Character and string constants
- Reserved words and identifiers
- Directives and instructions
 - Labels
 - Mnemonics and Operands
 - Comments
 - Instruction Format Examples

Add and Subtract

Directives

```
TITLE Add and Subtract, Version 2          (AddSub2r.asm)
INCLUDE Irvine32.inc
.data
val1 DWORD 10000h
val2 DWORD 40000h
val3 DWORD 20000h
myStr BYTE "Hello!"
finalVal DWORD ?
.code
Main PROC
```

Integer constants

Character and String Constants

Reserved words and Identifiers

Label

```
:
L1: mov eax, val1           ; get first value
    add eax, val2           ; add second value
    sub eax, val3           ; subtract third value
    mov finalVal, eax
```

Comment

Mnemonics and Operands

Instructions

```
:
Main ENDP
END main
```

Integer Constants

- Optional leading + or – sign
- binary, decimal, hexadecimal, or octal digits
- Common radix characters:
 - h – hexadecimal
 - d – decimal
 - b – binary
 - r – encoded real

Examples: 30d, 6Ah, 42, 1101b

Hexadecimal beginning with letter: 0A5h

Integer Expressions

- Operators and precedence levels:

Operator	Name	Precedence Level
()	parentheses	1
+ , -	unary plus, minus	2
* , /	multiply, divide	3
MOD	modulus	3
+ , -	add, subtract	4

- Examples:

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

Character and String Constants

- Enclose character in single or double quotes
 - 'A', "x"
 - ASCII character = 1 byte
- Enclose strings in single or double quotes
 - "ABC"
 - 'xyz'
 - Each character occupies a single byte
- Embedded quotes:
 - 'Say "Goodnight," Gracie'

Reserved Words and Identifiers

- Reserved words cannot be used as identifiers
 - Instruction mnemonics, directives, type attributes, operators, predefined symbols
 - See MASM reference in Appendix A
 - **Examples:** DATA, PROC, ADD, SUB
- Identifiers
 - 1-247 characters, including digits
 - not case sensitive
 - first character must be a letter, _, @, ?, or \$
 - Examples: val1, finalVal, DumpRegs

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
 - Examples:
 - **.data**
 - **main PROC**
 - **main ENDP**
- Different assemblers have different directives
 - NASM not the same as MASM

Instructions

- 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)

Examples:

mov eax,val1

add eax,val2

mov finalVal,eax

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

Labels

- Act as place markers
 - marks the address (offset) of code and data
- Follow identifier rules
- Data label
 - must be unique
 - example: **myArray** (not followed by colon)
- 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
 - constant expression
 - register
 - memory (data label)

Constants and constant expressions are often called immediate values

Comments

- Comments are good!
 - explain the program's purpose
 - when it was written, and by whom
 - revision information
 - tricky coding techniques
 - application-specific explanations
- Single-line comments
 - begin with semicolon (;)
- Multi-line comments
 - begin with COMMENT directive and a programmer-chosen character
 - end with the same programmer-chosen character

What's Next

- Basic Elements of Assembly Language
- **Example: Adding and Subtracting Integers**
- Assembling, Linking, and Running Programs
- Defining Data
- Symbolic Constants
- 64-Bit Programming

Example: Adding and Subtracting Integers

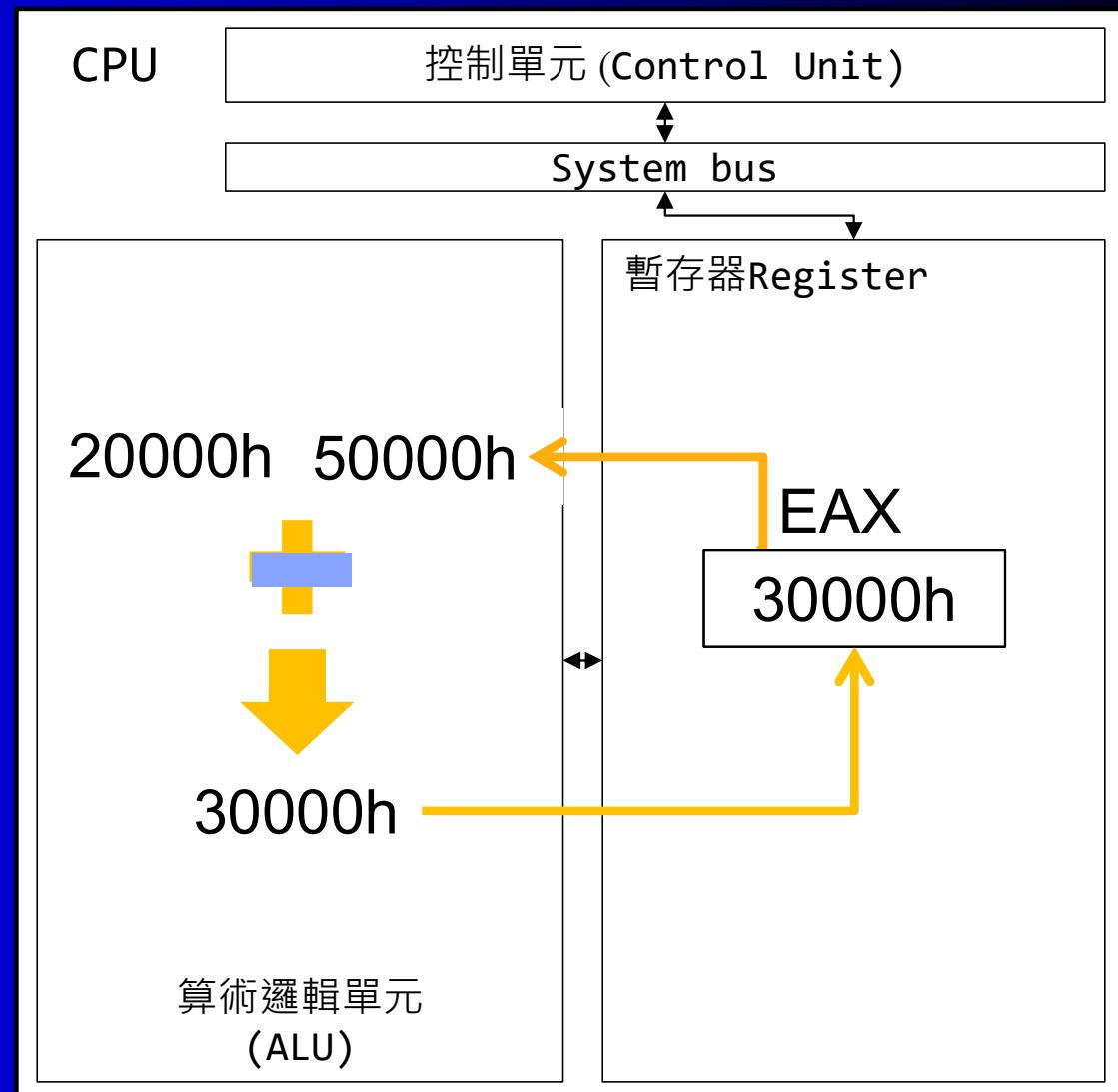
```
TITLE Add and Subtract          (AddSub.asm)
; This program adds and subtracts 32-bit integers.

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
```

Example: Adding and Subtracting Integers

```
.....  
.code  
main PROC  
    mov eax,10000h  
    add eax,40000h  
    sub eax,20000h  
    call DumpRegs  
    exit  
main ENDP  
END main
```



Example Output

Showing registers and flags in the debugger:

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

Suggested Coding Standards (1 of 3)

- Some approaches to capitalization
 - capitalize nothing
 - capitalize everything
 - capitalize all reserved words, including instruction mnemonics and register names
 - capitalize only directives and operators
- Other suggestions
 - descriptive identifier names
 - spaces surrounding arithmetic operators
 - blank lines between procedures

Suggested Coding Standards (2 of 3)

- Indentation and spacing
 - code and data labels – no indentation
 - executable instructions – indent 4-5 spaces
 - comments: right side of page, aligned vertically
 - 1-3 spaces between instruction and its operands
 - ex: mov ax,bx
 - 1-2 blank lines between procedures

Suggested Coding Standards (3 of 3)

No Indentation	Capitalizes []	<pre>TITLE Add and Subtract, Version 2 (a.asm) INCLUDE Irvine32.inc .data val1 DWORD 10000h val2 DWORD 40000h val3 DWORD 20000h myStr BYTE "Hello!" finalVal DWORD ? .code Main PROC . . L1: mov eax, val1 ; get first value add eax, val2 ; add second value sub eax, val3 ; subtract third value mov finalVal, eax . . Main ENDP END main</pre>	Descriptive Identifier Names *avoid undescriptive like i,j, and k and don't be too long*
Label : No Indentation			Comments are good (instruction and command align with tab)
<u>Instructions</u> Indent 4 space		<pre>L1: mov eax, val1 ; get first value add eax, val2 ; add second value sub eax, val3 ; subtract third value mov finalVal, eax . . Main ENDP END main</pre>	; get first value ; add second value ; subtract third value
			Space between instruction and its operand(s)

What's Next

- Basic Elements of Assembly Language
- Example: Adding and Subtracting Integers
- **64-Bit Programming**
- Assembling, Linking, and Running Programs
- Defining Data
- Symbolic Constants

64-Bit Programming

- MASM supports 64-bit programming, although the following directives are not permitted:
 - INVOKE, ADDR, .model, .386, .stack
 - (Other non-permitted directives will be introduced in later chapters)

64-Bit Version of AddTwoSum

64-Bit Version

```
ExitProcess PROTO  
.data  
    sum DWORD 0  
.code  
main PROC  
    mov eax, 5  
    add eax, 6  
    mov sum, eax  
  
    mov ecx, 0  
    call ExitProcess  
main ENDP  
END
```

32-Bit Version

```
.386  
.model flat,stdcall  
.stack 4096  
ExitProcess PROTO,  
    dwExitCode:DWORD  
.data  
    sum DWORD 0  
.code  
main PROC  
    mov eax, 5  
    add eax, 6  
    mov sum, eax  
  
    INVOKE ExitProcess,0  
main ENDP  
END
```

INCLUDE
Irvine32.inc

Things to Notice About the Previous Slide

- The following lines are not needed:
.386
.model flat,stdcall
.stack 4096
- INVOKE is not supported.
- CALL instruction cannot receive arguments
- Use 64-bit registers when possible

What's Next

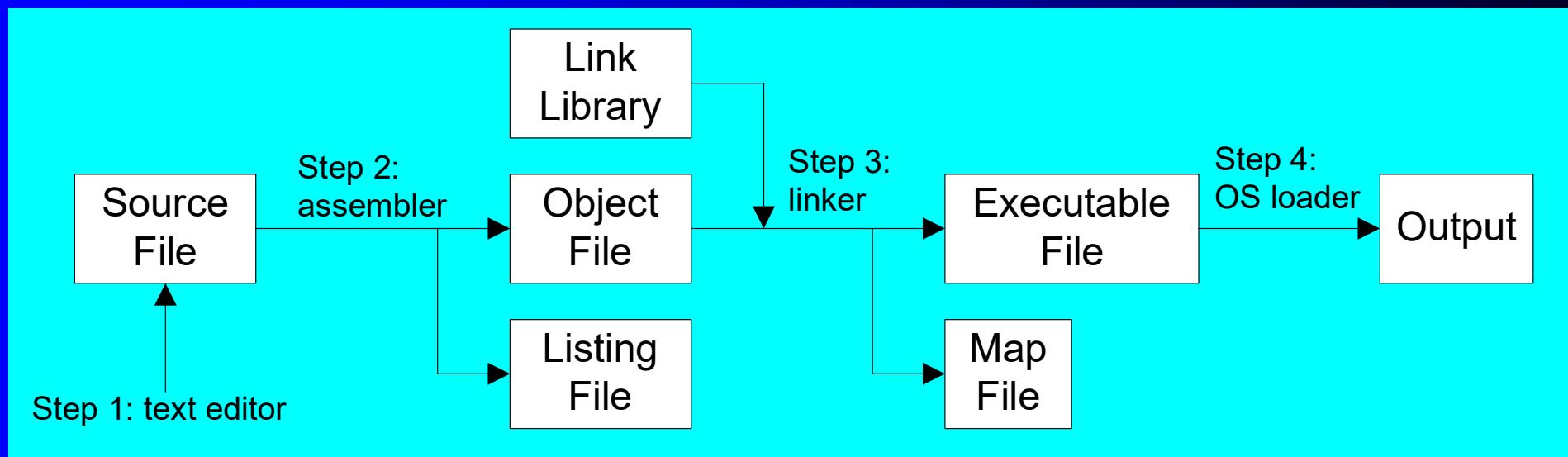
- Basic Elements of Assembly Language
- Example: Adding and Subtracting Integers
- 64-Bit Programming
- **Assembling, Linking, and Running Programs**
- Defining Data
- Symbolic Constants

Assembling, Linking, and Running Programs

- Assemble-Link-Execute Cycle

Assemble-Link Execute Cycle

- The following diagram describes the steps from creating a source program through executing the compiled program.
- If the source code is modified, Steps 2 through 4 must be repeated.



What's Next

- Basic Elements of Assembly Language
- Example: Adding and Subtracting Integers
- 64-Bit Programming
- Assembling, Linking, and Running Programs
- **Defining Data**
- Symbolic Constants

Defining Data

- Intrinsic Data Types
- Data Definition Statement
- Defining BYTE and SBYTE Data
- Defining WORD and SWORD Data
- Defining DWORD and SDWORD Data
- Defining QWORD, TBYTE, Real Number Data
- Little Endian Order
- Adding Variables to the AddSub Program
- Declaring Uninitialized Data

Intrinsic Data Types (1 of 2)

- BYTE, SBYTE
 - 8-bit unsigned integer; 8-bit signed integer
- WORD, SWORD
 - 16-bit unsigned & signed integer
- DWORD, SDWORD
 - 32-bit unsigned & signed integer
- QWORD
 - 64-bit integer
- TBYTE
 - 80-bit integer

Intrinsic Data Types (2 of 2)

- REAL4
 - 4-byte IEEE short real
- REAL8
 - 8-byte IEEE long real
- REAL10
 - 10-byte IEEE extended real

Data Definition Statement

- A data definition statement sets aside storage in memory for a variable.
- May optionally assign a name (label) to the data
- Syntax:

[name] directive initializer [,initializer] . . .

value1 BYTE 10

The diagram illustrates the mapping between the general syntax of a data definition statement and a specific example. Three yellow arrows point from the components of the syntax to the corresponding tokens in the code:

- An arrow points from "[name]" to "value1".
- An arrow points from "directive" to "BYTE".
- An arrow points from "initializer" to "10".

- All initializers become binary data in memory

Defining BYTE and SBYTE Data

Each of the following defines a single byte of storage:

```
value1 BYTE 'A'          ; character constant  
value2 BYTE 0            ; smallest unsigned byte  
value3 BYTE 255          ; largest unsigned byte  
value4 SBYTE -128         ; smallest signed byte  
value5 SBYTE +127         ; largest signed byte  
value6 BYTE ?             ; uninitialized byte
```

- MASM does not prevent you from initializing a BYTE with a negative value, but it's considered poor style.
- If you declare a SBYTE variable, the Microsoft debugger will automatically display its value in decimal with a leading sign.

Defining Byte Arrays

Examples that use multiple initializers:

```
list1 BYTE 10,20,30,40  
list2 BYTE 10,20,30,40  
          BYTE 50,60,70,80  
          BYTE 81,82,83,84  
list3 BYTE ?,32,41h,00100010b  
list4 BYTE 0Ah,20h,'A',22h
```

Defining Strings (1 of 3)

- A string is implemented as an array of characters
 - For convenience, it is usually enclosed in quotation marks
 - It often will be null-terminated
- Examples:

```
str1 BYTE "Enter your name",0
str2 BYTE 'Error: halting program',0
str3 BYTE 'A','E','I','O','U'
greeting BYTE "Welcome to the Encryption Demo program "
           BYTE "created by Kip Irvine.",0
```

Defining Strings (2 of 3)

- To continue a single string across multiple lines, end each line with a comma:

```
menu BYTE "Checking Account",0dh,0ah,0dh,0ah,  
        "1. Create a new account",0dh,0ah,  
        "2. Open an existing account",0dh,0ah,  
        "3. Credit the account",0dh,0ah,  
        "4. Debit the account",0dh,0ah,  
        "5. Exit",0ah,0ah,  
        "Choice> ",0
```

Defining Strings (3 of 3)

- End-of-line character sequence:
 - 0Dh = carriage return
 - 0Ah = line feed

```
str1 BYTE "Enter your name:      ",0Dh,0Ah  
        BYTE "Enter your address: ",0  
  
newLine BYTE 0Dh,0Ah,0
```

Idea: Define all strings used by your program in the same area of the data segment.

Using the DUP Operator

- Use DUP to allocate (create space for) an array or string. Syntax: *counter* DUP (*argument*)
- *Counter* and *argument* must be constants or constant expressions

```
var1 BYTE 20 DUP(0)          ; 20 bytes, all equal to zero
var2 BYTE 20 DUP(?)          ; 20 bytes, uninitialized
var3 BYTE 4 DUP("STACK")      ; 20 bytes: "STACKSTACKSTACKSTACK"
var4 BYTE 10,3 DUP(0),20      ; 5 bytes
```

Defining WORD and SWORD Data

- Define storage for 16-bit integers
 - or double characters
 - single value or multiple values

```
word1  WORD   65535          ; largest unsigned value
word2  SWORD  -32768         ; smallest signed value
word3  WORD   ?              ; uninitialized, unsigned
word4  WORD   "AB"           ; double characters
myList WORD   1,2,3,4,5       ; array of words
array  WORD   5 DUP(?)       ; uninitialized array
```

Defining DWORD and SDWORD Data

Storage definitions for signed and unsigned 32-bit integers:

```
val1 DWORD 12345678h ; unsigned
val2 SDWORD -2147483648 ; signed
val3 DWORD 20 DUP(?) ; unsigned array
val4 SDWORD -3,-2,-1,0,1 ; signed array
```

Defining QWORD, TBYTE, Real Data

Storage definitions for quadwords, tenbyte values, and real numbers:

```
quad1 QWORD 1234567812345678h
val1 TBYTE 100000000123456789Ah
rVal1 REAL4 -2.1
rVal2 REAL8 3.2E-260
rVal3 REAL10 4.6E+4096
ShortArray REAL4 20 DUP(0.0)
```

Little Endian Order

- All data types larger than a byte store their individual bytes in reverse order. The least significant byte occurs at the first (lowest) memory address.
- Example:

val1 DWORD 12345678h

0000:	78
0001:	56
0002:	34
0003:	12

Adding Variables to AddSub

```
TITLE Add and Subtract, Version 2          (AddSub2.asm)
; This program adds and subtracts 32-bit unsigned
; integers and stores the sum in a variable.
INCLUDE Irvine32.inc

.data
val1 DWORD 10000h
val2 DWORD 40000h
val3 DWORD 20000h
finalVal DWORD ?

.code
main PROC
    mov eax,val1           ; start with 10000h
    add eax,val2           ; add 40000h
    sub eax,val3           ; subtract 20000h
    mov finalVal,eax        ; store the result (30000h)
    call DumpRegs           ; display the registers
    exit
main ENDP
END main
```

Declaring Uninitialized Data

- Use the `.data?` directive to declare an uninitialized data segment:

```
.data?
```

- Within the segment, declare variables with "?" initializers:

```
smallArray DWORD 10 DUP(?)
```

Advantage: the program's EXE file size is reduced.

What's Next

- Basic Elements of Assembly Language
- Example: Adding and Subtracting Integers
- 64-Bit Programming
- Assembling, Linking, and Running Programs
- Defining Data
- **Symbolic Constants**

Symbolic Constants

- Equal-Sign Directive
- Calculating the Sizes of Arrays and Strings
- EQU Directive
- TEXT EQU Directive

Equal-Sign Directive

- *name = expression*
 - expression is a 32-bit integer (expression or constant)
 - may be redefined
 - *name* is called a symbolic constant
- good programming style to use symbols

```
COUNT = 500  
.  
. .  
mov ax,COUNT
```

Calculating the Size of a Byte Array

- current location counter: \$
 - subtract address of list
 - difference is the number of bytes

```
list BYTE 10,20,30,40  
ListSize = ($ - list)
```

list → 0000	10
0001	20
0002	30
0003	40
\$ → 0004	?

Calculating the Size of a Word Array

Divide total number of bytes by 2 (the size of a word)

```
list WORD 1000h,2000h,3000h,4000h  
ListSize = ($ - list) / 2
```

list → 0000	00
0001	10
0002	00
0003	20
0004	00
0005	30
0006	00
0007	40
\$ → 0008	?

Calculating the Size of a Doubleword Array

Divide total number of bytes by 4 (the size of a doubleword)

```
list DWORD 1,2,3,4  
ListSize = ($ - list) / 4
```

list → 0000	01
0001	00
0002	00
0003	00
0004	02
0005	00
0006	00
0007	00
...	...
000F	00
\$ → 0010	?

EQU Directive

- Define a symbol as either an integer or text expression.
- Cannot be redefined

```
PI EQU <3.1416>
pressKey EQU <"Press any key to continue...",0>
.data
prompt BYTE pressKey
```

TEXTEQU Directive

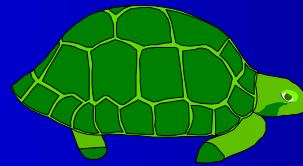
- Define a symbol as either an integer or text expression.
- Called a text macro
- Can be redefined

```
continueMsg TEXTEQU <"Do you wish to continue (Y/N)?>
rowSize = 5
.data
prompt1 BYTE continueMsg
count TEXTEQU %(rowSize * 2)           ; evaluates the expression
setupAL TEXTEQU <mov al,count>

.code
setupAL                                ; generates: "mov al,10"
```

Summary

- Integer expression, character constant
- directive – interpreted by the assembler
- instruction – executes at runtime
- code, data, and stack segments
- source, listing, object, map, executable files
- Data definition directives:
 - BYTE, SBYTE, WORD, SWORD, DWORD, SDWORD, QWORD, TBYTE, REAL4, REAL8, and REAL10
 - DUP operator, location counter (\$)
- Symbolic constant
 - EQU and TEXTEQU



4C61 46696E

Assembly Language for x86 Processors

7th Edition, Global Edition

Kip Irvine

Chapter 4: Data Transfers, Addressing, and Arithmetic

Slides prepared by the author

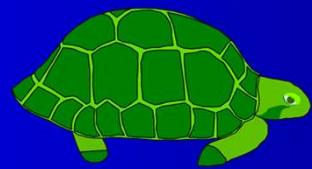
Revision date: 1/15/2014

Chapter Overview

- Data Transfer Instructions
- Addition and Subtraction
- Data-Related Operators and Directives
- Indirect Addressing
- JMP and LOOP Instructions
- 64-Bit Programming

Summary

- Data Transfer
 - MOV – data transfer from source to destination
 - MOVSX, MOVZX, XCHG
- Operand types
 - direct, direct-offset, indirect, indexed
- Arithmetic
 - INC, DEC, ADD, SUB, NEG
 - Sign, Carry, Zero, Overflow flags
- Operators
 - OFFSET, PTR, TYPE, LENGTHOF, SIZEOF, TYPEDEF
- JMP and LOOP – branching instructions



46 69 6E 61 6C

Data Transfer Instructions

- Operand Types
- Instruction Operand Notation
- Direct Memory Operands
- MOV Instruction
- Zero & Sign Extension
- XCHG Instruction
- Direct-Offset Instructions

Operand Types

- Immediate – a constant integer (8, 16, or 32 bits)
 - value is encoded within the instruction
- Register – the name of a register
 - register name is converted to a number and encoded within the instruction
- Memory – reference to a location in memory
 - memory address is encoded within the instruction, or a register holds the address of a memory location

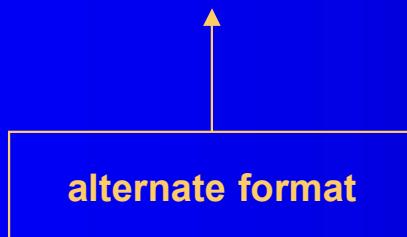
Instruction Operand Notation

Operand	Description
<i>reg8</i>	8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL
<i>reg16</i>	16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP
<i>reg32</i>	32-bit general-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
<i>reg</i>	Any general-purpose register
<i>sreg</i>	16-bit segment register: CS, DS, SS, ES, FS, GS
<i>imm</i>	8-, 16-, or 32-bit immediate value
<i>imm8</i>	8-bit immediate byte value
<i>imm16</i>	16-bit immediate word value
<i>imm32</i>	32-bit immediate doubleword value
<i>reg/mem8</i>	8-bit operand, which can be an 8-bit general register or memory byte
<i>reg/mem16</i>	16-bit operand, which can be a 16-bit general register or memory word
<i>reg/mem32</i>	32-bit operand, which can be a 32-bit general register or memory doubleword
<i>mem</i>	An 8-, 16-, or 32-bit memory operand

Direct Memory Operands

- A direct memory operand is a named reference to storage in memory
- The named reference (label) is automatically dereferenced by the assembler

```
.data  
var1 BYTE 10h  
.code  
mov al, var1 ; AL = 10h  
mov al, [var1] ; AL = 10h
```



MOV Instruction

- Move from source to destination. Syntax:

MOV destination,source

- No more than one memory operand permitted
- Size matched
- CS,DS , EIP,IP ,and immediate value cannot be the destination

```
.data
count BYTE 100
wVal WORD 2
.code
    mov bl,count
    mov ax,wVal
    mov count,al

    mov al,wVal          ; error
    mov ax,count          ; error
    mov eax,count         ; error
```

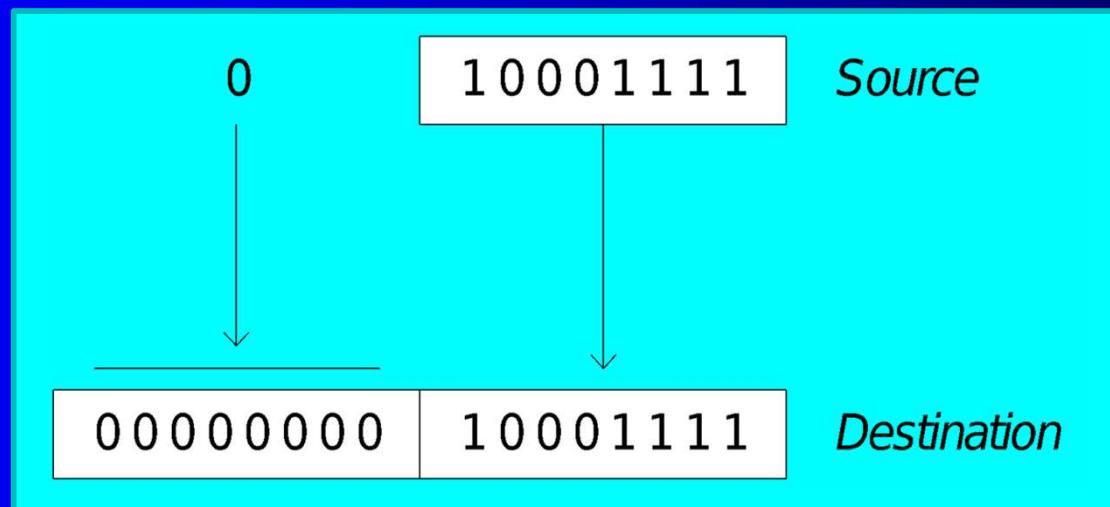
Your turn . . .

