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 7:
INTEGER ARITHMETIC

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

MUL Examples

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

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

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

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

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

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.

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 r/m8
DIV r/m16
DIV r/m32
```

Default Operands:

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

DIV Examples

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

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

Same division, using 32-bit operands:

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

64-Bit DIV Example

Divide 000001080000000033300020h by 00010000h:

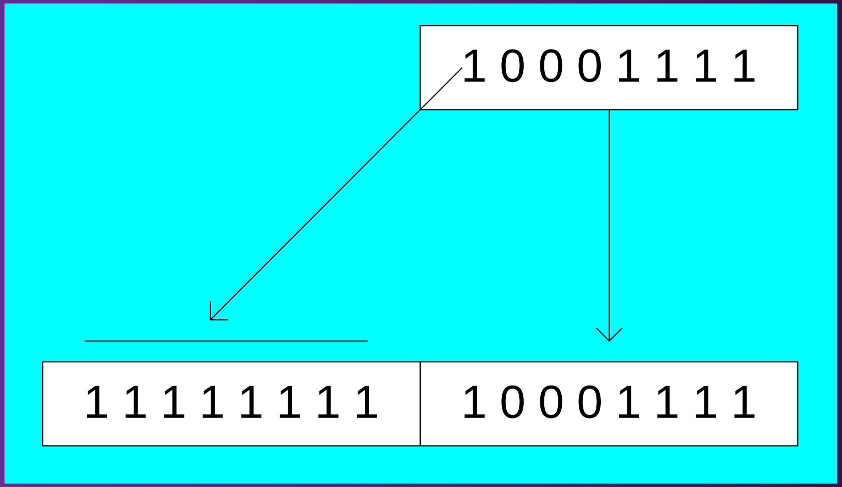
```
.data
dividend_hi QWORD 00000108h
dividend_lo QWORD 33300020h
divisor QWORD 00010000h

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

RAX (quotient):   0108000000003330h
RDX (remainder): 0000000000000020h

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:

```
mov eax,0FFFFFF9Bh      ; (-101)
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
```

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

7.3 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.

CHAPTER 8: ADVANCED PROCEDURES

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

Stack Frame (堆栈框架, 栈帧)

- Also known as an *activation record* (活动记录)
- Area of the stack set aside for a procedure's passed parameters, return address, 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
```

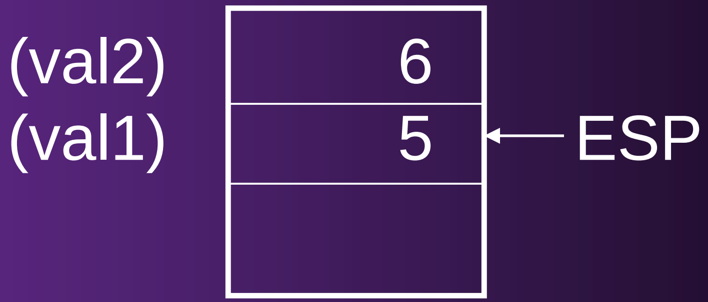
Passing Arguments by Value

- 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
```



