Akdeniz University
Computer Engineering Department

CSE206 Computer Organization

Week12-14: Assembly Language, Data Transfers, Addressing, and Arithmetic

Assoc.Prof.Dr. Taner Danışman

tdanisman@akdeniz.edu.tr

Course program (Textbook: Stalling 10th Edt.)

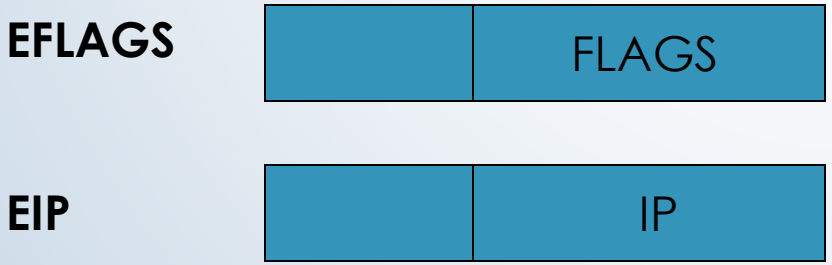
Week 1	10-Feb-25	Introduction	Ch1
Week 2	17-Feb-25	Computer Evolution	Ch2
Week 3	24-Feb-25	Computer Systems	Ch3
Week 4	3-Mar-25	Cache Memory, Direct Cache Mapping	Ch4
Week 5	10-Mar-25	Associative and Set Associative Mapping	Ch4
Week 6	17-Mar-25	Internal Memory, External Memory, I/O	Ch5-Ch6-Ch7
Week 7	24-Mar-25	Number Systems, Computer Arithmetic	Ch9-Ch10
Week 8	31-Mar-25	Midterm (Expected date, may change)	Ch1-...-Ch10
Week 9	7-Apr-25	Digital Logic	Ch11
Week 10	14-Apr-25	Instruction Sets	Ch12
Week 11	21-Apr-25	Addressing Modes	Ch13
Week 12	28-Apr-25	Processor Structure and Function	Ch14
Week 13	5-May-25	RISC, Inctruction Level Parallelism	Ch15-Ch16
Week 14	12-May-25	Assembly Language (TextBook : Assembly Language for x86 Processors)	Kip Irvine
Week 15	19-May-25	Assembly Language (TextBook : Assembly Language for x86 Processors)	Kip Irvine

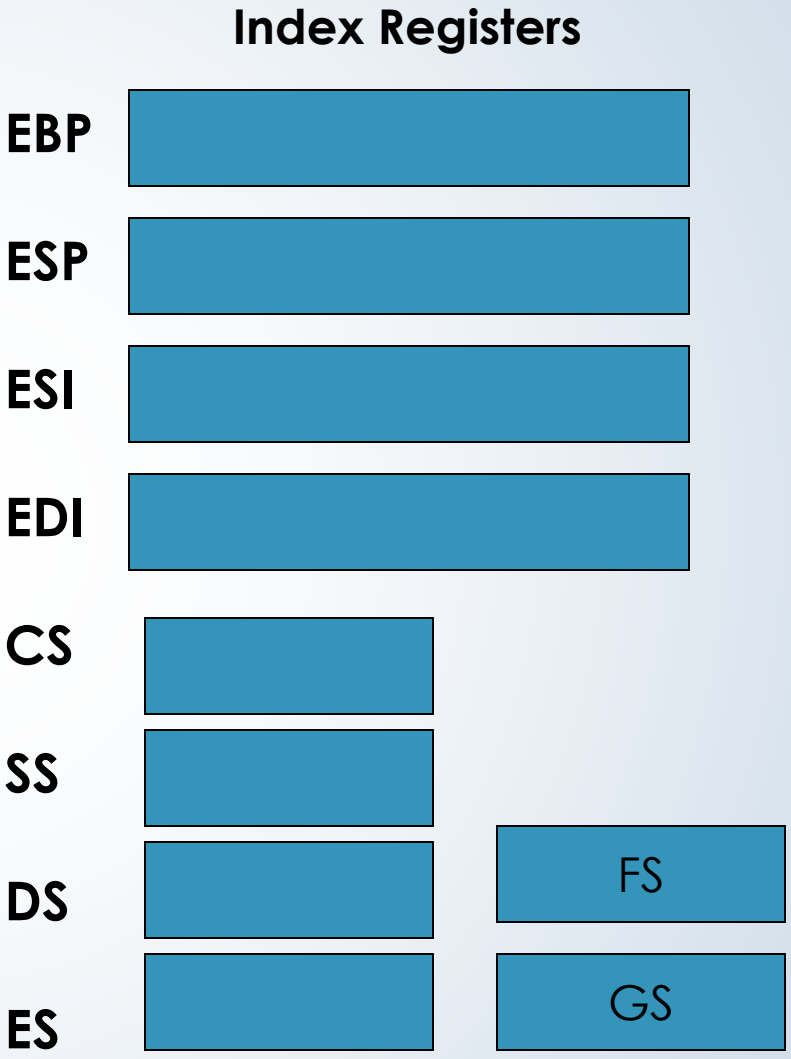
Registers

- Intel 16-bits registers

Registers

- Intel 32-bits registers





X64 Registers

64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	**eax**	**ax**	**al**
rbx	**ebx**	**bx**	**bl**
rcx	**ecx**	**cx**	**cl**
rdx	**edx**	**dx**	**dl**
rsi	**esi**	**si**	**sil**
rdi	**edi**	**di**	**dil**
rbp	**ebp**	**bp**	**bpl**
rsp	**esp**	**sp**	**spl**

64 extends x86's 8 general-purpose registers to be 64-bit, and adds 8 new 64-bit registers. The 64-bit registers have names beginning with "r", so for example the 64-bit extension of **eax** is called **rax**.

