

Assembly Language for x86 Processors

7th Edition

Kip Irvine

Chapter 1: Basic Concepts

Slides prepared by the author

Revision date: 1/15/2014

Chapter Overview

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

Welcome to Assembly Language

- Some Good 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:

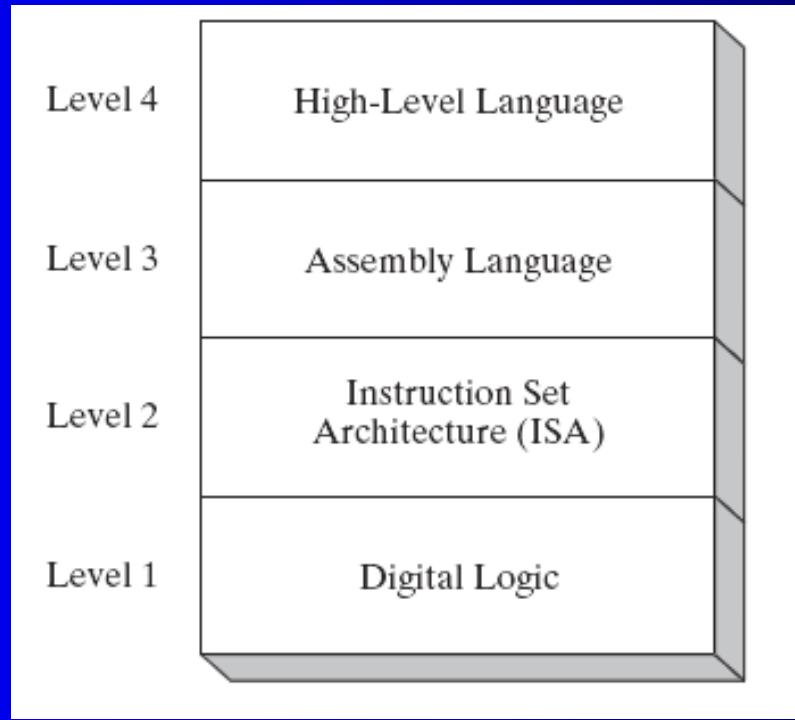
```
mov eax,A  
mul B  
add eax,C  
call WriteInt
```



Intel Machine Language:

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

Specific Machine Levels



(descriptions of individual levels follow . . .)

High-Level Language

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

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)

Instruction Set Architecture (ISA)

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

Digital Logic

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

next: Data Representation

What's Next

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

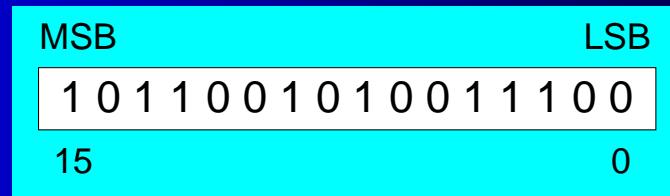
Data Representation

- Binary Numbers
 - Translating between binary and decimal
- Binary Addition
- Integer Storage Sizes
- Hexadecimal Integers
 - Translating between decimal and hexadecimal
 - Hexadecimal subtraction
- Signed Integers
 - Binary subtraction
- Character Storage

Binary Numbers

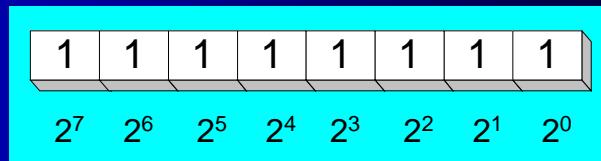
- Digits are 1 and 0
 - 1 = true
 - 0 = false
- MSB – most significant bit
- LSB – least significant bit

- 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

binary 00001001 = decimal 9:

$$(1 \times 2^3) + (1 \times 2^0) = 9$$

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

$$37 = 100101$$

Binary Addition

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

										carry: 1
										(4)
										(7)
<hr/>										
										(11)
bit position:										
7 6 5 4 3 2 1 0										

Integer Storage Sizes

Standard sizes:

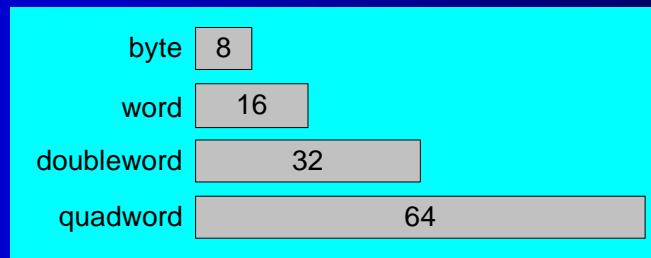


Table 1-4 Ranges of Unsigned Integers.

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?

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
<hr/>	<hr/>	<hr/>	<hr/>
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:

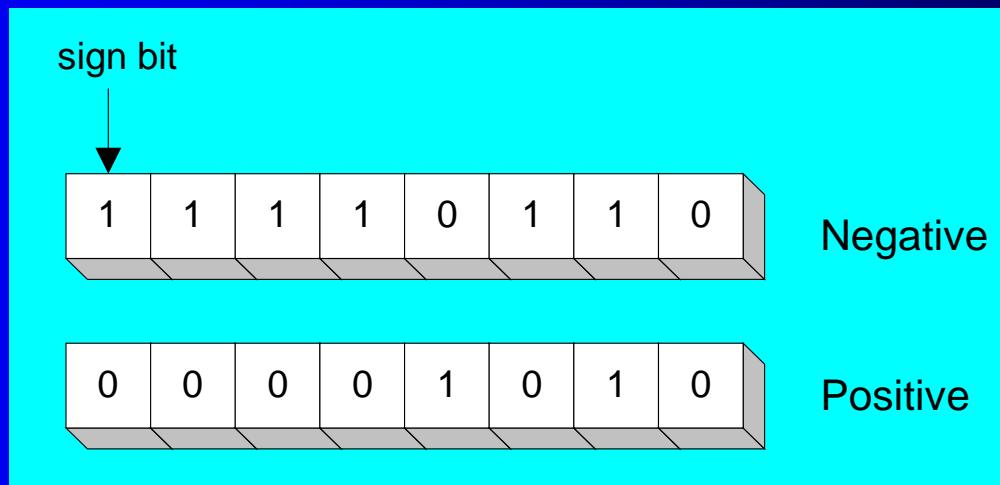
$$\begin{array}{r} 16 + 5 = 21 \\ \hline C6 & 75 \\ A2 & 47 \\ \hline 24 & 2E \end{array}$$

-1 ↓

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

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

Starting value	00000001
Step 1: reverse the bits	11111110
Step 2: add 1 to the value from Step 1	11111110 +00000001
Sum: two's complement representation	11111111

Note that $00000001 + 11111111 = 00000000$

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?

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

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

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

Boolean Algebra

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

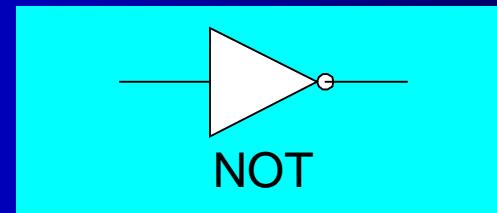
Expression	Description
$\neg X$	NOT X
$X \wedge Y$	X AND Y
$X \vee Y$	X OR Y
$\neg X \vee Y$	(NOT X) OR Y
$\neg(X \wedge Y)$	NOT (X AND Y)
$X \wedge \neg Y$	X AND (NOT Y)

NOT

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

X	$\neg X$
F	T
T	F

Digital gate diagram for NOT:

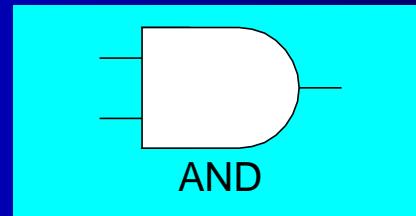


AND

- 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:

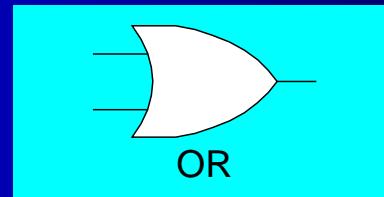


OR

- 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)

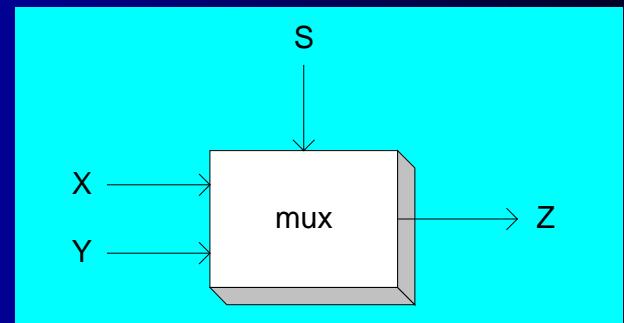
- Example: $X \wedge \neg Y$

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

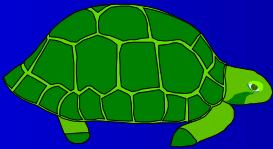
X	Y	S	$Y \wedge S$	$\neg S$	$X \wedge \neg S$	$(Y \wedge S) \vee (X \wedge \neg S)$
F	F	F	F	T	F	F
F	T	F	F	T	F	F
T	F	F	F	T	T	T
T	T	F	F	T	T	T
F	F	T	F	F	F	F
F	T	T	T	F	F	T
T	F	T	F	F	F	F
T	T	T	T	F	F	T



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

Kip Irvine

Chapter 2: x86 Processor Architecture

Slides prepared by the author

Revision date: 1/15/2014

Chapter Overview

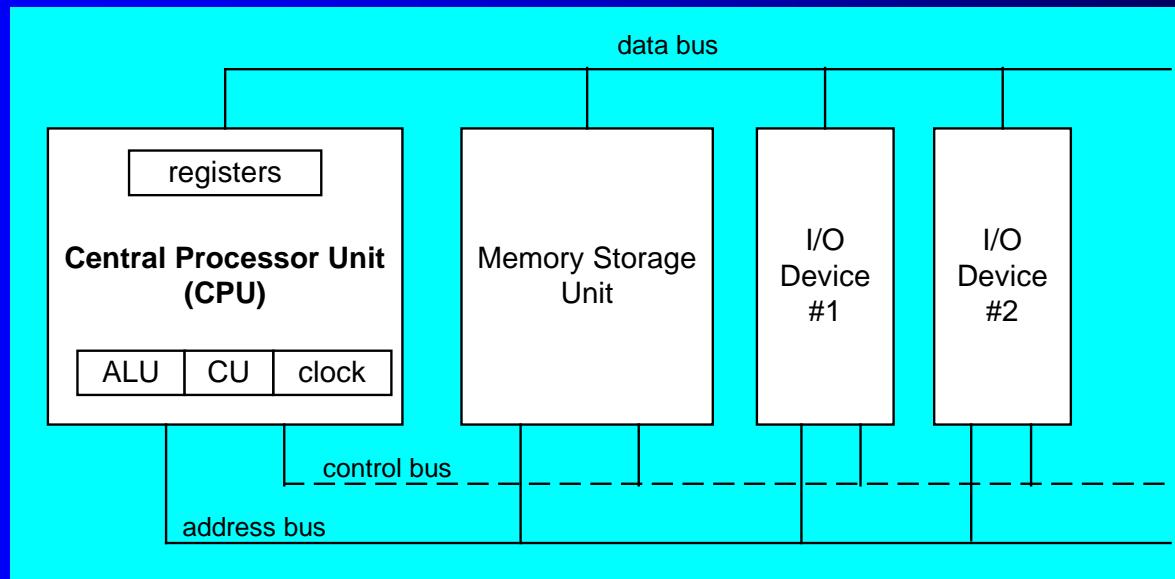
- General Concepts
- IA-32 Processor Architecture
- IA-32 Memory Management
- 64-bit Processors
- Components of an IA-32 Microcomputer
- Input-Output System

General Concepts

- Basic microcomputer design
- Instruction execution cycle
- Reading from memory
- How programs run

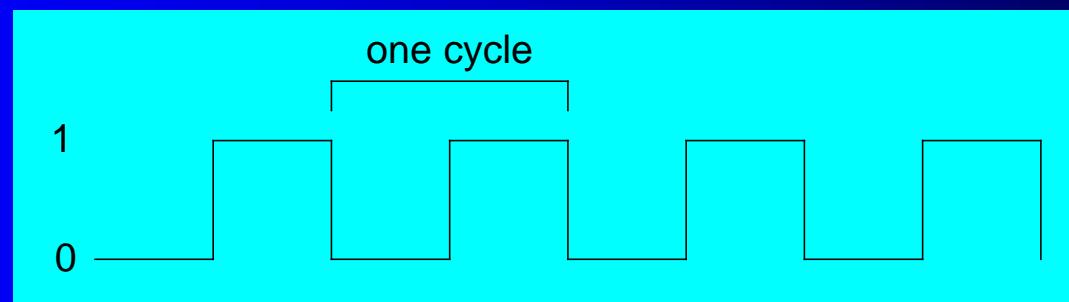
Basic Microcomputer Design

- 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



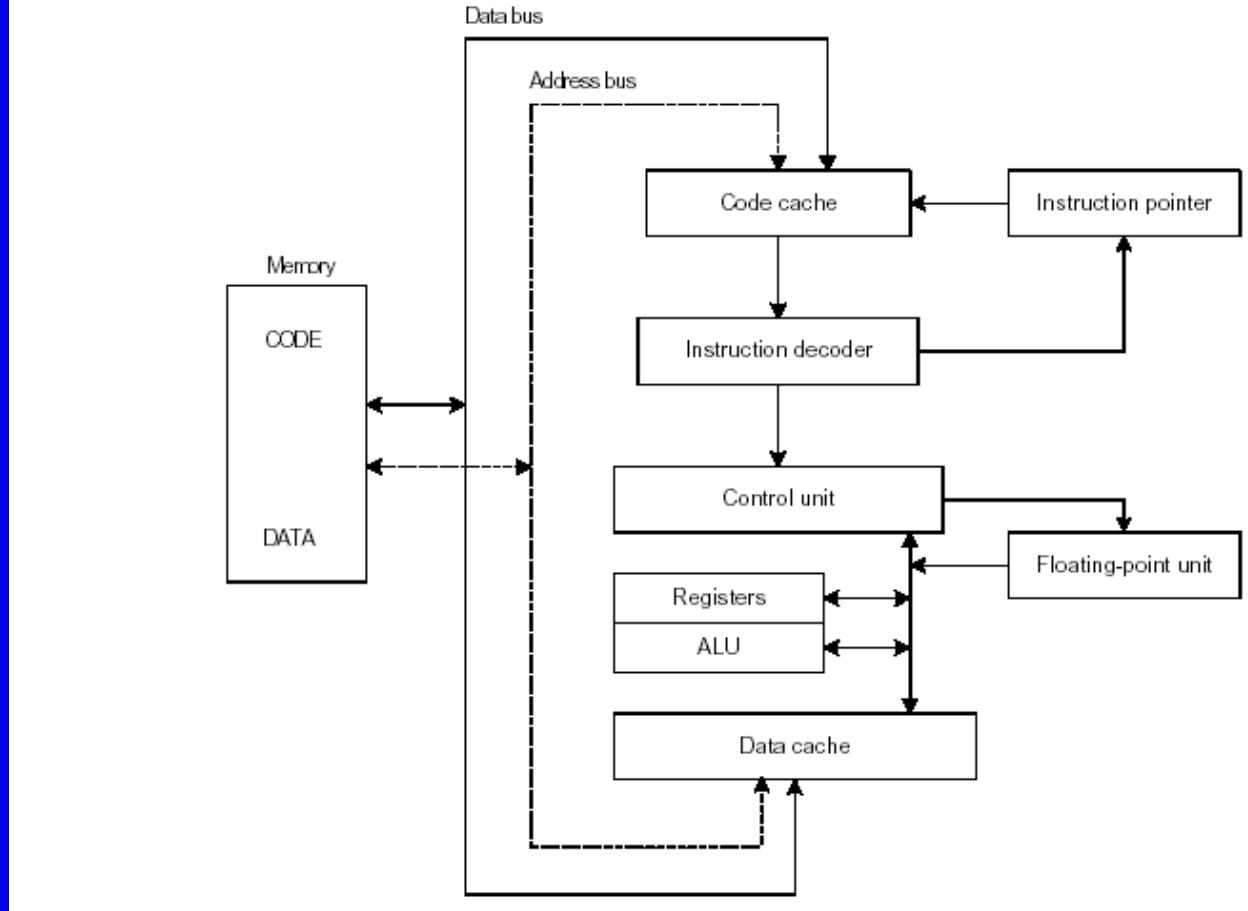
What's Next

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

Instruction Execution Cycle

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

Figure 2–2 Simplified Pentium CPU Block Diagram.



Reading from Memory

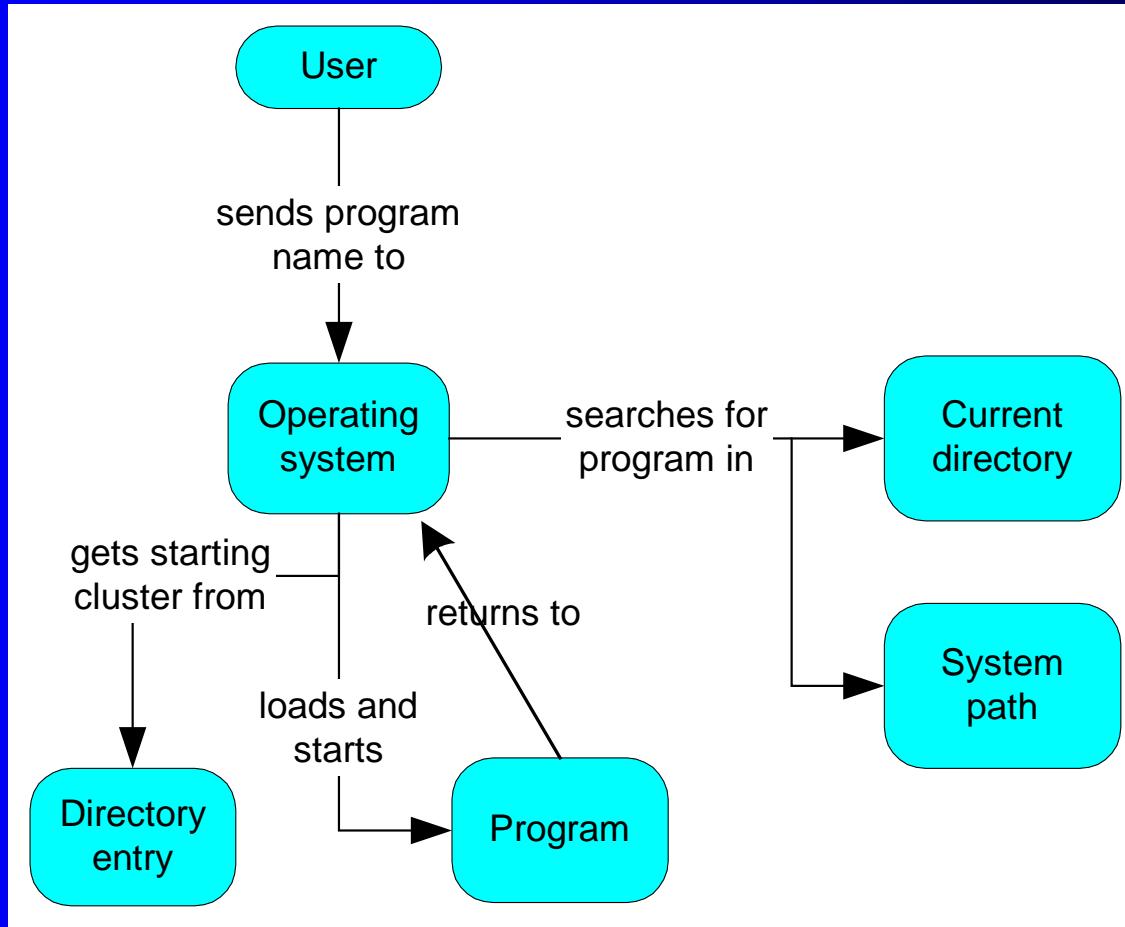
Multiple machine cycles are required when reading from memory, because it responds much more slowly than the CPU. The steps are:

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



IA-32 Processor Architecture

- Modes of operation
- Basic execution environment
- Floating-point unit
- Intel Microprocessor history

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

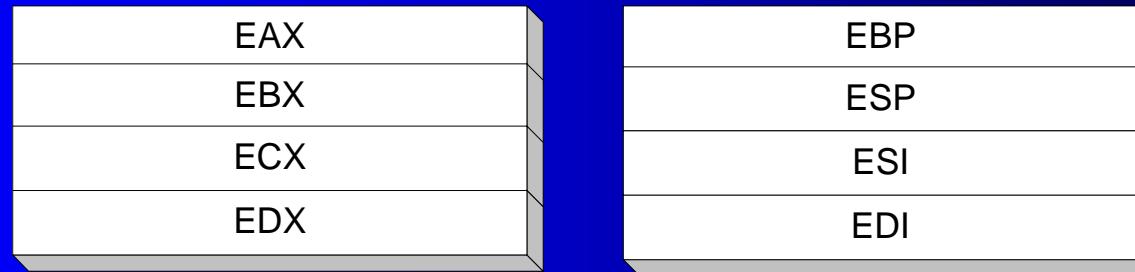
Addressable Memory

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

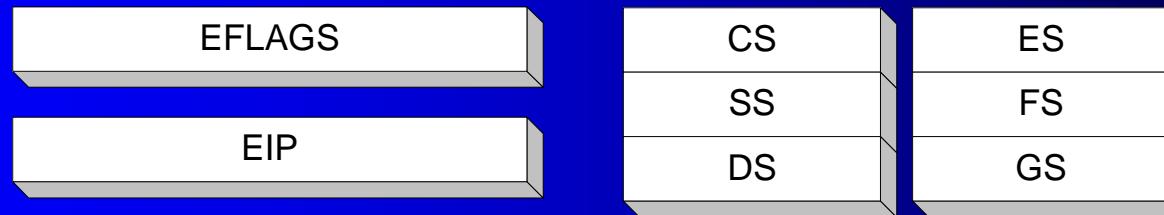
General-Purpose Registers

Named storage locations inside the CPU, optimized for speed.

32-bit General-Purpose Registers

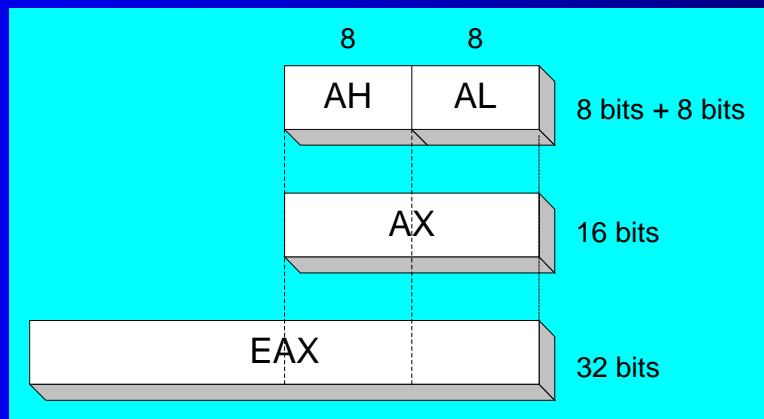


16-bit Segment Registers



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)

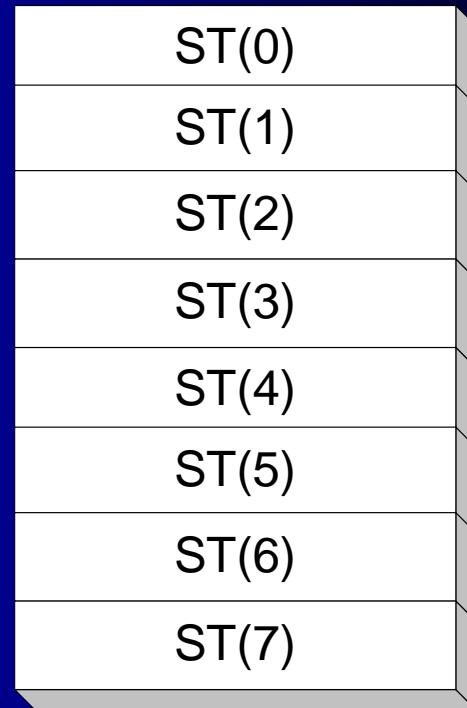
- EIP – instruction pointer
- EFLAGS
 - status and control flags
 - each flag is a single binary bit

Status Flags

- Carry
 - unsigned arithmetic out of range
- Overflow
 - signed arithmetic out of range
- Sign
 - result is negative
- Zero
 - result is zero
- Auxiliary Carry
 - carry from bit 3 to bit 4
- Parity
 - 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
- Calculating linear addresses
- Protected mode
- Multi-segment model
- Paging

Protected Mode (1 of 2)

- 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

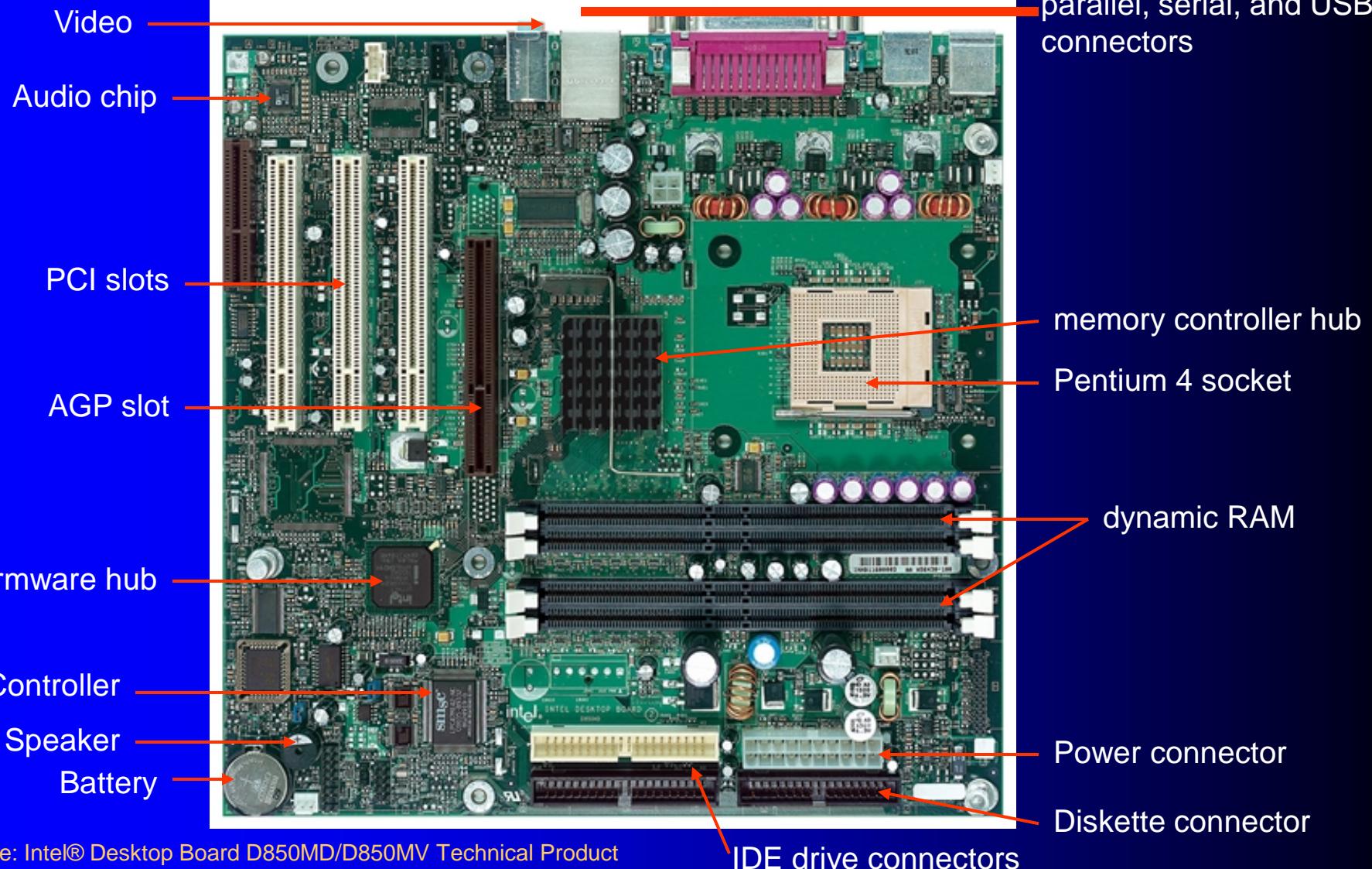
Components of an IA-32 Microcomputer

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

Motherboard

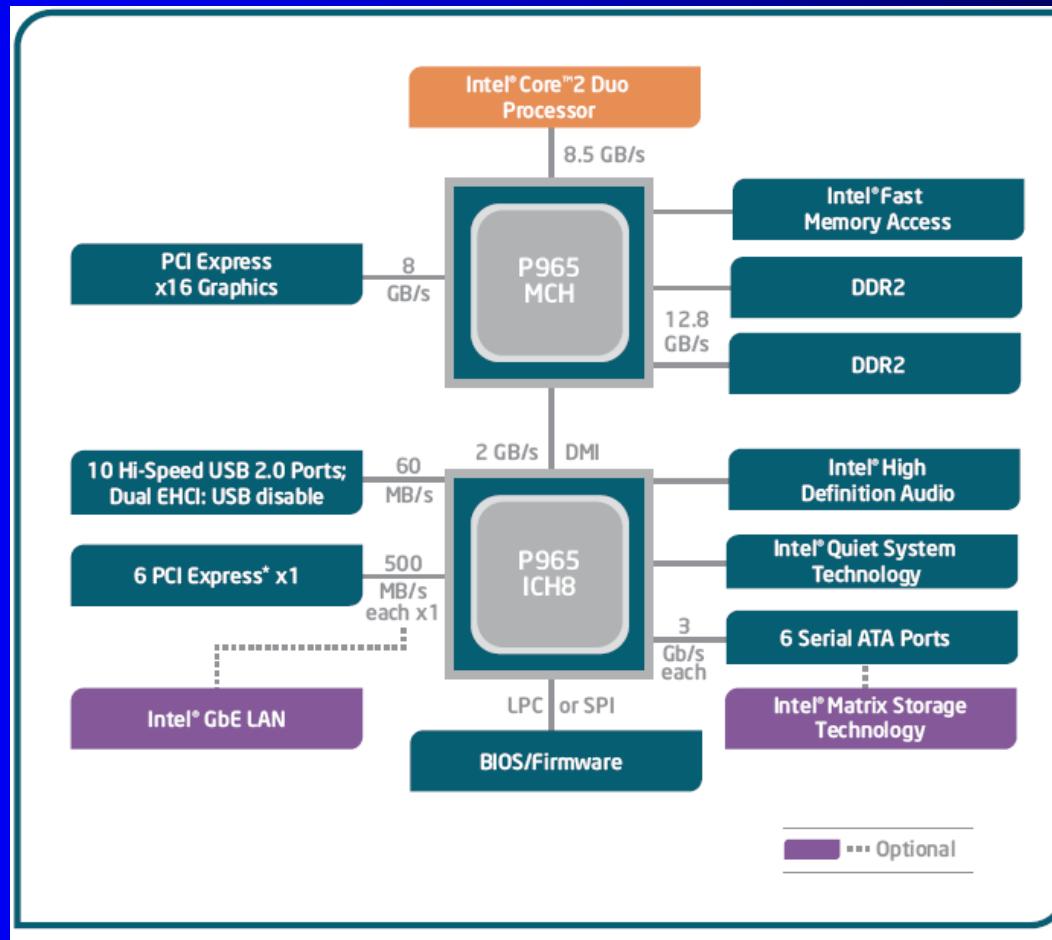
- CPU socket
- External cache memory slots
- Main memory slots
- BIOS chips
- Sound synthesizer chip (optional)
- Video controller chip (optional)
- IDE, parallel, serial, USB, video, keyboard, joystick, network, and mouse connectors
- PCI bus connectors (expansion cards)

Intel D850MD Motherboard



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

Intel 965 Express Chipset



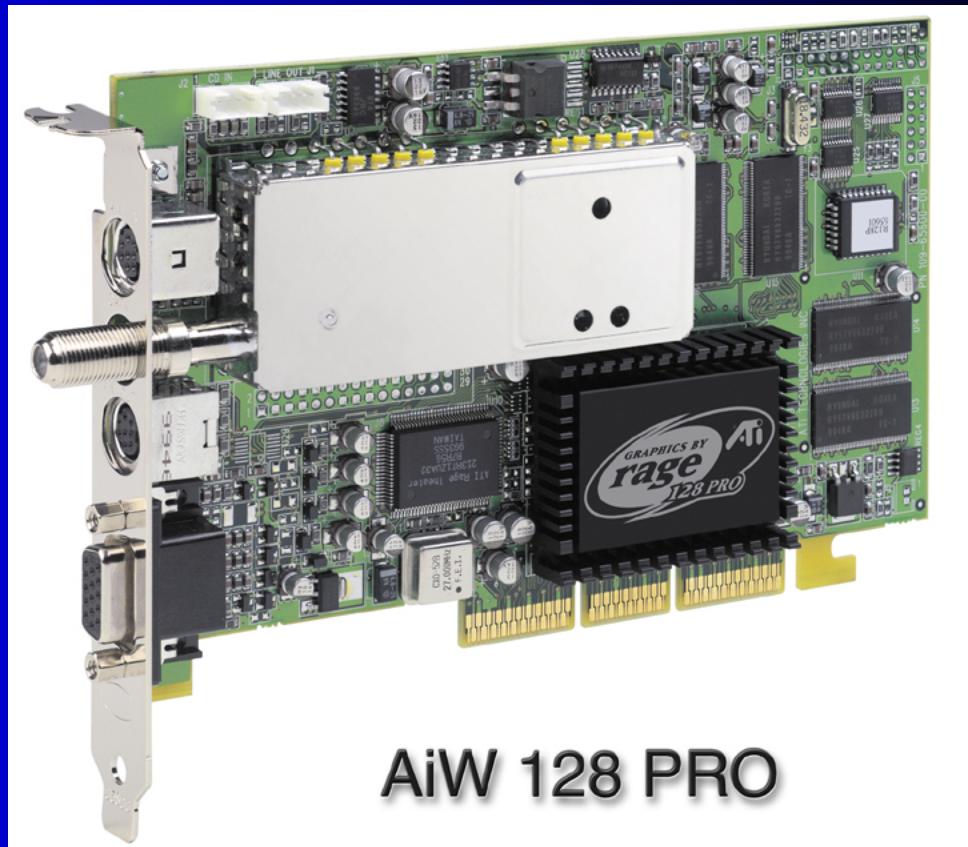
Video Output

- 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

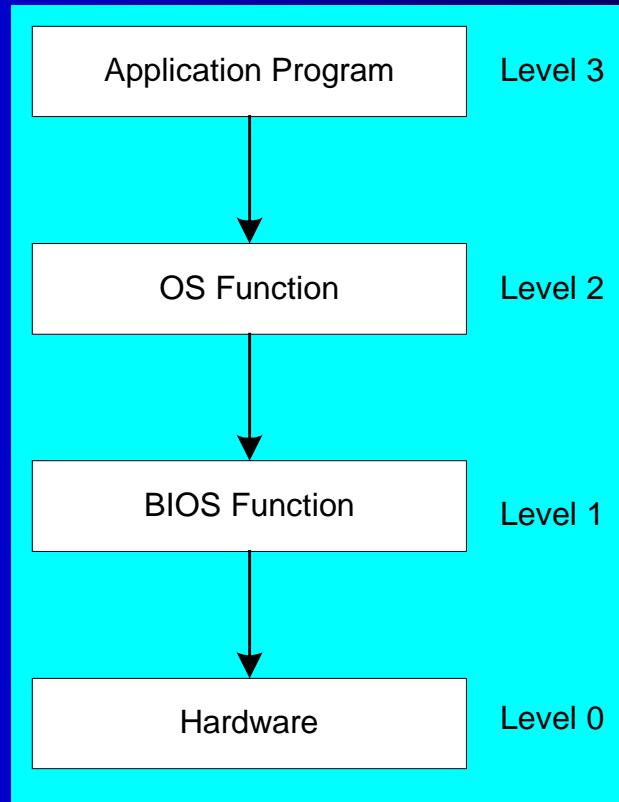
- General Concepts
- IA-32 Processor Architecture
- IA-32 Memory Management
- Components of an IA-32 Microcomputer
- **Input-Output System**

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

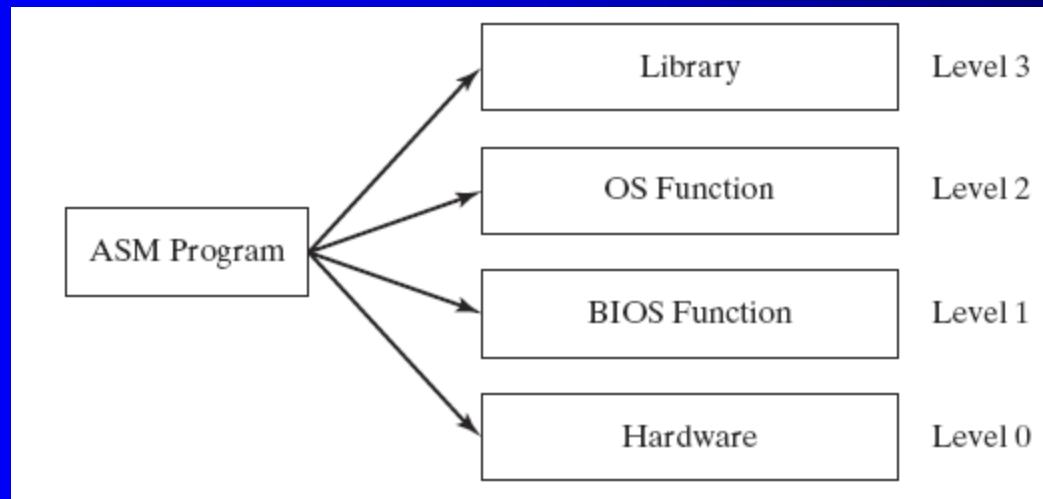
Displaying a String of Characters

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



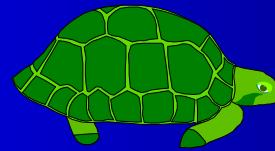
Programming levels

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

Kip Irvine

Chapter 3: Assembly Language Fundamentals

Slides prepared by the author

Revision date: 1/15/2014

Chapter Overview

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

Basic Elements of Assembly Language

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

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
- Identifiers
 - 1-247 characters, including digits
 - not case sensitive
 - first character must be a letter, _, @, ?, or \$

Directives

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

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)

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

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

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

```
; AddTwo.asm - adds two 32-bit integers

.386
.model flat,stdcall
.stack 4096
ExitProcess PROTO, dwExitCode:DWORD
.code
main PROC
    mov eax,5      ; move 5 to the EAX register
    add eax,6      ; add 6 to the EAX register
    INVOKE ExitProcess,0
main ENDP
END main
```

Example Output

Showing registers and flags in the debugger:

EAX=00030000	EBX=7FFDF000	ECX=00000101	EDX=FFFFFFFFF		
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 2)

- 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 2)

- 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

Required Coding Standards

- (to be filled in by the professor)

Program Template

; Program Template (Template.asm)

; Program Description:
; Author:
; Creation Date:
; Revisions:
; Date: Modified by:

```
.386
.model flat,stdcall
.stack 4096
ExitProcess PROTO, dwExitCode:DWORD
.data
; declare variables here
.code
main PROC
    ; write your code here
    INVOKE ExitProcess,0
main ENDP
; (insert additional procedures here)
END main
```

What's Next

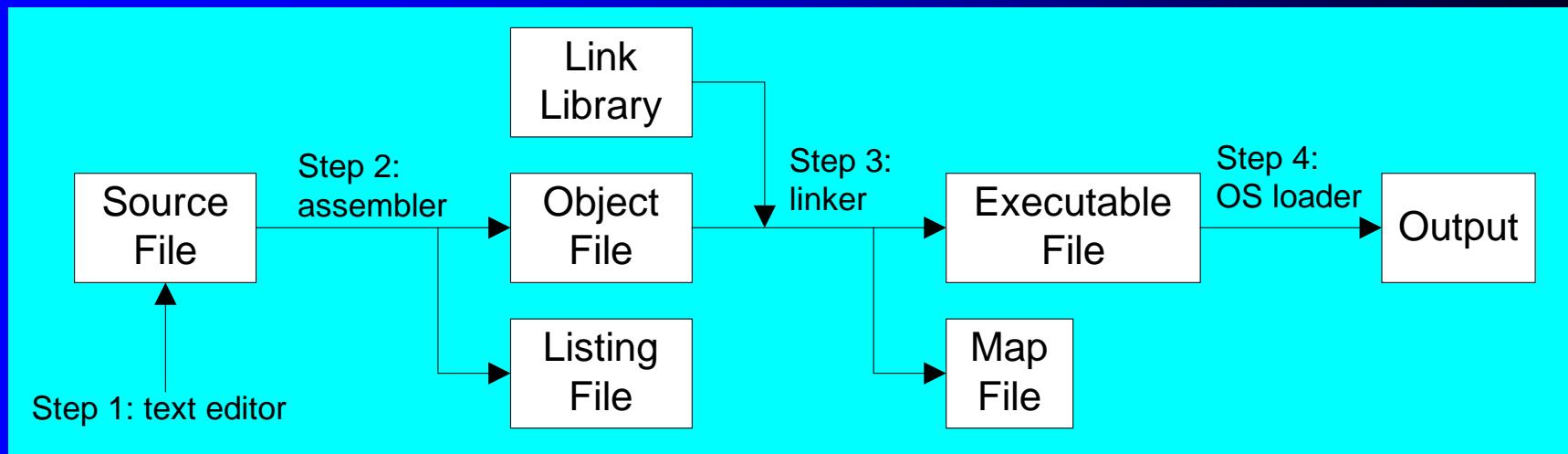
- Basic Elements of Assembly Language
- Example: Adding and Subtracting Integers
- **Assembling, Linking, and Running Programs**
- Defining Data
- Symbolic Constants
- 64-Bit Programming

Assembling, Linking, and Running Programs

- Assemble-Link-Execute Cycle
- Listing File
- Map File

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.



Listing File

- Use it to see how your program is compiled
- Contains
 - source code
 - addresses
 - object code (machine language)
 - segment names
 - symbols (variables, procedures, and constants)

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

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 Data
- Defining TBYTE Data
- Defining 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] . . .

The diagram illustrates the mapping of the syntax components to the assembly code. Three yellow arrows point downwards from the components to the corresponding tokens in the assembly code below. The first arrow points from 'name' to 'value1'. The second arrow points from 'directive' to 'BYTE'. The third arrow points from 'initializer' to '10'.

value1 BYTE 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 1000000000123456789Ah
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
- Assembling, Linking, and Running Programs
- Defining Data
- **Symbolic Constants**
- 64-Bit Programming

Symbolic Constants

- Equal-Sign Directive
- Calculating the Sizes of Arrays and Strings
- EQU Directive
- TEXTEQU 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)
```

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
```

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
```

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"
```

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**

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

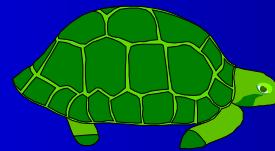
```
1: ; AddTwoSum_64.asm - Chapter 3 example.  
3: ExitProcess PROTO  
5: .data  
6: sum DWORD 0  
8: .code  
9: main PROC  
10:    mov  eax,5  
11:    add  eax,6  
12:    mov  sum,eax  
13:  
14:    mov  ecx,0  
15:    call ExitProcess  
16: main ENDP  
17: END
```

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

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

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

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

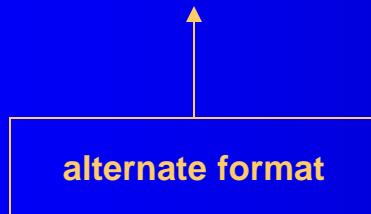
Instruction Operand Notation

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
- CS, EIP, and IP cannot be the destination
- No immediate to segment moves

```
.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
```

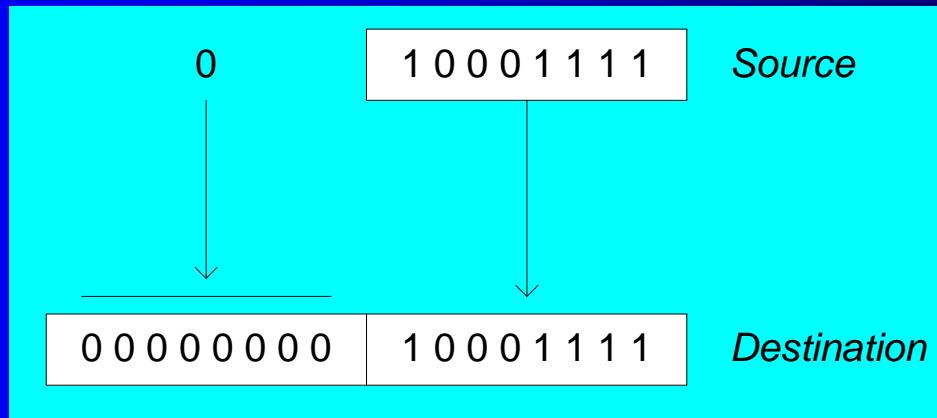
Your turn . . .

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.

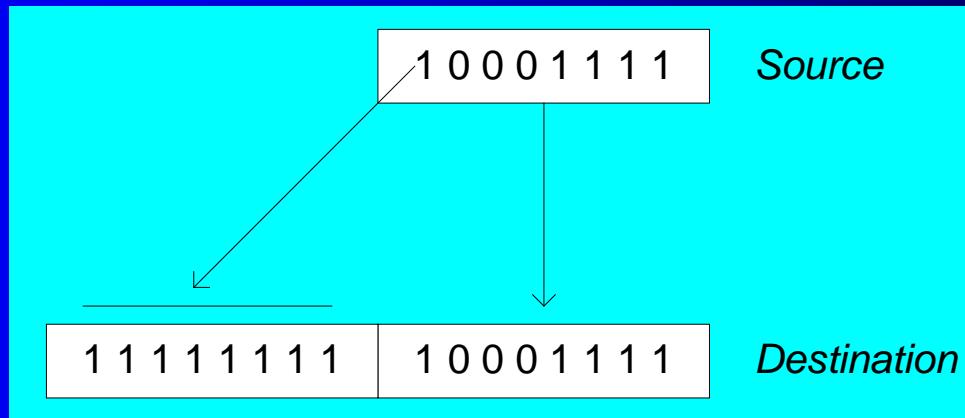


```
mov bl,10001111b  
movzx ax,bl           ; zero-extension
```

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           ; AL = 20h  
mov al,[arrayB+1]         ; 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
```

```
; 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

INC and DEC Instructions

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

INC and DEC Examples

```
.data
myWord WORD 1000h
myDword DWORD 1000000h
.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

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 = -26
    mov ebx,Yval
    sub ebx,Zval    ; EBX = -10
    add eax,ebx
    mov Rval,eax    ; -36
```

Your turn...

