Assembly Language Fundamentals

Integer Constants

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

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

Hexadecimal beginning with letter: 0A5h

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," Mohammad'

Labels

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

Data Allocation

Data Allocation

- Variable declaration in a high-level language such as C

```
char    response
int     value
float   total
double  average_value
```

specifies

» Amount storage required (1 byte, 2 bytes, …)

» Label to identify the storage allocated (**`response`**, **`value`**, …)

» Interpretation of the bits stored (signed, floating point, …)

– Bit pattern **`1000 1101 1011 1001`** is interpreted as

✈ –29,255 as a signed number

✈ 36,281 as an unsigned number

Data Allocation (cont'd)

- In assembly language, we use the *define* directive
  - ∗ Define directive can be used
    - » To reserve storage space
    - » To label the storage space
    - » To initialize
    - » But *no interpretation* is attached to the bits stored
      - – Interpretation is up to the program code
  - ∗ Define directive goes into the .DATA part of the assembly language program
- Define directive format

```
[var-name]  D?  init-value [,init-value],...
```

Data Allocation (cont'd)

- Five define directives

```
DB   Define Byte          ;allocates 1 byte
DW   Define Word          ;allocates 2 bytes
DD   Define Doubleword    ;allocates 4 bytes
DQ   Define Quadword      ;allocates 8 bytes
DT   Define Ten bytes     ;allocates 10 bytes
```

Examples

```
sorted     DB   'y'
response   DB   ?       ;no initialization
value      DW   25159
```

Data Allocation (cont'd)

- Multiple definitions can be abbreviated

Example

```
message  DB   'B'
         DB   'y'
         DB   'e'
         DB   0DH
         DB   0AH
```

can be written as

```
message  DB   'B','y','e',0DH,0AH
```

- More compactly as

```
message  DB   'Bye',0DH,0AH
```

Data Allocation (cont'd)

- Multiple definitions can be cumbersome to initialize data structures such as arrays

Example

To declare and initialize an integer array of 8 elements

```
marks   DW    0,0,0,0,0,0,0,0
```

- What if we want to declare and initialize to zero an array of 200 elements?
  - ∗ There is a better way of doing this than repeating zero 200 times in the above statement
    - » Assembler provides a directive to do this (DUP directive)

Data Allocation (cont'd)

- Multiple initializations
  - ∗ The DUP assembler directive allows multiple initializations to the same value
  - ∗ Previous marks array can be compactly declared as

```
marks   DW    8 DUP (0)
```

Examples

```
table1  DW  10 DUP (?)        ;10 words, uninitialized
message DB  3  DUP ('Bye!')   ;12 bytes, initialized
                              ;      as Bye!Bye!Bye!
Name1   DB  30 DUP ('?')      ;30 bytes, each
                              ;    initialized to ?
```

Data Allocation (cont'd)

- The DUP directive may also be nested

Example

```
stars DB 4 DUP(3 DUP ('*'),2 DUP ('?'),5 DUP ('!'))
```

Reserves 40-bytes space and initializes it as

```
***??!!!!!***??!!!!!***??!!!!!***??!!!!!
```

Example

```
matrix  DW  10 DUP (5 DUP (0))
```

defines a 10X5 matrix and initializes its elements to 0

This declaration can also be done by

```
matrix  DW  50 DUP (0)
```

Data Allocation (cont'd)

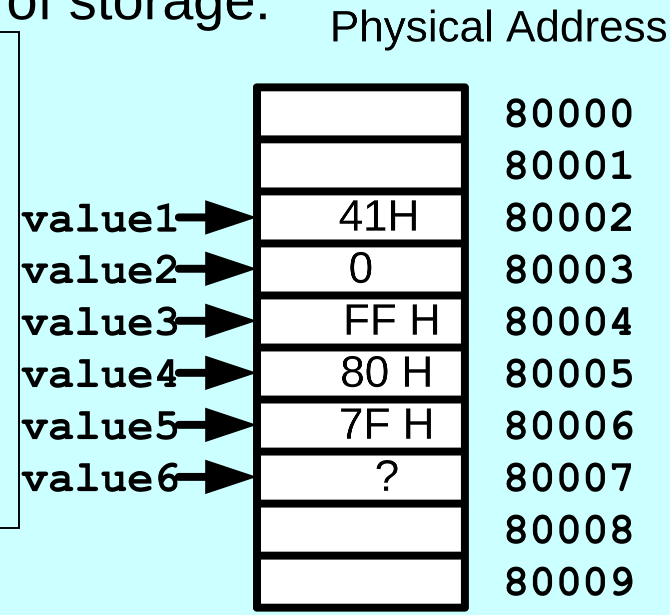
Correspondence to C Data Types

Directive	C data type
`DB`	**`char`**
`DW`	**`int, unsigned`**
`DD`	**`float, long`**
`DQ`	**`double`**
`DT`	internal intermediate **`float`** value

Defining BYTE

Each of the following defines a single byte of storage:

```
value1 DB 'A'; character constant
value2 DB 0; smallest unsigned byte
value3 DB 255; largest unsigned byte
value4 DB -128; smallest signed byte
value5 DB +127; largest signed byte
value6 DB ?; uninitialized byte
```