Explain why each of the following MOV statements are invalid:

```
.data
bVal    BYTE     100
bVal2   BYTE     ?
wVal    WORD     2
dVal    DWORD    5
.code
    mov ds,45           immediate move to DS not permitted
    mov esi,wVal        size mismatch
    mov eip,dVal        EIP cannot be the destination
    mov 25,bVal          immediate value cannot be destination
    mov bVal2,bVal        memory-to-memory move not permitted
```

Zero Extension

When you copy a smaller value into a larger destination, the MOVZX instruction fills (extends) the upper half of the destination with zeros.

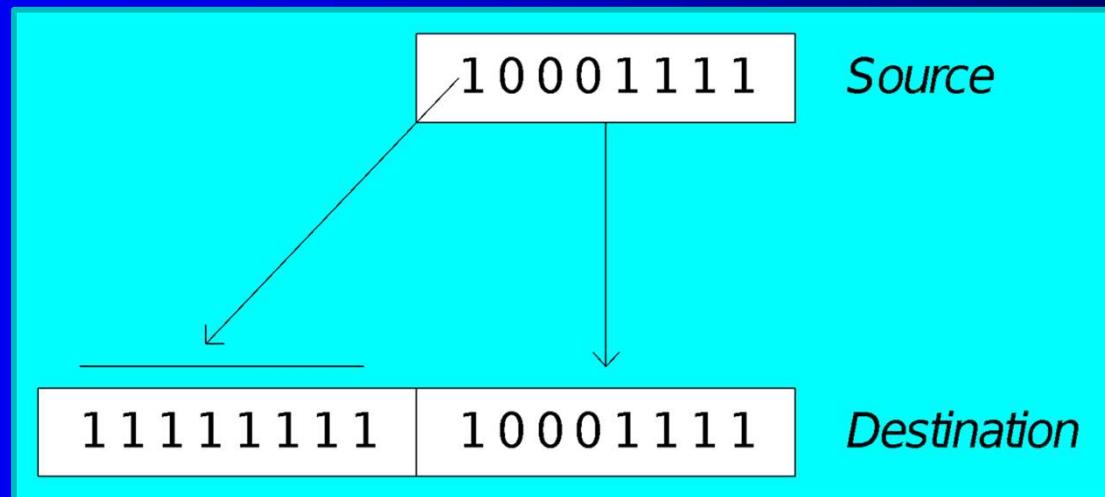


```
mov bl,1000111b  
movzx ax,bl ; zero-extension
```

The destination must be a register.

Sign Extension

The MOVSX instruction fills the upper half of the destination with a copy of the source operand's sign bit.



```
mov bl,1000111b  
movsx ax,bl ; sign extension
```

The destination must be a register.

XCHG Instruction

XCHG exchanges the values of two operands. At least one operand must be a register. No immediate operands are permitted.

```
.data  
var1 WORD 1000h  
var2 WORD 2000h  
.code  
xchg ax,bx      ; exchange 16-bit regs  
xchg ah,al      ; exchange 8-bit regs  
xchg var1,bx    ; exchange mem, reg  
xchg eax,ebx    ; exchange 32-bit regs  
  
xchg var1,var2  ; error: two memory operands
```

Direct-Offset Operands

A constant offset is added to a data label to produce an effective address (EA). The address is dereferenced to get the value inside its memory location.

```
.data  
arrayB BYTE 10h,20h,30h,40h  
.code  
mov al, arrayB+1  
mov al, [arrayB+1]
```



address	value
0x1000h	10h
0x1001h	20h
0x1002h	30h
0x1003h	40h

; AL = 20h
; alternative notation

Q: Why doesn't arrayB+1 produce 11h?

Direct-Offset Operands (cont)

A constant offset is added to a data label to produce an effective address (EA). The address is dereferenced to get the value inside its memory location.

```
.data  
arrayW WORD 1000h,2000h,3000h
```

```
arrayD DWORD 1,2,3,4
```

```
.code
```

```
mov ax,[arrayW+2] ; AX = 2000h  
mov ax,[arrayW+4] ; AX = 3000h  
mov eax,[arrayD+4] ; EAX = 00000002h
```

address	value	address	value
0x1034h	00h	0x1134h	01h
0x1035h	10h	0x1135h	00h
0x1036h	00h	0x1136h	00h
0x1037h	20h	0x1137h	00h
		0x1138h	02h
		0x1139h	00h
		0x113ah	00h
		0x113bh	00h

```
; Will the following statements assemble?  
mov ax,[arrayW-2] ; ??  
mov eax,[arrayD+16] ; ??
```

What will happen when they run?

Your turn. . .

Write a program that rearranges the values of three doubleword values in the following array as: 3, 1, 2.

```
.data  
arrayD DWORD 1,2,3
```

- Step1: copy the first value into EAX and exchange it with the value in the second position.

```
mov eax,arrayD  
xchg eax,[arrayD+4]
```

- Step 2: Exchange EAX with the third array value and copy the value in EAX to the first array position.

```
xchg eax,[arrayD+8]  
mov arrayD,eax
```

Evaluate this . . .

- We want to write a program that adds the following three bytes:

```
.data  
myBytes BYTE 80h,66h,0A5h
```

- What is your evaluation of the following code?

```
mov al,myBytes  
add al,[myBytes+1]  
add al,[myBytes+2]
```

- What is your evaluation of the following code?

```
mov ax,myBytes  
add ax,[myBytes+1]  
add ax,[myBytes+2]
```

- Any other possibilities?

Evaluate this . . . (cont)

```
.data  
myBytes BYTE 80h,66h,0A5h
```

- How about the following code. Is anything missing?

```
movzx ax,myBytes  
mov bl,[myBytes+1]  
add ax,bx  
mov bl,[myBytes+2]  
add ax,bx ; AX = sum
```

Yes: Move zero to BX before the MOVZX instruction.

What's Next

- Data Transfer Instructions
- **Addition and Subtraction**
- Data-Related Operators and Directives
- Indirect Addressing
- JMP and LOOP Instructions
- 64-Bit Programming

Addition and Subtraction

- INC and DEC Instructions
- ADD and SUB Instructions
- NEG Instruction
- Implementing Arithmetic Expressions
- Flags Affected by Arithmetic
 - Zero
 - Sign
 - Carry
 - Overflow
 - Overflow and Carry Flags A Hardware Viewpoint

INC and DEC Instructions

- Add 1, subtract 1 from destination operand
 - operand may be register or memory
- INC *destination*
 - Logic: $destination \leftarrow destination + 1$
- DEC *destination*
 - Logic: $destination \leftarrow destination - 1$

INC and DEC Examples

```
.data
myWord WORD 1000h
myDword DWORD 10000000h
.code
    inc myWord    ; 1001h
    dec myWord    ; 1000h
    inc myDword   ; 10000001h

    mov ax,00FFh
    inc ax ; AX = 0100h
    mov ax,00FFh
    inc al ; AX = 0000h
```

Your turn...

Show the value of the destination operand after each of the following instructions executes:

```
.data  
myByte BYTE 0FFh, 0  
.code  
    mov al,myByte      ; AL = FFh  
    mov ah,[myByte+1]   ; AH = 00h  
    dec ah             ; AH = FFh  
    inc al             ; AL = 00h  
    dec ax             ; AX = FEFF
```

ADD and SUB Instructions

- ADD destination, source
 - Logic: $destination \leftarrow destination + source$
- SUB destination, source
 - Logic: $destination \leftarrow destination - source$
- Same operand rules as for the MOV instruction (referred to page 9)

ADD and SUB Examples

```
.data
var1 DWORD 10000h
var2 DWORD 20000h
.code ; ---EAX---
        mov eax,var1 ; 00010000h
        add eax,var2 ; 00030000h
        add ax,0FFFFh ; 0003FFFFh
        add eax,1      ; 00040000h
        sub ax,1      ; 0004FFFFh
```

NEG (negate) Instruction

Reverses the sign of an operand. Operand can be a register or memory operand.

```
.data  
valB BYTE -1  
valW WORD +32767  
.code  
    mov al, valB    ; AL = -1  
    neg al         ; AL = +1  
    neg valW       ; valW = -32767
```

Suppose AX contains –32,768 and we apply NEG to it. Will the result be valid?

NEG Instruction and the Flags

The processor implements NEG using the following internal operation:

SUB 0, operand

Any nonzero operand causes the Carry flag to be set.

```
.data
valB BYTE 1,0
valC SBYTE -128
.code
    neg valB      ; CF = 1, OF = 0
    neg [valB + 1] ; CF = 0, OF = 0
    neg valC      ; CF = 1, OF = 1
```

Implementing Arithmetic Expressions

HLL compilers translate mathematical expressions into assembly language. You can do it also. For example:

$$Rval = -Xval + (Yval - Zval)$$

```
Rval DWORD ?
Xval DWORD 26
Yval DWORD 30
Zval DWORD 40
.code
    mov eax,Xval
    neg eax      ; EAX = -Xval
    mov ebx,Yval ; EBX = Yval
    sub ebx,Zval ; EBX = Yval-Zval
    add eax,ebx  ; EAX = -Xval+(Yval-Zval)
    mov Rval,eax ; Rval= -Xval+(Yval-Zval)
```

Your turn...

Translate the following expression into assembly language.
Do not permit Xval, Yval, or Zval to be modified:

$$Rval = Xval - (-Yval + Zval)$$

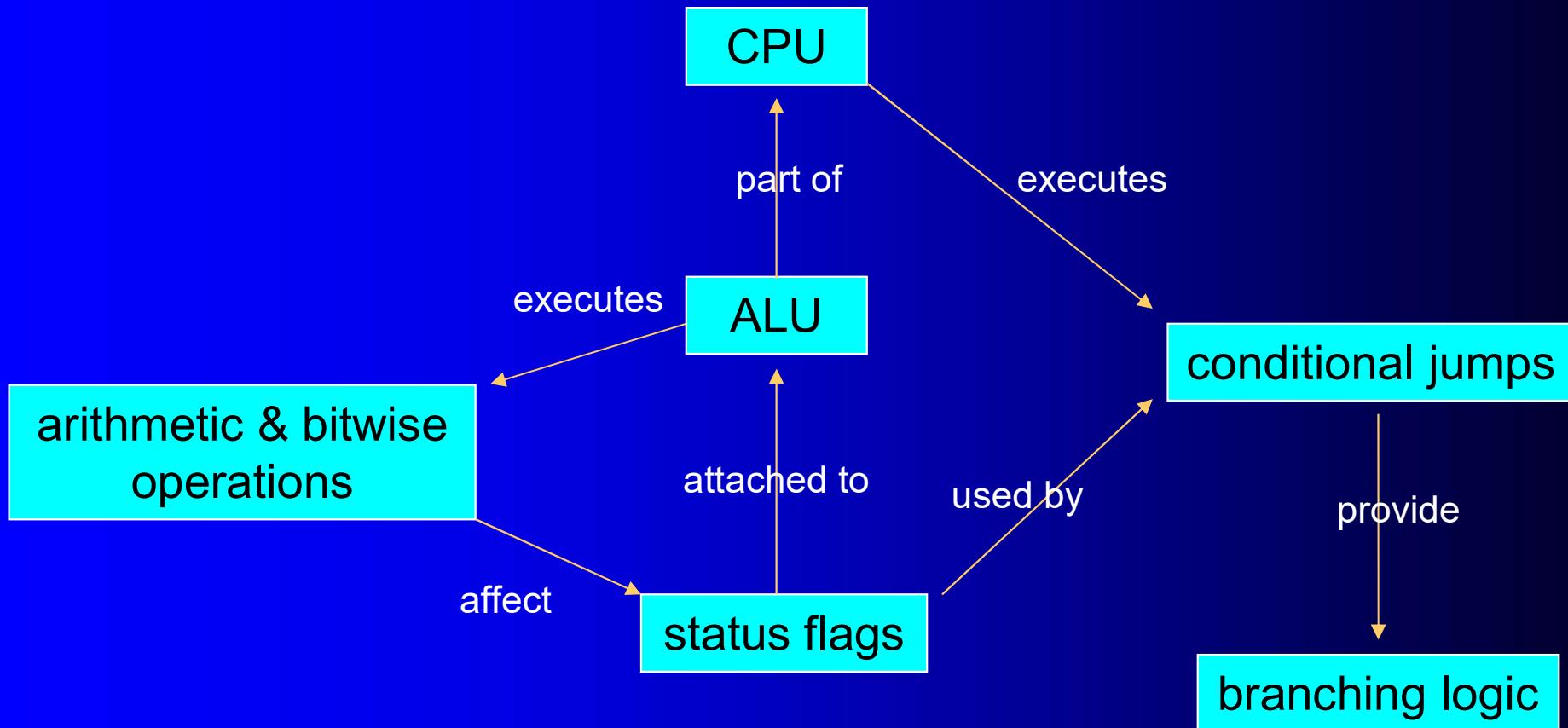
Assume that all values are signed doublewords.

```
    mov ebx,Yval  
    neg ebx  
    add ebx,Zval  
    mov eax,Xval  
    sub eax,ebx  
    mov Rval,eax
```

Flags Affected by Arithmetic

- The ALU has a number of status flags that reflect the outcome of arithmetic (and bitwise) operations
 - based on the contents of the destination operand
- Essential flags:
 - Zero flag – set when destination equals zero
 - Sign flag – set when destination is negative
 - Carry flag – set when unsigned value is out of range
 - Overflow flag – set when signed value is out of range
- The MOV instruction never affects the flags.

Concept Map

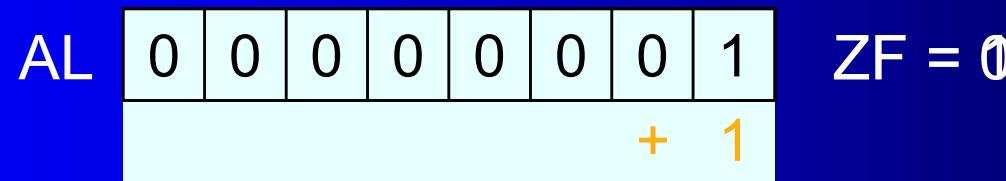


You can use diagrams such as these to express the relationships between assembly language concepts.

Zero Flag (ZF)

The Zero flag is set when the result of an operation produces zero in the destination operand.

```
mov cx,1  
sub cx,1      ; CX = 0, ZF = 1  
mov al,0FFh  
inc al       ; AL = 0, ZF = 1  
inc al       ; AL = 1, ZF = 0
```



Remember...

- A flag is **set** when it equals 1.
- A flag is **clear** when it equals 0.

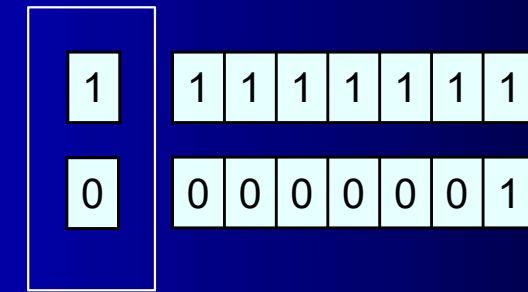
Sign Flag (SF)

The Sign flag is set when the destination operand is negative.
The flag is clear when the destination is positive.

The sign flag is a copy of the destination's highest bit:

```
mov al,0  
sub al,1  
add al,2
```

;SF = 1 AL
;SF = 0 AL



First bit is equal to Sign Flag

Signed and Unsigned Integers

A Hardware Viewpoint

- All CPU instructions operate exactly the same on signed and unsigned integers
- The CPU cannot distinguish between signed and unsigned integers
- YOU, the programmer, are solely responsible for using the correct data type with each instruction

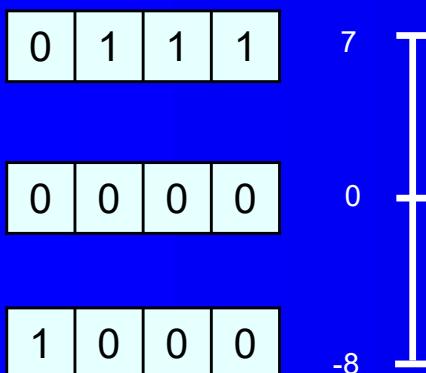
Added Slide. Gerald Cahill, Antelope Valley College

Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2015.

Overflow and Carry Flags

A Hardware Viewpoint

- How the ADD instruction affects OF and CF:
 - CF = (carry out of the MSB)
 - OF = (carry out of the MSB) XOR (carry into the MSB)
- How the SUB instruction affects OF and CF:
 - CF = INVERT (carry out of the MSB)
 - negate the source and add it to the destination
 - OF = (carry out of the MSB) XOR (carry into the MSB)



MSB = Most Significant Bit (high-order bit)
XOR = eXclusive-OR operation
NEG = Negate (same as SUB 0,operand)

Carry Flag (CF)

The Carry flag is set when the result of an operation generates an unsigned value that is out of range (too big or too small for the destination operand).

```
mov al,0FFh  
add al,1      ; CF = 1, AL = 00
```

; Try to go below zero:

```
mov al,0  
sub al,1      ; CF = 1, AL = FF
```

CF 0 AL 1 1 1 1 1 1 1 1
+ 1

CF 1 AL 0 0 0 0 0 0 0 0

CF 0 AL 0 0 0 0 0 0 0 0
- 1

CF 1 AL 1 1 1 1 1 1 1 1

Your turn . . .

For each of the following marked entries, show the values of the destination operand and the Sign, Zero, and Carry flags:

mov ax,00FFh		
add ax,1	; AX= 0100h	SF= 0 ZF= 0 CF= 0
sub ax,1	; AX= 00FFh	SF= 0 ZF= 0 CF= 0
add al,1	; AL= 00h	SF= 0 ZF= 1 CF= 1
mov bh,6Ch		
add bh,95h	; BH= 01h	SF= 0 ZF= 0 CF= 1
mov al,2		
sub al,3	; AL= FFh	SF= 1 ZF= 0 CF= 1

Overflow Flag (OF)

The Overflow flag is set when the signed result of an operation is invalid or out of range.

```
; Example 1  
mov al,+127  
add al,1      ; OF = 1,    AL = ??  
  
; Example 2  
mov al,7Fh    ; OF = 1,    AL = 80h  
add al,1
```

The two examples are identical at the binary level because 7Fh equals +127. To determine the value of the destination operand, it is often easier to calculate in hexadecimal.

A Rule of Thumb

- When adding two integers, remember that the Overflow flag is only set when . . .
 - Two positive operands are added and their sum is negative
 - Two negative operands are added and their sum is positive

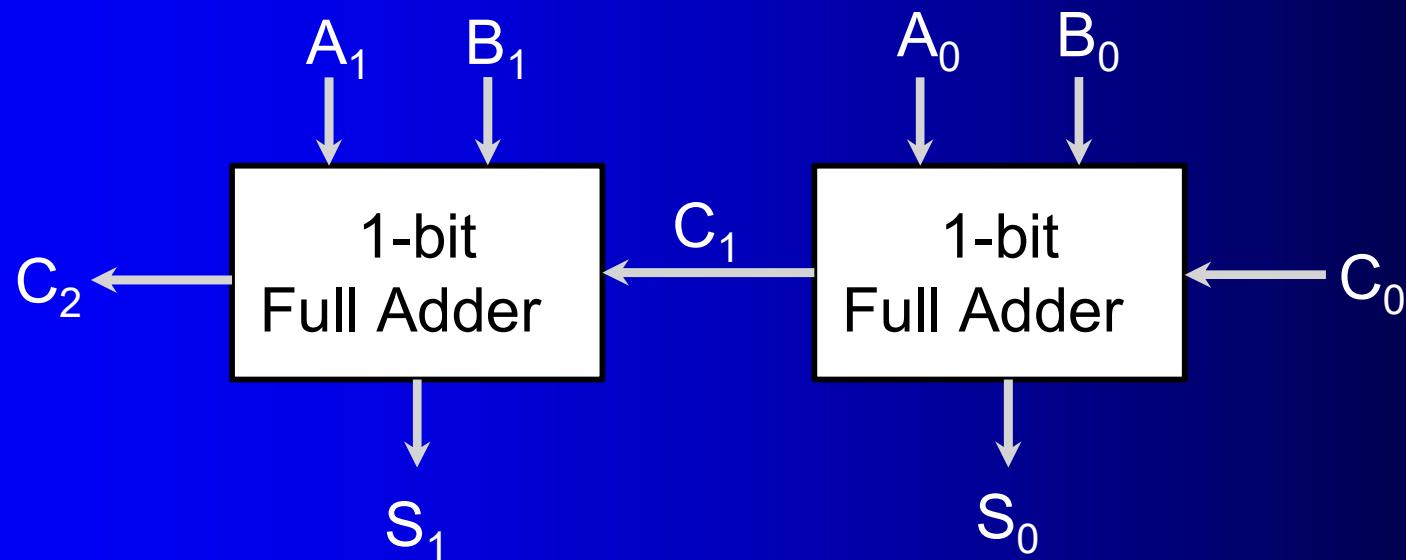
What will be the values of the Overflow flag?

```
mov al,80h  
add al,92h ; OF = 1
```

```
mov al,-2  
add al,+127 ; OF = 0
```

Overflow Flag – Hardware View (1/2)

- OF = (carry out of the MSB) XOR (carry into the MSB)



C_1	C_2	$C_1 \text{ XOR } C_2$
1	1	0
1	0	1
0	1	1
0	0	0

Overflow Flag – Hardware View (2/2)

- OF = (carry out of the MSB) XOR (carry into the MSB)

(Carry out, Carry in) = (0, 1)		(1, 1)
$\begin{array}{r} 01 \\ 0101 \text{ (5)} \\ 0110 \text{ (6)} \\ \hline 01011 \end{array}$	CF = 0 OF = 1	$\begin{array}{r} 11 \\ 1101 \text{ (-3)} \\ \hline 0111 \text{ (7)} \\ 10100 \end{array}$
(0, 0)		(1, 0)
$\begin{array}{r} 00 \\ 1100 \text{ (-4)} \\ \hline 0011 \text{ (3)} \\ 01111 \end{array}$	CF = 0 OF = 0	$\begin{array}{r} 10 \\ 1001 \text{ (-7)} \\ 1011 \text{ (-5)} \\ \hline 10100 \end{array}$

Your turn . . .

What will be the values of the given flags after each operation?

```
mov al,-128  
neg al           ; CF = 1    OF = 1
```

```
mov ax,8000h  
add ax,2         ; CF = 0    OF = 0
```

```
mov ax,0  
sub ax,2         ; CF = 1    OF = 0
```

```
mov al,-5  
sub al,+125     ; OF = 1
```

What's Next

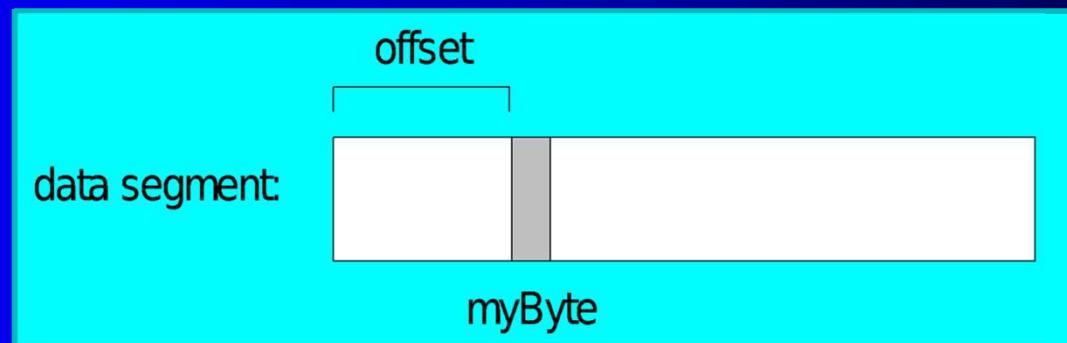
- Data Transfer Instructions
- Addition and Subtraction
- **Data-Related Operators and Directives**
- Indirect Addressing
- JMP and LOOP Instructions
- 64-Bit Programming

Data-Related Operators and Directives

- OFFSET Operator
- PTR Operator
- TYPE Operator
- LENGTHOF Operator
- SIZEOF Operator
- LABEL Directive

OFFSET Operator

- OFFSET returns the distance in bytes, of a label from the beginning of its enclosing segment
 - Protected mode: 32 bits
 - Real mode: 16 bits



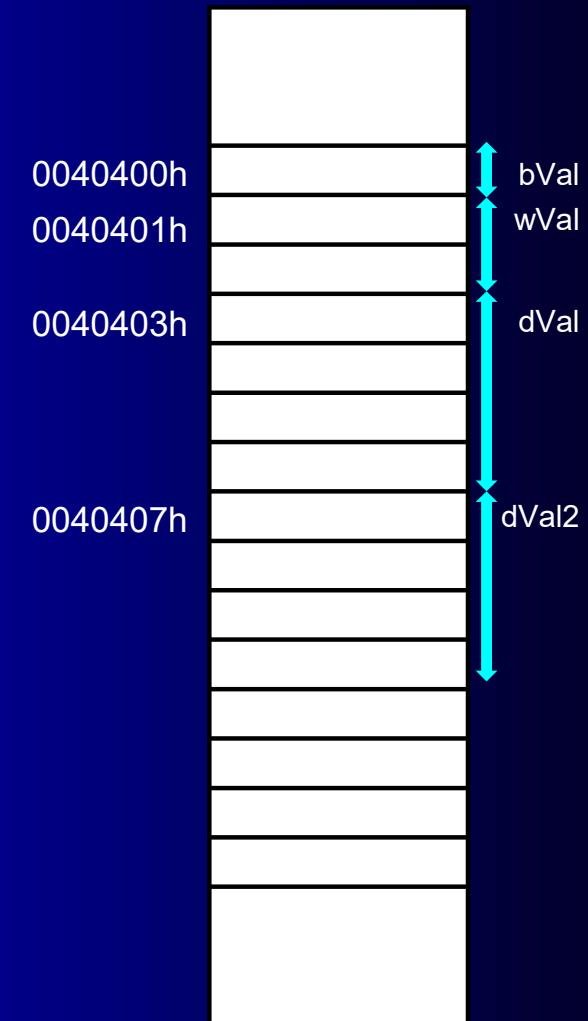
The Protected-mode programs we write use only a single segment (flat memory model).

