Intel x86 Assembly Fundamentals

Computer Organization and Assembly Languages

Yung-Yu Chuang

with slides by Kip Irvine

x86 Assembly Language Fundamentals

Instructions

- Assembled into machine code by assembler
- Executed at runtime by the CPU
- Member of the Intel IA-32 instruction set
- Four parts
  - Label (optional)
  - Mnemonic (required)
  - Operand (usually required)
  - Comment (optional)

Label:	Mnemonic	Operand(s)	;Comment

Labels

- Act as place markers
  - marks the address (offset) of code and data
- Easier to memorize and more flexible
  `mov ax, [0020]` → `mov ax, val`
- Follow identifier rules
- Data label
  - must be unique
  - example: **`myArray BYTE 10`**
- Code label (ends with a colon)
  - target of jump and loop instructions
  - example:

```
L1: mov ax, bx
    ...
    jmp L1
```

Reserved words and identifiers

- Reserved words cannot be used as identifiers
  - Instruction mnemonics, directives, type attributes, operators, predefined symbols
- Identifiers
  - 1-247 characters, including digits
  - case insensitive (by default)
  - first character must be a letter, **_, @,** or **$**
  - examples:

```
var1      Count     $first
_main     MAX       open_file
@@myfile  xVal      _12345
```


Mnemonics and operands

- Instruction mnemonics
  - "reminder"
  - examples: **MOV, ADD, SUB, MUL, INC, DEC**
- Operands
  - constant (immediate value), **96**
  - constant expression, **2+4**
  - Register, **eax**
  - memory (data label), **count**
- Number of operands: 0 to 3

```
- stc              ; set Carry flag
- inc ax           ; add 1 to ax
- mov count, bx    ; move BX to count
```


Directives

- Commands that are recognized and acted upon by the assembler
  - Part of assembler's syntax but not part of the Intel instruction set
  - Used to declare code, data areas, select memory model, declare procedures, etc.
  - case insensitive
- Different assemblers have different directives
  - NASM != MASM, for example
- Examples: **.data .code PROC**

Comments

- Comments are good!
  - explain the program's purpose
  - tricky coding techniques
  - application-specific explanations
- Single-line comments
  - begin with semicolon (;)
- block comments
  - begin with COMMENT directive and a programmer-chosen character and end with the same programmer-chosen character

```
COMMENT !
  This is a comment
  and this line is also a comment
!
```

Example: adding/subtracting integers

directive marking a comment

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

INCLUDE Irvine32.inc
.code
main PROC
    mov eax,10000h      ; EAX = 10000h
    add eax,40000h      ; EAX = 50000h
    sub eax,20000h      ; EAX = 30000h
    call DumpRegs       ; display registers
    exit
main ENDP
END main
```

copy definitions from Irvine32.inc

code segment. 3 segments: code, data, stack

beginning of a procedure

source

destination

defined in Irvine32.inc to end a program

marks the last line and define the startup procedure

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
```


Alternative version of AddSub

```
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
```


Program template

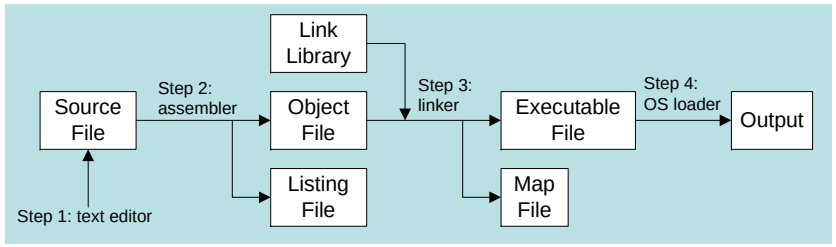
```
TITLE Program Template              (Template.asm)

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

.data
   ; (insert variables here)
.code
main PROC
   ; (insert executable instructions here)
   exit
main ENDP
   ; (insert additional procedures here)
END main
```

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.



Defining 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.
- Only size matters, other attributes such as signed are just reminders for programmers.
- Syntax:

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

  At least one initializer is required, can be **?**
- All initializers become binary data in memory

Integer constants

- [{+|-}] *digits* [*radix*]
- Optional leading + or – sign
- binary, decimal, hexadecimal, or octal digits
- Common radix characters:
  - **h** – hexadecimal
  - **d** – decimal (default)
  - **b** – binary
  - **r** – encoded real
  - **o** – octal

Examples: **30d, 6Ah, 42, 42o, 1101b**
Hexadecimal beginning with letter: **0A5h**

Integer expressions

- Operators and precedence levels:

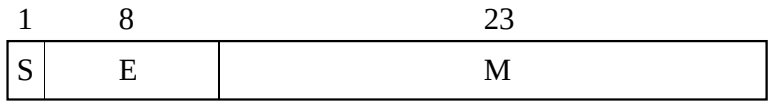
Operator	Name	Precedence Level
( )	parentheses	1
+,-	unary plus, minus	2
*,/	multiply, divide	3
MOD	modulus	3
+,-	add, subtract	4

- Examples:

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

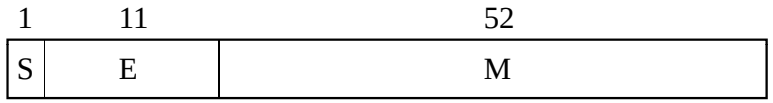
Real number constants (encoded reals)

- Fixed point v.s. floating point



$\pm 1.bbbb \times 2^{(E-127)}$

- Example **3F800000r=+1.0,37.75=42170000r**
- double



Real number constants (decimal reals)

- [*sign*]*integer*.[*integer*][*exponent*]
  sign → {+|-}
  exponent → E[{+|-}]integer
- Examples:

```
2.
+3.0
-44.2E+05
26.E5
```


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’**
  - **"This isn't a test"**

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
```

A variable name is a data label that implies an offset (an address).

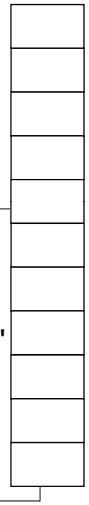
Defining multiple bytes

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 2)

- A string is implemented as an array of characters
  - For convenience, it is usually enclosed in quotation marks
  - It usually has a null byte at the end
- Examples:



```
str1 BYTE "Enter your name",0
str2 BYTE 'Error: halting program',0
str3 BYTE 'A','E','I','O','U'
greeting1 BYTE "Welcome to the Encryption Demo program "
          BYTE "created by Kip Irvine.",0
greeting2 \
   BYTE "Welcome to the Encryption Demo program "
   BYTE "created by Kip Irvine.",0
```


Defining strings (2 of 2)

- 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.
- Counter and argument must be constants or constant expressions

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


Defining WORD and SWORD data

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

```
word1 WORD   65535  ; largest unsigned
word2 SWORD –32768  ; smallest signed
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, (AddSub2.asm)
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 unintialized data segment:

```
.data?
```

- Within the segment, declare variables with "?" initializers: (will not be assembled into .exe)

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

```
.data
smallArray DWORD 10 DUP(0)
.data?
bigArray   DWORD 5000 DUP(?)
```


Mixing code and data

```
.code
mov eax, ebx
.data
temp DWORD ?
.code
mov temp, eax
```


Symbolic constants

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
  - Easier to modify
  - Easier to understand, **ESC_key**

```
Array DWORD COUNT DUP(0)
COUNT=5
mov al, COUNT
COUNT=10
mov al, COUNT
```

```
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 = 4
```

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

```
list BYTE 10,20,30,40
var2 BYTE 20 DUP(?)
ListSize = ($ - list)
```

```
myString BYTE “This is a long string.”
myString_len = ($ - myString)
```


Calculating the size of a word array

- current location counter: $
  - subtract address of list
  - difference is the number of bytes
  - divide by 2 (the size of a word)

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

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


EQU directive

- name EQU expression
  name EQU symbol
  name EQU <text>
- Define a symbol as either an integer or text expression.
- Can be useful for non-integer constants
- Cannot be redefined

