CHAPTER 3: ASSEMBLY LANGUAGE FUNDAMENTALS

Chapter Overview

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

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

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

; This program adds and subtracts 32-bit integers.
.386
.MODEL flat,stdcall
.STACK 4096

ExitProcess PROTO, dwExitCode:DWORD
DumpRegs PROTO

.code
main PROC
   mov eax,10000h              ; EAX = 10000h
   add eax,40000h              ; EAX = 50000h
   sub eax,20000h              ; EAX = 30000h
   call DumpRegs
   INVOKE ExitProcess,0
main ENDP
END main
```

Example Output

Program output, showing registers and flags:

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

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: begin at column 40-45, aligned vertically
  - 1-3 spaces between instruction and its operands
    - ex: mov ax,bx
  - 1-2 blank lines between procedures

Program Template (review)

```
; Program Template                    (Template.asm)

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

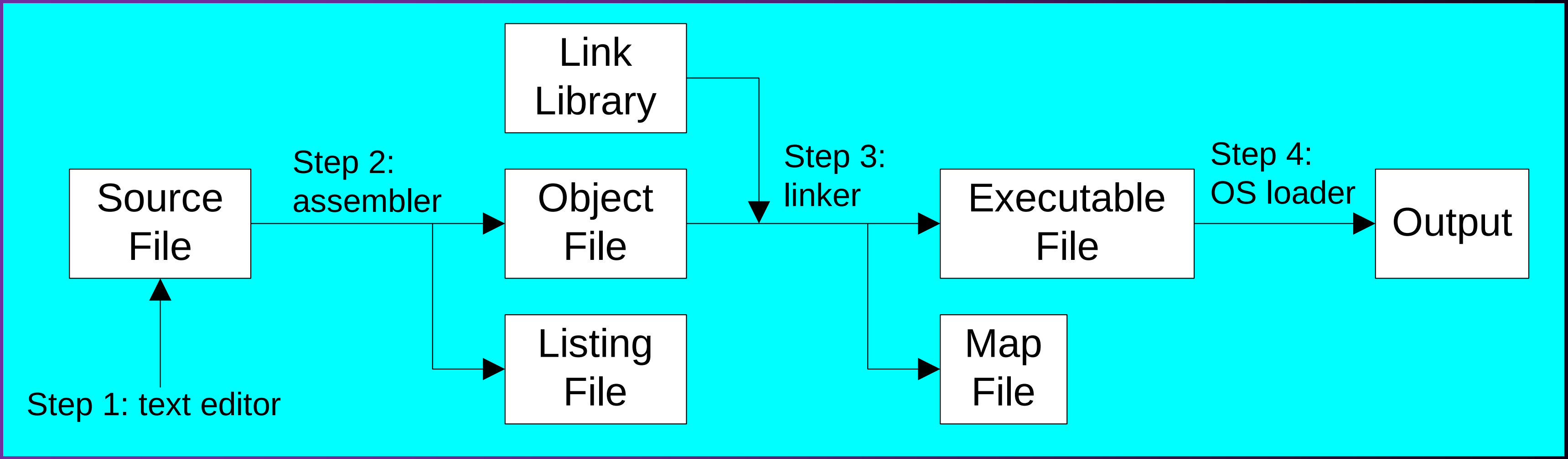
```
.386
.model flat,stdcall
.stack 4096
ExitProcess PROTO, dwExitCode:DWORD

.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

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.



- The assembler contains a ***preprocessor*** to process directives, etc.

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)
- Example: See § 3.3

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 (内部数据类型)
- 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*] . . .

```
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:

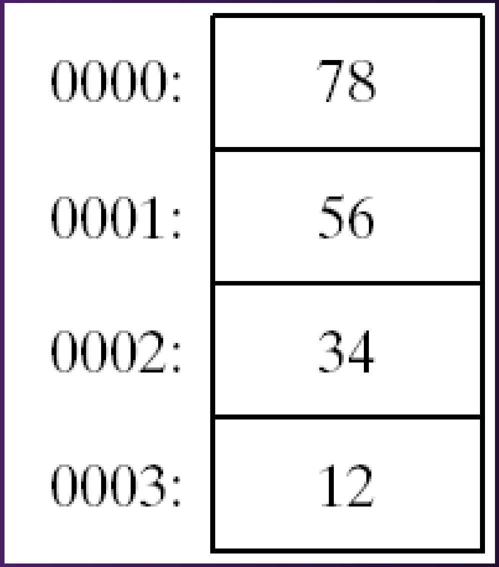
```
quad1 QWORD  1234567812345678h
val1  TBYTE  1000000000123456789Ah
rVal1 REAL4  -2.1
rVal2 REAL8  3.2E-260
rVal3 REAL10 4.6E+4096
ShortArray REAL4 20 DUP(0.0)
```