OFFSET Examples

Let's assume that the data segment begins at 00404000h:

```
.data  
bVal BYTE ? ;1  
wVal WORD ? ;2  
dVal DWORD ? ;4  
dVal2 DWORD ? ;4
```

```
.code  
mov esi,OFFSET bVal ; ESI = 00404000  
mov esi,OFFSET wVal ; ESI = 00404001  
mov esi,OFFSET dVal ; ESI = 00404003  
mov esi,OFFSET dVal2 ; ESI = 00404007
```



Relating to C/C++

The value returned by OFFSET is a pointer. Compare the following code written for both C++ and assembly language:

```
// C++ version:  
  
char array[1000];  
char * p = array;
```

```
; Assembly language:  
  
.data  
array BYTE 1000 DUP(?)  
.code  
mov     esi,OFFSET array
```

PTR Operator

Overrides the default type of a label (variable). Provides the flexibility to access part of a variable.

```
.data  
myDouble DWORD 12345678h  
.code  
mov ax,myDouble ; error - why?  
  
mov ax,WORD PTR myDouble 5678h ; loads  
  
mov WORD PTR myDouble,4321h ; saves  
4321h
```

Little endian order is used when storing data in memory (see Section 3.4.9).

Little Endian Order

- Little endian order refers to the way Intel stores integers in memory.
- Multi-byte integers are stored in reverse order, with the least significant byte stored at the lowest address
- For example, the doubleword 12345678h would be stored as:

doubleword	word	byte	offset	
12345678	5678	78	0000	myDouble
		56	0001	myDouble +1
	1234	34	0002	myDouble +2
		12	0003	myDouble +3

When integers are loaded from memory into registers, the bytes are automatically re-reversed into their correct positions.

PTR Operator Examples

.data

myDouble DWORD 12345678h

doubleword	word	byte	offset	
12345678	5678	78	0000	myDouble
		56	0001	myDouble + 1
	1234	34	0002	myDouble + 2
		12	0003	myDouble + 3

```
mov al,BYTE PTR myDouble           ; AL = 78h
mov al,BYTE PTR [myDouble+1]        ; AL = 56h
mov al,BYTE PTR [myDouble+2]        ; AL = 34h
mov ax,WORD PTR myDouble          ; AX = 5678h
mov ax,WORD PTR [myDouble+2]        ; AX = 1234h
```

PTR Operator (cont)

PTR can also be used to combine elements of a smaller data type and move them into a larger operand. The CPU will automatically reverse the bytes.

```
.data  
myBytes BYTE 12h,34h,56h,78h  
  
.code  
mov ax,WORD PTR [myBytes] ; AX = 3412h  
mov ax,WORD PTR [myBytes+2] ; AX = 7856h  
mov eax,DWORD PTR myBytes ; EAX = 78563412h
```

Your turn . . .

Write down the value of each destination operand:

```
.data  
varB BYTE 65h,31h,02h,05h  
varW WORD 6543h,1202h  
varD DWORD 12345678h  
  
.code  
mov ax,WORD PTR [varB+2] ; a. 0502h  
mov bl,BYTE PTR varD ; b. 78h  
mov bl,BYTE PTR [varW+2] ; c. 02h  
mov ax,WORD PTR [varD+2] ; d. 1234h  
mov eax,DWORD PTR varW ; e. 12026543h
```

TYPE Operator

The TYPE operator returns the size, in bytes, of a single element of a data declaration.

```
.data  
var1 BYTE ?  
var2 WORD ?  
var3 DWORD ?  
var4 QWORD ?  
  
.code  
mov eax,TYPE var1 ; 1  
mov eax,TYPE var2 ; 2  
mov eax,TYPE var3 ; 4  
mov eax,TYPE var4 ; 8
```

LENGTHOF Operator

The LENGTHOF operator counts the number of elements in a single data declaration.

```
.data LENGTHOF
byte1 BYTE 10,20,30; 3
array1 WORD 30 DUP(?) ,0,0 ; 32
array2 WORD 5 DUP(3 DUP(?)) ; 15
array3 DWORD 1,2,3,4 ; 4
digitStr BYTE "12345678",0 ; 9

.code
mov ecx,LENGTHOF array1 ; 32
```

SIZEOF Operator

The SIZEOF operator returns a value that is equivalent to multiplying LENGTHOF by TYPE.

```
.data  SIZEOF
byte1 BYTE 10,20,30          ; 3
array1 WORD 30 DUP(?) ,0,0   ; 64
array2 WORD 5 DUP(3 DUP(?)) ; 30
array3 DWORD 1,2,3,4; 16
digitStr BYTE "12345678",0   ; 9

.code
mov ecx,SIZEOF array1      ; 64
```

Spanning Multiple Lines (1 of 2)

A data declaration spans multiple lines if each line (except the last) ends with a comma. The LENGTHOF and SIZEOF operators include all lines belonging to the declaration:

```
.data  
array WORD 10,20,  
      30,40,  
      50,60  
  
.code  
mov eax,LENGTHOF array      ; 6  
mov ebx,SIZEOF array        ; 12
```

Spanning Multiple Lines (2 of 2)

In the following example, array identifies only the first WORD declaration. Compare the values returned by LENGTHOF and SIZEOF here to those in the previous slide:

```
.data  
array  WORD 10,20  
        WORD 30,40  
        WORD 50,60  
  
.code  
mov eax,LENGTHOF array      ; 2  
mov ebx,SIZEOF array        ; 4
```

LABEL Directive

- Assigns an alternate label name and type to an existing storage location
- LABEL does not allocate any storage of its own
- Removes the need for the PTR operator

```
.data
dwList    LABEL DWORD
wordList  LABEL WORD
intList   BYTE 00h,10h,00h,20h
.code
mov eax,dwList           ; 20001000h
mov cx,wordList          ; 1000h
mov dl,intList           ; 00h
```

What's Next

- Data Transfer Instructions
- Addition and Subtraction
- Data-Related Operators and Directives
- **Indirect Addressing**
- JMP and LOOP Instructions
- 64-Bit Programming

Indirect Addressing

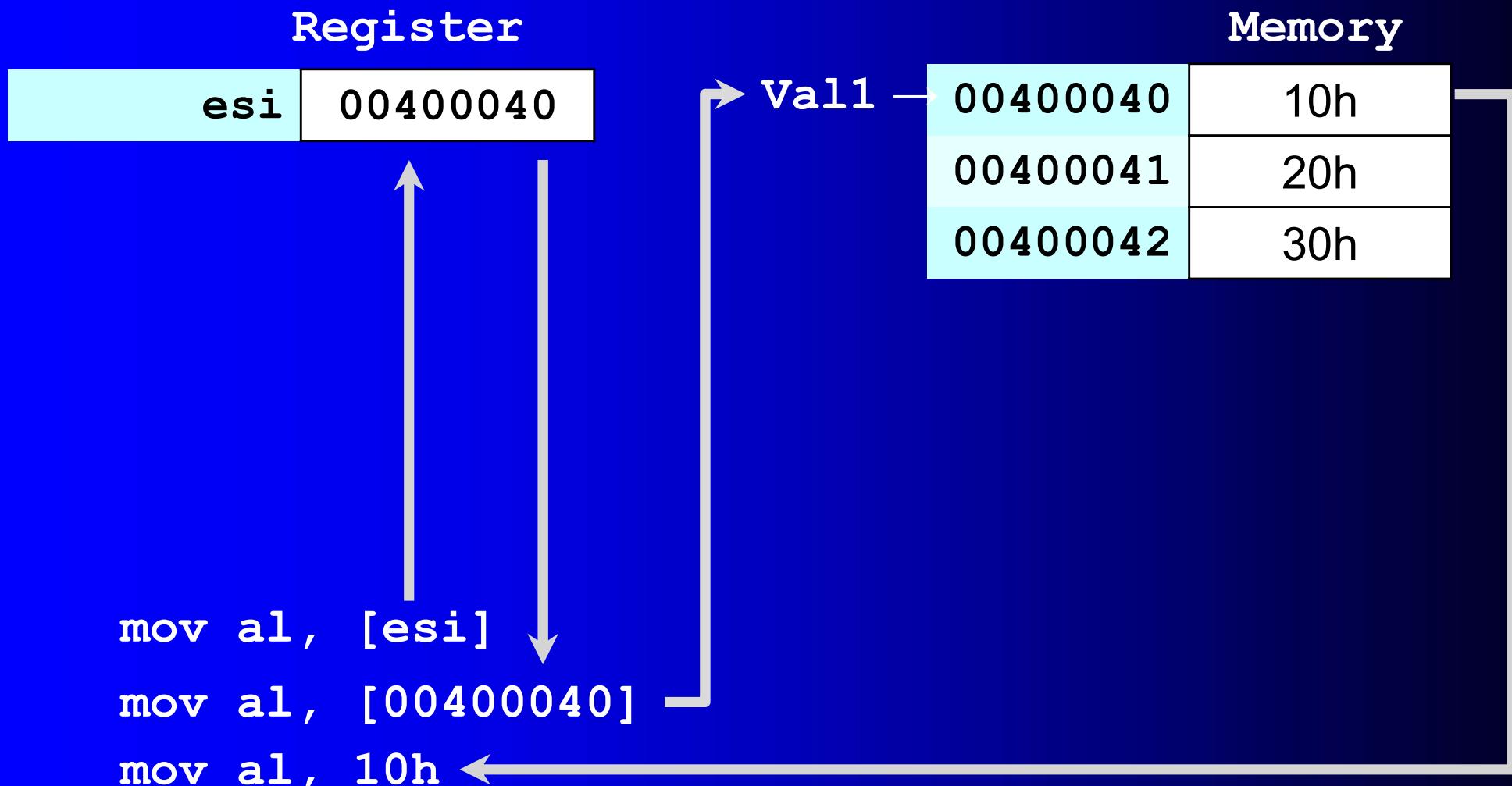
- Indirect Operands
- Array Sum Example
- Indexed Operands
- Pointers

Indirect Operands (1 of 3)

An indirect operand holds the address of a variable, usually an array or string. It can be dereferenced (just like a pointer).

```
.data  
val1 BYTE 10h,20h,30h  
.code  
mov esi,OFFSET val1  
mov al,[esi] ; dereference ESI (AL = 10h)  
  
inc esi  
mov al,[esi] ; AL = 20h  
  
inc esi  
mov al,[esi] ; AL = 30h
```

Indirect Operands (2 of 3)



Indirect Operands (3 of 3)

Use PTR to clarify the size attribute of a memory operand.

```
.data  
myCount WORD 0  
  
.code  
mov esi,OFFSET myCount  
inc [esi]      ; error: ambiguous  
inc WORD PTR [esi]  ; ok
```

Should PTR be used here?

```
add [esi],20
```

yes, because [esi] could point to a byte, word, or doubleword

Array Sum Example

Indirect operands are ideal for traversing an array. Note that the register in brackets must be incremented by a value that matches the array type.

```
.data  
arrayW WORD 1000h,2000h,3000h  
.code  
    mov esi,OFFSET arrayW  
    mov ax,[esi]  
    add esi,2 ; or: add esi,TYPE arrayW  
    add ax,[esi]  
    add esi,2  
    add ax,[esi]      ; AX = sum of the array
```

To Do: Modify this example for an array of doublewords.

Indexed Operands

An indexed operand adds a constant to a register to generate an effective address. There are two notational forms:

[label + reg]

label[reg]

```
.data  
arrayW WORD 1000h,2000h,3000h  
.code  
    mov esi,0  
    mov ax,[arrayW+esi]           ; AX = 1000h  
    mov ax,arrayW[esi]           ; alternate format  
    add esi,2  
    add ax,[arrayW+esi]  
etc.
```

ToDo: Modify this example for an array of doublewords.

Index Scaling

You can scale an indirect or indexed operand to the offset of an array element. This is done by multiplying the index by the array's TYPE:

```
.data  
arrayB BYTE 0,1,2,3,4,5  
arrayW WORD 0,1,2,3,4,5  
arrayD DWORD 0,1,2,3,4,5  
  
.code  
mov esi,4  
mov al,arrayB[esi*TYPE arrayB] ; 04  
mov bx,arrayW[esi*TYPE arrayW] ; 0004  
mov edx,arrayD[esi*TYPE arrayD] ; 00000004
```

Pointers

You can declare a pointer variable that contains the offset of another variable.

```
.data  
arrayW WORD 1000h,2000h,3000h  
ptrW DWORD arrayW  
.code  
    mov esi,ptrW  
    mov ax,[esi] ; AX = 1000h
```

Alternate format:

```
ptrW DWORD OFFSET arrayW
```

What's Next

- Data Transfer Instructions
- Addition and Subtraction
- Data-Related Operators and Directives
- Indirect Addressing
- **JMP and LOOP Instructions**
- 64-Bit Programming

JMP and LOOP Instructions

- JMP Instruction
- LOOP Instruction
- LOOP Example
- Summing an Integer Array
- Copying a String

JMP Instruction

- JMP is an unconditional jump to a label that is usually within the same procedure.
- Syntax: JMP *target*
- Logic: EIP \leftarrow *target*
- Example:

```
top:  
.  
.  
jmp top
```

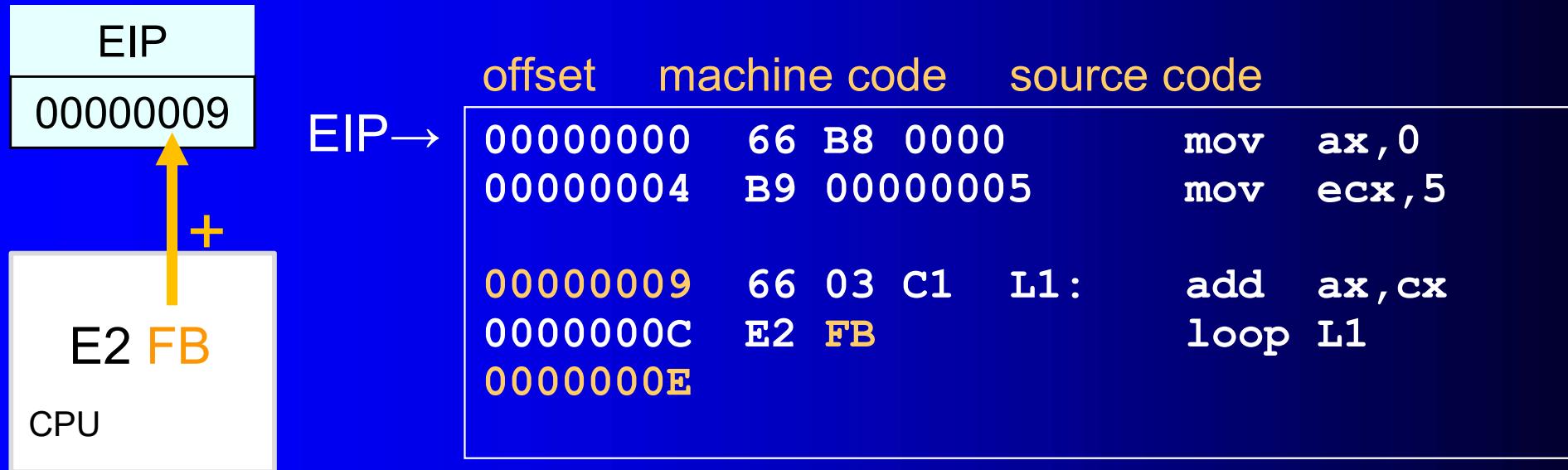
A jump outside the current procedure must be to a special type of label called a **global label** (see Section 5.5.2.3 for details).

LOOP Instruction

- The LOOP instruction creates a counting loop
- Syntax: LOOP *target*
- Logic:
 - $\text{ECX} \leftarrow \text{ECX} - 1$
 - if $\text{ECX} \neq 0$, jump to *target*
- Implementation:
 - The assembler calculates the distance, in bytes, between the offset of the following instruction and the offset of the target label. It is called the relative offset.
 - The relative offset is added to EIP.

LOOP Example

The following loop calculates the sum of the integers
 $5 + 4 + 3 + 2 + 1$:



When LOOP is assembled, the current location = 0000000E (offset of the next instruction). -5 (FBh) is added to the current location, causing a jump to location 00000009:

$$00000009 \leftarrow 0000000E + FB$$

Your turn . . .

If the relative offset is encoded in a single signed byte,

- (a) what is the largest possible backward jump?
- (b) what is the largest possible forward jump?

(a) -128

(b) +127

Your turn . . .

What will be the final value of AX?

10

How many times will the loop execute?

4,294,967,296

```
mov ax, 6  
mov ecx, 4  
L1:  
    inc ax  
    loop L1
```

```
x2:  
    mov ecx, 0  
    inc ax  
    loop x2
```

Nested Loop

If you need to code a loop within a loop, you must save the outer loop counter's ECX value. In the following example, the outer loop executes 100 times, and the inner loop 20 times.

```
.data  
count DWORD ?  
.code  
    mov ecx,100    ; set outer loop count  
L1:  
    mov count,ecx ; save outer loop count  
    mov ecx,20     ; set inner loop count  
L2:  
    .  
    .  
    loop L2        ; repeat the inner loop  
    mov ecx,count ; restore outer loop count  
    loop L1        ; repeat the outer loop
```

Summing an Integer Array

The following code calculates the sum of an array of 16-bit integers.

```
.data  
intarray WORD 100h,200h,300h,400h  
.code  
    mov edi,OFFSET intarray    ; address of intarray  
    mov ecx,LENGTHOF intarray ; loop counter  
    mov ax,0      ; zero the accumulator  
L1:  
    add ax,[edi]        ; add an integer  
    add edi,TYPE intarray    ; point to next integer  
    loop L1      ; repeat until ECX = 0
```

Your turn . . .

What changes would you make to the program on the previous slide if you were summing a doubleword array?

Copying a String

The following code copies a string from source to target:

```
.data
source  BYTE  "This is the source string",0
target  BYTE  $IZEOF source DUP(0)

.code
        mov    esi,0           ; index register
        mov    ecx,$IZEOF source ; loop counter
L1:
        mov    al,source[esi]    ; get char from source
        mov    target[esi],al     ; store it in the target
        inc    esi               ; move to next character
        loop   L1               ; repeat for entire string
```

good use of
SIZEOF

Your turn . . .

Rewrite the program shown in the previous slide, using indirect addressing rather than indexed addressing.

What's Next

- Data Transfer Instructions
- Addition and Subtraction
- Data-Related Operators and Directives
- Indirect Addressing
- JMP and LOOP Instructions
- **64-Bit Programming**

64-Bit Programming

- MOV instruction in 64-bit mode accepts operands of 8, 16, 32, or 64 bits
- When you move a 8, 16, or 32-bit constant to a 64-bit register, the upper bits of the destination are cleared.
- When you move a memory operand into a 64-bit register, the results vary:
 - 32-bit move clears high bits in destination
 - 8-bit or 16-bit move does not affect high bits in destination

More 64-Bit Programming

- MOVSXD sign extends a 32-bit value into a 64-bit destination register
- The OFFSET operator generates a 64-bit address
- LOOP uses the 64-bit RCX register as a counter
- RSI and RDI are the most common 64-bit index registers for accessing arrays.

Other 64-Bit Notes

- ADD and SUB affect the flags in the same way as in 32-bit mode
- You can use scale factors with indexed operands.

Assembly Language for x86 Processors

7th Edition , Global Edition

Kip R. Irvine

Chapter 5: Procedures

Slides prepared by the author

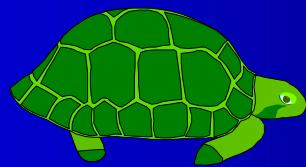
Revision date: 1/15/2014

Chapter Overview

- Stack Operations
- Defining and Using Procedures
- Linking to an External Library
- The Irvine32 Library
- 64-Bit Assembly Programming

Summary

- Procedure – named block of executable code
- Runtime stack – LIFO structure
 - holds return addresses, parameters, local variables
 - PUSH – add value to stack
 - POP – remove value from stack
- Use the Irvine32 library for all standard I/O and data conversion
 - Want to learn more? Study the library source code in the [c:\Irvine\Examples\Lib32](#) folder



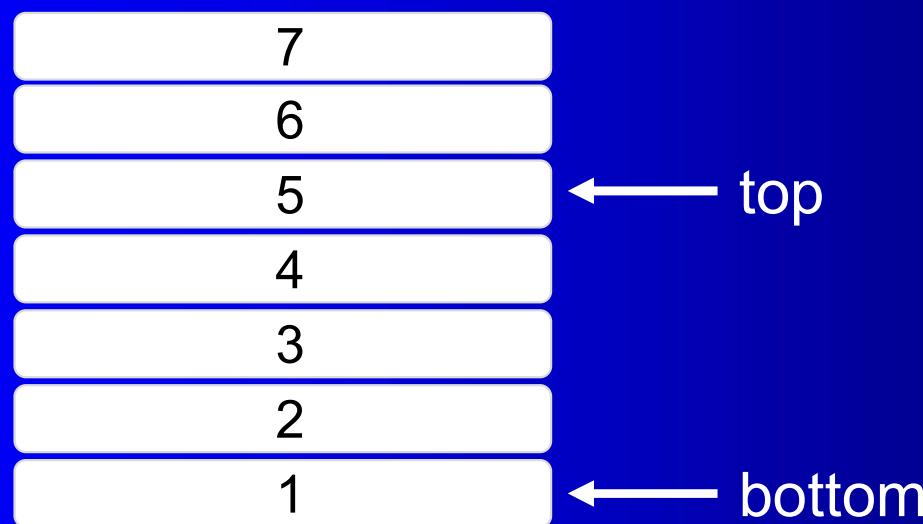
55 64 67 61 6E 67 65 6E

Stack Operations

- Runtime Stack
- PUSH Operation
- POP Operation
- PUSH and POP Instructions
- Using PUSH and POP
- Example: Nested Loop
- Example: Reversing a String
- Related Instructions

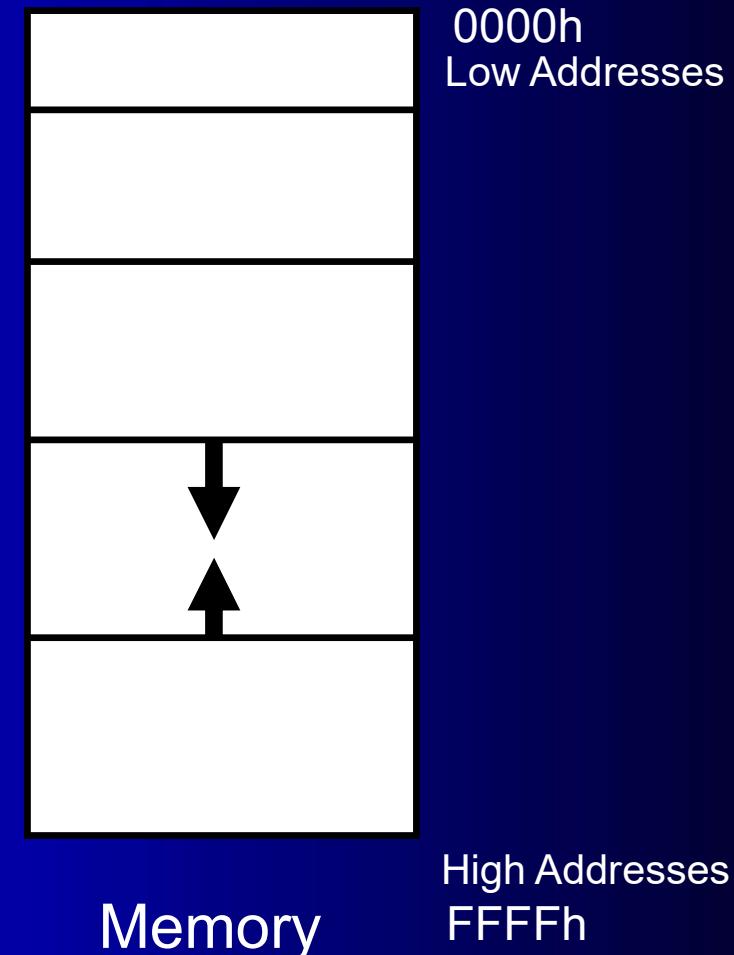
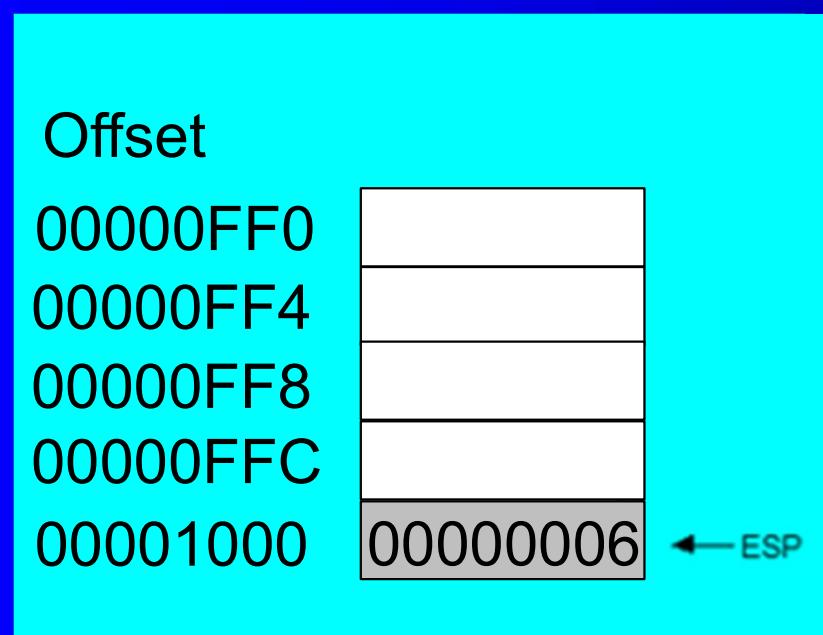
Runtime Stack (1/3)

- Imagine a stack of plates . . .
 - plates are only added to the top
 - plates are only removed from the top
 - Last In First Out (LIFO) structure