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

Rval = Xval - (-Yval + Zval)

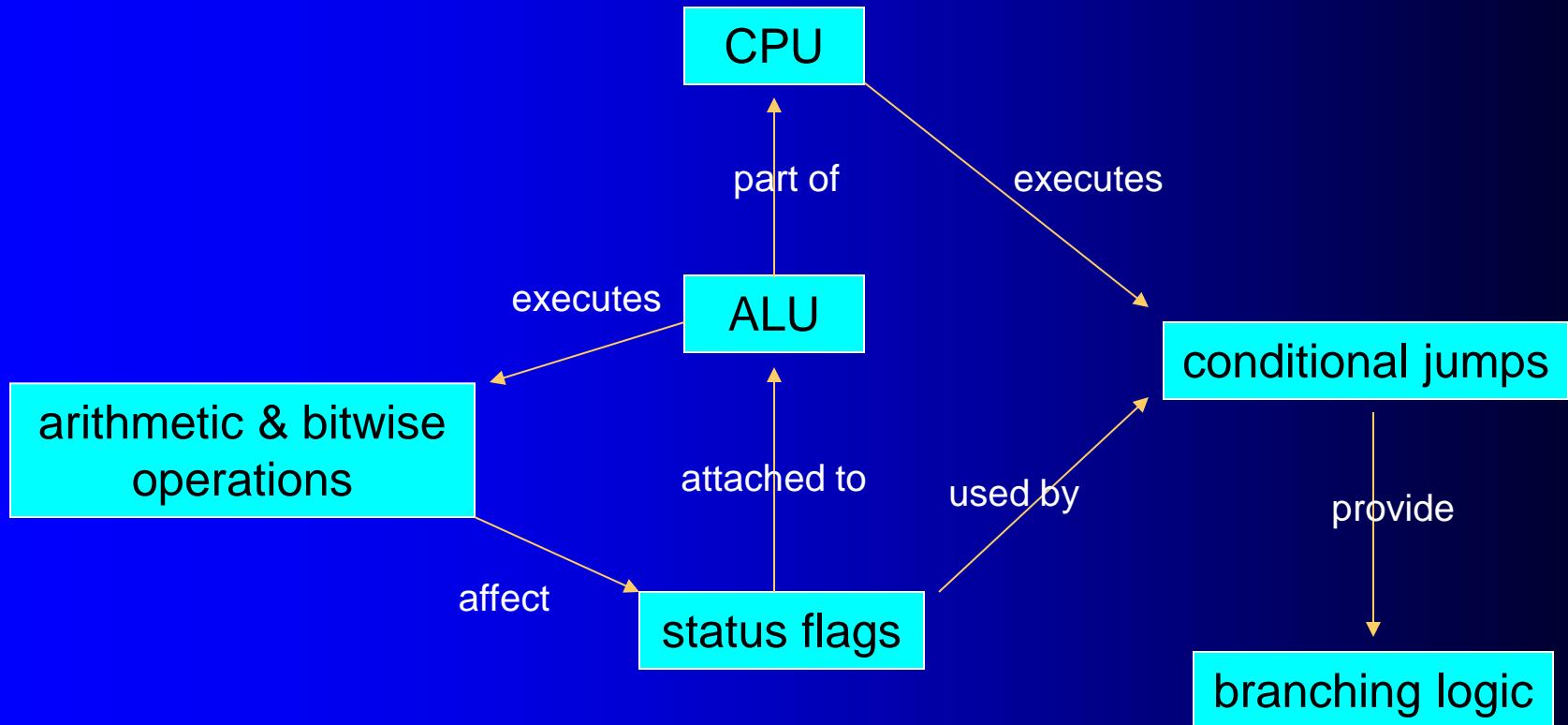
Assume that all values are signed doublewords.

```
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 ax,0FFFFh  
inc ax           ; AX = 0, ZF = 1  
inc ax           ; AX = 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.

```
mov cx,0  
sub cx,1 ; CX = -1, SF = 1  
add cx,2 ; CX = 1, SF = 0
```

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

```
mov al,0  
sub al,1 ; AL = 11111111b, SF = 1  
add al,2 ; AL = 00000001b, SF = 0
```

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

Overflow and Carry Flags

A Hardware Viewpoint

- How the ADD instruction affects OF and CF:
 - CF = (carry out of the MSB)
 - OF = CF XOR 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 = CF XOR 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
```

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
```

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

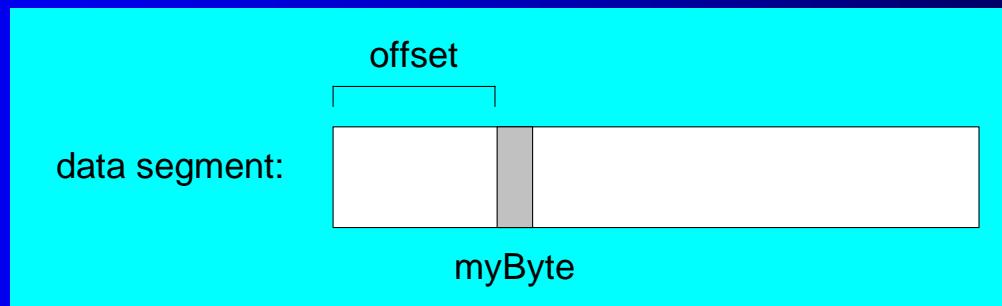
- Data Transfer Instructions
- Addition and Subtraction
- **Data-Related Operators and Directives**
- Indirect Addressing
- JMP and LOOP Instructions
- 64-Bit Programming

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 ?  
wVal WORD ?  
dVal DWORD ?  
dVal2 DWORD ?  
  
.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      ; loads 5678h  
  
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:

byte	offset
78	0000
56	0001
34	0002
12	0003

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 2)

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 2)

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
```

ToDo: 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.
```

To Do: 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:
 - $ECX \leftarrow ECX - 1$
 - if $ECX \neq 0$, 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:

offset	machine code	source code
00000000	66 B8 0000	mov ax,0
00000004	B9 00000005	mov ecx,5
00000009	66 03 C1	L1: add ax,cx
0000000C	E2 FB	loop L1
0000000E		

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

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

How many times will the loop execute?

4,294,967,296

```
mov ecx,0  
X2:  
    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  SIZEOF source DUP(0)

.code
    mov    esi,0                      ; index register
    mov    ecx,SIZEOF 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

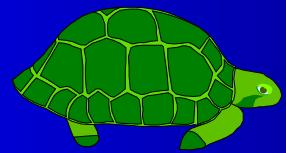
- MOVSXD sign extends a 32-bit value into a 64-bit destination register
- The OFFSET operator generates a 64-bit address
- LOOP uses the 64-bit RCX register as a counter
- RSI and RDI are the most common 64-bit index registers for accessing arrays.

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.

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



46696E616C

Assembly Language for x86 Processors

7th Edition

Kip R. Irvine

Chapter 5: Procedures

Slides prepared by the author

Revision date: 1/15/2014

Chapter Overview

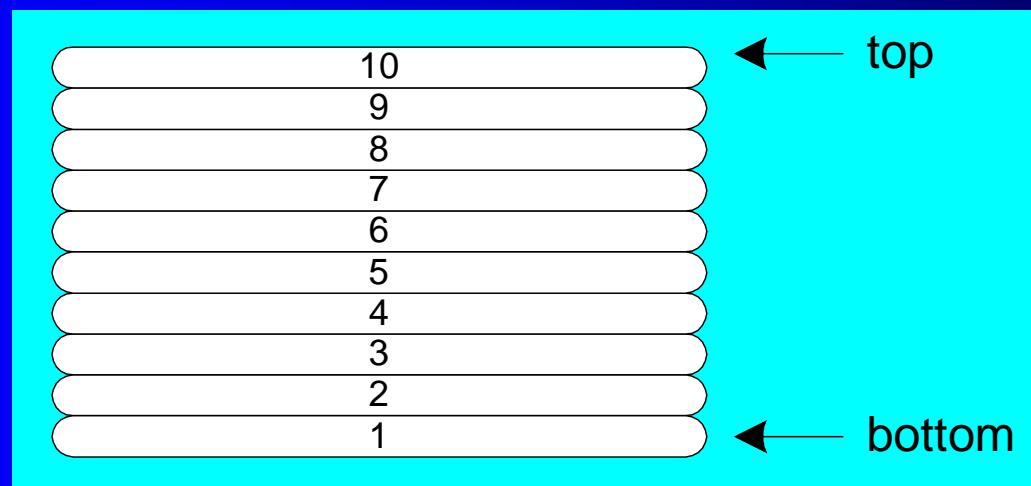
- Stack Operations
- Defining and Using Procedures
- Linking to an External Library
- The Irvine32 Library
- 64-Bit Assembly Programming

Stack Operations

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

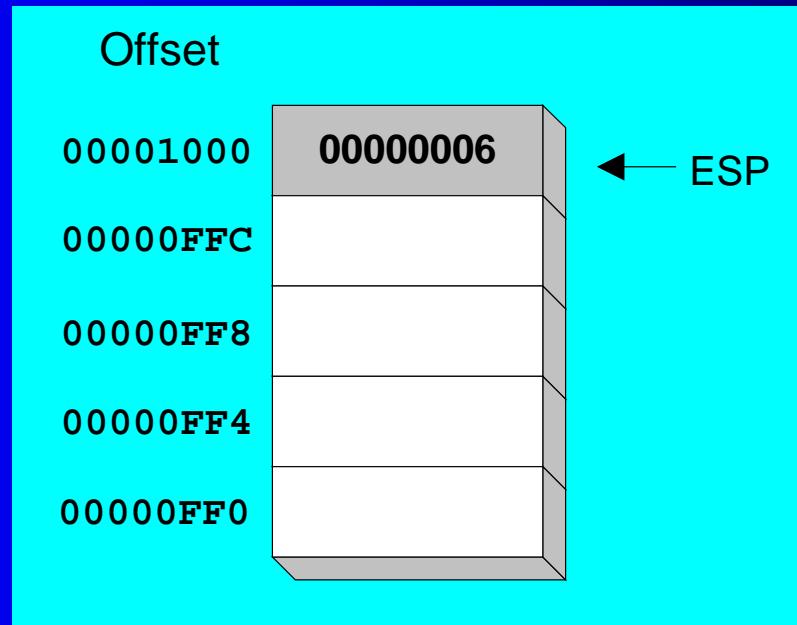
Runtime Stack

- Imagine a stack of plates . . .
 - plates are only added to the top
 - plates are only removed from the top
 - LIFO structure



Runtime Stack

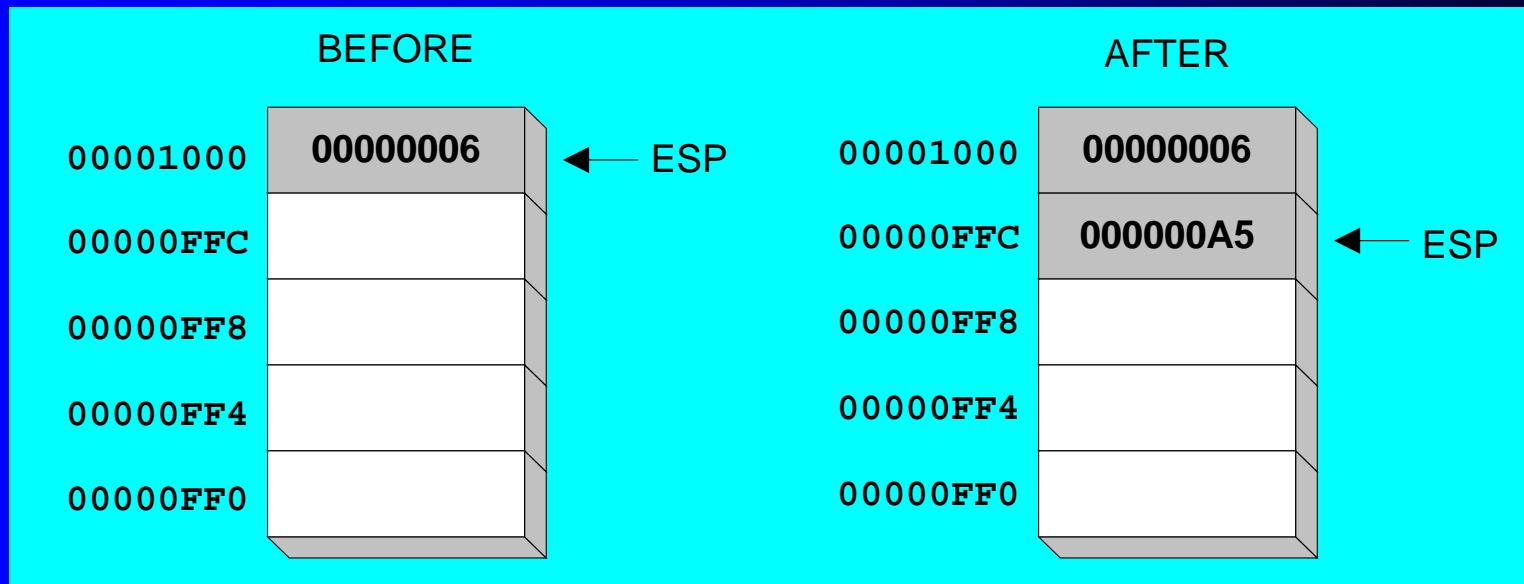
- Managed by the CPU, using two registers
 - SS (stack segment)
 - ESP (stack pointer) *



* SP in Real-address mode

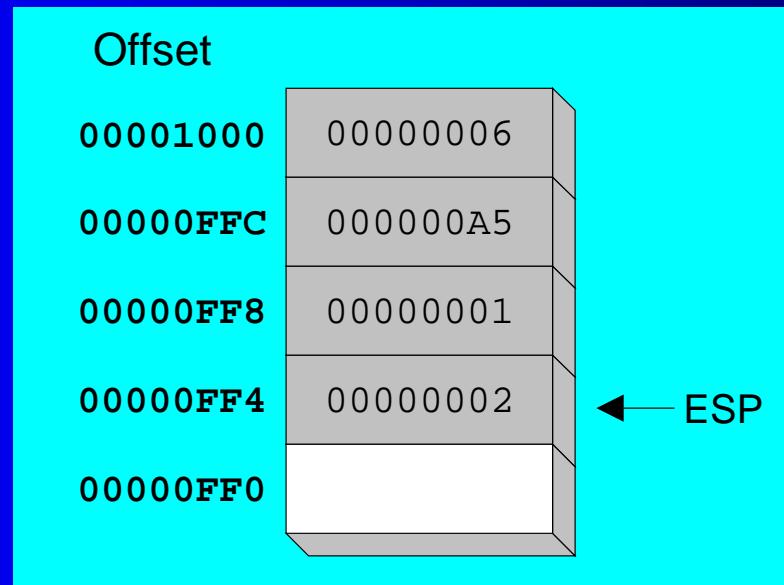
PUSH Operation (1 of 2)

- 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)

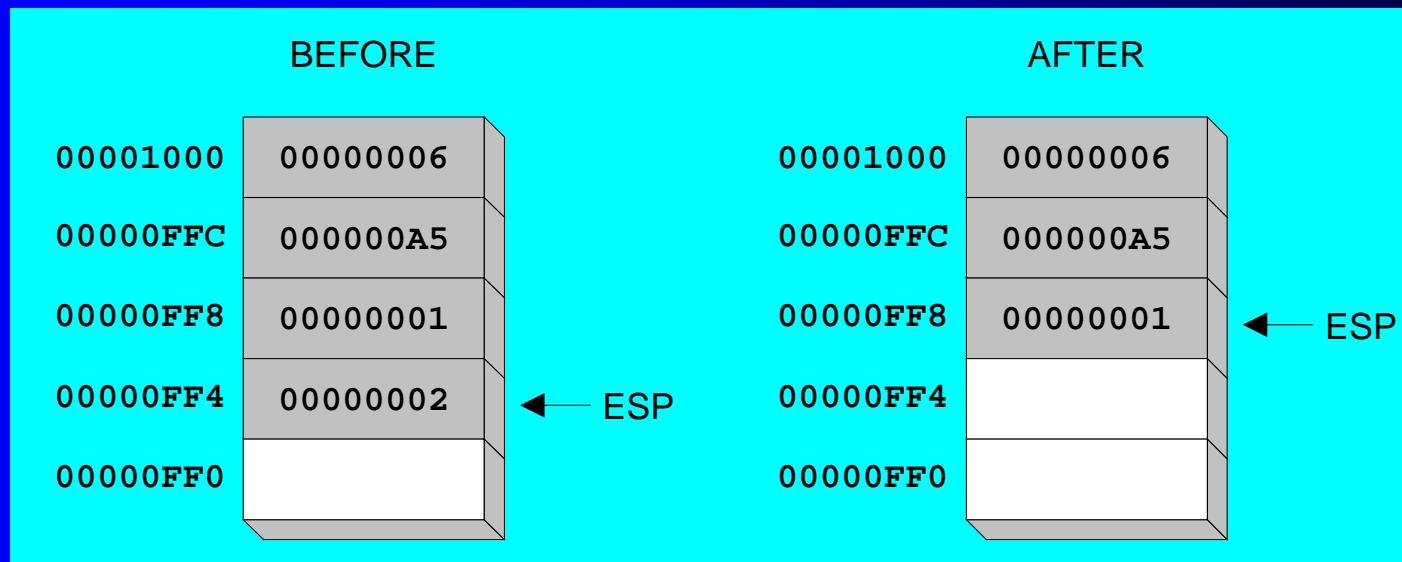
- Same stack after pushing two more integers:



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



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.

```
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

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
```

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
- Flowchart Symbols
- USES Operator

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 2)

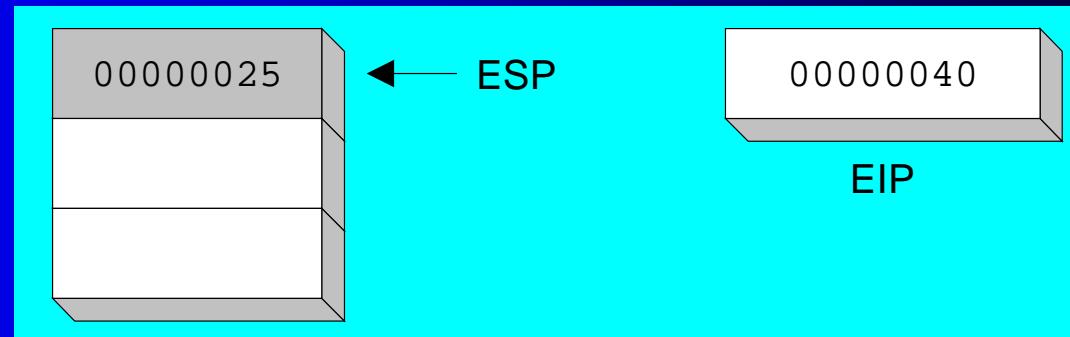
0000025 is the offset of the instruction immediately following the CALL instruction

00000040 is the offset of the first instruction inside MySub

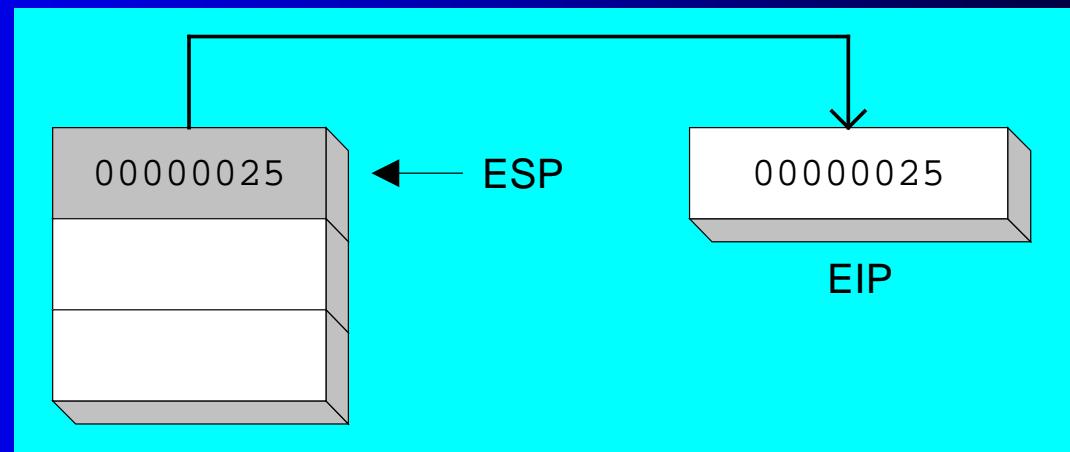
```
main PROC  
    00000020 call MySub  
    00000025 mov eax,ebx  
    .  
    .  
main ENDP  
  
MySub PROC  
    00000040 mov eax,edx  
    .  
    .  
    ret  
MySub ENDP
```

CALL-RET Example (2 of 2)

The CALL instruction pushes 00000025 onto the stack, and loads 00000040 into EIP



The RET instruction pops 00000025 from the stack into EIP



(stack shown before RET executes)

Nested Procedure Calls

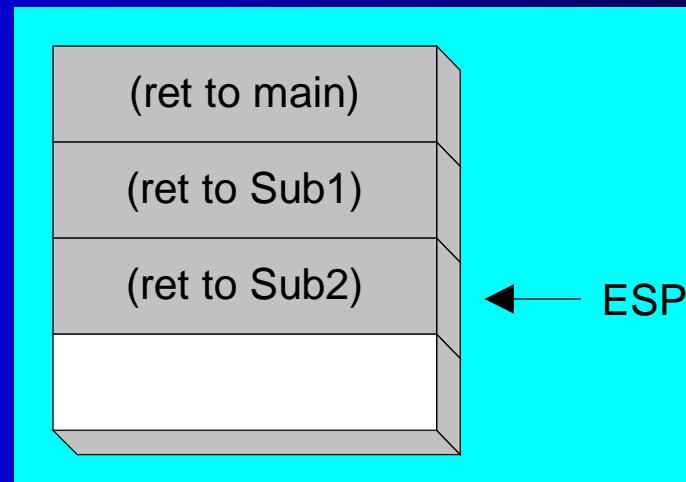
```
main PROC
    .
    .
    call Sub1
    exit
main ENDP

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
```

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

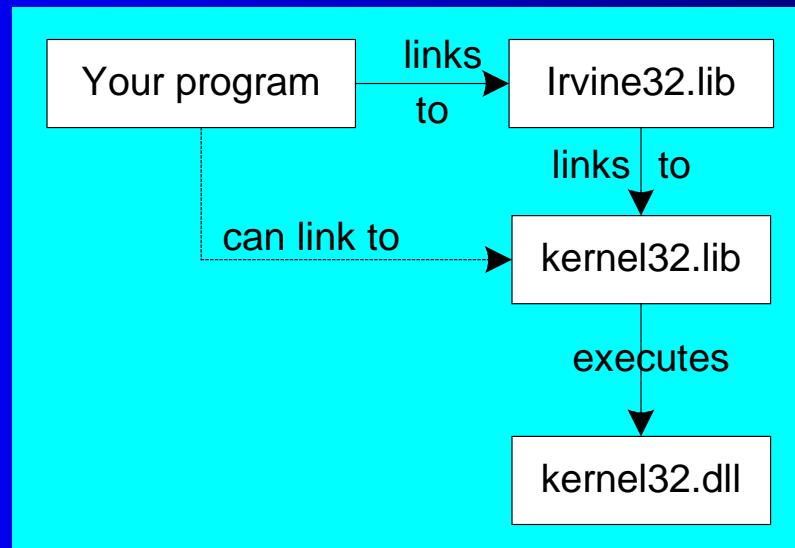
- What is a Link Library?
- How the Linker Works

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

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 4)

CloseFile – Closes an open disk file

Clrscr - 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 4)

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 FFFFFFFFh

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 4)

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 4)

Str_ucase - 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 4)

`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
```

Sample output:

```
EAX=00000613 EBX=00000000 ECX=000000FF EDX=00000000
ESI=00000000 EDI=00000100 EBP=0000091E ESP=000000F6
EIP=00401026 EFL=00000286 CF=0 SF=1 ZF=0 OF=0
```

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
```

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
```

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 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 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 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.

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
```

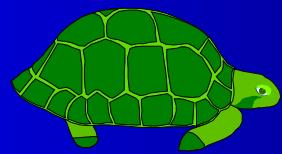
The x64 Calling Convention

- 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.

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

Assembly Language for x86 Processors

7th 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

Boolean and Comparison Instructions

- CPU Status Flags
- AND Instruction
- OR Instruction
- XOR Instruction
- NOT Instruction
- Applications
- TEST Instruction
- CMP Instruction

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
cleared	<hr/>
	00001011
	unchanged

AND

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

OR Instruction

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

`OR destination, source`

OR

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

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

XOR Instruction

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

XOR *destination, source*

	00111011			
XOR	00001111			
unchanged	<hr/> <table><tbody><tr><td>0</td><td>011</td><td>0100</td></tr></tbody></table> inverted	0	011	0100
0	011	0100		

XOR

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

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	00111011
	11000100 — inverted

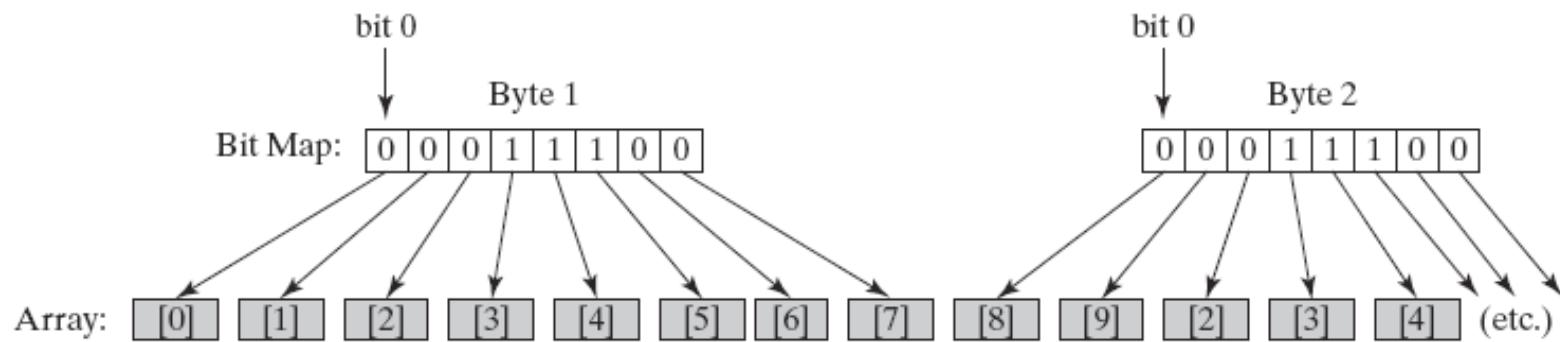
NOT

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

```
mov al,'a'          ; AL = 01100001b
and al,11011111b    ; AL = 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.
- 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.
- Example: jump to a label if either bit 0 or bit 1 in AL is set.

```
test al,00000011b  
jnz ValueFound
```

- Example: jump to a label if neither bit 0 nor bit 1 in AL is set.

```
test al,00000011b  
jz ValueNotFound
```

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)

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
```

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

- Jumps Based On . . .
 - Specific flags
 - 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

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

Jumps Based on Equality

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

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)

Jumps Based on Signed Comparisons

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
```

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: «ççÄää-Äç-ïÄÿü-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:

- $\text{ECX} \leftarrow \text{ECX} - 1$

- if $\text{ECX} > 0$ and $\text{ZF}=1$, 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:
 - $\text{ECX} \leftarrow \text{ECX} - 1;$
 - if $\text{ECX} > 0$ and $\text{ZF}=0$, 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:
    mov x,1         ; both are true
    ; 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
    call Crlf
    jmp L3
L2: add ebx,EntrySize             ; and exit the loop
    loop L1                         ; point to next entry
                                    ; repeat until ECX = 0

L3:                                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

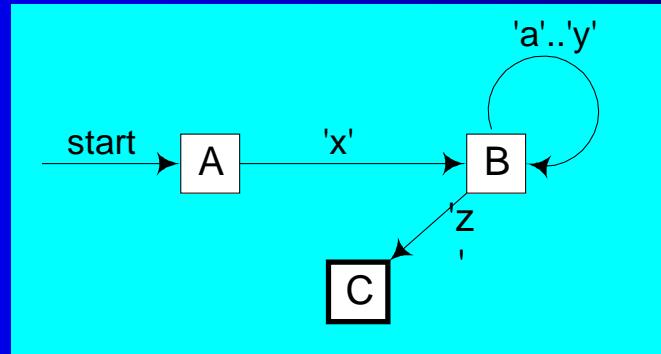
- A FSM is a specific instance of a more general structure called a directed graph.
- Three basic states, represented by nodes:
 - Start state
 - Terminal state(s)
 - Nonterminal state(s)

Finite-State Machine

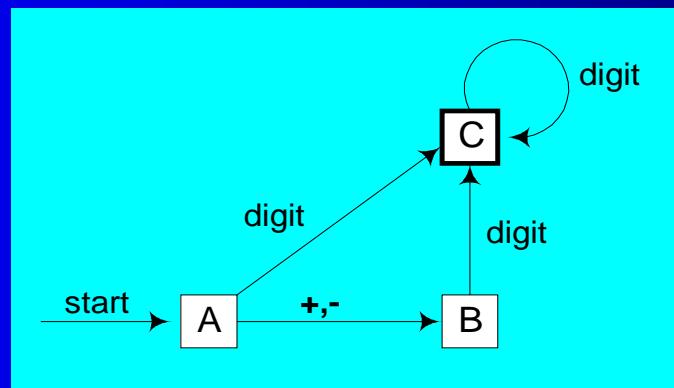
- Accepts any sequence of symbols that puts it into an accepting (final) state
- Can be used to recognize, or validate a sequence of characters that is governed by language rules (called a regular expression)
- Advantages:
 - Provides visual tracking of program's flow of control
 - Easy to modify
 -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

Finite-State Machine Examples

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

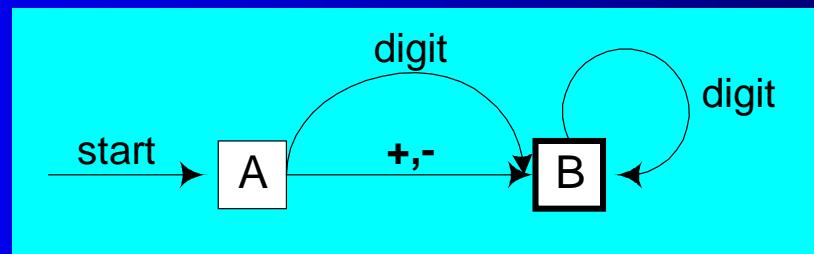


- FSM that recognizes signed integers:



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](#).

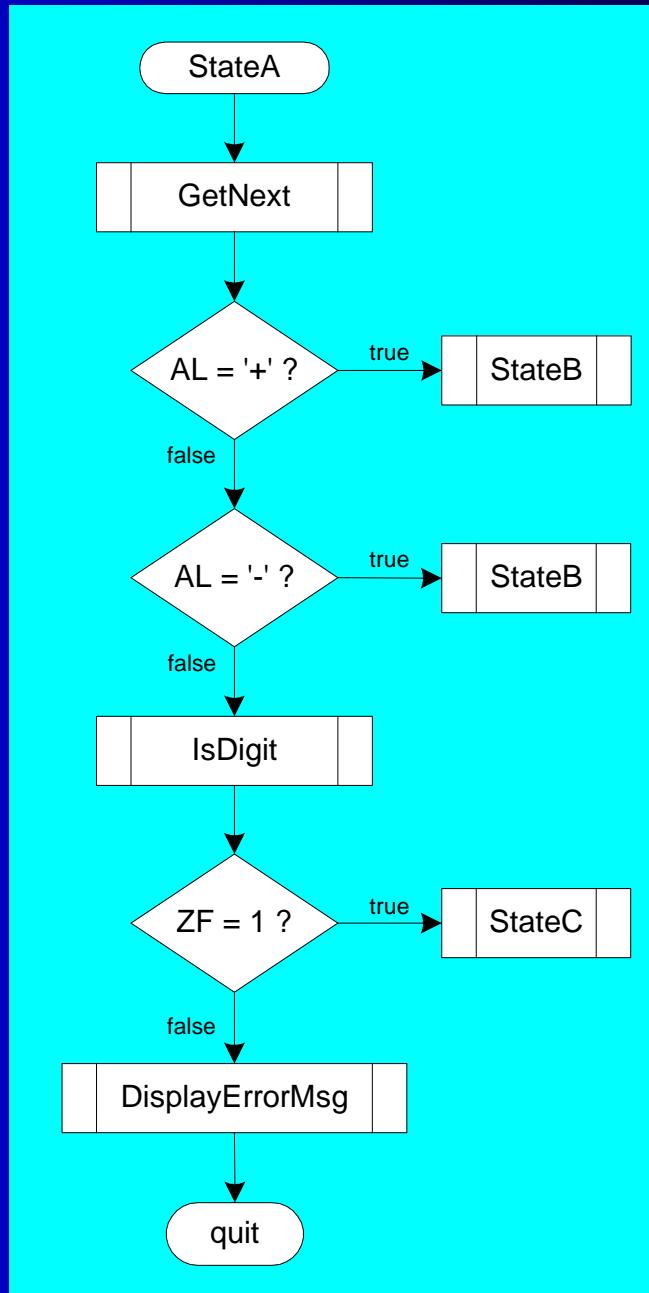
IsDigit Procedure

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 $expr1$ 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

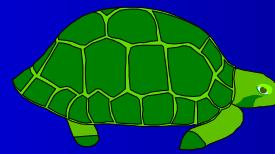
Tests the loop condition before executing the loop body. The .ENDW directive marks the end of the loop.

Example:

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

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

Assembly Language for x86 Processors

7th Edition

Kip R. Irvine

Chapter 7: Integer Arithmetic

Slides prepared by the author

Revision date: 1/15/2014

Chapter Overview

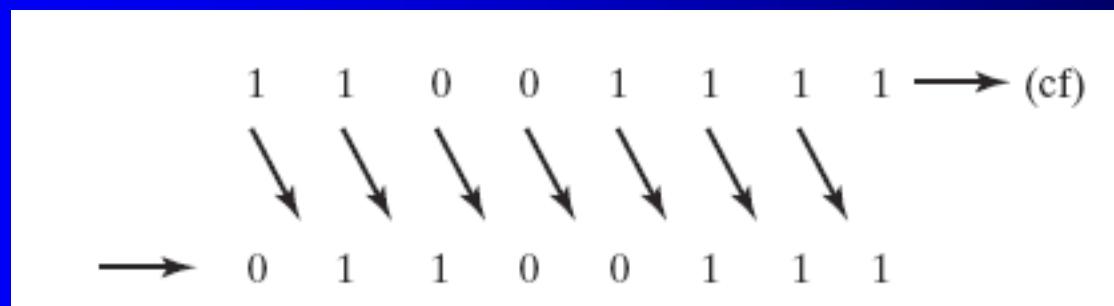
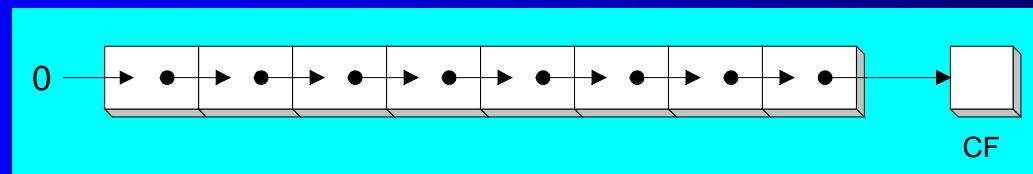
- **Shift and Rotate Instructions**
- Shift and Rotate Applications
- Multiplication and Division Instructions
- Extended Addition and Subtraction
- ASCII and Unpacked Decimal Arithmetic
- Packed Decimal Arithmetic

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

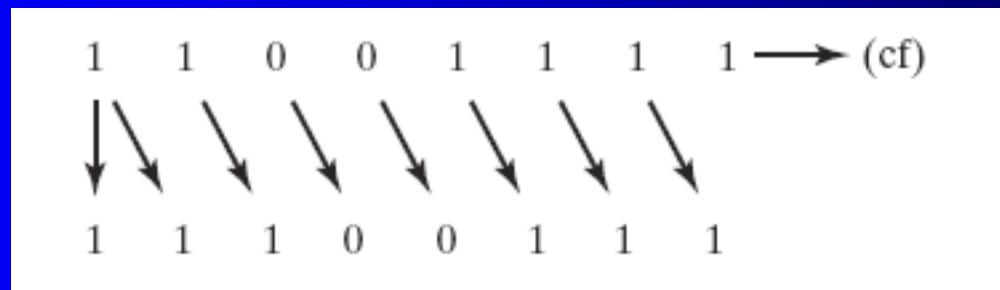
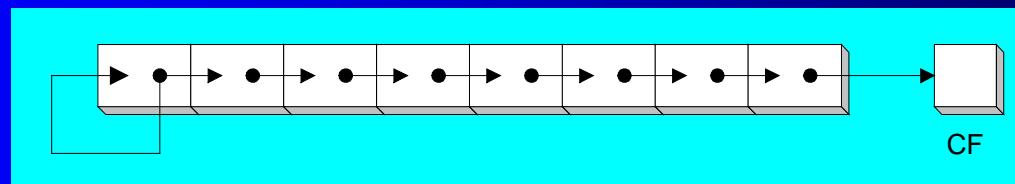
Logical Shift

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



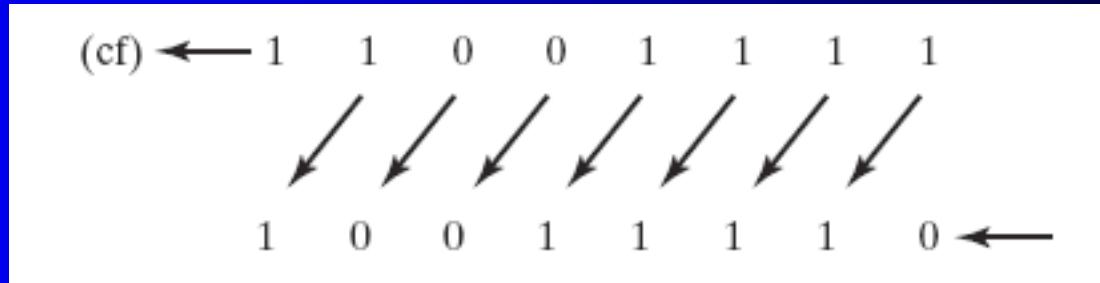
Arithmetic Shift

- 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

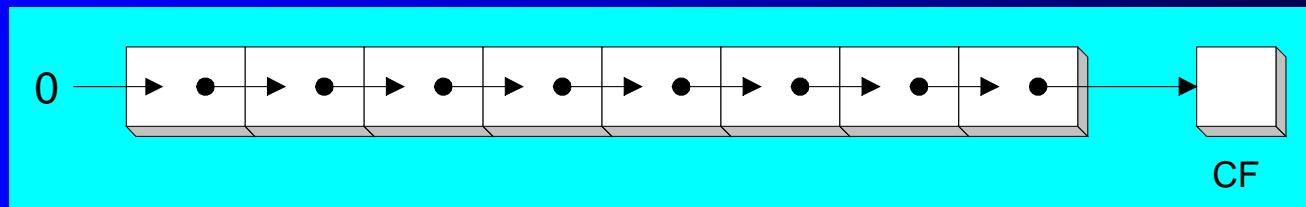
Shifting left n bits multiplies the operand by 2^n

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.

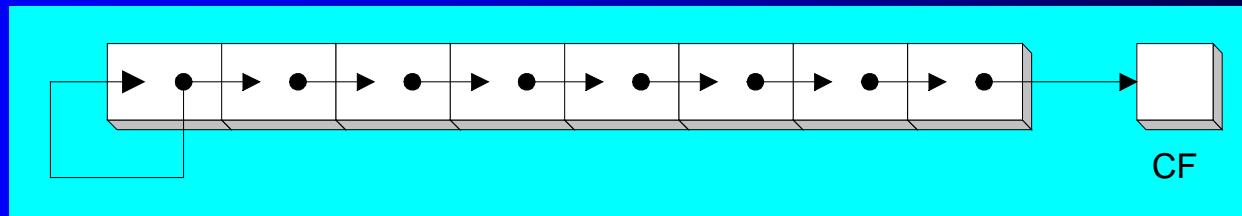


Shifting right n bits divides the operand by 2^n

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

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.

```
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**

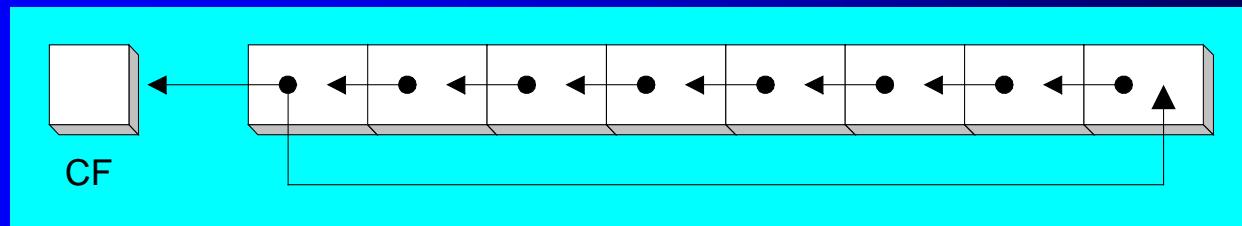
b. **A8h**

c. **C6h**

d. **F8h**

ROL Instruction

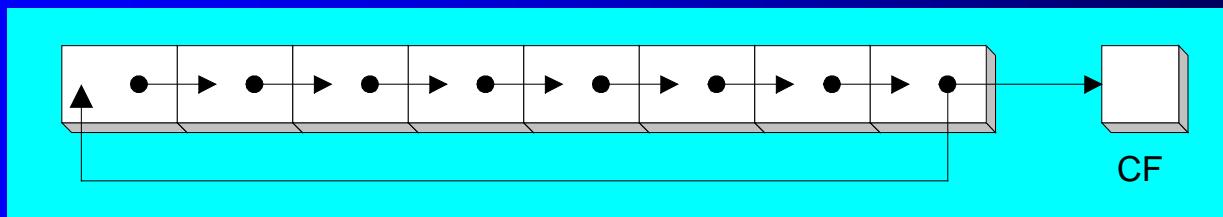
- ROL (rotate) shifts each bit to the left
- The highest bit is copied into both the Carry flag and into the lowest bit
- No bits are lost



```
mov al,11110000b  
rol al,1           ; AL = 11100001b  
  
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



```
mov al,11110000b  
ror al,1           ; AL = 01111000b  
  
mov dl,3Fh  
ror dl,4          ; DL = F3h
```

Your turn . . .

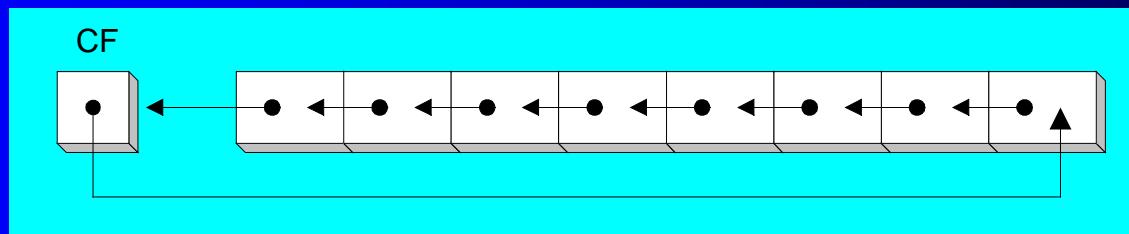
Indicate the hexadecimal value of AL after each rotation:

```
mov al,6Bh  
ror al,1  
rol al,3
```

- a. B5h
- b. ADh

RCL Instruction

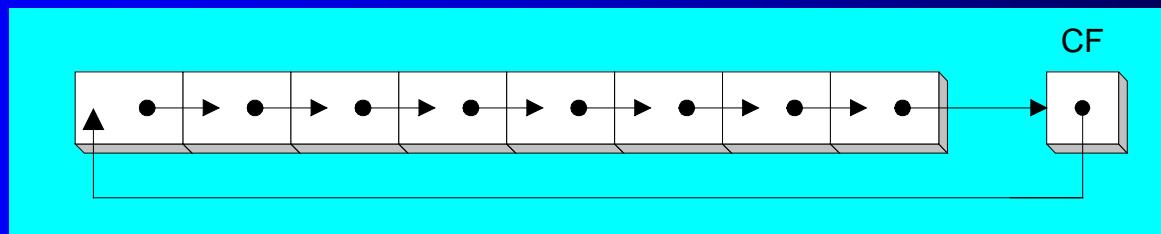
- RCL (rotate carry left) shifts each bit to the left
- Copies the Carry flag to the least significant bit
- Copies the most significant bit to the Carry flag



```
clc ; CF = 0  
mov bl,88h ; CF,BL = 0 10001000b  
rcl bl,1 ; CF,BL = 1 00010000b  
rcl bl,1 ; CF,BL = 0 00100001b
```

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



```
stc          ; CF = 1  
mov ah,10h    ; CF,AH = 1 00010000b  
rcr ah,1      ; CF,AH = 0 10001000b
```

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

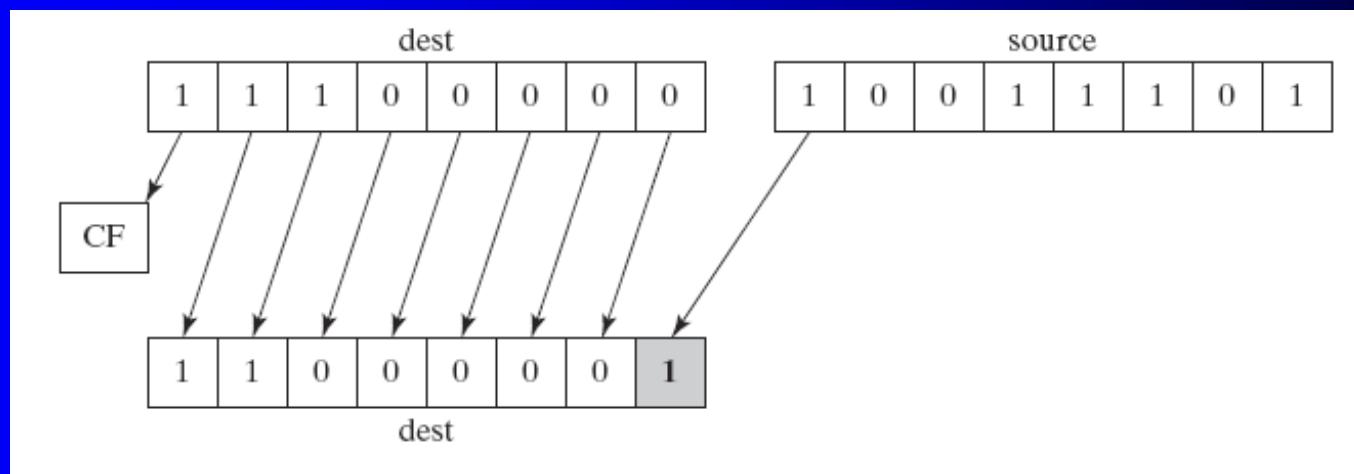
- Shifts a destination operand a given number of bits to the left
- The bit positions opened up by the shift are filled by the most significant bits of the source operand
- The source operand is not affected
- Syntax:
SHLD destination, source, count
- Operand types:

```
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
```

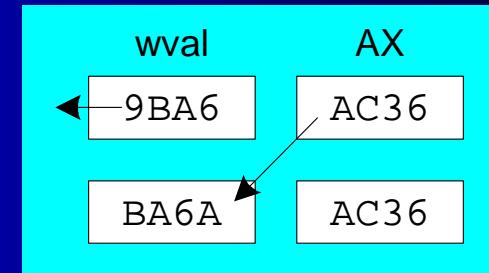


Another SHLD Example

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

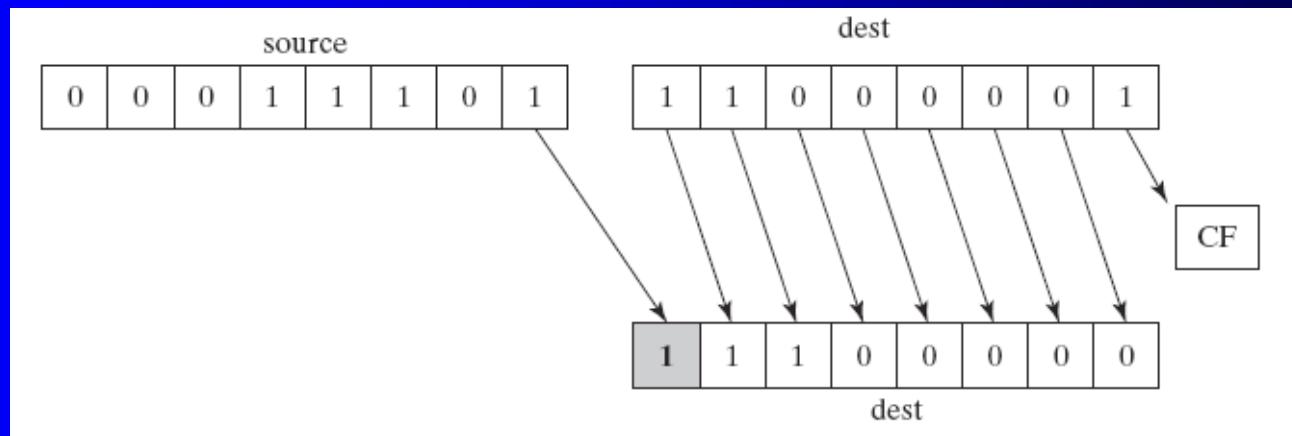
- Shifts a destination operand a given number of bits to the right
- The bit positions opened up by the shift are filled by the least significant bits of the source operand
- The source operand is not affected
- Syntax:
SHRD destination, source, count
- Operand types:

```
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
```



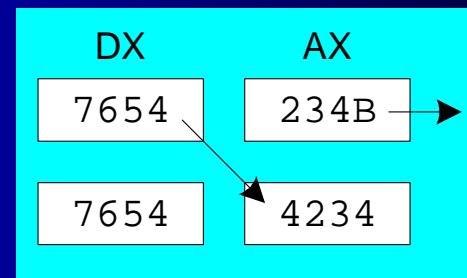
Another SHRD Example

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

- 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.
- 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

0 1 1 1 1 0 1 1	123
× 0 0 1 0 0 1 0 0	36
<hr/>	
0 1 1 1 1 0 1 1	123 SHL 2
+ 0 1 1 1 1 0 1 1	123 SHL 5
<hr/>	
0 0 0 1 0 0 0 1 0 1 0 0 1 1 0 0	4428

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:

```
EAX * 36  
= EAX * (32 + 4)  
= (EAX * 32)+(EAX * 4)
```

```
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

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 = 000000012345000h, CF=0
```

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  
mov bl,4  
imul bl          ; AX = 00C0h, OF=1
```

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:

`DIV reg/mem8`

`DIV reg/mem16`

`DIV reg/mem32`

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

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
dividend_hi QWORD 00000108h
dividend_lo QWORD 33300020h
divisor QWORD 00010000h

.code
mov rdx, dividend_hi
mov rax, dividend_lo
div divisor
; RAX = quotient
; RDX = remainder
```

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 . . .

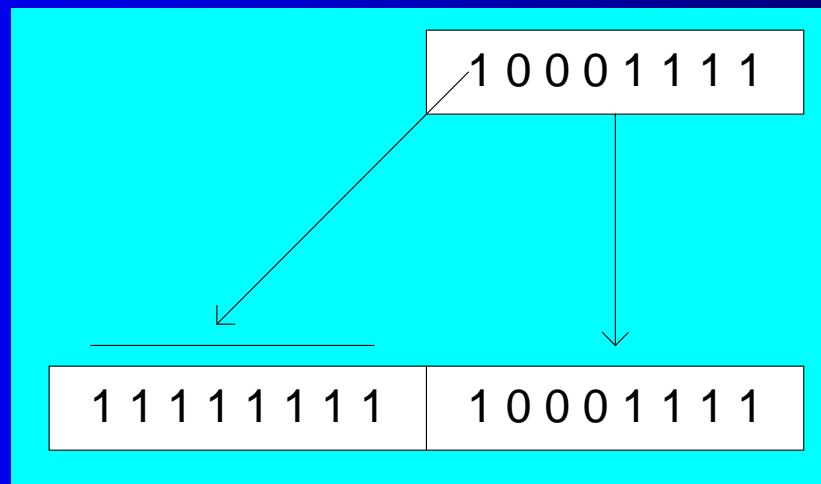
What will be the hexadecimal values of DX and AX after the following instructions execute? Or, if divide overflow occurs, you can indicate that as your answer:

```
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

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 (1 of 2)

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
mov  ebx,edx            ; EDX = remainder
; 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 = 00000001FFFFFFEh
```

Extended Addition Example

- Task: Add 1 to EDX:EAX
 - Starting value of EDX:EAX: 00000000FFFFFFFFh
 - 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,0FFFFFFFh ; 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

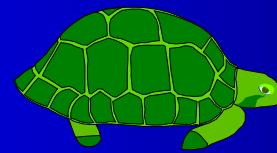
AL = 93h, CF = 1

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.

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

Assembly Language for x86 Processors

7th Edition

Kip R. Irvine

Chapter 8: Advanced Procedures

Slides prepared by the author.

Revision date: 1/15/2014

Chapter Overview

- **Stack Frames**
- Recursion
- INVOKE, ADDR, PROC, and PROTO
- Creating Multimodule Programs
- Advanced Use of Parameters (optional)
- Java Bytecodes (optional)

Stack Frames

- Stack Parameters
- Local Variables
- ENTER and LEAVE Instructions
- LOCAL Directive
- WriteStackFrame Procedure

Stack Frame

- Also known as an *activation record*
- Area of the stack set aside for a procedure's return address, passed parameters, saved registers, and local variables
- Created by the following steps:
 - Calling program pushes arguments on the stack and calls the procedure.
 - The called procedure pushes EBP on the stack, and sets EBP to ESP.
 - If local variables are needed, a constant is subtracted from ESP to make room on the stack.

Stack Parameters

- More convenient than register parameters
- Two possible ways of calling DumpMem. Which is easier?

