Computer Organization and Assembly Language (Notes)

Prepared by

Umer Naeem (umer.naeem@ucp.edu.pk)

Arif Mustafa (arif.mustafa@ucp.edu.pk)

Types of Registers

The registers are grouped into three categories:-

1. **<u>General Purpose registers</u>**
   - 1.1. *Data registers*
     - 1.1.1. ***AX*** is the primary accumulator.
     - 1.1.2. ***BX*** is known as the base register.
     - 1.1.3. ***CX*** is known as the count register.
     - 1.1.4. ***DX*** is known as the data register.
   - 1.2. *Pointer registers*
     - 1.2.1. Instruction Pointer ***IP***
     - 1.2.2. Stack Pointer ***SP***
     - 1.2.3. Base Pointer ***BP***
   - 1.3. *Index registers*
     - 1.3.1. Source Index ***SI***
     - 1.3.2. Destination Index ***DI***
2. **<u>Control registers</u>**
   - 2.1. Instruction Pointer and Flag register
3. **<u>Segment registers</u>**
   - 3.1. Code Segment ***CS***
   - 3.2. Data Segment ***DS***
   - 3.3. Stack Segment ***SS***
   - 3.4. Extra Segment ***ES***

Types of variables

Type	No. of bits	Example declaration:
Byte	8	Num1: db 43
Word=> 2 bytes	16	Num2: dw 0xABFF
double word=> 2 words	32	Num3: dd 0xABCDEF56

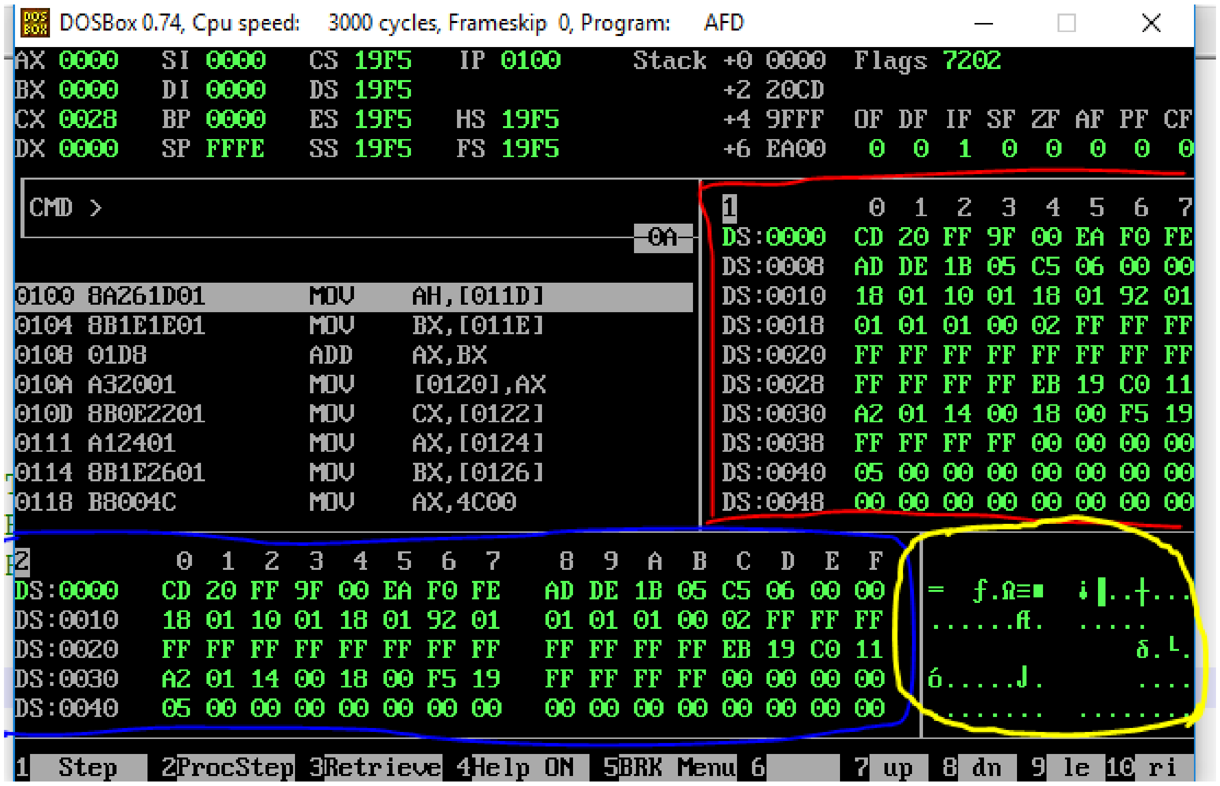
Note: size of both operands must be same for any type of instruction.

For example:
Mov ax,dh ;is wrong because destination is 2 bytes and source is 1 byte.

Viewing memory in DOSBOX

Areas highlighted in red(memory 1) “m1” and blue (memory 2) “m2” are showing the memory contents. *Note:* Two copies of the same memory is displayed in the given windows.

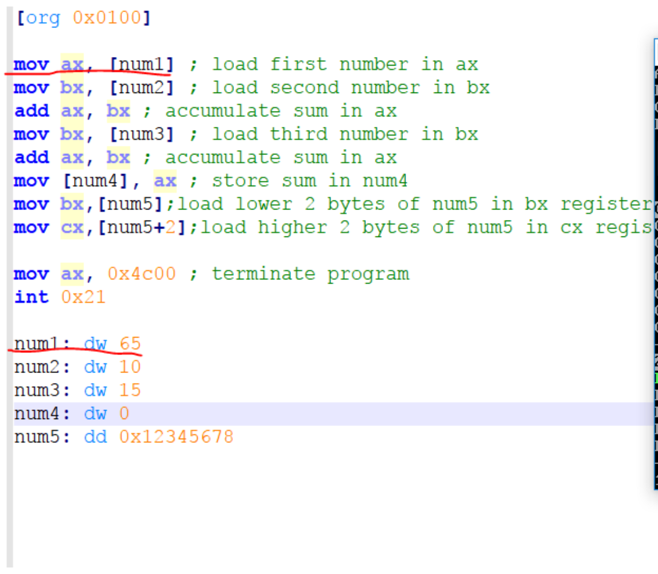
Area highlighted with yellow is showing the ascii values of the contents displayed in the memory m2.