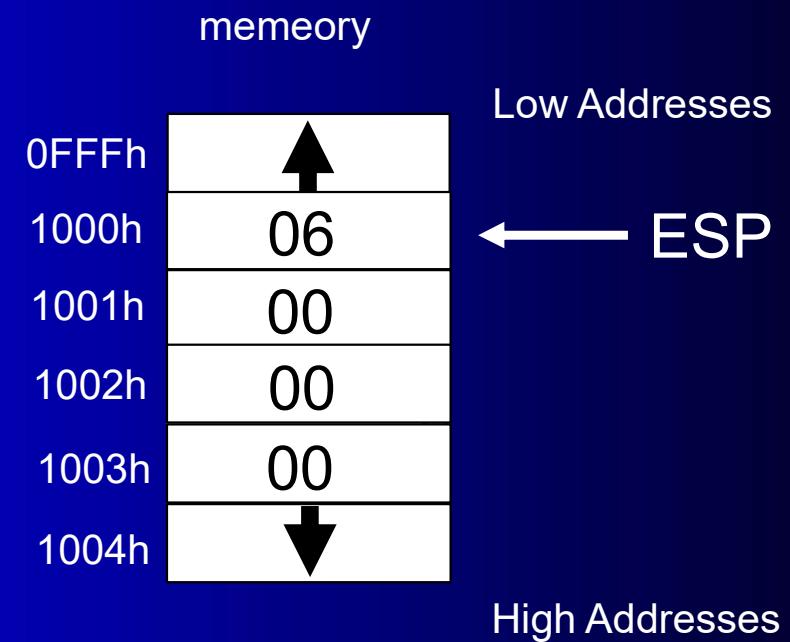
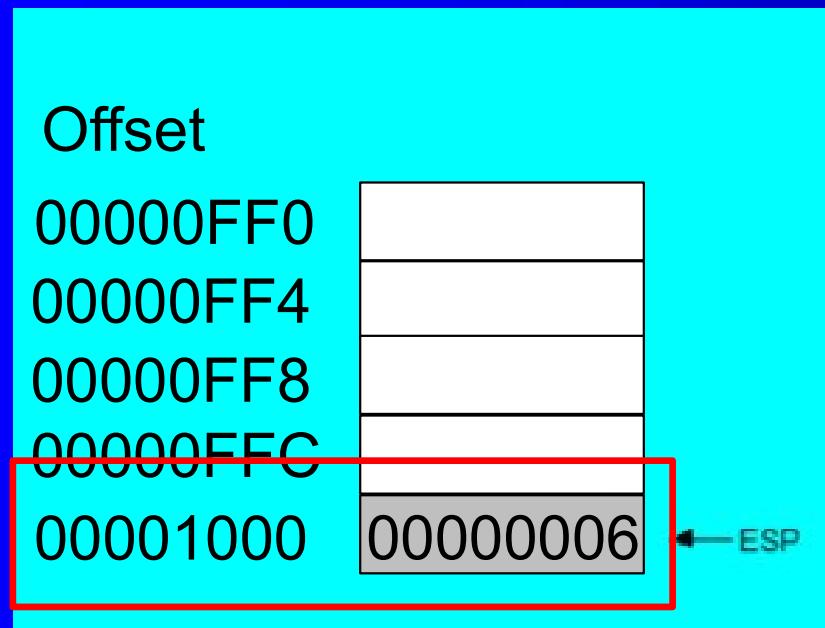
Runtime Stack (2/3)

- Managed by the CPU, using two registers
 - SS (stack segment) (ignore for Flat Memory Model)
 - ESP (stack pointer) *

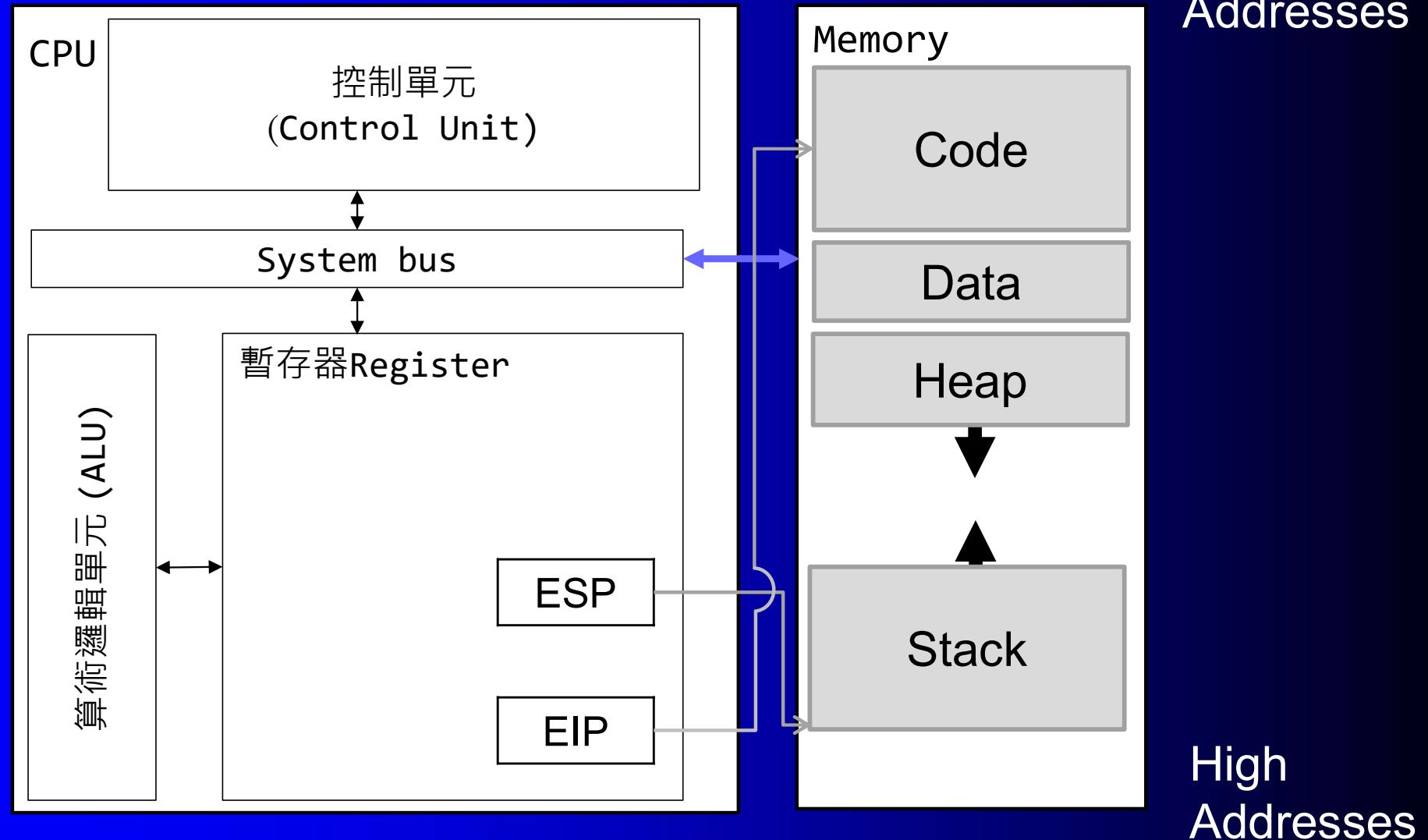


* SP in Real-address mode

In real memory...



Runtime Stack (3/3)



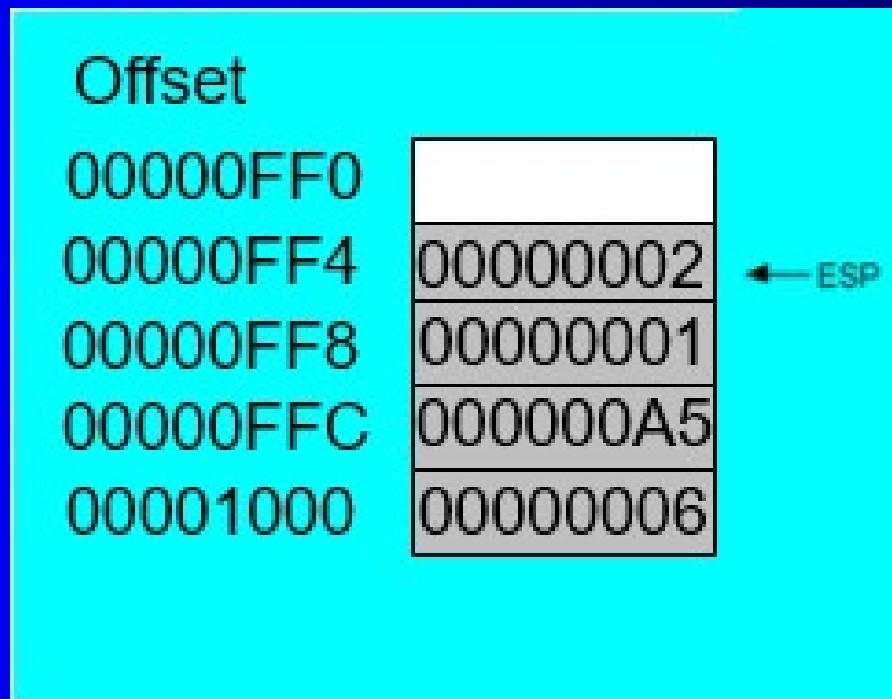
PUSH Operation (1 of 2)

- A 32-bit push operation decrements the stack pointer by 4 and copies a value into the location pointed to by the stack pointer.



PUSH Operation (2 of 2)

- Same stack after pushing two more integers:



The stack grows downward. The area below ESP is always available (unless the stack has overflowed).

POP Operation

- Copies value at stack[ESP] into a register or variable.
- Adds n to ESP, where n is either 2 or 4.
 - value of n depends on the attribute of the operand receiving the data

Offset	Before	Offset	After
00000FF0		00000FF0	
00000FF4	00000002	00000FF4	
00000FF8	00000001	00000FF8	00000001
00000FFC	000000A5	00000FFC	000000A5
00001000	00000006	00001000	00000006

PUSH and POP Instructions

- PUSH syntax:
 - PUSH *r/m16*
 - PUSH *r/m32*
 - PUSH *imm32*
- POP syntax:
 - POP *r/m16*
 - POP *r/m32*

Using PUSH and POP

Save and restore registers when they contain important values.
PUSH and POP instructions occur in the opposite order.

```
        ; DumpMem will use esi,  
        ; ecx, ebx  
push esi                         ; push registers  
push ecx  
push ebx  
  
mov  esi,OFFSET dwordVal          ; display some memory  
mov  ecx,LENGTHOF dwordVal  
mov  ebx,TYPE dwordVal  
call DumpMem  
  
pop  ebx                         ; restore registers  
pop  ecx  
pop  esi
```

Example: Nested Loop

Compare to [Chapter 4: Nested Loop](#)

When creating a nested loop, push the outer loop counter before entering the inner loop:

```
    mov ecx,100          ; set outer loop count
L1:                                ; begin the outer loop
    push ecx            ; save outer loop count

    mov ecx,20          ; set inner loop count
L2:                                ; begin the inner loop
    ;
    ;
    loop L2             ; repeat the inner loop

    pop ecx             ; restore outer loop count
    loop L1             ; repeat the outer loop
```

Nested Loop (Older version – Ch.4)

If you need to code a loop within a loop, you must save the outer loop counter's ECX value. In the following example, the outer loop executes 100 times, and the inner loop 20 times.

```
.data  
count DWORD ?  
.code  
    mov ecx,100          ; set outer loop count  
L1:  
    mov count,ecx        ; save outer loop count  
    mov ecx,20            ; set inner loop count  
L2: .  
    .  
    loop L2              ; repeat the inner loop  
    mov ecx,count         ; restore outer loop count  
    loop L1              ; repeat the outer loop
```

Example: Reversing a String

- Use a loop with indexed addressing
- Push each character on the stack
- Start at the beginning of the string, pop the stack in reverse order, insert each character back into the string
- [Source code](#)
- Q: Why must each character be put in EAX before it is pushed?

Because only word (16-bit) or doubleword (32-bit) values can be pushed on the stack.

Your turn . . .

- Using the String Reverse program as a starting point,
- #1: Modify the program so the user can input a string containing between 1 and 50 characters.
- #2: Modify the program so it inputs a list of 32-bit integers from the user, and then displays the integers in reverse order.

Related Instructions

- PUSHFD and POPFD
 - push and pop the EFLAGS register
- PUSHAD pushes the 32-bit general-purpose registers on the stack
 - order: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI
- POPAD pops the same registers off the stack in reverse order
 - PUSHA and POPA do the same for 16-bit registers

Your Turn . . .

- Write a program that does the following:
 - Assigns integer values to EAX, EBX, ECX, EDX, ESI, and EDI
 - Uses PUSHAD to push the general-purpose registers on the stack
 - Using a loop, your program should pop each integer from the stack and display it on the screen

What's Next

- Stack Operations
- **Defining and Using Procedures**
- Linking to an External Library
- The Irvine32 Library
- 64-Bit Assembly Programming

Defining and Using Procedures

- Creating Procedures
- Documenting Procedures
- Example: SumOf Procedure
- CALL and RET Instructions
- Nested Procedure Calls
- Local and Global Labels
- Procedure Parameters
- USES Operator
- USES Example

Creating Procedures

- Large problems can be divided into smaller tasks to make them more manageable
- A procedure is the ASM equivalent of a Java or C++ function
- Following is an assembly language procedure named sample:

```
sample PROC
    .
    .
    ret
sample ENDP
```

Documenting Procedures

Suggested documentation for each procedure:

- A description of all tasks accomplished by the procedure.
- **Receives:** A list of input parameters; state their usage and requirements.
- **Returns:** A description of values returned by the procedure.
- **Requires:** Optional list of requirements called **preconditions** that must be satisfied before the procedure is called.

If a procedure is called without its preconditions satisfied, it will probably not produce the expected output.

Example: SumOf Procedure

```
-----  
SumOf PROC  
;  
; Calculates and returns the sum of three 32-bit integers.  
; Receives: EAX, EBX, ECX, the three integers. May be  
; signed or unsigned.  
; Returns: EAX = sum, and the status flags (Carry,  
; Overflow, etc.) are changed.  
; Requires: nothing  
-----  
    add eax,ebx  
    add eax,ecx  
    ret  
SumOf ENDP
```

CALL and RET Instructions

- The CALL instruction calls a procedure
 - pushes offset of next instruction on the stack
 - copies the address of the called procedure into EIP
- The RET instruction returns from a procedure
 - pops top of stack into EIP

CALL-RET Example (1 of 3)

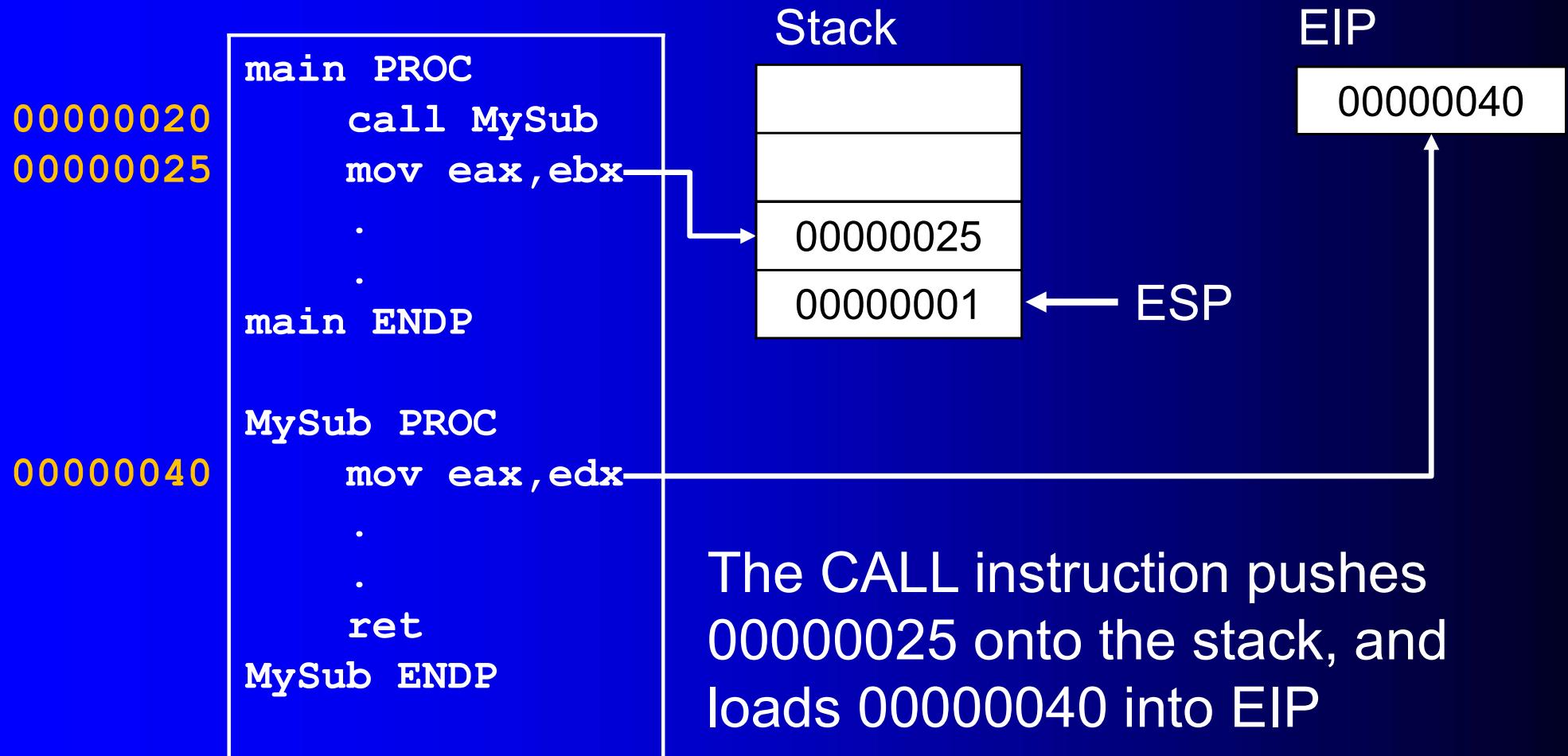
0000025 is the offset of the instruction immediately following the CALL instruction

00000040 is the offset of the first instruction inside MySub

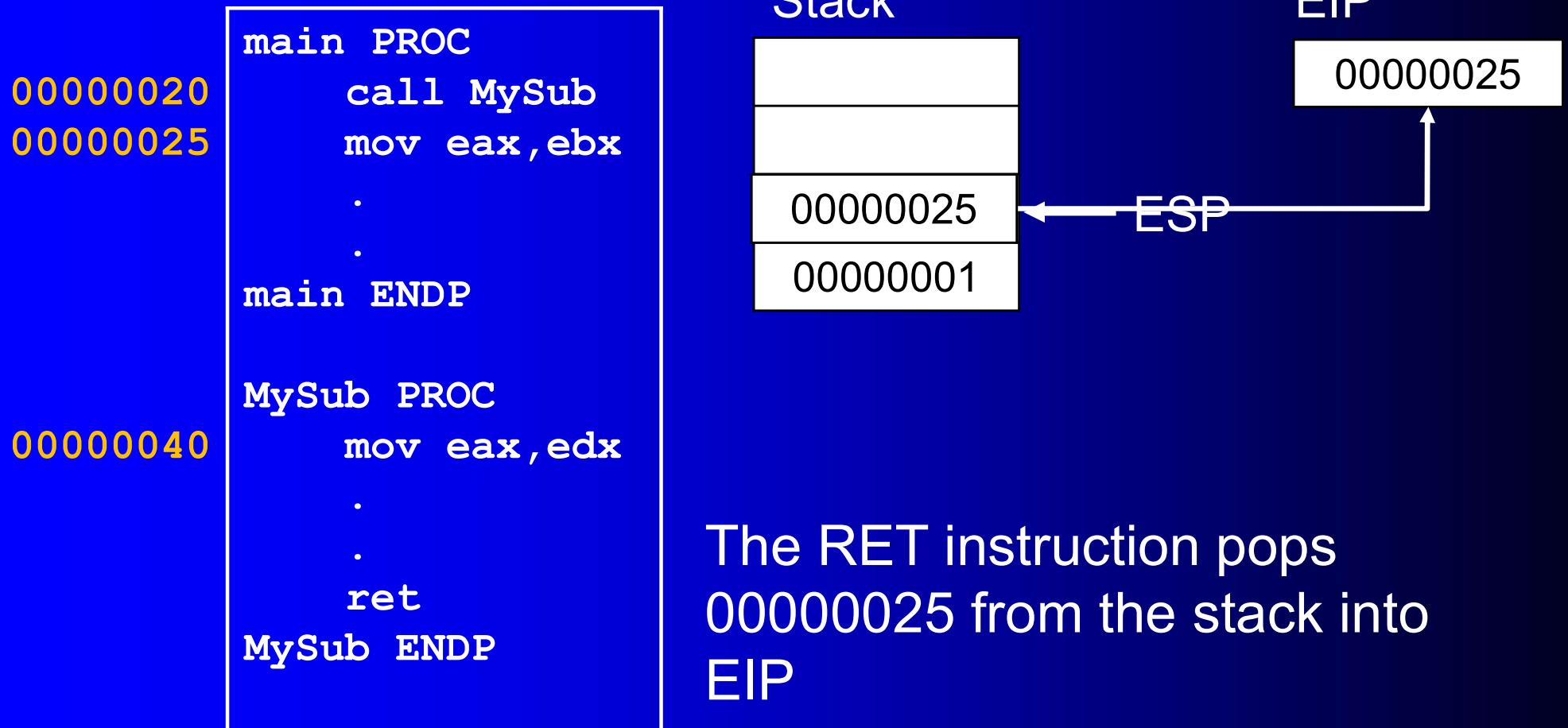
```
main PROC
    00000020 call MySub
    00000025 mov eax,ebx
    .
    .
main ENDP

MySub PROC
    00000040 mov eax,edx
    .
    .
    ret
MySub ENDP
```

CALL-RET Example (2 of 3)



CALL-RET Example (3 of 3)



Nested Procedure Calls

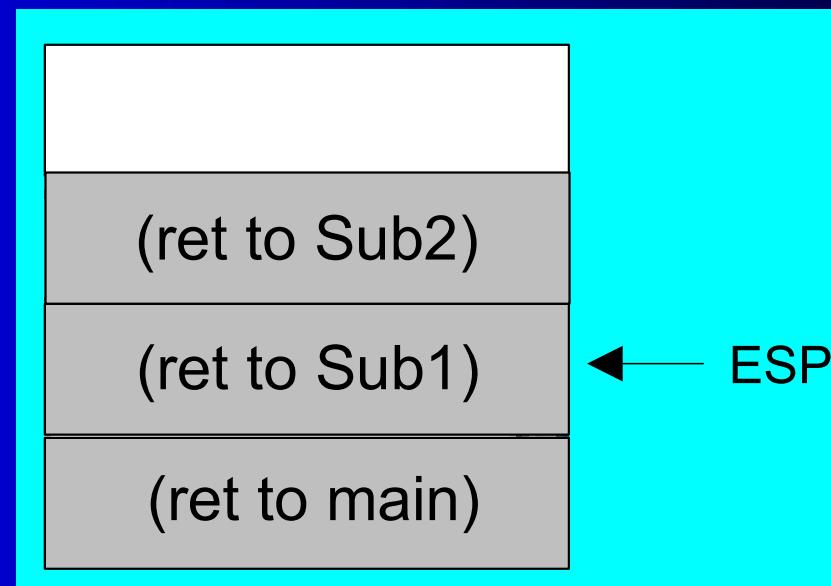
```
main PROC
    .
    .
    call Sub1
    exit
main ENDP

Sub1 PROC
    .
    .
    call Sub2
    ret
Sub1 ENDP

Sub2 PROC
    .
    .
    call Sub3
    ret
Sub2 ENDP

Sub3 PROC
    .
    .
    ret
Sub3 ENDP
```

By the time Sub3 is called, the stack contains all three return addresses:



Local and Global Labels

A local label is visible only to statements inside the same procedure. A global label is visible everywhere.

```
main PROC
    jmp L2
L1::          ; error
    exit
main ENDP

sub2 PROC
L2:           ; local label
    jmp L1
    ret
sub2 ENDP
```

Procedure Parameters (1 of 3)

- A good procedure might be usable in many different programs
 - but not if it refers to specific variable names
- Parameters help to make procedures flexible because parameter values can change at runtime

Procedure Parameters (2 of 3)

The ArraySum procedure calculates the sum of an array. It makes two references to specific variable names:

```
ArraySum PROC
    mov esi,0           ; array index
    mov eax,0           ; set the sum to zero
    mov ecx,LENGTHOF myarray ; set number of elements

L1: add eax,myArray[esi]      ; add each integer to sum
    add esi,4          ; point to next integer
    loop L1           ; repeat for array size

    mov theSum,eax    ; store the sum
    ret
ArraySum ENDP
```

What if you wanted to calculate the sum of two or three arrays within the same program?