Some Specialized Register Uses (1 of 2)

- General-Purpose
  - EAX – accumulator
  - ECX – loop counter
  - ESP – stack pointer
  - ESI, EDI – index registers
  - EBP – extended frame pointer (stack)
- Segment
  - CS – code segment
  - DS – data segment
  - SS – stack segment
  - ES, FS, GS - additional segments

Chapter Overview (Chp #3 from Irvine)

- http://kipirvine.com/asm/
- http://kipirvine.com/asm/examples/index.htm
- http://kipirvine.com/asm/examples/Irvine_7th_Edition.msi

- Read Chapter #3
- Basic Elements of Assembly Language
- Example: Adding and Subtracting Integers
- Assembling, Linking, and Running Programs
- Defining Data
- Symbolic Constants

Integer Constants

- Optional leading + or – sign
- binary, decimal, hexadecimal, or octal digits
- Common radix characters:
  - h – hexadecimal
  - d – decimal
  - b – binary
  - r – encoded real

Examples: 30d, 6Ah, 42, 1101b

Hexadecimal beginning with letter: 0A5h

Integer Expressions

- Operators and precedence levels:

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

- Examples:

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

Character and String Constants

- Enclose **character** in single or double quotes
  - 'A', "x"
  - ASCII character = 1 byte
- Enclose **strings** in single or double quotes
  - "ABC"
  - 'xyz'
  - Each character occupies a single byte
- Embedded quotes:
  - 'Say "Goodnight," Gracie'

Reserved Words and Identifiers

- Reserved words cannot be used as identifiers
  - Instruction mnemonics, directives, type attributes, operators, predefined symbols
- Identifiers
  - 1-247 characters, including digits
  - not case sensitive
  - first character must be a letter, _, @, ?, or $

Directives

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

Instructions

- Assembled into machine code by assembler
- Executed at runtime by the CPU
- We use the Intel IA-32 instruction set
- An instruction contains:
  - Label (optional)
  - Mnemonic (required)
  - Operand (depends on the instruction)
  - Comment (optional)

Labels

- Act as place markers
  - marks the address (offset) of code and data
- Follow identifer rules
- Data label
  - must be unique
  - example: **myArray** (not followed by colon)
- Code label
  - target of jump and loop instructions
  - example: **L1:** (followed by colon)

Mnemonics and Operands

- Instruction Mnemonics
  - memory aid
  - examples: MOV, ADD, SUB, MUL, INC, DEC
- Operands
  - constant
  - constant expression
  - register
  - memory (data label)

Constants and constant expressions are often called immediate values

Comments

- Comments are good!
  - explain the program's purpose
  - when it was written, and by whom
  - revision information
  - tricky coding techniques
  - application-specific explanations
- Single-line comments
  - begin with semicolon (;)
- Multi-line comments
  - begin with COMMENT directive and a programmer-chosen character
  - end with the same programmer-chosen character

Instruction Format Examples

- No operands
  - stc ; set Carry flag
- One operand
  - inc eax ; register
  - inc myByte ; memory
- Two operands
  - add ebx,ecx ; register, register
  - sub myByte,25 ; memory, constant
  - add eax,36 * 25 ; register, constant-expression

The Hello World Program -16 bit

```
.MODEL small
   .STACK 100h
   .DATA
Message DB ‘Hello, world!’,13,10,’$’
   .CODE
   mov  ax,@data
   mov  ds,ax              ;set DS to point to the data segment
   mov  ah,9                 ;DOS print string function
   mov  dx,OFFSET Message  ;point to “Hello, world!”
   int  21h                  ;display “Hello, world!”
   mov  ah,4ch               ;DOS terminate program function
   int  21h                  ;terminate the program
   END
```

Printing digits – 16 bit

```
data segment
data ends
code segment
    assume ds : data, cs : code
start :
    mov ax, data
    mov ds, ax

    mov bl, 00h
    mov ch, 00h
    mov cl, 0ah

l1   :
    mov dh, 00h
    mov dl, bl
    add dl, '0'
    mov ah, 02h
    int 21h
    inc bl
loop l1
```

```
    mov ah, 4ch
    int 21h
code ends
end start

; ------
; output
; ------

;    0123456789
```

General Program Template

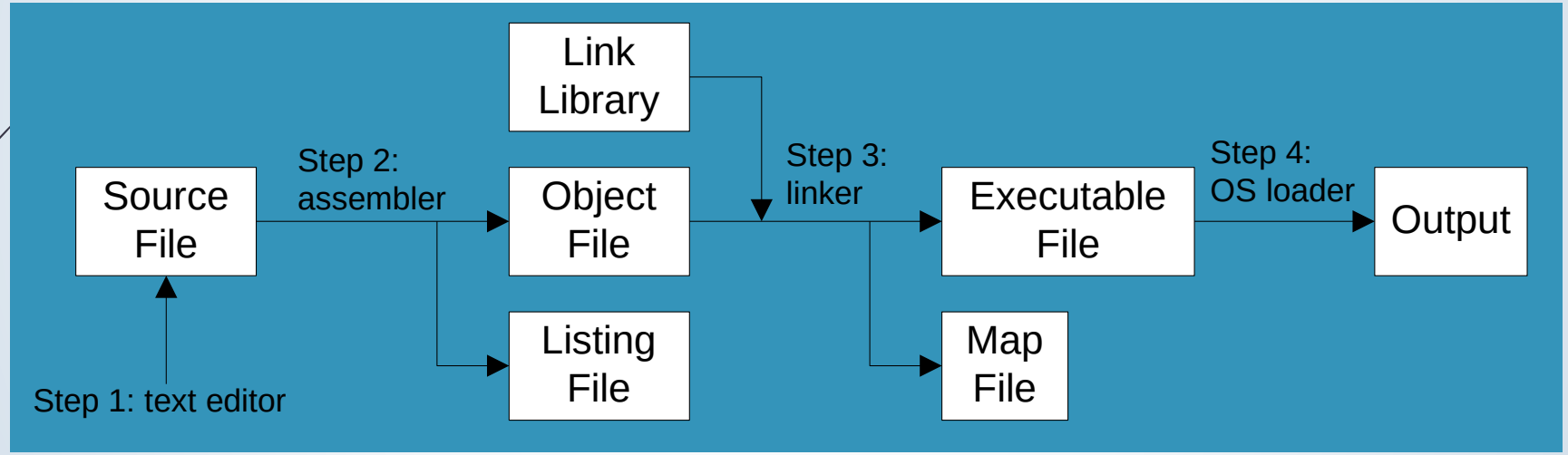
```
; Program Template                (Template.asm)

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

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

Assembling, Linking, and Running Programs

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

Assemble-Link Execute Cycle

- The following diagram describes the steps from creating a source program through executing the compiled program.
- If the source code is modified, Steps 2 through 4 must be repeated.



Listing File

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

Defining Data

- Intrinsic Data Types
- Data Definition Statement
- Defining BYTE and SBYTE Data
- Defining WORD and SWORD Data
- Defining DWORD and SDWORD Data
- Defining QWORD Data
- Defining TBYTE Data
- Defining Real Number Data
- Little Endian Order
- Adding Variables to the AddSub Program
- Declaring Uninitialized Data

Intrinsic Data Types (1 of 2)

- BYTE, SBYTE
  - 8-bit unsigned integer; 8-bit signed integer
- WORD, SWORD
  - 16-bit unsigned & signed integer
- DWORD, SDWORD
  - 32-bit unsigned & signed integer
- QWORD
  - 64-bit integer
- TBYTE
  - 80-bit integer

Intrinsic Data Types (2 of 2)

- REAL4
  - 4-byte IEEE short real
- REAL8
  - 8-byte IEEE long real
- REAL10
  - 10-byte IEEE extended real

Data Definition Statement

- A data definition statement sets aside storage in memory for a variable.
- May optionally assign a name (label) to the data
- Syntax:

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