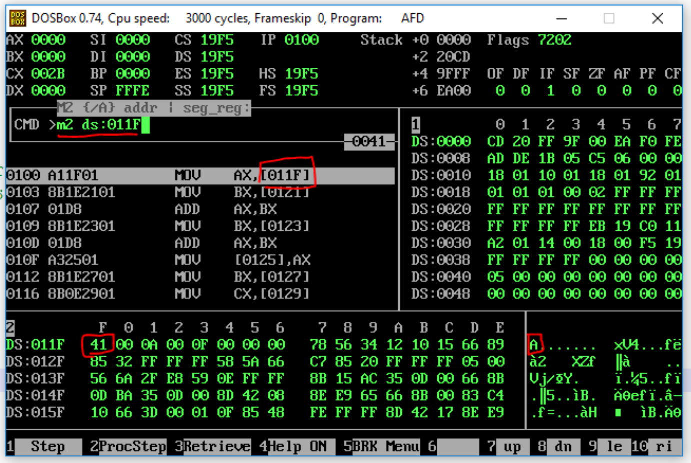
```
DOSBox 0.74, Cpu speed:   3000 cycles, Frameskip  0, Program:   AFD
AX 0000   SI 0000   CS 19F5   IP 0100     Stack +0 0000  Flags 7202
BX 0000   DI 0000   DS 19F5                     +2 20CD
CX 0028   BP 0000   ES 19F5   HS 19F5           +4 9FFF  OF DF IF SF ZF AF PF CF
DX 0000   SP FFFE   SS 19F5   FS 19F5           +6 EA00   0  0  1  0  0  0  0  0

 CMD >                                         1         0  1  2  3  4  5  6  7
                                          0A   DS:0000  CD 20 FF 9F 00 EA F0 FE
                                               DS:0008  AD DE 1B 05 C5 06 00 00
0100 8A261D01      MOV    AH,[011D]            DS:0010  18 01 10 01 18 01 92 01
0104 8B1E1E01      MOV    BX,[011E]            DS:0018  01 01 01 00 02 FF FF FF
0108 01D8          ADD    AX,BX                DS:0020  FF FF FF FF FF FF FF FF
010A A32001        MOV    [0120],AX            DS:0028  FF FF FF FF EB 19 C0 11
010D 8B0E2201      MOV    CX,[0122]            DS:0030  A2 01 14 00 18 00 F5 19
0111 A12401        MOV    AX,[0124]            DS:0038  FF FF FF FF 00 00 00 00
0114 8B1E2601      MOV    BX,[0126]            DS:0040  05 00 00 00 00 00 00 00
0118 B8004C        MOV    AX,4C00              DS:0048  00 00 00 00 00 00 00 00

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   =  ƒ.Ω≡■  ¡▌..┼...
DS:0010   18 01 10 01 18 01 92 01   01 01 01 00 02 FF FF FF   ......Æ.  .....
DS:0020   FF FF FF FF FF FF FF FF   FF FF FF FF EB 19 C0 11                δ.└.
DS:0030   A2 01 14 00 18 00 F5 19   FF FF FF FF 00 00 00 00   ó.....J.      ....
DS:0040   05 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00    .......  ........

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

Viewing sample variable in memory.

- To view memory from window m2 run the command “m2 ds:***Addressofvariable***”
  example: m2 ds:011F
- A variable with name “num1” is initialized at memory location 11F with value 65 decimal.
  41 hex = 65 decimal is the ascii of “A”.



```
[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 [num4], ax ; store sum in num4
mov bx,[num5];load lower 2 bytes of num5 in bx register
mov cx,[num5+2];load higher 2 bytes of num5 in cx regis

mov ax, 0x4c00 ; terminate program
int 0x21

num1: dw 65
num2: dw 10
num3: dw 15
num4: dw 0
num5: dd 0x12345678
```



```
DOSBox 0.74, Cpu speed:   3000 cycles, Frameskip  0, Program:   AFD
AX 0000   SI 0000   CS 19F5   IP 0100      Stack +0 0000  Flags 7202
BX 0000   DI 0000   DS 19F5                      +2 20CD
CX 002B   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
       M2 {/A} addr | seg_reg:
 CMD >m2 ds:011F                                        1          0  1  2  3  4  5  6  7
                                             -0041-    DS:0000  CD 20 FF 9F 00 EA F0 FE
                                                       DS:0008  AD DE 1B 05 C5 06 00 00
0100 A11F01            MOV     AX,[011F]               DS:0010  18 01 10 01 18 01 92 01
0103 8B1E2101          MOV     BX,[0121]               DS:0018  01 01 01 00 02 FF FF FF
0107 01D8              ADD     AX,BX                   DS:0020  FF FF FF FF FF FF FF FF
0109 8B1E2301          MOV     BX,[0123]               DS:0028  FF FF FF FF EB 19 C0 11
010D 01D8              ADD     AX,BX                   DS:0030  A2 01 14 00 18 00 F5 19
010F A32501            MOV     [0125],AX               DS:0038  FF FF FF FF 00 00 00 00
0112 8B1E2701          MOV     BX,[0127]               DS:0040  05 00 00 00 00 00 00 00
0116 8B0E2901          MOV     CX,[0129]               DS:0048  00 00 00 00 00 00 00 00