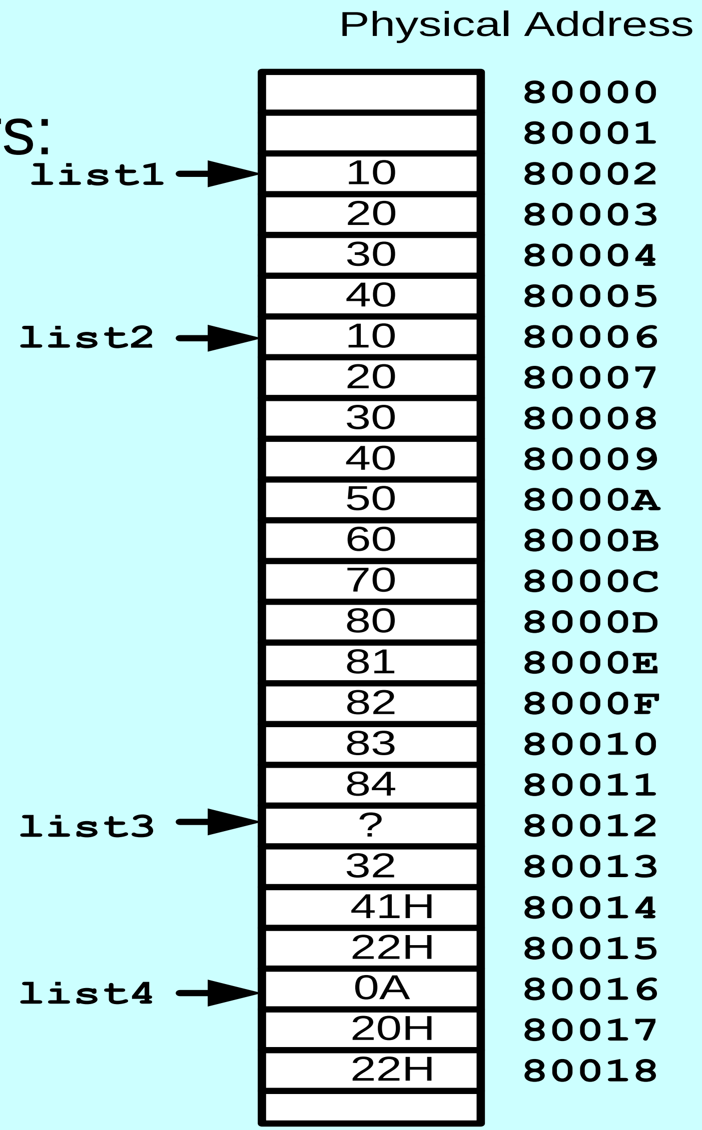


A variable name is a data label that implies an offset (an address).

Defining Bytes

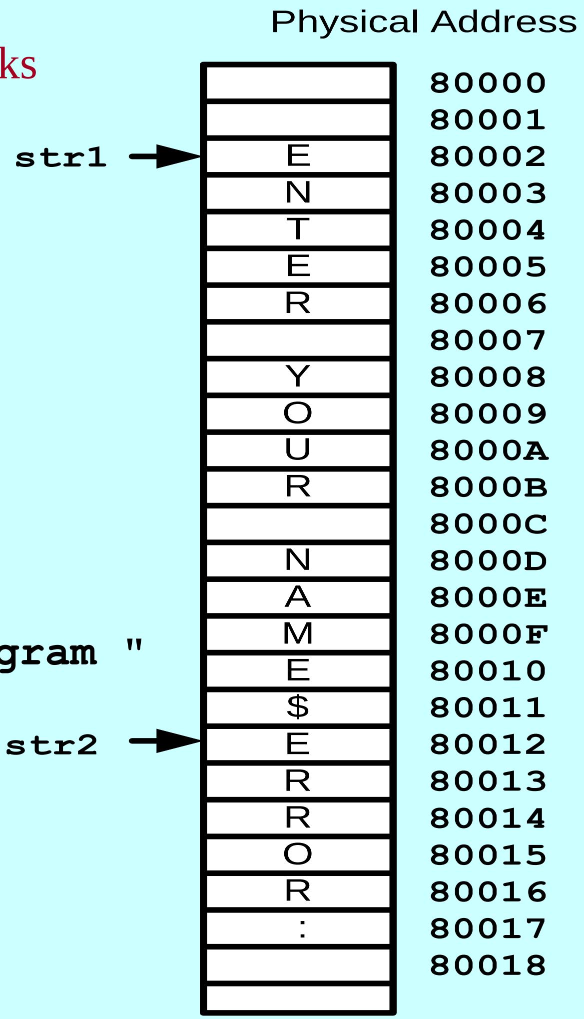
Examples that use multiple initializers:

```
list1 DB 10,20,30,40
list2 DB 10,20,30,40
      DB 50,60,70,80
      DB 81,82,83,84
list3 DB ?,32,41h,00100010b
list4 DB 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 usually has a null byte at the end
- Examples:




```
str1 DB "Enter your name", '$'
str2 DB 'Error: halting program', '$'
str3 DB 'A','E','I','O','U'
greeting  DB "Welcome to the Encryption Demo program "
          DB "created by someone.", '$'
```

Defining Strings (2 of 3)

- To continue a single string across multiple lines, end each line with a comma:

```
menu DB "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> ", '$'
```

Defining Strings (3 of 3)

- End-of-line character sequence:
  - ∗ 0Dh = carriage return
  - ∗ 0Ah = line feed

```
str1 DB "Enter your name:     ",0Dh,0Ah
     DB "Enter your address: ",'$'

