Data Transfer and Addressing

Contents

- Data Transfer
- Data Declaration
- Addressing Modes
  - Direct
  - Indirect

Data Transfer

- The MOV instruction copies data from a source operand to a destination operand.
- Known as a data transfer instruction, it is used in virtually every program.
- Its basic format shows that the first operand is the destination and the second operand is the source:

```
MOV destination, source
```

- It is equivalent to ***destination = source;*** in high level languages like in java or C++

Rules for MOV instruction

- MOV is very flexible in its use of operands, as long as the following rules are observed:
  - Both operands must be the same size.
  - Both operands cannot be memory operands.
  - The instruction pointer register (IP) cannot be a destination operand.

MOV instruction formats

- Here is a list of the standard MOV instruction formats:

```
MOV register, register
MOV register, immediate operand
MOV memory, register
MOV register, memory
MOV memory, immediate operand
```

Examples

```
[org 0x0100]

    MOV ax, 1 ; ax=1 here immediate operand is in decimal
    MOV bl, 0Ah  ;bl= 0A (immediate operdand is in hex)
    MOV bl, 8FE2h ; assembler will give a warning and will only move E2h to bl
    ;MOV ax, bl; assembler will  error "invalid combination of opcode and operand"
    Mov ah, 000010001b;  ; ah=000010001 , note that the immediate operand is in binary
    Mov ah, 10o ; ah=10o where o represents that the immediate operand is in octal
    Mov Cx, AX;  cx=ax



mov ax, 0x4c00 ; terminate program
int 0x21
```

Data Declaration

- Up till now we were either working with registers or immediate operands.
- Every reasonable program will require storing and retrieving data from memory.
  - Because registers are limited and or immediate operands need hardcoding

Data Declaration

- Data can be declared using following instruction

```
<label>:  <db/dw/dd> <values/s>
```

  - **db** will define a byte
  - **dw** will define a word (2 bytes)
  - **dd** will define double word (4 bytes)

  - Label is used by programmers to refer to the data again. Its like name of variable as used in HLL.
  - The size of data is not associated with label.
  - As a result a cell in memory will be reserved containing the desired value in it and it can be used in a variety of ways.

Data Declaration Examples

- Following is the code and listing file of declaring one **db**, one **dw** and one **dd** type. The data is declared below the code
- Note the labels num1, num2 and num3 appear nowhere in the machine code, they are symbol for us but an address for the processor while the conversion is done by the assembler.

```
;L3E2.asm Example of data declaration
[org 0x0100]

mov ax, 0x4c00 ; terminate program
int 0x21


num1: db 5
num2: dw 5
num3: dd 5
```

```
 1                                  ;L3E2.asm Example of data declaration
 2                                  [org 0x0100]
 3
 4 00000000 B8004C                  mov ax, 0x4c00 ; terminate program
 5 00000003 CD21                    int 0x21
 6
 7
 8 00000005 05                      num1: db 5
 9 00000006 0500                    num2: dw 5
10 00000008 05000000                num3: dd 5
11
```

Moving data to/from memory

- We will see with an example to add three numbers, present in memory, and store the result again in memory.
- Code is given below, note that label is written in square brackets
- The bracket is signaling that the operand is placed in memory at address num1.

```
;L3E4.asm Example of data declaration
[org 0x0100]

    mov ax, [num1] ; load first number in ax
    mov bx, [num2] ; load second number in bx
    add ax, bx ; accumulate sum in ax
    mov bx, [num3] ; load third number in bx
    add ax, bx ; accumulate sum in ax
    mov [sum], ax ; store sum in num4

    mov ax, 0x4c00 ; terminate program
    int 0x21

num1: dw 5
num2: dw 10
num3: dw 15
sum: dw 0 ; reserver a word in memory but value is not given
```