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
```

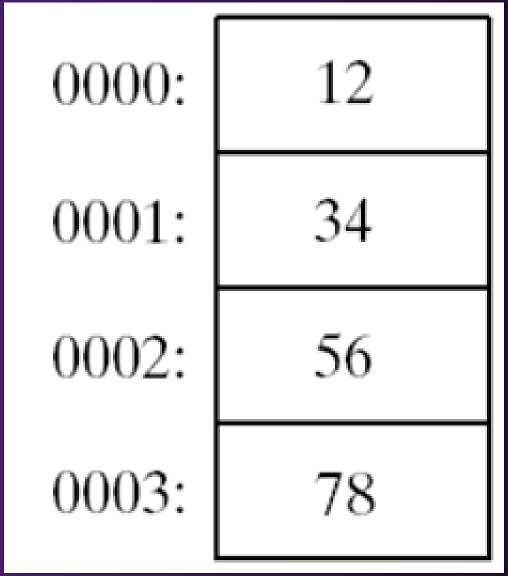


Big Endian Order

- All data types larger than a byte store their individual bytes in "usual" order. The most significant byte occurs at the first (lowest) memory address.

- Example:

```
val1 DWORD 12345678h
```



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 Unitialized 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**
- Real-Address Mode Programming
- 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 al,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 a number 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

```
; AddTwoSum_64.asm - Chapter 3 example.
ExitProcess PROTO
.data
sum QWORD 0
.code
main  PROC
   mov  rax,5
   add  rax,6
   mov  sum,rax

   mov  ecx,0
   call ExitProcess
main ENDP
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