newLine DB 0Dh,0Ah, '$'
```

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

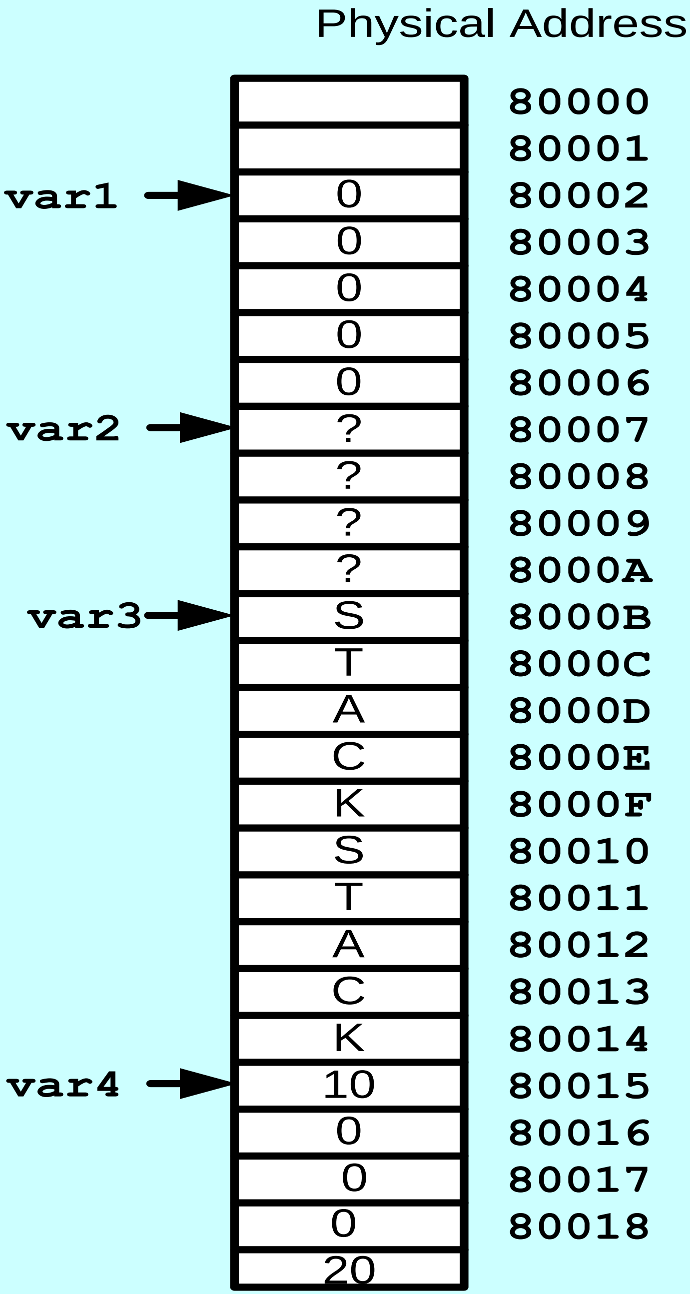
```
var1 DB 5 DUP(0)                  ; 20 bytes, all equal to zero
var2 DB 4 DUP(?)                  ; 20 bytes, uninitialized
var3 DB 4 DUP("STACK")          ; 20 bytes: "STACKSTACKSTACKSTACK"
var4 DB 10,3 DUP(0),20
```

```
var1 DB 5 DUP(0)

var2 DB 4 DUP(?)

var3 DB 2 DUP("STACK")

var4 DB 10,3 DUP(0),20
```

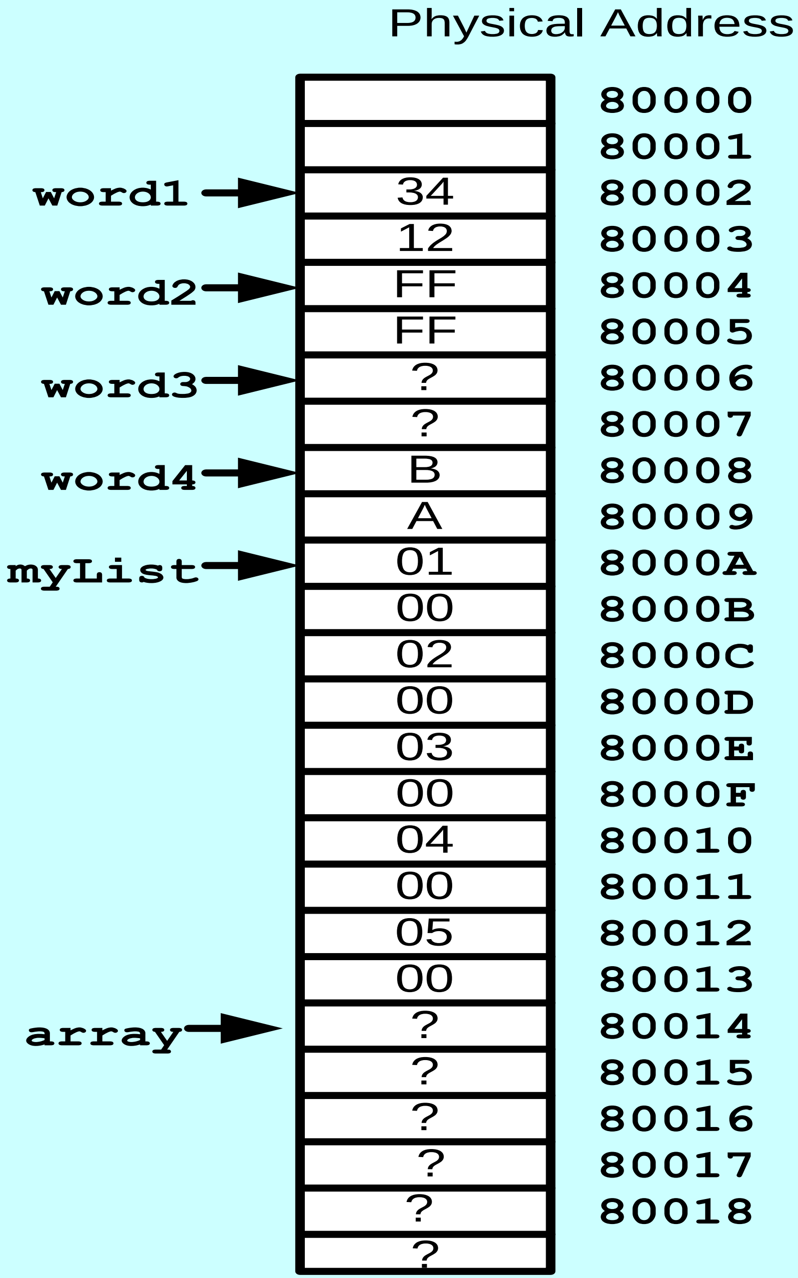


Defining DW

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

```
word1 DW   1234H            ; largest unsigned value
word2 DW -1                 ; smallest signed value
word3 DW   ?                ; uninitialized, unsigned
word4 DW "AB"               ; double characters
myList DW 1,2,3,4,5         ; array of words
array DW 5 DUP(?)           ; uninitialized array
```