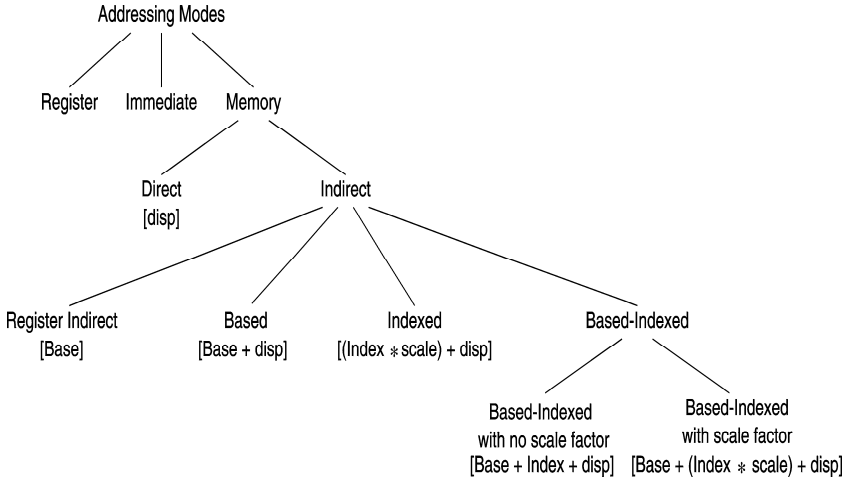
EQU directive

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

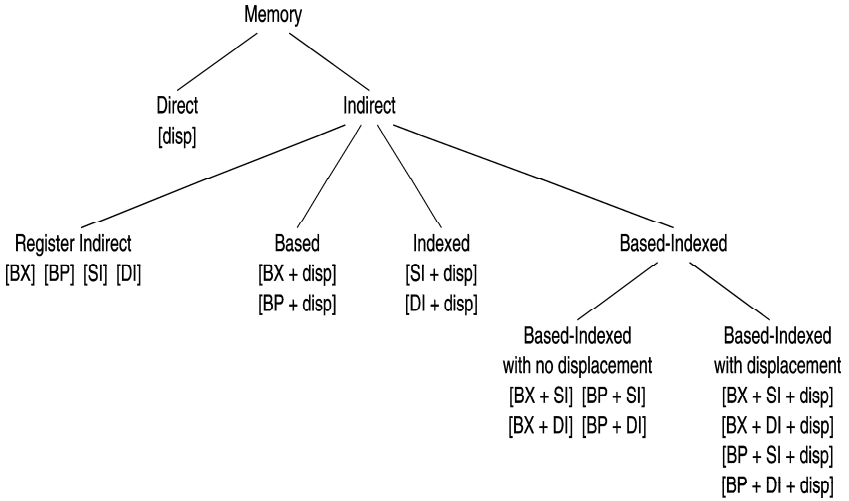
```
matrix1 EQU 10*10
matrix2 EQU <10*10>
.data
M1 WORD matrix1       ; M1 WORD 100
M2 WORD matrix2       ; M2 WORD 10*10
```

Addressing

Addressing Modes



Addressing Modes



32-Bit Addressing Modes

- These addressing modes use 32-bit registers

Segment + Base + (Index * Scale) + displacement

$$\begin{Bmatrix} CS: \\ DS: \\ SS: \\ ES: \\ FS: \\ GS: \end{Bmatrix} \left[\begin{Bmatrix} EAX \\ EBX \\ ECX \\ EDX \\ ESP \\ EBP \\ ESI \\ EDI \end{Bmatrix} + \left[\begin{Bmatrix} EAX \\ EBX \\ ECX \\ EDX \\ EBP \\ ESI \\ EDI \end{Bmatrix} * \begin{Bmatrix} 1 \\ 2 \\ 4 \\ 8 \end{Bmatrix} \right] + [displacement]$$

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 by the assembler

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

alternate format; I prefer this one.

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. (no range checking)

```
.data
arrayB BYTE 10h,20h,30h,40h
.code
mov al,arrayB+1      ; AL = 20h
mov al,[arrayB+1]    ; alternative notation
mov al,arrayB+3      ; AL = 40h
```

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
```

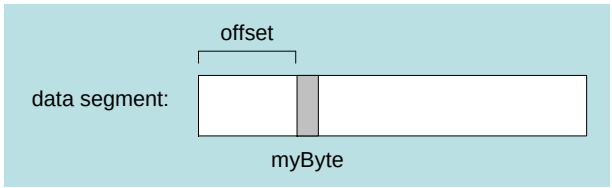
```
; will the following assemble and run?
mov ax,[arrayW-2]        ; ??
mov eax,[arrayD+16]      ; ??
```


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 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;
```

```
.data
array BYTE 1000 DUP(?)
.code
mov esi,OFFSET array   ; ESI is p
```


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
```

ALIGN Directive

- **ALIGN** *bound* aligns a variable on a byte, word, doubleword, or paragraph boundary for efficiency. (*bound* can be 1, 2, 4, or 16.)

```
bVal   BYTE ?    ; 00404000
ALIGN 2
wVal   WORD ?    ; 00404002
bVal2  BYTE ?    ; 00404004
ALIGN 4
dVal   DWORD ?   ; 00404008
dVal2  DWORD ?   ; 0040400C
```


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
```

To understand how this works, we need to know about little endian ordering of data in memory.

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+1]    ; AX = 5634h
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
```


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; it is just an alias.
- 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
```


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). **[reg]** uses reg as pointer to access memory

```
.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 when the size of a memory operand is ambiguous.

```
.data
myCount WORD 0
                          unable to determine the
.code                     size from the context
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       ; increment ESI by 2
  add ax,[esi]    ; AX = sum of the array
```

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.
```


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
```