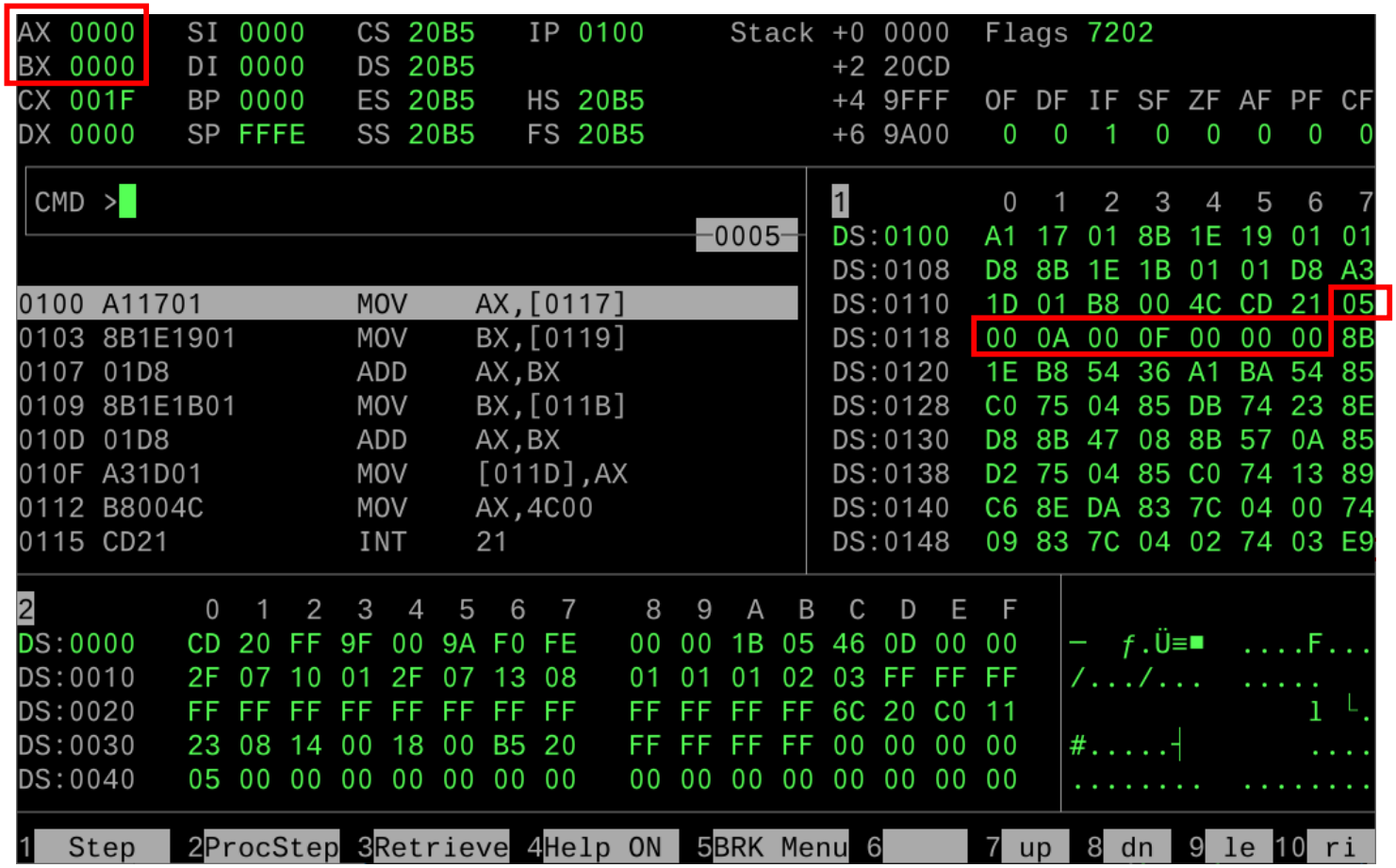
Example cont… listing file

- The size of the code is 17 bytes and from 17th to 1D byte the data is located
- Note the opcodes — the assembler has calculated the offset of num1 and used it to replace references to num1 in the whole program.

```
 1                                  ;L3E4.asm Example of data declaration
 2                                  [org 0x0100]
 3
 4 00000000 A1[1700]                    mov ax, [num1] ; load first number in ax
 5 00000003 8B1E[1900]                  mov bx, [num2] ; load second number in bx
 6 00000007 01D8                        add ax, bx ; accumulate sum in ax
 7 00000009 8B1E[1B00]                  mov bx, [num3] ; load third number in bx
 8 0000000D 01D8                        add ax, bx ; accumulate sum in ax
 9 0000000F A3[1D00]                    mov [sum], ax ; store sum in num4
10
11 00000012 B8004C                     mov ax, 0x4c00 ; terminate program
12 00000015 CD21                       int 0x21
13
14 00000017 0500                   num1: dw 5
15 00000019 0A00                   num2: dw 10
16 0000001B 0F00                   num3: dw 15
17 0000001D 0000                   sum: dw 0 ; reserver a word in memory but value is not given
```