```
word1 DW   1234H
word2 DW -1
word3 DW   ?
word4 DW "AB"
myList DW 1,2,3,4,5
array DW 5 DUP(?)
```

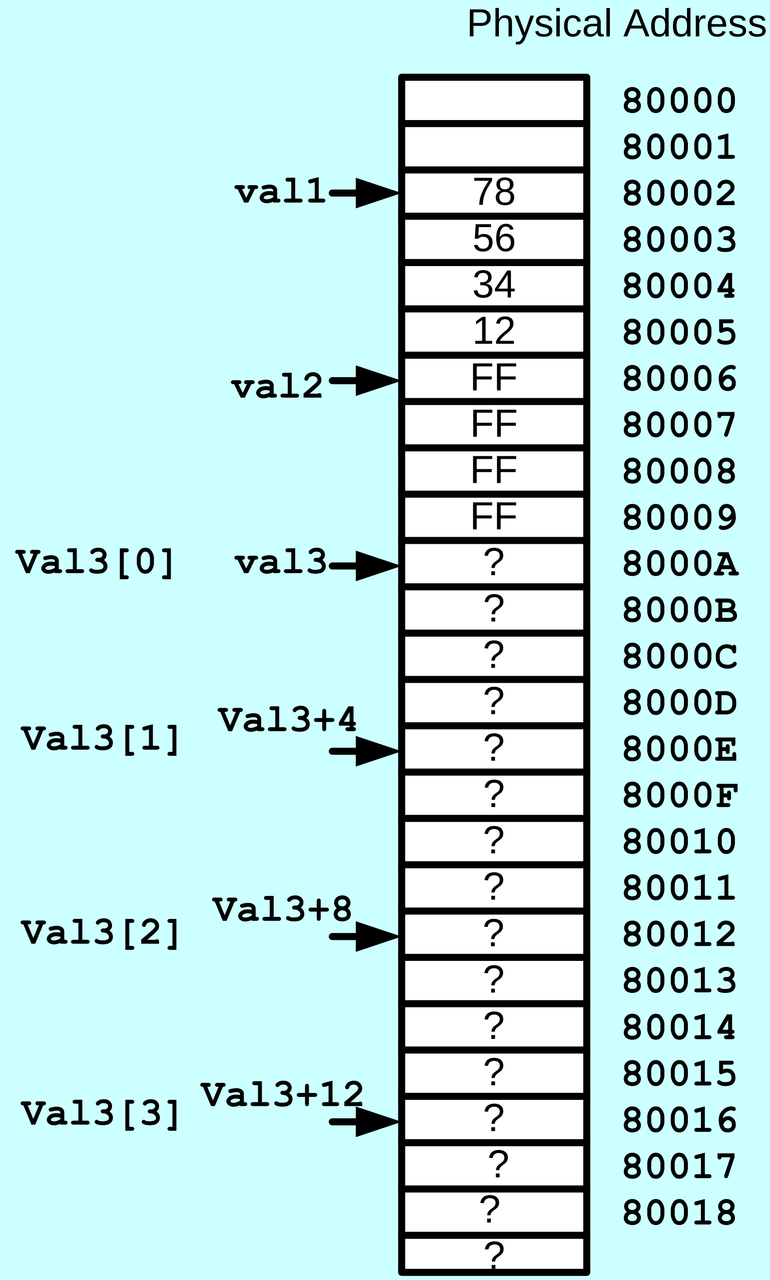


Defining DD

Storage definitions for signed and unsigned 32-bit integers:

```
val1 DD 12345678h             ; unsigned
val2 DD -1                    ; signed
val3 DD 20 DUP(?)             ; unsigned array
val4 DD -3,-2,-1,0,1          ; signed array
```

```
val1 DD 12345678h
val2 DD -1
val3 DD 20 DUP(?)
val4 DD -3,-2,-1,0,1
```



Defining QB, TB

Storage definitions for quadwords, tenbyte values, and real numbers:

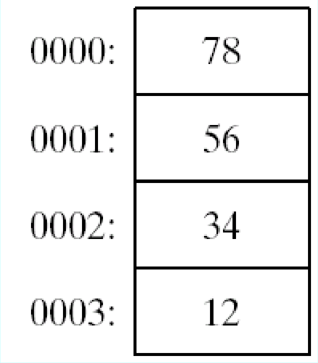
```
quad1 DQ  1234567812345678h
val1  DT  1000000000123456789Ah
```

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 DD 12345678h
```



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 DB pressKey
```

Addressing Modes

Where Are the Operands?

- Operands required by an operation can be specified in a variety of ways
- A few basic ways are:
  - * operand in a register
    - – register addressing mode
  - * operand in the instruction itself
    - – immediate addressing mode
  - * operand in memory
    - – variety of addressing modes
      - ✈ direct and indirect addressing modes
  - * operand at an I/O port
    - – Simple IN and OUT commands

Register Addressing

* Operand is in an internal register

Examples

```
mov     EAX,EBX   ; 32-bit copy
mov     BX,CX     ; 16-bit copy
mov     AL,CL     ; 8-bit copy
```

* The **`mov`** instruction

  `mov    destination,source`

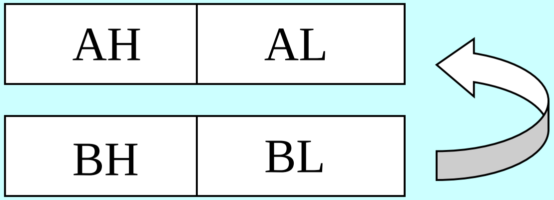
  copies data from **`source`** to **`destination`**

Register Addressing

- Operands of the instruction are the names of internal register
- The processor gets data from the register locations specified by instruction operands

For Example: *move the value of register BL to register AL*

MOV AL, BL



- If AX = 1000H and BX=A080H, after the execution of MOV AL, BL what are the new values of AX and BX?

In immediate and register addressing modes, the processor does not access memory. Thus, the execution of such instructions are fast.

Immediate Addressing Mode

Data is part of the instruction

» Operand is located in the code segment along with the instruction

» Typically used to specify a constant

Example

```
mov     AL,75
```

* This instruction uses register addressing mode for *destination* and immediate addressing mode for the *source*

Direct Addressing Mode

Data is in the data segment

» Need a logical address to access data
  – Two components: segment:offset
» Various addressing modes to specify the offset component
  – offset part is called ***effective address***

* The offset is specified directly as part of instruction
* We write assembly language programs using memory labels (e.g., declared using DB, DW, LABEL,...)
  » Assembler computes the offset value for the label
    – Uses symbol table to compute the offset of a label

Direct Addressing Mode

* Assembler builds a symbol table so we can refer to the allocated storage space by the associated label

Example

```
.DATA
value    DW   0
sum      DD   0
marks    DW   10 DUP (?)
message  DB   'The grade is:',0
char1    DB   ?
```

name	offset
value	0
sum	2
marks	6
message	26
char1	40

Direct Addressing Mode

Examples