2          F  0  1  2  3  4  5  6    7  8  9  A  B  C  D  E
DS:011F   41 00 0A 00 0F 00 00 00   78 56 34 12 10 15 66 89   A.......  xV4...fë
DS:012F   85 32 FF FF FF 58 5A 66   C7 85 20 FF FF FF 05 00   à2   XZf  ‖à     ..
DS:013F   56 6A 2F E8 59 0E FF FF   8B 15 AC 35 0D 00 66 8B   Vj/ΦY.    ï.¼5..fï
DS:014F   0D BA 35 0D 00 8D 42 08   8E E9 65 66 8B 00 83 C4   .‖5..ìB.  ÄΘefï.â─
DS:015F   10 66 3D 00 01 0F 85 48   FE FF FF 8D 42 17 8E E9   .f=...àH  ■  ìB.ÄΘ

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

A sample program to add three numbers using memory variables

```
[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 [num4], ax ; store sum in num4


mov ax, 0x4c00 ; terminate program
int 0x21

num1: dw 65
num2: dw 10
num3: dw 15
num4: dw 0
```

A sample program to read double word variable

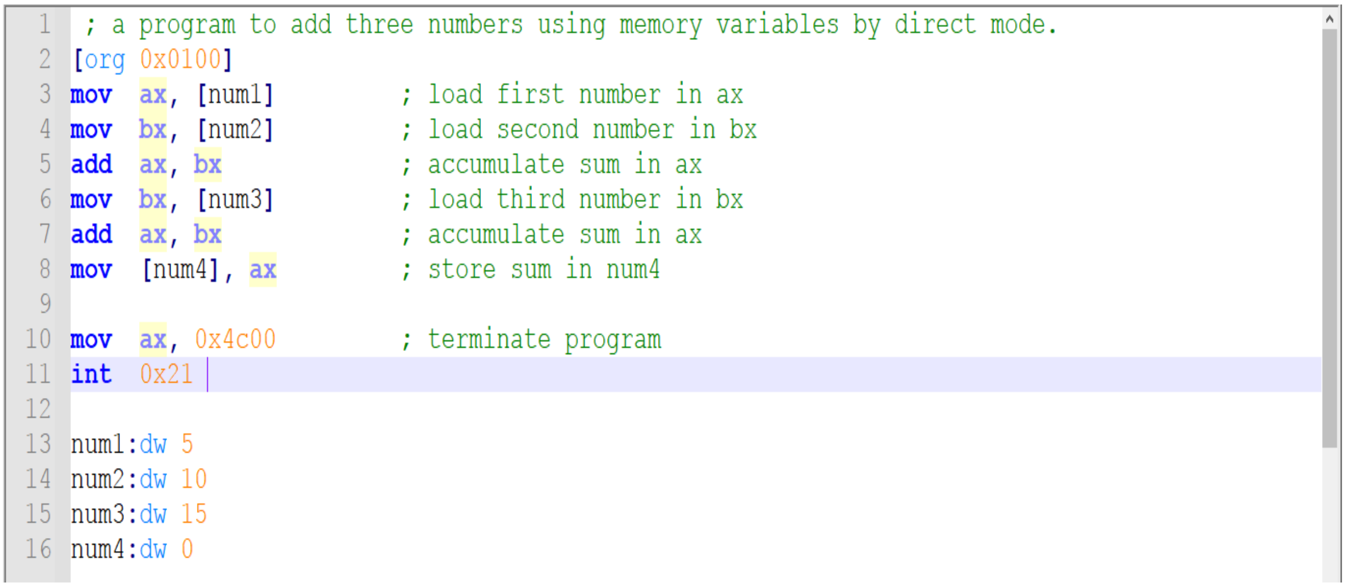
```
mov bx,[num5];load lower 2 bytes of num5 in bx register.
mov cx,[num5+2];load higher 2 bytes of num5 in cx register.
num5: dd 0x12345678
mov ax, 0x4c00 ; terminate program
int 0x21
```

Types of Addressing Modes

Direct A fixed offset is given in brackets and the memory at that offset is accessed. For example "mov [1234], ax" stores the contents of the AX registers in two bytes starting at address 1234 in the current data segment. The instruction "mov [1234], al" stores the contents of the AL register in the byte at offset 1234.	• Mov ax,[num1] ;reading • Mov [num2],ax ;writing
Based Register Indirect A base register is used in brackets and the actual address accessed depends on the value contained in that register. For example "mov [bx], ax" moves the two byte contents of the AX register to the address contained in the BX register in the current data segment. The instruction "mov [bp], al" moves the one byte content of the AL register to the address contained in the BP register in the current stack segment.	• Mov bx,var • Mov cx,[bx] • Mov [bx],ax
Indexed Register Indirect An index register is used in brackets and the actual address accessed depends on the value contained in that register. For example "mov [si], ax" moves the contents of the AX register to the word starting at address contained in SI in the current data segment. The instruction "mov [di], ax" moves the word contained in AX to the offset stored in DI in the current data segment.	• Mov si,var1 • Mov di,var2 • Mov [si], ax • Mov [di],bx • Mov cx,[si] • Mov dx,[di]