```
pushad  
mov esi,OFFSET array  
mov ecx,LENGTHOF array  
mov ebx,TYPE array  
call DumpMem  
popad
```

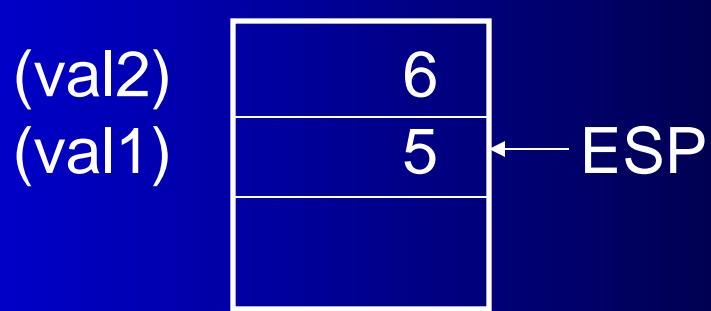
```
push TYPE array  
push LENGTHOF array  
push OFFSET array  
call DumpMem
```

Passing Arguments by Value

- Push argument values on stack
 - (Use only 32-bit values in protected mode to keep the stack aligned)
- Call the called-procedure
- Accept a return value in EAX, if any
- Remove arguments from the stack if the called-procedure did not remove them

Example

```
.data  
val1    DWORD 5  
val2    DWORD 6  
  
.code  
push val2  
push val1
```



Stack prior to CALL

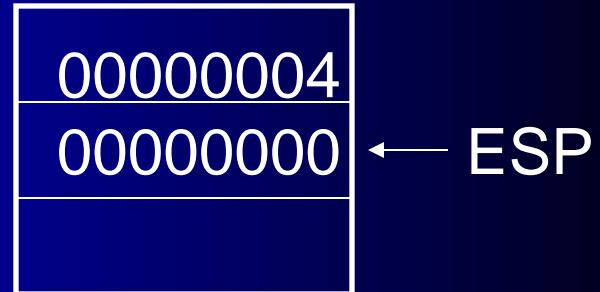
Passing by Reference

- Push the offsets of arguments on the stack
- Call the procedure
- Accept a return value in EAX, if any
- Remove arguments from the stack if the called procedure did not remove them

Example

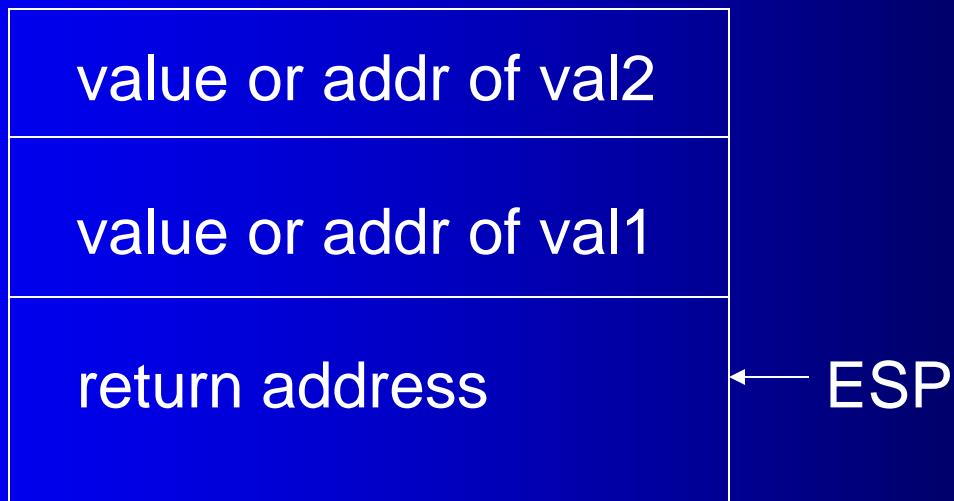
```
.data  
val1    DWORD  5  
val2    DWORD  6  
  
.code  
push OFFSET val2  
push OFFSET val1
```

(offset val2)
(offset val1)



Stack prior to CALL

Stack after the CALL



Passing an Array by Reference (1 of 2)

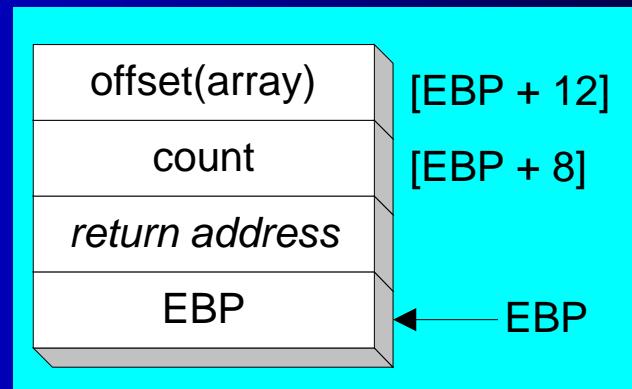
- The `ArrayFill` procedure fills an array with 16-bit random integers
- The calling program passes the address of the array, along with a count of the number of array elements:

```
.data  
count = 100  
array WORD count DUP(?)  
.code  
    push OFFSET array  
    push COUNT  
    call ArrayFill
```

Passing an Array by Reference (2 of 2)

ArrayFill can reference an array without knowing the array's name:

```
ArrayFill PROC
    push ebp
    mov ebp,esp
    pushad
    mov esi,[ebp+12]
    mov ecx,[ebp+8]
    .
    .
```



ESI points to the beginning of the array, so it's easy to use a loop to access each array element. [View the complete program.](#)

Accessing Stack Parameters (C/C++)

- C and C++ functions access stack parameters using constant offsets from EBP¹.
 - Example: [ebp + 8]
- EBP is called the base pointer or frame pointer because it holds the base address of the stack frame.
- EBP does not change value during the function.
- EBP must be restored to its original value when a function returns.

¹ BP in Real-address mode

RET Instruction

- *Return from subroutine*
- Pops stack into the instruction pointer (EIP or IP). Control transfers to the target address.
- Syntax:
 - **RET**
 - **RET n**
- Optional operand *n* causes *n* bytes to be added to the stack pointer after EIP (or IP) is assigned a value.

Who removes parameters from the stack?

Caller (C) or Called-procedure (STDCALL):

```
push val2  
push val1  
call AddTwo  
add esp,8
```

```
AddTwo PROC  
    push ebp  
    mov  ebp,esp  
    mov  eax,[ebp+12]  
    add  eax,[ebp+8]
```

```
    pop  ebp  
ret  8
```

(Covered later: The MODEL directive specifies calling conventions)

Your turn . . .

- Create a procedure named Difference that subtracts the first argument from the second one. Following is a sample call:

```
push 14          ; first argument
push 30          ; second argument
call Difference ; EAX = 16
```

Difference PROC

```
push ebp
mov  ebp,esp
mov  eax,[ebp + 8]      ; second argument
sub  eax,[ebp + 12]      ; first argument
pop  ebp
ret  8
```

Difference ENDP

Passing 8-bit and 16-bit Arguments

- Cannot push 8-bit values on stack
- Pushing 16-bit operand may cause page fault or ESP alignment problem
 - incompatible with Windows API functions
- Expand smaller arguments into 32-bit values, using MOVZX or MOVSX:

```
.data  
charVal BYTE 'x'  
.code  
    movzx eax,charVal  
    push  eax  
    call  Uppercase
```

Passing Multiword Arguments

- Push high-order values on the stack first; work backward in memory
- Results in little-endian ordering of data
- Example:

```
.data  
longVal DQ 1234567800ABCDEFh  
.code  
    push  DWORD PTR longVal + 4      ; high doubleword  
    push  DWORD PTR longVal          ; low doubleword  
    call   WriteHex64
```

Saving and Restoring Registers

- Push registers on stack just after assigning ESP to EBP
 - local registers are modified inside the procedure

```
MySub PROC  
    push ebp  
    mov ebp,esp  
    push ecx          ; save local registers  
    push edx
```

Stack Affected by USES Operator

```
MySub1 PROC USES ecx edx
    ret
MySub1 ENDP
```

- USES operator generates code to save and restore registers:

```
MySub1 PROC
    push  ecx
    push  edx

    pop   edx
    pop   ecx
    ret
```

Local Variables

- Only statements within subroutine can view or modify local variables
- Storage used by local variables is released when subroutine ends
- local variable name can have the same name as a local variable in another function without creating a name clash
- Essential when writing recursive procedures, as well as procedures executed by multiple execution threads

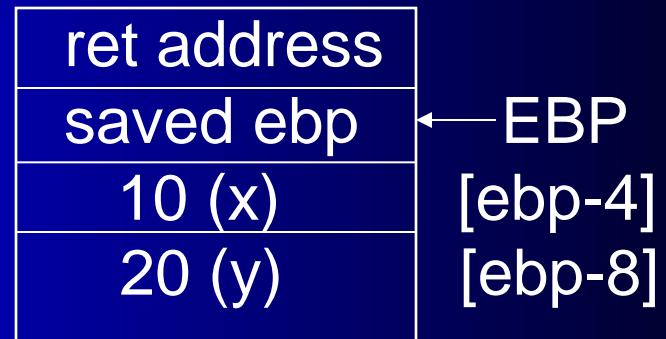
Creating LOCAL Variables

Example - create two DWORD local variables:

Say: int x=10, y=20;

```
MySub PROC
    push    ebp
    mov     ebp,esp
    sub    esp,8           ;create 2 DWORD variables

    mov     DWORD PTR [ebp-4],10 ; initialize x=10
    mov     DWORD PTR [ebp-8],20 ; initialize y=20
```



LEA Instruction

- LEA returns offsets of direct and indirect operands
 - OFFSET operator only returns constant offsets
- LEA required when obtaining offsets of stack parameters & local variables
- Example

```
CopyString PROC,  
    count:DWORD  
    LOCAL temp[20]:BYTE  
  
    mov edi,OFFSET count          ; invalid operand  
    mov esi,OFFSET temp          ; invalid operand  
    lea edi,count                ; ok  
    lea esi,temp                 ; ok
```

LEA Example

Suppose you have a Local variable at [ebp-8]

And you need the address of that local variable in ESI

You cannot use this:

```
mov esi, OFFSET [ebp-8] ; error
```

Use this instead:

```
lea esi,[ebp-8]
```

ENTER Instruction

- ENTER instruction creates stack frame for a called procedure
 - pushes EBP on the stack
 - sets EBP to the base of the stack frame
 - reserves space for local variables
 - Example:

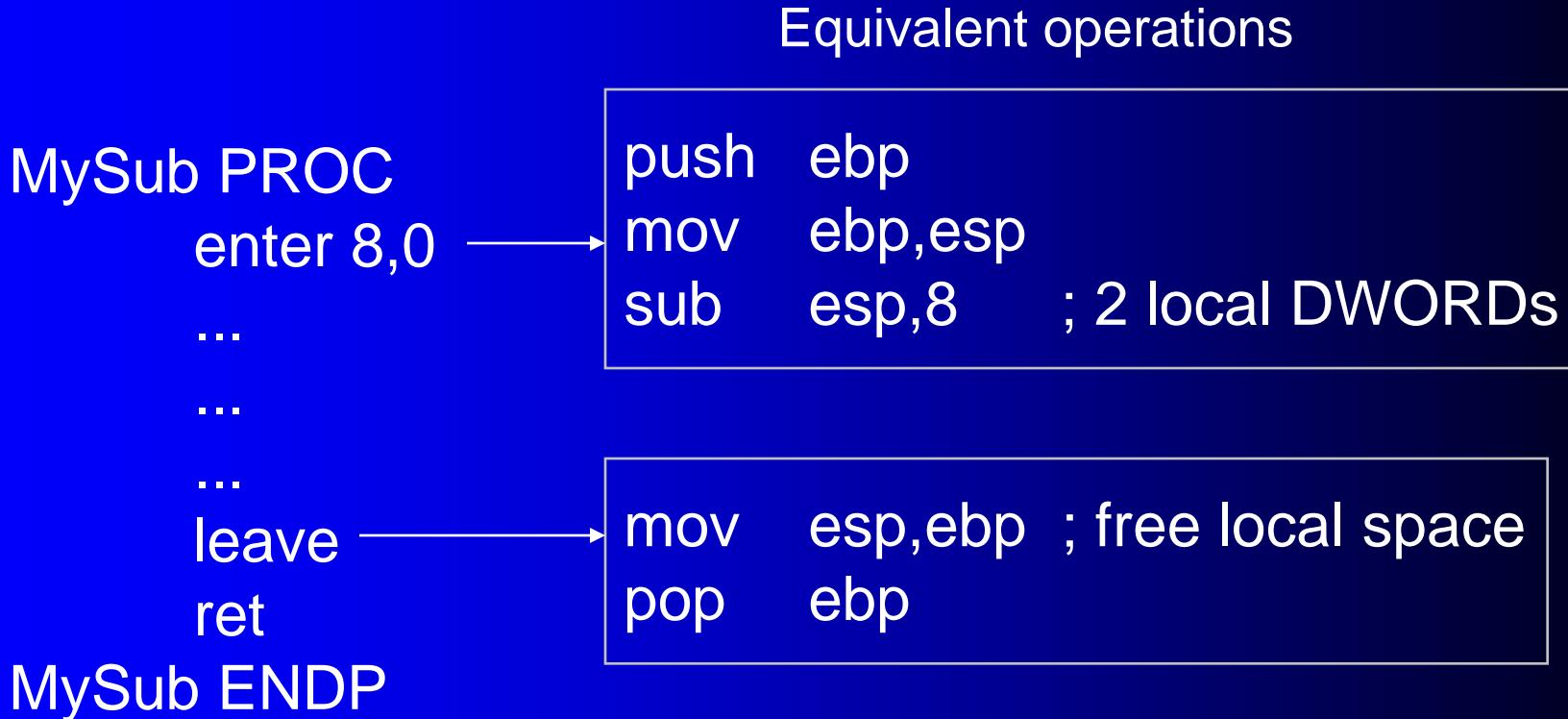
```
MySub PROC  
    enter 8,0
```

- Equivalent to:

```
MySub PROC  
    push ebp  
    mov ebp,esp  
    sub esp,8
```

LEAVE Instruction

Terminates the stack frame for a procedure.



LOCAL Directive

- The LOCAL directive declares a list of local variables
 - immediately follows the PROC directive
 - each variable is assigned a type
- Syntax:
LOCAL varlist

Example:

```
MySub PROC  
    LOCAL var1:BYTE, var2:WORD, var3:SDWORD
```

Using LOCAL

Examples:

```
LOCAL flagVals[20]:BYTE      ; array of bytes  
  
LOCAL pArray:PTR WORD        ; pointer to an array  
  
myProc PROC,  
    LOCAL t1:BYTE,           ; procedure  
                           ; local variables
```

LOCAL Example (1 of 2)

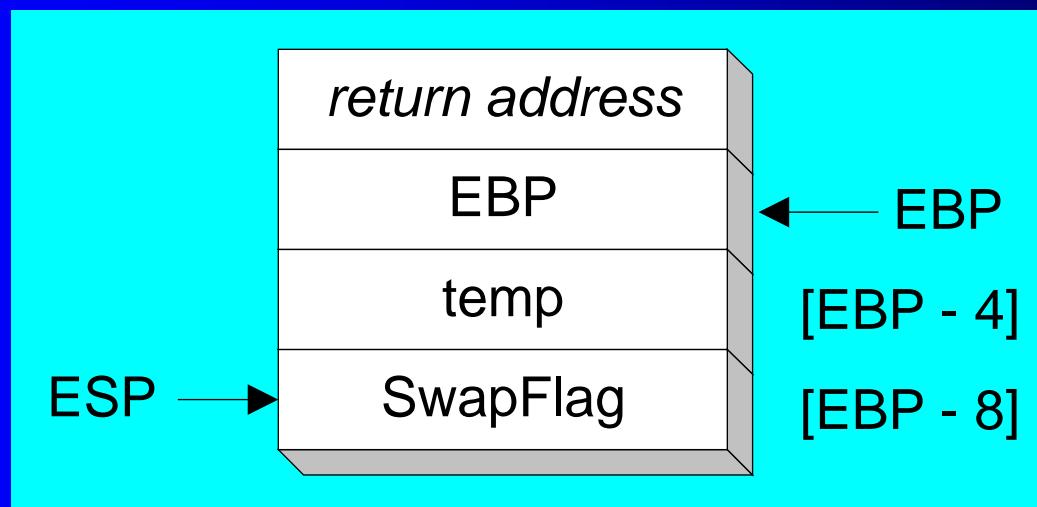
```
BubbleSort PROC  
    LOCAL temp:DWORD, SwapFlag:BYTE  
    . . .  
    ret  
BubbleSort ENDP
```

MASM generates the following code:

```
BubbleSort PROC  
    push ebp  
    mov  ebp,esp  
    add  esp,0FFFFFFF8h          ; add -8 to ESP  
    . . .  
    mov  esp,ebp  
    pop  ebp  
    ret  
BubbleSort ENDP
```

LOCAL Example (2 of 2)

Diagram of the stack frame for the BubbleSort procedure:



Non-Doubleword Local Variables

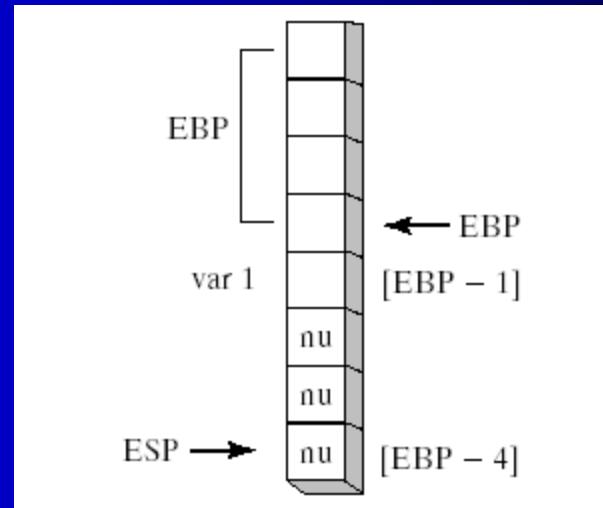
- Local variables can be different sizes
- How created in the stack by LOCAL directive:
 - 8-bit: assigned to next available byte
 - 16-bit: assigned to next even (word) boundary
 - 32-bit: assigned to next doubleword boundary

Local Byte Variable

Example1 PROC

```
LOCAL var1:BYTE  
mov al,var1           ; [EBP - 1]  
ret
```

Example1 ENDP



WriteStackFrame Procedure

- Displays contents of current stack frame
 - Prototype:

```
WriteStackFrame PROTO,  
    numParam:DWORD,      ; number of passed parameters  
    numLocalVal: DWORD, ; number of DWordLocal variables  
    numSavedReg: DWORD  ; number of saved registers
```

WriteStackFrame Example

```
main PROC
    mov eax, 0EAEEAEAEAh
    mov ebx, 0EBEBEBEBh
    INVOKE aProc, 1111h, 2222h
    exit
main ENDP

aProc PROC USES eax ebx,
    x: DWORD, y: DWORD
    LOCAL a:DWORD, b:DWORD
    PARAMS = 2
    LOCALS = 2
    SAVED_REGS = 2
    mov a, 0AAAAh
    mov b, 0BBBBh
    INVOKE WriteStackFrame, PARAMS, LOCALS, SAVED_REGS
```

The Microsoft x64 Calling Convention

- CALL subtracts 8 from RSP
- First four parameters are placed in RCX, RDX, R8, and R9. Additional parameters are pushed on the stack.
- Parameters less than 64 bits long are not zero extended
- Return value in RAX if \leq 64 bits
- Caller must allocate at least 32 bytes of shadow space so the subroutine can copy parameter values

The Microsoft x64 Calling Convention

- Caller must align RSP to 16-byte boundary
- Caller must remove all parameters from the stack after the call
- Return value larger than 64 bits must be placed on the runtime stack, with RCX pointing to it
- RBX, RBP, RDI, RSI, R12, R14, R14, and R15 registers are preserved by the subroutine; all others are not.

What's Next

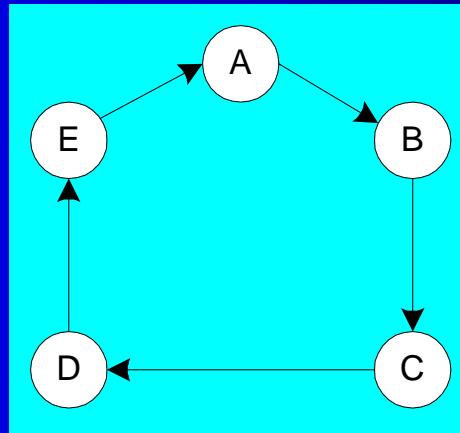
- Stack Frames
- **Recursion**
- INVOKE, ADDR, PROC, and PROTO
- Creating Multimodule Programs
- Advanced Use of Parameters (optional)
- Java Bytecodes (optional)

Recursion

- What is Recursion?
- Recursively Calculating a Sum
- Calculating a Factorial

What is Recursion?

- The process created when . . .
 - A procedure calls itself
 - Procedure A calls procedure B, which in turn calls procedure A
- Using a graph in which each node is a procedure and each edge is a procedure call, recursion forms a cycle:



Recursively Calculating a Sum

The CalcSum procedure recursively calculates the sum of an array of integers. Receives: ECX = count. Returns: EAX = sum

```
CalcSum PROC
    cmp ecx,0
    jz L2
    add eax,ecx
    dec ecx
    call CalcSum
    ; check counter value
    ; quit if zero
    ; otherwise, add to sum
    ; decrement counter
    ; recursive call
L2: ret
CalcSum ENDP
```

Stack frame:

Pushed On Stack	ECX	EAX
L1	5	0
L2	4	5
L2	3	9
L2	2	12
L2	1	14
L2	0	15

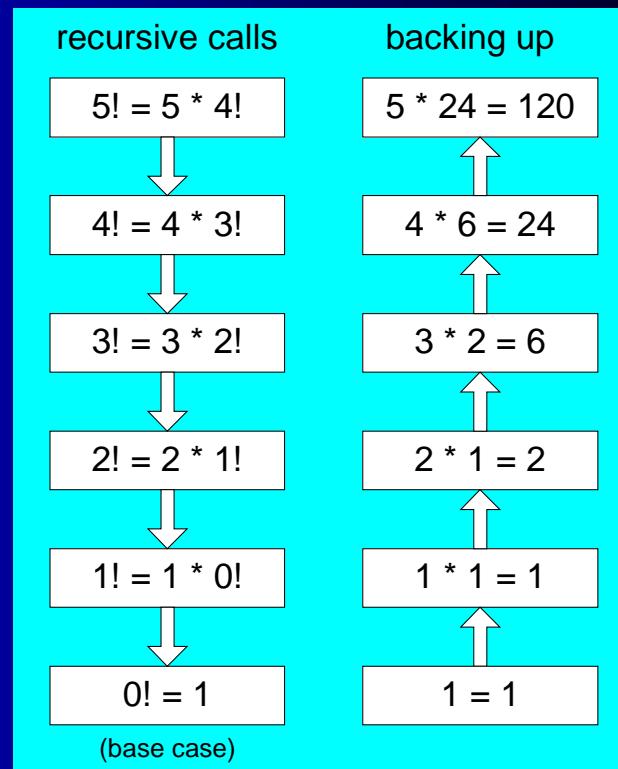
[View the complete program](#)

Calculating a Factorial (1 of 3)

This function calculates the factorial of integer n . A new value of n is saved in each stack frame:

```
int function factorial(int n)
{
    if(n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```

As each call instance returns, the product it returns is multiplied by the previous value of n .



Calculating a Factorial (2 of 3)

```
Factorial PROC
    push ebp
    mov  ebp,esp
    mov  eax,[ebp+8]           ; get n
    cmp  eax,0                ; n < 0?
    ja   L1                   ; yes: continue
    mov  eax,1                ; no: return 1
    jmp  L2

L1: dec  eax
    push eax                 ; Factorial(n-1)
    call Factorial

; Instructions from this point on execute when each
; recursive call returns.

ReturnFact:
    mov  ebx,[ebp+8]           ; get n
    mul  ebx                  ; eax = eax * ebx

L2: pop  ebp
    ret  4                   ; return EAX
                                ; clean up stack
Factorial ENDP
```

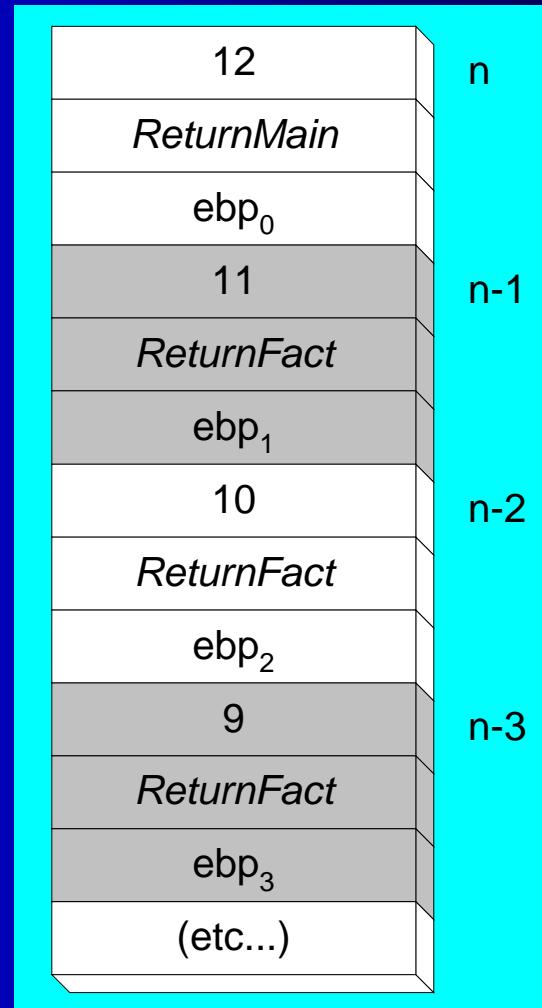
See the [program listing](#)

Calculating a Factorial (3 of 3)

Suppose we want to calculate 12!

This diagram shows the first few stack frames created by recursive calls to Factorial

Each recursive call uses
12 bytes of stack space.



What's Next

- Stack Frames
- Recursion
- **INVOKE, ADDR, PROC, and PROTO**
- Creating Multimodule Programs
- Java Bytecodes

INVOKE, ADDR, PROC, and PROTO

- INVOKE Directive
- ADDR Operator
- PROC Directive
- PROTO Directive
- Parameter Classifications
- Example: Exchaning Two Integers
- Debugging Tips

Not in 64-bit mode!

INVOKE Directive

- In 32-bit mode, the INVOKE directive is a powerful replacement for Intel's CALL instruction that lets you pass multiple arguments
- Syntax:
`INVOKE procedureName [, argumentList]`
- *ArgumentList* is an optional comma-delimited list of procedure arguments
- Arguments can be:
 - immediate values and integer expressions
 - variable names
 - address and ADDR expressions
 - register names

Invoke Examples

```
.data  
byteVal BYTE 10  
wordVal WORD 1000h  
.code  
; direct operands:  
Invoke Sub1,byteVal,wordVal  
  
; address of variable:  
Invoke Sub2,ADDR byteVal  
  
; register name, integer expression:  
Invoke Sub3,eax,(10 * 20)  
  
; address expression (indirect operand):  
Invoke Sub4,[ebx]
```

Not in 64-bit mode!

ADDR Operator

- Returns a near or far pointer to a variable, depending on which memory model your program uses:
 - Small model: returns 16-bit offset
 - Large model: returns 32-bit segment/offset
 - Flat model: returns 32-bit offset
- Simple example:

```
.data  
myWord WORD ?  
.code  
INVOKE mySub,ADDR myWord
```

Not in 64-bit mode!

PROC Directive (1 of 2)

- The PROC directive declares a procedure with an optional list of named parameters.
- Syntax:

label PROC paramList

- *paramList* is a list of parameters separated by commas. Each parameter has the following syntax:

paramName : *type*

type must either be one of the standard ASM types (BYTE, SBYTE, WORD, etc.), or it can be a pointer to one of these types.

PROC Directive (2 of 2)

- Alternate format permits parameter list to be on one or more separate lines:

label PROC, ← comma required
paramList

- The parameters can be on the same line . . .

param-1:type-1, param-2:type-2, . . . , param-n:type-n

- Or they can be on separate lines:

*param-1:type-1,
param-2:type-2,
. . . ,
param-n:type-n*

AddTwo Procedure (1 of 2)

- The AddTwo procedure receives two integers and returns their sum in EAX.

```
AddTwo PROC,  
    val1:DWORD, val2:DWORD  
  
    mov eax,val1  
    add eax,val2  
  
    ret  
AddTwo ENDP
```

PROC Examples (2 of 3)

FillArray receives a pointer to an array of bytes, a single byte fill value that will be copied to each element of the array, and the size of the array.

```
FillArray PROC,  
    pArray:PTR BYTE, fillVal:BYTE  
    arraySize:DWORD  
  
    mov ecx, arraySize  
    mov esi, pArray  
    mov al, fillVal  
L1:   mov [esi], al  
    inc esi  
    loop L1  
    ret  
FillArray ENDP
```

PROC Examples (3 of 3)

```
Swap PROC,  
    pValX:PTR DWORD,  
    pValY:PTR DWORD  
    . . .  
Swap ENDP
```

```
ReadFile PROC,  
    pBuffer:PTR BYTE  
    LOCAL fileHandle:DWORD  
    . . .  
ReadFile ENDP
```

PROTO Directive

- Creates a procedure prototype
- Syntax:
 - *label* PROTO *paramList*
- Parameter list not permitted in 64-bit mode
- Every procedure called by the INVOKE directive must have a prototype
- A complete procedure definition can also serve as its own prototype

PROTO Directive

- Standard configuration: PROTO appears at top of the program listing, INVOKE appears in the code segment, and the procedure implementation occurs later in the program:

```
MySub PROTO          ; procedure prototype  
  
.code  
INVOKE MySub          ; procedure call  
  
  
MySub PROC           ; procedure implementation  
.  
.  
.  
MySub ENDP
```

PROTO Example

- Prototype for the ArraySum procedure, showing its parameter list:

```
ArraySum PROTO,  
    ptrArray:PTR DWORD,      ; points to the array  
    szArray:DWORD           ; array size
```

Parameters are not permitted in 64-bit mode.

Parameter Classifications

- An **input parameter** is data passed by a calling program to a procedure.
 - The called procedure is not expected to modify the corresponding parameter variable, and even if it does, the modification is confined to the procedure itself.
- An **output parameter** is created by passing a pointer to a variable when a procedure is called.
 - The procedure does not use any existing data from the variable, but it fills in a new value before it returns.
- An **input-output parameter** is a pointer to a variable containing input that will be both used and modified by the procedure.
 - The variable passed by the calling program is modified.

Trouble-Shooting Tips

- Save and restore registers when they are modified by a procedure.
 - Except a register that returns a function result
- When using INVOKE, be careful to pass a pointer to the correct data type.
 - For example, MASM cannot distinguish between a DWORD argument and a PTR BYTE argument.
- Do not pass an immediate value to a procedure that expects a reference parameter.
 - Dereferencing its address will likely cause a general-protection fault.

What's Next

- Stack Frames
- Recursion
- INVOKE, ADDR, PROC, and PROTO
- **Creating Multimodule Programs**
- Advanced Use of Parameters (optional)
- Java Bytecodes (optional)

Multimodule Programs

- A **multimodule program** is a program whose source code has been divided up into separate ASM files.
- Each ASM file (**module**) is assembled into a separate OBJ file.
- All OBJ files belonging to the same program are linked using the **link** utility into a single EXE file.
 - This process is called **static linking**

Advantages

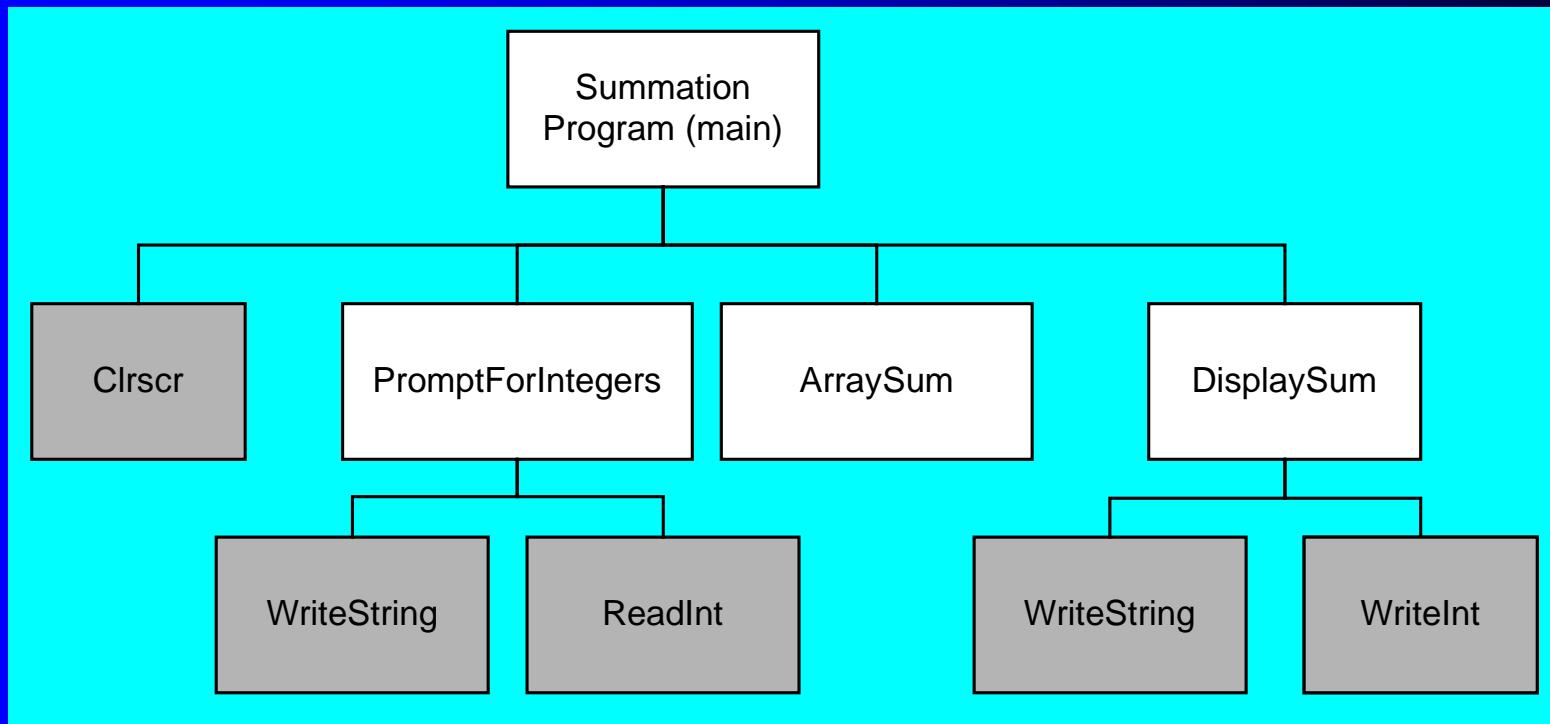
- Large programs are easier to write, maintain, and debug when divided into separate source code modules.
- When changing a line of code, only its enclosing module needs to be assembled again. Linking assembled modules requires little time.
- A module can be a container for logically related code and data (think object-oriented here...)
 - **encapsulation:** procedures and variables are automatically hidden in a module unless you declare them public

Creating a Multimodule Program

- Here are some basic steps to follow when creating a multimodule program:
 - Create the main module
 - Create a separate source code module for each procedure or set of related procedures
 - Create an include file that contains procedure prototypes for **external procedures** (ones that are called between modules)
 - Use the INCLUDE directive to make your procedure prototypes available to each module

Example: ArraySum Program

- Let's review the ArraySum program from Chapter 5.



Each of the four white rectangles will become a module. This will be a 32-bit application.

Sample Program output

```
Enter a signed integer: -25  
Enter a signed integer: 36  
Enter a signed integer: 42  
The sum of the integers is: +53
```

INCLUDE File

The `sum.inc` file contains prototypes for external functions that are not in the `Irvine32` library:

```
INCLUDE Irvine32.inc

PromptForIntegers PROTO,
    ptrPrompt:PTR BYTE,           ; prompt string
    ptrArray:PTR DWORD,          ; points to the array
    arraySize:DWORD               ; size of the array

ArraySum PROTO,
    ptrArray:PTR DWORD,          ; points to the array
    count:DWORD                  ; size of the array

DisplaySum PROTO,
    ptrPrompt:PTR BYTE,           ; prompt string
    theSum:DWORD                  ; sum of the array
```

Inspect Individual Modules

- Main
- PromptForIntegers
- ArraySum
- DisplaySum

What's Next

- Stack Frames
- Recursion
- INVOKE, ADDR, PROC, and PROTO
- Creating Multimodule Programs
- Advanced Use of Parameters (optional)
- **Java Bytecodes (optional)**

Java Bytecodes

- Stack-oriented instruction format
 - operands are on the stack
 - instructions pop the operands, process, and push result back on stack
- Each operation is atomic
- Might be translated into native code by a *just in time* compiler

Java Virtual Machine (JVM)

- Essential part of the Java Platform
- Executes compiled bytecodes
 - machine language of compiled Java programs

Java Methods

- Each method has its own stack frame
- Areas of the stack frame:
 - local variables
 - operands
 - execution environment

Bytecode Instruction Format

- 1-byte opcode
 - iload, istore, imul, goto, etc.
- zero or more operands
- Disassembling Bytecodes
 - use javap.exe, in the Java Development Kit (JDK)

Primitive Data Types

- Signed integers are in two's complement format, stored in big-endian order

Data Type	Bytes	Format
char	2	Unicode character
byte	1	signed integer
short	2	signed integer
int	4	signed integer
long	8	signed integer
float	4	IEEE single-precision real
double	8	IEEE double-precision real

JVM Instruction Set

- Comparison Instructions pop two operands off the stack, compare them, and push the result of the comparison back on the stack
- Examples: fcmp and dcmp

Results of Comparing <i>op1</i> and <i>op2</i>	Value Pushed on the Operand Stack
$op1 > op2$	1
$op1 = op2$	0
$op1 < op2$	-1

JVM Instruction Set

- Conditional Branching
 - jump to label if st(0) <= 0
`ifle label`
- Unconditional Branching
 - call subroutine
`jsr label`

Java Disassembly Examples

- Adding Two Integers

```
int A = 3;  
int B = 2;  
int sum = 0;  
sum = A + B;
```

```
0:  icanst_3  
1:  istore_0  
2:  icanst_2  
3:  istore_1  
4:  icanst_0  
5:  istore_2  
6:  iload_0  
7:  iload_1  
8:  iadd  
9:  istore_2
```

Java Disassembly Examples

- Adding Two Doubles

```
double A = 3.1;  
double B = 2;  
double sum = A + B;
```

```
0: ldc2_w #20;           // double 3.1d  
3: dstore_0  
4: ldc2_w #22;           // double 2.0d  
7: dstore_2  
8: dload_0  
9: dload_2  
10: dadd  
11: dstore_4
```

Java Disassembly Examples

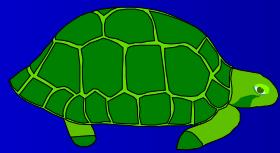
- Conditional Branch

```
double A = 3.0;
boolean result = false;
if( A > 2.0 )
    result = false;
else
    result = true;
```

```
0: ldc2_w #26;           // double 3.0d
3: dstore_0              // pop into A
4: iconst_0               // false = 0
5: istore_2               // store in result
6: dload_0
7: ldc2_w #22;           // double 2.0d
10: dcmpl
11: ifle 19               // if A <= 2.0, goto 19
14: iconst_0               // false
15: istore_2               // result = false
16: goto 21               // skip next two statements
19: iconst_1               // true
20: istore_2               // result = true
```

Summary

- Stack parameters
 - more convenient than register parameters
 - passed by value or reference
 - ENTER and LEAVE instructions
- Local variables
 - created on the stack below stack pointer
 - LOCAL directive
- Recursive procedure calls itself
- Calling conventions (C, stdcall)
- MASM procedure-related directives
 - INVOKE, PROC, PROTO
- Java Bytecodes – another approach to programming



53 68 75 72 79 6F

Assembly Language for x86 Processors

7th Edition

Kip R. Irvine

Chapter 9: Strings and Arrays

Slide show prepared by the author

Revision date: 1/15/2014

Chapter Overview

- **String Primitive Instructions**
- Selected String Procedures
- Two-Dimensional Arrays
- Searching and Sorting Integer Arrays
- Java Bytecodes: String Processing
(optional topic)

String Primitive Instructions

- MOVSB, MOVSW, and MOVSD
- CMPSB, CMPSW, and CMPSD
- SCASB, SCASW, and SCASD
- STOSB, STOSW, and STOSD
- LODSB, LODSW, and LODSD

MOVSB, MOVSW, and MOVSD (1 of 2)

- The MOVSB, MOVSW, and MOVSD instructions copy data from the memory location pointed to by ESI to the memory location pointed to by EDI.

```
.data
source DWORD 0FFFFFFFh
target DWORD ?
.code
mov esi,OFFSET source
mov edi,OFFSET target
movsd
```

MOVSB, MOVSW, and MOVSD (2 of 2)

- ESI and EDI are automatically incremented or decremented:
 - MOVSB increments/decrements by 1
 - MOVSW increments/decrements by 2
 - MOVSD increments/decrements by 4

Direction Flag

- The Direction flag controls the incrementing or decrementing of ESI and EDI.
 - DF = clear (0): increment ESI and EDI
 - DF = set (1): decrement ESI and EDI

The Direction flag can be explicitly changed using the CLD and STD instructions:

CLD ; clear Direction flag

STD ; set Direction flag

Using a Repeat Prefix

- REP (a repeat prefix) can be inserted just before MOVSB, MOVSW, or MOVSD.
- ECX controls the number of repetitions
- Example: Copy 20 doublewords from source to target

```
.data
source DWORD 20 DUP(?)
target DWORD 20 DUP(?)
.code
cld                      ; direction = forward
mov ecx,LENGTHOF source   ; set REP counter
mov esi,OFFSET source
mov edi,OFFSET target
rep movsd
```

Your turn . . .

- Use MOVSD to delete the first element of the following doubleword array. All subsequent array values must be moved one position forward toward the beginning of the array:

```
array DWORD 1,1,2,3,4,5,6,7,8,9,10
```

```
.data
array DWORD 1,1,2,3,4,5,6,7,8,9,10
.code
cld
mov ecx,(LENGTHOF array) - 1
mov esi,OFFSET array+4
mov edi,OFFSET array
rep movsd
```

CMPSB, CMPSW, and CMPSD

- The CMPSB, CMPSW, and CMPSD instructions each compare a memory operand pointed to by ESI to a memory operand pointed to by EDI.
 - CMPSB compares bytes
 - CMPSW compares words
 - CMPSD compares doublewords
- Repeat prefix often used
 - REPE (REPZ)
 - REPNE (REPNZ)

Comparing a Pair of Doublewords

If source > target, the code jumps to label L1;
otherwise, it jumps to label L2

```
.data
source DWORD 1234h
target DWORD 5678h

.code
mov esi,OFFSET source
mov edi,OFFSET target
cmpsd          ; compare doublewords
ja L1          ; jump if source > target
jmp L2          ; jump if source <= target
```

Your turn . . .