```
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:

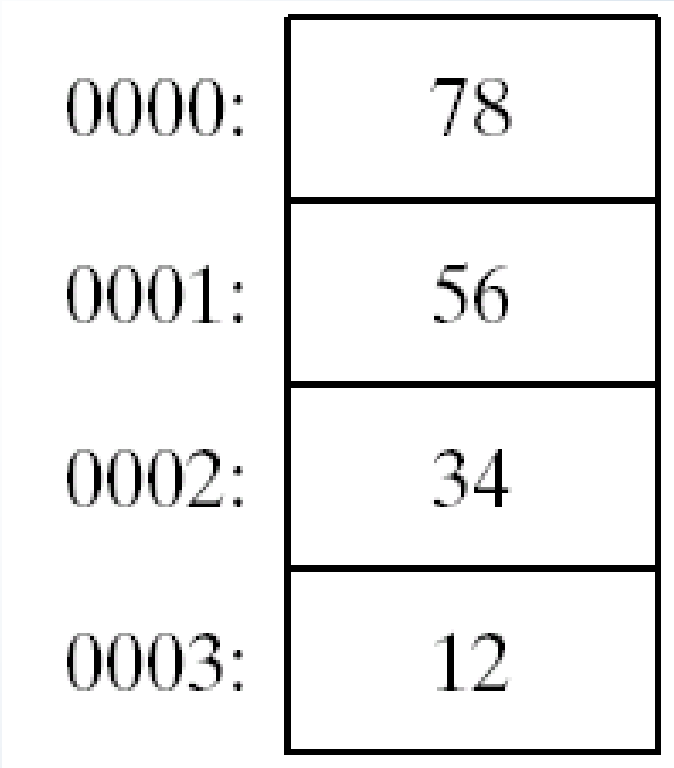
```
quad1 QWORD  1234567812345678h
val1  TBYTE  10000000000123456789Ah
rVal1 REAL4  -2.1
rVal2 REAL8  3.2E-260
rVal3 REAL10 4.6E+4096
ShortArray REAL4 20 DUP(0.0)
```

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
```



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 unintialized 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**

Symbolic Constants

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

Calculating the Size of a Byte Array

- current location counter: $
  - subtract address of list
  - difference is the number of bytes

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

Calculating the Size of a Word Array

Divide total number of bytes by 2 (the size of a word)

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

Calculating the Size of a Doubleword Array

Divide total number of bytes by 4 (the size of a doubleword)

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

EQU Directive

- Define a symbol as either an integer or text expression.
- Cannot be redefined

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

TEXTEQU Directive

- Define a symbol as either an integer or text expression.
- Called a **text macro**
- Can be redefined

```
continueMsg TEXTEQU <"Do you wish to continue (Y/N)?">
rowSize = 5
.data
prompt1 BYTE continueMsg
count TEXTEQU %(rowSize * 2)      ; evaluates the expression
setupAL TEXTEQU <mov al,count>

.code
setupAL                           ; generates: "mov al,10"
```

Addressing modes

- The three basic modes of addressing are
  - **Register addressing**
    - In this addressing mode, a register contains the operand. Depending upon the instruction, the register may be the first operand, the second operand or both.

```
MOV DX, TAX_RATE ; Register in first operand
MOV COUNT, CX ; Register in second operand
MOV EAX, EBX ; Both the operands are in registers
```

As processing data between registers does not involve memory, it provides fastest processing of data.

  - Immediate addressing
    - An immediate operand has a constant value or an expression. The first operand defines the length of the data.

```
BYTE_VALUE DB 150 ; A byte value is defined
WORD_VALUE DW 300 ; A word value is defined
ADD BYTE_VALUE, 65 ; An immediate operand 65 is added
MOV AX, 45H ; Immediate constant 45H is transferred to AX
```

When an instruction with two operands uses immediate addressing, the first operand may be a register or memory location, and the second operand is an immediate constant.

  - Memory addressing

Memory Addressing Modes

This way of addressing results in slower processing of data.

Direct Memory Addressing

- When operands are specified in memory addressing mode, direct access to main memory, usually to the data segment, is required.
- To locate the exact location of data in memory, we need the **segment start address**, which is typically found in the **DS register** and an offset value. This offset value is also called **effective address**.

```
ADD BYTE_VALUE, DL ; Adds the register in the memory location
MOV BX, WORD_VALUE ; Operand from the memory is added to register
```

Direct Offset Addressing

- This addressing mode uses the arithmetic operators to modify an address.