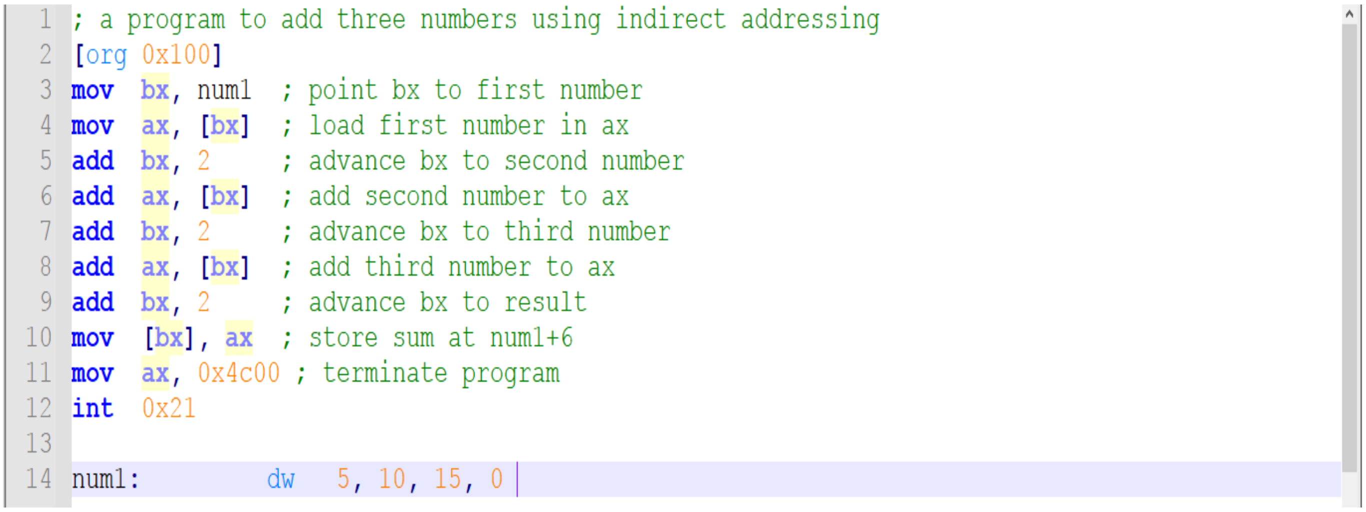
Based Register Indirect + Offset A base register is used with a constant offset in this addressing mode. The value contained in the base register is added with the constant offset to get the effective address. For example “mov [bx+300], ax” stores the word contained in AX at the offset attained by adding 300 to BX in the current data segment. The instruction “mov [bp+300], ax” stores the word in AX to the offset attained by adding 300 to BP in the current stack segment.	• mov [bx+3], ax • mov cl,[bp+5]

A program to add three variables using Direct addressing



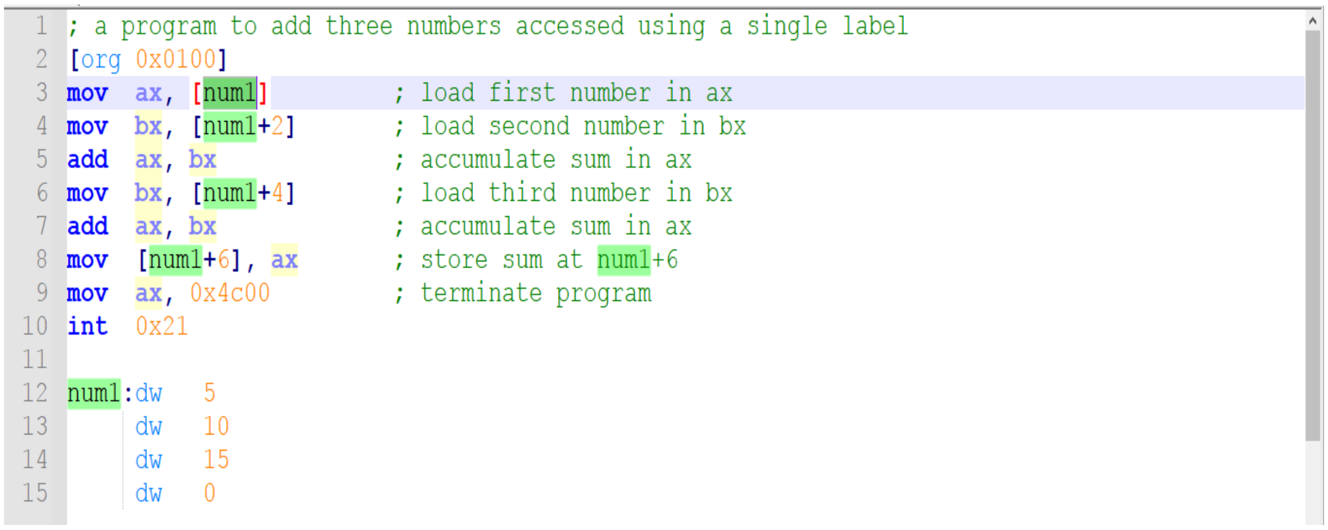
```
 ; a program to add three numbers using memory variables by direct mode.
[org 0x0100]
mov  ax, [num1]         ; load first number in ax
mov  bx, [num2]         ; load second number in bx
add  ax, bx             ; accumulate sum in ax
mov  bx, [num3]         ; load third number in bx
add  ax, bx             ; accumulate sum in ax
mov  [num4], ax         ; store sum in num4

mov  ax, 0x4c00         ; terminate program
int  0x21

num1:dw 5
num2:dw 10
num3:dw 15
num4:dw 0
```

A program to add three variables using InDirect addressing



```
; a program to add three numbers using indirect addressing
[org 0x100]
mov  bx, num1  ; point bx to first number
mov  ax, [bx]  ; load first number in ax
add  bx, 2     ; advance bx to second number
add  ax, [bx]  ; add second number to ax
add  bx, 2     ; advance bx to third number
add  ax, [bx]  ; add third number to ax
add  bx, 2     ; advance bx to result
mov  [bx], ax  ; store sum at num1+6
mov  ax, 0x4c00 ; terminate program
int  0x21

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

A program to add three numbers using only one location.