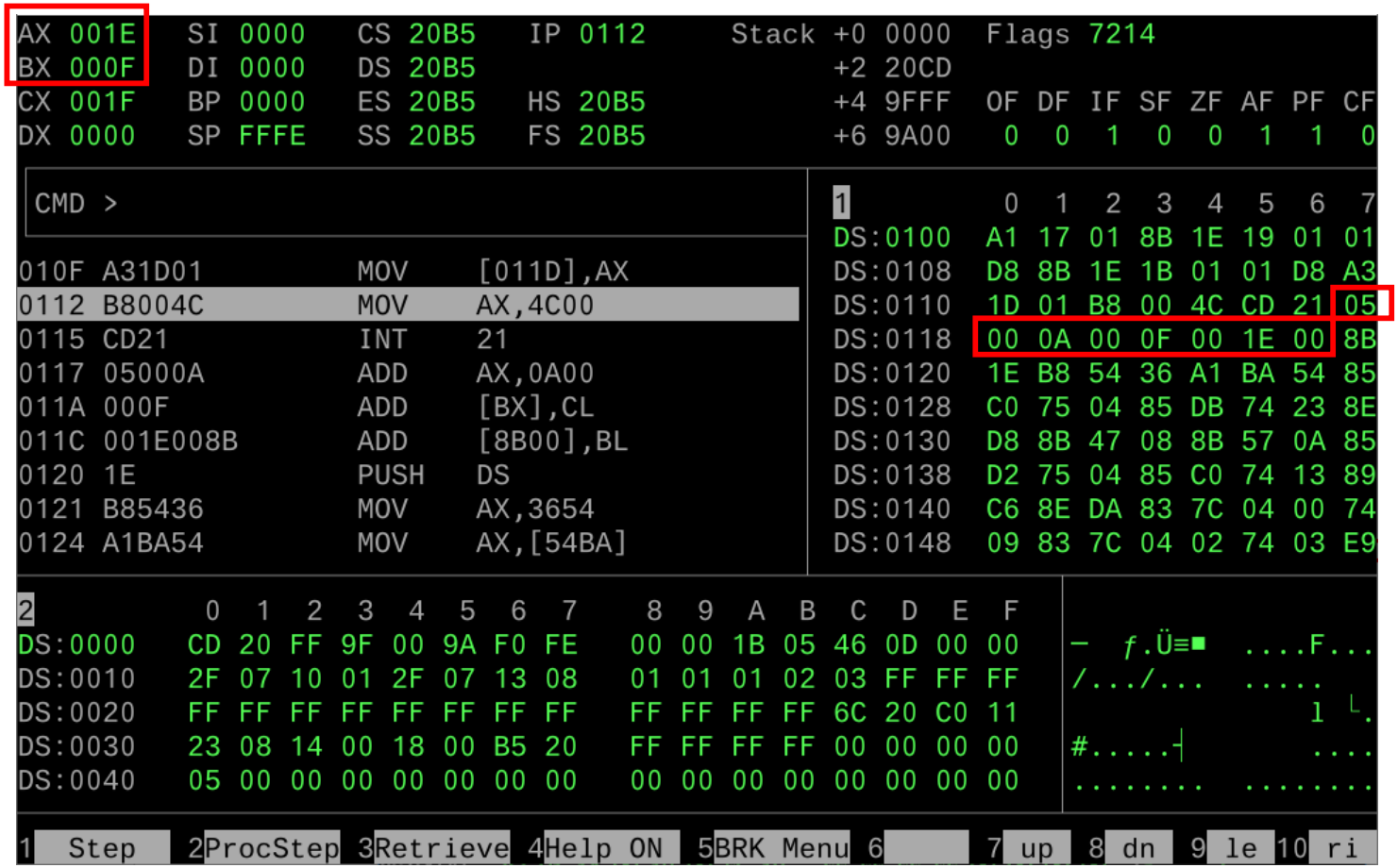
Example cont… in AFD

- Before running



Example cont… in AFD

- After running complete program



Examples of mov to/from memory

- Again note that the label is just one address in memory, there is no size associated with it.
- How much data is to be moved to and from memory depends on the size of source.
- Its is programmer's responsibility to carefully read and write data in memory.
- Following examples will explain each instruction in plain English.

```
mov [num1], ax ; move 2 bytes of register AX into memory, AL will be placed at num1 address and AH will be placed at num1+1
mov [num1], al ; move lower byte of register AX into memory, at address num1
mov ax, [num1] ; move 2 byte from memory to AX starting at address num1, byte at num1 will be moved to AL and byte at num1+1 Will be moved to AH.
mov ah, [num1] ; move 1 byte from memory to AH, starting from byte at num1
mov [num1], 5 ; will give an error as size of 5 is not known, solution given after a few slides
```

Difference between label and [label]

- The difference is same as ptr and *ptr in c++

```
;L3E5.asm difference between label and [label]
[org 0x0100]

     mov ax, [num1] ;  will load data at address num1 to ax
     mov ax, num1 ; will load the address of, this shows that assemble treats labels just as address

    mov ax, 0x4c00 ; terminate program
    int 0x21

num1: dw 5
```

```
 1                                  ;L3E5.asm difference between label and [label]
 2                                  [org 0x0100]
 3
 4 00000000 A1[0B00]                     mov ax, [num1] ;  will load data at address nu
 5 00000003 B8[0B00]                     mov ax, num1 ; will load the address of, this
 6
 7 00000006 B8004C                   mov ax, 0x4c00 ; terminate program
 8 00000009 CD21                     int 0x21
 9
10 0000000B 0500                    num1: dw 5