CHAPTER 4: DATA TRANSFERS, ADDRESSING, AND ARITHMETIC

Chapter Overview

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

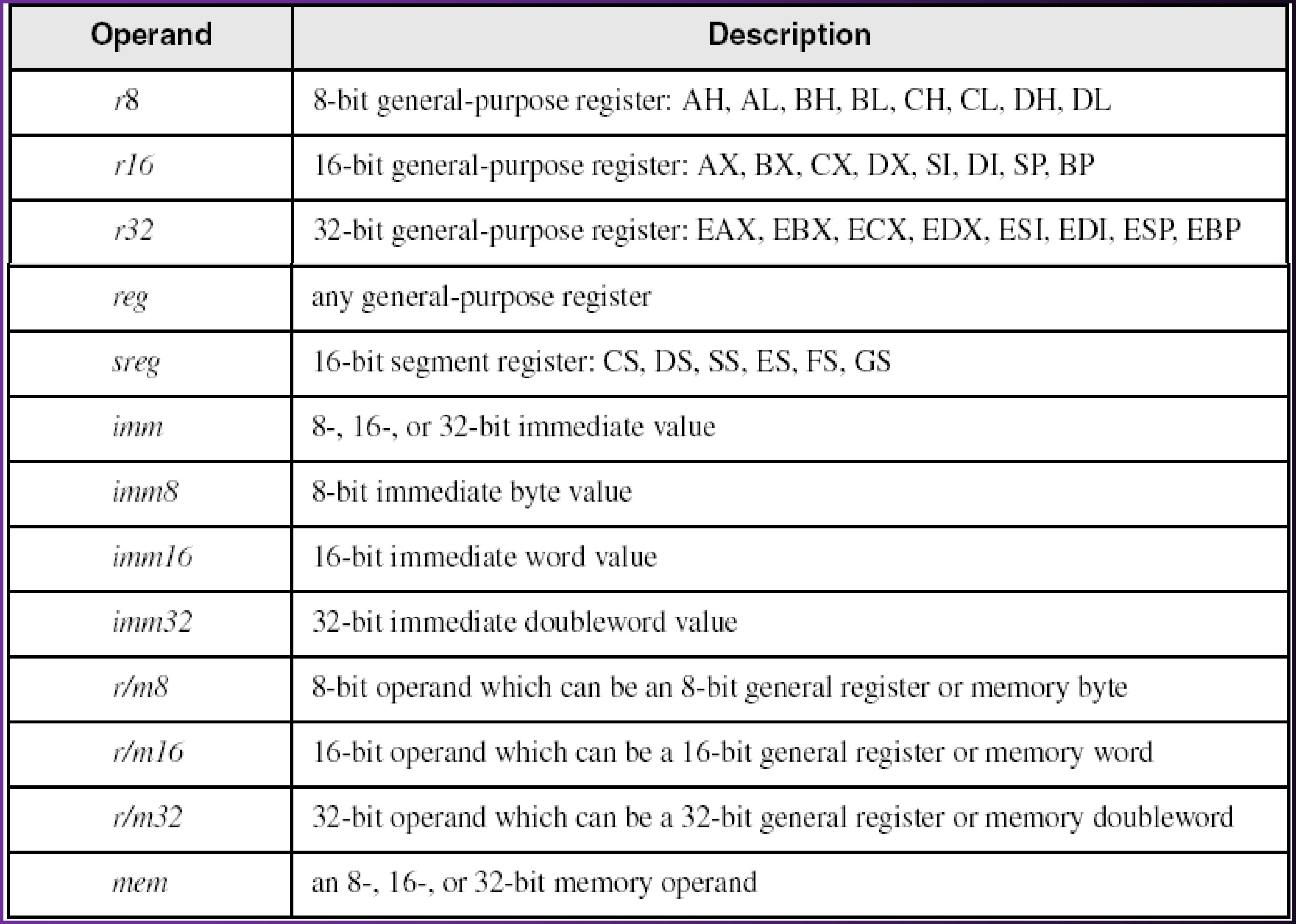
Data Transfer Instructions

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

Operand Types

- Three basic types of operands:
  - 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
r8	8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL
r16	16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP
r32	32-bit general-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
reg	any general-purpose register
sreg	16-bit segment register: CS, DS, SS, ES, FS, GS
imm	8-, 16-, or 32-bit immediate value
imm8	8-bit immediate byte value
imm16	16-bit immediate word value
imm32	32-bit immediate doubleword value
r/m8	8-bit operand which can be an 8-bit general register or memory byte
r/m16	16-bit operand which can be a 16-bit general register or memory word
r/m32	32-bit operand which can be a 32-bit general register or memory doubleword
mem	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 (解引用) by the assembler

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

alternate format

MOV Instruction

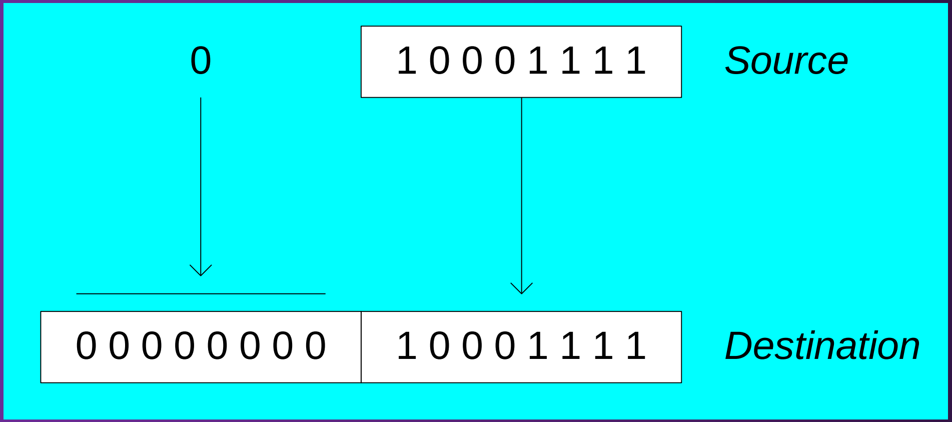
- Move from source to destination. Syntax:

  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
```

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.

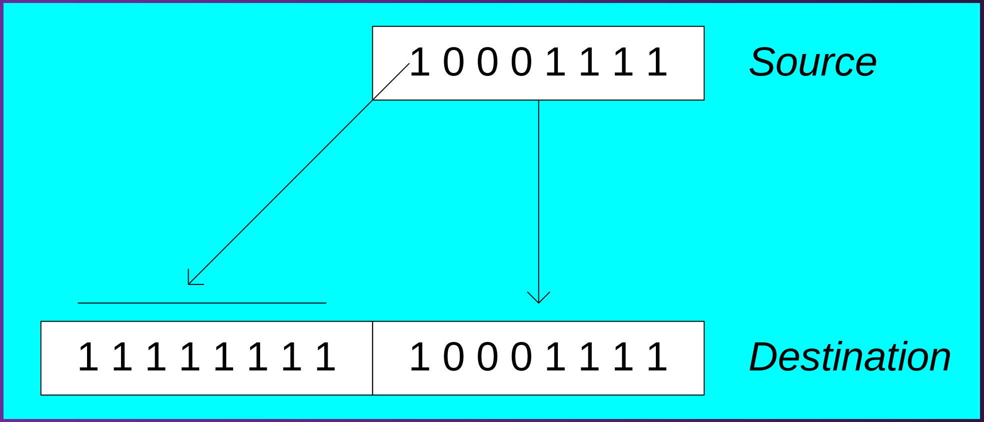