```
; a program to add three numbers accessed using a single label
[org 0x0100]
mov  ax, [num1]          ; load first number in ax
mov  bx, [num1+2]        ; load second number in bx
add  ax, bx              ; accumulate sum in ax
mov  bx, [num1+4]        ; load third number in bx
add  ax, bx              ; accumulate sum in ax
mov  [num1+6], ax        ; store sum at num1+6
mov  ax, 0x4c00          ; terminate program
int  0x21

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

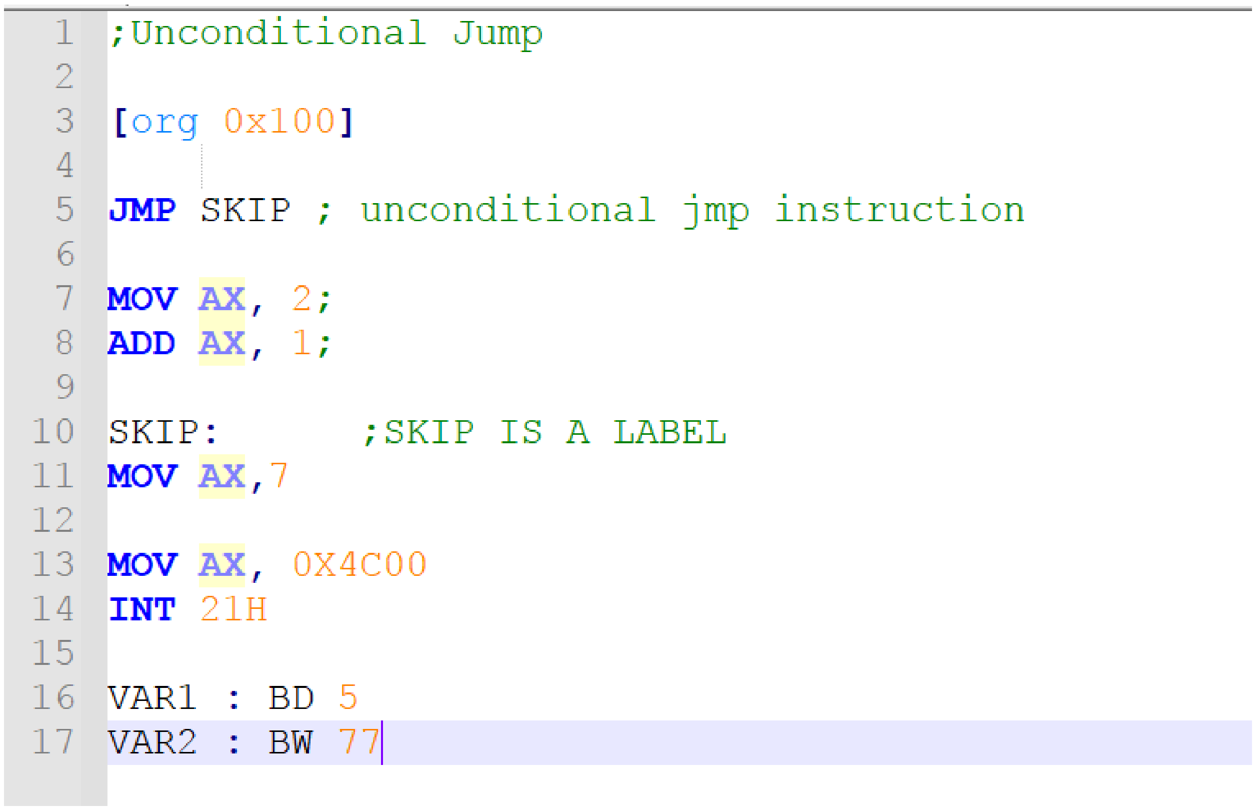
When using variables of different size

```
[org 0x100]
mov ax,0
mov bx,0
mov cx,0
mov dx,0
; add two variables their sum should be 5163 or 142B
;option1
Mov al,[var1]
Mov bl,[var2]
Add al,bl
;option2
Mov al,[var1]
Mov bx,[var2]
Add al,bx       ;this will show error because of size mismatch change al to ax then run
again
;option3
Mov ax,[var1] ;why ax is not showing the correct value of var1
Mov bx,[var2]
Add ax,bx
;option4
Mov al,[var1]
Mov ah,0        ;already 0 in ah
Mov bx,[var2]
Add ax,bx
mov ax,0x4c00
int 21h
var1: db 60     ;0x3C
var2: dw 5103  ;0x13EF
```

JUMP INSTRUCTIONS

Two main types of jump instructions

a) Unconditional jump:



```
;Unconditional Jump

[org 0x100]

JMP SKIP ; unconditional jmp instruction

MOV AX, 2;
ADD AX, 1;

SKIP:       ;SKIP IS A LABEL
MOV AX,7

MOV AX, 0X4C00
INT 21H

VAR1 : BD 5
VAR2 : BW 77
```

b) Conditional jump:

To start executing instructions based on some condition.