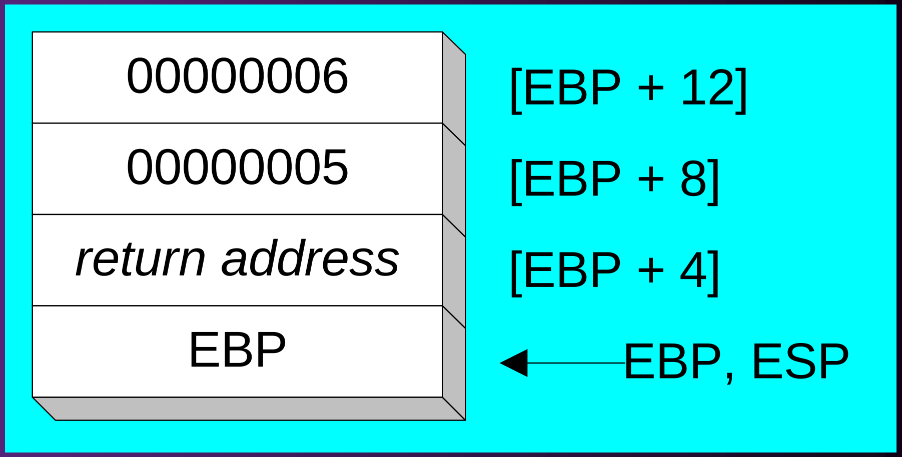
Stack prior to CALL

Passing Arguments by Value: AddTwo

int n = AddTwo(5, 6);

```
.data
sum DWORD ?
.code
    push 6                    ; second argument
    push 5                    ; first argument
    call AddTwo               ; EAX = sum
    mov  sum,eax              ; save the sum
```

```
AddTwo PROC
    push ebp
    mov  ebp,esp
    .
    .
```



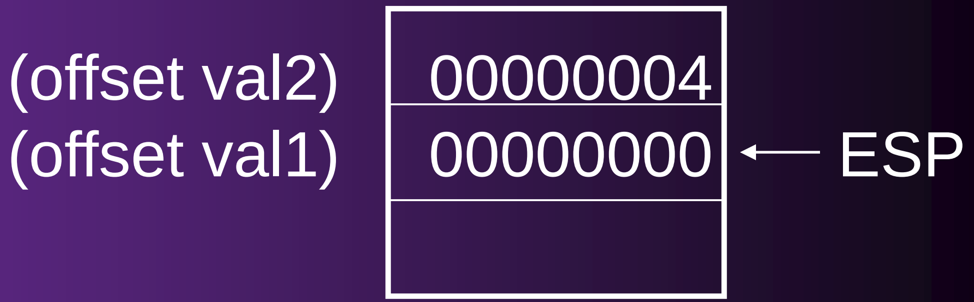
Passing by Reference

- 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
```



Stack prior to CALL

Stack after the CALL

value or addr of val2	[EBP+12]
value or addr of val1	[EBP+8]
return address	[EBP+4]
EBP	← ESP, EBP

Accessing Stack Parameters (C/C++)

- C and C++ functions access stack parameters using constant offsets from EBP[1].
  - 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

Stack Frames

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

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)

C Call : Caller releases stack

RET does not clean up the stack.

```
AddTwo_C PROC
    push ebp
    mov  ebp,esp
    mov  eax,[ebp + 12]     ; second parameter
    add  eax,[ebp + 8]      ; first parameter
    pop  ebp
    ret                     ; caller cleans up the stack
AddTwo_C ENDP

_Example1 PROC
        push 6
        push 5
        call AddTwo_C
        add  esp,8          ; clean up the stack
        call DumpRegs       ; sum is in EAX
        ret
_Example1 ENDP
```

STDCall : Procedure releases stack

The RET n instruction cleans up the stack.

```
AddTwo PROC
    push ebp
    mov  ebp,esp
    mov  eax,[ebp + 12]     ; second parameter
    add  eax,[ebp + 8]      ; first parameter
    pop  ebp
    ret  8                  ; clean up the stack
AddTwo ENDP

_Example2 PROC
        push 6
        push 5
        call AddTwo
        call DumpRegs       ; sum is in EAX
        ret
_Example2 ENDP
```

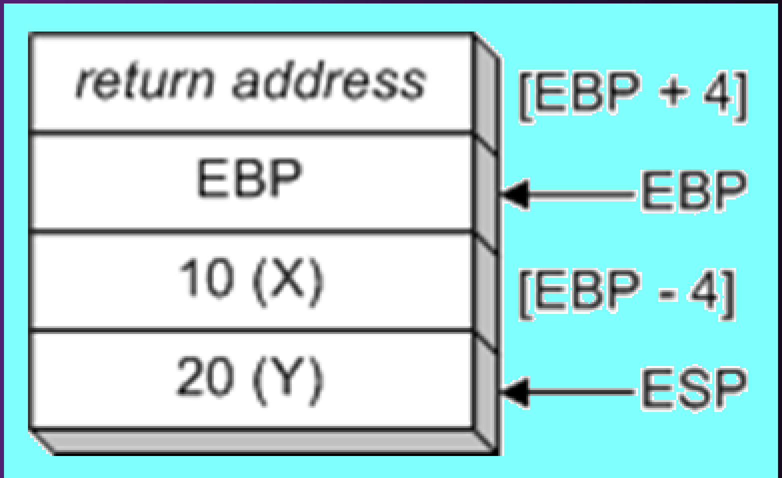
Local Variables

- 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

Local Variables

To explicitly create local variables, subtract total size from ESP.