```
mov     AL,char1
```

- » Assembler replaces **`char1`** by its effective address (i.e., its offset value from the symbol table)

```
mov     marks,56
```

- » **`marks`** is declared as

```
marks   DW   10 DUP (0)
```

- » Since the assembler replaces **`marks`** by its effective address, this instruction refers to the first element of **`marks`**
  - – In C, it is equivalent to

```
table1[0] = 56
```

Direct Addressing Example

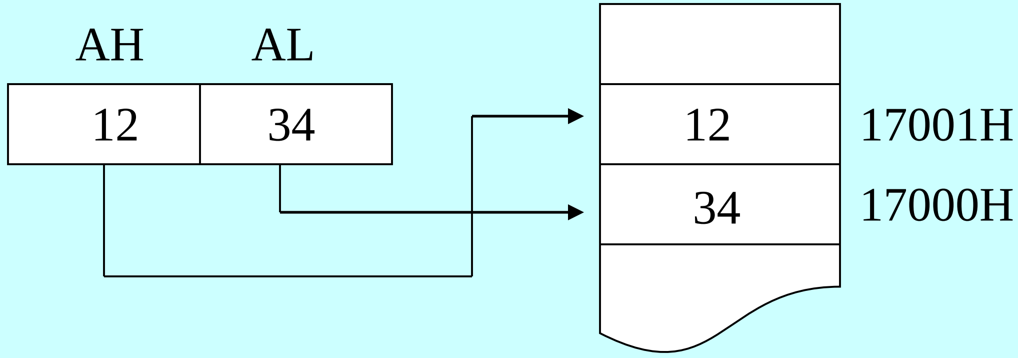
DS × 10H + Displacement = Memory location

— Example: *assume DS = 1000H, AX = 1234H*

MOV [7000H], AX

DS: 1 0 0 0 _
\+ Disp: 7 0 0 0

1 7 0 0 0



Direct Addressing Mode

- Problem with direct addressing
  - ∗ Useful only to specify simple variables
  - ∗ Causes serious problems in addressing data types such as arrays
    - » As an example, consider adding elements of an array
      - – Direct addressing does not facilitate using a loop structure to iterate through the array
      - – We have to write an instruction to add each element of the array
- Indirect addressing mode remedies this problem

Register Indirect Addressing

- One of the registers BX, BP, SI, DI appears in the instruction operand field. Its value is used as the memory displacement value.

  For Example: MOV DL, [SI]

- Memory address is calculated as following:

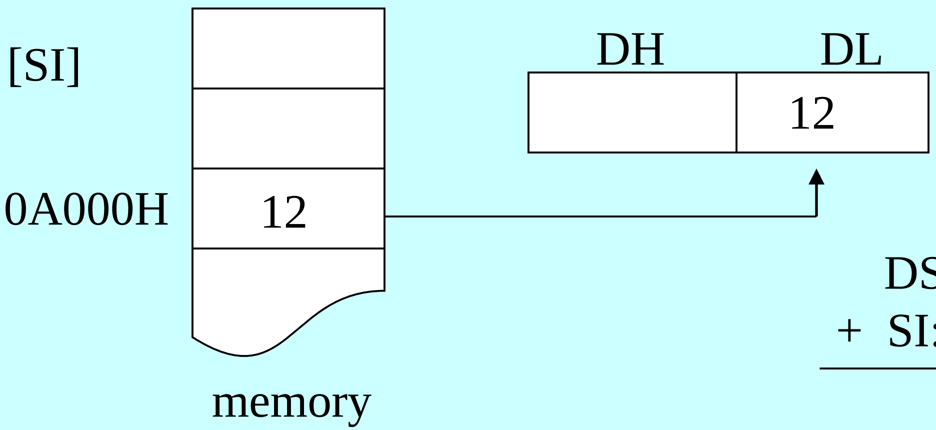
$$\begin{pmatrix} \text{DS} \\ \\ \text{SS} \end{pmatrix} \times 10\text{H} + \begin{pmatrix} \text{BX} \\ \text{SI} \\ \text{DI} \\ \\ \text{BP} \end{pmatrix} = \text{Memory address}$$

  - ❑ If BX, SI, or DI appears in the instruction operand field, segment register DS is used in address calculation
  - ❑ If BP appears in the instruction operand field, segment register SS is used in address calculation

Register Indirect Addressing

- Example 1: *assume DS = 0800H, SI=2000H*

MOV DL, [SI]




```
   DS:   0 8 0 0 _
+  SI:     2 0 0 0
------------------
         0 A 0 0 0
```

- Example 2: *assume SS = 0800H, BP=2000H, DL = 7*

MOV [BP], DL

Register Indirect Addressing

- Using indirect addressing mode, we can process arrays using loops

Example: Summing array elements

- Load the starting address (i.e., offset) of the array into BX
- Loop for each element in the array
  - » Get the value using the offset in BX
    - – Use indirect addressing
  - » Add the value to the running total
  - » Update the offset in BX to point to the next element of the array

Register Indirect Addressing
Loading offset value into a register

- Suppose we want to load BX with the offset value of **`table1`**
- We cannot write

```
mov     BX,table1
```

- Two ways of loading offset value
  - » Using OFFSET assembler directive
    - – Executed only at the assembly time
  - » Using **`lea`** instruction
    - – This is a processor instruction
    - – Executed at run time

Register Indirect Addressing
Loading offset value into a register (cont'd)

- Using **OFFSET** assembler directive
  - The previous example can be written as

```
mov     BX,OFFSET table1
```

- Using **`lea`** (load effective address) instruction
  - The format of **`lea`** instruction is