```

Other ways of declaring data (1)

- Declaring more than one number using one label:
  - Following example create three numbers of size 2 bytes each and stores them contiguously in memory.
  - These there numbers can be accessed using label
    - [num+offset]
    - Where offset is distance of data from label *nums* in bytes
    - *This is same as you did in C++ with pointers*
  - the offset of 5 is 0, offset of 10 is 2, offset of 15 is 4

```
nums:    dw 5
         dw 10
         dw 15
```

Example

```
;L3E6.asm add three numbers and store in sum
[org 0x0100]

    mov ax, [nums] ;  will load 5 in ax
    mov bx, [nums+2] ; will load 10 in bx
    add ax,bx
    mov bx, [nums+4] ; will load 15 in bx
    add ax,bx
    mov [sum], ax

    mov ax, 0x4c00 ; terminate program
    int 0x21

nums:   dw 5
        dw 10
        dw 15
sum:    dw 0
```

```
 1                                  ;L3E6.asm add three nu
 2                                  [org 0x0100]
 3
 4 00000000 A1[1700]                    mov ax, [nums] ;
 5 00000003 8B1E[1900]                  mov bx, [nums+2]
 6 00000007 01D8                        add ax,bx
 7 00000009 8B1E[1B00]                  mov bx, [nums+4]
 8 0000000D 01D8                        add ax,bx
 9 0000000F A3[1D00]                    mov [sum], ax
10
11 00000012 B8004C                     mov ax, 0x4c00 ; t
12 00000015 CD21                       int 0x21
13
14 00000017 0500                   nums:   dw 5
15 00000019 0A00                           dw 10
16 0000001B 0F00                           dw 15
17 0000001D 0000                   sum:    dw 0
```

Other ways of declaring data (1)

- Note the data under same label can also be of different type
- For example

```
nums:    dw 5
         db 10
         db 15
```

- The offset should be calculated accordingly.
- Offset of 5 is 0, offset of 10 is 2, offset of 15 is 3.

