

Name : Tazmeen Afroz  
Roll No: 22p-9252  
Section: BAI-4A

### COAL LAB TASK 3

file 1 : c02-01.asm

```
c02-00.asm  X      c02-00b.asm  X
1 ; a program to add three numbers using memory variables
2 [org 0x0100]
3
4 mov ax, [num1]      ; load first number in ax
5 ; mov [num1], [num2] ; illegal
6 mov bx, [num2]
7 add ax, bx
8 mov bx, [num3]
9 add ax, bx
10 mov [num4], ax
11 mov ax, 0x4C00
12 int 0x21
13
14
15 num1: dw 5
16 num2: dw 10
17 num3: dw 15
18 num4: dw 0
19
20
21 ; watch the listing carefully
```

#### 1. Data Section:

- The lines num1: dw 5, num2: dw 10, num3: dw 15, and num4: dw 0 define four **data words** (16 Bits) named num1, num2, num3, and num4. These words **reserve memory locations** and **initialize them with the given values** (5, 10, 15, and 0, respectively).

#### 2. Code Section:

- Instruction 1:** mov ax, [num1]

- This instruction loads the value stored at the memory address of num1 into the ax register. Since num1 holds 5, this instruction effectively moves 5 into ax.
- The [] brackets indicate **memory operands** (accessing data at a specific memory location).
- **Instruction 2:** mov [num1], [num2] (illegal)  
In older computer architectures, directly moving data between memory locations without going through registers was generally not possible due to limitations in the bus system.
- **Instruction 7:** mov [num4], ax
- This instruction stores the value in ax (30) into the memory location of num4.

DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, Program: AFD

Register	Value	Register	Value	Register	Value	Register	Value	Stack	Flags
AX	0000	SI	0000	CS	19F5	IP	0100	+0	0000
BX	0000	DI	0000	DS	19F5			+2	20CD
CX	001F	BP	0000	ES	19F5	HS	19F5	+4	9FFF
DX	0000	SP	FFFE	SS	19F5	FS	19F5	+6	EA00

CMD > 0005

Address	Instruction	Comment
0100	A11701	MOV AX, [0117]
0103	8B1E1901	MOV BX, [0119]
0107	01D8	ADD AX, BX
0109	8B1E1B01	MOV BX, [011B]
010D	01D8	ADD AX, BX
010F	A31D01	MOV [011D], AX
0112	B8004C	MOV AX, 4C00
0115	CD21	INT 21

Address	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
DS:0000	CD	20	FF	9F	00	EA	F0	FE	AD	DE	1B	05	C5	06	00	00
DS:0010	18	01	10	01	18	01	92	01	01	01	00	02	FF	FF	FF	FF
DS:0020	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	EB	19	C0	11
DS:0030	A2	01	14	00	18	00	F5	19	FF	FF	FF	FF	00	00	00	00
DS:0040	05	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

1 Step 2 ProcStep 3 Retrieve 4 Help ON 5 BRK Menu 6 7 up 8 dn 9 le 10 ri

- The first seven instructions in your assembly code occupy a total of 23 bytes (0x17 in hexadecimal).

- The label `num1` serves as a symbolic reference for the memory address where the value 5 is stored. During assembly, this label is replaced with its actual address, which is `0x0117` in this case.
- Since the program starts at memory address `0x0100`, `num1` resides at an offset of 23 bytes (`0x17`) from the program's base. This translates to an absolute memory address of `0x0117`.
- 

```

3
4 00000000 A1[1700]      mov ax, [num1]
5 00000003 8B1E[1900]    mov bx, [num1 + 2]    ; notice how we can do arithmetic here
6 00000007 01D8        add ax, bx          ; also, why +2 and not +1?
7 00000009 8B1E[1B00]    mov bx, [num1 + 4]

```

0100	A11701	MOV	AX, [0117]	DS:0117	05 00 0A 00 0F 00 00 00
0103	8B1E1901	MOV	BX, [0119]	DS:011F	46 F6 00 00 8B 46 F6 D1
0107	01D8	ADD	AX, BX	DS:0127	E0 D1 E0 C5 5E D8 01 C3
				DS:012F	8B 07 8B 57 02 85 D2 75
				DS:0137	04 85 C0 74 1C C7 46 DC

- **1. Instruction Change:**
  - The original instruction (`B80500`) directly loaded a value into the register.
  - The assembler replaced it with `A11701` to load the value from the memory location labeled `num1`.
- **2. Why the Change?**
  - `num1` is a variable, and its actual address depends on the program's location in memory.
  - Using the memory address ensures the correct value is loaded, regardless of where the program is loaded.
- **3. Address Adjustment:**
  - The number 117 in the instruction might seem wrong because the program starts at 100.