```
mov bl,10001111b
movzx ax,bl                      ; zero-extension
```

- The destination must be a register.
- The source cannot be immediate.

Sign Extension

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



```
mov bl,10001111b
movsx ax,bl                         ; sign extension
```

- The destination must be a register.
- The source cannot be immediate.

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 (直接偏移操作数)

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?

What's Next

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

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: *destination* $\leftarrow$ *destination* + 1
- DEC *destination*
    - Logic: *destination* $\leftarrow$ *destination* – 1

INC and DEC Examples

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

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

ADD and SUB Instructions

- ADD destination, source
  - Logic: *destination* ← *destination* + source
- SUB destination, source
  - Logic: *destination* ← *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
```

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
```

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.

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

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
```

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
```

What's Next

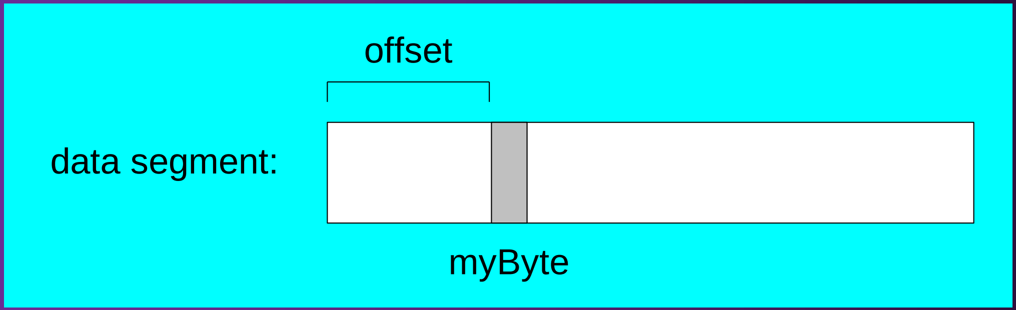
- Data Transfer Instructions
- Addition and Subtraction
- **Data-Related Operators and Directives**
- Indirect Addressing
- JMP and LOOP Instructions
- 64-Bit Programming

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 only have a single segment (we use the 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
```

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
```

Recall that little endian order is used when storing data in memory (see Section 3.4.11 in the 7th Ed).

Little Endian Order

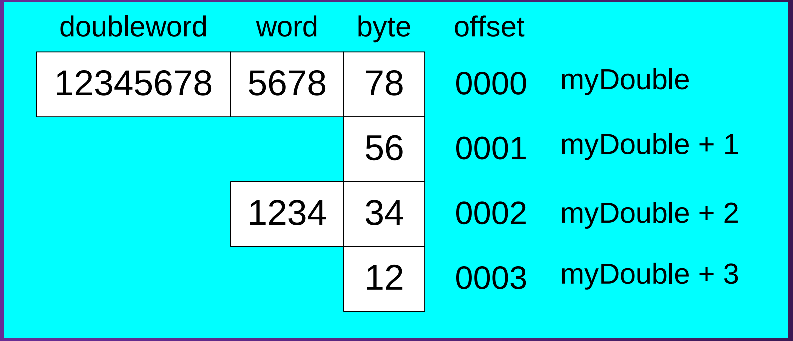
- Little endian order refers to the way Intel stores integers in memory.
- Multi-byte integers are stored in reverse order, with the least significant byte stored at the lowest address
- For example, the doubleword 12345678h would be stored as:

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
```



```
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
```

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 (间接寻址)

- 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
```

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 (变址操作数)

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

[*label* + *reg*]   *label*[*reg*]

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

ToDo: Modify this example for an array of doublewords.

Index Scaling (索引比例)

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 ← *target*
- Example:

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

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

LOOP Instruction

- The LOOP instruction creates a counting loop
- Syntax: LOOP *target*
- Logic:
  - ECX ← ECX – 1
  - if ECX != 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 00 00                mov  ax,0
00000004    B9 05 00 00 00             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 the current location, causing a jump to location 00000009:

$$00000009 \leftarrow 0000000E + FB$$

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
```

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.

