雨课堂 Rain Classroom

本次课程是

线上+线下

融合式教学

请**现场**的同学们：

1. 打开雨课堂，点击页面右下角喇叭按钮调至静音状态

请**远程上课**的同学们：

1. 打开雨课堂，点击页面右下角喇叭按钮调至静音状态
2. 打开“腾讯会议”（会议室：824 8461 5333），进入会议室，并关闭麦克风

CHAPTER 4: DATA TRANSFERS, ADDRESSING, AND ARITHMETIC

What's Next

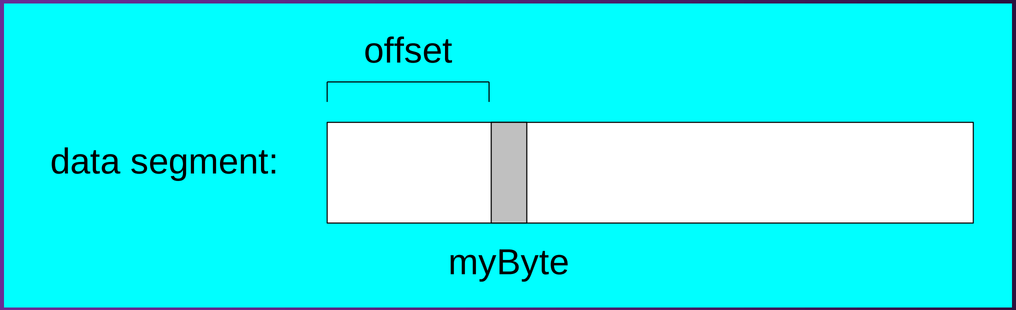
- Data Transfer Instructions
- Addition and Subtraction
- **Data-Related Operators and Directives**
- Indirect Addressing
- JMP and LOOP Instructions
- 64-Bit Programming

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 **`bVal`** is located at offset 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
```

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
```

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 ← 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

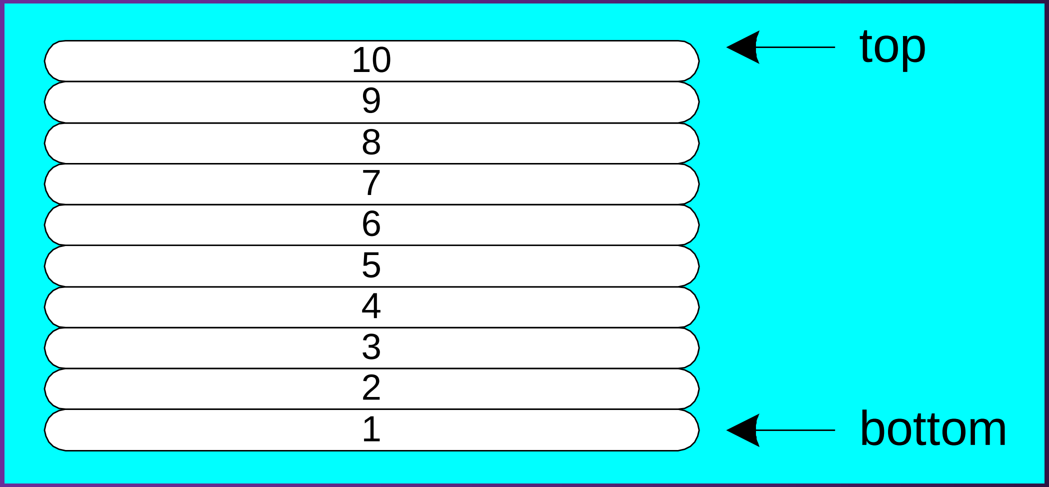
- **Stack Operations**
- Defining and Using Procedures
- Linking to an External Library
- The Irvine32 Library
- Program Design Using Procedures
- 64-Bit Assembly Programming