- Modify the program in the previous slide by declaring both source and target as WORD variables. Make any other necessary changes.

Comparing Arrays

Use a REPE (repeat while equal) prefix to compare corresponding elements of two arrays.

```
.data
source DWORD COUNT DUP(?)
target DWORD COUNT DUP(?)
.code
mov ecx,COUNT           ; repetition count
mov esi,OFFSET source
mov edi,OFFSET target
cld                      ; direction = forward
repe cmpsd               ; repeat while equal
```

Example: Comparing Two Strings (1 of 3)

This program compares two strings (source and destination). It displays a message indicating whether the lexical value of the source string is less than the destination string.

```
.data
source BYTE "MARTIN    "
dest    BYTE "MARTINEZ"
str1  BYTE "Source is smaller",0dh,0ah,0
str2  BYTE "Source is not smaller",0dh,0ah,0
```

Screen
output:

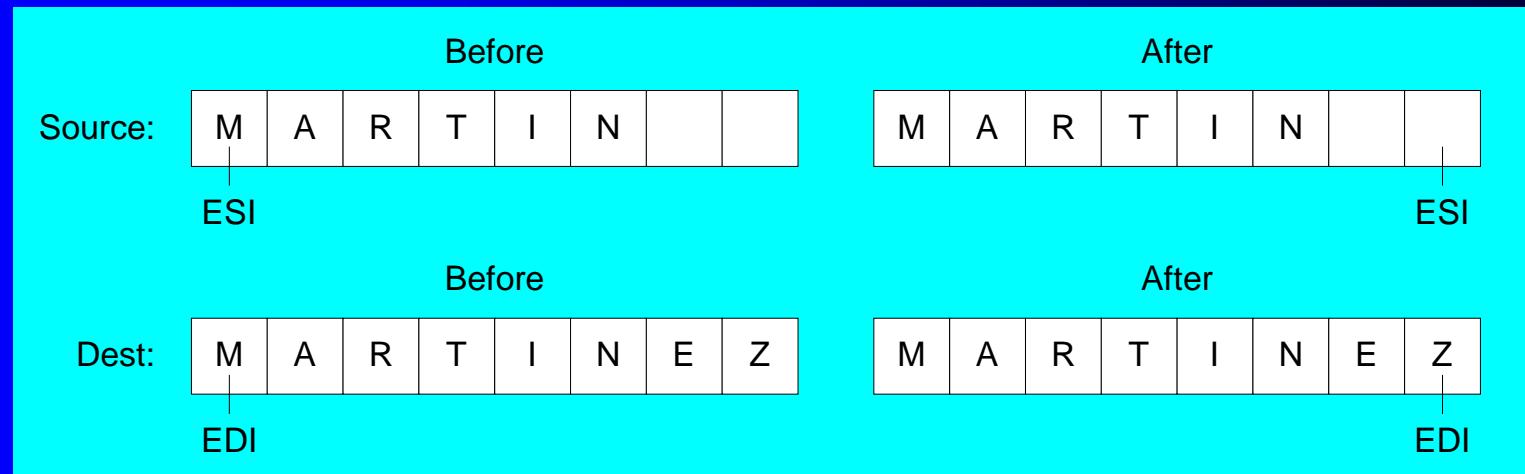
Source is smaller

Example: Comparing Two Strings (2 of 3)

```
.code
main PROC
    cld                      ; direction = forward
    mov  esi,OFFSET source
    mov  edi,OFFSET dest
    mov  ecx,LENGTHOF source
    repe cmpsb
    jb   source_smaller
    mov  edx,OFFSET str2      ; "source is not smaller"
    jmp  done
source_smaller:
    mov  edx,OFFSET str1      ; "source is smaller"
done:
    call WriteString
    exit
main ENDP
END main
```

Example: Comparing Two Strings (3 of 3)

- The following diagram shows the final values of ESI and EDI after comparing the strings:



SCASB, SCASW, and SCASD

- The SCASB, SCASW, and SCASD instructions compare a value in AL/AX/EAX to a byte, word, or doubleword, respectively, addressed by EDI.
- Useful types of searches:
 - Search for a specific element in a long string or array.
 - Search for the first element that does not match a given value.

SCASB Example

Search for the letter 'F' in a string named alpha:

```
.data  
alpha BYTE "ABCDEFGHI",0  
.code  
mov edi,OFFSET alpha  
mov al,'F' ; search for 'F'  
mov ecx,LENGTHOF alpha  
cld  
repne scasb ; repeat while not equal  
jnz quit  
dec edi ; EDI points to 'F'
```

What is the purpose of the JNZ instruction?

STOSB, STOSW, and STOSD

- The STOSB, STOSW, and STOSD instructions store the contents of AL/AX/EAX, respectively, in memory at the offset pointed to by EDI.
- Example: fill an array with 0FFh

```
.data  
Count = 100  
string1 BYTE Count DUP(?)  
.code  
mov al,0FFh ; value to be stored  
mov edi,OFFSET string1 ; ES:DI points to target  
mov ecx,Count ; character count  
cld ; direction = forward  
rep stosb ; fill with contents of AL
```

LODSB, LODSW, and LODSD

- LODSB, LODSW, and LODSD load a byte or word from memory at ESI into AL/AX/EAX, respectively.
- Example:

```
.data  
array BYTE 1,2,3,4,5,6,7,8,9  
.code  
    mov esi,OFFSET array  
    mov ecx,LENGTHOF array  
    cld  
L1:   lodsb          ; load byte into AL  
          or al,30h      ; convert to ASCII  
          call WriteChar ; display it  
    loop L1
```

Array Multiplication Example

Multiply each element of a doubleword array by a constant value.

```
.data
array DWORD 1,2,3,4,5,6,7,8,9,10
multiplier DWORD 10
.code
    cld                      ; direction = up
    mov esi,OFFSET array     ; source index
    mov edi,esi               ; destination index
    mov ecx,LENGTHOF array   ; loop counter

L1: lodsd                  ; copy [ESI] into EAX
    mul multiplier           ; multiply by a value
    stosd                   ; store EAX at [EDI]
    loop L1
```

Your turn . . .

- Write a program that converts each unpacked binary-coded decimal byte belonging to an array into an ASCII decimal byte and copies it to a new array.

```
.data  
array BYTE 1,2,3,4,5,6,7,8,9  
dest BYTE (LENGTHOF array) DUP(?)
```

```
        mov esi,OFFSET array  
        mov edi,OFFSET dest  
        mov ecx,LENGTHOF array  
        cld  
L1: lodsb          ; load into AL  
        or al,30h         ; convert to ASCII  
        stosb           ; store into memory  
        loop L1
```

What's Next

- String Primitive Instructions
- **Selected String Procedures**
- Two-Dimensional Arrays
- Searching and Sorting Integer Arrays

Selected 32-Bit String Procedures

The following string procedures may be found in the Irvine32 library

- Str_compare Procedure
- Str_length Procedure
- Str_copy Procedure
- Str_trim Procedure
- Str_ucase Procedure

Str_compare Procedure

- Compares *string1* to *string2*, setting the Carry and Zero flags accordingly
- Prototype:

```
Str_compare PROTO,  
    string1:PTR BYTE,          ; pointer to string  
    string2:PTR BYTE          ; pointer to string
```

Relation	Carry Flag	Zero Flag	Branch if True
string1 < string2	1	0	JB
string1 == string2	0	1	JE
string1 > string2	0	0	JA

Str_compare Source Code

```
Str_compare PROC USES eax edx esi edi,  
    string1:PTR BYTE, string2:PTR BYTE  
  
    mov esi,string1  
    mov edi,string2  
  
L1:   mov al,[esi]  
    mov dl,[edi]  
    cmp al,0           ; end of string1?  
    jne L2             ; no  
    cmp dl,0           ; yes: end of string2?  
    jne L2             ; no  
    jmp L3             ; yes, exit with ZF = 1  
  
L2:   inc esi          ; point to next  
    inc edi  
    cmp al,dl          ; chars equal?  
    je L1              ; yes: continue loop  
  
L3:   ret  
Str_compare ENDP
```

Str_length Procedure

- Calculates the length of a null-terminated string and returns the length in the EAX register.
- Prototype:

```
Str_length PROTO,  
    pString:PTR BYTE           ; pointer to string
```

Example:

```
.data  
myString BYTE "abcdefg",0  
.code  
    INVOKE Str_length,  
        ADDR myString  
; EAX = 7
```

Str_length Source Code

```
Str_length PROC USES edi,  
    pString:PTR BYTE           ; pointer to string  
  
    mov edi,pString  
    mov eax,0                  ; character count  
L1:  
    cmp byte ptr [edi],0        ; end of string?  
    je L2                      ; yes: quit  
    inc edi                    ; no: point to next  
    inc eax                    ; add 1 to count  
    jmp L1  
L2: ret  
Str_length ENDP
```

Str_copy Procedure

- Copies a null-terminated string from a source location to a target location.
- Prototype:

```
Str_copy PROTO,  
    source:PTR BYTE,           ; pointer to string  
    target:PTR BYTE           ; pointer to string
```

See the [CopyStr.asm](#) program for a working example.

Str_copy Source Code

```
Str_copy PROC USES eax ecx esi edi,  
    source:PTR BYTE,           ; source string  
    target:PTR BYTE           ; target string  
  
    INVOKE Str_length,source  ; EAX = length source  
    mov ecx,eax              ; REP count  
    inc ecx                  ; add 1 for null byte  
    mov esi,source  
    mov edi,target  
    cld                      ; direction = up  
    rep movsb                ; copy the string  
    ret  
  
Str_copy ENDP
```

Str_trim Procedure

- The Str_trim procedure removes all occurrences of a selected trailing character from a null-terminated string.
- Prototype:

```
Str_trim PROTO,  
    pString:PTR BYTE,           ; points to string  
    char:BYTE                  ; char to remove
```

Example:

```
.data  
myString BYTE "Hello###",0  
.code  
    INVOKE Str_trim, ADDR myString, '#'  
  
myString = "Hello"
```

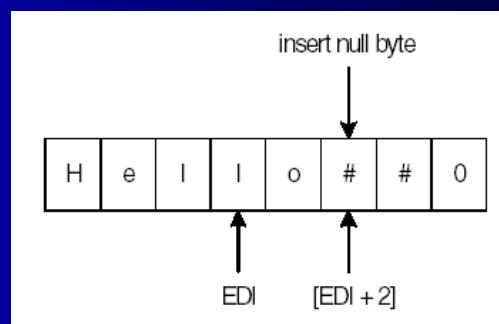
Str_trim Procedure

- Str_trim checks a number of possible cases (shown here with # as the trailing character):
 - The string is empty.
 - The string contains other characters followed by one or more trailing characters, as in "Hello##".
 - The string contains only one character, the trailing character, as in "#"
 - The string contains no trailing character, as in "Hello" or "H".
 - The string contains one or more trailing characters followed by one or more nontrailing characters, as in "#H" or "###Hello".

Testing the Str_trim Procedure

String Definition	EDI, When SCASB Stops	Zero Flag	ECX	Position to Store the Null
str BYTE "Hello##", 0	str + 3	0	> 0	[edi + 2]
str BYTE "#", 0	str - 1	1	0	[edi + 1]
str BYTE "Hello", 0	str + 3	0	> 0	[edi + 2]
str BYTE "H", 0	str - 1	0	0	[edi + 2]
str BYTE "#H", 0	str + 0	0	> 0	[edi + 2]

Using the first definition in the table, position of EDI when SCASB stops:



Str_trim Source Code

```
Str_trim PROC USES eax ecx edi,
    pString:PTR BYTE,           ; points to string
    char:BYTE                  ; char to remove
    mov  edi,pString
    INVOKE Str_length,edi      ; returns length in EAX
    cmp  eax,0                 ; zero-length string?
    je   L2                    ; yes: exit
    mov  ecx,eax               ; no: counter = string length
    dec  eax
    add  edi,eax               ; EDI points to last char
    mov  al,char                ; char to trim
    std
    repe scasb                 ; skip past trim character
    jne  L1                    ; removed first character?
    dec  edi                   ; adjust EDI: ZF=1 && ECX=0
    L1: mov  BYTE PTR [edi+2],0 ; insert null byte
    L2: ret
Str_trim ENDP
```

Str_ucase Procedure

- The Str_casecmp procedure converts a string to all uppercase characters. It returns no value.
- Prototype:

```
Str_casecmp PROTO,  
    pString:PTR BYTE           ; pointer to string
```

Example:

```
.data  
myString BYTE "Hello",0  
.code  
    INVOKE Str_casecmp,  
        ADDR myString
```

Str_ucase Source Code

```
Str_casecmp PROC USES eax esi,  
    pString:PTR BYTE  
    mov esi,pString  
  
L1: mov al,[esi]           ; get char  
    cmp al,0             ; end of string?  
    je L3               ; yes: quit  
    cmp al,'a'           ; below "a"?  
    jb L2               ; above "z"?  
    cmp al,'z'  
    ja L2               ; above "z"?  
    and BYTE PTR [esi],11011111b ; convert the char  
  
L2: inc esi              ; next char  
    jmp L1  
  
L3: ret  
Str_casecmp ENDP
```

String Procedures in the Irvine64 Library

- `Str_compare` – compares two strings pointed to by RSI and RDI. Sets the Carry and Zero flags in the same manner as the CMP instruction
- `Str_copy` – copies a source string to a location identified by a target pointer
- `Str_length` – returns the length of a null-terminated string

Example: 64-Bit Str_length

Gets the length of a string. Receives: RCX points to the string. Returns length of string in RAX.

```
Str_length PROC USES rdi
    mov    rdi,rcx           ; get pointer
    mov    eax,0              ; character counter
L1:
    cmp    BYTE PTR [rdi],0   ; end of string?
    je     L2                ; yes: quit
    inc    rdi               ; no: point to next
    inc    rax               ; add 1 to count
    jmp    L1
L2: ret                 ; return count in RAX
Str_length ENDP
```

What's Next

- String Primitive Instructions
- Selected String Procedures
- **Two-Dimensional Arrays**
- Searching and Sorting Integer Arrays

Two-Dimensional Arrays

- Base-Index Operands
- Base-Index Displacement

Base-Index Operand

- A **base-index** operand adds the values of two registers (called **base** and **index**), producing an **effective address**. Any two 32-bit general-purpose registers may be used. (*Note: esp is not a general-purpose register*)
 - In 64-bit mode, you use 64-bit registers for bases and indexes
- Base-index operands are great for accessing arrays of structures. (A structure groups together data under a single name.)

Structure Application

A common application of base-index addressing has to do with addressing arrays of structures (Chapter 10). The following defines a structure named COORD containing X and Y screen coordinates:

```
COORD STRUCT
    X WORD ?
                ; offset 00
    Y WORD ?
                ; offset 02
COORD ENDS
```

Then we can define an array of COORD objects:

```
.data
setOfCoordinates COORD 10 DUP(<>)
```

Structure Application

The following code loops through the array and displays each Y-coordinate:

```
    mov  ebx,OFFSET setOfCoordinates
    mov  esi,2          ; offset of Y value
    mov  eax,0
L1:   mov  ax,[ebx+esi]
        call WriteDec
        add  ebx,SIZEOF COORD
        loop L1
```

Base-Index-Displacement Operand

- A base-index-displacement operand adds base and index registers to a constant, producing an effective address. Any two 32-bit general-purpose register can be used.
- Common formats:

$[\text{base} + \text{index} + \text{displacement}]$

$\text{displacement} [\text{base} + \text{index}]$

64-bit Base-Index-Displacement Operand

- A 64-bit **base-index-displacement** operand adds base and index registers to a constant, producing a 64-bit **effective address**. Any two 64-bit general-purpose registers can be used.
- Common formats:

$[\text{base} + \text{index} + \text{displacement}]$

$\text{displacement} [\text{base} + \text{index}]$

Two-Dimensional Table Example

Imagine a table with three rows and five columns. The data can be arranged in any format on the page:

```
table  BYTE  10h,  20h,  30h,  40h,  50h
        BYTE  60h,  70h,  80h,  90h,  0A0h
        BYTE  0B0h, 0C0h, 0D0h, 0E0h, 0F0h
NumCols = 5
```

Alternative format:

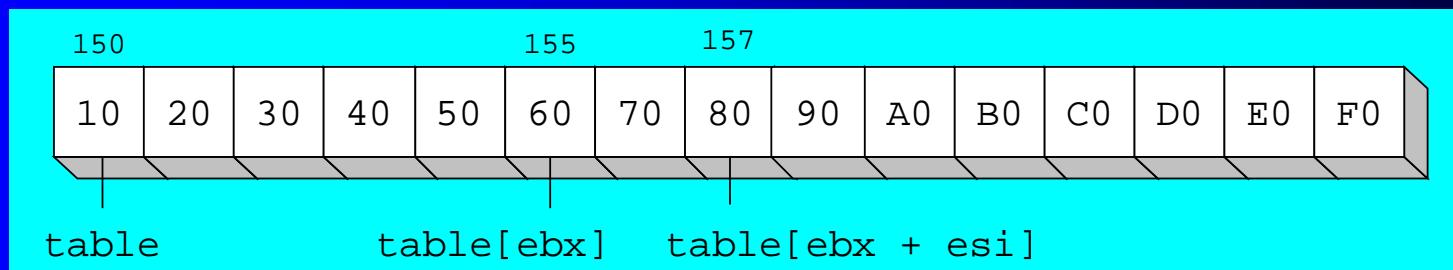
```
table  BYTE  10h,20h,30h,40h,50h,60h,70h,
        80h,90h,0A0h,
        0B0h,0C0h,0D0h,
        0E0h,0F0h
NumCols = 5
```

Two-Dimensional Table Example

The following 32-bit code loads the table element stored in row 1, column 2:

```
RowNumber = 1  
ColumnNumber = 2
```

```
mov ebx, NumCols * RowNumber  
mov esi, ColumnNumber  
mov al, table[ebx + esi]
```

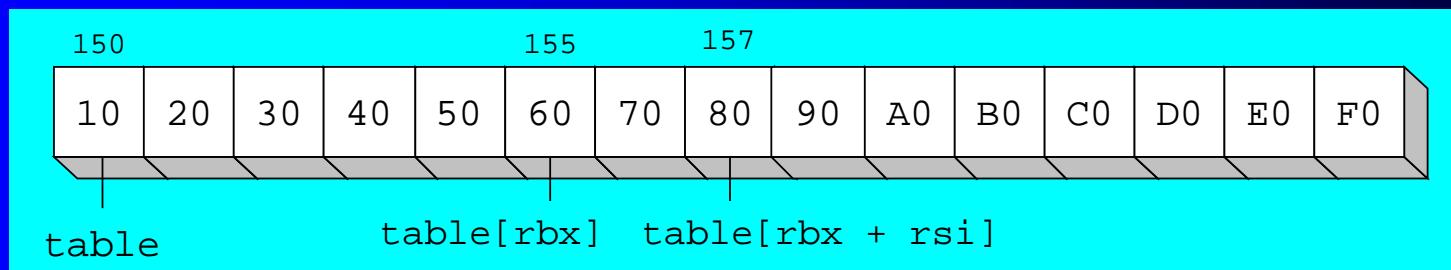


Two-Dimensional Table Example (64-bit)

The following 64-bit code loads the table element stored in row 1, column 2:

```
RowNumber = 1  
ColumnNumber = 2
```

```
mov rbx, NumCols * RowNumber  
mov rsi, ColumnNumber  
mov al, table[rbx + rsi]
```



What's Next

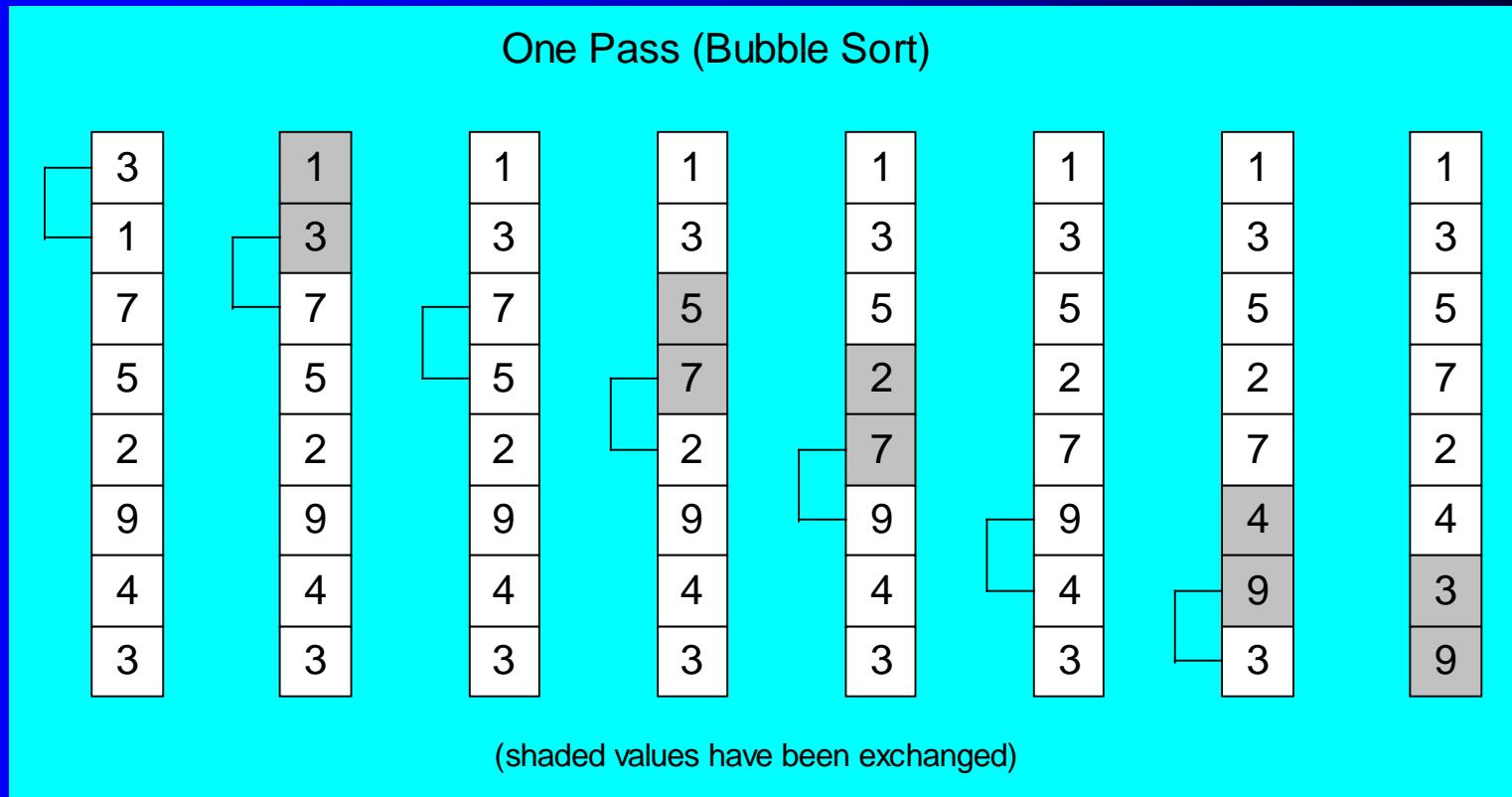
- String Primitive Instructions
- Selected String Procedures
- Two-Dimensional Arrays
- **Searching and Sorting Integer Arrays**

Searching and Sorting Integer Arrays

- Bubble Sort
 - A simple sorting algorithm that works well for small arrays
- Binary Search
 - A simple searching algorithm that works well for large arrays of values that have been placed in either ascending or descending order

Bubble Sort

Each pair of adjacent values is compared, and exchanged if the values are not ordered correctly:



Bubble Sort Pseudocode

N = array size, cx1 = outer loop counter, cx2 = inner loop counter:

```
cx1 = N - 1
while( cx1 > 0 )
{
    esi = addr(array)
    cx2 = cx1
    while( cx2 > 0 )
    {
        if( array[esi] < array[esi+4] )
            exchange( array[esi], array[esi+4] )
        add esi,4
        dec cx2
    }
    dec cx1
}
```

Bubble Sort Implementation

```
BubbleSort PROC USES eax ecx esi,  
    pArray:PTR DWORD,Count:DWORD  
    mov  ecx,Count  
    dec  ecx          ; decrement count by 1  
L1: push ecx          ; save outer loop count  
    mov  esi,pArray   ; point to first value  
L2: mov  eax,[esi]    ; get array value  
    cmp  [esi+4],eax  ; compare a pair of values  
    jge  L3            ; if [esi] <= [edi], skip  
    xchg eax,[esi+4]  ; else exchange the pair  
    mov  [esi],eax  
L3: add  esi,4        ; move both pointers forward  
    loop L2            ; inner loop  
    pop  ecx          ; retrieve outer loop count  
    loop L1            ; else repeat outer loop  
L4: ret  
BubbleSort ENDP
```

Binary Search

- Searching algorithm, well-suited to large ordered data sets
- Divide and conquer strategy
- Each "guess" divides the list in half
- Classified as an $O(\log n)$ algorithm:
 - As the number of array elements increases by a factor of n , the average search time increases by a factor of $\log n$.

Sample Binary Search Estimates

Array Size (n)	Maximum Number of Comparisons: $(\log_2 n) + 1$
64	7
1,024	11
65,536	17
1,048,576	21
4,294,967,296	33

Binary Search Pseudocode

```
int BinSearch( int values[],  
               const int searchVal, int count )  
{  
    int first = 0;  
    int last = count - 1;  
    while( first <= last )  
    {  
        int mid = (last + first) / 2;  
        if( values[mid] < searchVal )  
            first = mid + 1;  
        else if( values[mid] > searchVal )  
            last = mid - 1;  
        else  
            return mid;      // success  
    }  
    return -1;           // not found  
}
```

Binary Search Implementation (1 of 3)

```
BinarySearch PROC uses ebx edx esi edi,  
    pArray:PTR DWORD,           ; pointer to array  
    Count:DWORD,                ; array size  
    searchVal:DWORD             ; search value  
  
LOCAL first:DWORD,            ; first position  
      last:DWORD,              ; last position  
      mid:DWORD                ; midpoint  
    mov  first,0                ; first = 0  
    mov  eax,Count              ; last = (count - 1)  
    dec  eax  
    mov  last,eax  
    mov  edi,searchVal          ; EDI = searchVal  
    mov  ebx,pArray              ; EBX points to the array  
L1:                           ; while first <= last  
    mov  eax,first  
    cmp  eax,last  
    jg   L5                    ; exit search
```

Binary Search Implementation (2 of 3)

```
; mid = (last + first) / 2
    mov    eax,last
    add    eax,first
    shr    eax,1
    mov    mid,eax

; EDX = values[mid]
    mov    esi,mid
    shl    esi,2          ; scale mid value by 4
    mov    edx,[ebx+esi]   ; EDX = values[mid]

; if ( EDX < searchval(EDI) )
;     first = mid + 1;
    cmp    edx,edi
    jge    L2
    mov    eax,mid          ; first = mid + 1
    inc    eax
    mov    first,eax
    jmp    L4                ; continue the loop
```

base-index
addressing

Binary Search Implementation (3 of 3)

```
; else if( EDX > searchVal(EDI) )
;   last = mid - 1;
L2: cmp  edx,edi          ; (could be removed)
    jle  L3
    mov  eax,mid          ; last = mid - 1
    dec  eax
    mov  last,eax
    jmp  L4                ; continue the loop

; else return mid
L3: mov  eax,mid          ; value found
    jmp  L9                ; return (mid)

L4: jmp  L1                ; continue the loop
L5: mov  eax,-1           ; search failed
L9: ret
BinarySearch ENDP
```

Java Bytecodes: String Processing

- In Java, strings identifiers are references to other storage
 - think of a reference as a pointer, or address

Ways to load and store a string:

- **ldc** - loads a reference to a string literal from the constant pool
- **astore** - pops a string reference from the stack and stores it in a local variable

String Processing Example

- Java:

```
String empInfo = "10034Smith";
String id = empInfo.substring(0,5);
```

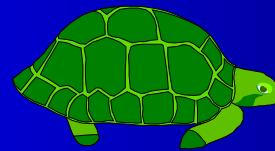
- **Bytecode disassembly:**

```
0: ldc #32;           // String 10034Smith
2: astore_0
3: aload_0
4: iconst_0
5: iconst_5
6: invokevirtual #34; // Method java/lang/String.substring
9: astore_1
```

invokevirtual calls a class method

Summary

- String primitives are optimized for efficiency
- Strings and arrays are essentially the same
- Keep code inside loops simple
- Use base-index operands with two-dimensional arrays
- Avoid the bubble sort for large arrays
- Use binary search for large sequentially ordered arrays



45 6E 64 65

Assembly Language for x86 Processors

7th Edition

Kip R. Irvine

Chapter 10: Structures and Macros

Slide show prepared by the author

Revision date: 1/15/2014

Chapter Overview

- **Structures**
- Macros
- Conditional-Assembly Directives
- Defining Repeat Blocks

Structures - Overview

- Defining Structures
- Declaring Structure Variables
- Referencing Structure Variables
- Example: Displaying the System Time
- Nested Structures
- Example: Drunkard's Walk
- Declaring and Using Unions

Structure

- A template or pattern given to a logically related group of variables.
- **field** - structure member containing data
- Program access to a structure:
 - entire structure as a complete unit
 - individual fields
- Useful way to pass multiple related arguments to a procedure
 - example: file directory information

Using a Structure

Using a structure involves three sequential steps:

1. Define the structure.
2. Declare one or more variables of the structure type, called **structure variables**.
3. Write runtime instructions that access the structure.

Structure Definition Syntax

```
name STRUCT  
  field-declarations  
name ENDS
```

- Field-declarations are identical to variable declarations

COORD Structure

- The COORD structure used by the MS-Windows programming library identifies X and Y screen coordinates

```
COORD STRUCT
```

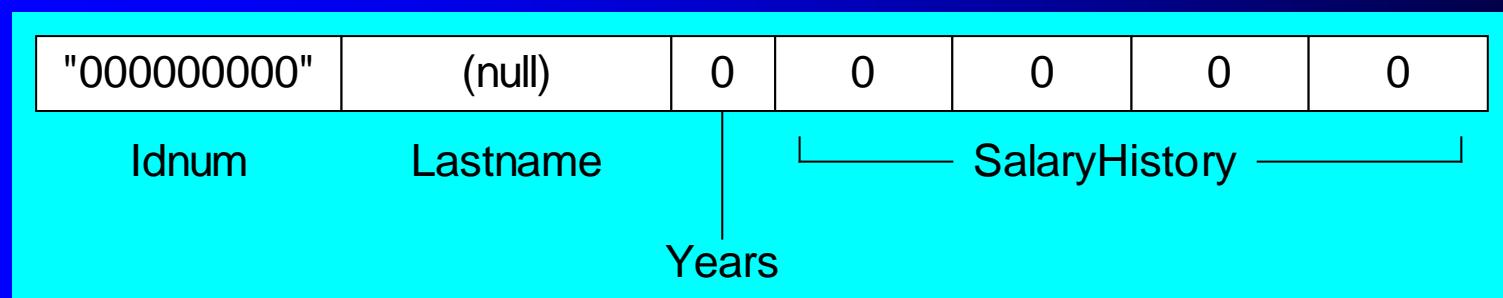
```
    X WORD ?           ; offset 00  
    Y WORD ?           ; offset 02
```

```
COORD ENDS
```

Employee Structure

A structure is ideal for combining fields of different types:

```
Employee STRUCT
    IdNum BYTE "000000000"
    LastName BYTE 30 DUP(0)
    Years WORD 0
    SalaryHistory DWORD 0,0,0,0
Employee ENDS
```



Declaring Structure Variables

- Structure name is a user-defined type
- Insert replacement initializers between brackets:

< . . . >

- Empty brackets <> retain the structure's default field initializers
- Examples:

```
.data  
point1 COORD <5,10>  
point2 COORD <>  
worker Employee <>
```

Initializing Array Fields

- Use the DUP operator to initialize one or more elements of an array field:

```
.data  
emp Employee <,,,2 DUP(20000)>
```

Array of Structures

- An array of structure objects can be defined using the DUP operator.
- Initializers can be used

```
NumPoints = 3
AllPoints COORD NumPoints DUP(<0,0>)

RD_Dept Employee 20 DUP(<>)

accounting Employee 10 DUP(<,,4 DUP(20000) >)
```

Referencing Structure Variables

```
Employee STRUCT ; bytes
    IdNum BYTE "00000000" ; 9
    LastName BYTE 30 DUP(0) ; 30
    Years WORD 0 ; 2
    SalaryHistory DWORD 0,0,0,0 ; 16
Employee ENDS ; 57

.data
worker Employee <>

mov eax,TYPE Employee ; 57
mov eax,SIZEOF Employee ; 57
mov eax,SIZEOF worker ; 57
mov eax,TYPE Employee.SalaryHistory ; 4
mov eax,LENGTHOF Employee.SalaryHistory ; 4
mov eax,SIZEOF Employee.SalaryHistory ; 16
```



. . . continued

```
mov dx,worker.Years
mov worker.SalaryHistory,20000          ; first salary
mov worker.SalaryHistory+4,30000        ; second salary
mov edx,OFFSET worker.LastName

mov esi,OFFSET worker
mov ax,(Employee PTR [esi]).Years

mov ax,[esi].Years      ; invalid operand (ambiguous)
```

Looping Through an Array of Points

Sets the X and Y coordinates of the AllPoints array to sequentially increasing values (1,1), (2,2), ...

```
.data
NumPoints = 3
AllPoints COORD NumPoints DUP(<0,0>)

.code
    mov edi,0           ; array index
    mov ecx,NumPoints   ; loop counter
    mov ax,1             ; starting X, Y values
L1:
    mov (COORD PTR AllPoints[edi]).X,ax
    mov (COORD PTR AllPoints[edi]).Y,ax
    add edi,TYPE COORD
    inc ax
    Loop L1
```

Example: Displaying the System Time (1 of 3)

- Retrieves and displays the system time at a selected screen location.
- Uses COORD and SYSTEMTIME structures:

```
SYSTEMTIME STRUCT
    wYear          WORD  ?
    wMonth         WORD  ?
    wDayOfWeek    WORD  ?
    wDay           WORD  ?
    wHour          WORD  ?
    wMinute        WORD  ?
    wSecond        WORD  ?
    wMilliseconds WORD  ?
SYSTEMTIME ENDS
```

Example: Displaying the System Time (2 of 3)

- GetStdHandle gets the standard console output handle.
- SetConsoleCursorPosition positions the cursor.
- GetLocalTime gets the current time of day.

```
.data
sysTime SYSTEMTIME <>
XYPos COORD <10,5>
consoleHandle DWORD ?

.code
INVOKE GetStdHandle, STD_OUTPUT_HANDLE
mov consoleHandle,eax
INVOKE SetConsoleCursorPosition, consoleHandle, XYPos
INVOKE GetLocalTime, ADDR sysTime
```

Example: Displaying the System Time (3 of 3)

- Display the time using library calls:

```
mov    edx,OFFSET TheTimeIs      ; "The time is "
call   WriteString
movzx eax,sysTime.wHour        ; hours
call   WriteDec
mov    edx,offset colonStr     ; ":"
call   WriteString
movzx eax,sysTime.wMinute      ; minutes
call   WriteDec
mov    edx,offset colonStr     ; ":"
call   WriteString
movzx eax,sysTime.wSecond      ; seconds
call   WriteDec
```

Nested Structures (1 of 2)

- Define a structure that contains other structures.
- Used nested braces (or brackets) to initialize each COORD structure.

```
Rectangle STRUCT
    UpperLeft COORD <>
    LowerRight COORD <>
Rectangle ENDS
```

```
.data
rect1 Rectangle { {10,10}, {50,20} }
rect2 Rectangle < <10,10>, <50,20> >
```

```
COORD STRUCT
    X WORD ?
    Y WORD ?
COORD ENDS
```

Nested Structures (2 of 2)

- Use the dot (.) qualifier to access nested fields.
- Use indirect addressing to access the overall structure or one of its fields

```
mov rect1.UpperLeft.X, 10
mov esi,OFFSET rect1
mov (Rectangle PTR [esi]).UpperLeft.Y, 10

// use the OFFSET operator
mov edi,OFFSET rect2.LowerRight
mov (COORD PTR [edi]).X, 50
mov edi,OFFSET rect2.LowerRight.X
mov WORD PTR [edi], 50
```

Example: Drunkard's Walk

- Random-path simulation
- Uses a nested structure to accumulate path data as the simulation is running
- Uses a multiple branch structure to choose the direction

```
WalkMax = 50
DrunkardWalk STRUCT
    path COORD WalkMax DUP(<0,0>)
    pathsUsed WORD 0
DrunkardWalk ENDS
```

[View the source code](#)

Declaring and Using Unions

- A union is similar to a structure in that it contains multiple fields
- All of the fields in a union begin at the same offset
 - (differs from a structure)
- Provides alternate ways to access the same data
- Syntax:

unionname UNION

union-fields

unionname ENDS

Integer Union Example

The Integer union consumes 4 bytes (equal to the largest field)

```
Integer UNION
    D DWORD 0
    W WORD 0
    B BYTE 0
Integer ENDS
```

D, W, and B are often called variant fields.

Integer can be used to define data:

```
.data
val1 Integer <12345678h>
val2 Integer <100h>
val3 Integer <>
```

Integer Union Example

The variant field name is required when accessing the union:

```
mov val3.B, al  
mov ax, val3.W  
add val3.D, eax
```

Union Inside a Structure

An Integer union can be enclosed inside a FileInfo structure:

```
Integer UNION
    D DWORD 0
    W WORD 0
    B BYTE 0
Integer ENDS

FileInfo STRUCT
    FileID Integer <>
    FileName BYTE 64 DUP(?)
FileInfo ENDS

.data
myFile FileInfo <>
.code
mov myFile.FileID.W, ax
```

What's Next

- Structures
- Macros
- Conditional-Assembly Directives
- Defining Repeat Blocks

Macros

- Introducing Macros
- Defining Macros
- Invoking Macros
- Macro Examples
- Nested Macros
- Example Program: Wrappers

Introducing Macros

- A macro¹ is a named block of assembly language statements.
- Once defined, it can be invoked (called) one or more times.
- During the assembler's preprocessing step, each macro call is expanded into a copy of the macro.
- The expanded code is passed to the assembly step, where it is checked for correctness.

¹Also called a macro procedure.

Defining Macros

- A macro must be defined before it can be used.
- Parameters are optional.
- Each parameter follows the rules for identifiers. It is a string that is assigned a value when the macro is invoked.
- Syntax:

```
macroname MACRO [parameter-1, parameter-2,...]
```

```
statement-list
```

```
ENDM
```

mNewLine Macro Example

This is how you define and invoke a simple macro.

```
mNewLine MACRO          ; define the macro
    call CrLf
ENDM
.data

.code
mNewLine          ; invoke the macro
```

The assembler will substitute "call crlf" for "mNewLine".

mPutChar Macro

Writes a single character to standard output.

Definition:

```
mPutchar MACRO char  
    push eax  
    mov al,char  
    call WriteChar  
    pop eax  
ENDM
```

Invocation:

```
.code  
mPutchar 'A'
```

Expansion:

```
1     push eax  
1     mov al,'A'  
1     call WriteChar  
1     pop eax
```

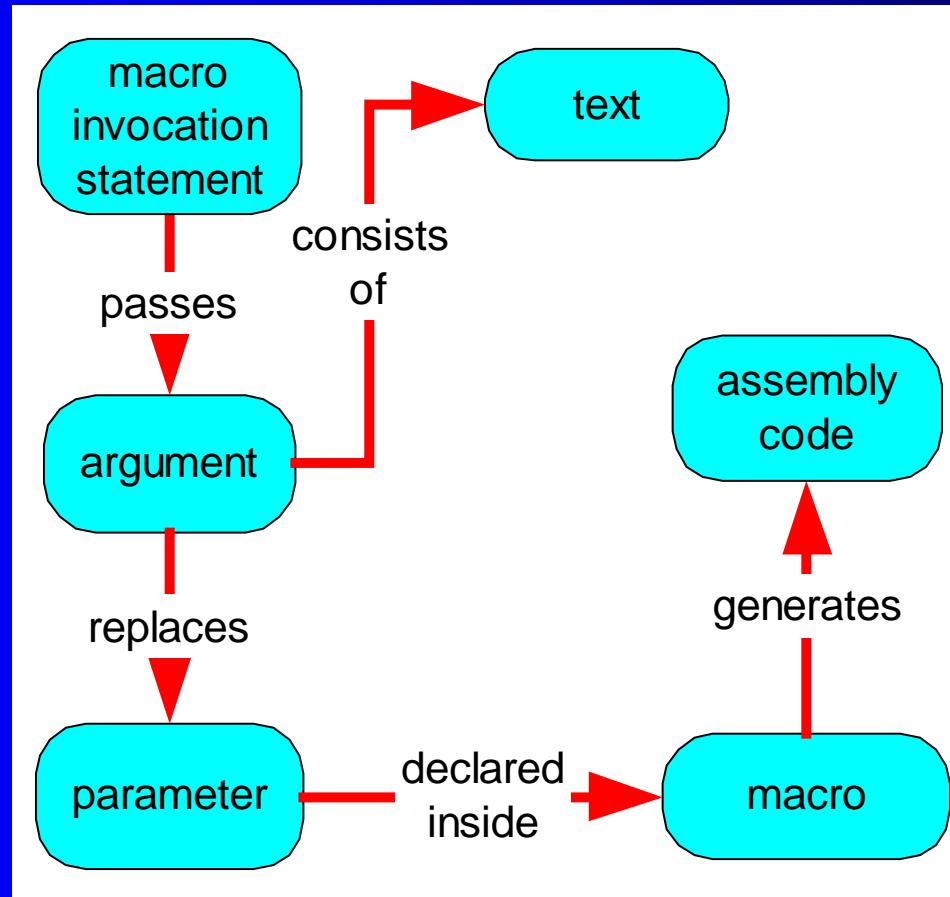
viewed in the
listing file

Invoking Macros (1 of 2)

- When you invoke a macro, each argument you pass matches a declared parameter.
- Each parameter is replaced by its corresponding argument when the macro is expanded.
- When a macro expands, it generates assembly language source code.
- Arguments are treated as simple text by the preprocessor.

Invoking Macros (2 of 2)

Relationships between macros, arguments, and parameters:



mWriteStr Macro (1 of 2)

Provides a convenient way to display a string, by passing the string name as an argument.

```
mWriteStr MACRO buffer
    push edx
    mov edx,OFFSET buffer
    call WriteString
    pop edx
ENDM
.data
str1 BYTE "Welcome!",0
.code
mWriteStr str1
```

mWriteStr Macro (2 of 2)

The expanded code shows how the str1 argument replaced the parameter named buffer:

```
mWriteStr MACRO buffer
    push edx
    mov edx,OFFSET buffer
    call WriteString
    pop edx
ENDM
```

```
1    push edx
1    mov edx,OFFSET str1
1    call WriteString
1    pop edx
```

Invalid Argument

- If you pass an invalid argument, the error is caught when the expanded code is assembled.
- Example:

```
.code  
mPutchar 1234h
```

```
1      push eax  
1      mov al,1234h          ; error!  
1      call WriteChar  
1      pop eax
```

Blank Argument

- If you pass a blank argument, the error is also caught when the expanded code is assembled.
- Example:

```
.code  
mPutchar
```

```
1      push eax  
1      mov al,  
1      call WriteChar  
1      pop eax
```

Macro Examples

- mReadStr - reads string from standard input
- mGotoXY - locates the cursor on screen
- mDumpMem - dumps a range of memory

mReadStr

The mReadStr macro provides a convenient wrapper around ReadString procedure calls.

```
mReadStr MACRO varName
    push ecx
    push edx
    mov edx,OFFSET varName
    mov ecx,(SIZEOF varName) - 1
    call ReadString
    pop edx
    pop ecx
ENDM
.data
firstName BYTE 30 DUP(?)
.code
mReadStr firstName
```

mGotoXY

The mGotoXY macro sets the console cursor position by calling the Gotoxy library procedure.

```
mGotoxy MACRO X:REQ, Y:REQ
    push edx
    mov dh,Y
    mov dl,X
    call Gotoxy
    pop edx
ENDM
```

The REQ next to X and Y identifies them as required parameters.

mDumpMem

The mDumpMem macro streamlines calls to the link library's DumpMem procedure.

```
mDumpMem MACRO address, itemCount, componentSize
    push ebx
    push ecx
    push esi
    mov  esi,address
    mov  ecx,itemCount
    mov  ebx,componentSize
    call DumpMem
    pop  esi
    pop  ecx
    pop  ebx
ENDM
```

mDump

The mDump macro displays a variable, using its known attributes. If <useLabel> is nonblank, the name of the variable is displayed.

```
mDump MACRO varName:REQ, useLabel
    IFB <varName>
        EXITM
    ENDIF
    call Crlf
    IFNB <useLabel>
        mWrite "Variable name: &varName"
    ELSE
        mWrite " "
    ENDIF
    mDumpMem OFFSET varName, LENGTHOF varName,
                TYPE varName
ENDM
```

mWrite

The mWrite macro writes a string literal to standard output. It is a good example of a macro that contains both code and data.

```
mWrite MACRO text
    LOCAL string
    .data
    string BYTE text,0
    .code
    push edx
    mov edx,OFFSET string
    call Writestring
    pop edx
ENDM
```

The LOCAL directive prevents **string** from becoming a global label.

Nested Macros

The mWriteLn macro contains a nested macro (a macro invoked by another macro).

```
mWriteLn MACRO text  
    mWrite text  
    call Crlf  
ENDM
```

```
mWriteLn "My Sample Macro Program"
```

```
2 .data  
2 ??0002 BYTE "My Sample Macro Program",0  
2 .code  
2 push edx  
2 mov edx,OFFSET ??0002  
2 call Writestring  
2 pop edx  
1 call Crlf
```

nesting level

Your turn . . .

- Write a nested macro named **mAskForString** that clears the screen, locates the cursor at a given row and column, prompts the user, and inputs a string. Use any macros shown so far.
- Use the following code and data to test your macro:

```
.data  
acctNum BYTE 30 DUP(?)  
.code  
main proc  
    mAskForString 5,10,"Input Account Number: ", \  
        acctNum
```

Solution . . .

. . . Solution

```
mAskForString MACRO row, col, prompt, inbuf
    call Clrscr
    mGotoXY col, row
    mWrite prompt
    mReadStr inbuf
ENDM
```

[View the solution program](#)

Example Program: Wrappers

- The *Wraps.asm* program demonstrates various macros from this chapter. It shows how macros can simplify the passing of register arguments to library procedures.
- View the [source code](#)

What's Next

- Structures
- Macros
- **Conditional-Assembly Directives**
- Defining Repeat Blocks

Conditional-Assembly Directives

- Checking for Missing Arguments
- Default Argument Initializers
- Boolean Expressions
- IF, ELSE, and ENDIF Directives
- The IFIDN and IFIDNI Directives
- Special Operators
- Macro Functions

Checking for Missing Arguments

- The IFB directive returns true if its argument is blank.
For example:

```
IFB <row>          ; if row is blank,  
    EXITM           ; exit the macro  
ENDIF
```

mWriteString Example

Display a message during assembly if the string parameter is empty:

```
mWriteStr MACRO string
    IFB <string>
        ECHO -----
        ECHO * Error: parameter missing in mWriteStr
        ECHO * (no code generated)
        ECHO -----
        EXITM
    ENDIF
    push edx
    mov edx,OFFSET string
    call WriteString
    pop edx
ENDM
```

Default Argument Initializers

- A default argument initializer automatically assigns a value to a parameter when a macro argument is left blank. For example, mWriteln can be invoked either with or without a string argument:

```
mWriteLn MACRO text:=<" ">
    mWrite text
    call Crlf
ENDM
.code
mWriteln "Line one"
mWriteln
mWriteln "Line three"
```

Sample output:

```
Line one
Line three
```

Boolean Expressions

A boolean expression can be formed using the following operators:

- LT - Less than
- GT - Greater than
- EQ - Equal to
- NE - Not equal to
- LE - Less than or equal to
- GE - Greater than or equal to

Only assembly-time constants may be compared using these operators.