Other ways of declaring data (2)

- Following example shows another way to store more than one number, of same type under same label.
- Each number will be accessed in same way [label+offset].

```
;L3E7.asm add three numbers from memory and store in sum
[org 0x0100]

    mov ax, [nums] ;  will load 5 in ax
    mov bx, [nums+2] ; will load 10 in bx
    add ax,bx
    mov bx, [nums+4] ; will load 15 in bx
    add ax,bx
    mov [sum], ax

    mov ax, 0x4c00 ; terminate program
    int 0x21

nums:   dw 5, 10, 15
sum:    dw 0
```

Direct Addressing

- The method of using label name (or memory address) in square brackets to access data from memory is called direct addressing.
- `[label]` used as operand is known as **direct operand**.
- `[label+/-offset ]` used as operand is known as **direct offset operand**.

Question

- Is there any error in following program? If yes, identify.

```
[org 0x0100]

    mov [num1], [num2] ;

    mov ax, 0x4c00 ; terminate program
    int 0x21

num1:   dw 5
num2:   dw 10
```

Question?

- What will be effect of following instructions?
  - mov ax, 0x0100
  - mov ax, [0100]

Question

- In which format data is written in memory?
  - Big endian
  - Little endian

Size Mismatch Errors

- The assembler allows the programmer to do everything he wants to do, and that can possibly run on the processor.
- The assembler only keeps us from writing illegal instructions which the processor cannot execute. That is, it only checks the syntax errors not the logical
- The programmer is responsible for accessing the data as word if it was declared as a word and accessing it as a byte if it was declared as a byte
- Keeping that in mind, identify what is wrong in code given on next slide

Identify the error

```
01      ; a program to add three numbers directly in memory
02      [org 0x0100]
03                  mov  ax, [num1]          ; load first number in ax
04                  mov  [num1+6], ax        ; store first number in result
05                  mov  ax, [num1+2]        ; load second number in ax
06                  add  [num1+6], ax        ; add second number to result
07                  mov  ax, [num1+4]        ; load third number in ax
08                  add  [num1+6], ax        ; add third number to result
09
10                  mov  ax, 0x4c00          ; terminate program
11                  int  0x21
12
13      num1:       dw   5, 10, 15, 0
```

After Correction

```
001     ; a program to add three numbers using byte variables
002     [org 0x0100]
003                     mov  al, [num1]          ; load first number in al
004                     mov  bl, [num1+1]        ; load second number in bl
005                     add  al, bl              ; accumulate sum in al
006                     mov  bl, [num1+2]        ; load third number in bl
007                     add  al, bl              ; accumulate sum in al
008                     mov  [num1+3], al        ; store sum at num1+3
009
010                     mov  ax, 0x4c00          ; terminate program
011                     int  0x21
012
013     num1:           db   5, 10, 15, 0
```

Size Matching Error

- The instruction "`mov [num1], 5`" is legal but there is no way for the processor to know the data movement size in this operation.
- The variable `num1` can be treated as a byte or as a word and similarly `5` can be treated as a byte or as a word.
- Such instructions are declared ambiguous by the assembler.
- Therefore, to resolve its ambiguity we clearly tell our intent to the assembler in one of the following ways.
  - `mov byte [num1], 5`
  - `mov word [num1], 5`

Indirect Addressing

- Direct addressing is rarely used for **array processing** because it is impractical to use constant offsets to address more than a few array elements.
- Instead, we use a register as a pointer (called indirect addressing) and manipulate the register's value.
- When an operand uses indirect addressing, it is called an **indirect operand**