Procedure Parameters (3 of 3)

This version of ArraySum returns the sum of any doubleword array whose address is in ESI. The sum is returned in EAX:

```
ArraySum PROC
; Receives: ESI points to an array of doublewords,
; ECX = number of array elements.
; Returns: EAX = sum
;-----
    mov eax,0          ; set the sum to zero

L1: add eax,[esi]      ; add each integer to sum
    add esi,4         ; point to next integer
    loop L1          ; repeat for array size

    ret
ArraySum ENDP
```

USES Operator

- Lists the registers that will be preserved

```
ArraySum PROC USES esi ecx
    mov eax,0                                ; set the sum to zero
    etc.
```

MASM generates the code shown in gold:

```
ArraySum PROC
    push esi
    push ecx
    .
    .
    pop ecx
    pop esi
    ret
ArraySum ENDP
```

USES Example (1/4)

```
main PROC
```

```
    call MySub
```

```
    exit
```

```
main ENDP
```

```
MySub PROC USES eax
```

```
    mov eax, edx
```

```
    ret
```

```
MySub ENDP
```

```
main PROC
```

```
    call MySub
```

```
    exit
```

```
main ENDP
```

```
MySub PROC
```

```
    mov eax, edx
```

```
    ret
```

```
MySub ENDP
```

Values of registers before calling MySub:

EAX

0000000A

EBX

7EF00000

ECX

00000000

EDX

00401065

ESI

00000000

EDI

00000000

EBP

0018FF94

ESP

0018FF8C

USES Example (2/4)

```
MySub PROC USES eax  
    mov eax, edx  
    ret  
MySub ENDP
```

Values of registers:

EAX	0000000A	ESI	00000000
EBX	7EF00000	EDI	00000000
ECX	00000000	EBP	0018FF94
EDX	00401065	ESP	0018FF84

Stack Memory:

0018FF84	0000000A
0018FF88	00401078
0018FF8C	00756D33

Return Address

```
MySub PROC  
    mov eax, edx  
    ret  
MySub ENDP
```

Values of registers:

EAX	0000000A	ESI	00000000
EBX	7EF00000	EDI	00000000
ECX	00000000	EBP	0018FF94
EDX	00401065	ESP	0018FF88

Stack Memory:

0018FF84	?
0018FF88	00401078
0018FF8C	00756D33

USES Example (3/4)

```
MySub PROC USES eax  
    mov eax, edx  
    ret  
MySub ENDP
```

Values of registers:

EAX	00401065	ESI	00000000
EBX	7EF00000	EDI	00000000
ECX	00000000	EBP	0018FF94
EDX	00401065	ESP	0018FF84

Stack Memory:

0018FF84	0000000A
0018FF88	00401078
0018FF8C	00756D33

```
MySub PROC  
    mov eax, edx  
    ret  
MySub ENDP
```

Values of registers:

EAX	00401065	ESI	00000000
EBX	7EF00000	EDI	00000000
ECX	00000000	EBP	0018FF94
EDX	00401065	ESP	0018FF88

Stack Memory:

0018FF84	?
0018FF88	00401078
0018FF8C	00756D33

USES Example (4/4)

```
main PROC  
    call MySub  
    exit  
main ENDP
```

Values of registers:

EAX	0000000A	ESI	00000000
EBX	7EF00000	EDI	00000000
ECX	00000000	EBP	0018FF94
EDX	00401065	ESP	0018FF8C

Stack Memory:

0018FF84	?
0018FF88	?
0018FF8C	00756D33

```
main PROC  
    call MySub  
    exit  
main ENDP
```

Values of registers:

EAX	00401065	ESI	00000000
EBX	7EF00000	EDI	00000000
ECX	00000000	EBP	0018FF94
EDX	00401065	ESP	0018FF8C

Stack Memory:

0018FF84	?
0018FF88	?
0018FF8C	00756D33

When not to push a register

The sum of the three registers is stored in EAX on line (3), but the POP instruction replaces it with the starting value of EAX on line (4):

```
SumOf PROC                ; sum of three integers
    push eax                ; 1
    add eax,ebx              ; 2
    add eax,ecx              ; 3
    pop eax                 ; 4
    ret
SumOf ENDP
```

What's Next

- Stack Operations
- Defining and Using Procedures
- **Linking to an External Library**
- The Irvine32 Library
- 64-Bit Assembly Programming

Linking to an External Library

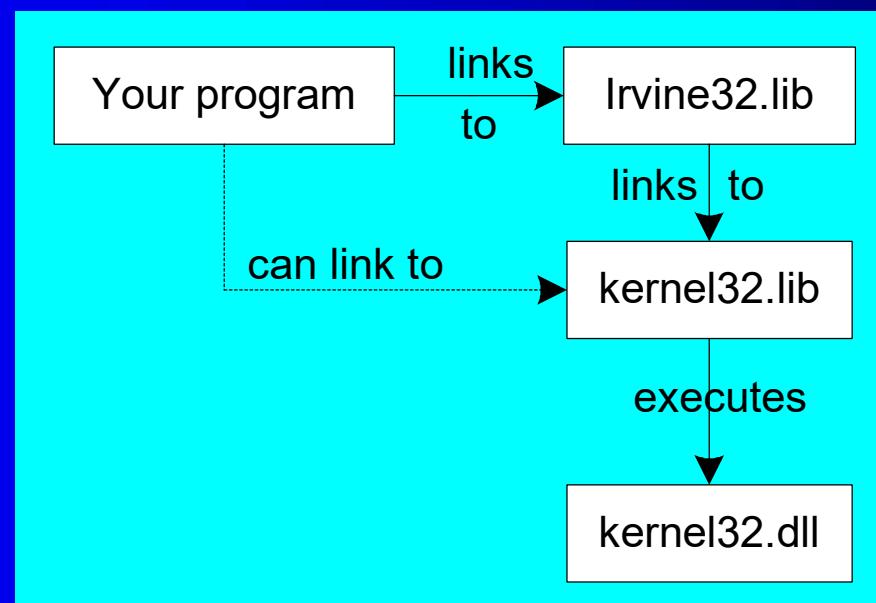
- What is a Link Library?
- How the Linker Works

What is a Link Library?

- A file containing procedures that have been compiled into machine code
 - constructed from one or more OBJ files
- To build a library, . . .
 - start with one or more ASM source files
 - assemble each into an OBJ file
 - create an empty library file (extension .LIB)
 - add the OBJ file(s) to the library file, using the Microsoft LIB utility

How The Linker Works

- Your programs link to Irvine32.lib using the linker command inside a batch file named make32.bat.
- Notice the two LIB files: Irvine32.lib, and kernel32.lib
 - the latter is part of the Microsoft *Win32 Software Development Kit (SDK)*



What's Next

- Stack Operations
- Defining and Using Procedures
- Linking to an External Library
- **The Irvine32 Library**
- 64-Bit Assembly Programming

The Irvine32 Library

- Calling Irvine32 Library Procedures
- Library Procedures – Overview
- Six Examples

Calling Irvine32 Library Procedures

- Call each procedure using the CALL instruction. Some procedures require input arguments. The INCLUDE directive copies in the procedure prototypes (declarations).
- The following example displays "1234" on the console:

```
INCLUDE Irvine32.inc
.code
    mov  eax,1234h          ; input argument
    call WriteHex           ; show hex number
    call Crlf               ; end of line
```

Library Procedures - Overview (1 of 5)

CloseFile – Closes an open disk file

Clscr - Clears console, locates cursor at upper left corner

CreateOutputFile - Creates new disk file for writing in output mode

Crlf - Writes end of line sequence to standard output

Delay - Pauses program execution for *n* millisecond interval

DumpMem - Writes block of memory to standard output in hex

DumpRegs – Displays general-purpose registers and flags (hex)

GetCommandtail - Copies command-line args into array of bytes

GetDateTime – Gets the current date and time from the system

GetMaxXY - Gets number of cols, rows in console window buffer

GetMseconds - Returns milliseconds elapsed since midnight

Library Procedures - Overview (2 of 5)

`GetTextColor` - Returns active foreground and background text colors in the console window

`Gotoxy` - Locates cursor at row and column on the console

`IsDigit` - Sets Zero flag if AL contains ASCII code for decimal digit (0–9)

`MsgBox`, `MsgBoxAsk` – Display popup message boxes

`OpenInputFile` – Opens existing file for input

`ParseDecimal32` – Converts unsigned integer string to binary

`ParseInteger32` - Converts signed integer string to binary

`Random32` - Generates 32-bit pseudorandom integer in the range 0 to `FFFFFFFFFFh`

`Randomize` - Seeds the random number generator

`RandomRange` - Generates a pseudorandom integer within a specified range

`ReadChar` - Reads a single character from standard input

Library Procedures - Overview (3 of 5)

ReadDec - Reads 32-bit unsigned decimal integer from keyboard

ReadFromFile – Reads input disk file into buffer

ReadHex - Reads 32-bit hexadecimal integer from keyboard

ReadInt - Reads 32-bit signed decimal integer from keyboard

ReadKey – Reads character from keyboard input buffer

ReadString - Reads string from stdin, terminated by [Enter]

SetTextColor - Sets foreground/background colors of all subsequent text output to the console

Str_compare – Compares two strings

Str_copy – Copies a source string to a destination string

Str_length – Returns the length of a string in EAX

Str_trim - Removes unwanted characters from a string.

Library Procedures - Overview (4 of 5)

Str_upper - Converts a string to uppercase letters.

WaitMsg - Displays message, waits for Enter key to be pressed

WriteBin - Writes unsigned 32-bit integer in ASCII binary format.

WriteBinB – Writes binary integer in byte, word, or doubleword format

WriteChar - Writes a single character to standard output

WriteDec - Writes unsigned 32-bit integer in decimal format

WriteHex - Writes an unsigned 32-bit integer in hexadecimal format

WriteHexB – Writes byte, word, or doubleword in hexadecimal format

WriteInt - Writes signed 32-bit integer in decimal format

Library Procedures - Overview (5 of 5)

`WriteStackFrame` - Writes the current procedure's stack frame to the console.

`WriteStackFrameName` - Writes the current procedure's name and stack frame to the console.

`WriteString` - Writes null-terminated string to console window

`WriteToFile` - Writes buffer to output file

`WriteWindowsMsg` - Displays most recent error message generated by MS-Windows

Example 1

Clear the screen, delay the program for 500 milliseconds, and dump the registers and flags.

```
.code
    call Clrscr
    mov  eax,500
    call Delay
    call DumpRegs
```

Example 1 - Sample Output

The image shows two separate command-line windows side-by-side.

The left window displays the output of the Microsoft Macro Assembler (MASM) Version 6.11. It includes the following text:

```
找不到 E:\AssemblyLanguage\Mine\GN_test\SlideSample.pdb
找不到 E:\AssemblyLanguage\Mine\GN_test\SlideSample.obj
找不到 E:\AssemblyLanguage\Mine\GN_test\SlideSample.exe
Microsoft (R) Macro Assembler Version 6.11
Copyright (C) Microsoft Corp 1981-1993. All rights reserved.

Assembling: SlideSample.asm
Microsoft (R) 32-Bit Incremental Linker Version 6.11
Copyright (C) Microsoft Corp 1981-1993. All rights reserved.

磁碟區 E 中的磁碟是 Data
磁碟區序號: 5229-B498

E:\AssemblyLanguage\Mine\GN_
2015/02/10 下午 12:13
2015/02/10 下午 12:14
2015/02/10 下午 12:14
2015/02/10 下午 12:14
2015/02/10 下午 12:14
      5 個檔案
      0 個目錄 28,
請按任意鍵繼續 . . .
```

The right window displays assembly statistics. It includes the following CPU register values:

```
EAX=000001F4 EBX=7EFDE000 ECX=00000000 EDX=00401000
ESI=00000000 EDI=00000000 EBP=0018FF94 ESP=0018FF8C
EIP=00401014 EFL=00000246 CF=0 SF=0 ZF=1 OF=0 AF=0 PF=1
```

Below the registers, it shows the current directory and a prompt:

```
E:\AssemblyLanguage\Mine\GN_test>_
```

Example 2

Display a null-terminated string and move the cursor to the beginning of the next screen line.

```
.data  
str1 BYTE "Assembly language is easy!",0  
  
.code  
    mov  edx,OFFSET str1  
    call WriteString  
    call Crlf
```

Sample output:

```
E:\AssemblyLanguage\Mine\GN_test>SlideSample  
Assembly language is easy!  
  
E:\AssemblyLanguage\Mine\GN_test>_
```

Example 2a

Display a null-terminated string and move the cursor to the beginning of the next screen line (use embedded CR/LF)

```
.data  
str1 BYTE "Assembly language is easy!", 0Dh, 0Ah, 0  
  
.code  
    mov  edx,OFFSET str1  
    call WriteString
```

Sample output:

```
E:\AssemblyLanguage\Mine\GN_test>SlideSample  
Assembly language is easy!  
  
E:\AssemblyLanguage\Mine\GN_test>_
```

Example 3

Display an unsigned integer in binary, decimal, and hexadecimal, each on a separate line.

```
IntVal = 35
.code
    mov  eax,IntVal
    call WriteBin          ; display binary
    call Crlf
    call WriteDec          ; display decimal
    call Crlf
    call WriteHex          ; display hexadecimal
    call Crlf
```

Sample output:

```
0000 0000 0000 0000 0000 0000 0010 0011
35
23
```

Example 4

Input a string from the user. EDX points to the string and ECX specifies the maximum number of characters the user is permitted to enter.

```
.data  
fileName BYTE 80 DUP(0)  
  
.code  
    mov edx,OFFSET fileName  
    mov ecx,SIZEOF fileName - 1  
    call ReadString
```

A null byte is automatically appended to the string.

Example 4 - Sample Output

```
E:\AssemblyLanguage\Mine\GN_test>SlideSample  
Call Procedure Example 4: ReadString.
```

Example 5

Generate and display ten pseudorandom signed integers in the range 0 – 99. Pass each integer to WriteInt in EAX and display it on a separate line.

```
.code
    mov ecx,10          ; loop counter

L1: mov eax,100        ; ceiling value
    call RandomRange   ; generate random int
    call WriteInt       ; display signed int
    call Crlf           ; goto next display line
    loop L1             ; repeat loop
```

Example 5 - Sample Output

```
E:\AssemblyLanguage\Mine\GN_test>SlideSample  
+94  
+2  
+67  
+57  
+7  
+40  
+58  
+48  
+73  
+94
```

Example 6

Display a null-terminated string with yellow characters on a blue background.

```
.data  
str1 BYTE "Color output is easy!",0  
  
.code  
    mov  eax,yellow + (blue * 16)  
    call SetTextColor  
    mov  edx,OFFSET str1  
    call WriteString  
    call Crlf
```

The background color is multiplied by 16 before being added to the foreground color.

Example 6 - Sample Output

```
E:\AssemblyLanguage\Mine\GN_test>SlideSample  
Color output is easy!
```

```
E:\AssemblyLanguage\Mine\GN_test>
```

What's Next

- Stack Operations
- Defining and Using Procedures
- Linking to an External Library
- The Irvine32 Library
- **64-Bit Assembly Programming**

64-Bit Assembly Programming

- The Irvine64 Library
- Calling 64-Bit Subroutines
- The x64 Calling Convention

The Irvine64 Library

- Crlf: Writes an end-of-line sequence to the console.
- Random64: Generates a 64-bit pseudorandom integer.
- Randomize: Seeds the random number generator with a unique value.
- ReadInt64: Reads a 64-bit signed integer from the keyboard.
- ReadString: Reads a string from the keyboard.
- Str_compare: Compares two strings in the same way as the CMP instruction.
- Str_copy: Copies a source string to a target location.
- Str_length: Returns the length of a null-terminated string in RAX.
- WriteInt64: Displays the contents in the RAX register as a 64-bit signed decimal integer.

The Irvine64 Library (cont'd)

- `WriteHex64`: Displays the contents of the RAX register as a 64-bit hexadecimal integer.
- `WriteHexB`: Displays the contents of the RAX register as an 8-bit hexadecimal integer .
- `WriteString`: Displays a null-terminated ASCII string.

Calling 64-Bit Subroutines

- Place the first four parameters in registers
- Add PROTO directives at the top of your program
 - examples:

```
ExitProcess PROTO      ; located in the Windows API  
WriteHex64 PROTO       ; located in the Irvine64 library
```

The x64 Calling Convention

- Must use this with the 64-bit Windows API
- CALL instruction subtracts 8 from RSP
- First four parameters must be placed in RCX, RDX, R8, and R9
- Caller must allocate at least 32 bytes of shadow space on the stack
- When calling a subroutine, the stack pointer must be aligned on a 16-byte boundary.

See the CallProc_64.asm example program.

Assembly Language for x86 Processors

7th Edition , Global Edition

Kip R. Irvine

Chapter 6: Conditional Processing

Slides prepared by the author

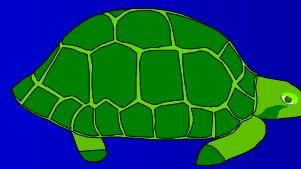
Revision date: 1/15/2014

Chapter Overview

- Boolean and Comparison Instructions
- Conditional Jumps
- Conditional Loop Instructions
- Conditional Structures
- Application: Finite-State Machines
- Conditional Control Flow Directives

Summary

- Bitwise instructions (AND, OR, XOR, NOT, TEST)
 - manipulate individual bits in operands
- CMP – compares operands using implied subtraction
 - sets condition flags
- Conditional Jumps & Loops
 - equality: JE, JNE
 - flag values: JC, JZ, JNC, JP, ...
 - signed: JG, JL, JNG, ...
 - unsigned: JA, JB, JNA, ...
 - LOOPZ, LOOPNZ, LOOPE, LOOPNE
- Flowcharts – logic diagramming tool
- Finite-state machine – tracks state changes at runtime



4C 6F 70 70 75 75 6E

Boolean and Comparison Instructions

- CPU Status Flags
- AND Instruction
- OR Instruction
- XOR Instruction
- NOT Instruction
- Applications
- TEST Instruction
- CMP Instruction
- Boolean Instructions in 64-Bit Mode

Status Flags - Review

- The **Zero flag** is set when the result of an operation equals zero.
- The **Carry flag** is set when an instruction generates a result that is too large (or too small) for the destination operand.
- The **Sign flag** is set if the destination operand is negative, and it is clear if the destination operand is positive.
- The **Overflow flag** is set when an instruction generates an invalid signed result (bit 7 carry is XORed with bit 6 Carry).
- The **Parity flag** is set when an instruction generates an even number of 1 bits in the low byte of the destination operand.
- The **Auxiliary Carry flag** is set when an operation produces a carry out from bit 3 to bit 4

AND Instruction

- Performs a Boolean AND operation between each pair of matching bits in two operands
- Syntax:

AND *destination, source*

(same operand types as MOV)

	00111011
AND	00001111
	<hr/>
cleared	0000 1011
	unchanged

AND

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

Applications

OR Instruction

- Performs a Boolean OR operation between each pair of matching bits in two operands
- Syntax:

`OR destination, source`

		0 0 1 1 1 0 1 1
OR		0 0 0 0 1 1 1 1
unchanged		0 0 1 1 1 1 1 1 set

OR

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

Applications

XOR Instruction

- Performs a Boolean exclusive-OR operation between each pair of matching bits in two operands
- Syntax:

`XOR destination, source`

$$\begin{array}{r} 00111011 \\ \text{XOR } 00001111 \\ \hline \text{unchanged } 0011\boxed{0}100 \quad \text{inverted} \end{array}$$

XOR

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

Encrypting a String

XOR is a useful way to toggle (invert) the bits in an operand.

NOT Instruction

- Performs a Boolean NOT operation on a single destination operand
- Syntax:

NOT *destination*

NOT	0 0 1 1 1 0 1 1
<hr/>	
	1 1 0 0 0 1 0 0 —— inverted

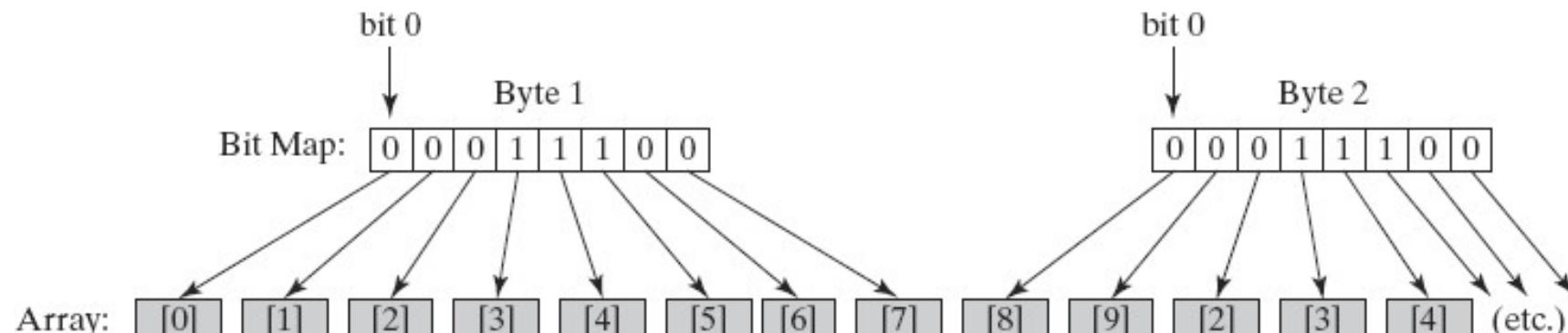
NOT

X	$\neg X$
F	T
T	F

Bit-Mapped Sets

- Binary bits indicate set membership
- Efficient use of storage
- Also known as *bit vectors*

FIGURE 6–1 Mapping Binary Bits to an Array.



Bit-Mapped Set Operations

- Set Complement

`mov eax, SetX`

`not eax`

- Set Intersection

`mov eax, setX`

`and eax, setY`

- Set Union

`mov eax, setX`

`or eax, setY`

Applications (1 of 5)

- Task: Convert the character in AL to upper case.

ASCII Code:

	Decimal	Binary		Decimal	Binary
'A'	65	01000001	'a'	97	01100001
'B'	66	01000010	'b'	98	01100010
'C'	67	01000011	'c'	99	01100011
'D'	68	01000100	'd'	100	01100100
'E'	69	01000101	'e'	101	01100101

- Solution: Use the AND instruction to clear bit 5.

```
mov al,'a' ; AL = 01100001b  
and al,11011111b ; AL = 01000001b
```

- 'a' : 01100001b
- 'A' : 01000001b

Applications (2 of 5)

- Task: Convert a binary decimal byte into its equivalent ASCII decimal digit.
- Solution: Use the OR instruction to set bits 4 and 5.

```
mov al,6           ; AL = 00000110b
or  al,00110000b   ; AL = 00110110b
```

The ASCII digit '6' = 00110110b

Applications (3 of 5)

- Task: Turn on the keyboard CapsLock key
- Solution: Use the OR instruction to set bit 6 in the keyboard flag byte at 0040:0017h in the BIOS data area.

```
mov ax,40h          ; BIOS segment
mov ds,ax
mov bx,17h          ; keyboard flag byte
or BYTE PTR [bx],01000000b ; CapsLock on
```

This code only runs in Real-address mode, and it does not work under Windows NT, 2000, or XP.

Applications (4 of 5)

- Task: Jump to a label if an integer is even.

Decimal	Binary	Decimal	Binary	Decimal	Binary
1	00000001	5	00000101	9	00001001
2	00000010	6	00000110	10	00001010
3	00000011	7	00000111	11	00001011
4	00000100	8	00001000	12	00001100

- Solution: AND the lowest bit with a 1. If the result is Zero, the number was even.

```
mov ax,wordVal  
and ax,1           ; low bit set?  
jz EvenValue      ; jump if Zero flag set
```

JZ (jump if Zero) is covered in Section 6.3.

Your turn: Write code that jumps to a label if an integer is negative.

Applications (5 of 5)

- Task: Jump to a label if the value in AL is not zero.
- Solution: OR the byte with itself, then use the JNZ (jump if not zero) instruction.

```
or  al,al  
jnz IsNotZero      ; jump if not zero
```

ORing any number with itself does not change its value.

TEST Instruction

- Performs a nondestructive AND operation between each pair of matching bits in two operands
- No operands are modified, but the Zero flag is affected.
- Usually used with jz (jump if ZF = 1) / jnz (jump if ZF = 0)
- Example: jump to a label if either bit 0 or bit 1 in AL is set.

```
test al,00000011b ; ZF = 0, if either bit0 or bit1 is set  
jnz ValueFound      ; jump if ZF = 0
```

- Example: jump to a label if bit 0 and bit 1 in AL are clear.

```
test al,00000011b ; ZF = 1, if both bit0 and bit1 are clear  
jz  ValueNotFound ; jump if ZF = 1
```

CMP Instruction (1 of 3)

- Compares the destination operand to the source operand
 - Nondestructive subtraction of source from destination (destination operand is not changed)
- Syntax: **CMP** *destination, source*
- Example: *destination == source*

```
mov al,5  
cmp al,5           ; Zero flag set
```

- Example: *destination < source*

```
mov al,4  
cmp al,5           ; Carry flag set
```

CMP Instruction (2 of 3)

- Example: destination > source

```
mov al, 6  
cmp al, 5 ; ZF = 0, CF = 0
```

(both the Zero and Carry flags are clear)

Unsigned Integer

Case	ZF	CF	
D = S	1	-	$D - S = 0$
D < S	0	1	$D - S < 0$ (Unsigned Overflow)
D > S	0	0	$D - S > 0$

CMP Instruction (3 of 3)

The comparisons shown here are performed with signed integers.

- Example: destination > source

```
mov al,5  
cmp al,-2 ; Sign flag == Overflow flag
```

- Example: destination < source

```
mov al,-1  
cmp al,5 ; Sign flag != Overflow flag
```

Signed Integer

Case	D	S	SF	OF
D > S	0101 (5)	1110 (-2)	0	0
	0110 (6)	1110(-2)	1	1
D < S	1110 (-2)	0111 (7)	0	1
	1111 (-1)	0101 (5)	1	0

Boolean Instructions in 64-Bit Mode

- 64-bit boolean instructions, for the most part, work the same as 32-bit instructions
- If the source operand is a constant whose size is less than 32 bits and the destination is the lower part of a 64-bit register or memory operand, all bits in the destination operand are affected
- When the source is a 32-bit constant or register, only the lower 32 bits of the destination operand are affected

What's Next

- Boolean and Comparison Instructions
- **Conditional Jumps**
- Conditional Loop Instructions
- Conditional Structures
- Application: Finite-State Machines
- Conditional Control Flow Directives

Conditional Jumps

- Jcond Instruction
- Jumps Based On . . .
 - Specific flags (not suggest to use)
 - Equality
 - Unsigned comparisons
 - Signed Comparisons
- Applications
- Encrypting a String
- Bit Test (BT) Instruction

Jcond Instruction

- A conditional jump instruction branches to a label when specific register or flag conditions are met
- Specific jumps:
 - JB, JC - jump to a label if the Carry flag is set
 - JE, JZ - jump to a label if the Zero flag is set
 - JS - jump to a label if the Sign flag is set
 - JNE, JNZ - jump to a label if the Zero flag is clear
 - JECXZ - jump to a label if ECX = 0

Jcond Ranges

- Prior to the 386:
 - jump must be within –128 to +127 bytes from current location counter
- x86 processors:
 - 32-bit offset permits jump anywhere in memory

Jumps Based on Specific Flags

CMP leftop, rightop
JZ LabelName

Mnemonic	Description	Flags
JZ	Jump if zero	ZF = 1
JNZ	Jump if not zero	ZF = 0
JC	Jump if carry	CF = 1
JNC	Jump if not carry	CF = 0
JO	Jump if overflow	OF = 1
JNO	Jump if not overflow	OF = 0
JS	Jump if signed	SF = 1
JNS	Jump if not signed	SF = 0
JP	Jump if parity (even)	PF = 1
JNP	Jump if not parity (odd)	PF = 0

NOT SUGGEST TO USE!!!!