```
BYTE_TABLE DB 14, 15, 22, 45 ; Tables of bytes
WORD  TABLE DW 134, 345, 564, 123 ; Tables of words
MOV CL, BYTE_TABLE[2] ; Gets the 3rd element of the BYTE_TABLE
MOV CL, BYTE_TABLE + 2 ; Gets the 3rd element of the BYTE_TABLE
MOV CX, WORD_TABLE[3] ; Gets the 4th element of the WORD_TABLE
MOV CX, WORD_TABLE + 3 ; Gets the 4th element of the WORD_TABLE
```

Indirect Memory Addressing

Memory Addressing Modes

- **Indirect Addressing**
  - This addressing mode utilizes the computer's ability of *Segment:Offset* addressing.
  - Generally, the base registers **EBX, EBP (or BX, BP)** and the index registers (**DI, SI**), coded within **square brackets** for memory references, are used for this purpose.
  - **Indirect addressing** is generally used for variables containing several elements like, **arrays**.

```
MY_TABLE TIMES 10 DW 0 ; Allocates 10 words (2 bytes) each initialized to 0
MOV EBX, [MY_TABLE] ; Effective Address of MY_TABLE in EBX
MOV [EBX], 110 ; MY_TABLE[0] = 110
ADD EBX, 2 ; EBX = EBX +2
MOV [EBX], 123 ; MY_TABLE[1] = 123
```

Chapter Overview

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

Data Transfer Instructions

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

Operand Types

- Immediate – a constant integer (8, 16, or 32 bits)
  - **value is encoded within the instruction**
- Register – the name of a register
  - register name is **converted to a number** and encoded within the instruction
- Memory – reference to a location in memory
  - memory address is encoded within the instruction, or a register holds the address of a memory location

Instruction Operand Notation

Operand	Description
reg8	8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL
reg16	16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP
reg32	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
reg/mem8	8-bit operand, which can be an 8-bit general register or memory byte
reg/mem16	16-bit operand, which can be a 16-bit general register or memory word
reg/mem32	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
```

MOV Instruction

- **Move from source to destination. Syntax:**

  MOV destination,source

- No more than one memory operand permitted
- CS, EIP, and IP cannot be the destination
- No immediate to segment moves

```
.data
count BYTE 100
wVal  WORD 2
.code
   mov bl,count
   mov ax,wVal
   mov count,al

   mov al,wVal          ; error
   mov ax,count         ; error
   mov eax,count        ; error
```

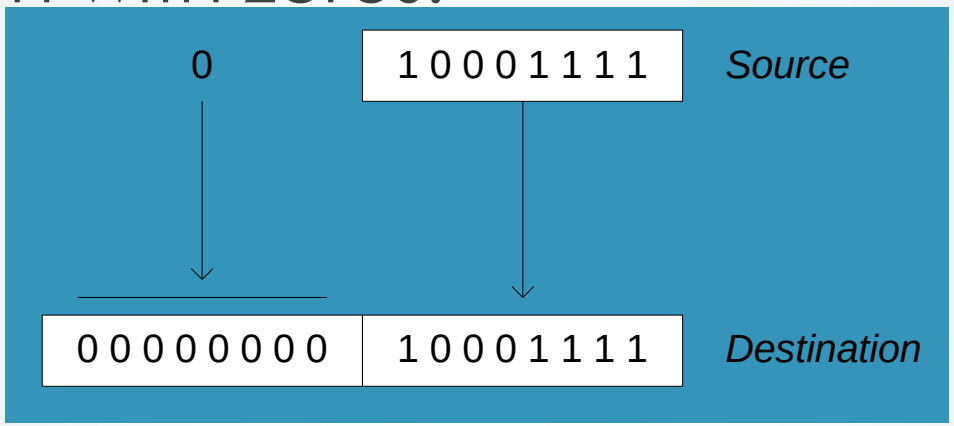
Your turn . . .

- Explain why each of the following MOV statements are invalid:

```
.data
bVal   BYTE   100
bVal2  BYTE   ?
wVal   WORD   2
dVal   DWORD  5
.code
    mov ds,45          immediate move to DS not permitted
    mov esi,wVal       size mismatch
    mov eip,dVal       EIP cannot be the destination
    mov 25,bVal        immediate value cannot be destination
    mov bVal2,bVal     memory-to-memory move not permitted
```

Zero Extension

- When you copy a smaller value into a larger destination, the MOVZX instruction fills (extends) the upper half of the destination with zeros.

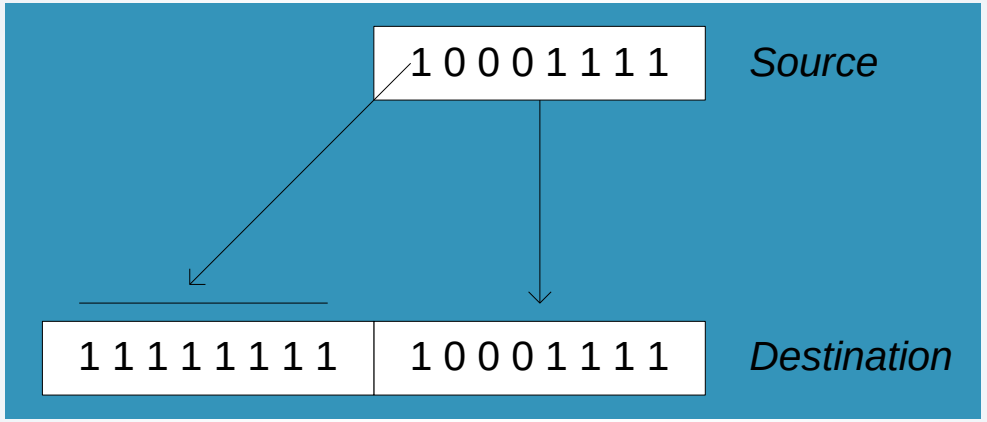



```
mov bl,10001111b
movzx ax,bl                    ; zero-extension
```

The destination must be a register.

Sign Extension

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




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

The destination must be a register.

XCHG Instruction

- XCHG exchanges the values of two operands. **At least one operand must be a register**. **No immediate operands are permitted.**

```
.data
var1 WORD 1000h
var2 WORD 2000h
.code
xchg ax,bx              ; exchange 16-bit regs
xchg ah,al              ; exchange 8-bit regs
xchg var1,bx            ; exchange mem, reg
xchg eax,ebx            ; exchange 32-bit regs

xchg var1,var2          ; error: two memory operands
```

Direct-Offset Operands

- A constant offset is **added** to a **data label** to produce an **effective address** (EA). The address is dereferenced to get the value inside its memory location.