- This is because the assembler adds the program's base address (100) to the offset (17) to get the absolute memory address (117).

```

14
15 num1: dw 5
16 num2: dw 10
17 num3: dw 15
18 num4: dw 0
19

```

File: c02-02.asm and c02-03.asm

We used dw(word/2bytes/16 bits) because ax is a 16 bits register and it stores 16 bits.

```

12
13 num1: dw 5
14 dw 10
15 dw 15
16 dw 0

```

The labels num2, num3, and num4 are removed and the data there will be accessed with reference to num1.

as we used dw(word/2bytes/16 bits) because ax is a 16 bits register and it stores 16 bits so to access the next num we use num1+2

which will be 0117+ 2 = 0119. and num1 + 4 to access next one and so on.

```

5 mov bx, [num1 + 2]
6 add ax, bx

```

```

mov bx, [num1 + 4]
add ax, bx
mov [num1 + 6], ax ; store sum at num1+6
mov ax, 0x4c00
int 0x21

num1: dw 5
dw 10
dw 15
dw 0

```

```
3 num1:  dw  5, 10, 15, 0
```

we don't need different variables to store multiple values, we can save space and declare all data on a single line separated by commas.

```
13 00000017 0500
14 00000019 0A00
15 0000001B 0F00
16 0000001D 0000
```

c02-02.lst

```
num1:  dw  5
      dw 10
      dw 15
```

```
3 00000017 05000A000F000000
```

```
num1:  dw  5, 10, 15, 0
```

c02-03.lst

see the difference here the operands are stored in different variables (there addresses are given).

File : c02-04.asm

```
mov  ax, [num1]
mov  [num1 + 6], ax    ; add this value to result
```

Storing the value of ax in result.

```
add  [num1 + 6], ax

mov  ax, [num1 + 4]
add  [num1+6], ax

mov  ax, 0x4c00
int  0x21
```

CMD >			
0005			
0100	A11901	MOV	AX,[0119]
0103	A31F01	MOV	[011F],AX
0106	A11B01	MOV	AX,[011B]
0109	01061F01	ADD	[011F],AX
010D	A11D01	MOV	AX,[011D]
0110	01061F01	ADD	[011F],AX
0114	B8004C	MOV	AX,4C00
0117	CD21	INT	21

The opcode of add is changed because the destination is now a memory location instead of a register.

**File : c02-05.asm**

```
; a program to add three numbers using byte variables
[org 0x0100]
```

```
mov ax, [num1]
```

```
mov    bx, [num1+1]
add    ax, bx
```

```
mov    bx, [num1+2]
add    ax, bx
```

```
mov    [num1+3], ax
```

```
mov ax, 0x4c00
int 0x21
```

```
num1: db 5, 10, 15, 0
```

```
; something's wrong with this code.  
; let's figure out what that is!
```

DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, Program: AFD - X

AX 0A05 SI 0000 CS 19F5 IP 0103 Stack +0 0000 Flags 7200  
 BX 0000 DI 0000 DS 19F5 +2 20CD  
 CX 001B BP 0000 ES 19F5 HS 19F5 +4 9FFF OF DF IF SF ZF AF PF CF  
 DX 0000 SP FFFE SS 19F5 FS 19F5 +6 EA00 0 0 1 0 0 0 0 0

CMD > DS:0117

0100 A11701 MOV AX, [0117]  
 0103 8B1E1801 MOV BX, [0118]  
 0107 01D8 ADD AX, BX  
 0109 8B1E1901 MOV BX, [0119]  
 010D 01D8 ADD AX, BX  
 010F A31A01 MOV [011A], AX  
 0112 B8004C MOV AX, 4C00  
 0115 CD21 INT 21  
 0117 050A0F ADD AX, 0F0A