Jumps Based on Equality

CMP leftop, rightop
JE LabelName

Mnemonic	Description
JE	Jump if equal ($leftOp = rightOp$)
JNE	Jump if not equal ($leftOp \neq rightOp$)
JCXZ	Jump if CX = 0
JECXZ	Jump if ECX = 0

Jumps Based on Unsigned Comparisons

CMP leftop, rightop
JA LabelName

Mnemonic	Description
JA	Jump if above (if $leftOp > rightOp$)
JNBE	Jump if not below or equal (same as JA)
JAE	Jump if above or equal (if $leftOp \geq rightOp$)
JNB	Jump if not below (same as JAE)
JB	Jump if below (if $leftOp < rightOp$)
JNAE	Jump if not above or equal (same as JB)
JBE	Jump if below or equal (if $leftOp \leq rightOp$)
JNA	Jump if not above (same as JBE)

Note that 'a': 61h, 'A': 41h, 'B':42h
Hence, 'a' > 'B' > 'A'

Jumps Based on Signed Comparisons

CMP leftop, rightop
JG LabelName

Mnemonic	Description
JG	Jump if greater (if $leftOp > rightOp$)
JNLE	Jump if not less than or equal (same as JG)
JGE	Jump if greater than or equal (if $leftOp \geq rightOp$)
JNL	Jump if not less (same as JGE)
JL	Jump if less (if $leftOp < rightOp$)
JNGE	Jump if not greater than or equal (same as JL)
JLE	Jump if less than or equal (if $leftOp \leq rightOp$)
JNG	Jump if not greater (same as JLE)

Applications (1 of 5)

- Task: Jump to a label if unsigned EAX is greater than EBX
- Solution: Use CMP, followed by JA

```
cmp eax,ebx  
ja Larger
```

- Task: Jump to a label if signed EAX is greater than EBX
- Solution: Use CMP, followed by JG

```
cmp eax,ebx  
jg Greater
```

Example

Application 1 – Example (1/2)

- Jump to Larger if unsigned EAX > EBX

```
mov eax, 0FFFFFFFh  
mov ebx, 0  
cmp eax,ebx  
ja Larger ; jump if unsignd eax > ebx
```

- Unsigned integer (EAX) = 4294967295

- Jump to Greater if signed EAX > EBX

```
mov eax, 0FFFFFFFh  
mov ebx, 0  
cmp eax,ebx  
jg Greater ; jump if signd eax > ebx
```

- Signed integer (EAX) = -1

Application 1 – Example (2/2)

```
mov eax, 0FFFFFFFh  
mov ebx, 0  
cmp eax,ebx
```

- After comparing, flags are:

ZF	CF	SF	OF
0	0	1	0

; (same as sub eax, ebx)

- ja and jg do jump while flags are set as below:

	ZF	CF	SF	OF
ja	0	0	-	-
jg	0	-	SF == OF	

; jump if unsigned leftop > rightop

; jump if signed leftop > right

Applications (2 of 5)

- Jump to label L1 if unsigned EAX is less than or equal to Val1

```
cmp eax,Val1  
jbe L1           ; below or equal
```

- Jump to label L1 if signed EAX is less than or equal to Val1

```
cmp eax,Val1  
jle L1
```

Applications (3 of 5)

- Compare unsigned AX to BX, and copy the larger of the two into a variable named Large

```
mov Large,bx  
cmp ax,bx  
jna Next  
mov Large,ax
```

Next:

- Compare signed AX to BX, and copy the smaller of the two into a variable named Small

```
mov Small,ax  
cmp bx,ax  
jnl Next  
mov Small,bx
```

Next:

Applications (4 of 5)

- Jump to label L1 if the memory word pointed to by ESI equals Zero

```
cmp WORD PTR [esi],0  
je L1
```

- Jump to label L2 if the doubleword in memory pointed to by EDI is even

```
test DWORD PTR [edi],1  
jz L2
```

Applications (5 of 5)

- Task: Jump to label L1 if bits 0, 1, and 3 in AL are all set.
- Solution: Clear all bits except bits 0, 1, and 3. Then compare the result with 00001011 binary.

```
and al,00001011b      ; clear unwanted bits
cmp al,00001011b      ; check remaining bits
je L1                  ; all set? jump to L1
```

Your turn . . .

- Write code that jumps to label L1 if either bit 4, 5, or 6 is set in the BL register.
- Write code that jumps to label L1 if bits 4, 5, and 6 are all set in the BL register.
- Write code that jumps to label L2 if AL has even parity.
- Write code that jumps to label L3 if EAX is negative.
- Write code that jumps to label L4 if the expression $(EBX - ECX)$ is greater than zero.

Encrypting a String

The following loop uses the XOR instruction to transform every character in a string into a new value.

```
KEY = 239 ; can be any byte value
BUFMAX = 128
.data
buffer BYTE BUFMAX+1 DUP(0)
bufSize DWORD BUFMAX

.code
    mov ecx,bufSize ; loop counter
    mov esi,0 ; index 0 in buffer
L1:
    xor buffer[esi],KEY ; translate a byte
    inc esi ; point to next byte
    loop L1
```

String Encryption Program

- Tasks:
 - Input a message (string) from the user
 - Encrypt the message
 - Display the encrypted message
 - Decrypt the message
 - Display the decrypted message

View the [Encrypt.asm](#) program's source code. Sample output:

```
Enter the plain text: Attack at dawn.  
Cipher text: <<¢¢Äiä-Ä¢-iÄyü-Gs  
Decrypted: Attack at dawn.
```

BT (Bit Test) Instruction

- Copies bit *n* from an operand into the Carry flag
- Syntax: BT *bitBase*, *n*
 - *bitBase* may be *r/m16* or *r/m32*
 - *n* may be *r16*, *r32*, or *imm8*
- Example: jump to label L1 if bit 9 is set in the AX register:

```
bt AX,9          ; CF = bit 9
jc L1           ; jump if Carry
```

What's Next

- Boolean and Comparison Instructions
- Conditional Jumps
- **Conditional Loop Instructions**
- Conditional Structures
- Application: Finite-State Machines
- Conditional Control Flow Directives

Conditional Loop Instructions

- LOOPZ and LOOPE
- LOOPNZ and LOOPNE

LOOPZ and LOOPE

- Syntax:
 - LOOPE *destination*
 - LOOPZ *destination*
- Logic:
 - $ECX \leftarrow ECX - 1$
 - if $ECX > 0$ and $ZF=1$, jump to *destination*
 - Useful when scanning an array for the first element that does not match a given value.

In 32-bit mode, ECX is the loop counter register. In 16-bit real-address mode, CX is the counter, and in 64-bit mode, RCX is the counter.

LOOPNZ and LOOPNE

- LOOPNZ (LOOPNE) is a conditional loop instruction
- Syntax:

 LOOPNZ *destination*

 LOOPNE *destination*

- Logic:
 - $ECX \leftarrow ECX - 1;$
 - if $ECX > 0$ and $ZF=0$, jump to *destination*
- Useful when scanning an array for the first element that matches a given value.

LOOPNZ Example

The following code finds the first positive value in an array:

```
.data
array SWORD -3,-6,-1,-10,10,30,40,4
sentinel SWORD 0
.code
    mov esi,OFFSET array
    mov ecx,LENGTHOF array
next:
    test WORD PTR [esi],8000h ; test sign bit
    pushfd                   ; push flags on stack
    add esi,TYPE array
    popfd                    ; pop flags from stack
    loopnz next              ; continue loop
    jnz quit                 ; none found
    sub esi,TYPE array       ; ESI points to value
quit:
```

Your turn . . .

Locate the first nonzero value in the array. If none is found, let ESI point to the sentinel value:

```
.data
array SWORD 50 DUP(?)
sentinel SWORD 0FFFFh
.code
    mov esi,OFFSET array
    mov ecx,LENGTHOF array
L1: cmp WORD PTR [esi],0 ; check for zero
```

(fill in your code here)

quit:

. . . (solution)

```
.data
array SWORD 50 DUP(?)
sentinel SWORD 0FFFFh
.code
    mov esi,OFFSET array
    mov ecx,LENGTHOF array
L1: cmp WORD PTR [esi],0           ; check for zero
    pushfd                         ; push flags on stack
    add esi,TYPE array
    popfd                           ; pop flags from stack
    loope L1                        ; continue loop
    jz quit                          ; none found
    sub esi,TYPE array              ; ESI points to value
quit:
```

What's Next

- Boolean and Comparison Instructions
- Conditional Jumps
- Conditional Loop Instructions
- **Conditional Structures**
- Application: Finite-State Machines
- Conditional Control Flow Directives

Conditional Structures

- Block-Structured IF Statements
- Compound Expressions with AND
- Compound Expressions with OR
- WHILE Loops
- Table-Driven Selection

Block-Structured IF Statements

Assembly language programmers can easily translate logical statements written in C++/Java into assembly language. For example:

```
if( op1 == op2 )  
    x = 1;  
else  
    x = 2;
```

```
mov eax,op1  
cmp eax,op2  
jne L1  
mov x,1  
jmp L2  
L1: mov x,2  
L2:
```

Your turn . . .

Implement the following pseudocode in assembly language. All values are unsigned:

```
if( ebx <= ecx )
{
    eax = 5;
    edx = 6;
}
```

```
cmp ebx,ecx
ja next
mov eax,5
mov edx,6
next:
```

(There are multiple correct solutions to this problem.)

Your turn . . .

Implement the following pseudocode in assembly language. All values are 32-bit signed integers:

```
if( var1 <= var2 )
    var3 = 10;
else
{
    var3 = 6;
    var4 = 7;
}
```

```
mov eax,var1
cmp eax,var2
jle L1
mov var3,6
mov var4,7
jmp L2
L1: mov var3,10
L2:
```

(There are multiple correct solutions to this problem.)

Compound Expression with AND (1 of 3)

- When implementing the logical AND operator, consider that HLLs use short-circuit evaluation
- In the following example, if the first expression is false, the second expression is skipped:

```
if (al > bl) AND (bl > cl)  
    x = 1;
```



Compound Expression with AND (2 of 3)

```
if (al > bl) AND (bl > cl)
    x = 1;
```

This is one possible implementation . . .

```
cmp al,bl           ; first expression...
ja L1
jmp next
L1:
    cmp bl,cl       ; second expression...
    ja L2
    jmp next
L2:                   ; both are true
    mov x,1          ; set X to 1
next:
```

Compound Expression with AND (3 of 3)

```
if (al > bl) AND (bl > cl)
    x = 1;
```

But the following implementation uses 29% less code by reversing the first relational operator. We allow the program to "fall through" to the second expression:

```
cmp al,bl          ; first expression...
jbe next           ; quit if false
cmp bl,cl          ; second expression...
jbe next           ; quit if false
mov X,1             ; both are true
next:
```

Your turn . . .

Implement the following pseudocode in assembly language. All values are unsigned:

```
if( ebx <= ecx  
    && ecx > edx )  
{  
    eax = 5;  
    edx = 6;  
}
```

```
cmp ebx,ecx  
ja next  
cmp ecx,edx  
jbe next  
mov eax,5  
mov edx,6  
next:
```

(There are multiple correct solutions to this problem.)

Compound Expression with OR (1 of 2)

- When implementing the logical OR operator, consider that HLLs use short-circuit evaluation
- In the following example, if the first expression is true, the second expression is skipped:

```
if (al > bl) OR (bl > cl)  
    x = 1;
```



Compound Expression with OR (2 of 2)

```
if (al > bl) OR (bl > cl)
    x = 1;
```

We can use "fall-through" logic to keep the code as short as possible:

```
cmp al,bl          ; is AL > BL?
ja L1              ; yes
cmp bl,cl          ; no: is BL > CL?
jbe next           ; no: skip next statement
L1: mov x,1         ; set X to 1
next:
```

WHILE Loops

A WHILE loop is really an IF statement followed by the body of the loop, followed by an unconditional jump to the top of the loop. Consider the following example:

```
while( eax < ebx)
    eax = eax + 1;
```

This is a possible implementation:

```
top: cmp eax,ebx          ; check loop condition
     jae next             ; false? exit loop
     inc eax              ; body of loop
     jmp top               ; repeat the loop
next:
```

Your turn . . .

Implement the following loop, using unsigned 32-bit integers:

```
while( ebx <= val1)
{
    ebx = ebx + 5;
    val1 = val1 - 1
}
```

```
top: cmp ebx,val1           ; check loop condition
     ja  next               ; false? exit loop
     add ebx,5              ; body of loop
     dec val1
     jmp top                ; repeat the loop
next:
```

Table-Driven Selection (1 of 4)

- Table-driven selection uses a table lookup to replace a multiway selection structure
- Create a table containing lookup values and the offsets of labels or procedures
- Use a loop to search the table
- Suited to a large number of comparisons

Table-Driven Selection (2 of 4)

Step 1: create a table containing lookup values and procedure offsets:

```
.data
CaseTable BYTE 'A'          ; lookup value
            DWORD Process_A      ; address of procedure
EntrySize = ($ - CaseTable)
BYTE 'B'
DWORD Process_B
BYTE 'C'
DWORD Process_C
BYTE 'D'
DWORD Process_D

NumberOfEntries = ($ - CaseTable) / EntrySize
```

Table-Driven Selection (3 of 4)

Table of Procedure Offsets:

'A'	00000120	'B'	00000130	'C'	00000140	'D'	00000150
							address of Process_B lookup value

Table-Driven Selection (4 of 4)

Step 2: Use a loop to search the table. When a match is found, call the procedure offset stored in the current table entry:

```
mov ebx,OFFSET CaseTable           ; point EBX to the table
mov ecx,NumberOfEntries            ; loop counter

L1: cmp al,[ebx]                  ; match found?
    jne L2
    call NEAR PTR [ebx + 1]         ; yes: call the procedure
    call WriteString               ; display message
    call Crlf
    jmp L3
L2: add ebx,EntrySize             ; and exit the loop
    loop L1
L3:                                ; point to next entry
                                ; repeat until ECX = 0
```

required for
procedure pointers



What's Next

- Boolean and Comparison Instructions
- Conditional Jumps
- Conditional Loop Instructions
- Conditional Structures
- **Application: Finite-State Machines**
- Conditional Control Flow Directives

Application: Finite-State Machines

- A finite-state machine (FSM) is a graph structure that changes state based on some input. Also called a state-transition diagram.
- We use a graph to represent an FSM, with squares or circles called nodes, and lines with arrows between the circles called edges.

Application: Finite-State Machines

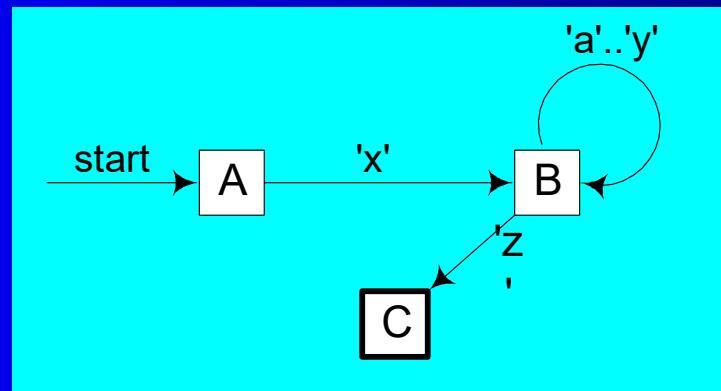
- A FSM is a specific instance of a more general structure called a directed graph.
- Three basic states, represented by nodes:
 - Start state
 - Terminal state(s)
 - Nonterminal state(s)

Finite-State Machine

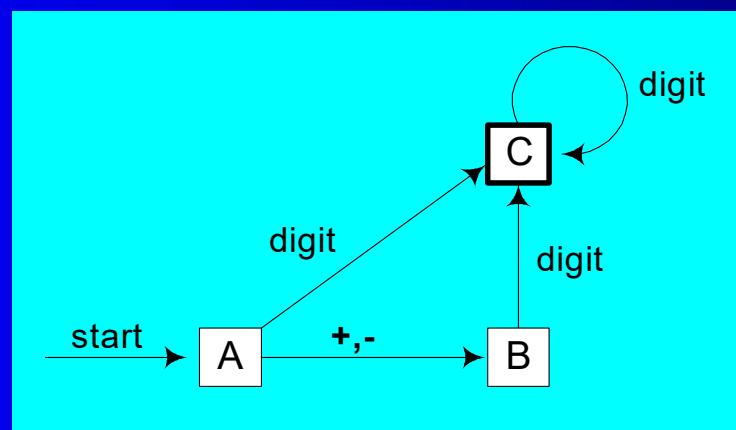
- Accepts any sequence of symbols that puts it into an accepting (final) state
- Can be used to recognize, or validate a sequence of characters that is governed by language rules (called a regular expression)
- Advantages:
 - Provides visual tracking of program's flow of control
 - Easy to modify
 - Easily implemented in assembly language

Finite-State Machine Examples

- FSM that recognizes strings beginning with 'x', followed by letters 'a'..'y', ending with 'z':

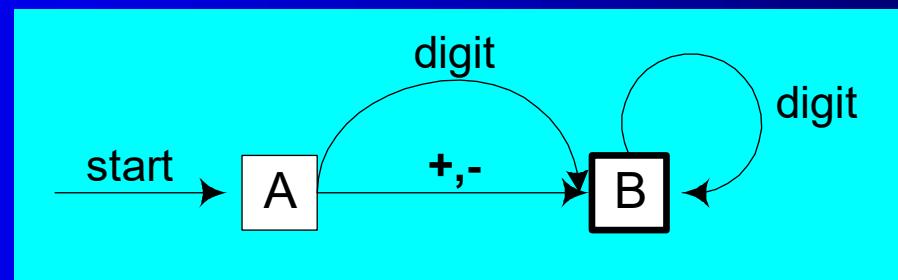


- FSM that recognizes signed integers:



Your Turn . . .

- Explain why the following FSM does not work as well for signed integers as the one shown on the previous slide:



Implementing an FSM

The following is code from State A in the Integer FSM:

StateA:

```
call Getnext          ; read next char into AL
cmp al,'+'           ; leading + sign?
je StateB            ; go to State B
cmp al,'-'           ; leading - sign?
je StateB            ; go to State B
call IsDigit          ; ZF = 1 if AL = digit
jz StateC             ; go to State C
call DisplayErrorMsg ; invalid input found
jmp Quit
```

View the [Finite.asm source code.](#)

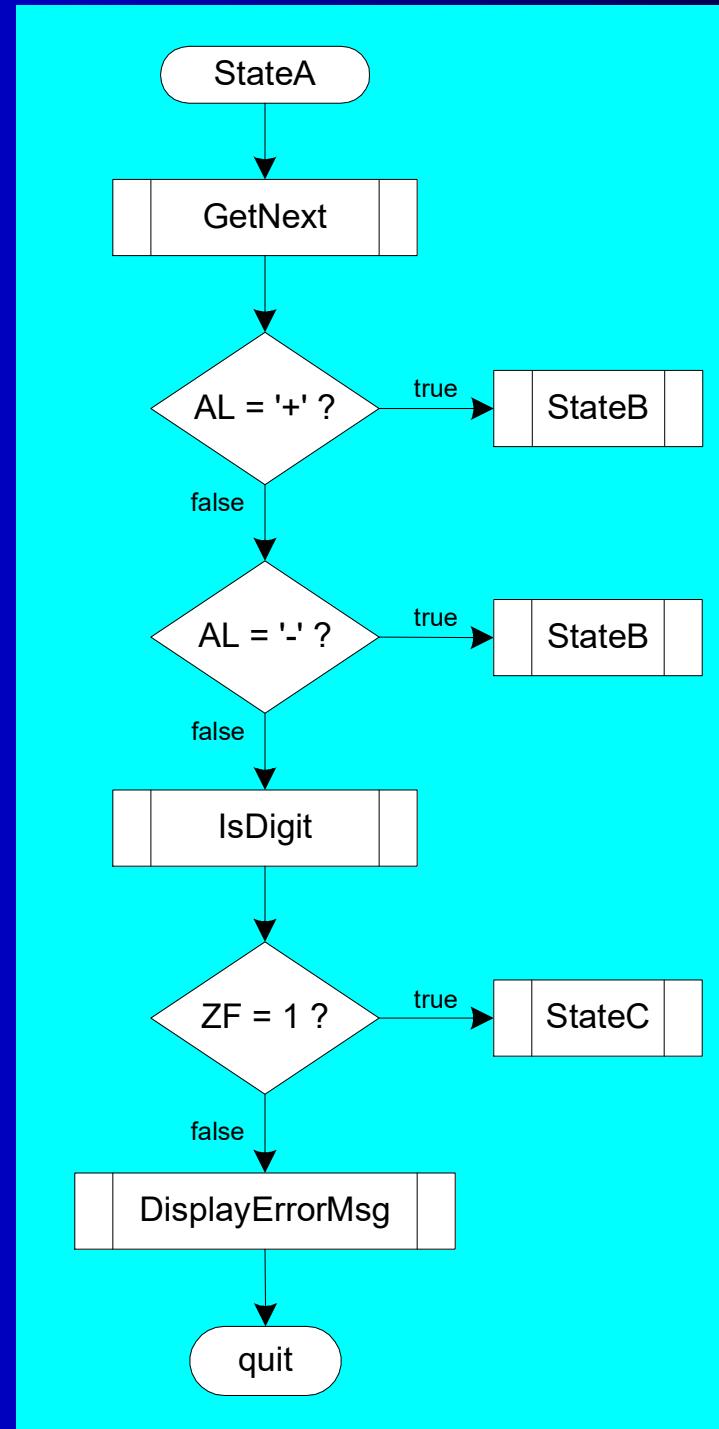
IsDigit Procedure

Receives a character in AL. Sets the Zero flag if the character is a decimal digit.

```
IsDigit PROC
    cmp    al,'0'           ; ZF = 0
    jb     ID1
    cmp    al,'9'           ; ZF = 0
    ja     ID1
    test   ax,0             ; ZF = 1
ID1: ret
IsDigit ENDP
```

Flowchart of State A

State A accepts a plus or minus sign, or a decimal digit.



Your Turn . . .

- Draw a FSM diagram for hexadecimal integer constant that conforms to MASM syntax.
- Draw a flowchart for one of the states in your FSM.
- Implement your FSM in assembly language. Let the user input a hexadecimal constant from the keyboard.

What's Next

- Boolean and Comparison Instructions
- Conditional Jumps
- Conditional Loop Instructions
- Conditional Structures
- Application: Finite-State Machines
- **Conditional Control Flow Directives**

Creating IF Statements

- Runtime Expressions
- Relational and Logical Operators
- MASM-Generated Code
- .REPEAT Directive
- .WHILE Directive

Runtime Expressions

- .IF, .ELSE, .ELSEIF, and .ENDIF can be used to evaluate runtime expressions and create block-structured IF statements.
- Examples:

```
.IF eax > ebx  
    mov edx,1  
.ELSE  
    mov edx,2  
.ENDIF
```

```
.IF eax > ebx && eax > ecx  
    mov edx,1  
.ELSE  
    mov edx,2  
.ENDIF
```

- MASM generates "hidden" code for you, consisting of code labels, CMP and conditional jump instructions.

Relational and Logical Operators

Operator	Description
$expr1 == expr2$	Returns true when $expression1$ is equal to $expr2$.
$expr1 != expr2$	Returns true when $expr1$ is not equal to $expr2$.
$expr1 > expr2$	Returns true when $expr1$ is greater than $expr2$.
$expr1 >= expr2$	Returns true when $expr1$ is greater than or equal to $expr2$.
$expr1 < expr2$	Returns true when $expr1$ is less than $expr2$.
$expr1 <= expr2$	Returns true when $expr1$ is less than or equal to $expr2$.
$! expr$	Returns true when $expr$ is false.
$expr1 \&& expr2$	Performs logical AND between $expr1$ and $expr2$.
$expr1 \ \ expr2$	Performs logical OR between $expr1$ and $expr2$.
$expr1 \& expr2$	Performs bitwise AND between $expr1$ and $expr2$.
CARRY?	Returns true if the Carry flag is set.
OVERFLOW?	Returns true if the Overflow flag is set.
PARITY?	Returns true if the Parity flag is set.
SIGN?	Returns true if the Sign flag is set.
ZERO?	Returns true if the Zero flag is set.

Signed and Unsigned Comparisons

```
.data  
val1    DWORD 5  
result  DWORD ?  
  
.code  
mov eax,6  
.IF eax > val1  
    mov result,1  
.ENDIF
```

Generated code:

```
mov eax,6  
cmp eax,val1  
jbe @C0001  
mov result,1  
@C0001:
```

MASM automatically generates an unsigned jump (JBE) because val1 is unsigned.

Signed and Unsigned Comparisons

```
.data  
val1 SDWORD 5  
result SDWORD ?  
  
.code  
mov eax, 6  
.IF eax > val1  
    mov result, 1  
.ENDIF
```

Generated code:

```
mov eax, 6  
cmp eax, val1  
jle @C0001  
mov result, 1  
@C0001:
```

MASM automatically generates a signed jump (JLE) because **val1** is signed.

Signed and Unsigned Comparisons

```
.data  
result DWORD ?  
  
.code  
mov ebx,5  
mov eax,6  
.IF eax > ebx  
    mov result,1  
.ENDIF
```

Generated code:

```
mov ebx,5  
mov eax,6  
cmp eax,ebx  
jbe @C0001  
mov result,1  
@C0001:
```

MASM automatically generates an unsigned jump (JBE) when both operands are registers . . .

Signed and Unsigned Comparisons

```
.data  
result SDWORD ?  
  
.code  
mov ebx,5  
mov eax,6  
.IF SDWORD PTR eax > ebx  
    mov result,1  
.ENDIF
```

Generated code:

```
mov ebx,5  
mov eax,6  
cmp eax,ebx  
jle @C0001  
mov result,1  
@C0001:
```

... unless you prefix one of the register operands with the SDWORD PTR operator. Then a signed jump is generated.

.REPEAT Directive

Executes the loop body before testing the loop condition associated with the .UNTIL directive.

Example:

```
; Display integers 1 - 10:  
  
mov eax, 0  
.REPEAT  
    inc eax  
    call WriteDec  
    call Crlf  
.UNTIL eax == 10
```

.WHILE Directive

Tests the loop condition before executing the loop body. The .ENDW directive marks the end of the loop.