```
void MySub()
{
    int X=10;
    int Y=20;
}
```




```
MySub PROC
      push   ebp
      mov    ebp,esp
      sub    esp,8                        ; create variables
      mov    DWORD PTR [ebp-4],10         ; X
      mov    DWORD PTR [ebp-8],20         ; Y
      ; ... Do something
      mov    esp,ebp            ; remove locals from stack
      pop    ebp
      ret
MySub ENDP
```

LocalVars.asm

ENTER and LEAVE

- ENTER instruction creates stack frame for a called procedure
  - pushes EBP on the stack (*push ebp*)
  - sets EBP to the base of the stack frame (*mov ebp, esp*)
  - reserves space for local variables (*sub esp, n*)
  - Syntax: ENTER numBytesReserved, nestingLevel (=0)

- LEAVE instruction terminates the stack frame for a called procedure
  - restores ESP to release local variables (*mov esp, ebp*)
  - pops EBP for the caller (*pop ebp*)

LEAVE Instruction

Terminates the stack frame for a procedure.

Equivalent operations

```
MySub PROC
        enter 8,0
        ...
        ...
        ...
        leave
        ret
MySub ENDP
```

```
push   ebp
mov    ebp,esp
sub    esp,8      ; 2 local DWORDs
```

```
mov    esp,ebp   ; free local space
pop    ebp
```

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:

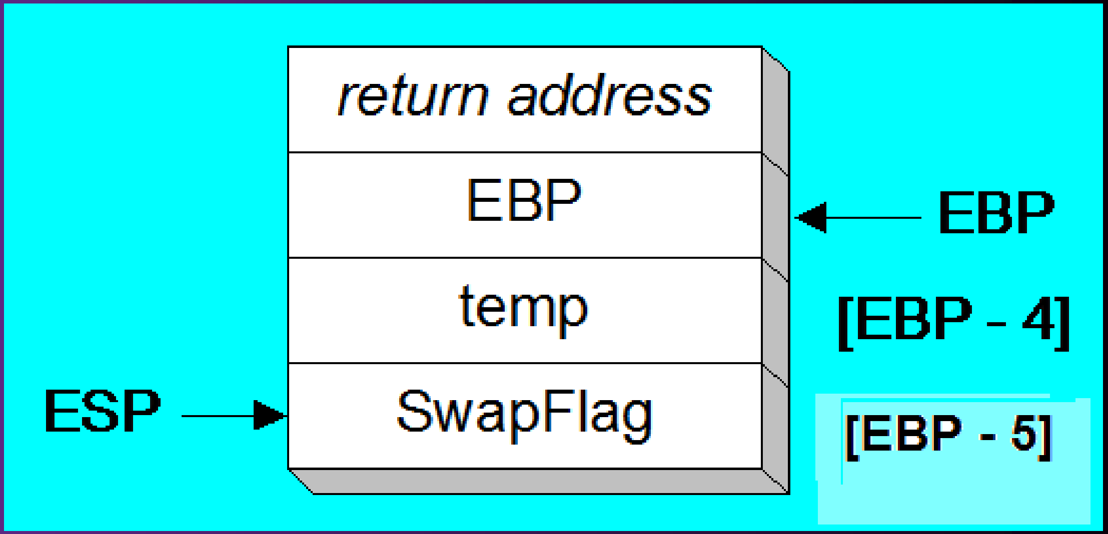
```
LOCAL flagVals[20]:BYTE     ; array of bytes