IF, ELSE, and ENDIF Directives

A block of statements is assembled if the boolean expression evaluates to **true**. An alternate block of statements can be assembled if the expression is false.

```
IF boolean-expression
    statements
[ELSE
    statements]
ENDIF
```

Simple Example

The following IF directive permits two MOV instructions to be assembled if a constant named RealMode is equal to 1:

```
IF RealMode EQ 1
    mov ax,@data
    mov ds,ax
ENDIF
```

RealMode can be defined in the source code any of the following ways:

```
RealMode = 1
RealMode EQU 1
RealMode TEXT EQU 1
```

The IFIDN and IFIDNI Directives

- IFIDN compares two symbols and returns true if they are equal (case-sensitive)
- IFIDNI also compares two symbols, using a case-insensitive comparison
- Syntax:

```
IFIDNI <symbol>, <symbol>
```

```
    statements
```

```
ENDIF
```

Can be used to prevent the caller of a macro from passing an argument that would conflict with register usage inside the macro.

IFIDNI Example

Prevents the user from passing EDX as the second argument to the mReadBuf macro:

```
mReadBuf MACRO bufferPtr, maxChars
    IFIDNI <maxChars>,<EDX>
        ECHO Warning: Second argument cannot be EDX
        ECHO ****
        EXITM
    ENDIF
    .
    .
ENDM
```

Special Operators

- The substitution (&) operator resolves ambiguous references to parameter names within a macro.
- The expansion operator (%) expands text macros or converts constant expressions into their text representations.
- The literal-text operator (<>) groups one or more characters and symbols into a single text literal. It prevents the preprocessor from interpreting members of the list as separate arguments.
- The literal-character operator (!) forces the preprocessor to treat a predefined operator as an ordinary character.

Substitution (&)

Text passed as regName is substituted into the literal string definition:

```
ShowRegister MACRO regName
.data
tempStr BYTE " &regName=",0
.
.
.code
ShowRegister EDX           ; invoke the macro
```

Macro expansion:

```
tempStr BYTE " EDX=",0
```

Expansion (%)

Forces the evaluation of an integer expression. After the expression has been evaluated, its value is passed as a macro argument:

```
mGotoXY %(5 * 10),%(3 + 4)
```

The preprocessor generates the following code:

```
1 push edx
1 mov dl,50
1 mov dh,7
1 call Gotoxy
1 pop edx
```

Literal-Text (<>)

The first macro call passes three arguments. The second call passes a single argument:

```
mWrite "Line three", 0dh, 0ah  
mWrite <"Line three", 0dh, 0ah>
```

Literal-Character (!)

The following declaration prematurely ends the text definition when the first > character is reached.

```
BadYValue TEXTEQU Warning: <Y-coordinate is > 24>
```

The following declaration continues the text definition until the final > character is reached.

```
BadYValue TEXTEQU <Warning: Y-coordinate is !> 24>
```

Macro Functions (1 of 2)

- A macro function returns an integer or string constant
- The value is returned by the EXITM directive
- Example: The IsDefined macro acts as a wrapper for the IFDEF directive.

```
IsDefined MACRO symbol
    IFDEF symbol
        EXITM <-1>                ;; True
    ELSE
        EXITM <0>                 ;; False
    ENDIF
ENDM
```

Notice how the assembler defines True and False.

Macro Functions (2 of 2)

- When calling a macro function, the argument(s) must be enclosed in parentheses
- The following code permits the two MOV statements to be assembled only if the RealMode symbol has been defined:

```
IF IsDefined( RealMode )
    mov ax,@data
    mov ds,ax
ENDIF
```

What's Next

- Structures
- Macros
- Conditional-Assembly Directives
- **Defining Repeat Blocks**

Defining Repeat Blocks

- WHILE Directive
- REPEAT Directive
- FOR Directive
- FORC Directive
- Example: Linked List

WHILE Directive

- The WHILE directive repeats a statement block as long as a particular constant expression is true.
- Syntax:

```
WHILE constExpression
```

```
    statements
```

```
ENDM
```

WHILE Example

Generates Fibonacci integers between 1 and F0000000h at assembly time:

```
.data
val1 = 1
val2 = 1
DWORD val1          ; first two values
DWORD val2
val3 = val1 + val2

WHILE val3 LT 0F0000000h
    DWORD val3
    val1 = val2
    val2 = val3
    val3 = val1 + val2
ENDM
```

REPEAT Directive

- The REPEAT directive repeats a statement block a fixed number of times.
- Syntax:

```
REPEAT constExpression
```

```
    statements
```

```
ENDM
```

ConstExpression, an unsigned constant integer expression, determines the number of repetitions.

REPEAT Example

The following code generates 100 integer data definitions in the sequence 10, 20, 30, . . .

```
    ival = 10
REPEAT 100
    DWORD ival
    ival = ival + 10
ENDM
```

How might we assign a data name to this list of integers?

Your turn . . .

What will be the last integer to be generated by the following loop? 500

```
rows = 10
columns = 5
.data
ival = 10
REPEAT rows * columns
    DWORD ival
    ival = ival + 10
ENDM
```

FOR Directive

- The FOR directive repeats a statement block by iterating over a comma-delimited list of symbols.
- Each symbol in the list causes one iteration of the loop.
- Syntax:

FOR parameter,<arg1,arg2,arg3,...>

statements

ENDM

FOR Example

The following Window structure contains frame, title bar, background, and foreground colors. The field definitions are created using a FOR directive:

```
Window STRUCT  
    FOR color,<frame,titlebar,background,foreground>  
        color DWORD ?  
    ENDM  
Window ENDS
```

Generated code:

```
Window STRUCT  
    frame DWORD ?  
    titlebar DWORD ?  
    background DWORD ?  
    foreground DWORD ?  
Window ENDS
```

FORC Directive

- The FORC directive repeats a statement block by iterating over a string of characters. Each character in the string causes one iteration of the loop.
- Syntax:

FORC parameter, <string>

statements

ENDM

FORC Example

Suppose we need to accumulate seven sets of integer data for an experiment. Their label names are to be Group_A, Group_B, Group_C, and so on. The FORC directive creates the variables:

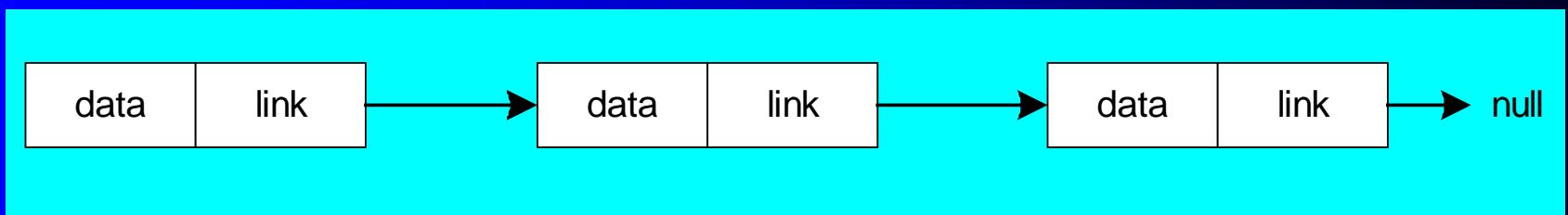
```
FORC code,<ABCDEFG>
    Group_&code WORD ?
ENDM
```

Generated code:

```
Group_A WORD ?
Group_B WORD ?
Group_C WORD ?
Group_D WORD ?
Group_E WORD ?
Group_F WORD ?
Group_G WORD ?
```

Example: Linked List (1 of 5)

- We can use the REPT directive to create a singly linked list at assembly time.
- Each node contains a pointer to the next node.
- A null pointer in the last node marks the end of the list



Linked List (2 of 5)

- Each node in the list is defined by a ListNode structure:

```
ListNode STRUCT
    NodeData DWORD ?          ; the node's data
    NextPtr  DWORD ?          ; pointer to next node
ListNode ENDS

TotalNodeCount = 15
NULL = 0
Counter = 0
```

Linked List (3 of 5)

- The REPEAT directive generates the nodes.
- Each ListNode is initialized with a counter and an address that points 8 bytes beyond the current node's location:

```
.data  
LinkedList LABEL PTR ListNode  
REPEAT TotalNodeCount  
    Counter = Counter + 1  
    ListNode <Counter, ($ + Counter * SIZEOF ListNode)>  
ENDM
```

The value of \$ does not change—it remains fixed at the location of the LinkedList label.

Linked List (4 of 5)

The following hexadecimal values in each node show how each **NextPtr** field contains the address of its following node.

offset	contents	
00000000	00000001	
00000008	00000008	← NextPtr
00000008	00000002	
00000010	00000010	←
00000010	00000003	
00000018	00000018	←
00000018	00000004	
00000020	00000020	
00000020	(etc.)	

Linked List (5 of 5)

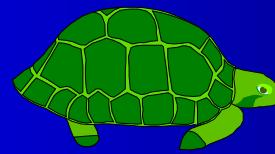
[View the program's source code](#)

Sample output:

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15
```

Summary

- Use a structure to define complex types
 - contains fields of different types
- Macro – named block of statements
 - substituted by the assembler preprocessor
- Conditional assembly directives
 - IF, IFNB, IFIDNI, ...
- Operators: &, %, <>, !
- Repeat block directives (assembly time)
 - WHILE, REPEAT, FOR, FORC



4D 77 69 73 68 6F

Assembly Language for x86 Processors

7th Edition

Kip R. Irvine

Chapter 11: MS-Windows Programming

Slide show prepared by the author

Revision date: 1/15/2014

Chapter Overview

- **Win32 Console Programming**
- Writing a Graphical Windows Application
- Dynamic Memory Allocation
- x86 Memory Management

Useful Questions

- How do 32-bit programs handle text input-output?
- How are colors handled in 32-bit console mode?
- How does the Irvine32 link library work?
- How are times and dates handled in MS-Windows?
- How can I use MS-Windows functions to read and write data files?
- Is it possible to write a graphical Windows application in assembly language?
- How do Protected mode programs translate segments and offsets to physical addresses?
- I've heard that virtual memory is good. But why is that so?

Win32 Console Programming

- Background Information
 - Win32 Console Programs
 - API and SDK
 - Windows Data Types
 - Standard Console Handles
- Console Input
- Console Output
- Reading and Writing Files
- Console Window Manipulation
- Controlling the Cursor
- Controlling the Text Color
- Time and Date Functions

Win32 Console Programs

- Run in Protected mode
- Emulate MS-DOS
- Standard text-based input and output
- Linker option : /SUBSYSTEM:CONSOLE
- The **console input buffer** contains a queue of input records, each containing data about an input event.
- A **console screen buffer** is a two-dimensional array of character and color data that affects the appearance of text in the console window.

Classifying Console Functions

- Text-oriented (high-level) console functions
 - Read character streams from input buffer
 - Write character streams to screen buffer
 - Redirect input and output
- Event-oriented (low-level) console functions
 - Retrieve keyboard and mouse events
 - Detect user interactions with the console window
 - Control window size & position, text colors

API and SDK

- Microsoft Win32 Application Programming Interface
 - API: a collection of types, constants, and functions that provide a way to directly manipulate objects through programming
- Microsoft Platform Software Development Kit
 - SDK: a collection of tools, libraries, sample code, and documentation that helps programmers create applications
 - Platform: an operating system or a group of closely related operating systems

Translating Windows Data Types

Windows Type(s)	MASM Type
BOOL	DWORD
LONG	SDWORD
COLORREF, HANDLE, LPARAM, LPCTSTR, LPTSTR, LPVOID, LRESULT, UINT, WNDPROC, WPARAM	DWORD
BSTR, LPCSTR, LPSTR	PTR BYTE
WORD	WORD
LPCRECT	PTR RECT

Standard Console Handles

A handle is an unsigned 32-bit integer. The following MS-Windows constants are predefined to specify the type of handle requested:

- STD_INPUT_HANDLE
 - standard input
- STD_OUTPUT_HANDLE
 - standard output
- STD_ERROR_HANDLE
 - standard error output

GetStdHandle

- GetStdHandle returns a handle to a console stream
- Specify the type of handle (see previous slide)
- The handle is returned in EAX
- Prototype:

```
GetStdHandle PROTO,  
    nStdHandle:DWORD           ; handle type
```

- Sample call:

```
INVOKE GetStdHandle, STD_OUTPUT_HANDLE  
mov myHandle, eax
```

Console Input

- The ReadConsole function provides a convenient way to read text input and put it in a buffer.
- Prototype:

```
ReadConsole PROTO,  
    handle:DWORD,                      ; input handle  
    pBuffer:PTR BYTE,                  ; pointer to buffer  
    maxBytes:DWORD,                   ; number of chars to read  
    pBytesRead:PTR DWORD,             ; ptr to num bytes read  
    notUsed:DWORD                     ; (not used)
```

Single-Character Input

Here's how to input single characters:

- Get a copy of the current console flags by calling `GetConsoleMode`. Save the flags in a variable.
- Change the console flags by calling `SetConsoleMode`.
- Input a character by calling `ReadConsole`.
- Restore the previous values of the console flags by calling `SetConsoleMode`.

Excerpts from ReadChar (1 of 2)

From the ReadChar procedure in the Irvine32 library:

```
.data
consoleInHandle DWORD ?
saveFlags DWORD ?                                ; backup copy of flags

.code
; Get & save the current console input mode flags
Invoke GetConsoleMode, consoleInHandle, ADDR saveFlags

; Clear all console flags
Invoke SetConsoleMode, consoleInHandle, 0
```

Excerpts from ReadChar (2 of 2)

From the ReadChar procedure in the Irvine32 library:

```
; Read a single character from input
Invoke ReadConsole,
    consoleInHandle,          ; console input handle
    ADDR buffer,              ; pointer to buffer
    1,                         ; max characters to read
    ADDR bytesRead,           ; return num bytes read
    0                          ; not used

; Restore the previous flags state
Invoke SetConsoleMode, consoleInHandle, saveFlags
```

COORD and SMALL_RECT

- The COORD structure specifies X and Y screen coordinates in character measurements, which default to 0-79 and 0-24.
- The SMALL_RECT structure specifies a window's location in character measurements.

```
COORD STRUCT
```

```
    X WORD ?
```

```
    Y WORD ?
```

```
COORD ENDS
```

```
SMALL_RECT STRUCT
```

```
    Left WORD ?
```

```
    Top WORD ?
```

```
    Right WORD ?
```

```
    Bottom WORD ?
```

```
SMALL_RECT ENDS
```

WriteConsole

- The WriteConsole function writes a string to the screen, using the console output handle. It acts upon standard ASCII control characters such as tab, carriage return, and line feed.
- Prototype:

```
WriteConsole PROTO,  
    handle:DWORD,                      ; output handle  
    pBuffer:PTR BYTE,                  ; pointer to buffer  
    bufsize:DWORD,                     ; size of buffer  
    pCount:PTR DWORD,                 ; output count  
    lpReserved:DWORD                  ; (not used)
```

WriteConsoleOutputCharacter

- The WriteConsoleOutputCharacter function copies an array of characters to consecutive cells of the console screen buffer, beginning at a specified location.
- Prototype:

```
WriteConsoleOutputCharacter PROTO,  
    handleScreenBuf:DWORD,      ; console output handle  
    pBuffer:PTR BYTE,          ; pointer to buffer  
    bufsize:DWORD,              ; size of buffer  
    xyPos:COORD,                ; first cell coordinates  
    pCount:PTR DWORD           ; output count
```

File Manipulation

- Win32 API Functions that create, read, and write to files:
 - CreateFile
 - ReadFile
 - WriteFile
 - SetFilePointer

CreateFile

- CreateFile either creates a new file or opens an existing file. If successful, it returns a handle to the open file; otherwise, it returns a special constant named INVALID_HANDLE_VALUE.
- Prototype:

```
CreateFile PROTO,  
    pFilename:PTR BYTE,           ; ptr to filename  
    desiredAccess:DWORD,          ; access mode  
    shareMode:DWORD,              ; share mode  
    lpSecurity:DWORD,             ; ptr to security attrs  
    creationDisposition:DWORD,    ; file creation options  
    flagsAndAttributes:DWORD,     ; file attributes  
    htemplate:DWORD               ; handle to template file
```

CreateFile Examples (1 of 3)

Opens an existing file for reading:

```
INVOKE CreateFile,  
    ADDR filename,          ; ptr to filename  
    GENERIC_READ,           ; access mode  
    DO_NOT_SHARE,           ; share mode  
    NULL,                  ; ptr to security attributes  
    OPEN_EXISTING,          ; file creation options  
    FILE_ATTRIBUTE_NORMAL,  ; file attributes  
    0                      ; handle to template file
```

CreateFile Examples (2 of 3)

Opens an existing file for writing:

```
INVOKE CreateFile,
    ADDR filename,
    GENERIC_WRITE,           ; access mode
    DO_NOT_SHARE,
    NULL,
    OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL,
    0
```

CreateFile Examples (3 of 3)

Creates a new file with normal attributes, erasing any existing file by the same name:

```
INVOKE CreateFile,  
    ADDR filename,  
    GENERIC_WRITE,  
    DO_NOT_SHARE,  
    NULL,  
    CREATE_ALWAYS,           ; overwrite existing file  
    FILE_ATTRIBUTE_NORMAL,  
    0
```

ReadFile

- ReadFile reads text from an input file
- Prototype:

```
ReadFile PROTO,  
    handle:DWORD,                      ; handle to file  
    pBuffer:PTR BYTE,                  ; ptr to buffer  
    nBufsize:DWORD,                   ; num bytes to read  
    pBytesRead:PTR DWORD,             ; bytes actually read  
    pOverlapped:PTR DWORD            ; ptr to asynch info
```

WriteFile

- WriteFile writes data to a file, using an output handle. The handle can be the screen buffer handle, or it can be one assigned to a text file.
- Prototype:

```
WriteFile PROTO,  
    fileHandle:DWORD,                      ; output handle  
    pBuffer:PTR BYTE,                      ; pointer to buffer  
    nBufsize:DWORD,                        ; size of buffer  
    pBytesWritten:PTR DWORD,                ; num bytes written  
    pOverlapped:PTR DWORD                 ; ptr to asynch info
```

SetFilePointer

SetFilePointer moves the position pointer of an open file. You can use it to append data to a file, and to perform random-access record processing:

```
SetFilePointer PROTO,  
    handle:DWORD,                      ; file handle  
    nDistanceLo:SDWORD,                 ; bytes to move pointer  
    pDistanceHi:PTR SDWORD,            ; ptr to bytes to move  
    moveMethod:DWORD                   ; starting point
```

Example:

```
; Move to end of file:  
  
Invoke SetFilePointer,  
    fileHandle,0,0,FILE_END
```

64-Bit Windows API

- Input and output handles are 64 bits
- Before calling a system function, reserve at least 32 bytes of shadow space by subtracting from the stack pointer (RSP).
- Restore RSP after the system call
- Pass integers in 64-bit registers
- First four arguments should be placed in RCX, RDX, R8, and R9 registers
- 64-bit integer values are returned in RAX

Example: Calling GetStdHandle

```
.data  
STD_OUTPUT_HANDLE EQU -11  
consoleOutHandle QWORD ?  
  
.code  
sub  rsp,40          ; reserve shadow space & align RSP  
mov   rcx,STD_OUTPUT_HANDLE  
call  GetStdHandle  
mov   consoleOutHandle,rax  
add   rsp,40
```

Example: Calling WriteConsole

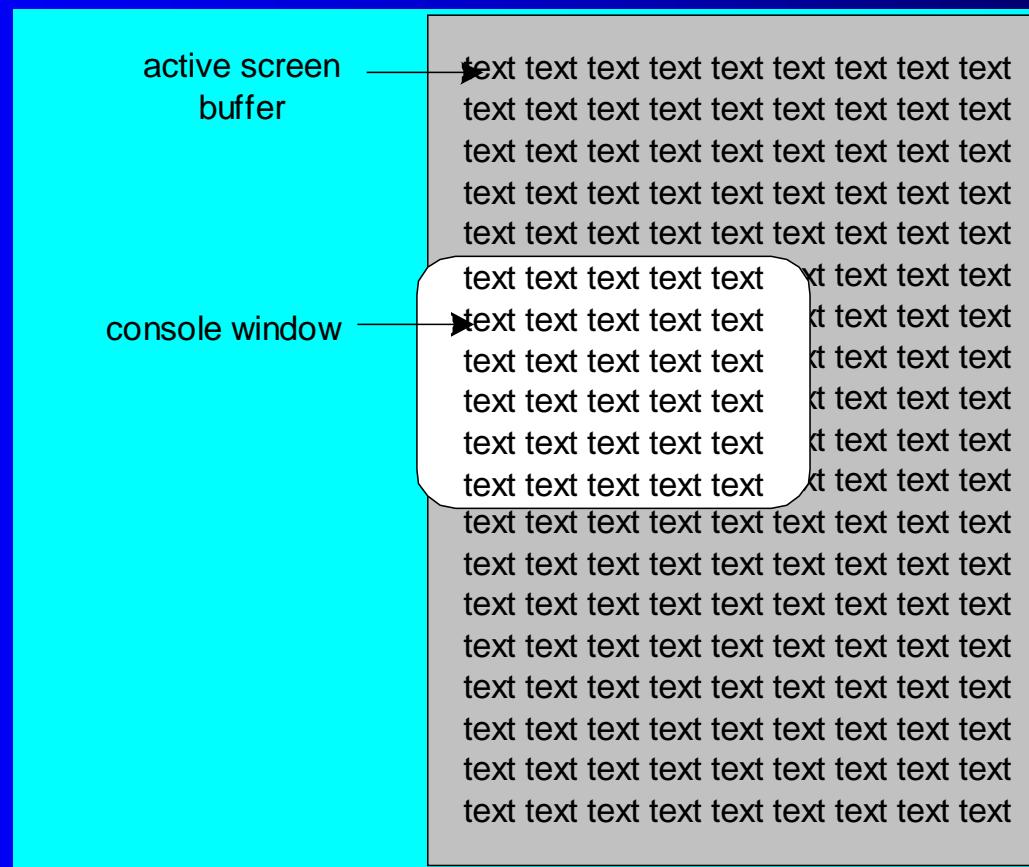
```
sub  rsp, 5 * 8          ; space for 5 parameters
movr cx,rdx
call Str_length          ; returns length of string in EAX
mov  rcx,consoleOutHandle
mov  rdx,rdx              ; string pointer
mov  r8, rax              ; length of string
lea   r9,bytesWritten
mov  qword ptr [rsp + 4 * SIZEOF QWORD],0 ; (always zero)
call WriteConsoleA
```

Console Window Manipulation

- Screen buffer
- Console window
- Controlling the cursor
- Controlling the text color

Screen Buffer and Console Window

- The active screen buffer includes data displayed by the console window.



SetConsoleTitle

SetConsoleTitle changes the console window's title.
Pass it a null-terminated string:

```
.data  
titleStr BYTE "Console title",0  
.code  
INVOKE SetConsoleTitle, ADDR titleStr
```

GetConsoleScreenBufferInfo

GetConsoleScreenBufferInfo returns information about the current state of the console window. It has two parameters: a handle to the console screen, and a pointer to a structure that is filled in by the function:

```
.data  
outHandle DWORD ?  
consoleInfo CONSOLE_SCREEN_BUFFER_INFO <>  
.code  
    INVOKE GetConsoleScreenBufferInfo,  
        outHandle,  
        ADDR consoleInfo
```

CONSOLE_SCREEN_BUFFER_INFO

```
CONSOLE_SCREEN_BUFFER_INFO STRUCT  
    dwSize          COORD <>  
    dwCursorPosition COORD <>  
    wAttributes     WORD ?  
    srWindow        SMALL_RECT <>  
    maxWinSize      COORD <>  
CONSOLE_SCREEN_BUFFER_INFO ENDS
```

- dwSize - size of the screen buffer (char columns and rows)
- dwCursorPosition - cursor location
- wAttributes - colors of characters in console buffer
- srWindow - coords of console window relative to screen buffer
- maxWinSize - maximum size of the console window

SetConsoleWindowInfo

- SetConsoleWindowInfo lets you set the size and position of the console window relative to its screen buffer.
- Prototype:

```
SetConsoleWindowInfo PROTO,  
    nStdHandle:DWORD,                      ; screen buffer handle  
    bAbsolute:DWORD,                        ; coordinate type  
    pConsoleRect:PTR SMALL_RECT            ; window rectangle
```

SetConsoleScreenBufferSize

- SetConsoleScreenBufferSize lets you set the screen buffer size to X columns by Y rows.
- Prototype:

```
SetConsoleScreenBufferSize PROTO,  
    outHandle:DWORD,           ; handle to screen buffer  
    dwSize:COORD              ; new screen buffer size
```

Controlling the Cursor

- `GetConsoleCursorInfo`
 - returns the size and visibility of the console cursor
- `SetConsoleCursorInfo`
 - sets the size and visibility of the cursor
- `SetConsoleCursorPosition`
 - sets the X, Y position of the cursor

CONSOLE_CURSOR_INFO

- Structure containing information about the console's cursor size and visibility:

```
CONSOLE_CURSOR_INFO STRUCT  
    dwSize     DWORD ?  
    bVisible   DWORD ?  
CONSOLE_CURSOR_INFO ENDS
```

SetConsoleTextAttribute

- Sets the foreground and background colors of all subsequent text written to the console.
- Prototype:

```
SetConsoleTextAttribute PROTO,  
    outHandle:DWORD,           ; console output handle  
    nColor:DWORD               ; color attribute
```

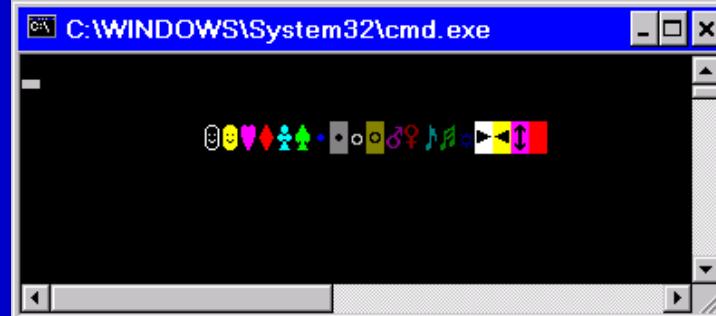
WriteConsoleOutputAttribute

- Copies an array of attribute values to consecutive cells of the console screen buffer, beginning at a specified location.
- Prototype:

```
WriteConsoleOutputAttribute PROTO,  
    outHandle:DWORD,           ; output handle  
    pAttribute:PTR WORD,      ; write attributes  
    nLength:DWORD,            ; number of cells  
    xyCoord:COORD,            ; first cell coordinates  
    lpCount:PTR DWORD         ; number of cells written
```

WriteColors Program

- Creates an array of characters and an array of attributes, one for each character
 - Copies the attributes to the screen buffer
 - Copies the characters to the same screen buffer cells as the attributes
 - Sample output:



(starts in row 2, column 10)

View the source code

Time and Date Functions

- GetLocalTime, SetLocalTime
- GetTickCount, Sleep
- GetDateTime
- SYSTEMTIME Structure
- Creating a Stopwatch Timer

GetLocalTime, SetLocalTime

- `GetLocalTime` returns the date and current time of day, according to the system clock.
- `SetLocalTime` sets the system's local date and time.

```
GetLocalTime PROTO,  
    pSystemTime:PTR SYSTEMTIME
```

```
SetLocalTime PROTO,  
    pSystemTime:PTR SYSTEMTIME
```

GetTickCount, Sleep

- GetTickCount function returns the number of milliseconds that have elapsed since the system was started.
- Sleep pauses the current program for a specified number of milliseconds.

```
GetTickCount PROTO      ; return value in EAX
```

```
Sleep PROTO,  
dwMilliseconds:DWORD
```

GetDateTime

The **GetDateTime** procedure in the **Irvine32** library calculates the number of 100-nanosecond time intervals that have elapsed since January 1, 1601. Pass it a pointer to an empty 64-bit **FILETIME** structure, which is then filled in by the procedure:

```
GetDateTime PROC,  
    pStartTime:PTR QWORD
```

```
FILETIME STRUCT  
    loDateTime DWORD ?  
    hiDateTime DWORD ?  
FILETIME ENDS
```

SYSTEMTIME Structure

- SYSTEMTIME is used by date and time-related Windows API functions:

```
SYSTEMTIME STRUCT
    wYear WORD ?          ; year (4 digits)
    wMonth WORD ?         ; month (1-12)
    wDayOfWeek WORD ?     ; day of week (0-6)
    wDay WORD ?           ; day (1-31)
    wHour WORD ?          ; hours (0-23)
    wMinute WORD ?        ; minutes (0-59)
    wSecond WORD ?        ; seconds (0-59)
    wMilliseconds WORD ?   ; milliseconds (0-999)
SYSTEMTIME ENDS
```

Creating a Stopwatch Timer

- The Timer.asm program demonstrates a simple stopwatch timer
- It has two important functions:
 - **TimerStart** - receives a pointer to a doubleword, into which it saves the current time
 - **TimerStop** - receives a pointer to the same doubleword, and returns the difference (in milliseconds) between the current time and the previously recorded time
- Calls the Win32 **GetTickCount** function
- [View the source code](#)

What's Next

- Win32 Console Programming
- **Writing a Graphical Windows Application**
- Dynamic Memory Allocation
- x86 Memory Management

Writing a Graphical Windows Application

- Required Files
- POINT, RECT Structures
- MSGStruct, WNDCLASS Structures
- MessageBox Function
- WinMain, WinProc Procedures
- ErrorHandler Procedure
- Message Loop & Processing Messages
- Program Listing

Required Files

- `make32.bat` - Batch file specifically for building this program
- `WinApp.asm` - Program source code
- `GraphWin.inc` - Include file containing structures, constants, and function prototypes used by the program
- `kernel32.lib` - Same MS-Windows API library used earlier in this chapter
- `user32.lib` - Additional MS-Windows API functions

POINT and RECT Structures

- POINT - X, Y screen coordinates
- RECT - Holds the graphical coordinates of two opposing corners of a rectangle

```
POINT STRUCT
```

```
    ptX    DWORD  ?  
    ptY    DWORD  ?
```

```
POINT ENDS
```

```
RECT STRUCT
```

```
    left     DWORD  ?  
    top      DWORD  ?  
    right    DWORD  ?  
    bottom   DWORD  ?
```

```
RECT ENDS
```

MSGStruct Structure

MSGStruct - holds data for MS-Windows messages (usually passed by the system and received by your application):

```
MSGStruct STRUCT  
    msgWnd             DWORD ?  
    msgMessage         DWORD ?  
    msgWparam          DWORD ?  
    msgLparam          DWORD ?  
    msgTime            DWORD ?  
    msgPt              POINT <>  
MSGStruct ENDS
```

WNDCLASS Structure (1 of 2)

Each window in a program belongs to a class, and each program defines a window class for its main window:

```
WNDCLASS STRUC
    style          DWORD ?      ; window style options
    lpfnWndProc   DWORD ?      ; WinProc function pointer
    cbClsExtra    DWORD ?      ; shared memory
    cbWndExtra    DWORD ?      ; number of extra bytes
    hInstance     DWORD ?      ; handle to current program
    hIcon          DWORD ?      ; handle to icon
    hCursor        DWORD ?      ; handle to cursor
    hbrBackground  DWORD ?      ; handle to background brush
    lpszMenuName  DWORD ?      ; pointer to menu name
    lpszClassName DWORD ?      ; pointer to WinClass name
WNDCLASS ENDS
```

WNDCLASS Structure (2 of 2)

- **style** is a conglomerate of different style options, such as WS_CAPTION and WS_BORDER, that control the window's appearance and behavior.
- **lpfnWndProc** is a pointer to a function (in our program) that receives and processes event messages triggered by the user.
- **cbClsExtra** refers to shared memory used by all windows belonging to the class. Can be null.
- **cbWndExtra** specifies the number of extra bytes to allocate following the window instance.
- **hInstance** holds a handle to the current program instance.
- **hIcon** and **hCursor** hold handles to icon and cursor resources for the current program.
- **hbrBackground** holds a background (color) brush.
- **lpszMenuName** points to a menu string.
- **lpszClassName** points to a null-terminated string containing the window's class name.

MessageBox Function

Displays text in a box that pops up and waits for the user to click on a button:

```
MessageBox PROTO,  
    hWnd:DWORD,  
    pText:PTR BYTE,  
    pCaption:PTR BYTE,  
    style:DWORD
```

hWnd is a handle to the current window. pText points to a null-terminated string that will appear inside the box. pCaption points to a null-terminated string that will appear in the box's caption bar. style is an integer that describes both the dialog box's icon (optional) and the buttons (required).

MessageBox Example

Displays a message box that shows a question, including an OK button and a question-mark icon:

```
.data
hMainWnd        DWORD ?
QuestionText    BYTE "Register this program now?"
QuestionTitle   BYTE "Trial Period Has Expired"

.code
INVOKE MessageBox,
    hMainWnd,
    ADDR QuestionText,
    ADDR QuestionTitle,
    MB_OK + MB_ICONQUESTION
```

WinMain Procedure

Every Windows application needs a startup procedure, usually named WinMain, which is responsible for the following tasks:

- Get a handle to the current program
- Load the program's icon and mouse cursor
- Register the program's main window class and identify the procedure that will process event messages for the window
- Create the main window
- Show and update the main window
- Begin a loop that receives and dispatches messages

WinProc Procedure

- WinProc receives and processes all event messages relating to a window
 - Some events are initiated by clicking and dragging the mouse, pressing keyboard keys, and so on
- WinProc decodes each message, carries out application-oriented tasks related to the message

```
WinProc PROC,  
    hWnd:DWORD,           ; handle to the window  
    localMsg:DWORD,        ; message ID  
    wParam:DWORD,          ; parameter 1 (varies)  
    lParam:DWORD           ; parameter 2 (varies)
```

(Contents of wParam and lParam vary, depending on the message.)

Sample WinProc Messages

- In the example program from this chapter, the WinProc procedure handles three specific messages:
 - WM_LBUTTONDOWN, generated when the user presses the left mouse button
 - WM_CREATE, indicates that the main window was just created
 - WM_CLOSE, indicates that the application's main window is about to close

(many other messages are possible)

ErrorHandler Procedure

- The ErrorHandler procedure has several important tasks to perform:
 - Call GetLastError to retrieve the system error number
 - Call FormatMessage to retrieve the appropriate system-formatted error message string
 - Call MessageBox to display a popup message box containing the error message string
 - Call LocalFree to free the memory used by the error message string

(sample)

Error Handler Sample

```
Invoke GetLastErrorHandler ; Returns message ID in EAX
Mov messageID, eax

; Get the corresponding message string.
Invoke FormatMessage, FORMAT_MESSAGE_ALLOCATE_BUFFER + \
FORMAT_MESSAGE_FROM_SYSTEM, NULL, messageID, NULL,
Addr pErrorMsg, NULL, NULL

; Display the error message.
Invoke MessageBox, NULL, pErrorMsg, ADDR ErrorTitle,
MB_ICONERROR + MB_OK

; Free the error message string.
Invoke LocalFree, pErrorMsg
```

Message Loop

In WinMain, the message loop receives and dispatches (relays) messages:

```
Message_Loop:  
    ; Get next message from the queue.  
    INVOKE GetMessage, ADDR msg, NULL,NULL,NULL  
  
    ; Quit if no more messages.  
.IF eax == 0  
    jmp Exit_Program  
.ENDIF  
  
    ; Relay the message to the program's WinProc.  
    INVOKE DispatchMessage, ADDR msg  
  
    jmp Message_Loop
```

Processing Messages

WinProc receives each message and decides what to do with it:

```
WinProc PROC, hWnd:DWORD, localMsg:DWORD,  
    wParam:DWORD, lParam:DWORD  
  
    mov eax, localMsg  
  
.IF eax == WM_LBUTTONDOWN      ; mouse button?  
        INVOKE MessageBox, hWnd, ADDR PopupText,  
                    ADDR PopupTitle, MB_OK  
        jmp WinProcExit  
  
.ELSEIF eax == WM_CREATE       ; create window?  
        INVOKE MessageBox, hWnd, ADDR AppLoadMsgText,  
                    ADDR AppLoadMsgTitle, MB_OK  
        jmp WinProcExit  
  
(etc.)
```

Program Listing

- View the program listing (WinApp.asm)
- Run the program

When linking the program, remember to replace
/SUBSYSTEM:CONSOLE
with: /SUBSYSTEM:WINDOWS

What's Next

- Win32 Console Programming
- Writing a Graphical Windows Application
- **Dynamic Memory Allocation**
- x86 Memory Management

Dynamic Memory Allocation

- Reserving memory at runtime for objects
 - aka *heap allocation*
 - standard in high-level languages (C++, Java)
- Heap manager
 - allocates large blocks of memory
 - maintains free list of pointers to smaller blocks
 - manages requests by programs for storage

Windows Heap-Related Functions

Function	Description
GetProcessHeap	Returns a 32-bit integer handle to the program's existing heap area in EAX. If the function succeeds, it returns a handle to the heap in EAX. If it fails, the return value in EAX is NULL.
HeapAlloc	Allocates a block of memory from a heap. If it succeeds, the return value in EAX contains the address of the memory block. If it fails, the returned value in EAX is NULL.
HeapCreate	Creates a new heap and makes it available to the calling program. If the function succeeds, it returns a handle to the newly created heap in EAX. If it fails, the return value in EAX is NULL.
HeapDestroy	Destroys the specified heap object and invalidates its handle. If the function succeeds, the return value in EAX is nonzero.
HeapFree	Frees a block of memory previously allocated from a heap, identified by its address and heap handle. If the block is freed successfully, the return value is nonzero.
HeapReAlloc	Reallocates and resizes a block of memory from a heap. If the function succeeds, the return value is a pointer to the reallocated memory block. If the function fails and you have not specified HEAP_GENERATE_EXCEPTIONS, the return value is NULL.
HeapSize	Returns the size of a memory block previously allocated by a call to HeapAlloc or HeapReAlloc. If the function succeeds, EAX contains the size of the allocated memory block, in bytes. If the function fails, the return value is SIZE_T – 1. (SIZE_T equals the maximum number of bytes to which a pointer can point.)

Sample Code

- Get a handle to the program's existing heap:

```
.data  
hHeap HANDLE ?  
  
.code  
INVOKE GetProcessHeap  
.IF eax == NULL ; cannot get handle  
    jmp quit  
.ELSE  
    mov hHeap,eax ; handle is OK  
.ENDIF
```

Sample Code

- Allocate block of memory from existing heap:

```
.data
hHeap HANDLE ?          ; heap handle
pArray DWORD ?          ; pointer to array

.code
Invoke HeapAlloc, hHeap, HEAP_ZERO_MEMORY, 1000
.IF eax == NULL
    mWrite "HeapAlloc failed"
    jmp quit
.ELSE
    mov pArray,eax
.ENDIF
```

Sample Code

- Free a block of memory previously created by calling HeapAlloc:

```
.data  
hHeap HANDLE ? ; heap handle  
pArray DWORD ? ; pointer to array  
  
.code  
INVOKE HeapFree,  
    hHeap, ; handle to heap  
    0, ; flags  
    pArray ; pointer to array
```

Sample Programs

- Heaptest1.asm
 - Allocates and fills an array of bytes
- Heaptest2.asm
 - Creates a heap and allocates multiple memory blocks until no more memory is available

What's Next

- Win32 Console Programming
- Writing a Graphical Windows Application
- Dynamic Memory Allocation
- **x86 Memory Management**

x86 Memory Management

- Reviewing Some Terms
- New Terms
- Translating Addresses
- Converting Logical to Linear Address
- Page Translation

Reviewing Some Terms

- Multitasking permits multiple programs (or tasks) to run at the same time. The processor divides up its time between all of the running programs.
- Segments are variable-sized areas of memory used by a program containing either code or data.
- Segmentation provides a way to isolate memory segments from each other. This permits multiple programs to run simultaneously without interfering with each other.
- A segment descriptor is a 64-bit value that identifies and describes a single memory segment: it contains information about the segment's base address, access rights, size limit, type, and usage.

New Terms

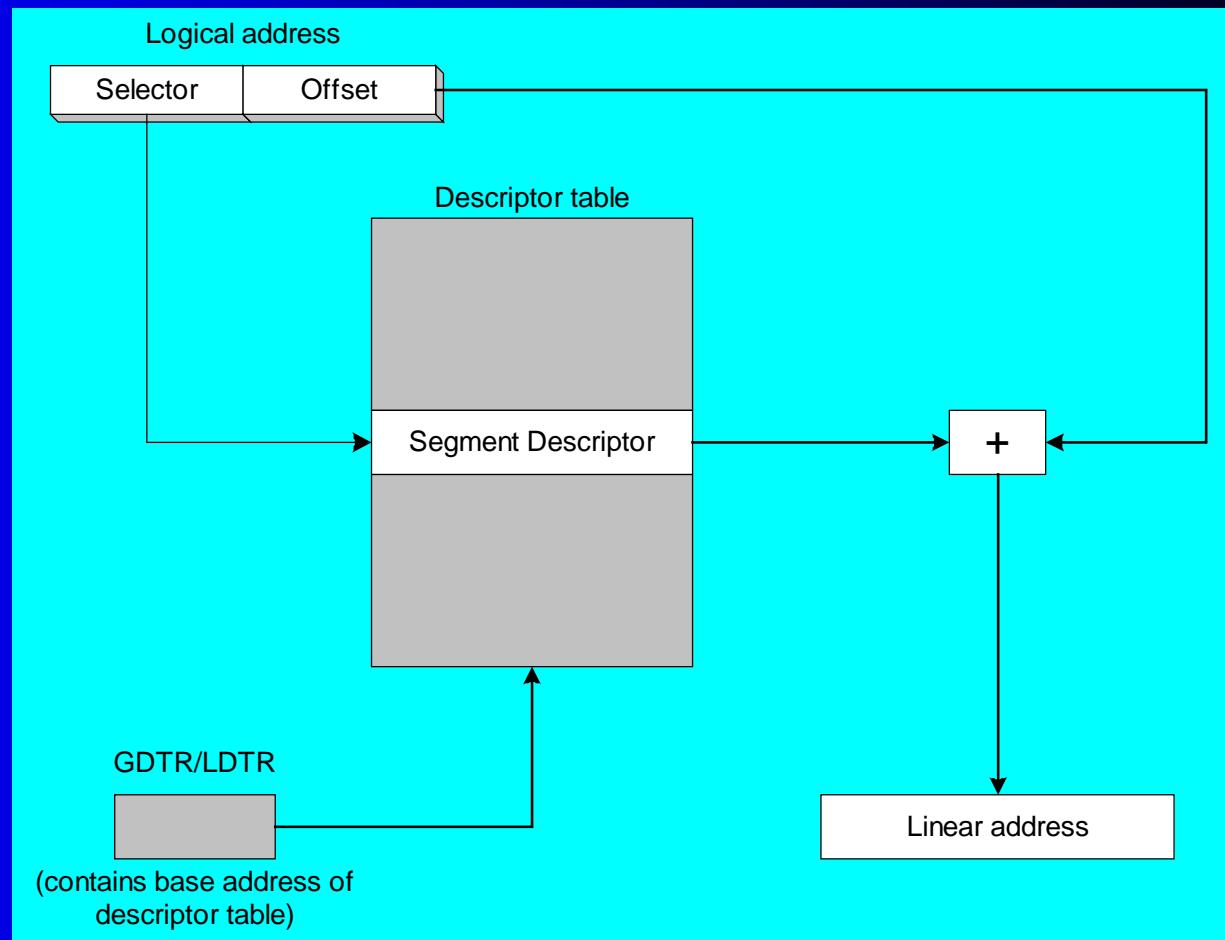
- A **segment selector** is a 16-bit value stored in a segment register (CS, DS, SS, ES, FS, or GS).
 - provides an indirect reference to a memory segment
- A **logical address** is a combination of a segment selector and a 32-bit offset.

Translating Addresses

- The x86 processor uses a one- or two-step process to convert a variable's logical address into a unique memory location.
- The first step combines a segment value with a variable's offset to create a linear address.
- The second optional step, called page translation, converts a linear address to a physical address.

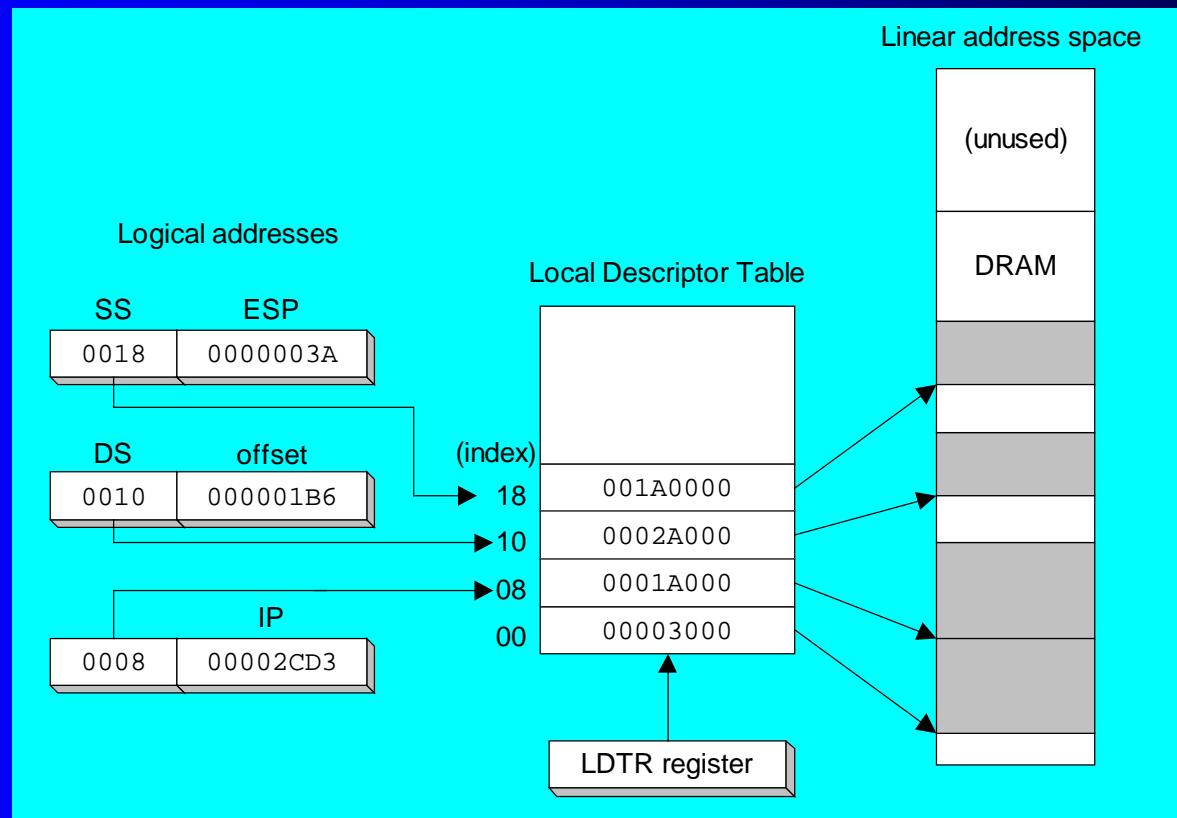
Converting Logical to Linear Address

The segment selector points to a segment descriptor, which contains the base address of a memory segment. The 32-bit offset from the logical address is added to the segment's base address, generating a 32-bit linear address.



Indexing into a Descriptor Table

Each segment descriptor indexes into the program's local descriptor table (LDT). Each table entry is mapped to a linear address:



Paging (1 of 2)

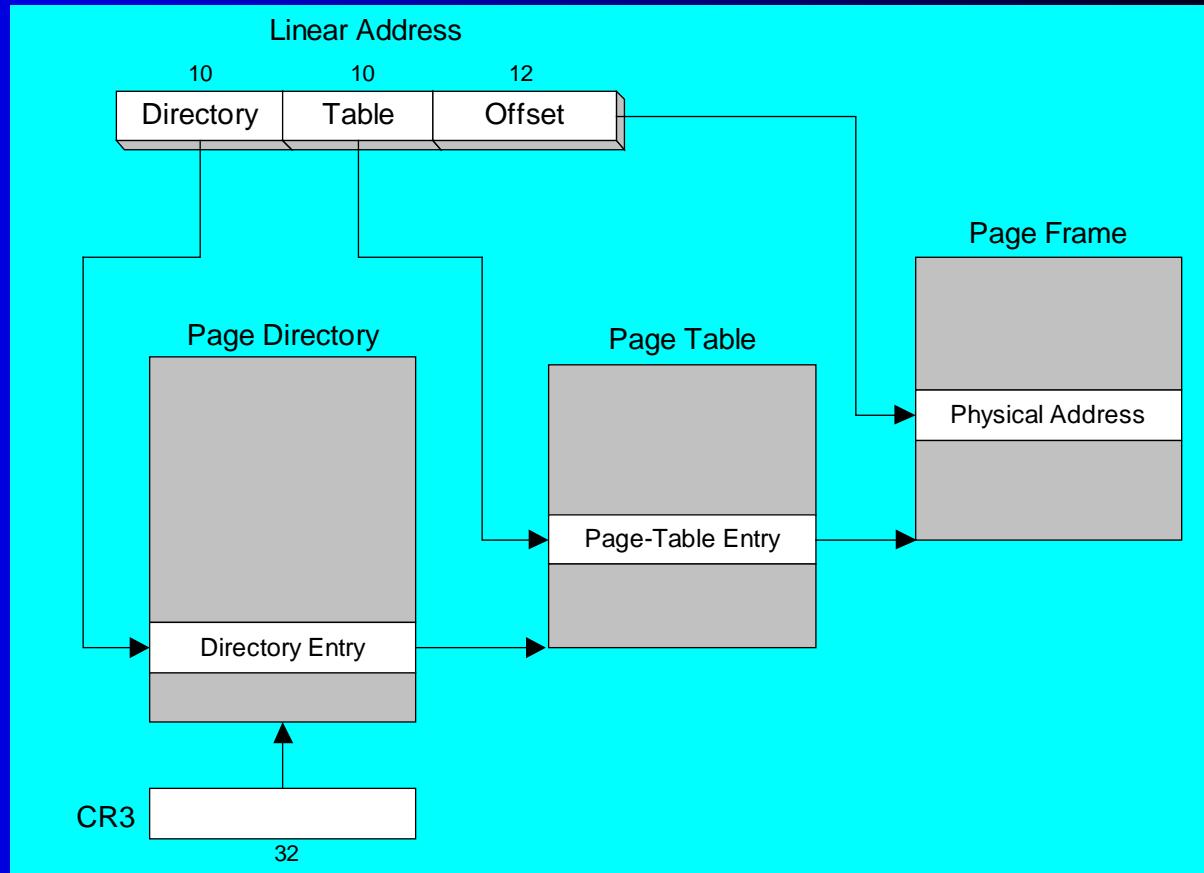
- Paging makes it possible for a computer to run a combination of programs that would not otherwise fit into memory.
- Only part of a program must be kept in memory, while the remaining parts are kept on disk.
- The memory used by the program is divided into small units called pages.
- As the program runs, the processor selectively unloads inactive pages from memory and loads other pages that are immediately required.

Paging (2 of 2)

- OS maintains page directory and page tables
- Page translation: CPU converts the linear address into a physical address
- Page fault: occurs when a needed page is not in memory, and the CPU interrupts the program
- OS copies the page into memory, program resumes execution

Page Translation

A linear address is divided into a page directory field, page table field, and page frame offset. The CPU uses all three to calculate the physical address.



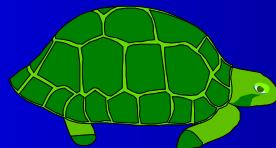
Review Questions

1. Define the following terms:
 - a. Multitasking.
 - b. Segmentation.
2. Define the following terms:
 - a. Segment selector
 - b. Logical address
3. (True/False): A segment selector points to an entry in a segment descriptor table.
4. (True/False): A segment descriptor contains the base location of a segment.
5. (True/False): A segment selector is 32 bits.
6. (True/False): A segment descriptor does not contain segment size information.
7. Describe a linear address.
8. How does paging relate to linear memory?

Summary

- 32-bit console programs
 - read from the keyboard and write plain text to the console window using Win32 API functions
- Important functions
 - ReadConsole, WriteConsole, GetStdHandle, ReadFile, WriteFile, CreateFile, CloseHandle, SetFilePointer
- Dynamic memory allocation
 - HeapAlloc, HeapFree
- x86 Memory management
 - segment selectors, linear address, physical address
 - segment descriptor tables
 - paging, page directory, page tables, page translation

The End



Assembly Language for x86 Processors

7th Edition

Kip R. Irvine

Chapter 12: Floating-Point Processing and Instruction Encoding

Slide show prepared by the author

Revision date: 1/15/2014

Chapter Overview

- **Floating-Point Binary Representation**
- Floating-Point Unit
- x86 Instruction Encoding

Floating-Point Binary Representation

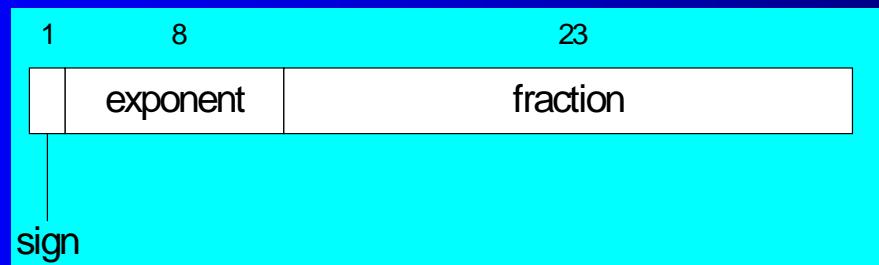
- IEEE Floating-Point Binary Reals
- The Exponent
- Normalized Binary Floating-Point Numbers
- Creating the IEEE Representation
- Converting Decimal Fractions to Binary Reals

IEEE Floating-Point Binary Reals

- Types
 - Single Precision
 - 32 bits: 1 bit for the sign, 8 bits for the exponent, and 23 bits for the fractional part of the significand.
 - Double Precision
 - 64 bits: 1 bit for the sign, 11 bits for the exponent, and 52 bits for the fractional part of the significand.
 - Double Extended Precision
 - 80 bits: 1 bit for the sign, 16 bits for the exponent, and 63 bits for the fractional part of the significand.

Single-Precision Format

Approximate normalized range: 2^{-126} to 2^{127} .
Also called a *short real*.



Components of a Single-Precision Real

- Sign
 - 1 = negative, 0 = positive
- Significand
 - decimal digits to the left & right of decimal point
 - weighted positional notation
 - Example:
$$123.154 = (1 \times 10^2) + (2 \times 10^1) + (3 \times 10^0) + (1 \times 10^{-1}) + (5 \times 10^{-2}) + (4 \times 10^{-3})$$
- Exponent
 - unsigned integer
 - integer bias (127 for single precision)

Decimal Fractions vs Binary Floating-Point

Binary Floating-Point	Base 10 Fraction
11.11	3 3/4
101.0011	5 3/16
1101.100101	13 37/64
0.00101	5/32
1.011	1 3/8
0.0000000000000000000000000001	1/8388608

Table 17-3 Binary and Decimal Fractions.

Binary	Decimal Fraction	Decimal Value
.1	1/2	.5
.01	1/4	.25
.001	1/8	.125
.0001	1/16	.0625
.00001	1/32	.03125

The Exponent

- Sample Exponents represented in Binary
- Add 127 to actual exponent to produce the biased exponent

Exponent (E)	Biased (E + 127)	Binary
+5	132	10000100
0	127	01111111
-10	117	01110101
+127	254	11111110
-126	1	00000001
-1	126	01111110

Normalizing Binary Floating-Point Numbers

- Mantissa is normalized when a single 1 appears to the left of the binary point
- Unnormalized: shift binary point until exponent is zero
- Examples

Unnormalized	Normalized
1110.1	1.1101×2^3
.000101	1.01×2^{-4}
1010001.	1.010001×2^6

Real-Number Encodings

- Normalized finite numbers
 - all the nonzero finite values that can be encoded in a normalized real number between zero and infinity
- Positive and Negative Infinity
- NaN (not a number)
 - bit pattern that is not a valid FP value
 - Two types:
 - quiet
 - signaling

Real-Number Encodings (*cont*)

- Specific encodings (single precision):

Value	Sign, Exponent, Significand		
Positive zero	0	00000000	00000000000000000000000000000000
Negative zero	1	00000000	00000000000000000000000000000000
Positive infinity	0	11111111	00000000000000000000000000000000
Negative infinity	1	11111111	00000000000000000000000000000000
QNaN	x	11111111	1xxxxxxxxxxxxxxxxxxxxxx
SNaN	x	11111111	0xxxxxxxxxxxxxxxxxxxxxx ^a

Examples (Single Precision)

- Order: sign bit, exponent bits, and fractional part (mantissa)

Binary Value	Biased Exponent	Sign, Exponent, Fraction		
-1.11	127	1	01111111	11000000000000000000000000000000
+1101.101	130	0	10000010	10110100000000000000000000000000
-.00101	124	1	01111100	01000000000000000000000000000000
+100111.0	132	0	10000100	00111000000000000000000000000000
+.0000001101011	120	0	01111000	10101100000000000000000000000000

Converting Fractions to Binary Reals

- Express as a sum of fractions having denominators that are powers of 2
- Examples

Decimal Fraction	Factored As...	Binary Real
1/2	1/2	.1
1/4	1/4	.01
3/4	1/2 + 1/4	.11
1/8	1/8	.001
7/8	1/2 + 1/4 + 1/8	.111
3/8	1/4 + 1/8	.011
1/16	1/16	.0001
3/16	1/8 + 1/16	.0011
5/16	1/4 + 1/16	.0101

Converting Single-Precision to Decimal

1. If the MSB is 1, the number is negative; otherwise, it is positive.
2. The next 8 bits represent the exponent. Subtract binary 01111111 (decimal 127), producing the unbiased exponent. Convert the unbiased exponent to decimal.
3. The next 23 bits represent the significand. Notate a “1.”, followed by the significand bits. Trailing zeros can be ignored. Create a floating-point binary number, using the significand, the sign determined in step 1, and the exponent calculated in step 2.
4. Unnormalize the binary number produced in step 3. (Shift the binary point the number of places equal to the value of the exponent. Shift right if the exponent is positive, or left if the exponent is negative.)
5. From left to right, use weighted positional notation to form the decimal sum of the powers of 2 represented by the floating-point binary number.

Example

Convert 0 10000010 101100000000000000000000 to Decimal

1. The number is positive.
2. The unbiased exponent is binary 00000011, or decimal 3.
3. Combining the sign, exponent, and significand, the binary number is +1.01011 X 2³.
4. The unnormalized binary number is +1010.11.
5. The decimal value is +10 3/4, or +10.75.

What's Next

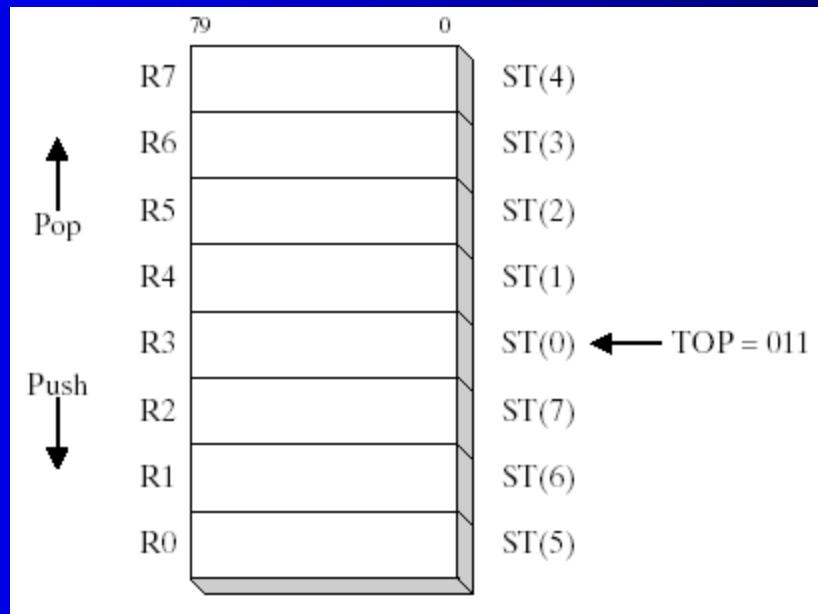
- Floating-Point Binary Representation
- **Floating-Point Unit**
- x86 Instruction Encoding

Floating Point Unit

- FPU Register Stack
- Rounding
- Floating-Point Exceptions
- Floating-Point Instruction Set
- Arithmetic Instructions
- Comparing Floating-Point Values
- Reading and Writing Floating-Point Values
- Exception Synchronization
- Mixed-Mode Arithmetic
- Masking and Unmasking Exceptions

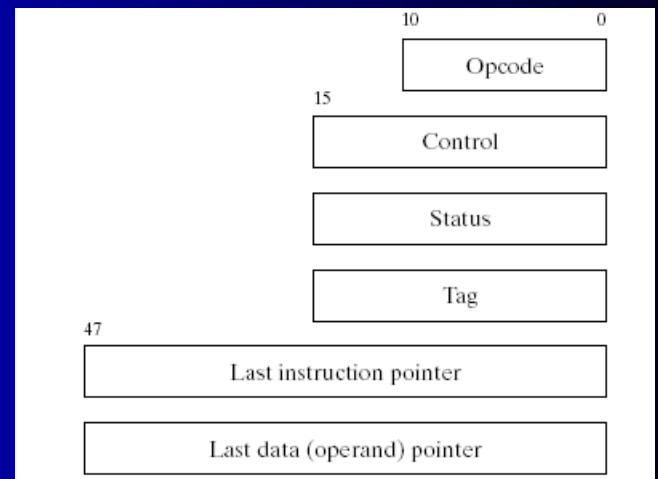
FPU Register Stack

- Eight individually addressable 80-bit data registers named R0 through R7
- Three-bit field named TOP in the FPU status word identifies the register number that is currently the top of stack.



Special-Purpose Registers

- Opcode register: stores opcode of last noncontrol instruction executed
- Control register: controls precision and rounding method for calculations
- Status register: top-of-stack pointer, condition codes, exception warnings
- Tag register: indicates content type of each register in the register stack
- Last instruction pointer register: pointer to last non-control executed instruction
- Last data (operand) pointer register: points to data operand used by last executed instruction



Rounding

- FPU attempts to round an infinitely accurate result from a floating-point calculation
 - may be impossible because of storage limitations
- Example
 - suppose 3 fractional bits can be stored, and a calculated value equals +1.0111.
 - rounding up by adding .0001 produces 1.100
 - rounding down by subtracting .0001 produces 1.011

Floating-Point Exceptions

- Six types of exception conditions
 - Invalid operation
 - Divide by zero
 - Denormalized operand
 - Numeric overflow
 - Inexact precision
- Each has a corresponding *mask* bit
 - if set when an exception occurs, the exception is handled automatically by FPU
 - if clear when an exception occurs, a software exception handler is invoked

FPU Instruction Set

- Instruction mnemonics begin with letter F
- Second letter identifies data type of memory operand
 - B = bcd
 - I = integer
 - no letter: floating point
- Examples
 - FLBD load binary coded decimal
 - FISTP store integer and pop stack
 - FMUL multiply floating-point operands

FPU Instruction Set

- Operands
 - zero, one, or two
 - no immediate operands
 - no general-purpose registers (EAX, EBX, ...)
 - integers must be loaded from memory onto the stack and converted to floating-point before being used in calculations
 - if an instruction has two operands, one must be a FPU register

FP Instruction Set

- Data Types

TABLE 17-11 Intrinsic Data Types.

Type	Usage
QWORD	64-bit integer
TBYTE	80-bit (10-byte) integer
REAL4	32-bit (4-byte) IEEE short real
REAL8	64-bit (8-byte) IEEE long real
REAL10	80-bit (10-byte) IEEE extended real

Load Floating-Point Value

- FLD
- copies floating point operand from memory into the top of the FPU stack, ST(0)
- Example

```
FLD m32fp  
FLD m64fp  
FLD m80fp  
FLD ST(i)
```

```
.data  
db1One    REAL8 234.56  
db1Two    REAL8 10.1  
.code  
fld db1One           ; ST(0) = db1One  
fld db1Two           ; ST(0) = db1Two, ST(1) = db1One
```

Store Floating-Point Value

- FST
 - copies floating point operand from the top of the FPU stack into memory
- FSTP
 - pops the stack after copying

```
FST m32fp  
FST m64fp  
FST ST(i)
```

Arithmetic Instructions

- Same operand types as FLD and FST

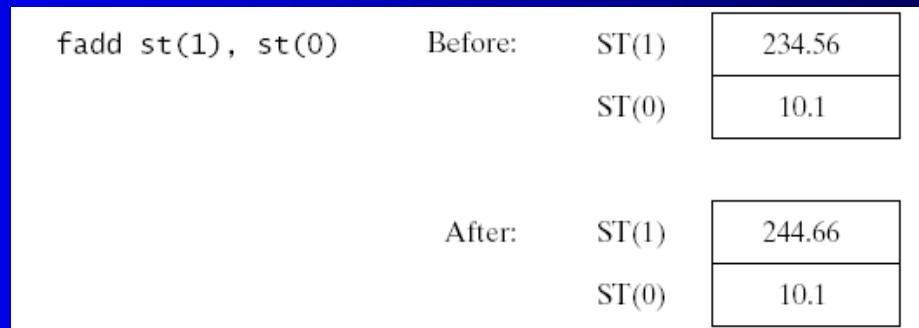
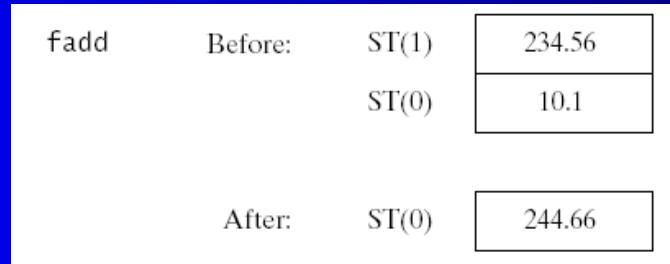
Table 17-12 Basic Floating-Point Arithmetic Instructions.

FCHS	Change sign
FADD	Add source to destination
FSUB	Subtract source from destination
FSUBR	Subtract destination from source
FMUL	Multiply source by destination
FDIV	Divide destination by source
FDIVR	Divide source by destination

Floating-Point Add

- FADD
 - adds source to destination
 - No-operand version pops the FPU stack after subtracting
- Examples:

FADD⁴
FADD *m32fp*
FADD *m64fp*



Floating-Point Subtract

- FSUB
 - subtracts source from destination.
 - No-operand version pops the FPU stack after subtracting
- Example:

```
FSUB5
FSUB m32fp
FSUB m64fp
FSUB ST(0), ST(i)
FSUB ST(i), ST(0)
```

<code>fsub mySingle</code>	<code>; ST(0) -= mySingle</code>
<code>fsub array[edi*8]</code>	<code>; ST(0) -= array[edi*8]</code>

Floating-Point Multiply

- FMUL
 - Multiplies source by destination, stores product in destination
- FDIV
 - Divides destination by source, then pops the stack

```
FMUL6
FMUL m32fp
FMUL m64fp
FMUL ST(0), ST(i)
FMUL ST(i), ST(0)
```

```
FDIV7
FDIV m32fp
FDIV m64fp
FDIV ST(0), ST(i)
FDIV ST(i), ST(0)
```

The no-operand versions of FMUL and FDIV pop the stack after multiplying or dividing.

Comparing FP Values

- FCOM instruction
- Operands:

Instruction	Description
FCOM	Compare ST(0) to ST(1)
FCOM <i>m32fp</i>	Compare ST(0) to <i>m32fp</i>
FCOM <i>m64fp</i>	Compare ST(0) to <i>m64fp</i>
FCOM ST(<i>i</i>)	Compare ST(0) to ST(<i>i</i>)

FCOM

- Condition codes set by FPU
 - codes similar to CPU flags

Condition	C3 (Zero Flag)	C2 (Parity Flag)	C0 (Carry Flag)	Conditional Jump to Use
ST(0) > SRC	0	0	0	JA, JNBE
ST(0) < SRC	0	0	1	JB, JNAE
ST(0) = SRC	1	0	0	JE, JZ
Unordered ^a	1	1	1	(None)

^aIf an invalid arithmetic operand exception is raised (because of invalid operands) and the exception is masked, C3, C2, and C0 are set according to the row marked *Unordered*.

Branching after FCOM

- Required steps:
 1. Use the FNSTSW instruction to move the FPU status word into AX.
 2. Use the SAHF instruction to copy AH into the EFLAGS register.
 3. Use JA, JB, etc to do the branching.

Fortunately, the FCOMI instruction does steps 1 and 2 for you.

```
fcomi ST(0), ST(1)
jnb    Label1
```

Comparing for Equality

- Calculate the absolute value of the difference between two floating-point values

```
.data
epsilon REAL8 1.0E-12          ; difference value
val2 REAL8 0.0                  ; value to compare
val3 REAL8 1.001E-13           ; considered equal to val2

.code
; if( val2 == val3 ), display "Values are equal".
    fld epsilon
    fld val2
    fsub val3
    fabs
    fcomi ST(0),ST(1)
    ja skip
    mWrite <"Values are equal",0dh,0ah>
skip:
```

Floating-Point I/O

- Irvine32 library procedures
 - ReadFloat
 - reads FP value from keyboard, pushes it on the FPU stack
 - WriteFloat
 - writes value from ST(0) to the console window in exponential format
 - ShowFPUStack
 - displays contents of FPU stack

Exception Synchronization

- Main CPU and FPU can execute instructions concurrently
 - if an unmasked exception occurs, the current FPU instruction is interrupted and the FPU signals an exception
 - But the main CPU does not check for pending FPU exceptions. It might use a memory value that the interrupted FPU instruction was supposed to set.
 - Example:

```
.data  
intValue DWORD 25  
  
.code  
fldl intValue ; load integer into ST(0)  
incl intValue ; increment the integer
```

Exception Synchronization

- (continued)
- For safety, insert a fwait instruction, which tells the CPU to wait for the FPU's exception handler to finish:

```
.data  
intValue DWORD 25  
.code  
fld intValue      ; load integer into ST(0)  
fwait             ; wait for pending exceptions  
inc intValue      ; increment the integer
```

FPU Code Example

expression: $\text{valD} = -\text{valA} + (\text{valB} * \text{valC})$.

```
.data
valA REAL8 1.5
valB REAL8 2.5
valC REAL8 3.0
valD REAL8 ?          ; will be +6.0

.code
fld valA              ; ST(0) = valA
fchs                 ; change sign of ST(0)
fld valB              ; load valB into ST(0)
fmul valC             ; ST(0) *= valC
fadd                 ; ST(0) += ST(1)
fstp valD             ; store ST(0) to valD
```

Mixed-Mode Arithmetic

- Combining integers and reals.
 - Integer arithmetic instructions such as ADD and MUL cannot handle reals
 - FPU has instructions that promote integers to reals and load the values onto the floating point stack.
- Example: $Z = N + X$

```
.data
N SDWORD 20
X REAL8 3.5
Z REAL8 ?

.code
fld N          ; load integer into ST(0)
fwait          ; wait for exceptions
fadd X         ; add mem to ST(0)
fstp Z         ; store ST(0) to mem
```

Masking and Unmasking Exceptions

- Exceptions are masked by default
 - Divide by zero just generates infinity, without halting the program
- If you unmask an exception
 - processor executes an appropriate exception handler
 - Unmask the divide by zero exception by clearing bit 2:

```
.data  
ctrlWord WORD ?  
  
.code  
fstcw ctrlWord          ; get the control word  
and ctrlWord,111111111111011b ; unmask divide by zero  
fldcw ctrlWord          ; load it back into FPU
```

What's Next

- Floating-Point Binary Representation
- Floating-Point Unit
- **x86 Instruction Encoding**

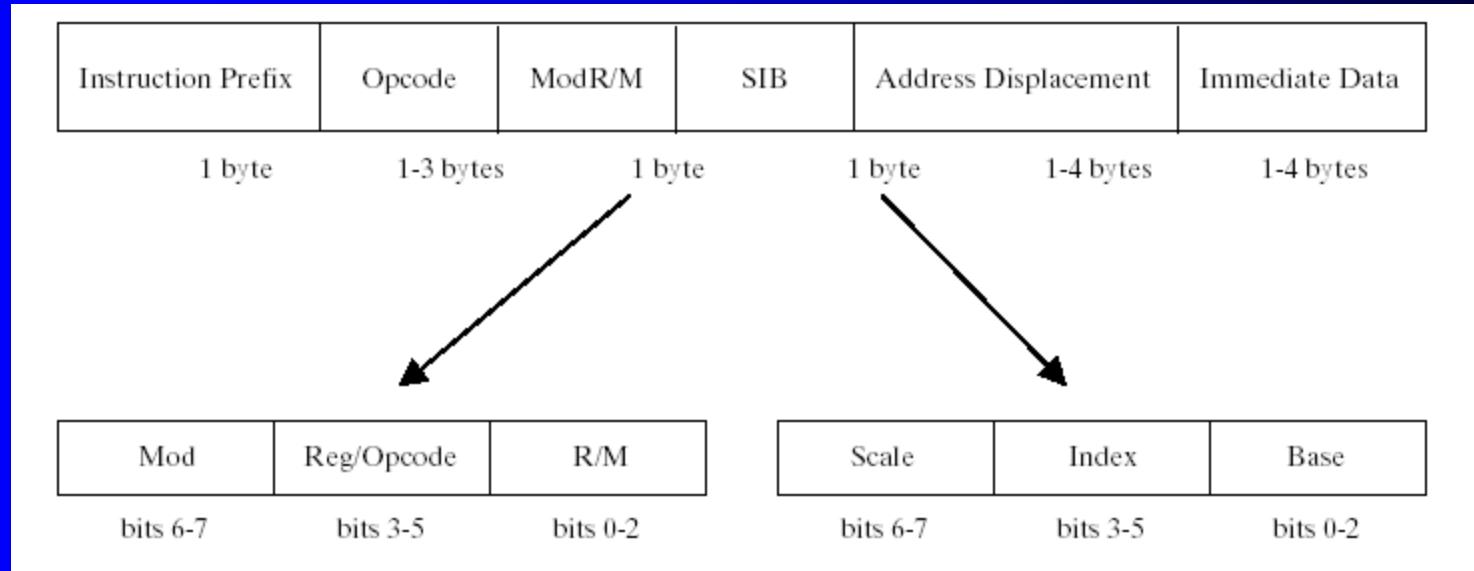
x86 Instruction Encoding

- x86 Instruction Format
- Single-Byte Instructions
- Move Immediate to Register
- Register-Mode Instructions
- x86 Processor Operand-Size Prefix
- Memory-Mode Instructions

x86 Instruction Format

- Fields
 - Instruction prefix byte (operand size)
 - opcode
 - Mod R/M byte (addressing mode & operands)
 - scale index byte (for scaling array index)
 - address displacement
 - immediate data (constant)
- Only the opcode is required

x86 Instruction Format



Single-Byte Instructions

- Only the opcode is used
- Zero operands
 - Example: AAA
- One implied operand
 - Example: INC DX

Move Immediate to Register

- Op code, followed by immediate value
- Example: move immediate to register
- Encoding format: $B8+rw\ dw$
 - ($B8$ = opcode, $+rw$ is a register number, dw is the immediate operand)
 - register number added to $B8$ to produce a new opcode

Table 17-21 Register Numbers (8/16 bit).

Register	Code
AX/AL	0
CX/CL	1
DX/DL	2
BX/BL	3
SP/AH	4
BP/CH	5
SI/DH	6
DI/BH	7

Register-Mode Instructions

- Mod R/M byte contains a 3-bit register number for each register operand
 - bit encodings for register numbers:

R/M	Register	R/M	Register
000	AX or AL	100	SP or AH
001	CX or CL	101	BP or CH
010	DX or DL	110	SI or DH
011	BX or BL	111	DI or BH

- Example: MOV AX, BX

mod	reg	r/m
11	011	000

x86 Operand Size Prefix

- Overrides default segment attribute (16-bit or 32-bit)
- Special value recognized by processor: 66h
- Intel ran out of opcodes for x86 processors
 - needed backward compatibility with 8086
- On x86 system, prefix byte used when 16-bit operands are used

x86 Operand Size Prefix

- Sample encoding for 16-bit target:

```
.model small  
.286  
.stack 100h  
.code  
main PROC  
    mov    ax,dx          ; 8B C2  
    mov    al,d1          ; 8A C2
```

- Encoding for 32-bit target:

```
.model small  
.386  
.stack 100h  
.code  
main PROC  
    mov    eax,edx        ; 8B C2  
    mov    ax,dx          ; 66 8B C2  
    mov    al,d1          ; 8A C2
```

overrides default
operand size

Memory-Mode Instructions

- Wide variety of operand types (addressing modes)
- 256 combinations of operands possible
 - determined by Mod R/M byte
- Mod R/M encoding:
 - mod = addressing mode
 - reg = register number
 - r/m = register or memory indicator

mod	reg	r/m
00	000	100

MOV Instruction Examples

- Selected formats for 8-bit and 16-bit MOV instructions:

Opcode	Instruction	Description
88/r	MOV eb,rb	Move byte register into EA byte
89/r	MOV ew,rw	Move word register into EA word
8A/r	MOV rb,eb	Move EA byte into byte register
8B/r	MOV rw,ew	Move EA word into word register
8C/0	MOV ew,ES	Move ES into EA word
8C/1	MOV ew,CS	Move CS into EA word
8C/2	MOV ew,SS	Move SS into EA word
8C/3	MOV DS,ew	Move DS into EA word
8E/0	MOV ES,mw	Move memory word into ES
8E/0	MOV ES,rw	Move word register into ES
8E/2	MOV SS,mw	Move memory word into SS

Sample MOV Instructions

Assume that `myWord` is located at offset 0102h.

Instruction	Machine Code	Addressing Mode
<code>mov ax,myWord</code>	A1 02 01	direct (optimized for AX)
<code>mov myWord,bx</code>	89 1E 02 01	direct
<code>mov [di],bx</code>	89 1D	indexed
<code>mov [bx+2],ax</code>	89 47 02	base-disp
<code>mov [bx+si],ax</code>	89 00	base-indexed
<code>mov word ptr [bx+di+2],1234h</code>	C7 41 02 34 12	base-indexed-disp

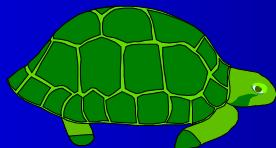
Summary

- Binary floating point number contains a sign, significand, and exponent
 - single precision, double precision, extended precision
- Not all significands between 0 and 1 can be represented correctly
 - example: 0.2 creates a repeating bit sequence
- Special types
 - Normalized finite numbers
 - Positive and negative infinity
 - NaN (not a number)

Summary - 2

- Floating Point Unit (FPU) operates in parallel with CPU
 - register stack: top is ST(0)
 - arithmetic with floating point operands
 - conversion of integer operands
 - floating point conversions
 - intrinsic mathematical functions
- x86 Instruction set
 - complex instruction set, evolved over time
 - backward compatibility with older processors
 - encoding and decoding of instructions

The End



Assembly Language for x86 Processors

7th Edition

Kip R. Irvine

Chapter 13: High-Level Language Interface

Slide show prepared by the author

Revision date: 1/15/2014

Chapter Overview

- **Introduction**
- **Inline Assembly Code**
- **Linking 32-Bit Assembly Language Code to C/C++**

Why Link ASM and HLL Programs?

- Use high-level language for overall project development
 - Relieves programmer from low-level details
- Use assembly language code
 - Speed up critical sections of code
 - Access nonstandard hardware devices
 - Write platform-specific code
 - Extend the HLL's capabilities

General Conventions

- Considerations when calling assembly language procedures from high-level languages:
 - Both must use the same **naming convention** (rules regarding the naming of variables and procedures)
 - Both must use the same **memory model**, with compatible segment names
 - Both must use the same **calling convention**

Calling Convention

- Identifies specific registers that must be preserved by procedures
- Determines how arguments are passed to procedures: in registers, on the stack, in shared memory, etc.
- Determines the order in which arguments are passed by calling programs to procedures
- Determines whether arguments are passed by value or by reference
- Determines how the stack pointer is restored after a procedure call
- Determines how functions return values

External Identifiers

- An **external identifier** is a name that has been placed in a module's object file in such a way that the linker can make the name available to other program modules.
- The linker resolves references to external identifiers, but can only do so if the same naming convention is used in all program modules.

What's Next

- Introduction
- **Inline Assembly Code**
- Linking 32-Bit Assembly Language Code to C/C++

Inline Assembly Code

- Assembly language source code that is inserted directly into a HLL program.
- Compilers such as Microsoft Visual C++ and Borland C++ have compiler-specific directives that identify inline ASM code.
- Efficient inline code executes quickly because CALL and RET instructions are not required.
- Simple to code because there are no external names, memory models, or naming conventions involved.
- Decidedly not portable because it is written for a single platform.

_asm Directive in Microsoft Visual C++

- Can be placed at the beginning of a single statement
- Or, It can mark the beginning of a block of assembly language statements
- Syntax:

```
__asm statement

__asm {
    statement-1
    statement-2
    ...
    statement-n
}
```

Commenting Styles

All of the following comment styles are acceptable, but the latter two are preferred:

```
mov    esi,buf      ; initialize index register
mov    esi,buf      // initialize index register
mov    esi,buf      /* initialize index register */
```

You Can Do the Following . . .

- Use any instruction from the Intel instruction set
- Use register names as operands
- Reference function parameters by name
- Reference code labels and variables that were declared outside the asm block
- Use numeric literals that incorporate either assembler-style or C-style radix notation
- Use the PTR operator in statements such as `inc BYTE PTR [esi]`
- Use the EVEN and ALIGN directives
- Use LENGTH, TYPE, and SIZE directives

You Cannot Do the Following . . .

- Use data definition directives such as DB, DW, or BYTE
- Use assembler operators other than PTR
- Use STRUCT, RECORD, WIDTH, and MASK
- Use the OFFSET operator (but LEA is ok)
- Use macro directives such as MACRO, REPT, IRC, IRP
- Reference segments by name.
 - (You can, however, use segment register names as operands.)

Register Usage

- In general, you can modify EAX, EBX, ECX, and EDX in your inline code because the compiler does not expect these values to be preserved between statements
- Conversely, always save and restore ESI, EDI, and EBP.

See the Inline Test demonstration program.

File Encryption Example

- Reads a file, encrypts it, and writes the output to another file.
- The TranslateBuffer function uses an `__asm` block to define statements that loop through a character array and XOR each character with a predefined value.

[View the Encode2.cpp program listing](#)

What's Next

- Introduction
- Inline Assembly Code
- **Linking 32-Bit Assembly Language Code to C/C++**

Linking Assembly Language to Visual C++

- Basic Structure - Two Modules
 - The first module, written in assembly language, contains the external procedure
 - The second module contains the C/C++ code that starts and ends the program
- The C++ module adds the **extern** qualifier to the external assembly language function prototype.
- The "C" specifier must be included to prevent name decoration by the C++ compiler:

```
extern "C" functionName( parameterList );
```

Name Decoration

HLL compilers do this to uniquely identify overloaded functions. A function such as:

```
int ArraySum( int * p, int count )
```

would be exported as a decorated name that encodes the return type, function name, and parameter types. For example:

```
int_ArraySum_pInt_int
```

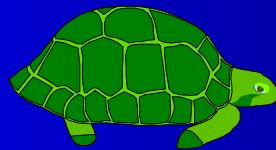
The problem with name decoration is that the C++

C++ compilers vary in the way they decorate function names.

Summary

- Use assembly language top optimize sections of applications written in high-level languages
 - inline asm code
 - linked procedures
- Naming conventions, name decoration
- Calling convention determined by HLL program
- OK to call C functions from assembly language

The End



Assembly Language for x86 Processors

7th Edition

Kip R. Irvine

Chapter 14: 16-Bit MS-DOS Programming

Slide show prepared by the author

Revision date: 1/15/2014

Chapter Overview

- **MS-DOS and the IBM-PC**
- MS-DOS Function Calls (INT 21h)
- Standard MS-DOS File I/O Services

MS-DOS and the IBM-PC

- Real-Address Mode
- MS-DOS Memory Organization
- MS-DOS Memory Map
- Redirecting Input-Output
- Software Interrupts
- INT Instruction
- Interrupt Vectoring Process
- Common Interrupts

Real-Address Mode

- Real-address mode (16-bit mode) programs have the following characteristics:
 - Max 1 megabyte addressable RAM
 - Single tasking
 - No memory boundary protection
 - Offsets are 16 bits
- IBM PC-DOS: first Real-address OS for IBM-PC
 - Has roots in Gary Kildall's highly successful Digital Research CP/M
 - Later renamed to MS-DOS, owned by Microsoft

MS-DOS Memory Organization

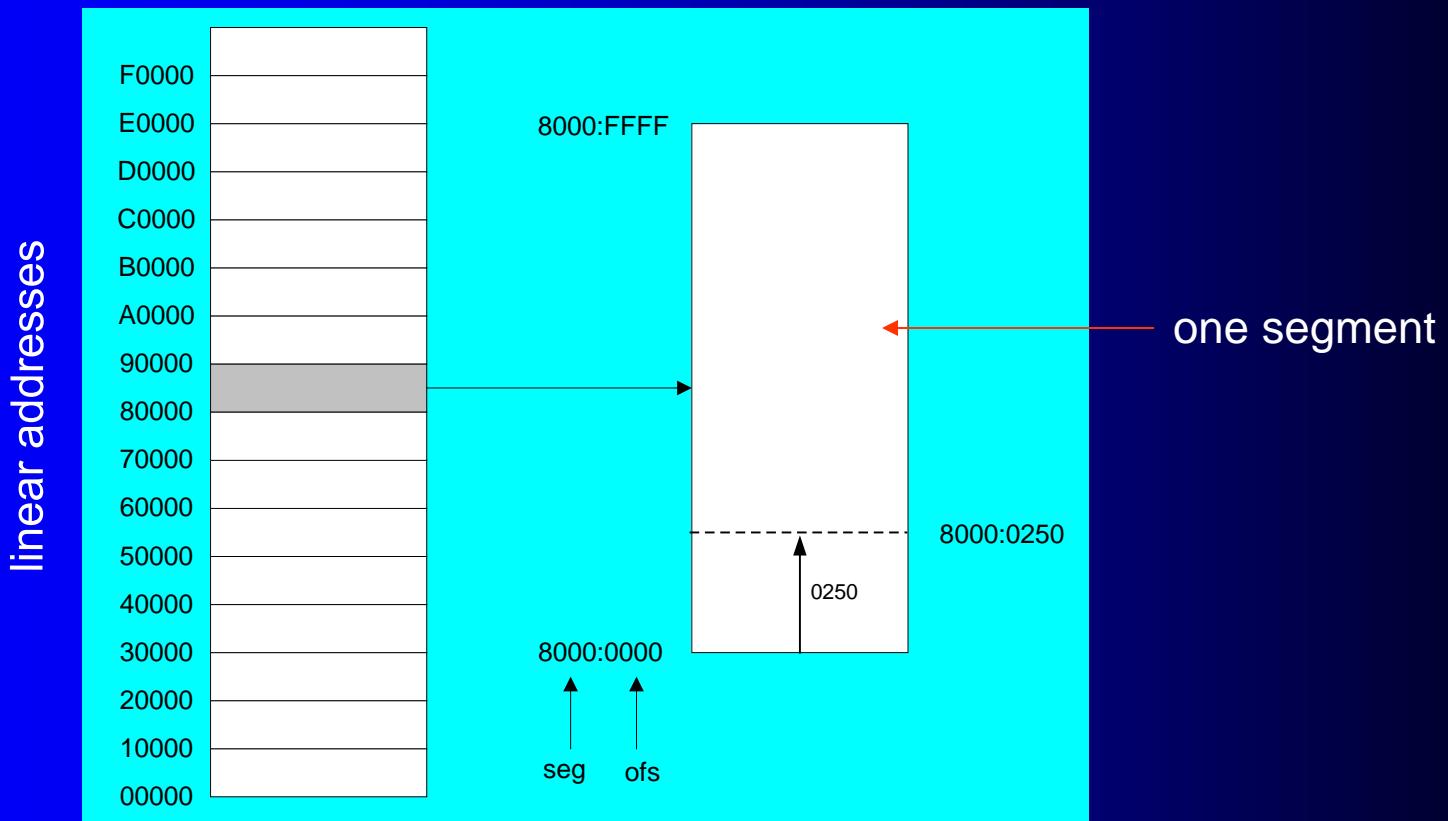
- Interrupt Vector Table
- BIOS & DOS data
- Software BIOS
- MS-DOS kernel
- Resident command processor
- Transient programs
- Video graphics & text
- Reserved (device controllers)
- ROM BIOS

Real-Address mode

- 1 MB RAM maximum addressable
- Application programs can access any area of memory
- Single tasking
- Supported by MS-DOS operating system

Segmented Memory

Segmented memory addressing: absolute (linear) address is a combination of a 16-bit segment value added to a 16-bit offset



Calculating Linear Addresses

- Given a segment address, multiply it by 16 (add a hexadecimal zero), and add it to the offset
- Example: convert 08F1:0100 to a linear address

Adjusted Segment value: 0 8 F 1 0

Add the offset: 0 1 0 0

Linear address: 0 9 0 1 0

Your turn . . .

What linear address corresponds to the segment/offset address 028F:0030?

$$028F0 + 0030 = 02920$$

Always use hexadecimal notation for addresses.

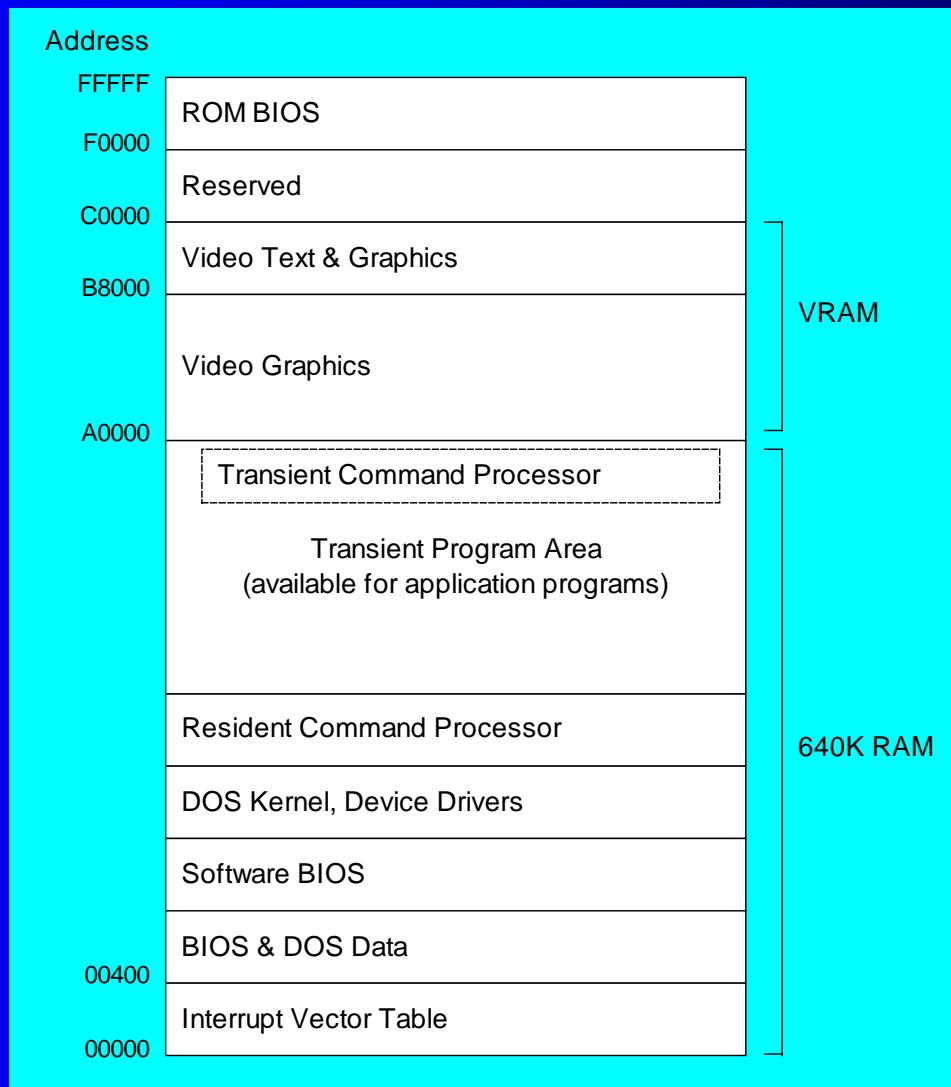
Your turn . . .

What segment addresses correspond to the linear address 28F30h?

Many different segment-offset addresses can produce the linear address 28F30h. For example:

28F0:0030, 28F3:0000, 28B0:0430, . . .

MS-DOS Memory Map



Redirecting Input-Output (1 of 2)

- Input-output devices and files are interchangeable
- Three primary types of I/O:
 - Standard input (console, keyboard)
 - Standard output (console, display)
 - Standard error (console, display)
- Symbols borrowed from Unix:
 - < symbol: *get input from*
 - > symbol: *send output to*
 - | symbol: pipe output from one process to another
- Predefined device names:
 - PRN, CON, LPT1, LPT2, NUL, COM1, COM2

Redirecting Input-Output (2 of 2)

- Standard input, standard output can both be redirected
- Standard error cannot be redirected
- Suppose we have created a program named `myprog.exe` that reads from standard input and writes to standard output. Following are MS-DOS commands that demonstrate various types of redirection:

```
myprog < infile.txt
```

```
myprog > outfile.txt
```

```
myprog < infile.txt > outfile.txt
```

INT Instruction

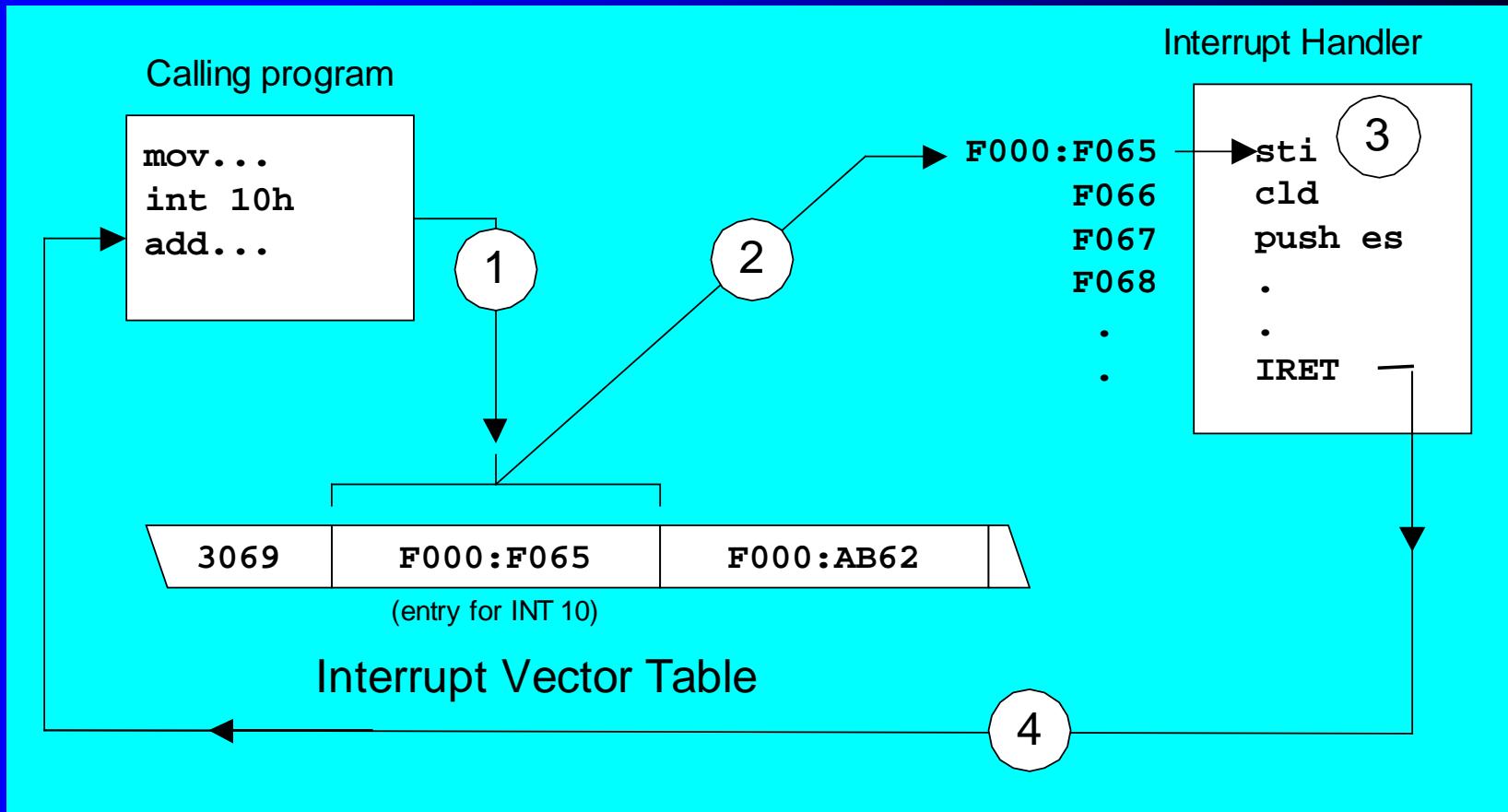
- The INT instruction executes a software interrupt.
- The code that handles the interrupt is called an interrupt handler.
- Syntax:

```
INT number
      (number = 0..FFh)
```

The Interrupt Vector Table (IVT) holds a 32-bit segment-offset address for each possible interrupt handler.

Interrupt Service Routine (ISR) is another name for interrupt handler.

Interrupt Vectoring Process



Common Interrupts

- INT 10h Video Services
- INT 16h Keyboard Services
- INT 17h Printer Services
- INT 1Ah Time of Day
- INT 1Ch User Timer Interrupt
- INT 21h MS-DOS Services

What's Next

- MS-DOS and the IBM-PC
- **MS-DOS Function Calls (INT 21h)**
- Standard MS-DOS File I/O Services

MS-DOS Function Calls (INT 21h)

- ASCII Control Characters
- Selected Output Functions
- Selected Input Functions
- Example: String Encryption
- Date/Time Functions

INT 4Ch: Terminate Process

- Ends the current process (program), returns an optional 8-bit return code to the calling process.
- A return code of 0 usually indicates successful completion.