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

Q: Why doesn't arrayB+1 produce 11h?

Direct-Offset Operands (cont)

- A constant offset is **added** to a **data label** to produce an **effective address** (EA). The address is dereferenced to get the value inside its memory location.

```
.data
arrayW  WORD 1000h,2000h,3000h
arrayD  DWORD 1,2,3,4
.code
mov ax,[arrayW+2]              ; AX = 2000h
mov ax,[arrayW+4]              ; AX = 3000h
mov eax,[arrayD+4]             ; EAX = 00000002h
```

Your turn. . .

Write a program that rearranges the values of three doubleword values in the following array as: 3, 1, 2.

```
.data
arrayD DWORD 1,2,3
```

- Step1: copy the first value into EAX and exchange it with the value in the second position.

```
mov eax,arrayD
xchg eax,[arrayD+4]
```

- Step 2: Exchange EAX with the third array value and copy the value in EAX to the first array position.

```
xchg eax,[arrayD+8]
mov  arrayD,eax
```

Evaluate this . . .

- We want to write a program that adds the following three bytes:

```
.data
myBytes BYTE 80h,66h,0A5h
```

- What is your evaluation of the following code?

```
mov al,myBytes
add al,[myBytes+1]
add al,[myBytes+2]
```

- What is your evaluation of the following code?

```
mov ax,myBytes
add ax,[myBytes+1]
add ax,[myBytes+2]
```

- Any other possibilities?

Evaluate this . . . (cont)

```
.data
myBytes BYTE 80h,66h,0A5h
```

- How about the following code. Is anything missing?

```
movzx ax,myBytes
mov   bl,[myBytes+1]
add   ax,bx
mov   bl,[myBytes+2]
add   ax,bx                    ; AX = sum
```

Yes: Move zero to BX before the MOVZX or mov bl,… instruction. ☺

What's Next

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

Addition and Subtraction

- INC and DEC Instructions
- ADD and SUB Instructions
- NEG Instruction
- Implementing Arithmetic Expressions
- Flags Affected by Arithmetic
  - Zero
  - Sign
  - Carry
  - Overflow

INC and DEC Instructions

- Add 1, subtract 1 from destination operand
  - **operand may be register or memory**
- INC *destination*
  - Logic: *destination* ← *destination* + 1
- DEC *destination*
  - Logic: *destination* ← *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
```

Your turn...

Show the value of the destination operand after each of the following instructions executes:

```
.data
myByte BYTE 0FFh, 0
.code
   mov al,myByte              ; AL = FFh
   mov ah,[myByte+1]          ; AH = 00h
   dec ah                     ; AH = FFh
   inc al                     ; AL = 00h
   dec ax                     ; AX = FEFF
```

ADD and SUB Instructions

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

Your turn...

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

```
Rval = Xval - (-Yval + Zval)
```

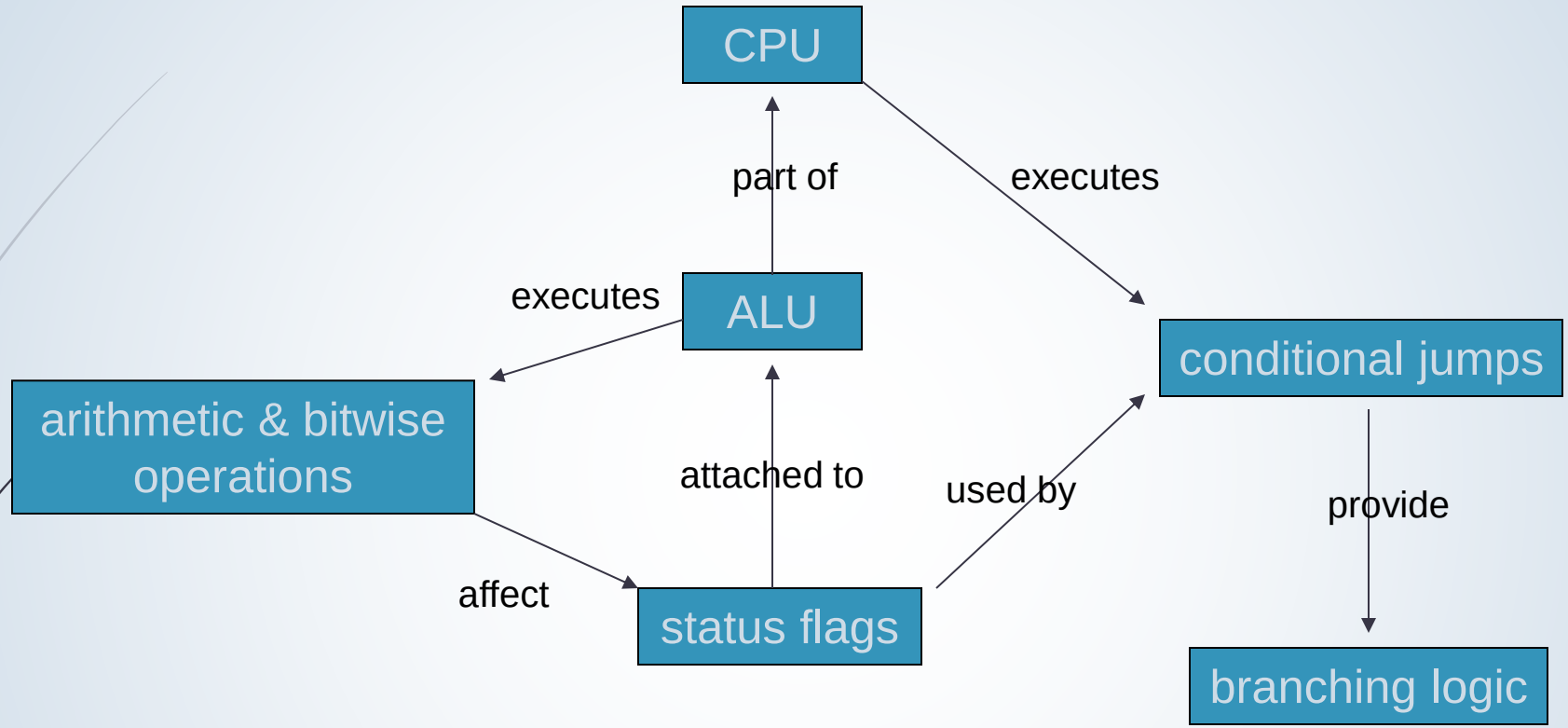
Assume that all values are signed doublewords.