Some conditional jumps are as follows:

JC	Jump if carry flag set
JNC	Jump if not carry
JZ	Jump if zero flag set
JE	Jump if zero flag set
JNZ	Jump if zero flag not set
JNE	Jump if zero flag not set

JS	Jump if sign flag is set
JNS	Jump if sign flag not set
JP	Jump if parity flag is set
JNP	Jump if not parity
JO	Jump if overflow
JNO	Jump if not overflow

1. Conditional jumps after *signed operand comparison*

JG	Jump if greater
JNG	Jump if not greater
JGE	Jump if greater or equal
JNGE	Jump if not greater or equal
JL	Jump if less
JNL	jump if not less
JLE	Jump if less or equal
JNLE	jump if not less or equal

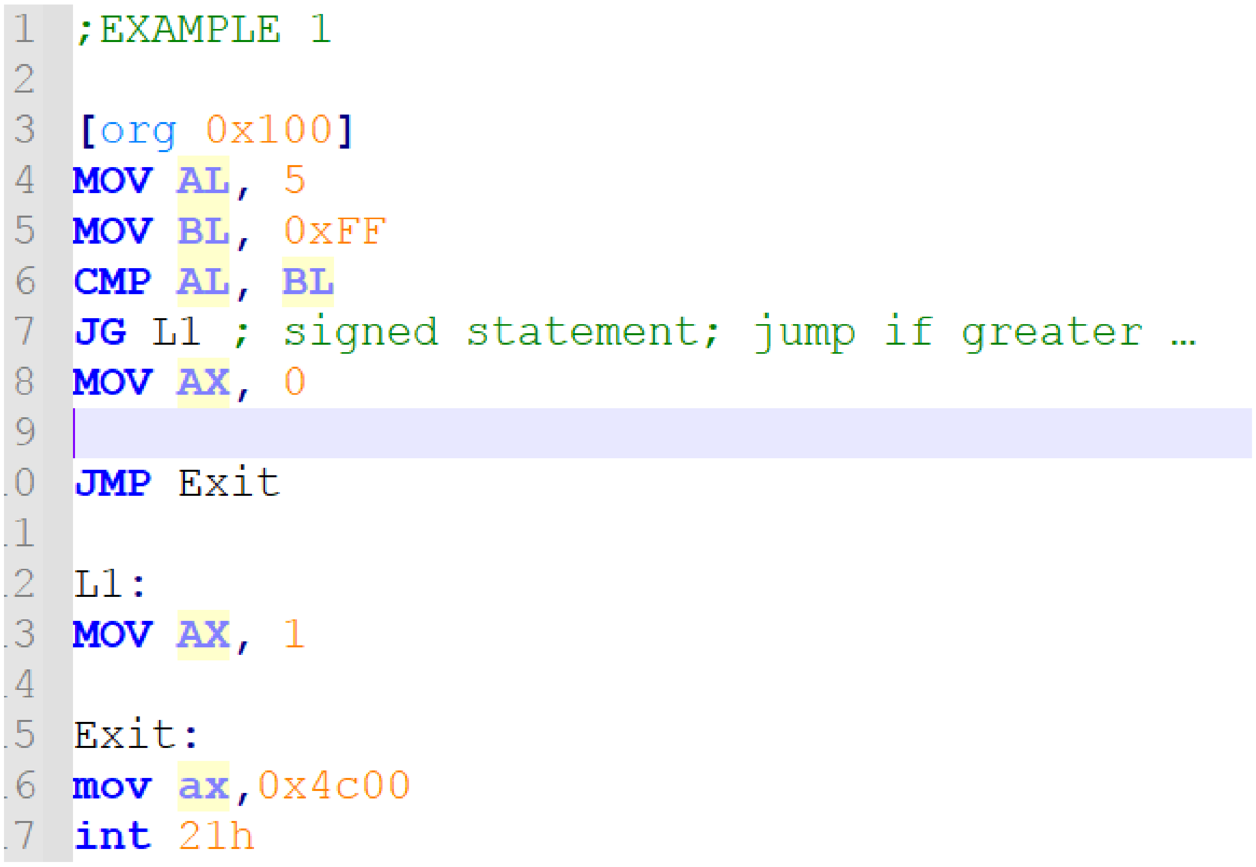
2. Conditional jumps after *unsigned operand comparison*

JA	Jump if above
JNA	Jump if not above
JAE	Jump if above or equal
JNAE	Jump if not above or equal
JB	Jump if below
JNB	Jump if not below
JBE	Jump if below or equal
JNBE	jump if not below or equal

- **Compare Instruction**
  cmp operand1,operand2
  It subtracts operand2 from operand1 and updates the flags only ***without updating the value of the operands.***