```
mov ah,4Ch          ; terminate process
mov al,0            ; return code
int 21h
```

; Same as:

```
.EXIT 0
```

Selected Output Functions

- ASCII control characters
- 02h, 06h - Write character to standard output
- 05h - Write character to default printer
- 09h - Write string to standard output
- 40h - Write string to file or device

ASCII Control Characters

Many INT 21h functions act upon the following control characters:

- 08h - Backspace (moves one column to the left)
- 09h - Horizontal tab (skips forward n columns)
- 0Ah - Line feed (moves to next output line)
- 0Ch - Form feed (moves to next printer page)
- 0Dh - Carriage return (moves to leftmost output column)
- 1Bh - Escape character

INT 21h Functions 02h and 06h: Write Character to Standard Output

Write the letter 'A' to standard output:

```
mov ah,02h  
mov dl,'A'  
int 21h
```

or: mov ah,2

Write a backspace to standard output:

```
mov ah,06h  
mov dl,08h  
int 21h
```

INT 21h Function 05h: Write Character to Default Printer

Write the letter 'A':

```
mov ah,05h  
mov dl,65  
int 21h
```

Write a horizontal tab:

```
mov ah,05h  
mov dl,09h  
int 21h
```

INT 21h Function 09h: Write String to Standard Output

- The string must be terminated by a '\$' character.
- DS must point to the string's segment, and DX must contain the string's offset:

```
.data  
string BYTE "This is a string$"  
  
.code  
mov ah,9  
mov dx,OFFSET string  
int 21h
```

INT 21h Function 40h: Write String to File or Device

Input: BX = file or device handle (console = 1), CX = number of bytes to write, DS:DX = address of array

```
.data  
message "Writing a string to the console"  
bytesWritten WORD ?  
  
.code  
    mov ah,40h  
    mov bx,1  
    mov cx,LENGTHOF message  
    mov dx,OFFSET message  
    int 21h  
    mov bytesWritten,ax
```

Selected Input Functions

- 01h, 06h - Read character from standard input
- 0Ah - Read array of buffered characters from standard input
- 0Bh - Get status of the standard input buffer
- 3Fh - Read from file or device

INT 21h Function 01h:

Read single character from standard input

- Echoes the input character
- Waits for input if the buffer is empty
- Checks for Ctrl-Break (^C)
- Acts on control codes such as horizontal Tab

```
.data  
char BYTE ?  
.code  
mov ah,01h  
int 21h  
mov char,al
```

INT 21h Function 06h:

Read character from standard input without waiting

- Does not echo the input character
- Does not wait for input (use the Zero flag to check for an input character)
- Example: repeats loop until a character is pressed.

```
.data
char BYTE ?
.code
L1: mov ah,06h          ; keyboard input
    mov dl,0FFh         ; don't wait for input
    int 21h
    jz L1               ; no character? repeat loop
    mov char,al          ; character pressed: save it
    call DumpRegs        ; display registers
```

INT 21h Function 0Ah:

Read buffered array from standard input (1 of 2)

- Requires a predefined structure to be set up that describes the maximum input size and holds the input characters.
- Example:

```
count = 80
```

```
KEYBOARD STRUCT
    maxInput BYTE count          ; max chars to input
    inputCount BYTE ?            ; actual input count
    buffer BYTE count DUP(?)    ; holds input chars
KEYBOARD ENDS
```

INT 21h Function 0Ah (2 of 2)

Executing the interrupt:

```
.data  
kybdData KEYBOARD <>  
  
.code  
    mov ah,0Ah  
    mov dx,OFFSET kybdData  
    int 21h
```

INT 21h Function 0Bh:

Get status of standard input buffer

- Can be interrupted by Ctrl-Break (^C)
- Example: loop until a key is pressed. Save the key in a variable:

```
L1: mov ah,0Bh      ; get buffer status
    int 21h
    cmp al,0      ; buffer empty?
    je L1         ; yes: loop again
    mov ah,1      ; no: input the key
    int 21h
    mov char,al   ; and save it
```

Example: String Encryption

Reads from standard input, encrypts each byte, writes to standard output.

```
XORVAL = 239           ; any value between 0-255
.code
main PROC
    mov  ax,@data
    mov  ds,ax
L1:   mov  ah,6          ; direct console input
    mov  dl,0FFh         ; don't wait for character
    int  21h             ; AL = character
    jz   L2              ; quit if ZF = 1 (EOF)
    xor  al,XORVAL
    mov  ah,6          ; write to output
    mov  dl,al
    int  21h
    jmp  L1              ; repeat the loop
L2:   exit
```

INT 21h Function 3Fh:

Read from file or device

- Reads a block of bytes.
- Can be interrupted by Ctrl-Break (^C)
- Example: Read string from keyboard:

```
.data
inputBuffer BYTE 127 dup(0)
bytesRead WORD ?

.code
mov ah,3Fh
mov bx,0           ; keyboard handle
mov cx,127         ; max bytes to read
mov dx,OFFSET inputBuffer ; target location
int 21h
mov bytesRead,ax   ; save character count
```

Date/Time Functions

- 2Ah - Get system date
- 2Bh - Set system date *
- 2Ch - Get system time
- 2Dh - Set system time *

* may be restricted by your user profile if running a console window under Windows NT, 2000, and XP.

INT 21h Function 2Ah: Get system date

- Returns year in CX, month in DH, day in DL, and day of week in AL

```
mov ah,2Ah  
int 21h  
mov year,cx  
mov month,dh  
mov day,dl  
mov dayOfWeek,al
```

INT 21h Function 2Bh: Set system date

- Sets the system date. AL = 0 if the function was not successful in modifying the date.

```
mov ah, 2Bh  
mov cx, year  
mov dh, month  
mov dl, day  
int 21h  
cmp al, 0  
jne failed
```

INT 21h Function 2Ch: Get system time

- Returns hours (0-23) in CH, minutes (0-59) in CL, and seconds (0-59) in DH, and hundredths (0-99) in DL.

```
mov ah,2Ch  
int 21h  
mov hours,cl  
mov minutes,cl  
mov seconds,dh
```

INT 21h Function 2Dh: Set system time

- Sets the system date. AL = 0 if the function was not successful in modifying the time.

```
mov ah,2Dh  
mov ch,hours  
mov cl,minutes  
mov dh,seconds  
int 21h  
cmp al,0  
jne failed
```

Example: Displaying the Date and Time

- Displays the system date and time, using INT 21h Functions 2Ah and 2Ch.
- Demonstrates simple date formatting
- [View the source code](#)
- Sample output:

```
Date: 12-8-2001,    Time: 23:01:23
```

To Do: write a procedure named ShowDate that displays any date in mm-dd-yyyy format.

What's Next

- MS-DOS and the IBM-PC
- MS-DOS Function Calls (INT 21h)
- **Standard MS-DOS File I/O Services**

Standard MS-DOS File I/O Services

- 716Ch - Create or open file
- 3Eh - Close file handle
- 42h - Move file pointer
- 5706h - Get file creation date and time
- Selected Irvine16 Library Procedures
- Example: Read and Copy a Text File
- Reading the MS-DOS Command Tail
- Example: Creating a Binary File

INT 21h Function 716Ch: Create or open file

- AX = 716Ch
- BX = access mode (0 = read, 1 = write, 2 = read/write)
- CX = attributes (0 = normal, 1 = read only, 2 = hidden,
3 = system, 8 = volume ID, 20h = archive)
- DX = action (1 = open, 2 = truncate, 10h = create)
- DS:SI = segment/offset of filename
- DI = alias hint (optional)

Example: Create a New File

```
mov  ax,716Ch          ; extended open/create
mov  bx,2              ; read-write
mov  cx,0              ; normal attribute
mov  dx,10h + 02h      ; action: create + truncate
mov  si,OFFSET Filename
int  21h
jc   failed
mov  handle,ax         ; file handle
mov  actionTaken,cx    ; action taken to open file
```

Example: Open an Existing File

```
mov  ax,716Ch          ; extended open/create
mov  bx,0              ; read-only
mov  cx,0              ; normal attribute
mov  dx,1              ; open existing file
mov  si,OFFSET Filename
int  21h
jc   failed
mov  handle,ax          ; file handle
mov  actionTaken,cx      ; action taken to open file
```

INT 21h Function 3Eh: Close file handle

- Use the same file handle that was returned by INT 21h when the file was opened.
- Example:

```
.data  
filehandle WORD ?  
.code  
    mov  ah,3Eh  
    mov  bx,filehandle  
    int  21h  
    jc   failed
```

INT 21h Function 42h: Move file pointer

Permits random access to a file (text or binary).

```
mov    ah,42h
mov    al,0          ; offset from beginning
mov    bx,handle
mov    cx,offsetHi
mov    dx,offsetLo
int    21h
```

AL indicates how the pointer's offset is calculated:

- 0: Offset from the beginning of the file
- 1: Offset from the current pointer location
- 2: Offset from the end of the file

INT 21h Function 5706h: Get file creation date and time

- Obtains the date and time when a file was created (not necessarily the same date and time when the file was last modified or accessed.)

```
mov ax,5706h
mov bx,handle          ; handle of open file
int 21h
jc error
mov date,dx
mov time,cx
mov milliseconds,si
```

Selected Irvine16 Library Procedures

- 16-Bit ReadString procedure
- 16-Bit WriteString procedure

ReadString Procedure

The ReadString procedure from the Irvine16 library reads a string from standard input and returns a null-terminated string. When calling it, pass a pointer to a buffer in DX. Pass a count of the maximum number of characters to input, plus 1, in CX. Writestring inputs the string from the user, returning when either of the following events occurs:

- 1.CX –1 characters were entered.
- 2.The user pressed the Enter key.

```
.data  
buffer BYTE 20 DUP(?)  
.code  
mov dx,OFFSET buffer  
mov cx,LENGTHOF buffer  
call ReadString
```

ReadString Procedure

You can also call it using 32-bit registers:

```
.data  
buffer BYTE 20 DUP(?)  
.code  
mov edx,OFFSET buffer  
mov ecx,LENGTHOF buffer  
call ReadString
```

ReadString returns a count of the number of characters actually read in the EAX register.

ReadString Implementation

```
ReadString PROC
    push cx          ; save registers
    push si
    push cx          ; save character count
    mov  si,dx        ; point to input buffer
    dec  cx          ; save room for null byte
L1:  mov  ah,1        ; function: keyboard input
    int  21h        ; returns character in AL
    cmp  al,0Dh      ; end of line?
    je   L2          ; yes: exit
    mov  [si],al      ; no: store the character
    inc  si          ; increment buffer pointer
    loop L1          ; loop until CX=0
L2:  mov  BYTE PTR [si],0 ; insert null byte
    pop  ax          ; original digit count
    sub  ax,cx        ; AX = size of input string
    pop  si          ; restore registers
    pop  cx
    ret
ReadString ENDP          ; returns AX = size of string
```

16-Bit WriteString Procedure

Receives: DX contains the offset of a null-terminated string.

```
WriteString PROC
    pusha
    INVOKE Str_length,dx      ; AX = string length
    mov   cx,ax                ; CX = number of bytes
    mov   ah,40h                ; write to file or device
    mov   bx,1                  ; standard output handle
    int   21h                  ; call MS-DOS
    popa
    ret
WriteString ENDP
```

(May be different from the version printed on page 482.)

Example: Read and Copy a Text File

- The *Readfile.asm* program demonstrates several INT 21h functions:
 - Function 716Ch: Create new file or open existing file
 - Function 3Fh: Read from file or device
 - Function 40h: Write to file or device
 - Function 3Eh: Close file handle

[View the source code](#)

Reading the MS-DOS Command Tail

- When a program runs, any additional text on its command line is automatically stored in the 128-byte MS-DOS command tail area, at offset 80h in the program segment prefix (PSP).
- Example: run a program named attr.exe and pass it "FILE1.DOC" as the command tail:

Offset:	80	81	82	83	84	85	86	87	88	89	8A	8B
Contents:	0A	20	46	49	4C	45	31	2E	44	4F	43	0D
	F	I	L	E	1	.	D	O	C			

[View the Get_CommandTail library procedure source code.](#)

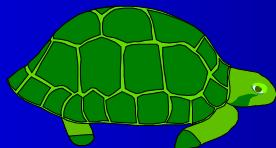
Example: Creating a Binary File

- A **binary file** contains fields that are generally not recognizable when displayed on the screen.
- Advantage: Reduces I/O processing time
 - Example: translating a 5-digit ASCII integer to binary causes approximately 100 instructions to execute.
- Disadvantage: may require more disk space
 - Example: array of 4 doublewords:
 - "795 43 1234 2" - requires 13 bytes in ASCII
 - requires 16 bytes in binary

Summary

- MS-DOS applications
 - 16-bit segments, segmented addressing, running in real-address mode
 - complete access to memory and hardware
- Software interrupts
 - processed by interrupt handlers
 - INT (call to interrupt procedure) instruction
 - pushes flags & return address on the stack
 - uses interrupt vector table to find handler
- Program Segment Prefix (PSP)
- BIOS Services (INT 10h, INT 16h, INT 17h, ...)
- MS-DOS Services (INT 21h)

The End



Assembly Language for x86 Processors

7th Edition

Kip R. Irvine

Chapter 15: Disk Fundamentals

Slide show prepared by the author

Revision date: 1/15/2014

Chapter Overview

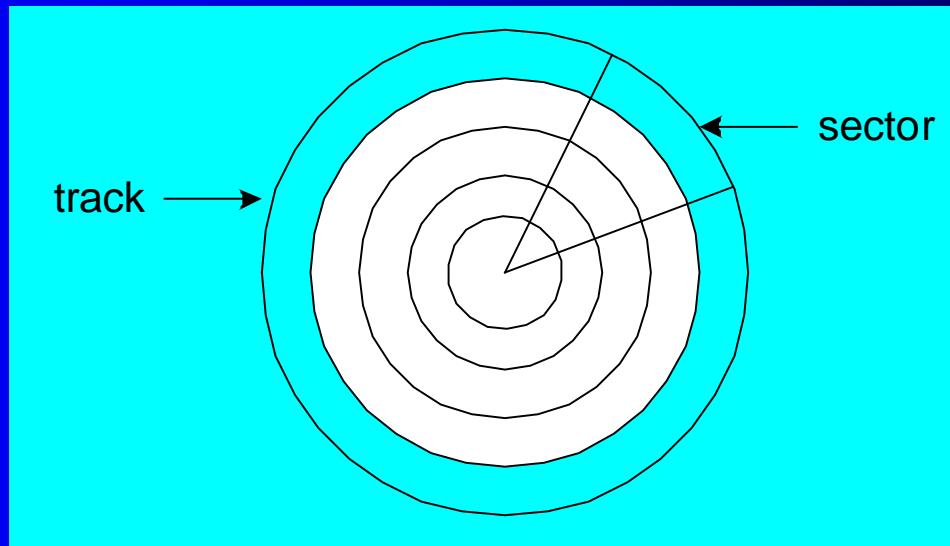
- **Disk Storage Systems**
- File Systems
- Disk Directory
- Reading and Writing Disk Sectors (7305h)
- System-Level File Functions

Disk Storage Systems

- Tracks, Cylinders, and Sectors
- Disk Partitions (Volumes)

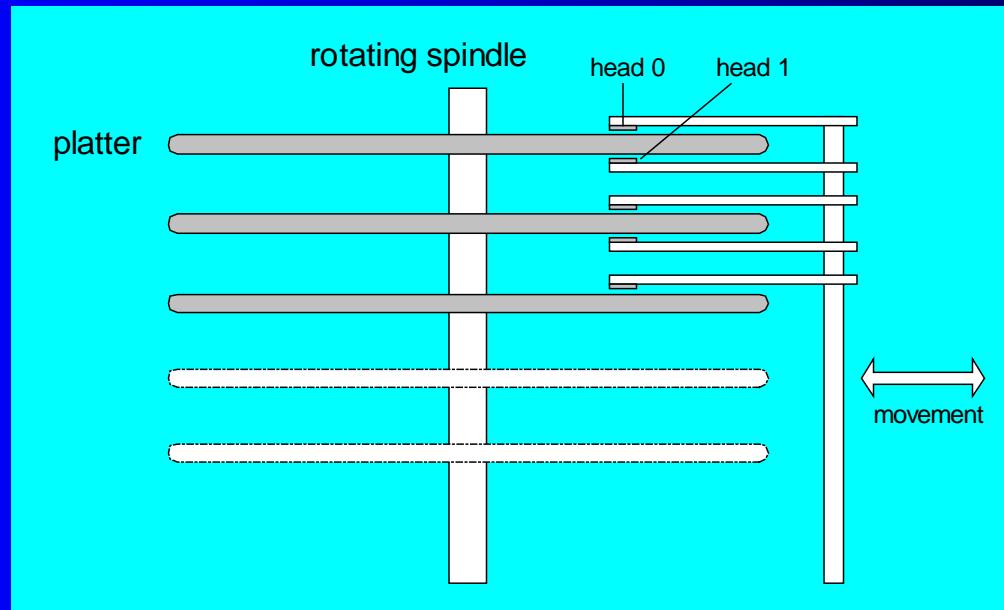
Tracks and Sectors

- Physical disk geometry - a way of describing the disk's structure to make it readable by the system BIOS
- Track - concentric circle containing data
- Sector - part of a track



Cylinders and Seeking

- Cylinder - all tracks readable from one head position
- Seek - move read/write heads between tracks



Disk Formatting

- Physical formatting
 - aka low-level formatting
 - Usually done at the factory.
 - Must be done before logical formatting
 - Defines the tracks, sectors, and cylinders
- Logical formatting
 - Permits disk to be accessed using sequentially numbered logical sectors
 - Installs a file system (ex: NTFS)
 - May install an operating system

Fragmentation

- A **fragmented file** is one whose sectors are no longer located in contiguous areas of the disk.
 - causes read/write heads to skip
 - slower file access
 - possible read/write errors



Translation

- Translation - conversion of physical disk geometry to a sequence of logical sector numbers
- Performed by a hard disk controller (firmware)
- Logical sector numbers are numbered sequentially, have no direct mapping to hardware

Disk Partitions

- Logical units that divide a physical hard disk
 - Also called volumes
- Primary partition
 - Up to four permitted
 - Each may boot a different OS
- Extended partition
 - Maximum of one permitted
 - May be divided into multiple logical partitions, each with a different drive letter
- Primary and Extended
 - Up to three primary and one extended

Logical Partitions

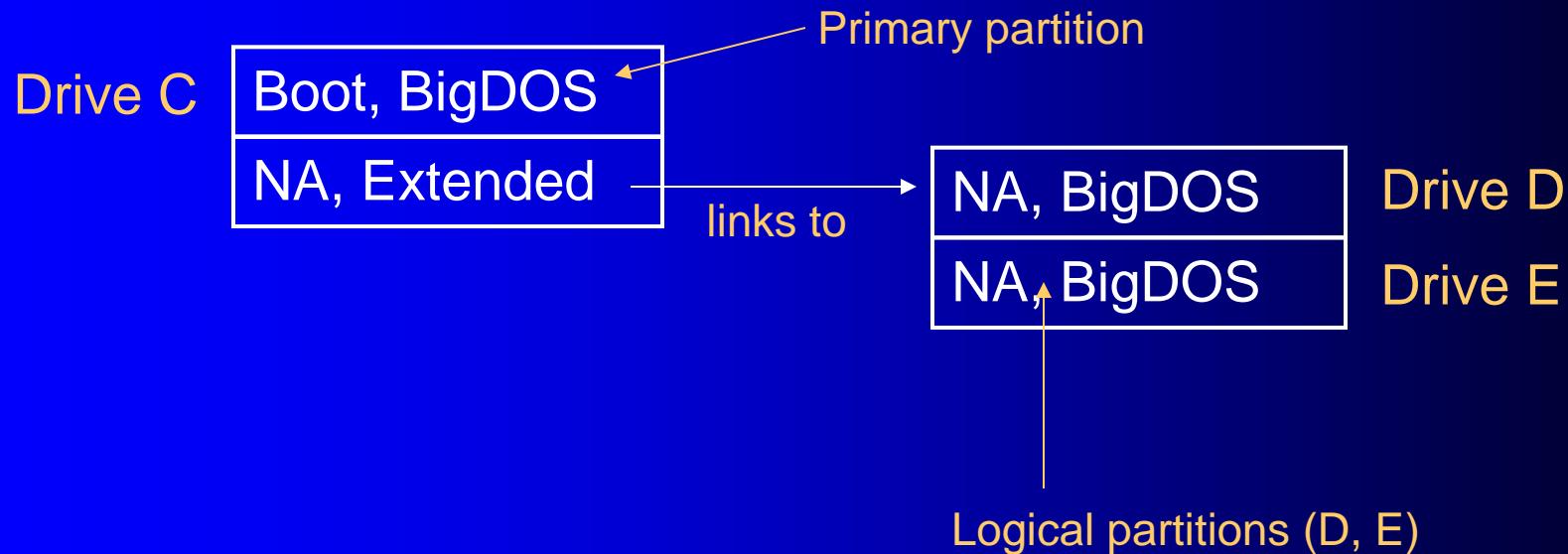
- Created from an extended partition
- No limit on the number
- Each has a separate drive letter
- Usually contain data
- Can be bootable (ex: Linux)

Disk Partition Table

- Located in the disk's Master Boot Record (MBR), following a block of executable code
- Four entries, one for each possible partition
- Each entry contains the following fields:
 - state (*non-active*, *bootable*)
 - type of partition (*BigDOS*, *Extended*, . . .)
 - beginning head, cylinder, & sector numbers
 - ending head, cylinder, & sector numbers
 - offset of partition from MBR
 - number of sectors in the partition

See also: www.datarescue.com/laboratory/partition.htm

Cascading Partition Tables



Boot = bootable (system)

NA = non active

BigDOS = over 32 MB

Dual-Boot Example

Volume	Layout	Type	File System	Status	Capacity	Free Space	% Free
	Partition	Basic		Healthy	5.13 GB	5.13 GB	100 %
	Partition	Basic		Healthy	2.01 GB	2.01 GB	100 %
BACKUP (E:)	Partition	Basic	FAT32	Healthy	7.80 GB	4.84 GB	62 %
DATA_1 (D:)	Partition	Basic	FAT32	Healthy	7.80 GB	2.66 GB	34 %
SYSTEM 98	Partition	Basic	FAT32	Healthy	1.95 GB	1.12 GB	57 %
WIN2000-A (C:)	Partition	Basic	NTFS	Healthy (System)	3.91 GB	1.43 GB	36 %

- System 98 and Win2000-A are bootable partitions
 - One is called the system partition when active
- DATA_1 and BACKUP are logical partitions
 - Their data can be shared by both operating systems

Master Boot Record (MBR)

- The MBR contains the following elements:
 - Disk partititon table
 - A program that jumps to the boot sector of the system partition

What's Next

- Disk Storage Systems
- **File Systems**
- Disk Directory
- Reading and Writing Disk Sectors (7305h)
- System-Level File Functions

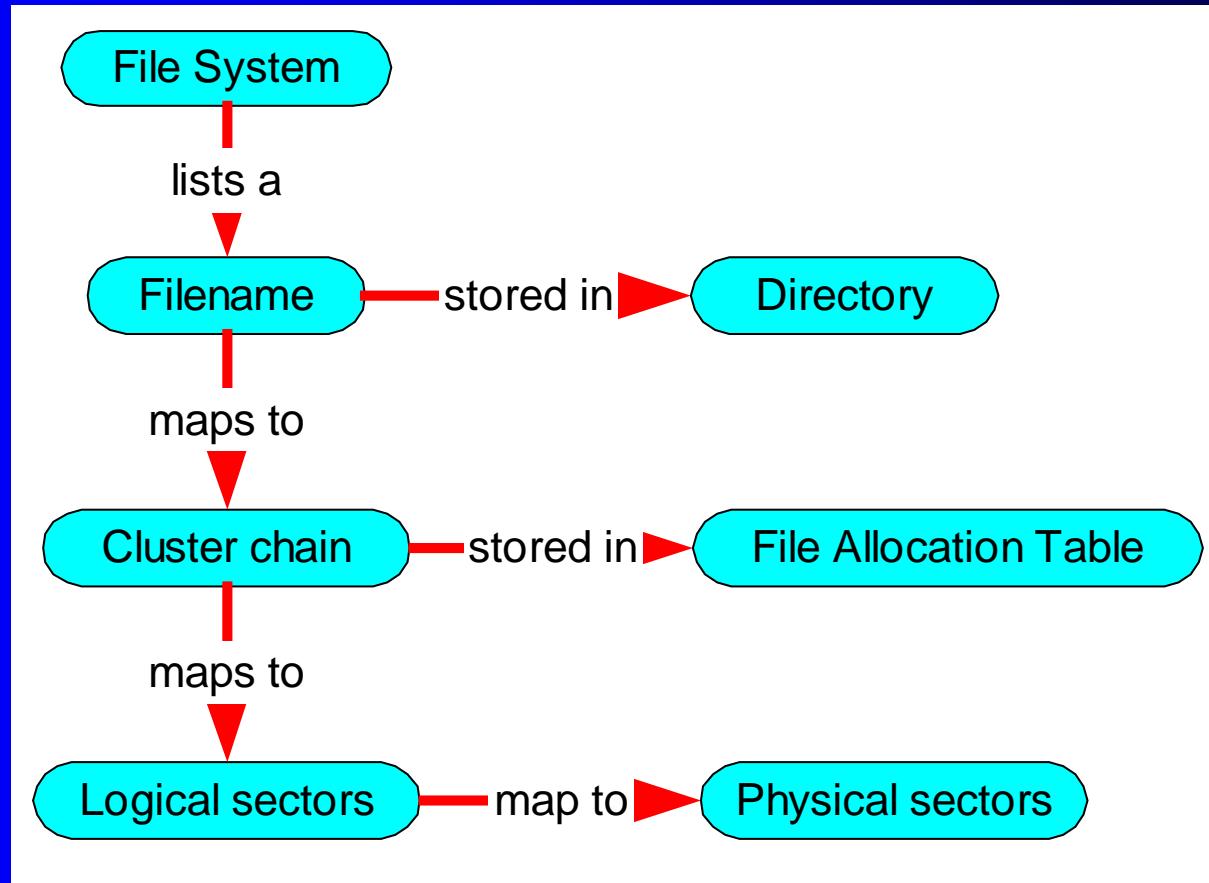
File Systems

- Directory, File, Cluster Relationships
- Clusters
- FAT12
- FAT16
- FAT32
- NTFS
- Primary Disk Areas

File System

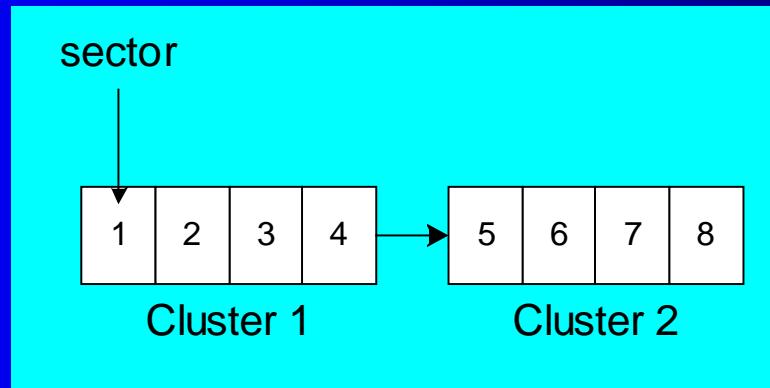
- This is what it does for you:
 - Keeps track of allocated and free space
 - Maintains directories and filenames
 - Tracks the sector location of each file and directory

Directory, File, Cluster, Sector Relationships



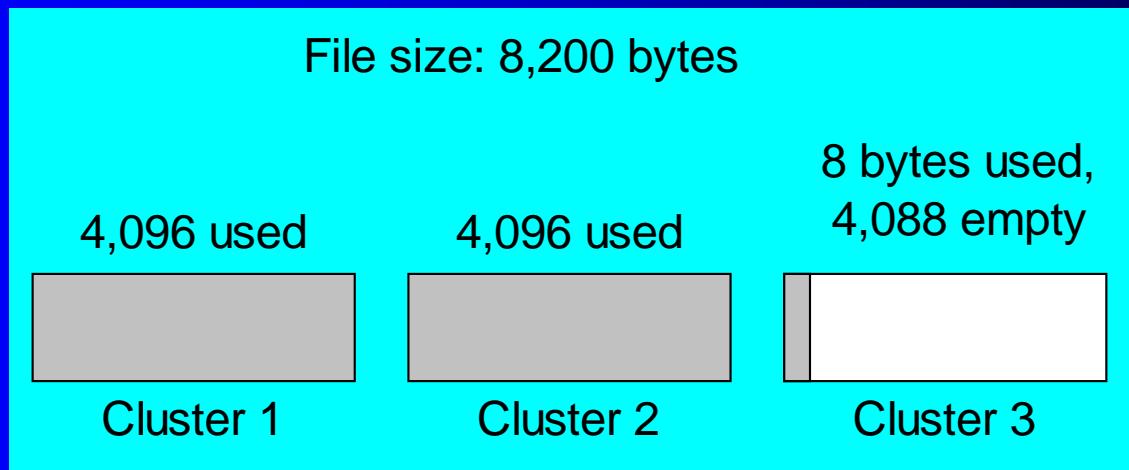
Cluster (1 of 2)

- Smallest unit of space used by a file
- Consists of one or more adjacent sectors
- Size depends on both the type of file system in use and the disk partition size
- A **file** is a linked sequence of clusters. Example:



Cluster (2 of 2)

- A file always uses at least one cluster
- A high percentage of space may be wasted
- Example: 8,200-byte file requires three 4K clusters:



FAT12

- Designed for diskettes
- Each FAT entry is 12 bits
- Very little fault tolerance
 - two copies of the FAT (cluster table)
- Optimal storage for small files
 - 512-byte clusters

FAT16

- MS-DOS format for hard disks
- 16-bit FAT entries
- Large cluster size when disk > 1 GB
 - inefficient for small files
- Max 2 GB size under MS-DOS
- Little or no recovery from read/write errors

FAT32

- Supports long filenames
- Supported by all version of MS-Windows from Windows 95 onward
 - (except Windows NT)
- 32-bit FAT entries
- 32 GB maximum volume size
- Improved recovery from read/write errors

NTFS

- Supported by Windows NT, 2000, and XP
- Handles large volumes
 - can span multiple hard drives
- Efficient cluster size (4K) on large volumes
- Unicode filenames
- Permissions on files & folders
- Share folders across network
- Built-in compression and encryption
- Track changes in a *change journal*
- Disk quotas for individuals or groups
- Robust error recovery
- Disk mirroring

Primary Disk Areas

- A disk or volume is divided into predefined areas and assigned specific logical sectors.
- Example: 1.44 MB diskette
 - Boot record (sector 0)
 - File allocation table (sectors 1 – 18)
 - Root directory (sectors 19 – 32)
 - Data area (sectors 33 – 2,879)

Your turn . . .

1. A 1.44 MB diskette has 512 bytes per cluster. Suppose a certain file begins in cluster number 5. Which logical disk sector contains the beginning of the file? (*Hint: see page 503*).

2. Suppose a certain hard drive has 4 KB per cluster, and we know that the data area begins in sector 100. If a particular file begins in cluster 10, which logical sectors are used by the cluster?

(answers on next panel . . .)

Answers

1. The data area begins in Sector 33. Each cluster = 1 sector, so the file begins in sector $33 + 5 = \text{sector } 38$.
2. The hard drive has 8 sectors per cluster. The starting cluster number of the file is $100 + (8 * 10) = 180$. Therefore, sectors 180 – 187 are used by the file's first cluster.

Boot Record (1 of 2)

- Fields in a typical MS-DOS boot record:
 - Jump to boot code (JMP instruction)
 - Manufacturer name, version number
 - Bytes per sector
 - Sectors per cluster
 - Number of reserved sectors (preceding FAT #1)
 - Number of copies of FAT
 - Maximum number of root directory entries
 - Number of disk sectors for drives under 32 MB
 - Media descriptor byte
 - Size of FAT, in sectors
 - Sectors per track

Boot Record (2 of 2)

(continued)

- Number of drive heads
- Number of hidden sectors
- Number of disk sectors for drives over 32 MB
- Drive number (modified by MS-DOS)
- Reserved
- Extended boot signature (always 29h)
- Volume ID number (binary)
- Volume label
- File-system type (ASCII)
- Start of boot program and data

What's Next

- Disk Storage Systems
- File Systems
- **Disk Directory**
- Reading and Writing Disk Sectors (7305h)
- System-Level File Functions

Keeping Track of Files

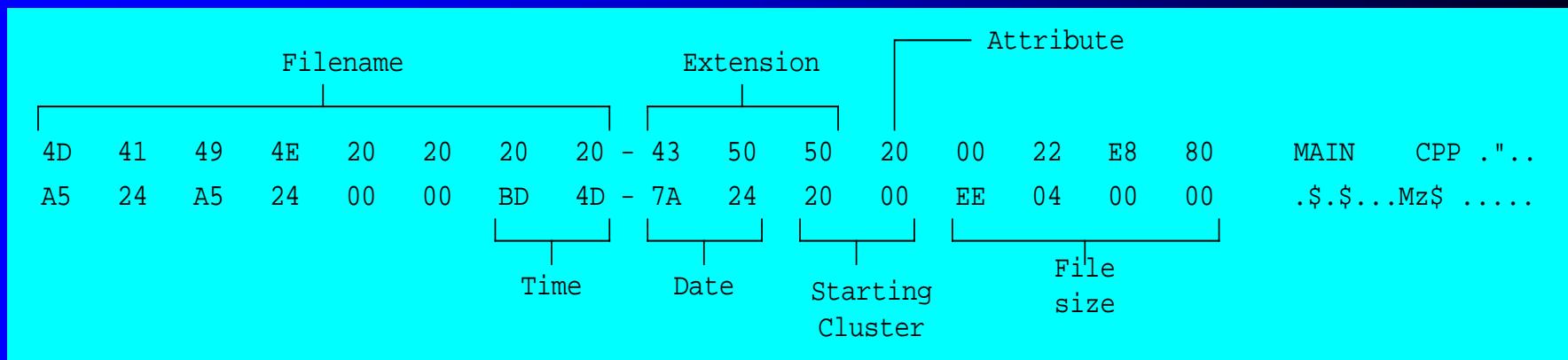
- MS-DOS Directory Structure
- Long Filenames in MS-Windows
- File Allocation Table

MS-DOS Directory Structure (1 of 2)

Hexadecimal Offset	Field Name	Format
00-07	Filename	ASCII
08-0A	Extension	ASCII
0B	Attribute	8-bit binary
0C-15	Reserved by MS-DOS	
16-17	Time stamp	16-bit binary
18-19	Date stamp	16-bit binary
1A-1B	Starting cluster number	16-bit binary
1C-1F	File size	32-bit binary

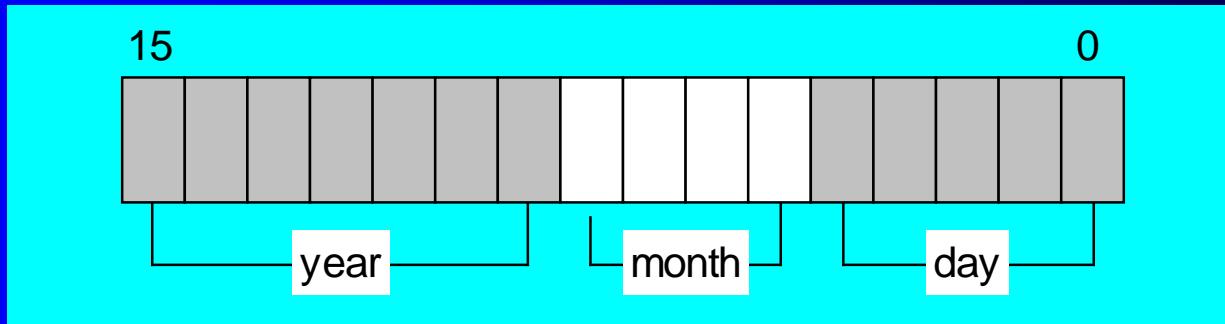
MS-DOS Directory Structure (2 of 2)

Time field equals 4DBDh (9:45:58), and the Date field equals 247Ah (March 26, 1998). Attribute is normal:

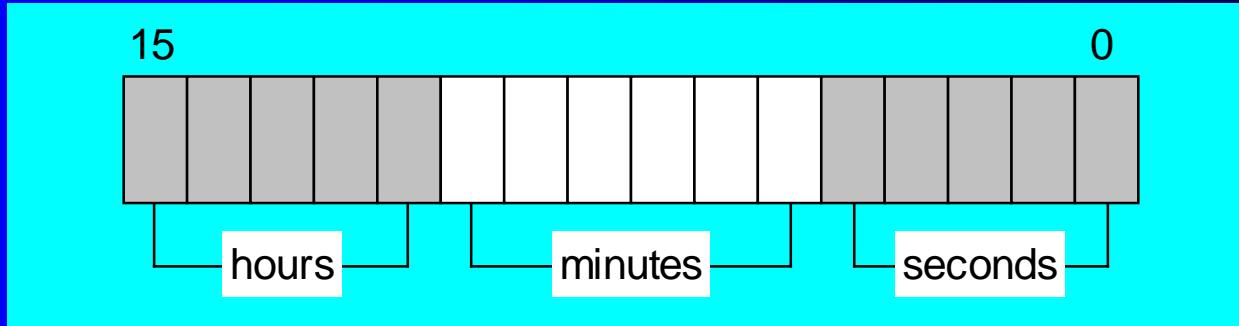


Date and Time Fields

- Date stamp field:

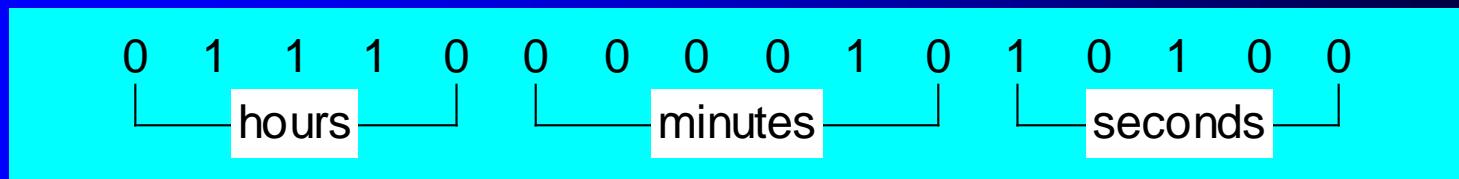


- Time stamp field:

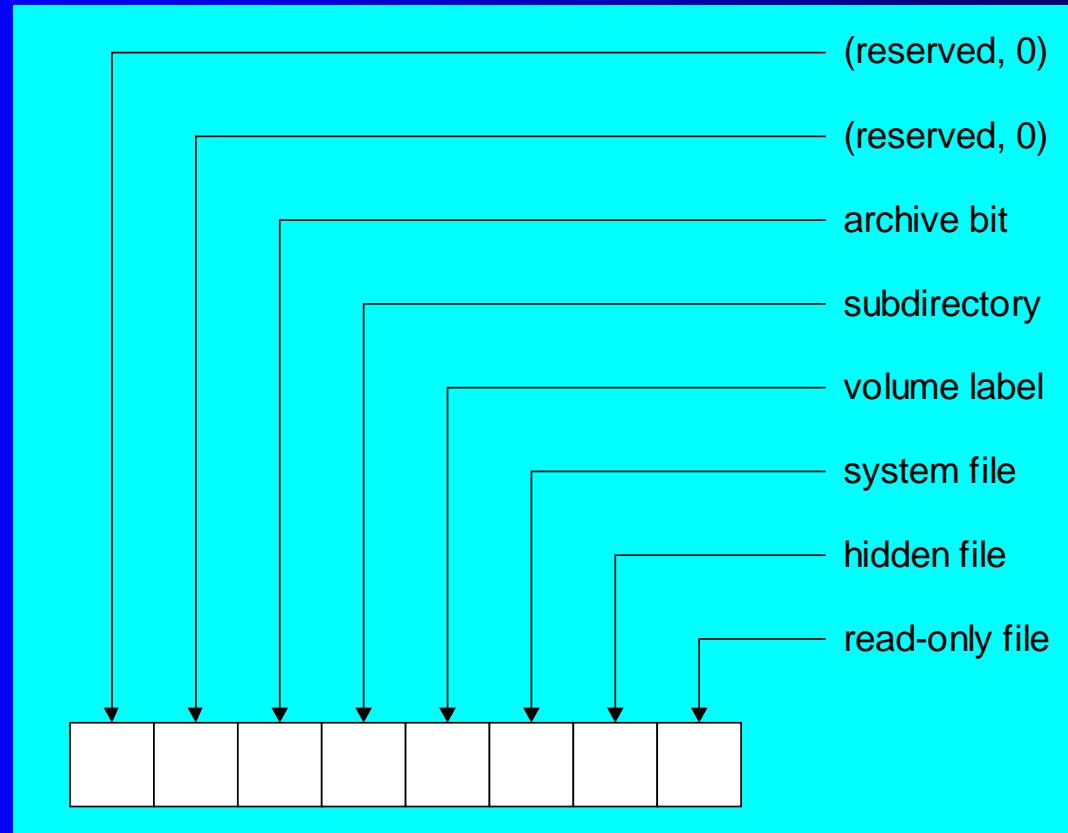


Your turn . . .

- What time value is represented here?



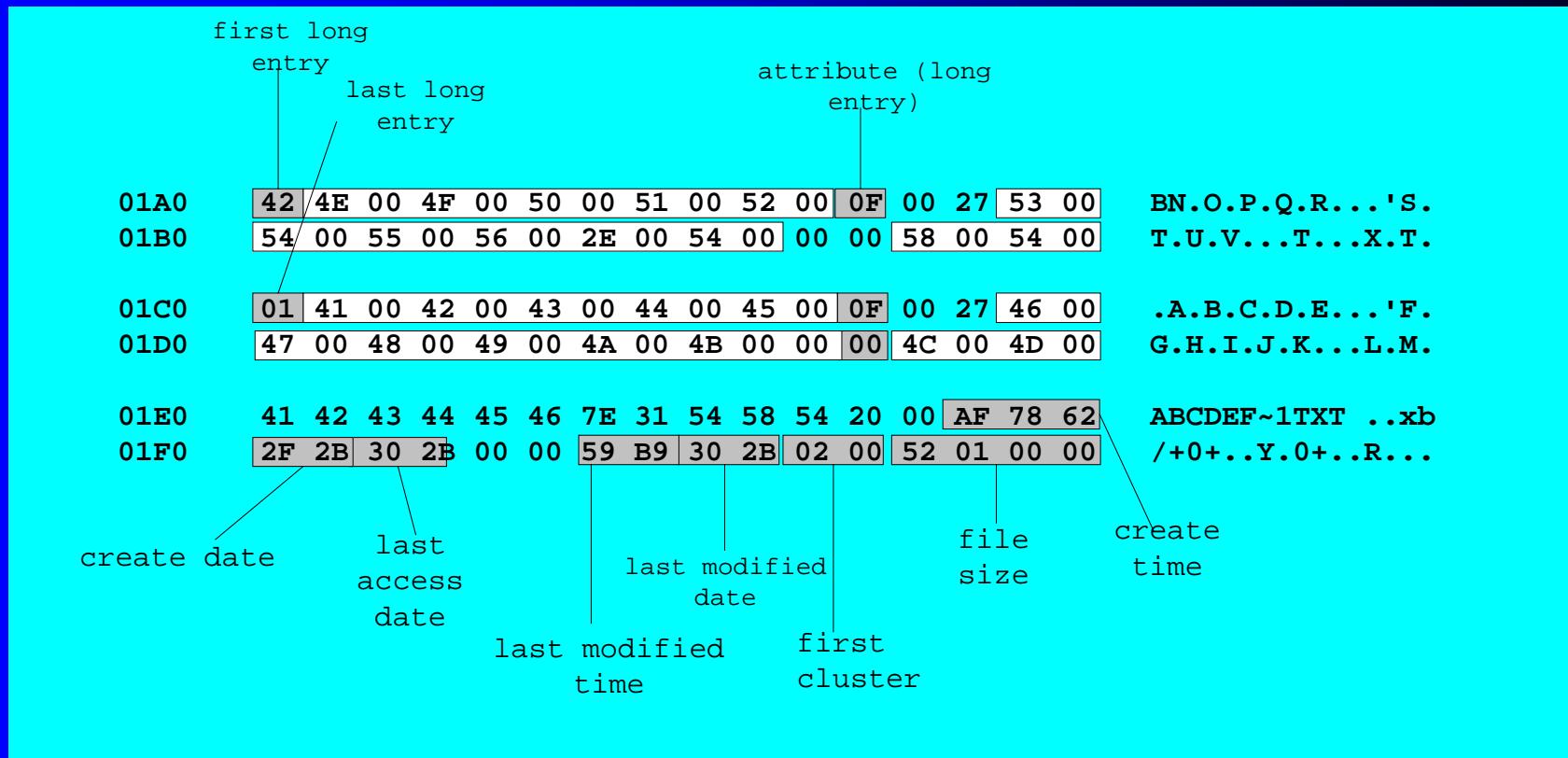
File Attribute Values



What type of file has attribute 00100111 . . . ?

Long Filenames in Windows

Filename: ABCDEFGHIJKLMNOPQRSTUVWXYZ.TXT



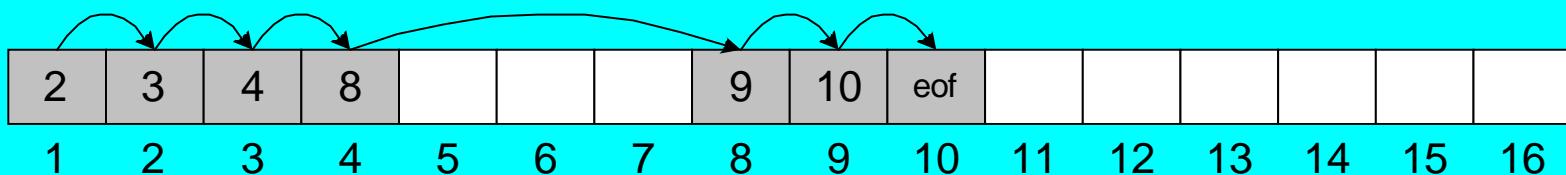
File Allocation Table (1 of 2)

- A map of all clusters on the disk, showing their ownership by specific files
- Each entry corresponds to a cluster number
- Each cluster contains one or more sectors
- Each file is represented in the FAT as a linked list, called a cluster chain.
- Three types of FAT's, named after the length of each FAT entry:
 - FAT-12
 - FAT-16
 - FAT-32

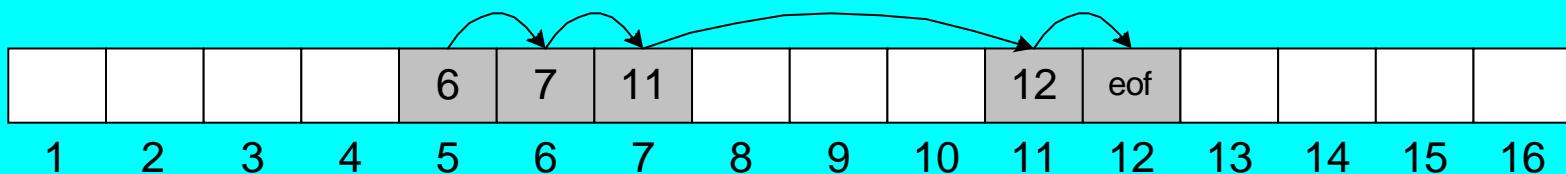
File Allocation Table (2 of 2)

- Each entry contains an n -bit integer that identifies the next entry. ($n=12, 16$, or 32)
- Two cluster chains are shown in the following diagram, one for File1, and another for File2:

File1: starting cluster number = 1, size = 7 clusters



File2: starting cluster number = 5, size = 5 clusters



What's Next

- Disk Storage Systems
- File Systems
- Disk Directory
- **Reading and Writing Disk Sectors (7305h)**
- System-Level File Functions

Reading and Writing Disk Sectors (7305h)

- INT 21h, Function 7305h (absolute disk read and write)
- Reads and writes logical disk sectors
- Runs only in 16-bit Real-address mode
- Does not work under Windows 2000, XP, Vista, Windows 7
 - Tight security!

DISKIO Structure

- Used by Function 7305h:

```
DISKIO STRUCT
    startSector DWORD 0          ; starting sector number
    numSectors WORD 1           ; number of sectors
    bufferOfs WORD buffer      ; buffer offset
    bufferSeg WORD @DATA       ; buffer segment
DISKIO ENDS
```

Example

Example: Read one or more sectors from drive C:

```
.data
buffer BYTE 512 DUP(?)
diskStruct DISKIO <>
.code
    mov ax,7305h          ; absolute Read/Write
    mov cx,0FFFFh         ; always this value
    mov dl,3               ; drive C
    mov bx,OFFSET diskStruct
    mov si,0               ; read sector
    int 21h
```

Sector Display Program

Pseudocode:

```
Ask for starting sector number and drive number
do while (keystroke <> ESC)
    Display heading
    Read one sector
    If MS-DOS error then exit
    Display one sector
    Wait for keystroke
    Increment sector number
end do
```

[View the source code](#)

What's Next

- Disk Storage Systems
- File Systems
- Disk Directory
- Reading and Writing Disk Sectors (7305h)
- **System-Level File Functions**

System-Level File Functions

- Common Disk-Related Functions
- Get Disk Free Space
- Create Subdirectory
- Remove Subdirecrory
- Set Current Directory
- Get Current Directory

Common Disk-Related Functions

Function Number	Function Name
0Eh	Set default drive
19h	Get default drive
7303h	Get disk free space
39h	Create subdirectory
3Ah	Remove subdirectory
3Bh	Set current directory
41h	Delete file
43h	Get/set file attribute
47h	Get current directory path
4Eh	Find first matching file
4Fh	Find next matching file
56h	Rename file
57h	Get/set file date and time
59h	Get extended error information

ExtGetDskFreSpcStruc Structure (1 of 2)

Structure data returned by Function 7303h:

- **StructSize**: A return value that represents the size of the ExtGetDskFreSpcStruc structure, in bytes.
- **Level**: Always 0.
- **SectorsPerCluster**: The number of sectors inside each cluster.
- **BytesPerSector**: The number of bytes in each sector.
- **AvailableClusters**: The number of available clusters.
- **TotalClusters**: The total number of clusters in the volume.

ExtGetDskFreSpcStruc (2 of 2)

- AvailablePhysSectors: The number of physical sectors available in the volume, without adjustment for compression.
- TotalPhysSectors: The total number of physical sectors in the volume, without adjustment for compression.
- AvailableAllocationUnits: The number of available allocation units in the volume, without adjustment for compression.
- TotalAllocationUnits: The total number of allocation units in the volume, without adjustment for compression.
- Rsvd: Reserved member.

Function 7303h – Get Disk Free Space

- AX = 7303h
- ES:DI points to a ExtGetDskFreSpcStruc
- CX = size of the ExtGetDskFreSpcStruc variable
- DS:DX points to a null-terminated string containing the drive name

View the [DiskSpc.asm](#) program

Create Subdirectory

```
.data  
pathname BYTE "\ASM",0  
  
.code  
    mov ah,39h          ; create subdirectory  
    mov dx,OFFSET pathname  
    int 21h  
    jc DisplayError  
    .  
    .  
DisplayError:
```

Remove Subdirectory

```
.data  
pathname    BYTE    'C:\ASM',0  
  
.code  
    mov ah,3Ah                      ; remove subdirectory  
    mov dx,OFFSET pathname  
    int 21h  
    jc  DisplayError  
    .  
    .  
DisplayError:
```

Set Current Directory

```
.data  
pathname    BYTE  "C:\ASM\PROGS",0  
  
.code  
    mov ah,3Bh          ; set current directory  
    mov dx,OFFSET pathname  
    int 21h  
    jc  DisplayError  
    .  
    .  
DisplayError:
```

Get Current Directory

```
.data  
pathname BYTE 64 dup(0)      ; path stored here by MS-DOS  
  
.code  
    mov ah,47h                  ; get current directory path  
    mov dl,0                    ; on default drive  
    mov si,OFFSET pathname  
    int 21h  
    jc  DisplayError  
    .  
    .  
DisplayError:
```

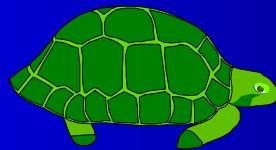
Your turn . . .

- Write a program that creates a hidden, read-only directory named `__secret`. Create a hidden file inside the new directory named `$$temp`. Try to remove the directory by calling Function 3Ah. Display the error code returned by MS-DOS.

Summary

- Disk controller: acts as a broker between the hardware and the operating system
- Disk characteristics
 - composed of tracks, cylinders, sectors
 - average seek time, data transfer rate
- Formatting & logical characteristics
 - master boot record, contains disk partition table
 - clusters – logical storage units
 - file allocation table – used by some systems
 - directory – root directory, subdirectories

The End



Assembly Language for x86 Processors

7th Edition

Kip R. Irvine

Chapter 16: BIOS-Level Programming

Slide show prepared by the author

Revision date: 1/15/2014

Chapter Overview

- Introduction
- Keyboard Input with INT 16h
- VIDEO Programming with INT 10h
- Drawing Graphics Using INT 10h
- Memory-Mapped Graphics
- Mouse Programming

Personalities

- Bill Gates: co-authored QBASIC interpreter
- Gary Kildall: creator of CP/M-86 operating system
 - multitasking capabilities
- Peter Norton:
 - Inside the IBM-PC first book to thoroughly explore IBM-PC software and hardware
 - created the **Norton Utilities** software
- Michael Abrash: columnist, expert programmer
 - worked on Quake and Doom computer games
 - optimized graphics code in Windows NT
 - book: **The Zen of Code Optimization**

PC-BIOS

- The BIOS (Basic Input-Output System) provides low-level hardware drivers for the operating system.
 - accessible to 16-bit applications
 - written in assembly language, of course
 - source code published by IBM in early 1980's
- Advantages over MS-DOS:
 - permits graphics and color programming
 - faster I/O speeds
 - read mouse, serial port, parallel port
 - low-level disk access

BIOS Data Area

- Fixed-location data area at address 00400h
 - this area is also used by MS-DOS
 - this area is accessible under Windows 98 & Windows Me, but not under Windows NT, 2000, or XP.
- Contents:
 - Serial and parallel port addresses
 - Hardware list, memory size
 - Keyboard status flags, keyboard buffer pointers, keyboard buffer data
 - Video hardware configuration
 - Timer data

What's Next

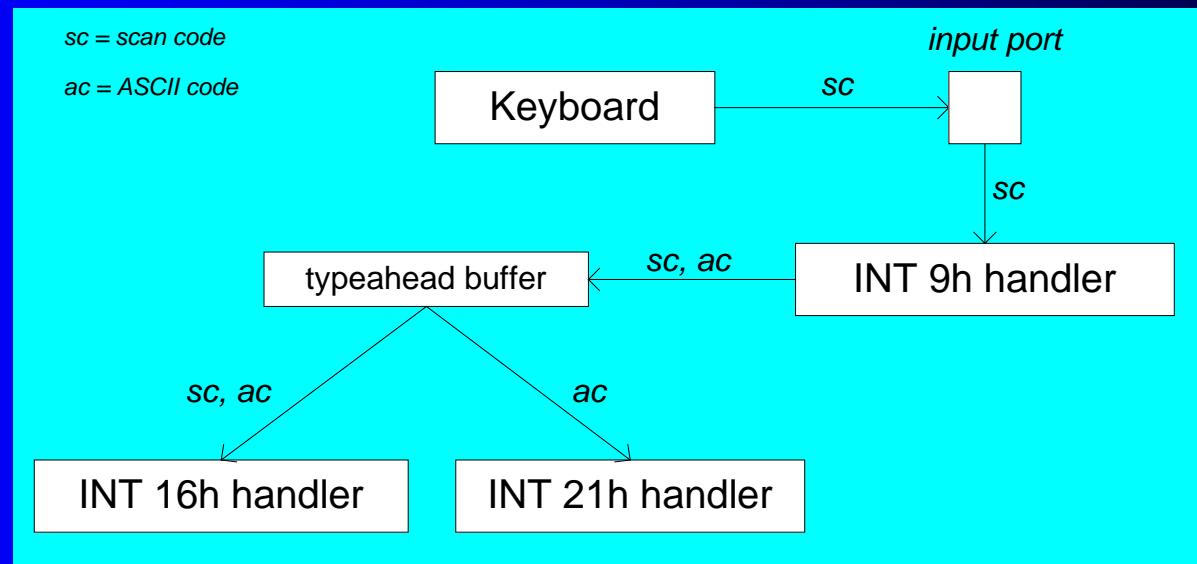
- Introduction
- **Keyboard Input with INT 16h**
- VIDEO Programming with INT 10h
- Drawing Graphics Using INT 10h
- Memory-Mapped Graphics
- Mouse Programming

Keyboard Input with INT 16h

- How the Keyboard Works
- INT 16h Functions

How the Keyboard Works

- Keystroke sends a scan code to the keyboard serial input port
- Interrupt triggered: INT 9h service routine executes
- Scan code and ASCII code inserted into keyboard typeahead buffer



Keyboard Flags

16-bits, located at 0040:0017h – 0018h.

Bit	Description
0	Right Shift key is down
1	Left Shift key is down
2	Either Ctrl key is down
3	Either Alt key is down
4	Scroll Lock toggle is on
5	Num Lock toggle is on
6	Caps Lock toggle is on
7	Insert toggle is on
8	Left Ctrl key is down

Bit	Description
9	Left Alt key is down
10	Right Ctrl key is down
11	Right Alt key is down
12	Scroll key is down
13	Num Lock key is down
14	Caps Lock key is down
15	SysReq key is down

INT 16h Functions

- Provide low-level access to the keyboard, more so than MS-DOS.
- Input-output cannot be redirected at the command prompt.
- Function number is always in the AH register
- Important functions:
 - set typematic rate
 - push key into buffer
 - wait for key
 - check keyboard buffer
 - get keyboard flags

Function 10h: Wait for Key

If a key is waiting in the buffer, the function returns it immediately. If no key is waiting, the program pauses (blocks), waiting for user input.

```
.data  
scanCode BYTE ?  
ASCIICode BYTE ?  
  
.code  
mov ah,10h  
int 16h  
mov scanCode,ah  
mov ASCIICode,al
```

Function 12h: Get Keyboard Flags

Retrieves a copy of the keyboard status flags from the BIOS data area.

```
.data  
keyFlags WORD ?  
  
.code  
mov ah,12h  
int 16h  
mov keyFlags,ax
```

Clearing the Keyboard Buffer

Function 11h clears the Zero flag if a key is waiting in the keyboard typeahead buffer.

```
L1: mov ah,11h          ; check keyboard buffer
    int 16h            ; any key pressed?
    jz  noKey          ; no: exit now
    mov ah,10h          ; yes: remove from buffer
    int 16h
    cmp ah,scanCode    ; was it the exit key?
    je  quit           ; yes: exit now (ZF=1)
    jmp L1              ; no: check buffer again

noKey:
    or   al,1            ; no key pressed
                           ; clear zero flag

quit:
```

What's Next

- Introduction
- Keyboard Input with INT 16h
- **VIDEO Programming with INT 10h**
- Drawing Graphics Using INT 10h
- Memory-Mapped Graphics
- Mouse Programming

VIDEO Programming with INT 10h

- Basic Background
- Controlling the Color
- INT 10h Video Functions
- Library Procedure Examples

Video Modes

- Graphics video modes
 - draw pixel by pixel
 - multiple colors
- Text video modes
 - character output, using hardware or software-based font table
 - mode 3 (color text) is the default
 - default range of 80 columns by 25 rows.
 - color attribute byte contains foreground and background colors

Three Levels of Video Access

- MS-DOS function calls
 - slow, but they work on any MS-DOS machine
 - I/O can be redirected
- BIOS function calls
 - medium-fast, work on nearly all MS-DOS-based machines
 - I/O cannot be redirected
- Direct memory-mapped video
 - fast – works only on 100% IBM-compatible computers
 - cannot be redirected
 - does not work under Windows NT, 2000, or XP

Controlling the Color

- Mix primary colors: red, yellow, blue
 - called subtractive mixing
 - add the intensity bit for 4th channel
- Examples:
 - red + green + blue = light gray (0111)
 - intensity + green + blue = white (1111)
 - green + blue = cyan (0011)
 - red + blue = magenta (0101)
- Attribute byte:
 - 4 MSB bits = background
 - 4 LSB bits = foreground

Constructing Attribute Bytes

- Color constants defined in `Irvine32.inc` and `Irvine16.inc`:
- Examples:
 - Light gray text on a blue background:
 - (blue SHL 4) OR lightGray
 - White text on a red background:
 - (red SHL 4) OR white

INT 10h Video Functions

- AH register contains the function number
- 00h: Set video mode
 - text modes listed in Table 15-5
 - graphics modes listed in Table 15-6
- 01h: Set cursor lines
- 02h: Set cursor position
- 03h: Get cursor position and size
- 06h: Scroll window up
- 07h: Scroll window down
- 08h: Read character and attribute

INT 10h Video Functions (*cont*)

- 09h: Write character and attribute
- 0Ah: Write character
- 10h (AL = 03h): Toggle blinking/intensity bit
- 0Fh: Get video mode
- 13h: Write string in teletype mode

Displaying a Color String

Write one character and attribute:

```
mov  si,OFFSET string
. . .
mov  ah,9          ; write character/attribute
mov  al,[si]        ; character to display
mov  bh,0          ; video page 0
mov  bl,color       ; attribute
or   bl,10000000b   ; set blink/intensity bit
mov  cx,1          ; display it one time
int  10h
```

Gotoxy Procedure

```
;-----  
Gotoxy PROC  
;  
; Sets the cursor position on video page 0.  
; Receives: DH,DL = row, column  
; Returns: nothing  
;-----  
    pusha  
    mov ah,2  
    mov bh,0  
    int 10h  
    popa  
    ret  
Gotoxy ENDP
```

Clrscr Procedure

```
Clrscr PROC  
    pusha  
    mov     ax,0600h          ; scroll window up  
    mov     cx,0               ; upper left corner (0,0)  
    mov     dx,184Fh           ; lower right corner (24,79)  
    mov     bh,7               ; normal attribute  
    int     10h               ; call BIOS  
    mov     ah,2               ; locate cursor at 0,0  
    mov     bh,0               ; video page 0  
    mov     dx,0               ; row 0, column 0  
    int     10h  
    popa  
    ret  
Clrscr ENDP
```

What's Next

- Introduction
- Keyboard Input with INT 16h
- VIDEO Programming with INT 10h
- **Drawing Graphics Using INT 10h**
- Memory-Mapped Graphics
- Mouse Programming

Drawing Graphics Using INT 10h

- INT 10h Pixel-Related Functions
- DrawLine Program
- Cartesian Coordinates Program
- Converting Cartesian Coordinates to Screen Coordinates

INT 10h Pixel-Related Functions

- Slow performance
- Easy to program
- 0Ch: Write graphics pixel
- 0Dh: Read graphics pixel

DrawLine Program

- Draws a straight line, using INT 10h function calls
- Saves and restores current video mode
- Excerpt from the *DrawLine* program ([DrawLine.asm](#)):

```
mov ah,0Ch          ; write pixel
mov al,color        ; pixel color
mov bh,0            ; video page 0
mov cx,currentX
int 10h
```

Cartesian Coordinates Program

- Draws the X and Y axes of a Cartesian coordinate system
- Uses video mode 6A (800 x 600, 16 colors)
- Name: Pixel2.asm
- Important procedures:
 - DrawHorizLine
 - DrawVerticalLine

Converting Cartesian Coordinates to Screen Coordinates

- Screen coordinates place the origin (0,0) at the upper-left corner of the screen
- Graphing functions often need to display negative values
 - move origin point to the middle of the screen
- For Cartesian coordinates X, Y and origin points $sOrigX$ and $sOrigY$, screen X and screen Y are calculated as:
 - $sx = (sOrigX + X)$
 - $sy = (sOrigY - Y)$

What's Next

- Introduction
- Keyboard Input with INT 16h
- VIDEO Programming with INT 10h
- Drawing Graphics Using INT 10h
- **Memory-Mapped Graphics**
- Mouse Programming

Memory-Mapped Graphics

- Binary values are written to video RAM
 - video adapter must use standard address
- Very fast performance
 - no BIOS or DOS routines to get in the way

Mode 13h: 320 X 200, 256 Colors

- Mode 13h graphics (320 X 200, 256 colors)
 - Fairly easy to program
 - read and write video adapter via IN and OUT instructions
 - pixel-mapping scheme (1 byte per pixel)

Mode 13h Details

- OUT Instruction

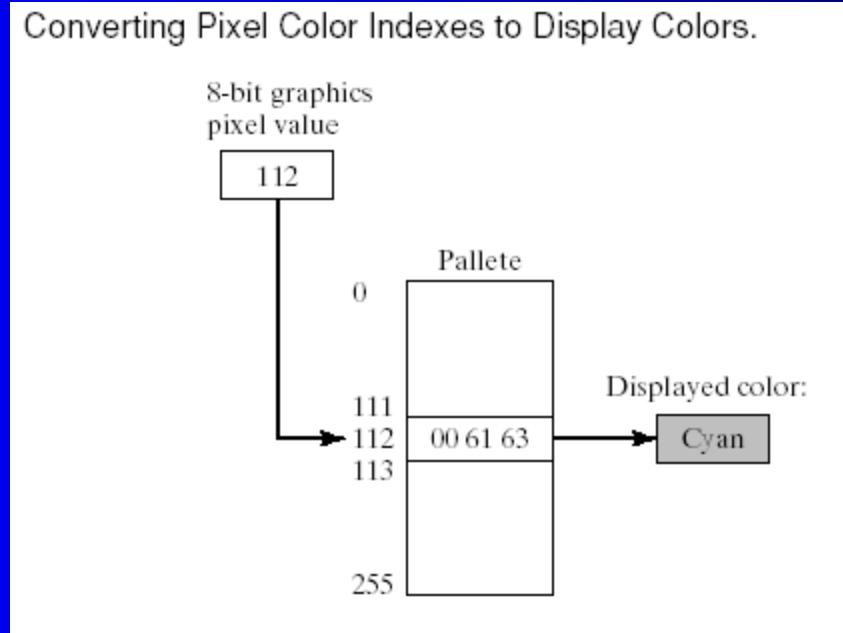
- 16-bit port address assigned to DX register
- output value in AL, AX, or EAX
- Example:

```
mov dx,3c8h          ; port address
mov al,20h          ; value to be sent
out dx,al          ; send to the port
```

- Color Indexes

- color integer value is an index into a table of colors called a palette

Color Indexes in Mode 13h



RGB Colors

Additive mixing of light (red, green, blue). Intensities vary from 0 to 255.

Examples:

Red	Green	Blue	Color
0	30	30	cyan
30	30	0	yellow
30	0	30	magenta
40	0	63	lavender

Red	Green	Blue	Color
0	0	0	black
20	20	20	dark gray
35	35	35	medium gray
50	50	50	light gray
63	63	63	white

Red	Green	Blue	Color
63	0	0	bright red
10	0	0	dark red
30	0	0	medium red
63	40	40	pink

What's Next

- Introduction
- Keyboard Input with INT 16h
- VIDEO Programming with INT 10h
- Drawing Graphics Using INT 10h
- Memory-Mapped Graphics
- **Mouse Programming**

Mouse Programming

- MS-DOS functions for reading the mouse
- Mickey – unit of measurement (200th of an inch)
 - mickeys-to-pixels ratio (8 x 16) is variable
- INT 33h functions
- Mouse Tracking Program Example

Reset Mouse and Get Status

- INT 33h, AX = 0
- Example:

```
mov    ax,0  
int    33h  
cmp    ax,0  
je     MouseNotAvailable  
mov    numberOfButtons,bx
```

Show/Hide Mouse

- INT 33h, AX = 1 (show), AX = 2 (hide)
- Example:

```
mov  ax,1          ; show  
int 33h  
mov  ax,2          ; hide  
int 33h
```

Get Mouse Position & Status

- INT 33h, AX = 4
- Example:

```
mov  ax,4  
mov  cx,200      ; X-position  
mov  dx,100      ; Y-position  
int  33h
```

Get Button Press Information

- INT 33h, AX = 5
- Example:

```
mov  ax,5
mov  bx,0          ; button ID
int  33h
test ax,1          ; left button down?
jz   skip          ; no - skip
mov  X_coord,cx   ; yes: save coordinates
mov  Y_coord,dx
```

Other Mouse Functions

- AX = 6: Get Button Release Information
- AX = 7: Set Horizontal Limits
- AX = 8: Set Vertical Limits

Mouse Tracking Program

- Tracks the movement of the text mouse cursor
- X and Y coordinates are continually updated in the lower-right corner of the screen
- When the user presses the left button, the mouse's position is displayed in the lower left corner of the screen
- Source code (c:\Irvine\Examples\ch15\mouse.asm)

Set Mouse Position

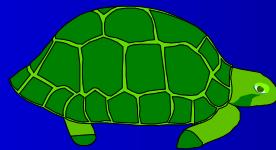
- INT 33h, AX = 3
- Example:

```
mov  ax,3
int 33h
test bx,1
jne Left_Button_Down
test bx,2
jne Right_Button_Down
test bx,4
jne Center_Button_Down
mov  Xcoord,cx
mov  yCoord,dx
```

Summary

- Working at the BIOS level gives you a high level of control over hardware
- Use INT 16h for keyboard control
- Use INT 10h for video text
- Use memory-mapped I/O for graphics
- Use INT 33h for the mouse

The End



Assembly Language for x86 Processors

7th Edition

Kip R. Irvine

Chapter 17: Expert MS-DOS Programming

Slide show prepared by the author

Revision date: 1/15/2014

Chapter Overview

- **Defining Segments**
- Runtime Program Structure
- Interrupt Handling
- Hardware Control Using I/O Ports

Defining Segments

- Simplified Segment Directives
- Explicit Segment Definitions
- Segment Overrides
- Combining Segments

Simplified Segment Directives

- .MODEL – program memory model
- .CODE – code segment
- .CONST – define constants
- .DATA – near data segment
- .DATA? – uninitialized data
- .FARDATA – far data segment
- .FARDATA? – far uninitialized data
- .STACK – stack segment
- .STARTUP – initialize DS and ES
- .EXIT – halt program

Memory Models

Model	Description
tiny	A single segment, containing both code and data. This model is used by .com programs.
small	One code segment and one data segment. All code and data are near, by default.
medium	Multiple code segments and a single data segment.
compact	One code segment and multiple data segments.
large	Multiple code and data segments.
huge	Same as the large model, except that individual data items may be larger than a single segment.
flat	Protected mode. Uses 32-bit offsets for code and data. All data and code (including system resources) are in a single 32-bit segment.

NEAR and FAR Segments

- NEAR segment
 - requires only a 16-bit offset
 - faster execution than FAR
- FAR segment
 - 32-bit offset: requires setting both segment and offset values
 - slower execution than NEAR

.MODEL Directive

- The .MODEL directive determines the names and grouping of segments
- `.model tiny`
 - code and data belong to same segment (NEAR)
 - .com file extension
- `.model small`
 - both code and data are NEAR
 - data and stack grouped into DGROUP
- `.model medium`
 - code is FAR, data is NEAR

.MODEL Directive

- `.model compact`
 - code is NEAR, data is FAR
- `.model huge & .model large`
 - both code and data are FAR
- `.model flat`
 - both code and data are 32-bit NEAR

.MODEL Directive

- Syntax:

.MODEL type, language, stackdistance

- *Language* can be:

- C, BASIC, FORTRAN, PASCAL, SYSCALL, or STDCALL (details in Chapters 8 and 12).

- *Stackdistance* can be:

- NEARSTACK: (default) places the stack segment in the group DGROUP along with the data segment
 - FARSTACK: stack and data are **not** grouped together

.STACK Directive

- Syntax:
`.STACK [stacksize]`
- *Stacksize* specifies size of stack, in bytes
 - default is 1024
- Example: set to 2048 bytes:
 - `.stack 2048`

.CODE Directive

- Syntax:

`.CODE [segname]`

- optional *segname* overrides the default name
- Small, compact memory models
 - NEAR code segment
 - segment is named `_TEXT`
- Medium, large, huge memory models
 - FAR code segment
 - segment is named `modulename_TEXT`

Whenever the CPU executes a FAR call or jump, it loads CS with the new segment address.

Calling Library Procedures

- You must use .MODEL small, stdcall
 - (designed for the small memory model)
- You can only call Irvine16 library procedures from segments named _TEXT.
 - (default name when .CODE is used)
- Advantages
 - calls and jumps execute more quickly
 - simple use of data—DS never needs to change
- Disadvantages
 - segment names restricted
 - limited to 64K code, and 64K data

Multiple Code Segments

Example, p. 585, shows calling Irvine16 procedures from main, and calling an MS-DOS interrupt from Display.

```
.code

main PROC
    mov    ax,@data
    mov    ds,ax
    call   WriteString
    call   Display
    .exit
main ENDP
```

```
.code OtherCode

Display PROC
    mov   ah,9
    mov   dx,OFFSET msg2
    int   21h
    ret
Display ENDP
```

Near Data Segments

- .DATA directive creates a Near segment
 - Up to 64K in Real-address mode
 - Up to 512MB in Protected mode (Windows NT)
 - 16-bit offsets are used for all code and data
 - automatically creates segment named DGROUP
 - can be used in any memory model
- Other types of data:
 - .DATA? (uninitialized data)
 - .CONST (constant data)

Far Data Segments

- **.FARDATA**
 - creates a FAR_DATA segment
- **.FARDATA?**
 - creates a FAR_BSS segment
- Code to access data in a far segment:

```
.FARDATA  
myVar  
.CODE  
    mov ax,SEG myVar  
    mov ds,ax
```

The SEG operator returns the segment value of a label. Similar to @data.

Data-Related Symbols

- `@data` returns the group of the data segment
- `@DataSize` returns the size of the memory model set by the `.MODEL` directive
- `@WordSize` returns the size attribute of the current segment
- `@CurSeg` returns the name of the current segment

Explicit Segment Definitions

- Use them when you cannot or do not want to use simplified segment directives
- All segment attributes must be specified
- The ASSUME directive is required

SEGMENT Directive

Syntax:

```
name SEGMENT [align] [combine] ['class']  
    statements  
name ENDS
```

- *name* identifies the segment; it can either be unique or the name of an existing segment.
- *align* can be BYTE, WORD, DWORD, PARA, or PAGE.
- *combine* can be PRIVATE, PUBLIC, STACK, COMMON, MEMORY, or AT *address*.
- *class* is an identifier used when identifying a particular type of segment such as CODE or STACK.

Segment Example

```
ExtraData SEGMENT PARA PUBLIC 'DATA'  
    var1 BYTE 1  
    var2 WORD 2  
ExtraData ENDS
```

- name: ExtraData
- paragraph align type (starts on 16-bit boundary)
- public combine type: combine with all other public segments having the same name
- 'DATA' class: 'DATA' (load into memory along with other segments whose class is 'DATA')

ASSUME Directive

- Tells the assembler how to calculate the offsets of labels
- Associates a segment register with a segment name

Syntax:

```
ASSUME segreg:segname [,segreg:segname] ...
```

Examples:

```
ASSUME cs:myCode, ds:Data, ss:myStack  
ASSUME es:ExtraData
```

Multiple Data Segments (1 of 2)

```
cseg SEGMENT 'CODE'  
ASSUME cs:cseg, ds:edata1, es:edata2, ss:mystack  
  
main PROC  
    mov ax,edata1          ; DS points to edata1  
    mov ds,ax  
    mov ax,SEG val2        ; ES points to edata2  
    mov es,ax  
    mov ax,vall            ; edata1 segment assumed  
    mov bx,val2            ; edata2 segment assumed  
  
    mov ax,4C00h            ; (same as .exit)  
    int 21h  
main ENDP  
cseg ENDS
```

Multiple Data Segments (1 of 2)

```
data1 SEGMENT 'DATA'  
    val1 WORD 1001h  
data1 ENDS  
  
data2 SEGMENT 'DATA'  
    val2 WORD 1002h  
data2 ENDS  
  
mystack SEGMENT PARA STACK 'STACK'  
    BYTE 100h DUP('S')  
mystack ENDS  
  
END main
```

Segment Overrides

- A segment override instructs the processor to use a different segment from the default when calculating an effective address
- Syntax:

```
segreg:segname  
segname:label
```

```
cseg SEGMENT 'CODE'  
ASSUME cs:cseg, ss:mystack  
  
main PROC  
    ...  
    mov ax,ds:val1  
    mov bx,OFFSET AltSeg:var2
```

Combining Segments

- Segments can be merged into a single segment by the linker, if . . .
 - their names are the same,
 - and they both have combine type PUBLIC,
 - . . . even when they appear in different source code modules
- Example:
 - cseg SEGMENT PUBLIC 'CODE'
- See the program in the Examples\ch16\Seg2\ directory

What's Next

- Defining Segments
- **Runtime Program Structure**
- Interrupt Handling
- Hardware Control Using I/O Ports

Runtime Program Structure

- COM Programs
- EXE Programs

When you run a program, . . .

MS-DOS performs the following steps, in order:

1. checks for a matching internal command name
2. looks for a matching file with .COM, .EXE, or .BAT extensions, in that order, in the current directory
3. looks in the first directory in the PATH variable, for .COM, .EXE, and .BAT file
4. continues to second directory in the PATH, and so on

Program Segment Prefix (PSP)

- 256-byte memory block created when a program is loaded into memory
- contains pointer to Ctrl-Break handler
- contains pointers saved by MS-DOS
- Offset 2Ch: 16-bit segment address of current environment string
- Offset 80h: disk transfer area, and copy of the current MS-DOS command tail

COM Programs

- Unmodified binary image of a program
- PSP created at offset 0 by loader
- Code, data, stack all in the same segment
- Code entry point is at offset 0100h, data follows immediately after code
- Stack located at the end of the segment
- All segments point to base of PSP
- Based on TINY memory model
- Linker uses the /T option
- Can only run under MS-DOS

Sample COM Program

```
TITLE Hello Program in COM format      (HelloCom.asm)
```

```
.MODEL tiny
.code
ORG 100h           ; must be before main
main PROC
    mov ah,9
    mov dx,OFFSET hello_message
    int 21h
    mov ax,4C00h
    int 21h
main ENDP

hello_message BYTE 'Hello, world!',0dh,0ah,'$'

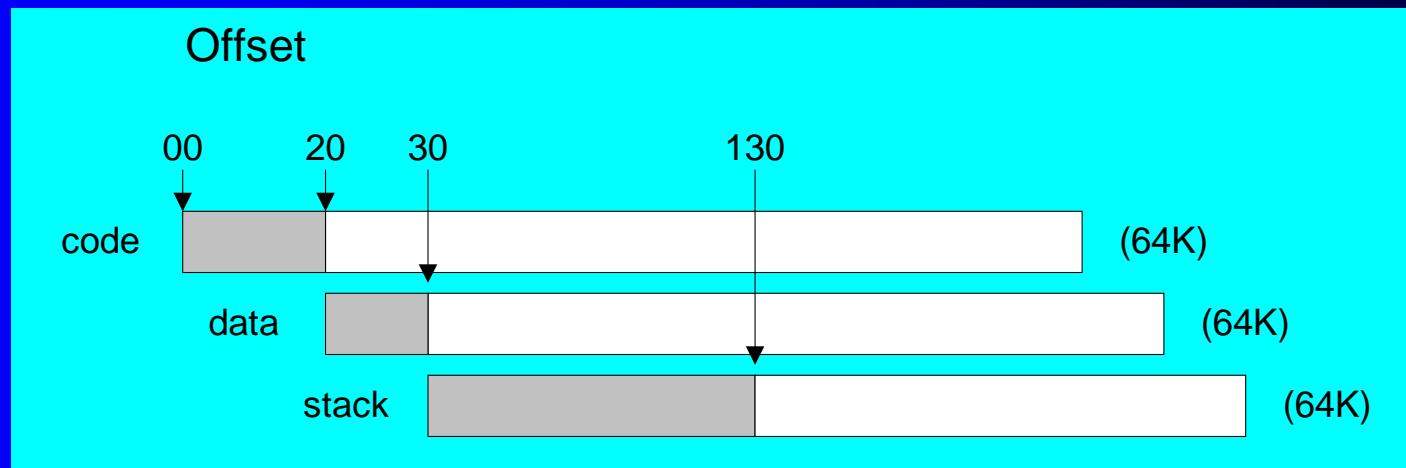
END main
```

EXE Programs

- Use memory more efficiently than COM programs
- Stored on disk in two parts:
 - EXE header record
 - load module (code and data)
- PSP created when loaded into memory
- DS and ES set to the load address
- CS and IP set to code entry point
- SS set to the beginning of the stack segment, and SP set to the stack size

EXE Programs

Sample EXE structure shows overlapping code, data, and stack segments:



EXE Header Record

- A relocation table, containing addresses to be calculated when the program is loaded.
- The file size of the EXE program, measured in 512-byte units.
- Minimum allocation: min number of paragraphs needed above the program.
- Maximum allocation: max number of paragraphs needed above the program.
- Starting IP and SP values.
- Displacement (in paragraphs) of the stack and code segments from the beginning of the load module.
- A checksum of all words in the file, used in catching data errors when loading the program into memory.

What's Next

- Defining Segments
- Runtime Program Structure
- **Interrupt Handling**
- Hardware Control Using I/O Ports

Interrupt Handling

- Overview
- Hardware Interrupts
- Interrupt Control Instructions
- Writing a Custom Interrupt Handler
- Terminate and Stay Resident Programs
- The No_Reset Program

Overview

- Interrupt handler (interrupt service routine) – performs common I/O tasks
 - can be called as functions
 - can be activated by hardware events
- Examples:
 - video output handler
 - critical error handler
 - keyboard handler
 - divide by zero handler
 - Ctrl-Break handler
 - serial port I/O

Interrupt Vector Table

- Each entry contains a 32-bit segment/offset address that points to an interrupt service routine
- Offset = *interruptNumber* * 4
- The following are only examples:

Interrupt Number	Offset	Interrupt Vectors
00-03	0000	02C1:5186 0070:0C67 0DAD:2C1B 0070:0C67
04-07	0010	0070:0C67 F000:FF54 F000:837B F000:837B
08-0B	0020	0D70:022C 0DAD:2BAD 0070:0325 0070:039F
0C-0F	0030	0070:0419 0070:0493 0070:050D 0070:0C67
10-13	0040	C000:0CD7 F000:F84D F000:F841 0070:237D

Hardware Interrupts

- Generated by the Intel 8259 Programmable Interrupt Controller (PIC)
 - in response to a hardware signal
- Interrupt Request Levels (IRQ)
 - priority-based interrupt scheduler
 - brokers simultaneous interrupt requests
 - prevents low-priority interrupt from interrupting a high-priority interrupt

Common IRQ Assignments

- 0 System timer
- 1 Keyboard
- 2 Programmable Interrupt Controller
- 3 COM2 (serial)
- 4 COM1 (serial)
- 5 LPT2 (printer)
- 6 Floppy disk controller
- 7 LPT1 (printer)

Common IRQ Assignments

- 8 CMOS real-time clock
- 9 modem, video, network, sound, and USB controllers
- 10 (available)
- 11 (available)
- 12 mouse
- 13 Math coprocessor
- 14 Hard disk controller
- 15 (available)

Interrupt Control Instructions

- STI – set interrupt flag
 - enables external interrupts
 - always executed at beginning of an interrupt handler
- CLI – clear interrupt flag
 - disables external interrupts
 - used before critical code sections that cannot be interrupted
 - suspends the system timer

Writing a Custom Interrupt Handler

- Motivations
 - Change the behavior of an existing handler
 - Fix a bug in an existing handler
 - Improve system security by disabling certain keyboard commands
- What's Involved
 - Write a new handler
 - Load it into memory
 - Replace entry in interrupt vector table
 - Chain to existing interrupt handler (usually)

Get Interrupt Vector

- INT 21h Function 35h – Get interrupt vector
 - returns segment-offset addr of handler in ES:BX

```
.data
int9Save LABEL WORD
DWORD ?           ; store old INT 9 address here
.code
mov ah,35h          ; get interrupt vector
mov al,9            ; for INT 9
int 21h             ; call MS-DOS
mov int9Save,BX    ; store the offset
mov [int9Save+2],ES ; store the segment
```

Set Interrupt Vector

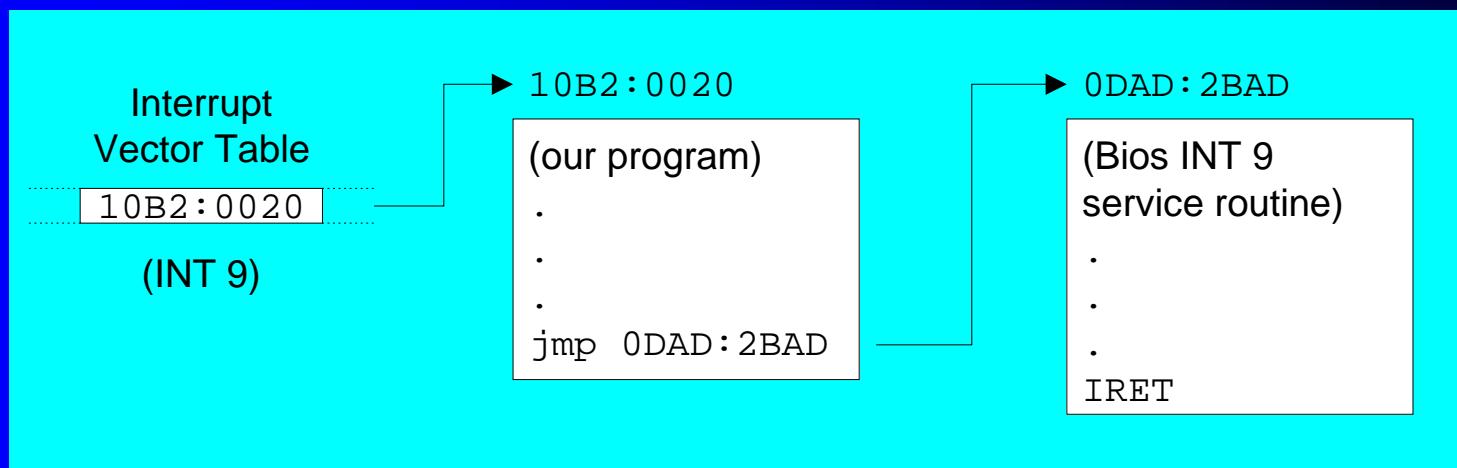
- INT 21h Function 25h – Set interrupt vector
 - installs new interrupt handler, pointed to by DS:DX

```
mov ax,SEG kybd_rtn          ; keyboard handler
mov ds,ax                      ; segment
mov dx,OFFSET kybd_rtn        ; offset
mov ah,25h                      ; set Interrupt vector
mov al,9h                        ; for INT 9h
int 21h
.
.
kybd_rtn PROC                 ; (new handler begins here)
```

See the CtrlBrk.asm program.

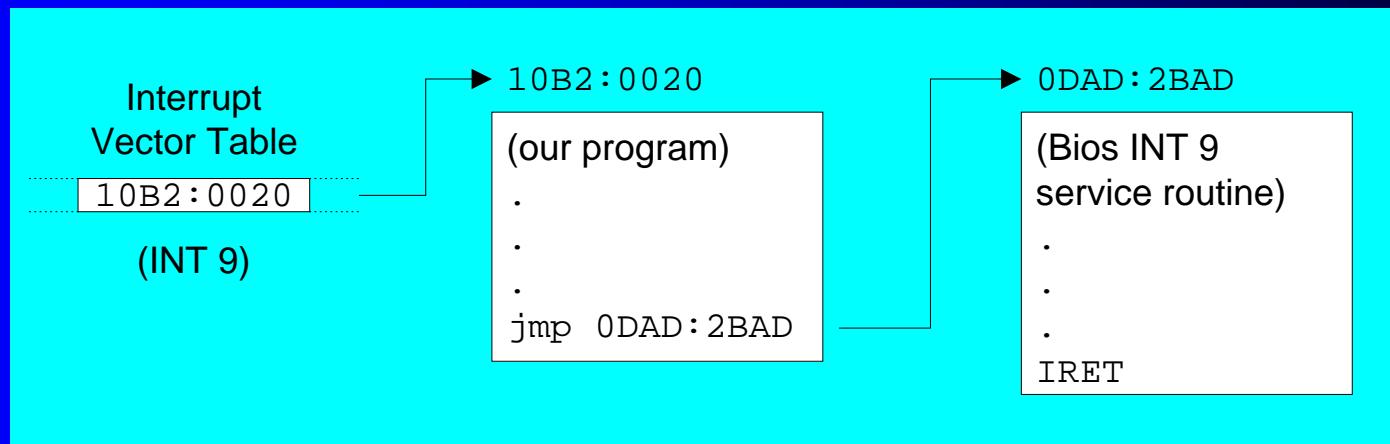
Keyboard Processing Steps

1. Key pressed, byte sent by hardware to keyboard port
2. 8259 controller interrupts the CPU, passing it the interrupt number
3. CPU looks up interrupt vector table entry 9h, branches to the address found there



Keyboard Processing Steps

4. Our handler executes, intercepting the byte sent by the keyboard
5. Our handler jumps to the regular INT 9 handler
6. The INT 9h handler finishes and returns
7. System continues normal processing



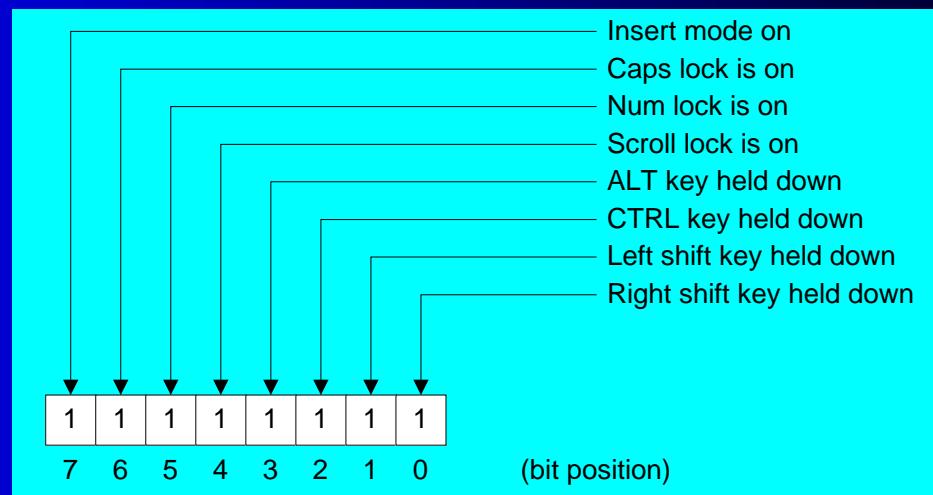
Terminate and Stay Resident Programs

- (TSR): Installed in memory, stays there until removed
 - by a removal program, or by rebooting
- Keyboard example
 - replace the INT 9 vector so it points to our own handler
 - check, or filter certain keystroke combinations, using our handler
 - forward-chain to the existing INT 9 handler to do normal keyboard processing

The No_Reset Program (1 of 5)

- Inspects each incoming key
- If the Del key is received,
 - checks for the Ctrl and Alt keys
 - permits a system reset only if the Right shift key is also held down

The keyboard status byte indicates the current state of special keys:



The No_Reset Program (2 of 5)

- [View the source code](#)
- Resident program begins with:

```
int9_handler PROC FAR
    sti                      ; enable hardware interrupts
    pushf                   ; save regs & flags
    push es
    push ax
    push di
```

The No_Reset Program (3 of 5)

- Locate the keyboard flag byte and copy into AH:

```
L1: mov  ax,40h          ; DOS data segment is at 40h
    mov  es,ax
    mov  di,17h          ; location of keyboard flag
    mov  ah,es:[di]       ; copy keyboard flag into AH
```

- Check to see if the Ctrl and Alt keys are held down:

```
L2: test ah,ctrl_key   ; Ctrl key held down?
    jz   L5
    test ah,alt_key     ; ALT key held down?
    jz   L5
```

The No_Reset Program (4 of 5)

- Test for the Del and Right shift keys:

```
L3: in    al,kybd_port      ; read keyboard port
     cmp   al,del_key        ; Del key pressed?
     jne   L5                ; no: exit
     test  ah,rt_shift       ; right shift key pressed?
     jnz   L5                ; yes: allow system reset
```

- Turn off the Ctrl key and write the keyboard flag byte back to memory:

```
L4: and  ah,NOT ctrl_key   ; turn off bit for CTRL
     mov   es:[di],ah        ; store keyboard_flag
```

The No_Reset Program (5 of 5)

- Pop the flags and registers off the stack and execute a far jump to the existing BIOS INT 9h routine:

```
L5: pop    di          ; restore regs & flags  
      pop    ax  
      pop    es  
      popf  
      jmp   cs:[old_interrupt9] ; jump to INT 9 routine
```

What's Next

- Defining Segments
- Runtime Program Structure
- Interrupt Handling
- **Hardware Control Using I/O Ports**

Hardware Control Using I/O Ports

- Two types of hardware I/O
 - memory mapped
 - program and hardware device share the same memory address, as if it were a variable
 - port based
 - data written to port using the OUT instruction
 - data read from port using the IN instruction

Input-Output Ports

- ports numbered from 0 to FFFFh
- keyboard controller chip sends 8-bit scan code to port 60h
 - triggers a hardware interrupt 9
- IN and OUT instructions:
IN accumulator, port
OUT port, accumulator
 - accumulator is AL, AX, or EAX
 - port is a constant between 0 and FFh, or a value in DX between 0 and FFFFh

PC Sound Program

- Generates sound through speaker
- speaker control port: 61h
- Intel 8255 Programmable Peripheral Interface chip turns the speaker on and off
- Intel 8253 Timer chip controls the frequency
- [Source code](#)

Summary

- Explicit segment definitions used often in custom code libraries
- Directives: SEGMENT, ENDS, ASSUME
- Transient programs
- Program segment prefix (PSP)
- Interrupt handlers, interrupt vector table
- Hardware interrupt, 8259 Programmable Interrupt Controller, interrupt flag
- Terminate and Stay Resident (TSR)
- Memory-mapped and port-based I/O

The End