```
mov ebx,Yval
neg ebx
add ebx,Zval
mov eax,Xval
sub eax,ebx
mov Rval,eax
```

Flags Affected by Arithmetic

- The ALU has a number of status flags that reflect the outcome of arithmetic (and bitwise) operations
  - based on the contents of the destination operand
- Essential flags:
  - **Zero flag** – set when destination equals zero
  - **Sign flag** – set when destination is negative
  - **Carry flag** – set when unsigned value is out of range
  - **Overflow flag** – set when signed value is out of range
- **The MOV instruction never affects the flags.**

Concept Map



You can use diagrams such as these to express the relationships between assembly language concepts.

Zero Flag (ZF)

The Zero flag is set when the result of an operation produces zero in the destination operand.

```
mov cx,1
sub cx,1                ; CX = 0, ZF = 1
mov ax,0FFFFh
inc ax                  ; AX = 0, ZF = 1
inc ax                  ; AX = 1, ZF = 0
```

Remember...

- A flag is set when it equals 1.
- A flag is clear when it equals 0.

Sign Flag (SF)

The Sign flag is set when the destination operand is negative.
The flag is clear when the destination is positive.

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

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

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

Signed and Unsigned Integers
A Hardware Viewpoint

- All CPU instructions operate exactly the same on signed and unsigned integers
- The CPU cannot distinguish between signed and unsigned integers
- **YOU**, the programmer, are solely responsible for using the correct data type with each instruction

Added Slide. Gerald Cahill, Antelope Valley College

Overflow and Carry Flags
A Hardware Viewpoint

- How the ADD instruction affects OF and CF:
  - CF = (carry out of the MSB)
  - OF = CF XOR MSB
- How the SUB instruction affects OF and CF:
  - CF = INVERT (carry out of the MSB)
  - negate the source and add it to the destination
  - OF = CF XOR MSB

MSB = Most Significant Bit (high-order bit)
XOR = eXclusive-OR operation
NEG = Negate (same as SUB 0,operand)

Carry Flag (CF)

The Carry flag is set when the result of an operation generates an unsigned value **that is out of range** (**too big or too small for the destination operand**).

```
mov al,0FFh
add al,1                    ; CF = 1, AL = 00

; Try to go below zero:

mov al,0
sub al,1                    ; CF = 1, AL = FF
```

Your turn . . .

For each of the following marked entries, show the values of the destination operand and the Sign, Zero, and Carry flags:

```
mov ax,00FFh
add ax,1              ; AX=0100h  SF=0 ZF=0 CF=0
sub ax,1              ; AX=00FFh  SF=0 ZF=0 CF=0
add al,1              ; AL=00h    SF=0 ZF=1 CF=1
mov bh,6Ch
add bh,95h            ; BH=01h    SF=0 ZF=0 CF=1

mov al,2
sub al,3              ; AL=FFh    SF=1 ZF=0 CF=1
```

Overflow Flag (OF)

The Overflow flag is set when the signed result of an operation is invalid or out of range.

```
; Example 1
mov al,+127
add al,1                    ; OF = 1,   AL = ??

; Example 2
mov al,7Fh                  ; OF = 1,   AL = 80h
add al,1
```

The two examples are identical at the binary level because 7Fh equals +127. To determine the value of the destination operand, it is often easier to calculate in hexadecimal.

A Rule of Thumb

- When adding two integers, remember that the Overflow flag is only set when . . .
  - Two positive operands are added and their sum is negative
  - Two negative operands are added and their sum is positive

```
What will be the values of the Overflow flag?
    mov al,80h
    add al,92h              ; OF = 1

    mov al,-2
    add al,+127             ; OF = 0
```

What's Next

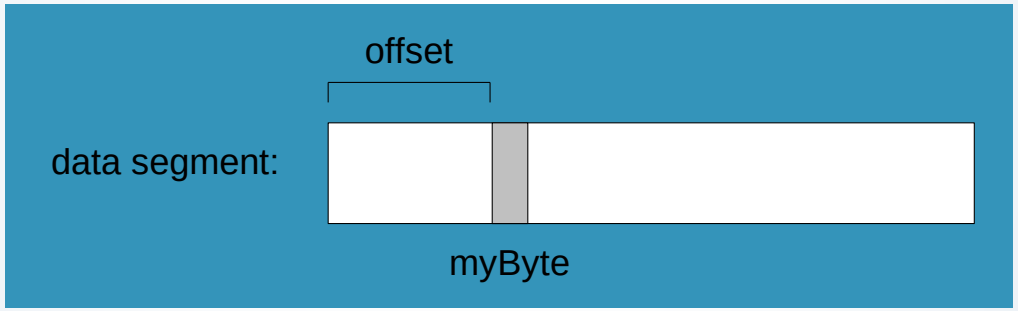
- Data Transfer Instructions
- Addition and Subtraction
- **Data-Related Operators and Directives**
- Indirect Addressing
- JMP and LOOP Instructions
- 64-Bit Programming

Data-Related Operators and Directives

- OFFSET Operator
- PTR Operator
- TYPE Operator
- LENGTHOF Operator
- SIZEOF Operator
- LABEL Directive

OFFSET Operator

- **OFFSET returns the distance in bytes**, of a label from the beginning of its enclosing segment
  - Protected mode: 32 bits
  - Real mode: 16 bits



The Protected-mode programs we write use only a single segment (flat memory model).

OFFSET Examples

Let's assume that the data segment begins at 00404000h:

```
.data
bVal BYTE ?
wVal WORD ?
dVal DWORD ?
dVal2 DWORD ?