Register Indirect Addressing

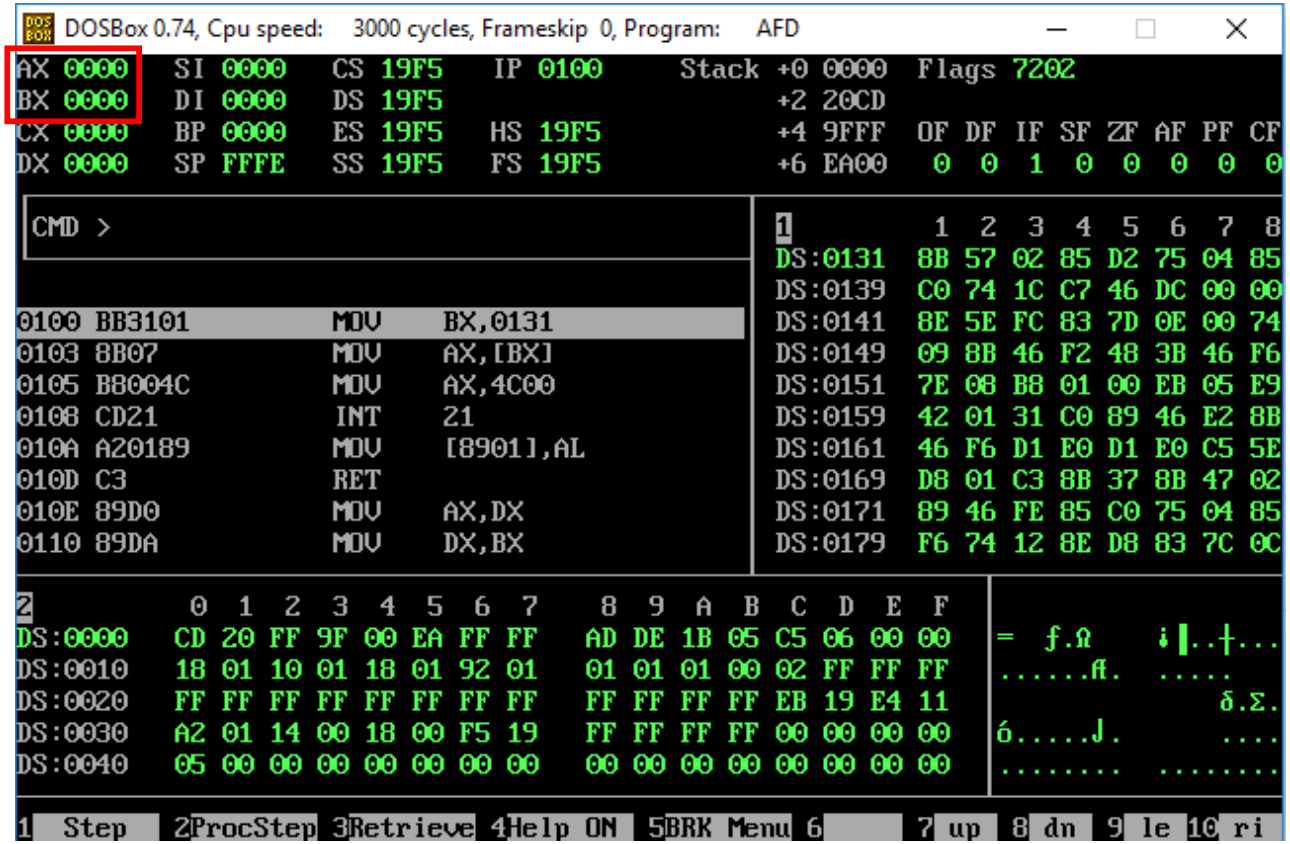
- When address is stored in a register, then register can be used as a pointer to that location in memory to access data
- For example if bx=0131h then following will move two bytes starting at location 0131.

```
mov ax, [bx]
```

  - The byte located on 0131 will be transferred to AL and byte on 0132 will be transferred to AH (because data in memory is stored in little endian format).