Stack Operations

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

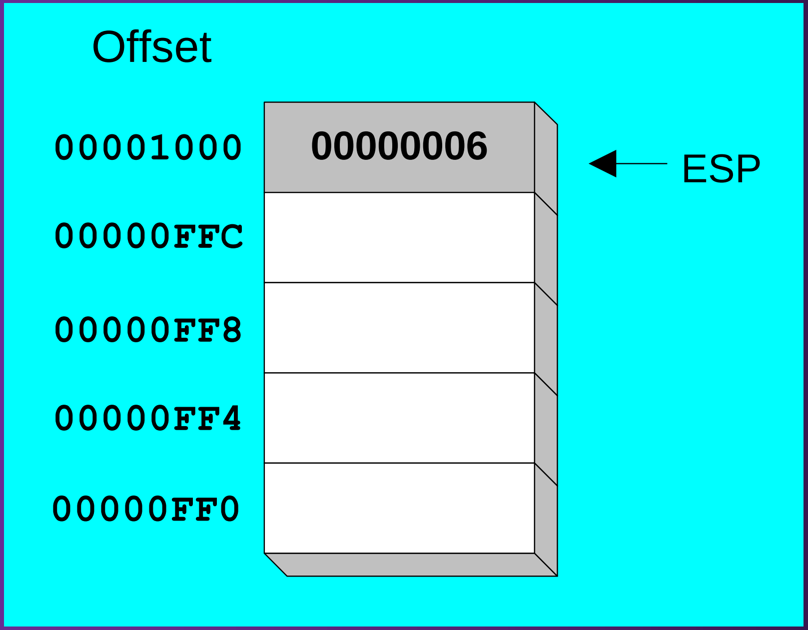
Runtime Stack

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



Runtime Stack

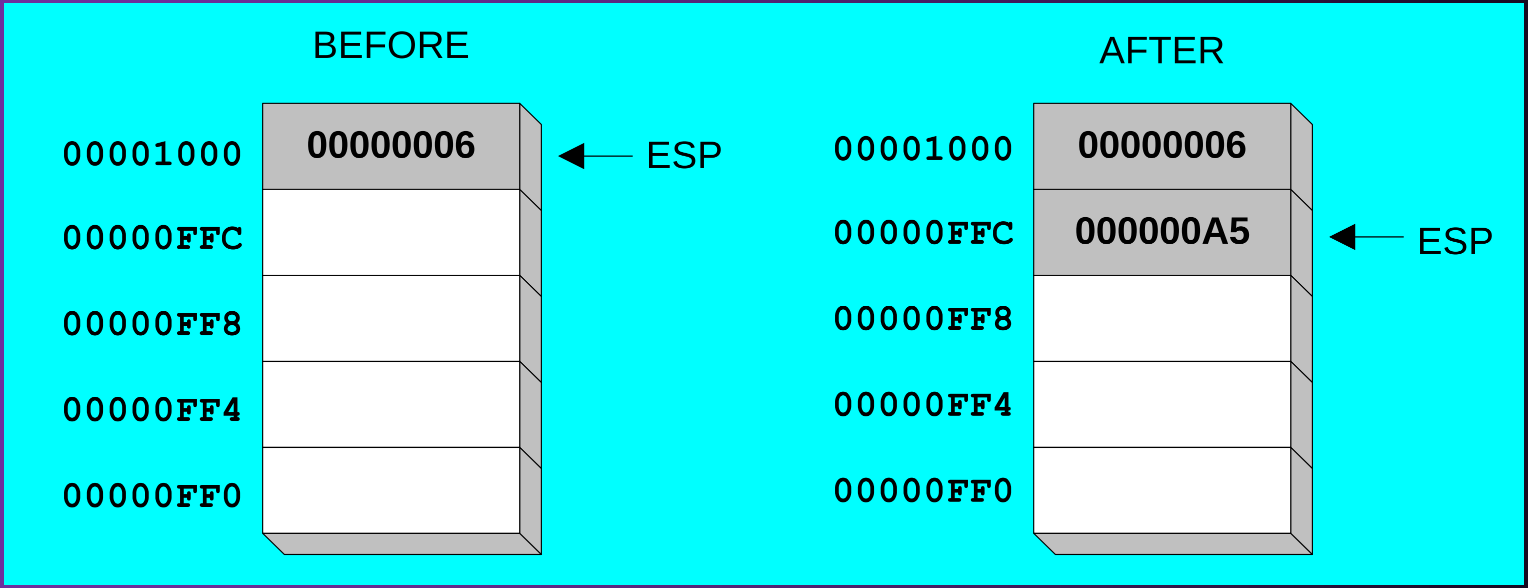
- Managed by the CPU, using two registers
  - SS (stack segment)
  - ESP (stack pointer) *