.code
mov esi,OFFSET bVal ; ESI = 00404000
mov esi,OFFSET wVal ; ESI = 00404001
mov esi,OFFSET dVal ; ESI = 00404003
mov esi,OFFSET dVal2; ESI = 00404007
```

Relating to C/C++

The value returned by OFFSET is a pointer. Compare the following code written for both C++ and assembly language:

```
// C++ version:

char array[1000];
char * p = array;
```

```
; Assembly language:

.data
array BYTE 1000 DUP(?)
.code
mov esi,OFFSET array
```

PTR Operator

Overrides the default type of a label (variable). Provides the flexibility to access **part of a variable.**

```
.data
myDouble DWORD 12345678h
.code
mov ax,myDouble                    ; error – why?

mov ax,WORD PTR myDouble           ; loads 5678h

mov WORD PTR myDouble,4321h        ; saves 4321h
```

Little endian order is used when storing data in memory (see Section 3.4.9).

Little Endian Order

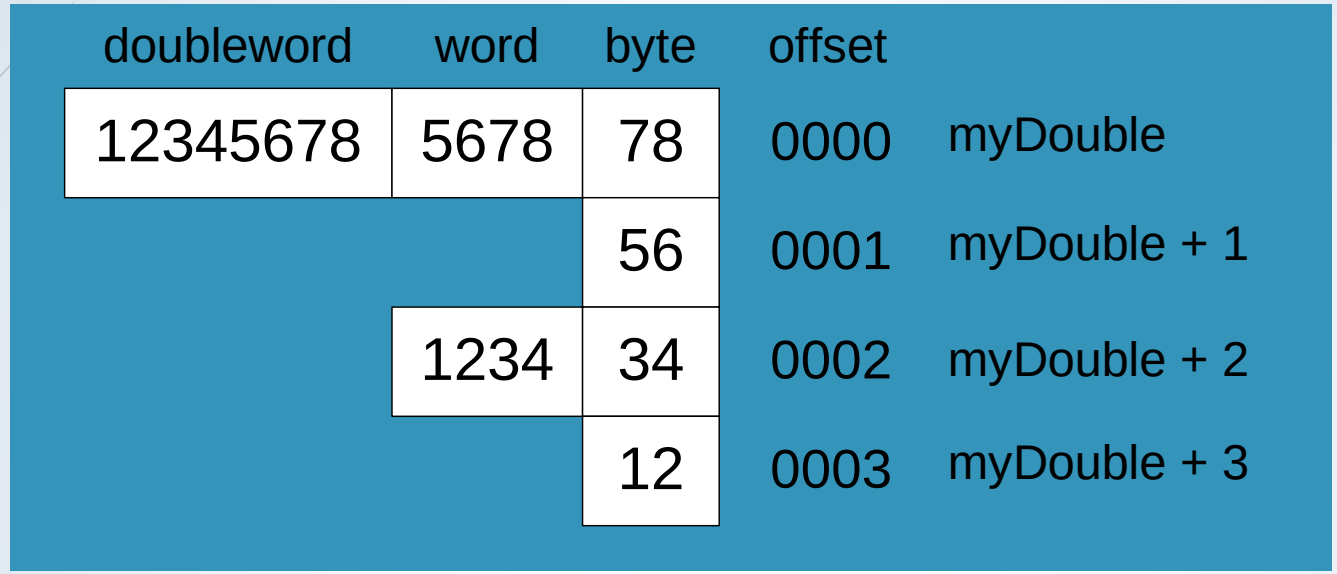
- Little endian order refers to the way Intel stores integers in memory.
- Multi-byte integers are stored in reverse order, with the least significant byte stored at the lowest address
- For example, the doubleword 12345678h would be stored as:

byte	offset
78	0000
56	0001
34	0002
12	0003

When integers are loaded from memory into registers, the bytes are automatically re-reversed into their correct positions.

PTR Operator Examples

```
.data
myDouble DWORD 12345678h
```




```
mov al,BYTE PTR  myDouble          ; AL = 78h
mov al,BYTE PTR [myDouble+1]       ; AL = 56h
mov al,BYTE PTR [myDouble+2]       ; AL = 34h
mov ax,WORD PTR  myDouble          ; AX = 5678h
mov ax,WORD PTR [myDouble+2]       ; AX = 1234h
```

PTR Operator (cont)

PTR can also be used to combine elements of a smaller data type and move them into a larger operand. The CPU will automatically reverse the bytes.

```
.data
myBytes BYTE 12h,34h,56h,78h

.code
mov ax,WORD PTR [myBytes]          ; AX = 3412h
mov ax,WORD PTR [myBytes+2]        ; AX = 7856h
mov eax,DWORD PTR myBytes          ; EAX = 78563412h
```

Your turn . . .

Write down the value of each destination operand:

```
.data
varB BYTE 65h,31h,02h,05h
varW WORD 6543h,1202h
varD DWORD 12345678h

.code
mov ax,WORD PTR [varB+2]          ; a. 0502h
mov bl,BYTE PTR varD              ; b. 78h
mov bl,BYTE PTR [varW+2]          ; c. 02h
mov ax,WORD PTR [varD+2]          ; d. 1234h
mov eax,DWORD PTR varW            ; e. 12026543h
```

TYPE Operator

The TYPE operator returns the **size, in bytes, of a single element of a data declaration.**

```
.data
var1 BYTE ?
var2 WORD ?
var3 DWORD ?
var4 QWORD ?

.code
mov eax,TYPE var1    ; 1
mov eax,TYPE var2    ; 2
mov eax,TYPE var3    ; 4
mov eax,TYPE var4    ; 8
```

LENGTHOF Operator

The LENGTHOF operator counts the number of elements in a single data declaration.

```
.data                                   LENGTHOF
byte1  BYTE 10,20,30                    ; 3
array1 WORD 30 DUP(?),0,0               ; 32
array2 WORD 5 DUP(3 DUP(?))             ; 15
array3 DWORD 1,2,3,4                    ; 4
digitStr BYTE "12345678",0              ; 9

.code
mov ecx,LENGTHOF array1                 ; 32
```

SIZEOF Operator

The SIZEOF operator returns a value that is equivalent to multiplying LENGTHOF by TYPE.