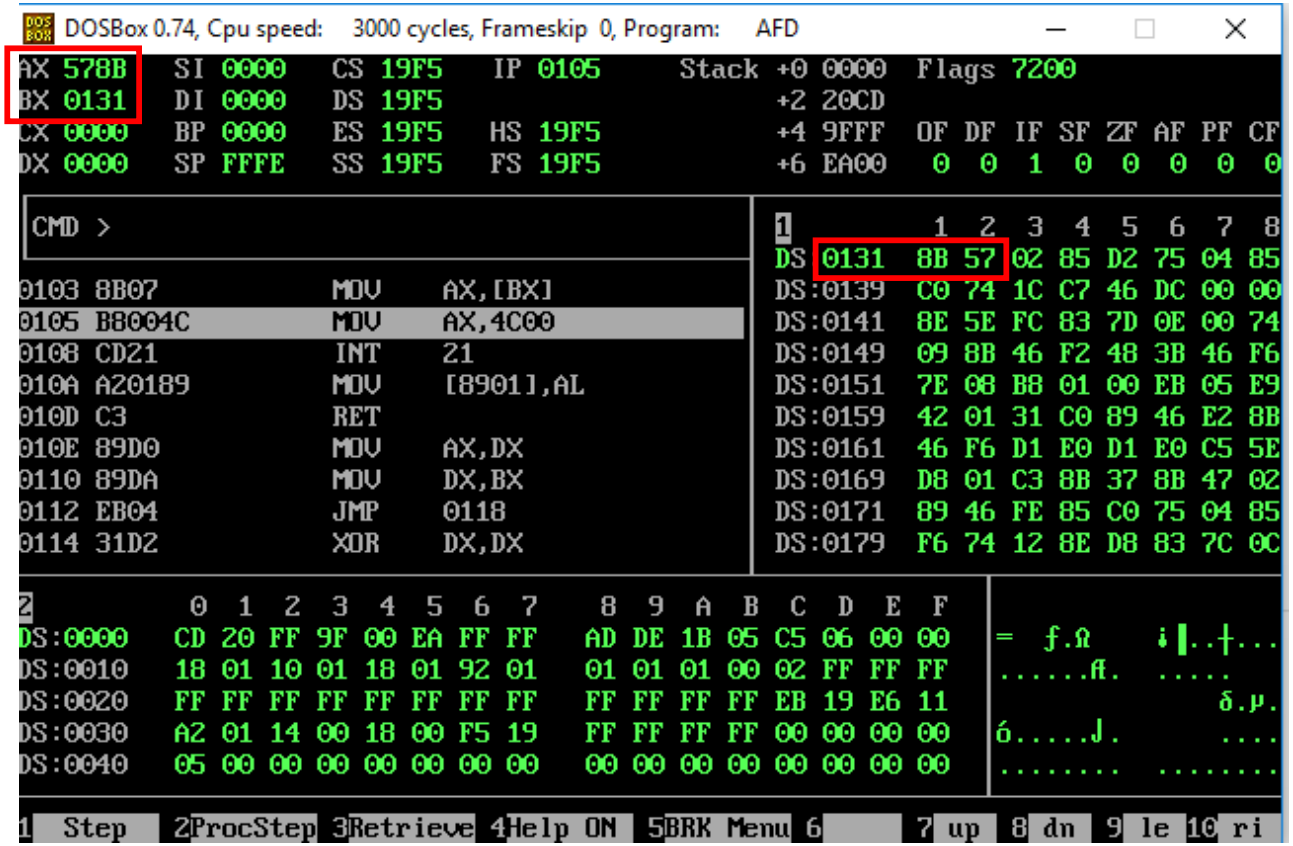
Register Indirect Addressing Example

- Before running



Register Indirect Addressing Example

- After running line 1 and 2



Register Indirect Addressing cont..