CHAPTER 5: PROCEDURES

Chapter Overview

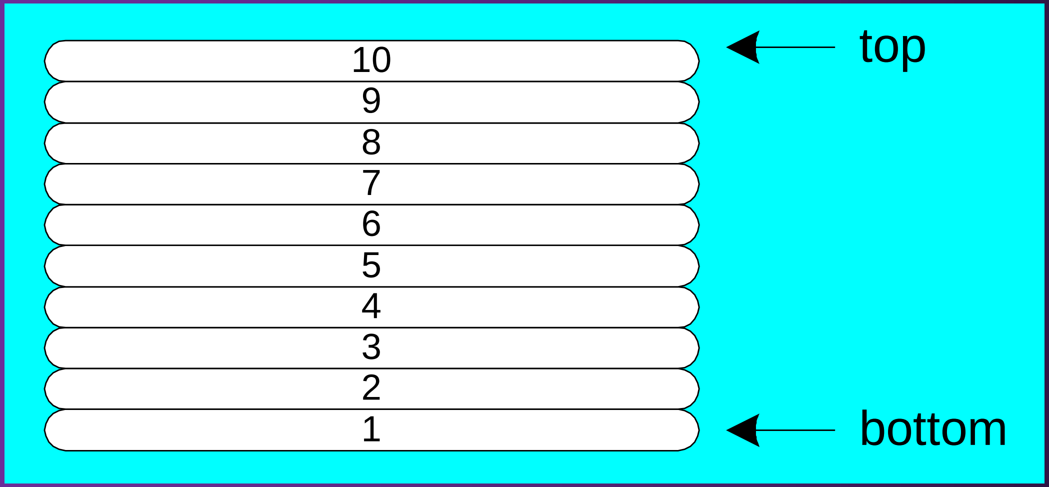
- **Stack Operations**
- Defining and Using Procedures
- Linking to an External Library
- The Irvine32 Library
- Program Design Using Procedures
- 64-Bit Assembly Programming

Stack Operations

- Runtime Stack
- PUSH Operation
- POP Operation
- PUSH and POP Instructions
- Using PUSH and POP
- Related Instructions