Example:

```
; Display integers 1 - 10:  
  
mov eax,0  
.WHILE eax < 10  
    inc eax  
    call WriteDec  
    call Crlf  
.ENDW
```

Assembly Language for x86 Processors

7th Edition , Global Edition

Kip R. Irvine

Chapter 7: Integer Arithmetic

Slides prepared by the author

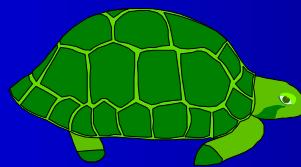
Revision date: 1/15/2014

Chapter Overview

- Shift and Rotate Instructions
- Shift and Rotate Applications
 - Binary Multiplication
 - Displaying Binary Bits
 - Isolating a Bit String
 - Shifting Multiple Doublewords
- Multiplication and Division Instructions
- Extended Addition and Subtraction
- ASCII and packed Decimal Arithmetic

Summary

- Shift and rotate instructions are some of the best tools of assembly language
 - finer control than in high-level languages
 - SHL, SHR, SAR, ROL, ROR, RCL, RCR
- MUL and DIV – integer operations
 - close relatives of SHL and SHR
 - CBW, CDQ, CWD: preparation for division
- 32-bit Mode only:
 - Extended precision arithmetic: ADC, SBB
 - ASCII decimal operations (AAA, AAS, AAM, AAD)
 - Packed decimal operations (DAA, DAS)



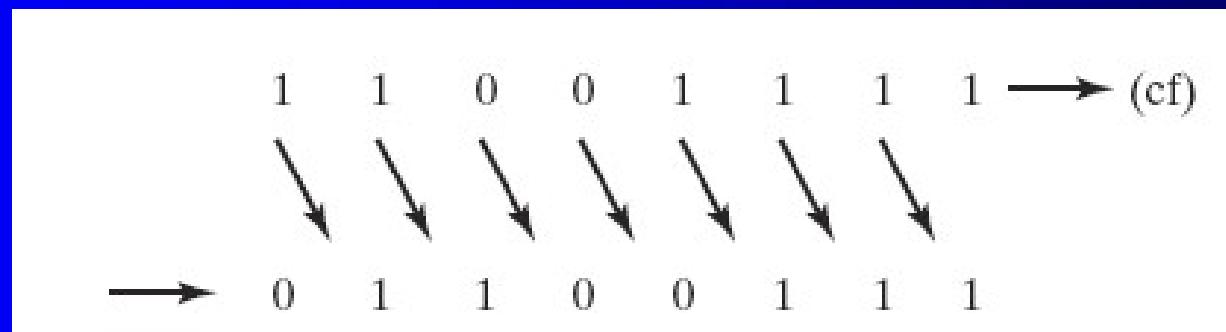
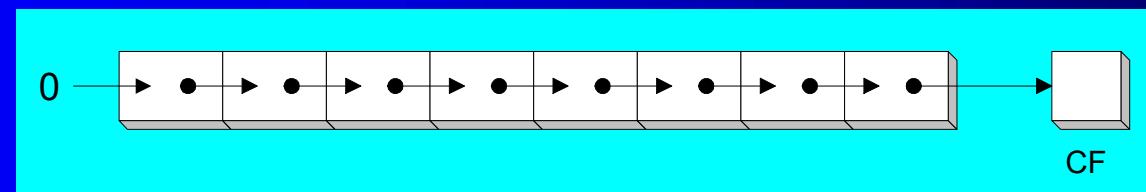
55 74 67 61 6E 67 65 6E

Shift and Rotate Instructions

- Logical vs Arithmetic Shifts
- SHL Instruction
- SHR Instruction
- SAL and SAR Instructions
- ROL Instruction
- ROR Instruction
- RCL and RCR Instructions
- SHLD/SHRD Instructions

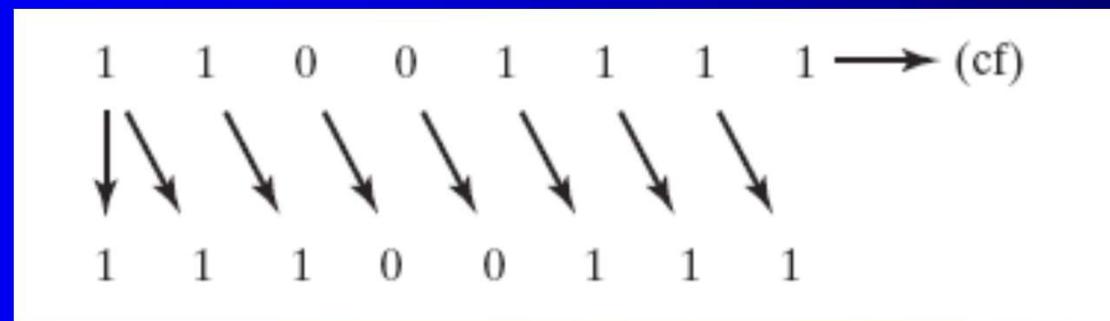
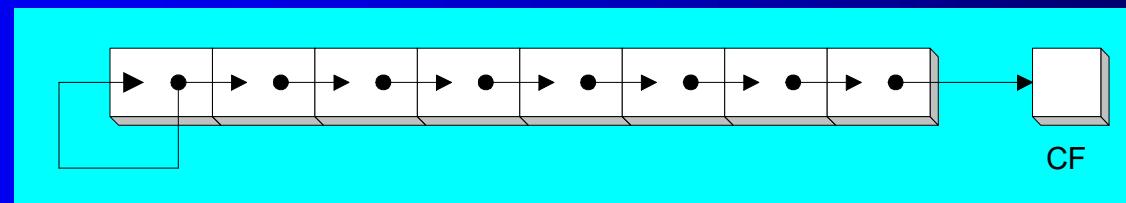
Logical Shift

- A logical shift fills the newly created bit position with zero:



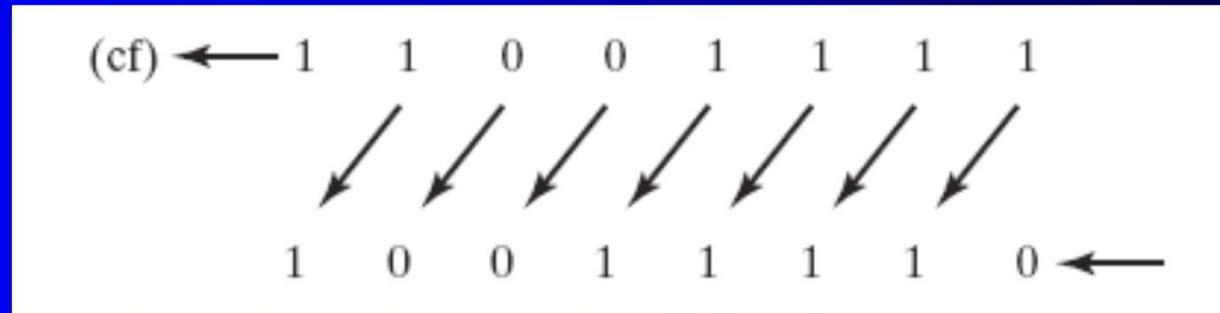
Arithmetic Shift

- An arithmetic shift fills the newly created bit position with a copy of the number's sign bit:



SHL Instruction

- The SHL (shift left) instruction performs a logical left shift on the destination operand, filling the lowest bit with 0.



- Operand types for SHL:

`SHL reg, imm8`

`SHL mem, imm8`

`SHL reg, CL`

`SHL mem, CL`

(Same for all shift and rotate instructions)

Fast Multiplication

Shifting left 1 bit multiplies a number by 2

```
mov dl,5  
shl dl,1
```

Before: 0 0 0 0 0 1 0 1 = 5
After: 0 0 0 0 1 0 1 0 = 10

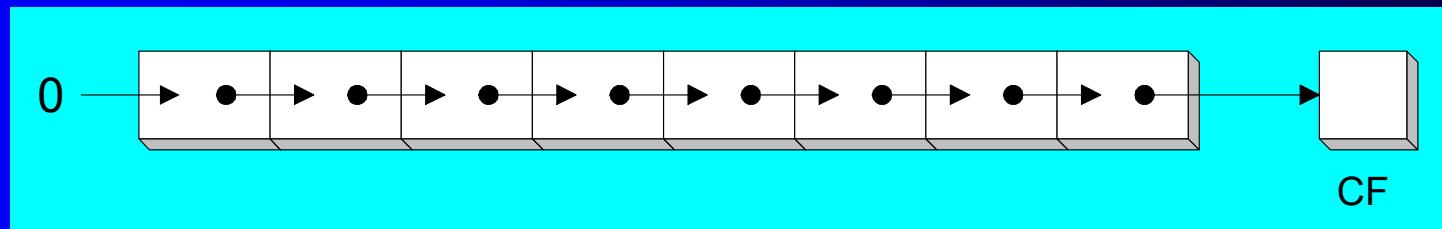
Shifting left n bits multiplies the operand by 2^n

For example, $5 * 2^2 = 20$

```
mov dl,5  
shl dl,2 ; DL = 20
```

SHR Instruction

- The SHR (shift right) instruction performs a logical right shift on the destination operand. The highest bit position is filled with a zero.



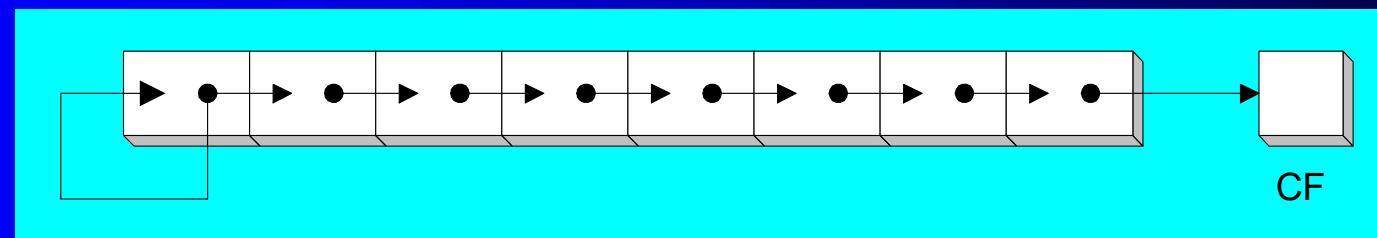
Shifting right n bits divides the operand by 2^n

```
mov dl,80  
shr dl,1      ; DL = 40  
shr dl,2      ; DL = 10
```

DL
0 1 0 1 0 0 0 0
0 0 1 0 1 0 0 0
0 0 0 0 1 0 1 0

SAL and SAR Instructions

- SAL (shift arithmetic left) is identical to SHL.
- SAR (shift arithmetic right) performs a right arithmetic shift on the destination operand.



An arithmetic shift preserves the number's sign.

DL

1	0	1	1	0	0	0	0
1	1	0	1	1	0	0	0
1	0	0	1	0	1	1	0

mov dl,-80
sar dl,1 ; DL = -40
sar dl,2 ; DL = -10

Your turn . . .

Indicate the hexadecimal value of AL after each shift:

mov al, 6Bh

shr al,1

shl al,3

mov al,8Ch

sar al,1

sar al,3

a. **35h**

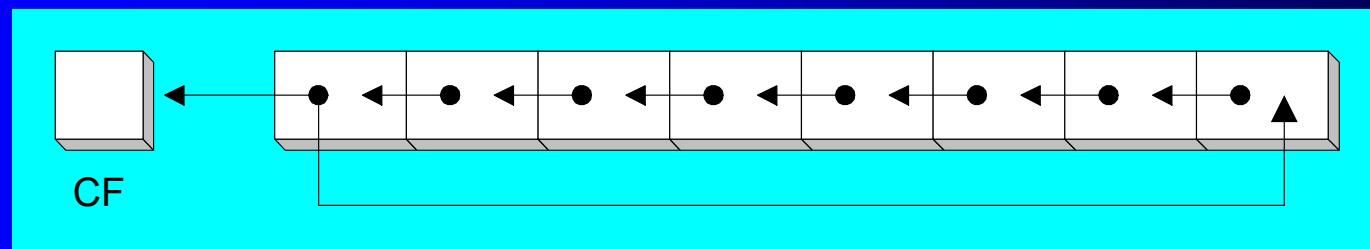
b. **A8h**

c. **C6h**

d. **F8h**

ROL Instruction

- ROL (rotate) shifts each bit to the left
- The highest bit is copied into both the Carry flag and into the lowest bit
- No bits are lost



```
mov al,11110000b  
rol al,1
```

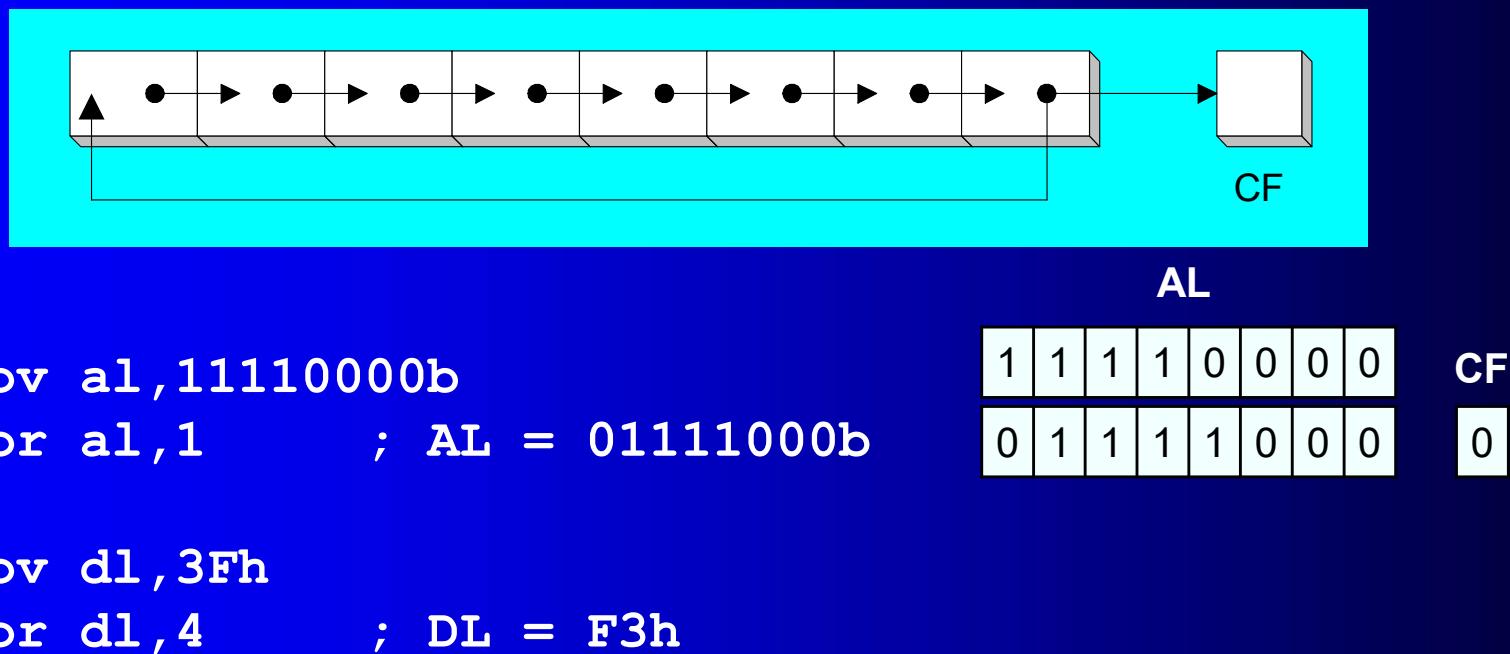
AL

CF	1	1	1	1	0	0	0	0
1	1	1	1	0	0	0	0	1

```
mov dl,3Fh  
rol dl,4 ; DL = F3h
```

ROR Instruction

- ROR (rotate right) shifts each bit to the right
- The lowest bit is copied into both the Carry flag and into the highest bit
- No bits are lost



Your turn . . .

Indicate the hexadecimal value of AL after each rotation:

mov al, 6Bh

ror al,1

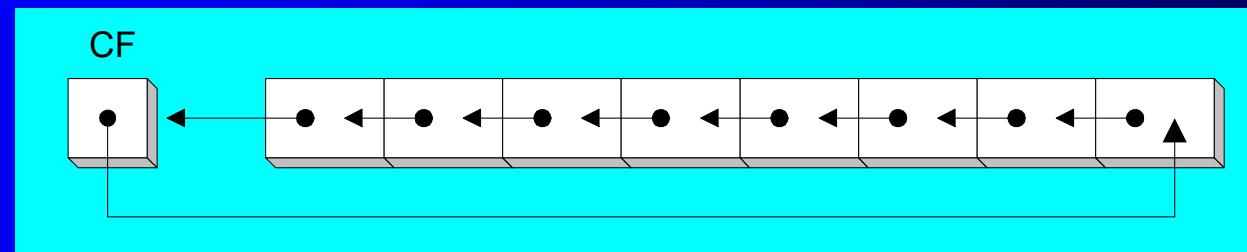
rol al,3

a. B5h

b. ADh

RCL Instruction

- RCL (rotate carry left) shifts each bit to the left
- Copies the Carry flag to the least significant bit
- Copies the most significant bit to the Carry flag

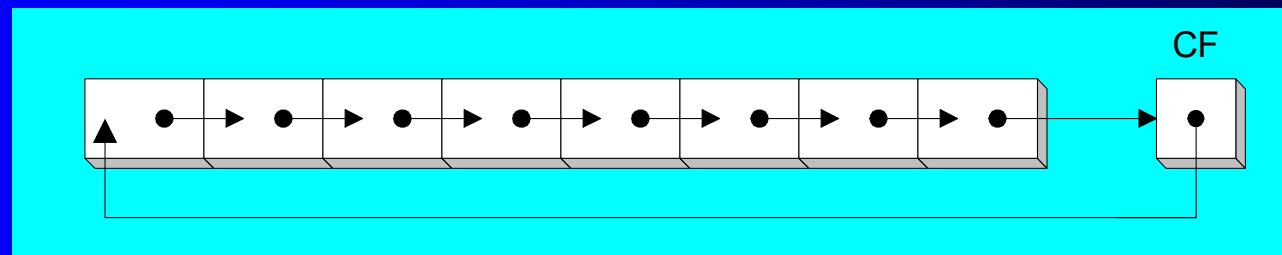


```
clc      ; CF = 0
mov bl,88h
rcl bl,1
rcl bl,1
```

CF	BL
0	1 0 0 0 1 0 0 0
0	0 0 0 1 0 0 0 0
1	0 0 1 0 0 0 0 1
0	0 0 1 0 0 0 0 1

RCR Instruction

- RCR (rotate carry right) shifts each bit to the right
- Copies the Carry flag to the most significant bit
- Copies the least significant bit to the Carry flag



<code>stc ; CF = 1</code>	AH	CF																		
	<table border="1" style="margin-left: auto; margin-right: auto;"><tr><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	1	0	0	0	0	0	1	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	0	0	0												
1	0	0	0	1	0	0	0	0												
<code>mov ah,10h</code>		1																		
<code>rcr ah,1</code>		0																		

Your turn . . .

Indicate the hexadecimal value of AL after each rotation:

stc

mov al, 6Bh

rcr al, 1

rcl al, 3

a. **B5h**

b. **AEh**

SHLD Instruction

- Shifts a destination operand a given number of bits to the left
- The bit positions opened up by the shift are filled by the most significant bits of the source operand
- The source operand is not affected
- Syntax:

SHLD destination, source, count

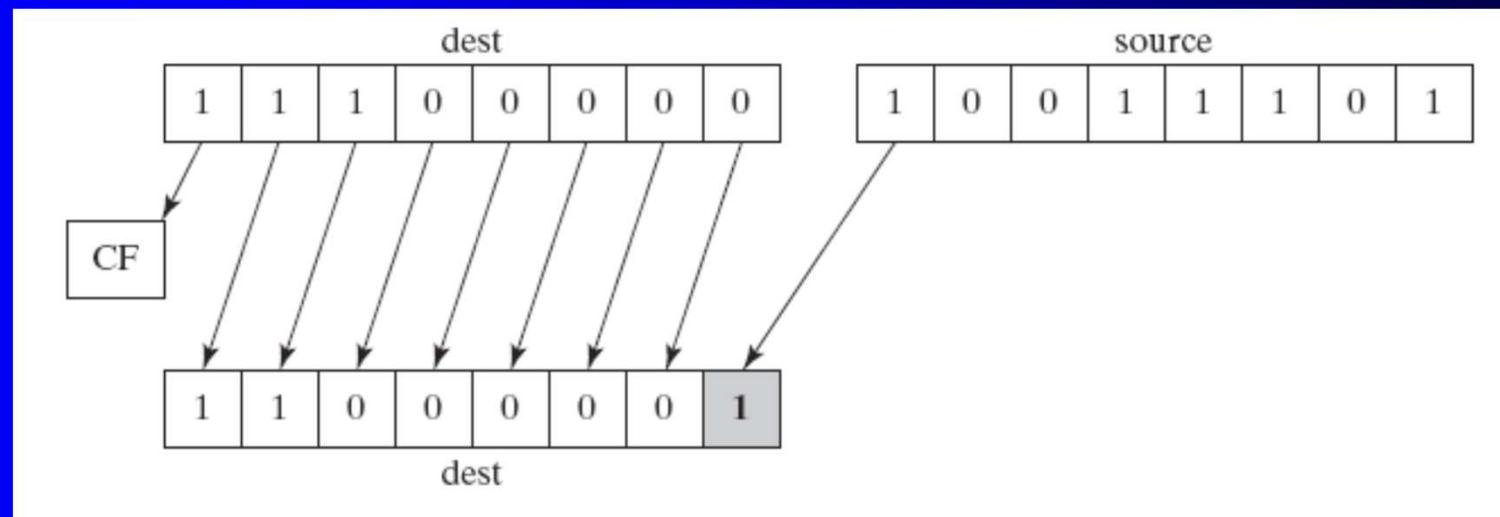
- Operand types:

```
SHLD reg16/32, reg16/32, imm8/CL  
SHLD mem16/32, reg16/32, imm8/CL
```

SHLD Example

Shift count of 1:

```
mov al,11100000b  
mov bl,10011101b  
shld al,bl,1
```

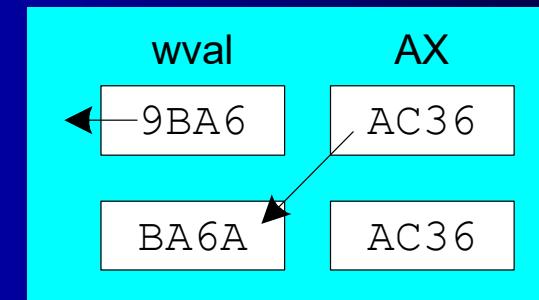


Another SHLD Example

Shift wval 4 bits to the left and replace its lowest 4 bits with the high 4 bits of AX:

```
.data  
wval WORD 9BA6h  
.code  
mov ax,0AC36h  
shld wval,ax,4
```

Before:



After:

SHRD Instruction

- Shifts a destination operand a given number of bits to the right
- The bit positions opened up by the shift are filled by the least significant bits of the source operand
- The source operand is not affected
- Syntax:

SHRD *destination, source, count*

- Operand types:

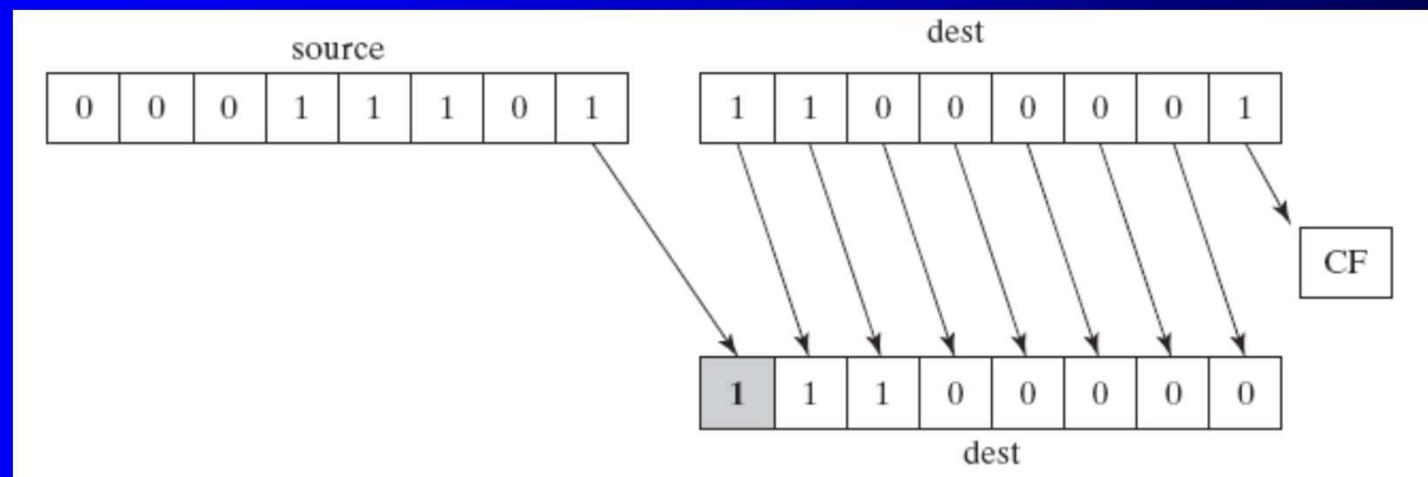
```
SHRD reg16/32, reg16/32, imm8/CL
```

```
SHRD mem16/32, reg16/32, imm8/CL
```

SHRD Example

Shift count of 1:

```
mov al,11000001b  
mov bl,00011101b  
shrd al,bl,1
```



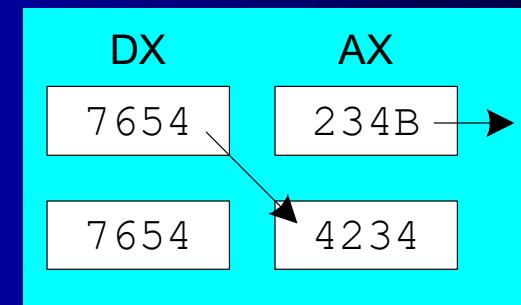
Another SHRD Example

Shift AX 4 bits to the right and replace its highest 4 bits with the low 4 bits of DX:

```
mov ax,234Bh  
mov dx,7654h  
shrd ax,dx,4
```

Before:

After:



Your turn . . .

Indicate the hexadecimal values of each destination operand:

```
mov  ax,7C36h
mov  dx,9FA6h
shld dx,ax,4          ; DX = FA67h
shrd dx,ax,8          ; DX = 36FAh
```

What's Next

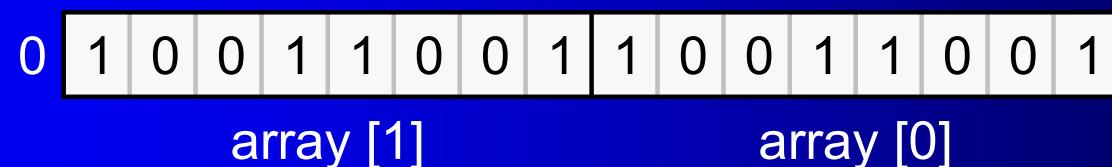
- Shift and Rotate Instructions
- **Shift and Rotate Applications**
- Multiplication and Division Instructions
- Extended Addition and Subtraction
- ASCII and Unpacked Decimal Arithmetic
- Packed Decimal Arithmetic

Shift and Rotate Applications

- Shifting Multiple Doublewords
- Binary Multiplication
- Displaying Binary Bits
- Isolating a Bit String

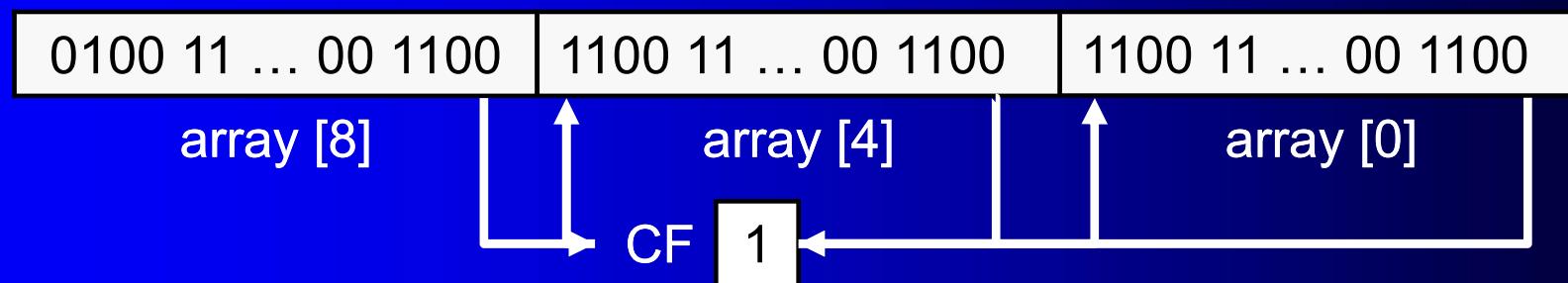
Shifting Multiple Doublewords (1/2)

- Programs sometimes need to shift all bits within an array, as one might when moving a bitmapped graphic image from one screen location to another.