* SP in Real-address mode

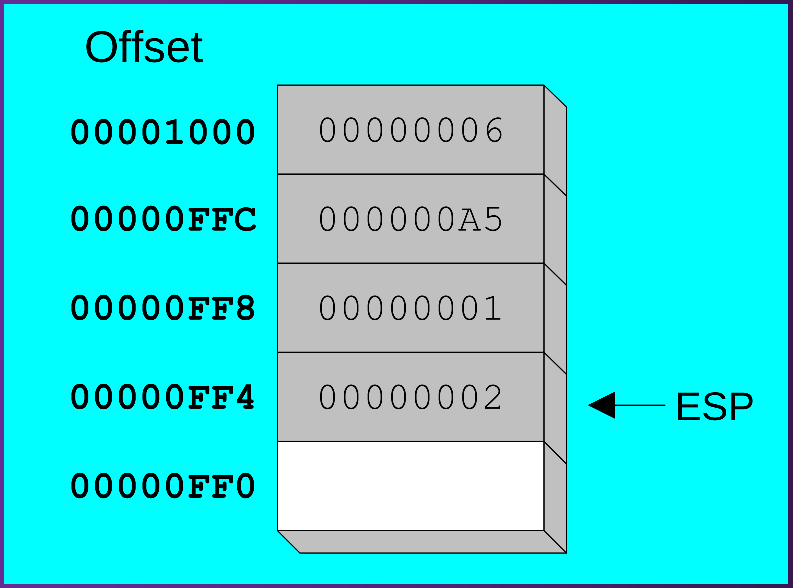
PUSH Operation (1 of 2)

- 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)

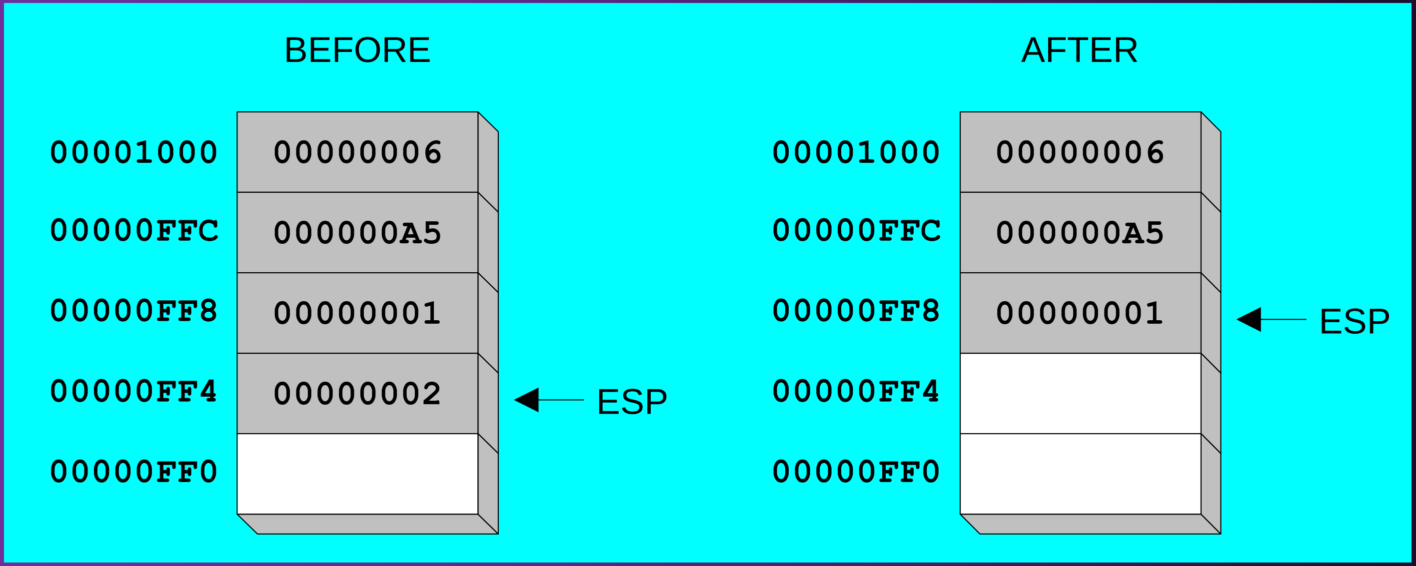
- Same stack after pushing two more integers:



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)

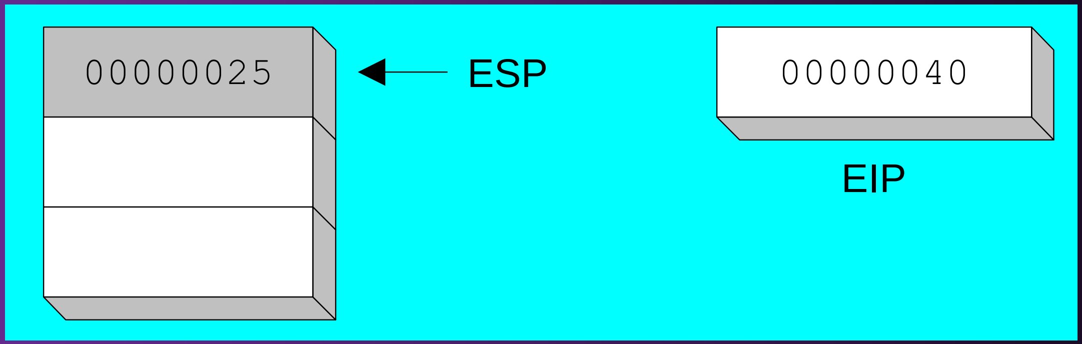
0000025 is the offset of the instruction immediately following the CALL instruction

00000040 is the offset of the first instruction inside MySub

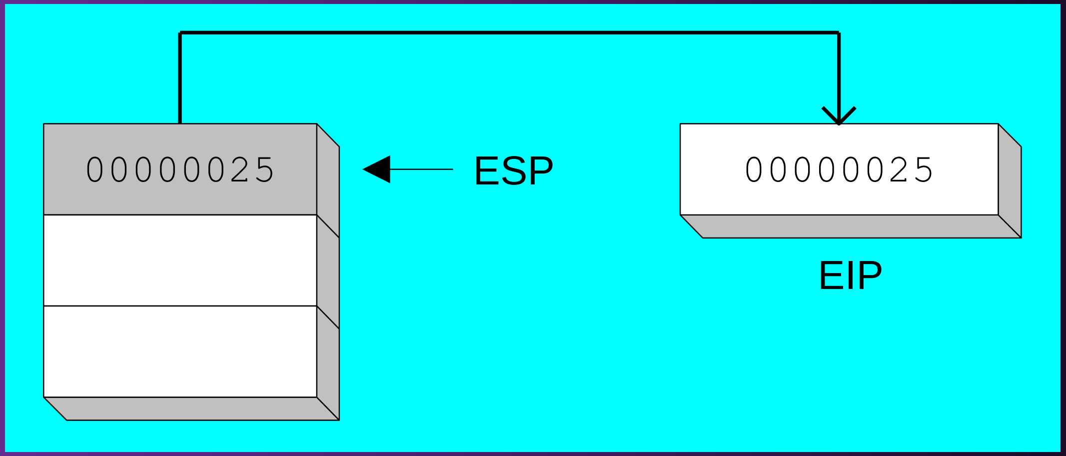
```
main PROC
    00000020 call MySub
    00000025 mov eax,ebx
    .
    .
main ENDP

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
```

Summary (Chap 4)

- Operators
  - OFFSET, PTR, TYPE, LENGTHOF, SIZEOF, TYPEDEF
- Operand types
  - indirect, indexed
- 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

Homework

- Reading Chap 4 -- 5

- Homework
  - Coding Exercises

- Project Assignment

Happy National Day!