```
lea     register,source
```

  - The previous example can be written as

```
lea     BX,table1
```

Register Indirect Addressing
Loading offset value into a register (cont'd)

Which one to use -- OFFSET or **`lea`**?

* Use OFFSET if possible
  * » OFFSET incurs only one-time overhead (at assembly time)
  * » **`lea`** incurs run time overhead (every time you run the program)
* May have to use **`lea`** in some instances
  * » When the needed data is available at run time only
    * – An index passed as a parameter to a procedure
  * » We can write

    ```
    lea   BX,table1[SI]
    ```

    to load BX with the address of an element of **`table1`** whose index is in SI register
  * » We cannot use the OFFSET directive in this case

Based Addressing

- The operand field of the instruction contains a base register (BX or BP) and an 8-bit (or 16-bit) constant (displacement)

For Example: MOV AX, [BX+4]

- Calculate memory address

$$\begin{bmatrix} DS \\ \\ SS \end{bmatrix} \times 10H + \begin{bmatrix} BX \\ \\ BP \end{bmatrix} + \text{Displacement} = \text{Memory address}$$

- ❑ If BX appears in the instruction operand field, segment register DS is used in address calculation
- ❑ If BP appears in the instruction operand field, segment register SS is used in address calculation

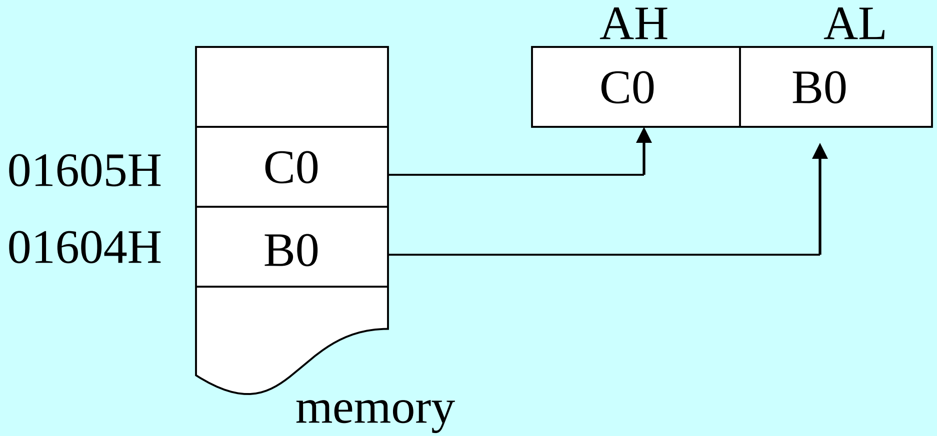
What's difference between register indirect addressing and based addressing?

Based Addressing

- Example 1: *assume DS = 0100H, BX=0600H*

MOV AX, [BX+4]

```
     DS:     0 1 0 0  _
+    BX:       0 6 0  0
+    Disp.:  0 0 0  4
--------------------------
             0 1 6 0 4
```



- Example 2: *assume SS = 0A00H, BP=0012H, CH = ABH*

MOV [BP-7], CH

Indexed Addressing

- The operand field of the instruction contains an index register (SI or DI) and an 8-bit (or 16-bit) constant (displacement)

For Example: MOV [DI-8], BL

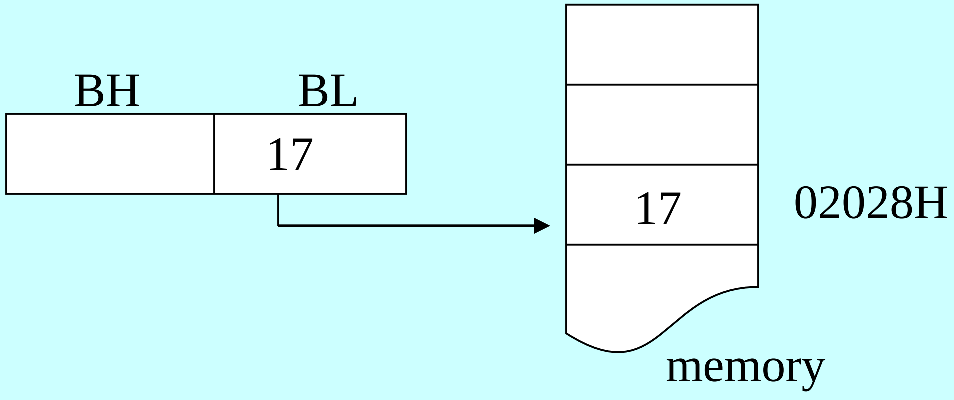
- Calculate memory address

$$DS \times 10H + \left\{ \begin{array}{c} SI \\ DI \end{array} \right\} + \text{Displacement} = \text{Memory address}$$

- Example: *assume DS = 0200H, DI=0030H BL = 17H*

MOV [DI-8], BL

```
    DS:    0 2 0 0 _
+   DI:      0 0 3 0
-   Disp.:   0 0 0 8
--------------------
           0 2 0 2 8
```



Based Indexed Addressing

- The operand field of the instruction contains a base register (BX or BP) and an index register

  For Example: MOV [BP] [SI], AH

  or MOV [BP+SI], AH

- Calculate memory address

$$\begin{bmatrix} \text{DS} \\ \\ \text{SS} \end{bmatrix} \times 10\text{H} + \begin{bmatrix} \text{BX} \\ \\ \text{BP} \end{bmatrix} + \{\text{SI } \textit{or} \text{ DI}\} = \text{Memory address}$$

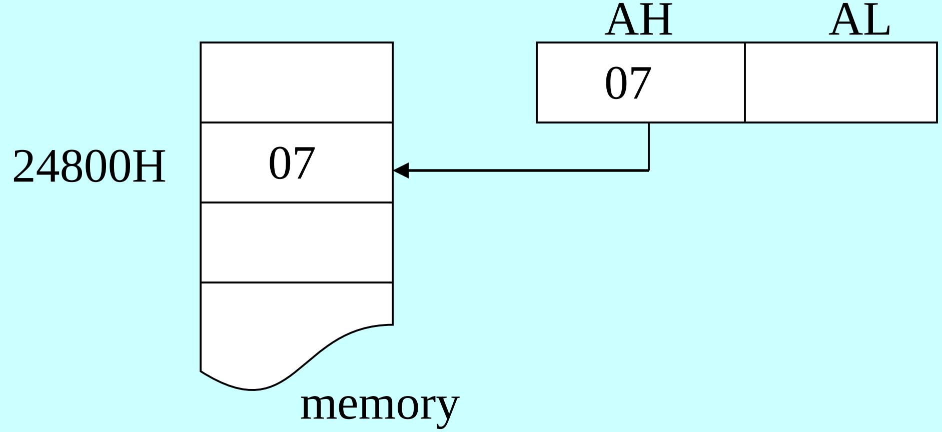
- If BX appears in the instruction operand field, segment register DS is used in address calculation
- If BP appears in the instruction operand field, segment register SS is used in address calculation

Based Indexed Addressing

- Example 1: *assume SS = 2000H, BP=4000H, SI=0800H, AH=07H*

MOV [BP] [SI], AH