- There are four registers in iAPX88 architecture that can hold address of data and they are
  - BX and BP (base registers)
  - SI and DI (index registers)
- There are minute differences in their working which will be discussed later.
- Let us see an example of adding numbers in loop using register indirect addressing.
- The given example is for 10 numbers, but you can get an idea how useful it will be if we have to add hundreds of numbers

Example

```
; a program to add ten numbers
[org 0x0100]
        mov bx, num1 ; point bx to first number
        mov cx, 10 ; load count of numbers in cx
        mov ax, 0 ; initialize sum to zero
       l1: add ax, [bx] ; add number to ax
        add bx, 2 ; advance bx to next number
        sub cx, 1 ; numbers to be added reduced
        jnz l1 ; if numbers remain add next
        mov [total], ax ; write back sum in memory
        mov ax, 0x4c00 ; terminate program
        int 0x21
num1: dw 10, 20, 30, 40, 50, 10, 20, 30, 40, 50
total: dw 0
```

Things to Note:

- `l1` is code label. Again it is just an address.
- Assembler will treat code label in same way as data label.
- Every where `l1` occurs in code, assembler will convert it to address
- `jnz` will use the zero flag to make a decision.
- Zero flag becomes 0 if result of arithmetic operation was zero.
- Why is 2 added in bx?

Example (listing file)

```
 1                                  ; a program to add ten numbers
 2                                  [org 0x0100]
 3 00000000 BB[1D00]                        mov bx, num1 ; point bx to first number
 4 00000003 B90A00                          mov cx, 10 ; load count of numbers in cx
 5 00000006 B80000                          mov ax, 0 ; initialize sum to zero
 6 00000009 0307                           l1: add ax, [bx] ; add number to ax
 7 0000000B 81C30200                        add bx, 2 ; advance bx to next number
 8 0000000F 81E90100                        sub cx, 1 ; numbers to be added reduced
 9 00000013 75F4                            jnz l1 ; if numbers remain add next
10 00000015 A3[3100]                        mov [total], ax ; write back sum in memory
11 00000018 B8004C                          mov ax, 0x4c00 ; terminate program
12 0000001B CD21                            int 0x21
13 0000001D 0A0014001E00280032-     num1: dw 10, 20, 30, 40, 50, 10, 20, 30, 40, 50
14 00000026 000A0014001E002800-
15 0000002F 3200
16 00000031 0000                    total: dw 0
```

Combination of Direct and Indirect Addressing

- Direct and indirect addressing modes can be using in combination.
- Some examples are

```
mov ax, [num1+bx]; where bx contains the offset
mov ax, [bx+300]; indirect addressing with offset
mov ax, [bx+si];
```

Addressing Modes Summary (1)

Addressing Mode	Example
Direct	
• without offset	`mov [total], ax`
• Direct + offset	`mov [nums+2], bx`
Indirect	
• Based Register Indirect	`mov [bx], ax`
• Index Register Indirect	`mov [di], bx`
• Based Register Indirect + offset	`mov [bx+300], ax`
• Index Register Indirect + offset	`mov [di+300], al`
• Base + Index	`mov [bp+di], ah`
• Base + Index + offset	`mov [bx+si+300], ax`

Addressing Modes Summary (2)

- Things that are not allowed:
  - Base register + base register, e.g. ~~[BX+BP]~~
  - Index register + index register, e.g. ~~[SI+DI]~~
  - Base minus index, e.g. ~~[BX-SI]~~
    - Offsets can be subtracted though e.g. [BX+SI-200]
  - Part of register cannot be used to access memory address e.g. ~~[BH]~~ or ~~[BL]~~
    - Addresses in program are always 16 bits
    - Something like `mov [bp], al` is fine

Important thing to remember

- Programmer has a full control of memory.
- If you write any (valid) 16 bit address in square brackets you will be able to access it , either it is in form of label/registers +/- offset or simple constant number.
- It is up to you to access it carefully without creating logical errors

Reading

- BH 2.1 to 2.5, 2.8