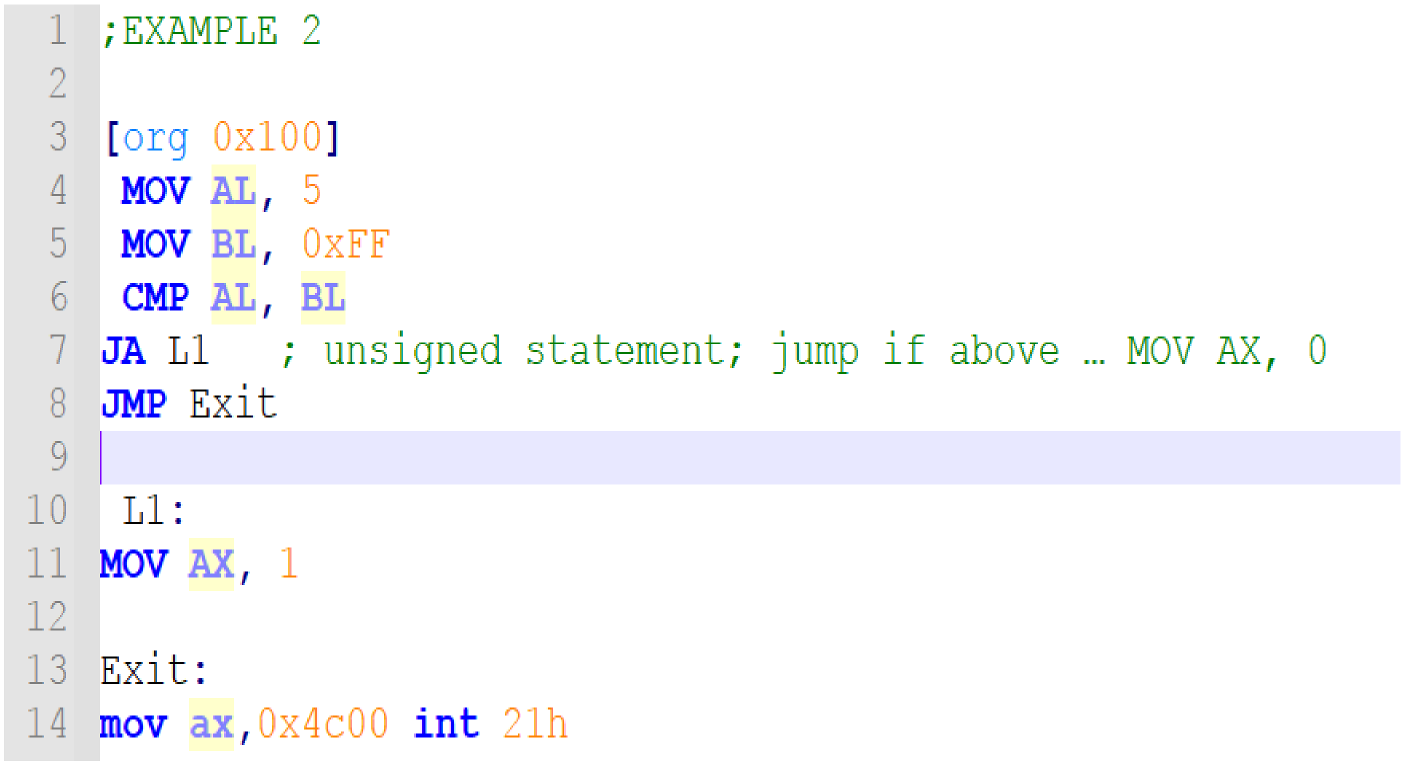
Example: Signed number comparison

In this example it compares two numbers and stores **1** in ax if *al* is greater than *bl* else **0**.



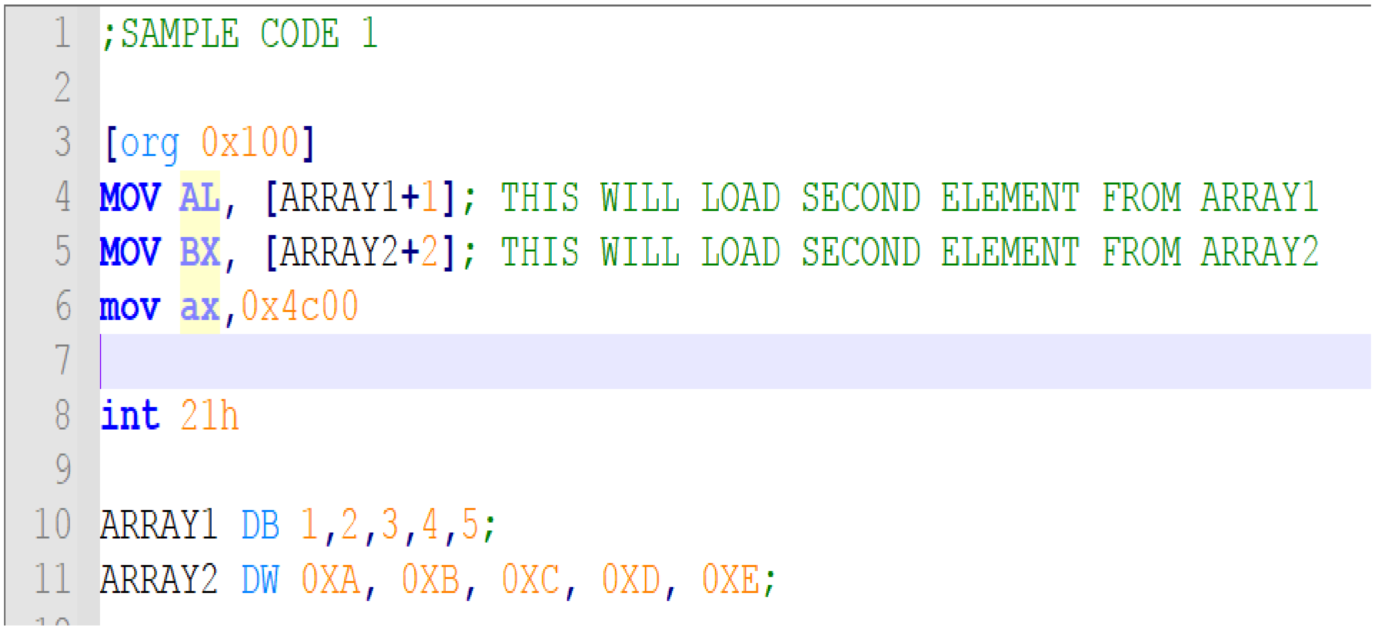
```
;EXAMPLE 1

[org 0x100]
MOV AL, 5
MOV BL, 0xFF
CMP AL, BL
JG L1 ; signed statement; jump if greater …
MOV AX, 0

JMP Exit

L1:
MOV AX, 1

Exit:
mov ax,0x4c00
int 21h
```