```
    SS:   2 0 0 0  _
+  BP:      4 0 0  0
+  SI.:      0 8 0  0
-----------------------
          2 4 8 0 0
```



- Example 2: *assume DS = 0B00H, BX=0112H, DI = 0003H, CH=ABH*

MOV [BX+DI], CH

Based Indexed with Displacement Addressing

- The operand field of the instruction contains a base register (BX or BP), an index register, and a displacement

For Example: MOV CL, [BX+DI+2080H]

- Calculate memory address

$$\begin{bmatrix} \text{DS} \\ \\ \text{SS} \end{bmatrix} \times 10\text{H} + \begin{bmatrix} \text{BX} \\ \\ \text{BP} \end{bmatrix} + \{\text{SI } \textit{or} \text{ DI}\} + \text{Disp.} = \text{Memory address}$$

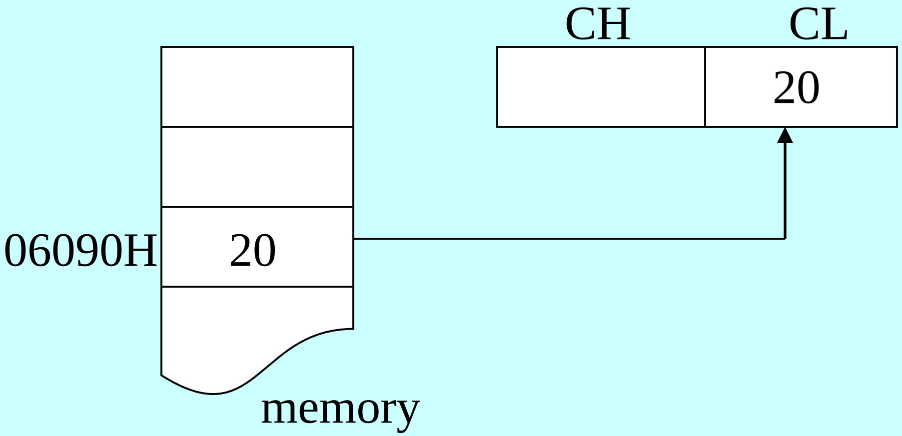
- ❑ If BX appears in the instruction operand field, segment register DS is used in address calculation
- ❑ If BP appears in the instruction operand field, segment register SS is used in address calculation

Based Indexed with Displacement Addressing

- Example 1: *assume DS = 0300H, BX=1000H, DI=0010H*

MOV CL, [BX+DI+2080H]

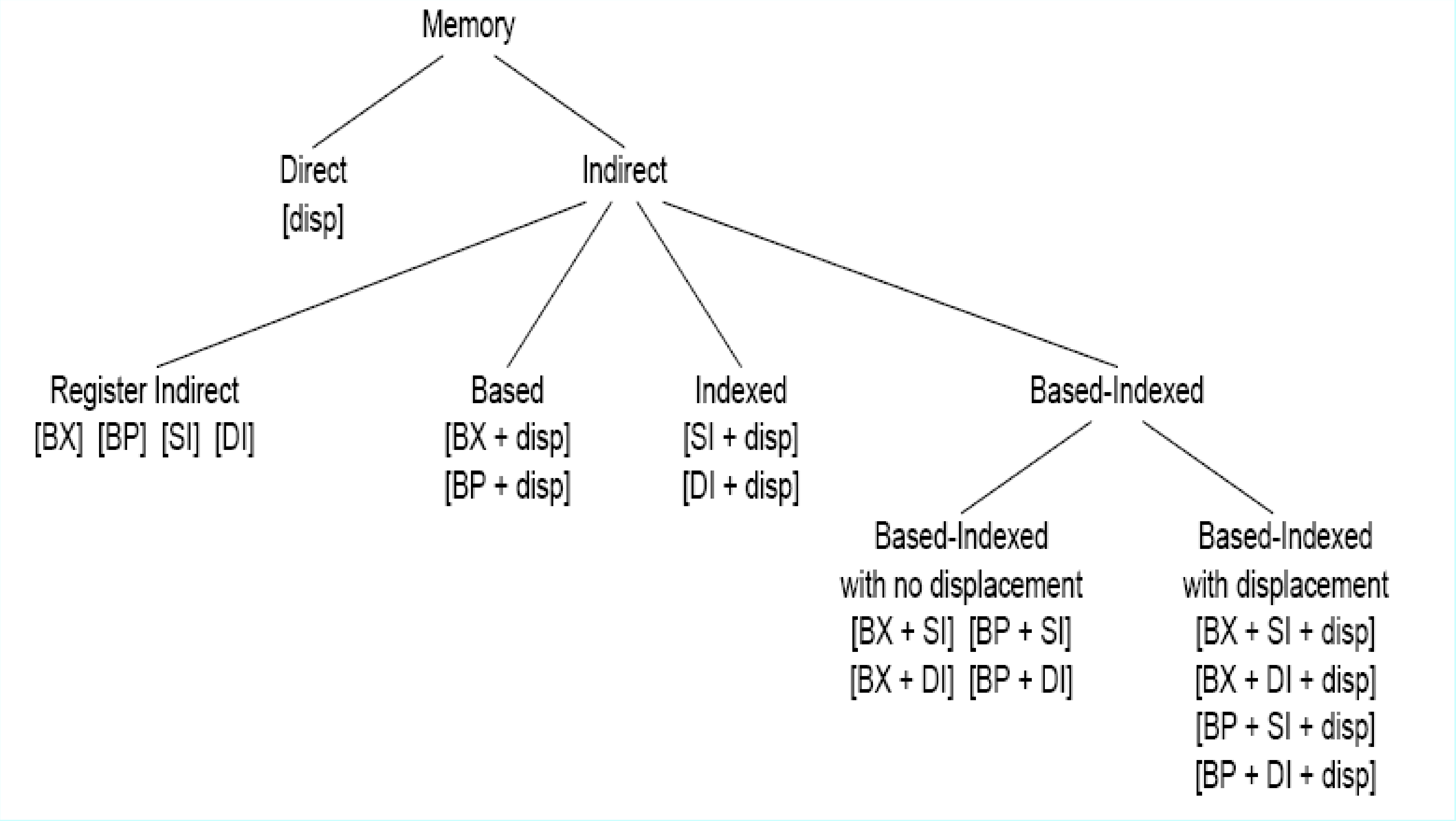
```
   DS:    0 3 0 0  _
+  BX:      1 0 0  0
+  DI.:     0 0 1 0
+ Disp.     2 0 8 0
--------------------
          0 6 0 9 0
```



- Example 2: *assume SS = 1100H, BP=0110H, SI = 000AH, CH=ABH*

MOV [BP+SI+0010H], CH

Addressing Modes: Summary



Default Segments

- In register indirect addressing mode
  - 16-bit addresses
    - Effective addresses in BX, SI, or DI is taken as the offset into the data segment (relative to DS)
    - For BP and SP registers, the offset is taken to refer to the stack segment (relative to SS)
  - 32-bit addresses
    - Effective address in EAX, EBX, ECX, EDX, ESI, and EDI is relative to DS
    - Effective address in EBP and ESP is relative to SS
  - **`push`** and **`pop`** are always relative to SS

Default Segments (cont'd)

- Default segment override
  - ∗ Possible to override the defaults by using override prefixes
    - » **CS, DS, SS, ES**
  - ∗ Example 1
    - » We can use