```
.data                                   SIZEOF
byte1  BYTE 10,20,30                    ; 3
array1 WORD 30 DUP(?),0,0               ; 64
array2 WORD 5 DUP(3 DUP(?))             ; 30
array3 DWORD 1,2,3,4                    ; 16
digitStr BYTE "12345678",0              ; 9

.code
mov ecx,SIZEOF array1                   ; 64
```

Spanning Multiple Lines (1 of 2)

A data declaration spans multiple lines if each line (except the last) ends with a comma. The LENGTHOF and SIZEOF operators include all lines belonging to the declaration:

```
.data
array WORD 10,20,
    30,40,
    50,60

.code
mov eax,LENGTHOF array              ; 6
mov ebx,SIZEOF array                ; 12
```

LABEL Directive

- Assigns an alternate label name and type to an existing storage location
- **LABEL does not allocate any storage of its own**
- Removes the need for the PTR operator

```
.data
dwList   LABEL DWORD
wordList LABEL WORD
intList  BYTE 00h,10h,00h,20h
.code
mov eax,dwList              ; 20001000h
mov cx,wordList             ; 1000h
mov dl,intList              ; 00h
```

What's Next

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

Indirect Addressing

- Indirect Operands
- Array Sum Example
- Indexed Operands
- Pointers

Indirect Operands (1 of 2)

An indirect operand holds the address of a variable, usually an array or string. It can be dereferenced (just like a pointer).

```
.data
val1 BYTE 10h,20h,30h
.code
mov esi,OFFSET val1
mov al,[esi]              ; dereference ESI (AL = 10h)

inc esi
mov al,[esi]              ; AL = 20h

inc esi
mov al,[esi]              ; AL = 30h
```

Indirect Operands (2 of 2)

Use PTR to clarify the size attribute of a memory operand.

```
.data
myCount WORD 0

.code
mov esi,OFFSET myCount
inc [esi]                ; error: ambiguous
inc WORD PTR [esi]       ; ok
```

Array Sum Example

Indirect operands are ideal for traversing an array.
Note that the register in brackets must be incremented by a value that matches the array type.

```
.data
arrayW WORD 1000h,2000h,3000h
.code
    mov esi,OFFSET arrayW
    mov ax,[esi]
    add esi,2                   ; or: add esi,TYPE arrayW
    add ax,[esi]
    add esi,2
    add ax,[esi]                ; AX = sum of the array
```

ToDo: Modify this example for an array of doublewords.

Indexed Operands

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

`[label + reg]` `label[reg]`

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

Index Scaling

You can scale an indirect or indexed operand to the offset of an array element. This is done by multiplying the index by the array's TYPE:

```
.data
arrayB BYTE  0,1,2,3,4,5
arrayW WORD  0,1,2,3,4,5
arrayD DWORD 0,1,2,3,4,5

.code
mov esi,4
mov al,arrayB[esi*TYPE arrayB]    ; 04
mov bx,arrayW[esi*TYPE arrayW]    ; 0004
mov edx,arrayD[esi*TYPE arrayD]   ; 00000004
```

Pointers

You can declare a pointer variable that contains the offset of another variable.

```
.data
arrayW WORD 1000h,2000h,3000h
ptrW DWORD arrayW
.code
   mov esi,ptrW
   mov ax,[esi]              ; AX = 1000h
```

Alternate format:

```
ptrW DWORD OFFSET arrayW
```

What's Next

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

JMP and LOOP Instructions

- JMP Instruction
- LOOP Instruction
- LOOP Example
- Summing an Integer Array
- Copying a String

JMP Instruction

- JMP is an unconditional jump to a label that is usually within the same procedure.
- Syntax: JMP *target*
- Logic: EIP ← *target*
- Example:

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

A jump outside the current procedure must be to a special type of label called a global label (see Section 5.5.2.3 for details).

LOOP Instruction

- The LOOP instruction creates a counting loop
- Syntax: LOOP *target*
- Logic:
  - ECX ← 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 0000            mov   ax,0
00000004    B9 00000005           mov   ecx,5

00000009    66 03 C1          L1: add   ax,cx
0000000C    E2 FB                 loop  L1
0000000E
```

When LOOP is assembled, the current location = 0000000E (offset of the next instruction). –5 (FBh) is added to the the current location, causing a jump to location 00000009:

00000009 ← 0000000E + FB

Your turn . . .

If the relative offset is encoded in a single signed byte,

(a) what is the largest possible backward jump?

(b) what is the largest possible forward jump?

(a) –128

(b) +127

Your turn . . .

What will be the final value of AX?

10

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

How many times will the loop execute?

4,294,967,296

```
    mov ecx,0
X2:
    inc ax
    loop X2
```

Nested Loop

If you need to code a loop within a loop, you must save the outer loop counter's ECX value. In the following example, the outer loop executes 100 times, and the inner loop 20 times.

```
.data
count DWORD ?
.code
    mov ecx,100          ; set outer loop count
L1:
    mov count,ecx        ; save outer loop count
    mov ecx,20           ; set inner loop count
L2: .
    .
    loop L2              ; repeat the inner loop
    mov ecx,count        ; restore outer loop count
    loop L1              ; repeat the outer loop
```

Summing an Integer Array

The following code calculates the sum of an array of 16-bit integers.

```
.data
intarray WORD 100h,200h,300h,400h
.code
    mov edi,OFFSET intarray        ; address of intarray
    mov ecx,LENGTHOF intarray      ; loop counter
    mov ax,0                       ; zero the accumulator
L1:
    add ax,[edi]                   ; add an integer
    add edi,TYPE intarray          ; point to next integer
    loop L1                        ; repeat until ECX = 0
```

Copying a String

The following code copies a string from source to target:

```
.data
source  BYTE  "This is the source string",0
target  BYTE  SIZEOF source DUP(0)

.code
    mov  esi,0                    ; index register
    mov  ecx,SIZEOF source        ; loop counter
L1:
    mov  al,source[esi]           ; get char from source
    mov  target[esi],al           ; store it in the target
    inc  esi                      ; move to next character
    loop L1                       ; repeat for entire string
```

good use of SIZEOF

Summary

- Data Transfer
  - MOV – data transfer from source to destination
  - MOVSX, MOVZX, XCHG
- Operand types
  - direct, direct-offset, indirect, indexed
- Arithmetic
  - INC, DEC, ADD, SUB, NEG
  - Sign, Carry, Zero, Overflow flags
- Operators
  - OFFSET, PTR, TYPE, LENGTHOF, SIZEOF, TYPEDEF
- JMP and LOOP – branching instructions