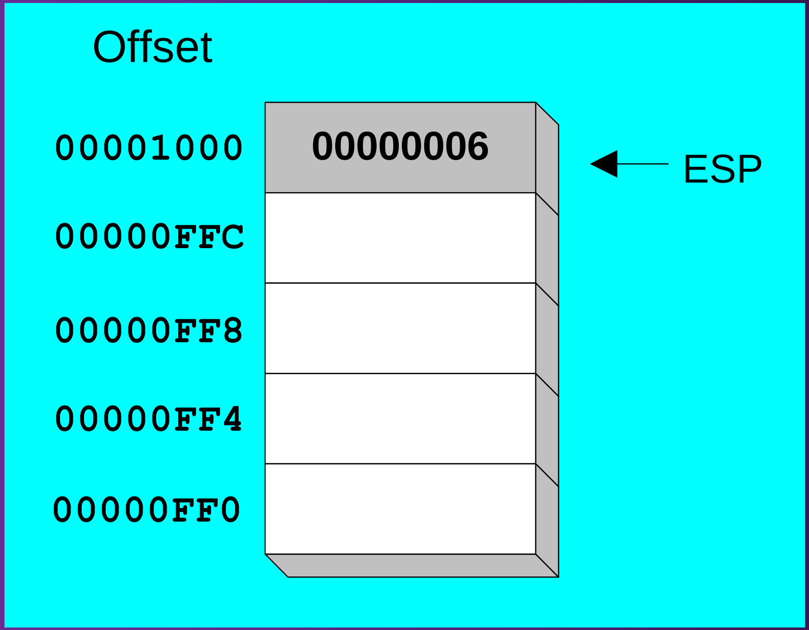
Runtime Stack

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



Runtime Stack

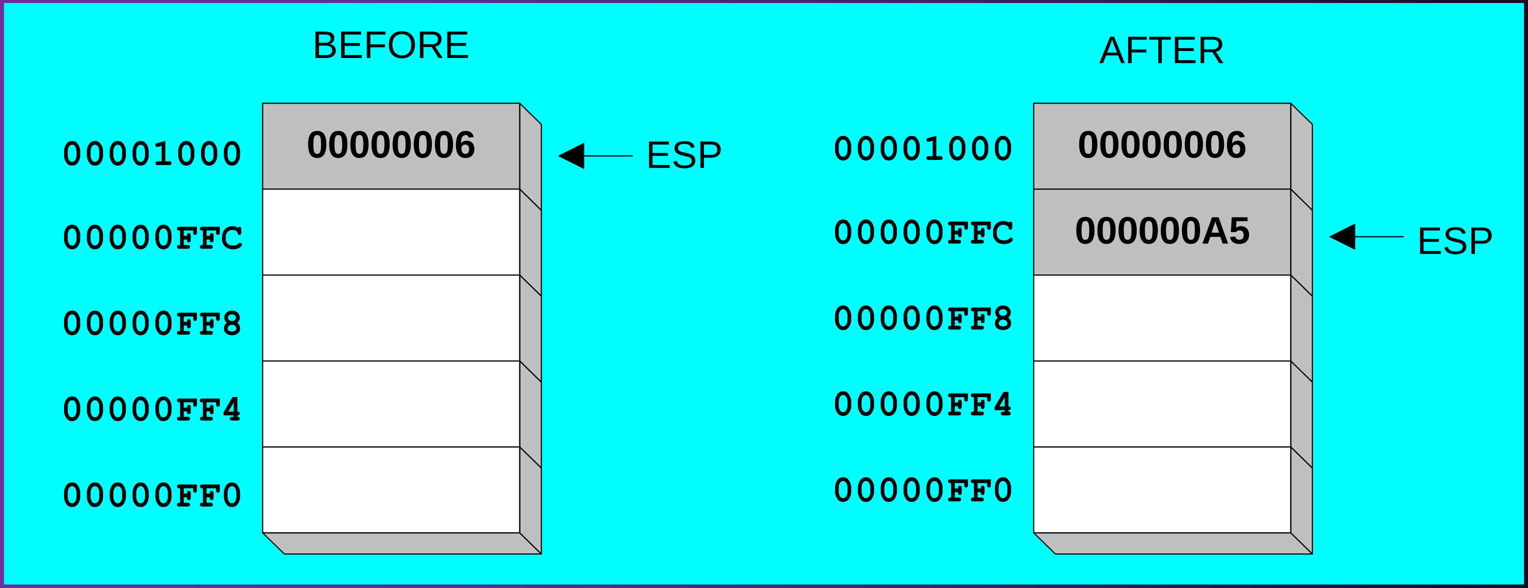
- Managed by the CPU, using two registers
  - SS (stack segment)
  - ESP (stack pointer) *



* SP in Real-address mode

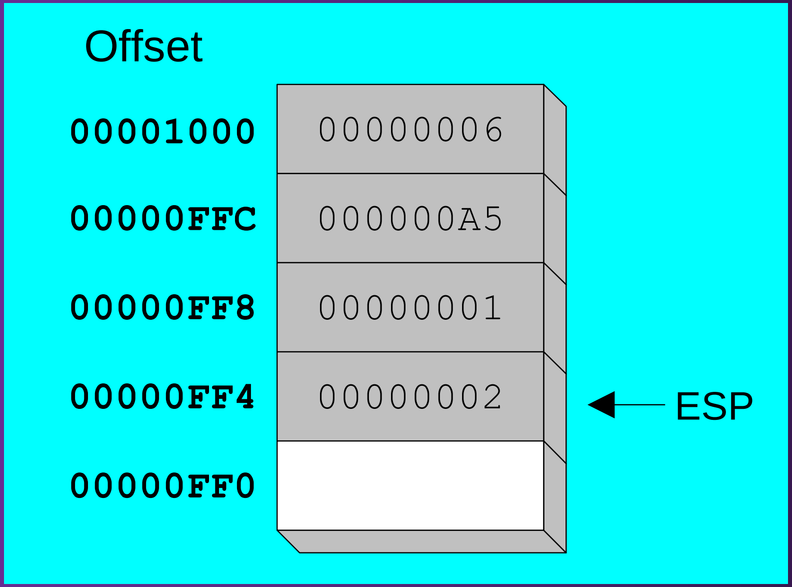
PUSH Operation (1 of 2)

- 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)