Shifting Multiple Doublewords (2/2)

- The following shifts an array of 3 doublewords 1 bit to the right (view complete [source code](#)):



```
.data
ArraySize = 3
array DWORD ArraySize DUP(99999999h) ; 1001 1001...
.code
mov esi,0
shr array[esi + 8],1 ; high dword
rcr array[esi + 4],1 ; middle dword, include Carry
rcr array[esi],1 ; low dword, include Carry
```

Binary Multiplication

- multiply 123 * 36

$$\begin{array}{rcl} & 01111011 & 123 \\ \times & 00100100 & 36 \\ \hline & 01111011 & 123 \text{ SHL } 2 \\ + & 01111011 & 123 \text{ SHL } 5 \\ \hline & 0001000101001100 & 4428 \end{array}$$

Binary Multiplication

- We already know that SHL performs unsigned multiplication efficiently when the multiplier is a power of 2.
- You can factor any binary number into powers of 2.
 - For example, to multiply EAX * 36, factor 36 into $32 + 4$ and use the distributive property of multiplication to carry out the operation:

$$\begin{aligned} \text{EAX} * 36 \\ = \text{EAX} * (32 + 4) \\ = (\text{EAX} * 32) + (\text{EAX} * 4) \end{aligned}$$

```
mov eax,123
mov ebx,eax
shl eax,5          ; mult by 25
shl ebx,2          ; mult by 22
add eax,ebx
```

Your turn . . .

Multiply AX by 26, using shifting and addition instructions.
Hint: $26 = 16 + 8 + 2$.

```
mov ax,2          ; test value

mov dx,ax
shl dx,4         ; AX * 16
push edx         ; save for later

mov dx,ax
shl dx,3         ; AX * 8
shl ax,1         ; AX * 2
add ax,dx        ; AX * 10
pop edx          ; recall AX * 16
add ax,dx        ; AX * 26
```

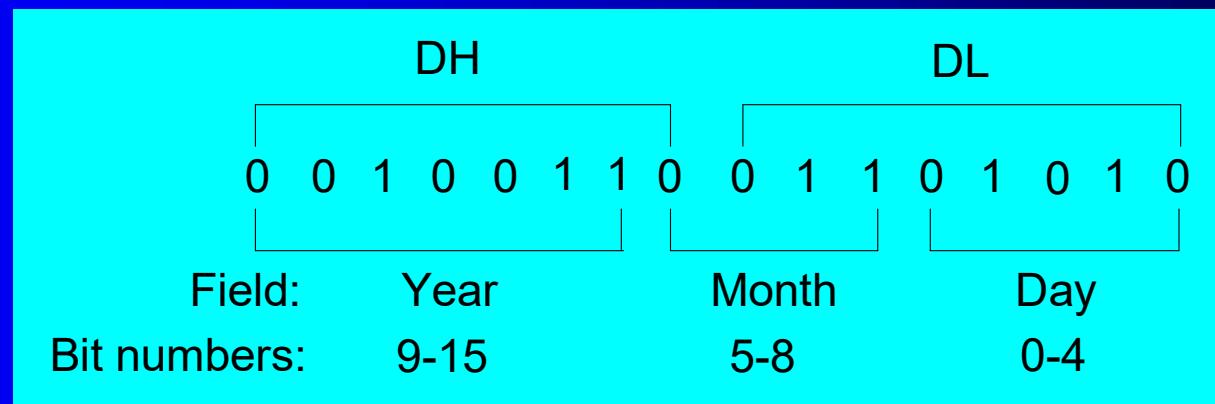
Displaying Binary Bits

Algorithm: Shift MSB into the Carry flag; If CF = 1, append a "1" character to a string; otherwise, append a "0" character. Repeat in a loop, 32 times.

```
.data  
buffer BYTE 32 DUP(0),0  
.code  
    mov ecx,32  
    mov esi,OFFSET buffer  
L1: shl eax,1  
    mov BYTE PTR [esi],'0'  
    jnc L2  
    mov BYTE PTR [esi],'1'  
L2: inc esi  
    loop L1
```

Isolating a Bit String

- The MS-DOS file date field packs the year, month, and day into 16 bits:



Isolate the Month field:

```
mov ax,dx          ; make a copy of DX
shr ax,5          ; shift right 5 bits
and al,00001111b  ; clear bits 4-7
mov month,al      ; save in month variable
```

What's Next

- Shift and Rotate Instructions
- Shift and Rotate Applications
- **Multiplication and Division Instructions**
- Extended Addition and Subtraction
- ASCII and Unpacked Decimal Arithmetic
- Packed Decimal Arithmetic

Multiplication and Division Instructions

- MUL Instruction
- IMUL Instruction
- DIV Instruction
- Signed Integer Division
- CBW, CWD, CDQ Instructions
- IDIV Instruction
- Implementing Arithmetic Expressions

MUL Instruction

- In 32-bit mode, MUL (unsigned multiply) instruction multiplies an 8-, 16-, or 32-bit operand by either AL, AX, or EAX.
- The instruction formats are:

MUL r/m8

MUL r/m16

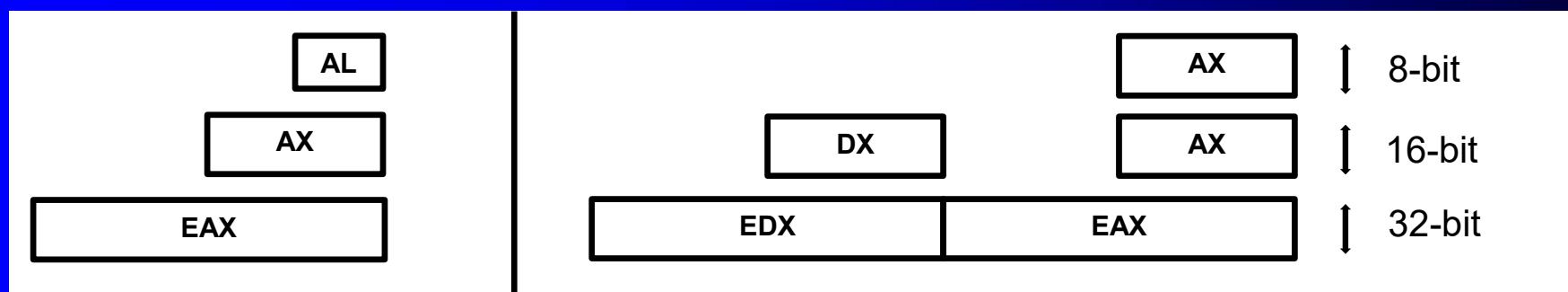
MUL r/m32

Table 7-2 MUL Operands.

Multiplicand	Multiplier	Product
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX

Multiplicand

Product



64-Bit MUL Instruction

- In 64-bit mode, MUL (unsigned multiply) instruction multiplies a 64-bit operand by RAX, producing a 128-bit product.
- The instruction formats are:

MUL r/m64

Example:

```
mov rax,0FFFF0000FFFF0000h  
mov rbx,2  
mul rbx      ; RDX:RAX = 0000000000000001FFFE0001FFFE0000
```

MUL Examples

100h * 2000h, using 16-bit operands:

```
.data  
val1 WORD 2000h  
val2 WORD 100h  
.code  
mov ax, val1  
mul val2      ; DX:AX = 00200000h, CF=1
```

The Carry flag indicates whether or not the upper half of the product contains significant digits.

12345h * 1000h, using 32-bit operands:

```
mov eax, 12345h  
mov ebx, 1000h  
mul ebx      ; EDX:EAX = 0000000012345000h, CF=0
```

Binary Multiplications with SHL

Your turn . . .

What will be the hexadecimal values of DX, AX, and the Carry flag after the following instructions execute?

```
mov ax,1234h  
mov bx,100h  
mul bx
```

DX = 0012h, AX = 3400h, CF = 1

Your turn . . .

What will be the hexadecimal values of EDX, EAX, and the Carry flag after the following instructions execute?

```
mov eax,00128765h  
mov ecx,10000h  
mul ecx
```

EDX = 00000012h, EAX = 87650000h, CF = 1

IMUL Instruction

- IMUL (signed integer multiply) multiplies an 8-, 16-, or 32-bit signed operand by either AL, AX, or EAX
- Preserves the sign of the product by sign-extending it into the upper half of the destination register

Example: multiply 48 * 4, using 8-bit operands:

```
mov al,48      ; AL = 30h
mov bl,4       ; BL = 04h
imul bl        ; AX = 00C0h, OF=1
```

00110000b	AL
00000100b	BL

00000000	AX
----------	----

OF=1 because AH is not a sign extension of AL.

Using IMUL in 64-Bit Mode

- You can use 64-bit operands. In the two-operand format, a 64-bit register or memory operand is multiplied against RDX
 - 128-bit product produced in RDX:RAX
- The three-operand format produces a 64-bit product in RAX

```
.data  
multiplicand QWORD -16  
.code  
imul rax, multiplicand, 4 ; RAX = FFFFFFFFFFFFFFC0 (-64)
```

IMUL Examples

Multiply 4,823,424 * -423:

```
mov eax,4823424  
mov ebx,-423  
imul ebx          ; EDX:EAX = FFFFFFFF86635D80h, OF=0
```

OF=0 because EDX is a sign extension of EAX.

Your turn . . .

What will be the hexadecimal values of DX, AX, and the Carry flag after the following instructions execute?

```
mov ax,8760h  
mov bx,100h  
imul bx
```

DX = FF87h, AX = 6000h, OF = 1

DIV Instruction

- The DIV (unsigned divide) instruction performs 8-bit, 16-bit, and 32-bit division on unsigned integers
- A single operand is supplied (register or memory operand), which is assumed to be the divisor
- Instruction formats:

```
mov ax, Dividend  
div r/m8
```

```
mov dx, Dividend_Hign  
mov ax, Dividend_Low  
div r/m16
```

```
mov edx, Dividend_Hign  
mov eax, Dividend_Low  
div r/m32
```

Default Operands:

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

DIV Examples

Divide 8003h by 100h, using 16-bit operands:

```
mov dx,0          ; clear dividend, high  
mov ax,8003h     ; dividend, low  
mov cx,100h       ; divisor  
div cx           ; AX = 0080h, DX = 3
```

Same division, using 32-bit operands:

```
mov edx,0         ; clear dividend, high  
mov eax,8003h     ; dividend, low  
mov ecx,100h       ; divisor  
div ecx          ; EAX = 00000080h, DX = 3
```

64-Bit DIV Example

Divide 000001080000000033300020h by
00010000h:

```
.data

```

Default Operands:

<i>Dividend</i>	<i>Divisor</i>	<i>Quotient</i>	<i>Remainder</i>
RDX:RAX	r/m64	RAX	RDX

quotient: 0108000000003330h
remainder: 0000000000000020h

Your turn . . .

What will be the hexadecimal values of DX and AX after the following instructions execute? Or, if divide overflow occurs, you can indicate that as your answer:

```
mov dx,0087h  
mov ax,6000h  
mov bx,100h  
div bx
```

DX = 0000h, AX = 8760h

Your turn . . .

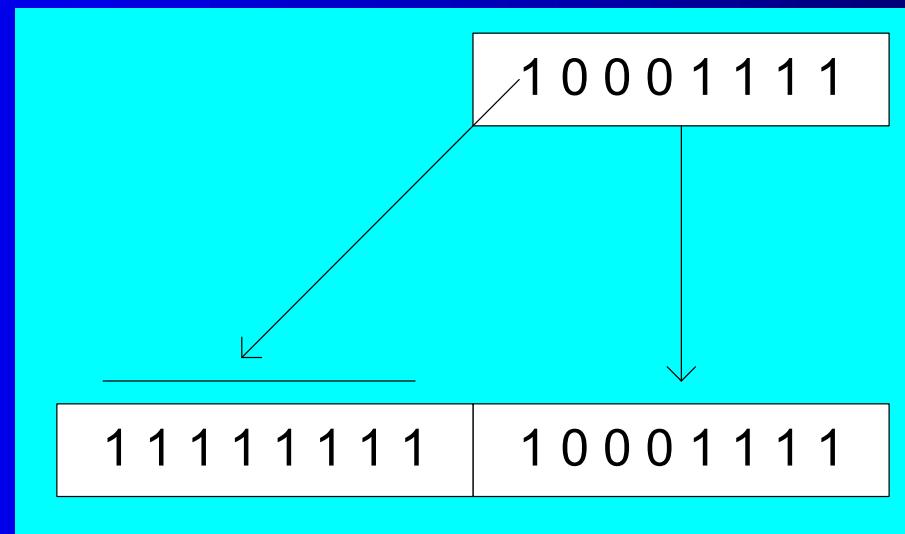
What will be the hexadecimal values of DX and AX after the following instructions execute? Or, if divide overflow occurs, you can indicate that as your answer:

```
mov dx,0087h  
mov ax,6002h  
mov bx,10h  
div bx
```

Divide Overflow

Signed Integer Division (IDIV)

- Signed integers must be sign-extended before division takes place
 - fill high byte/word/doubleword with a copy of the low byte/word/doubleword's sign bit
- For example, the high byte contains a copy of the sign bit from the low byte:



CBW, CWD, CDQ Instructions

- The CBW, CWD, and CDQ instructions provide important sign-extension operations:
 - CBW (convert byte to word) extends AL into AH
 - CWD (convert word to doubleword) extends AX into DX
 - CDQ (convert doubleword to quadword) extends EAX into EDX
- Example:

```
.data  
dwordVal SDWORD -101      ; FFFFFFF9Bh  
.code  
mov eax,dwordVal  
cdq           ; EDX:EAX = FFFFFFFFFFFFFF9Bh
```

IDIV Instruction

- IDIV (signed divide) performs signed integer division
- Same syntax and operands as DIV instruction

Default Operands:

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

Example: 8-bit division of -48 by 5

```
mov al, -48
cbw          ; extend AL into AH
mov bl, 5
idiv bl      ; AL = -9, AH = -3
```

IDIV Examples

Example: 16-bit division of -48 by 5

```
mov  ax,-48
 cwd             ; extend AX into DX
 mov  bx,5
 idiv bx        ; AX = -9,   DX = -3
```

Example: 32-bit division of -48 by 5

```
mov  eax,-48
 cdq            ; extend EAX into EDX
 mov  ebx,5
 idiv ebx       ; EAX = -9,   EDX = -3
```

Your turn . . .

What will be the hexadecimal values of DX and AX after the following instructions execute? Or, if divide overflow occurs, you can indicate that as your answer:

```
mov  ax,0FDFFh          ; -513  
 cwd  
 mov  bx,100h  
 idiv bx
```

DX = FFFFh (-1), AX = FFFEh (-2)

Unsigned Arithmetic Expressions

- Some good reasons to learn how to implement integer expressions:
 - Learn how do compilers do it
 - Test your understanding of MUL, IMUL, DIV, IDIV
 - Check for overflow (Carry and Overflow flags)

Example: `var4 = (var1 + var2) * var3`

```
; Assume unsigned operands
mov  eax,var1
add  eax,var2          ; EAX = var1 + var2
mul  var3              ; EAX = EAX * var3
jc   TooBig            ; check for carry
mov  var4,eax          ; save product
```

Signed Arithmetic Expressions

Example: **eax = (-var1 * var2) + var3**

```
mov  eax,var1
neg  eax
imul var2
jo   TooBig           ; check for overflow
add  eax,var3
jo   TooBig           ; check for overflow
```

Example: **var4 = (var1 * 5) / (var2 - 3)**

```
mov  eax,var1          ; left side
mov  ebx,5
imul ebx               ; EDX:EAX = product
mov  ebx,var2          ; right side
sub  ebx,3
idiv ebx               ; EAX = quotient
mov  var4,eax
```

Signed Arithmetic Expressions (2 of 2)

Example: `var4 = (var1 * -5) / (-var2 % var3);`

```
mov  eax,var2          ; begin right side
neg  eax
cdq
idiv var3              ; sign-extend dividend
                       ; EDX = remainder
mov  ebx,edx            ; EBX = right side
mov  eax,-5             ; begin left side
imul var1               ; EDX:EAX = left side
idiv ebx                ; final division
mov  var4,eax            ; quotient
```

Sometimes it's easiest to calculate the right-hand term of an expression first.

Your turn . . .

Implement the following expression using signed 32-bit integers:

eax = (ebx * 20) / ecx

```
mov eax,20  
imul ebx  
idiv ecx
```

Your turn . . .

Implement the following expression using signed 32-bit integers. Save and restore ECX and EDX:

eax = (ecx * edx) / eax

```
push  edx
push  eax          ; EAX needed later
mov   eax,ecx
imul  edx          ; left side: EDX:EAX
pop   ebx          ; saved value of EAX
idiv  ebx          ; EAX = quotient
pop   edx          ; restore EDX, ECX
```

Your turn . . .

Implement the following expression using signed 32-bit integers. Do not modify any variables other than var3:

```
var3 = (var1 * -var2) / (var3 - ebx)
```

```
mov  eax,var1
mov  edx,var2
neg  edx
imul edx          ; left side: EDX:EAX
mov  ecx,var3
sub  ecx,ebx
idiv ecx          ; EAX = quotient
mov  var3,eax
```

What's Next

- Shift and Rotate Instructions
- Shift and Rotate Applications
- Multiplication and Division Instructions
- **Extended Addition and Subtraction**
- ASCII and UnPacked Decimal Arithmetic
- Packed Decimal Arithmetic

Extended Addition and Subtraction

- ADC Instruction
- Extended Precision Addition
- SBB Instruction
- Extended Precision Subtraction

The instructions in this section do not apply to 64-bit mode programming.

Extended Precision Addition

- Adding two operands that are longer than the computer's word size (32 bits).
 - Virtually no limit to the size of the operands
- The arithmetic must be performed in steps
 - The Carry value from each step is passed on to the next step.

ADC Instruction

- ADC (add with carry) instruction adds both a source operand and the contents of the Carry flag to a destination operand.
- Operands are binary values
 - Same syntax as ADD, SUB, etc.
- Example
 - Add two 32-bit integers (FFFFFFFh + FFFFFFFh), producing a 64-bit sum in EDX:EAX:

```
mov edx,0  
mov eax,0FFFFFFFh  
add eax,0FFFFFFFh  
adc edx,0          ;EDX:EAX = 00000001FFFFFFh
```

Extended Addition Example

- Task: Add 1 to EDX:EAX
 - Starting value of EDX:EAX: 00000000FFFFFFFFFFh
 - Add the lower 32 bits first, setting the Carry flag.
 - Add the upper 32 bits, and include the Carry flag.

```
mov edx,0          ; set upper half
mov eax,0FFFFFFFFFFh ; set lower half
add eax,1          ; add lower half
adc edx,0          ; add upper half
```

EDX:EAX = 00000001 00000000

SBB Instruction

- The SBB (subtract with borrow) instruction subtracts both a source operand and the value of the Carry flag from a destination operand.
- Operand syntax:
 - Same as for the ADC instruction

Extended Subtraction Example

- Task: Subtract 1 from EDX:EAX
 - Starting value of EDX:EAX: 0000000100000000h
 - Subtract the lower 32 bits first, setting the Carry flag.
 - Subtract the upper 32 bits, and include the Carry flag.

```
mov edx,1          ; set upper half
mov eax,0          ; set lower half
sub eax,1          ; subtract lower half
sbb edx,0          ; subtract upper half
```

EDX:EAX = 00000000 FFFFFFFF

What's Next

- Shift and Rotate Instructions
- Shift and Rotate Applications
- Multiplication and Division Instructions
- Extended Addition and Subtraction
- **ASCII and UnPacked Decimal Arithmetic**
- Packed Decimal Arithmetic

ASCII and Packed Decimal Arithmetic

- Binary Coded Decimal
- ASCII Decimal
- AAA Instruction
- AAS Instruction
- AAM Instruction
- AAD Instruction
- Packed Decimal Integers
- DAA Instruction
- DAS Instruction

The instructions in this section do not apply to 64-bit mode programming.

Binary-Coded Decimal

- Binary-coded decimal (BCD) integers use 4 binary bits to represent each decimal digit
- A number using **unpacked BCD** representation stores a decimal digit in the lower four bits of each byte
 - For example, 5,678 is stored as the following sequence of hexadecimal bytes:

05	06	07	08
----	----	----	----

ASCII Decimal

- A number using ASCII Decimal representation stores a single ASCII digit in each byte
 - For example, 5,678 is stored as the following sequence of hexadecimal bytes:

35	36	37	38
----	----	----	----

AAA Instruction

- The AAA (ASCII adjust after addition) instruction adjusts the binary result of an ADD or ADC instruction. It makes the result in AL consistent with ASCII decimal representation.
 - The Carry value, if any ends up in AH
- Example: Add '8' and '2'

```
mov ah,0
mov al,'8'          ; AX = 0038h
add al,'2'          ; AX = 006Ah
aaa                ; AX = 0100h (adjust result)
or   ax,3030h       ; AX = 3130h = '10'
```

AAS Instruction

- The AAS (ASCII adjust after subtraction) instruction adjusts the binary result of an SUB or SBB instruction. It makes the result in AL consistent with ASCII decimal representation.
 - It places the Carry value, if any, in AH
- Example: Subtract '9' from '8'

```
mov ah,0
mov al,'8'          ; AX = 0038h
sub al,'9'          ; AX = 00FFh
aas                 ; AX = FF09h, CF=1
or al,30h           ; AL = '9'
```

AAM Instruction

- The AAM (ASCII adjust after multiplication) instruction adjusts the binary result of a MUL instruction. The multiplication must have been performed on unpacked BCD numbers.

```
mov bl,05h          ; first operand
mov al,06h          ; second operand
mul bl             ; AX = 001Eh
aam                ; AX = 0300h
```

AAD Instruction

- The AAD (ASCII adjust before division) instruction adjusts the unpacked BCD dividend in AX before a division operation

```
.data
quotient  BYTE ?
remainder  BYTE ?

.code
mov ax,0307h          ; dividend
aad                  ; AX = 0025h
mov bl,5              ; divisor
div bl                ; AX = 0207h
mov quotient,al
mov remainder,ah
```

What's Next

- Shift and Rotate Instructions
- Shift and Rotate Applications
- Multiplication and Division Instructions
- Extended Addition and Subtraction
- ASCII and UnPacked Decimal Arithmetic
- **Packed Decimal Arithmetic**

Packed Decimal Arithmetic

- Packed decimal integers store two decimal digits per byte
 - For example, 12,345,678 can be stored as the following sequence of hexadecimal bytes:

12	34	56	78
----	----	----	----

Packed decimal is also known as packed BCD.

Good for financial values – extended precision possible, without rounding errors.

DAA Instruction

- The DAA (decimal adjust after addition) instruction converts the binary result of an ADD or ADC operation to packed decimal format.
 - The value to be adjusted must be in AL
 - If the lower digit is adjusted, the Auxiliary Carry flag is set.
 - If the upper digit is adjusted, the Carry flag is set.

DAA Logic

```
If (AL(lo) > 9) or (AuxCarry = 1)
    AL = AL + 6
    AuxCarry = 1
Else
    AuxCarry = 0
Endif

If (AL(hi) > 9) or Carry = 1
    AL = AL + 60h
    Carry = 1
Else
    Carry = 0
Endif
```

If AL = AL + 6 sets the Carry flag, its value is used when evaluating AL(hi).

DAA Examples

- Example: calculate BCD 35 + 48

```
mov al,35h  
add al,48h          ; AL = 7Dh  
daa                ; AL = 83h, CF = 0
```

- Example: calculate BCD 35 + 65

```
mov al,35h  
add al,65h          ; AL = 9Ah  
daa                ; AL = 00h, CF = 1
```

- Example: calculate BCD 69 + 29

```
mov al,69h  
add al,29h          ; AL = 92h  
daa                ; AL = 98h, CF = 0
```

Your turn . . .

- A temporary malfunction in your computer's processor has disabled the DAA instruction. Write a procedure in assembly language that performs the same actions as DAA.
- Test your procedure using the values from the previous slide.

DAS Instruction

- The DAS (decimal adjust after subtraction) instruction converts the binary result of a SUB or SBB operation to packed decimal format.
- The value must be in AL
- Example: subtract BCD 48 from 85

```
mov al,48h  
sub al,35h          ; AL = 13h  
das                ; AL = 13h CF = 0
```

DAS Logic

```
If  (AL(10) > 9) OR (AuxCarry = 1)
    AL = AL - 6;
    AuxCarry = 1;
Else
    AuxCarry = 0;
Endif

If  (AL > 9FH) or (Carry = 1)
    AL = AL - 60h;
    Carry = 1;
Else
    Carry = 0;
Endif
```

If $AL = AL - 6$ sets the Carry flag, its value is used when evaluating AL in the second IF statement.

DAS Examples (1 of 2)

- Example: subtract BCD $48 - 35$

```
mov al,48h  
sub al,35h          ; AL = 13h  
das                ; AL = 13h CF = 0
```

- Example: subtract BCD $62 - 35$

```
mov al,62h  
sub al,35h          ; AL = 2Dh, CF = 0  
das                ; AL = 27h, CF = 0
```

- Example: subtract BCD $32 - 29$

```
mov al,32h  
add al,29h          ; AL = 09h, CF = 0  
daa                ; AL = 03h, CF = 0
```

DAS Examples (2 of 2)

- Example: subtract BCD 32 – 39

```
mov al,32h  
sub al,39h          ; AL = F9h, CF = 1  
das                ; AL = 93h, CF = 1
```

Steps:

AL = F9h

CF = 1, so subtract 6 from F9h

AL = F3h

F3h > 9Fh, so subtract 60h from F3h

AL = 93h, CF = 1

Your turn . . .

- A temporary malfunction in your computer's processor has disabled the DAS instruction. Write a procedure in assembly language that performs the same actions as DAS.
- Test your procedure using the values from the previous two slides.