```
add     AX,SS:[BX]
```

      to refer to a data item on the stack
  - ∗ Example 2
    - » We can use

```
add     AX,DS:[BP]
```

      to refer to a data item in the data segment

Data Transfer Instructions

The `mov` instruction

* The format is

```
mov     destination,source
```

  » Copies the value from **`source`** to **`destination`**

  » **`source`** is not altered as a result of copying

  » Both operands should be of same size

  » **`source`** and **`destination`** cannot both be in memory

  – Most Pentium instructions do not allow both operands to be located in memory

  – Pentium provides special instructions to facilitate memory-to-memory block copying of data

Data Transfer Instructions (cont'd)

The `mov` instruction

* Five types of operand combinations are allowed:

Instruction type	**Example**
`mov register,register`	`mov DX,CX`
`mov register,immediate`	`mov BL,100`
`mov register,memory`	`mov BX,count`
`mov memory,register`	`mov count,SI`
`mov memory,immediate`	`mov count,23`

* The operand combinations are valid for all instructions that require two operands

Data Transfer Instructions (cont'd)

Source Operand	**Destination Operand**			
	General Register	Segment Register	Memory Location	Constant
General Register	Yes	Yes	Yes	No
Segment Register	Yes	No	Yes	No
Memory Location	Yes	Yes	No	No
Constant	Yes	No	Yes	No

Data Transfer Instructions (cont'd)

Ambiguous moves: PTR directive

- For the following data definitions

```
.DATA
table1    DW    20 DUP (0)
status    DB    7 DUP (1)
```

the last two **mov** instructions are ambiguous

```
mov    BX,OFFSET table1
mov    SI,OFFSET status
mov    [BX],100
mov    [SI],100
```

* Not clear whether the assembler should use byte or word equivalent of 100

Data Transfer Instructions (cont'd)

Ambiguous moves: PTR directive

- The PTR assembler directive can be used to clarify
- The last two **`mov`** instructions can be written as

```
mov     WORD PTR [BX],100
mov     BYTE PTR [SI],100
```

  * WORD and BYTE are called ***type specifiers***
- We can also use the following type specifiers:

`DWORD` for doubleword values

`QWORD` for quadword values

`TWORD` for ten byte values

Data Transfer Instructions (cont'd)

The `xchg` instruction

- The syntax is

```
xchg    operand1,operand2
```

Exchanges the values of **operand1** and **operand2**

Examples

```
xchg    EAX,EDX
xchg    response,CL
xchg    total,DX
```

- Without the **xchg** instruction, we need a temporary register to exchange values using only the **mov** instruction

Data Transfer Instructions (cont'd)

The **`xchg`** instruction

- The **`xchg`** instruction is useful for conversion of 16-bit data between little endian and big endian forms
  - * Example:

    ```
    mov     AL,AH
    ```

    converts the data in AX into the other endian form
- Pentium provides **`bswap`** instruction to do similar conversion on 32-bit data

    ```
    bswap    32-bit register
    ```

  - * **`bswap`** works only on data located in a 32-bit register

Printing to Screen

- INT 21H, Function 02H (AH = 02H).
  - This function writes a single character to the screen.
  - It is a DOS routine
  - Example:

```
MOV AH, 02H
MOV DL, 'A' ; THE PRINTED CHARCTED SHOULD BE
                 PLACED HERE
INT 21H
```

- INT 21H, Function 09H (AH = 09H); DX contains the offset of string ending with $
  - This function displays a string.
  - Example:

```
MOV AH, 02H
LEA DX, msg ;
INT 21H
```

Reading from keybaord

- INT 21H, Function 01H (AH = 01H).
  - This function waits for keyboard Example:

```
MOV AH, 01H
INT 21H  ; AL will contain the key pressed.
```

- INT 10H, Function 02H (AH = 02H), will set the cursor position
  - DL contains the column number (0 to 79)
  - DH contains row number (0 to 24)
  - BH contains page number (default is 0)

```
MOV AH, 02H
MOV DL, 1
MOV DH, 1
MOV BH, 0
INT 10H
```

Example: Case Conversion

```
.data
    MSG1        DB      'Enter a lower case letter:$'
    MSG2        DB      0DH, 0AH, 'In Upper Case it is:'
    CharDB      ?,'$'

.code
    LEA DX, MSG1
    MOV AH, 9
    INT 21H
    MOV AH, 1
    INT 21H
    SUB AL, 20H
    MOV Char, AL
    LEA DX, MSG2
    MOV AH, 09H
    INT 21H
```