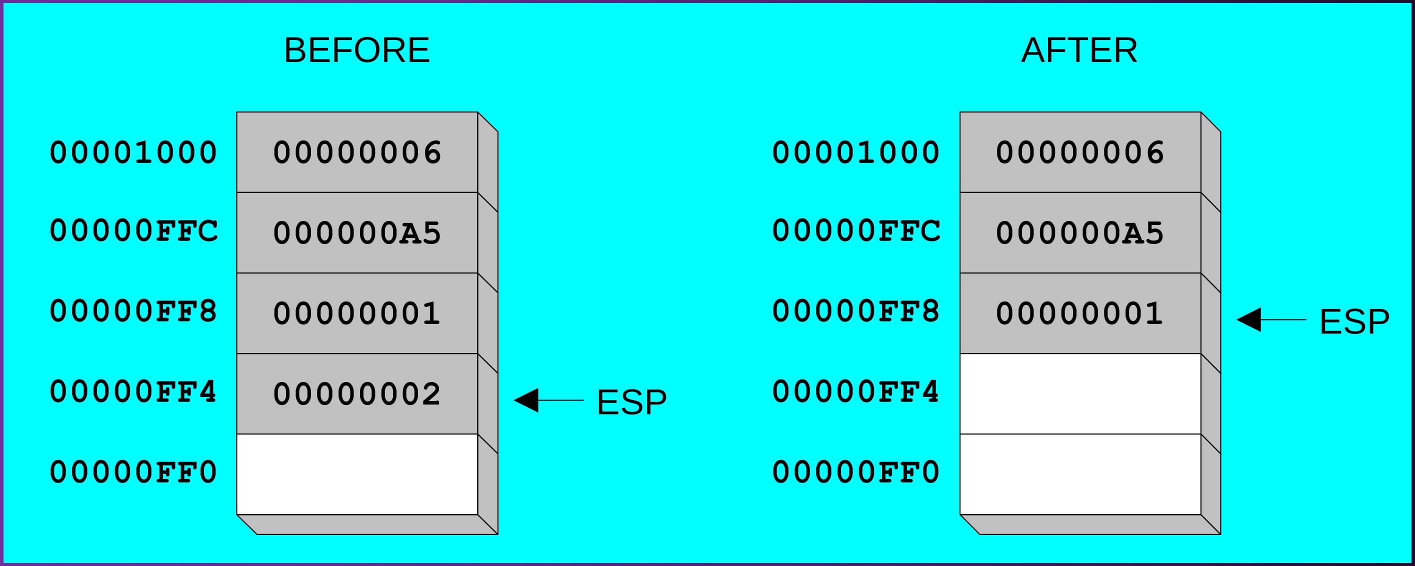
- Same stack after pushing two more integers:



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
```

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

What's Next

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

Defining and Using Procedures

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

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
```

To call SumOf, prepare arguments: EAX, EBX, ECX

```
        mov eax,10
        mov ebx,20
        mov ecx,30
        call SumOf
        WriteDec
```

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)

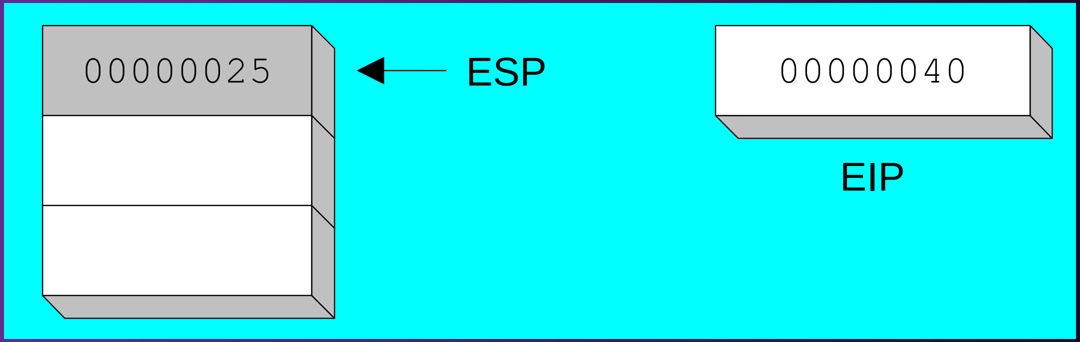
0000025 is the offset of the instruction immediately following the CALL instruction