LOCAL pArray:PTR WORD       ; pointer to an array

myProc PROC                 ; procedure
   LOCAL t1:BYTE,           ; local variables
```

LOCAL Example

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



MASM generates:

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

See LocalExample.asm

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]
```

What's Next

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

What's Next

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

INVOKE, ADDR, PROC, and PROTO

- INVOKE Directive
- ADDR Operator
- PROC Directive
- PROTO Directive
- Parameter Classifications
- Example: Exchanging 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
  - Syntax:

  label PROC [attributes] [USES regList], paramList

- The USES clause must be on the same line as PROC.
- Attributes: distance, language type, visibility
- ParamList is a list of parameters separated by commas.

  label PROC, parameter1, parameter2, …, parameterN

  - 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

- 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
```

PROTO Directive

- Creates a procedure prototype
- Syntax:
  - *label* PROTO *paramList*
- 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
```

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.

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

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

CHAPTER 9:
STRINGS AND ARRAYS

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 0FFFFFFFFh
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('z')
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
```

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
```

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
```

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 "ABCDEFGH",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
```

What's Next

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

Selected String Procedures

The following string procedures may be found in the Irvine32 and Irvine16 libraries:

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

What's Next

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

Two-Dimensional Arrays

- Base-Index Operands
- Base-Index Displacement

Base-Index Operand

- A base-index (基址变址) 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.
- 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 definds 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
    mov  ecx,lengthof setOfCoordinates
L1: mov  ax,[ebx+esi]
    call WriteDec
    add  ebx,SIZEOF COORD
    loop L1
```

Base-Index-Displacement Operand

- A base-index-displacement (相对基址变址) operand adds base and index registers to a constant, producing an effective address. Any two 32-bit general-purpose registers may be used.
- Common formats:

[*base* + *index* + *displacement*]

displacement [*base* + *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:

[*base* + *index* + *displacement*]

displacement [*base* + *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:

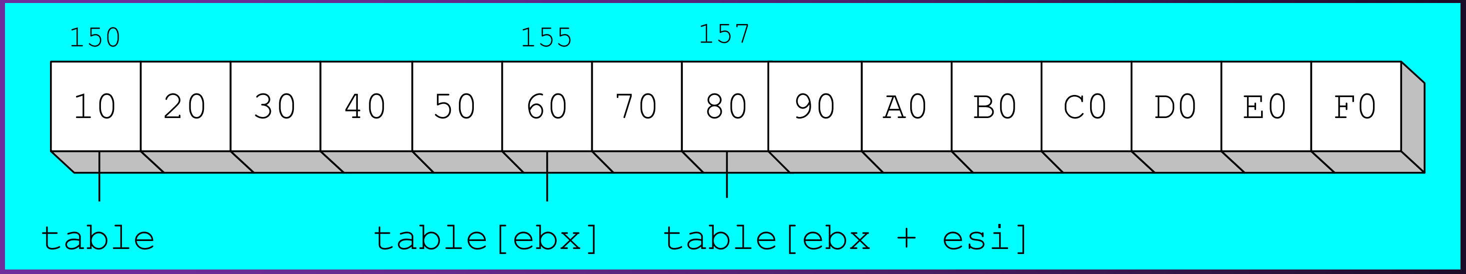
```
table  BYTE  10h,20h,30h,40h,50h,60h,70h,
       80h,90h,0A0h,
       0B0h,0C0h,0D0h,
       0E0h,0F0h
NumCols = 5
```

Two-Dimensional Table Example

The following 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]
```