DS:0117 05 0A 0F 00 89 46 E6 C7  
 DS:011F 48 F8 00 00 8B 48 F8 D1  
 DS:0127 E0 D1 E0 C5 5E D8 01 C3  
 DS:012F 8B 07 8B 57 02 85 D2 75  
 DS:0137 04 85 C0 74 1C C7 46 DC  
 DS:013F 00 00 8E 5E FC 83 7D 0E  
 DS:0147 00 74 09 8B 46 F2 48 3B  
 DS:014F 46 F6 7E 08 B8 01 00 EB  
 DS:0157 05 E9 42 01 31 C0 89 46  
 DS:015F E2 8B 46 F6 D1 E0 D1 E0

2 0 1 2 3 4 5 6 7 8 9 A B C D E F  
 DS:0000 CD 20 FF 9F 00 EA F0 FE AD DE 1B 05 C5 06 00 00  
 DS:0010 18 01 10 01 18 01 92 01 01 01 01 00 02 FF FF FF  
 DS:0020 FF FF FF FF FF FF FF FF FF FF FF FF FF 19 C0 11

= f.Ω■ i | . + . . .  
 . . . . . ff . . . . . δ L

**Problem:** The code mixes byte variables (defined with `db`) and a 16-bit register (`ax`). This leads to incorrect addition because `ax` stores extra garbage bits when loading single bytes.

**Solution:**

1. Use `al` (lower 8 bits of `ax`) with `db`:

- Load each byte value into `al`: `mov al, [num1]`, `mov al, [num1+1]`, `mov al, [num1+2]`
- Perform additions using `al`: `add al, [num1+1]`, `add al, [num1+2]`
- Store the result (in `al`) back to memory: `mov [num1+3], al`

2. Use `ax` with `dw` :

- Change `num1` to `dw` (defines word variables): `num1: dw 5, 10, 15, 0`

file : `c02-06.asm`

```
1 ; a program to add three numbers using byte variables
2 [org 0x0100]
3     ; mov ax, 0x8787
4     ; xor ax, ax                ; We need to make sure AX is empty
5
6     mov ah, [num1]              ; Intel Software Developer Manual - F
7
8     mov bl, [num1+1]
9     add ah, bh
10
11    mov bh, [num1+2]
12    add ah, bh
13
14    mov [num1+3], ah
15
16    mov ax, 0x4c00
17    int 0x21
18
19 num1: db 5, 10, 15, 0
```

`db` signifies that each element in `num1` is a single byte (8 bits) of data.

The instructions `mov al, [num1]`, `mov al, [num1+1]`, and `mov al, [num1+2]` load the byte values from memory locations `num1`, `num1+1`, and `num1+2`, respectively, into the lower 8 bits (AL) of the `ax` register.

Adding 1 to the base address (num1) moves to the next byte location (num1+1).

The instruction `mov [num1+3], al` stores the final result (the sum) from AL back into the memory location num1+3.

File c02-07.asm

```
1 ; a program to add three numbers using byte variables
2 [org 0x0100]
3
4 ; for (int c = 3    c > 0    c--) {
5 ;   result += data[c];
6 ;}
7
8
9
10 ; initialize stuff
11 mov ax, 0          ; reset the accumulator
12 mov cx, 3          ; set the iterator count
13 mov bx, num1        ; set the base
14
15 outloop:
16     add ax, [bx]
17     add bx, 2
18
19     sub cx, 1
20     jnz outloop
21
22
23     mov [result], ax
24
25     mov ax, 0x4c00
26     int 0x21
27
28
29 ; Intel Software Developer Manual - EFLAGS and Instructions (Page 435)
30
31 num1: dw 5, 10, 15
32 result: dw 0
```

The loop (outloop) continues as long as the counter (cx) is greater than 0.

- The `jnz` (jump if not zero) instruction checks the **Zero Flag (ZF)** after the `sub cx, 1` instruction.
- If cx becomes **zero** after the subtraction, the Zero Flag is set (ZF = 1).
- If cx is still **positive** (greater than zero), the Zero Flag remains **cleared** (ZF = 0).