00000040 is the offset of the first instruction inside MySub

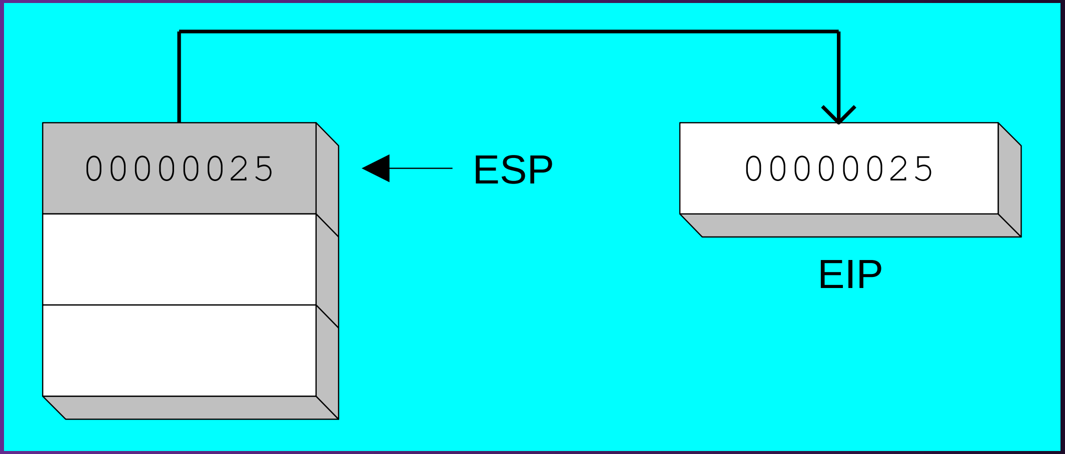
```
main PROC
    00000020 call MySub
    00000025 mov eax,ebx
    .
    .
main ENDP

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)

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**
- Program Design Using Procedures
- 64-Bit Assembly Programming

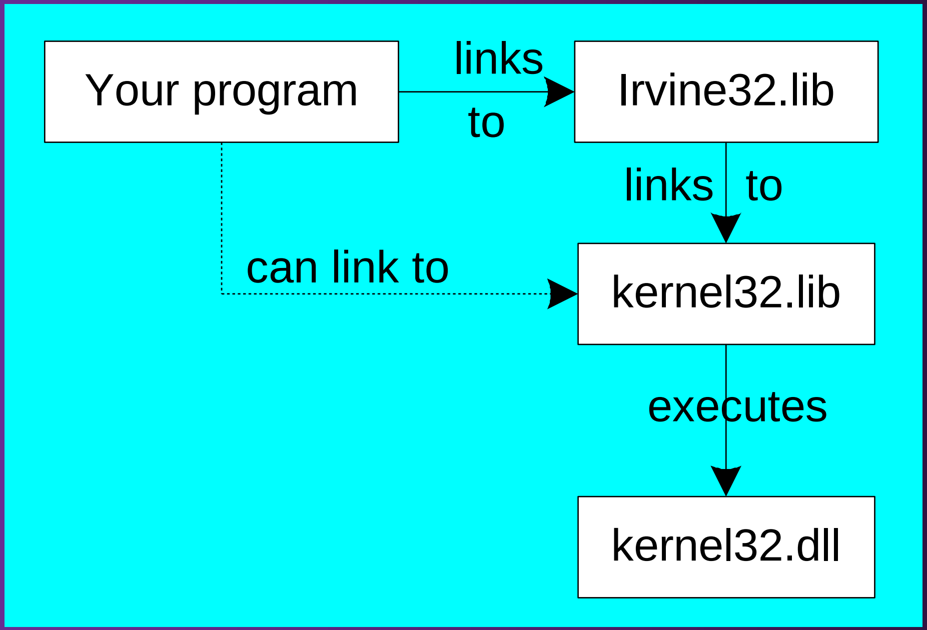
What is a Link Library?

- 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

Take a quick look at Irvine32.asm

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)*



Calling Irvine32 Library Procedure

- 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
```

What's Next

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

Program Design Using Procedures

- Top-Down Design (functional decomposition) involves the following:
  - design your program before starting to code
  - break large tasks into smaller ones
  - use a hierarchical structure based on procedure calls
  - test individual procedures separately

What's Next

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

64-Bit Assembly Programming

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

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 (Chap 3)

- 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 ($)

Summary (Chap 4)

- 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

Summary (Chap 5)

- 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

Homework

- Reading Chap 3 -- 5
- Homework
  - Coding Exercises

Thanks!