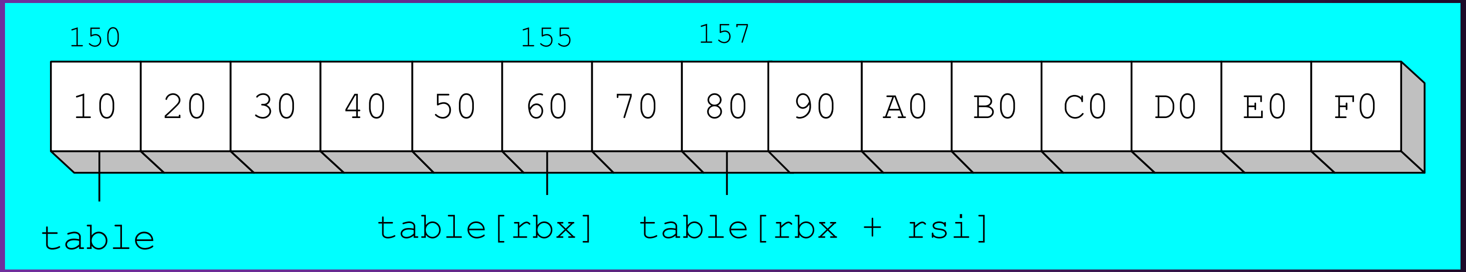


Two-Dimensional Table Example (64-bit)

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**
- Java Bytecodes: String Processing (optional topic)

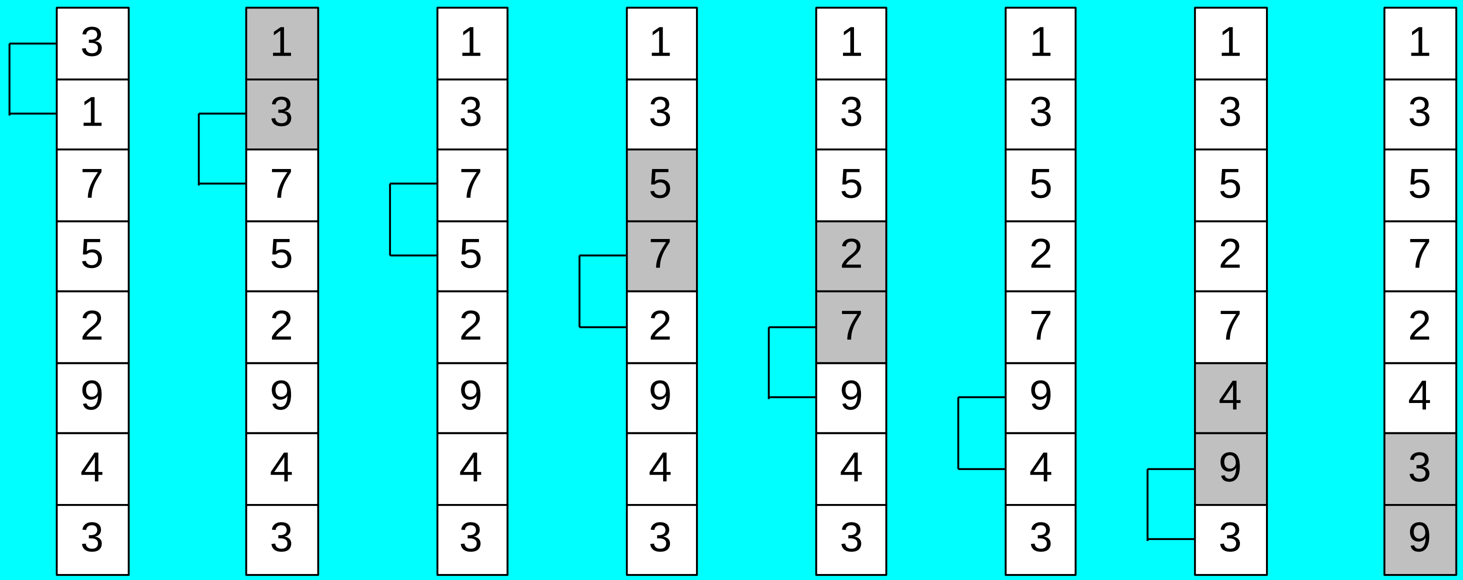
Searching and Sorting Integer Arrays

- 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:

One Pass (Bubble Sort)



(shaded values have been exchanged)

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
```

Summary (Chap 7)

- MUL and DIV – integer operations
  - close relatives of SHL and SHR
  - CBW, CDQ, CWD: preparation for division

Summary (Chap 8)

- 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

Summary (Chap 9)

- 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

Homework

- Reading Chap 7 -- 9
- Exercises

Thanks!