Example: Unsigned number comparison



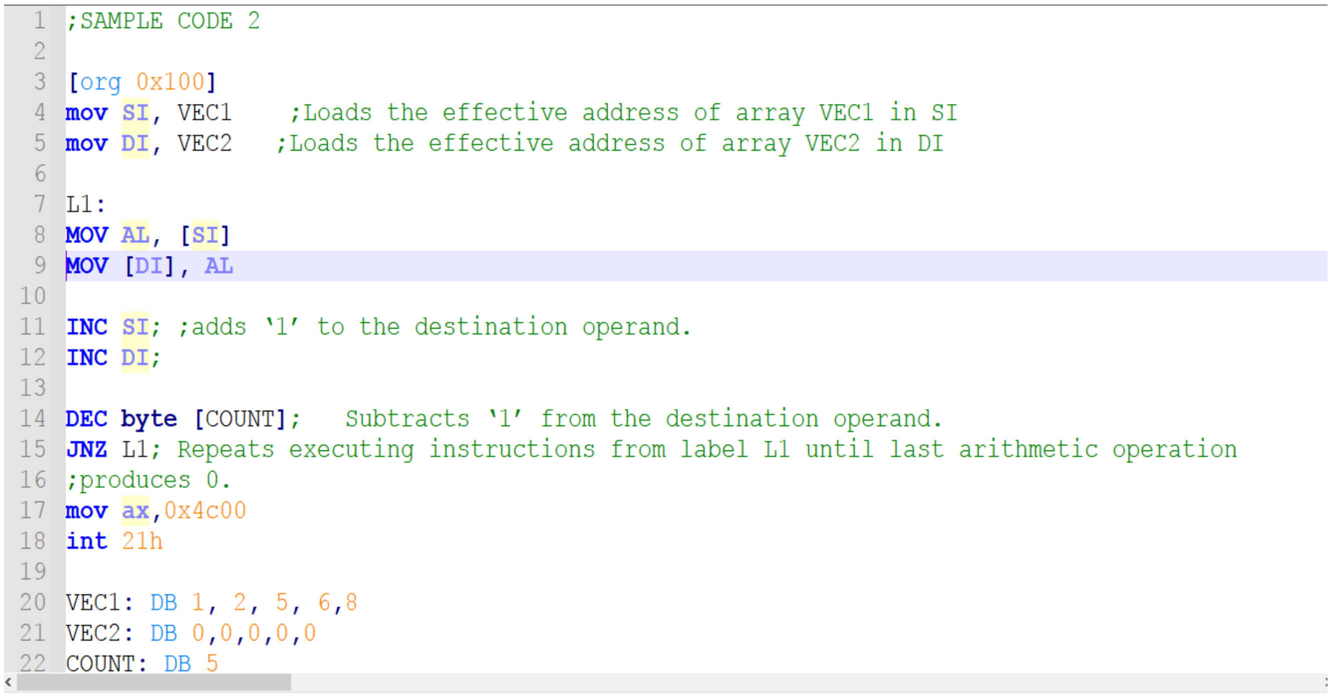
```
;EXAMPLE 2

[org 0x100]
 MOV AL, 5
 MOV BL, 0xFF
 CMP AL, BL
JA L1   ; unsigned statement; jump if above … MOV AX, 0
JMP Exit

 L1:
MOV AX, 1

Exit:
mov ax,0x4c00 int 21h
```

TO ACCESS NEXT ELEMENTS WITHIN AN ARRAY ADD OFFSETS TO ARRAY NAME DEPENDING UPON ARRAY SIZE TYPE.



```
;SAMPLE CODE 1

[org 0x100]
MOV AL, [ARRAY1+1]; THIS WILL LOAD SECOND ELEMENT FROM ARRAY1
MOV BX, [ARRAY2+2]; THIS WILL LOAD SECOND ELEMENT FROM ARRAY2
mov ax,0x4c00

int 21h

ARRAY1 DB 1,2,3,4,5;
ARRAY2 DW 0XA, 0XB, 0XC, 0XD, 0XE;
```

Access data from an array (Indexed register Indirect mode)

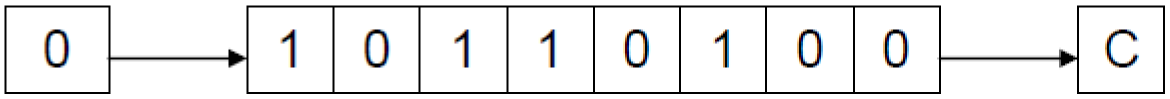


```
;SAMPLE CODE 2

[org 0x100]
mov SI, VEC1    ;Loads the effective address of array VEC1 in SI
mov DI, VEC2   ;Loads the effective address of array VEC2 in DI

L1:
MOV AL, [SI]
MOV [DI], AL

INC SI; ;adds ‘1’ to the destination operand.
INC DI;

DEC byte [COUNT];   Subtracts ‘1’ from the destination operand.
JNZ L1; Repeats executing instructions from label L1 until last arithmetic operation
;produces 0.
mov ax,0x4c00
int 21h

VEC1: DB 1, 2, 5, 6,8
VEC2: DB 0,0,0,0,0
COUNT: DB 5
```

Shifting and Rotations variations

Shift Logical Right (SHR)

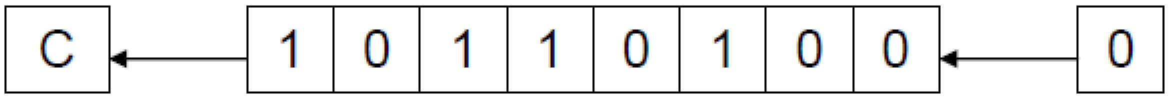
The shift logical right operation inserts a zero from the left and moves every bit one position to the right and copies the rightmost bit in the carry flag. Imagine that there is a pipe filled to capacity with eight balls. The pipe is open from both ends and there is a basket at the right end to hold anything dropping from there. The operation of shift logical right is to force a white ball from the left end. The operation is depicted in the following illustration.



White balls represent zero bits while black balls represent one bits. Sixteen bit shifting is done the same way with a pipe of double capacity.

Shift Logical Left (SHL) / Shift Arithmetic Left (SAL)

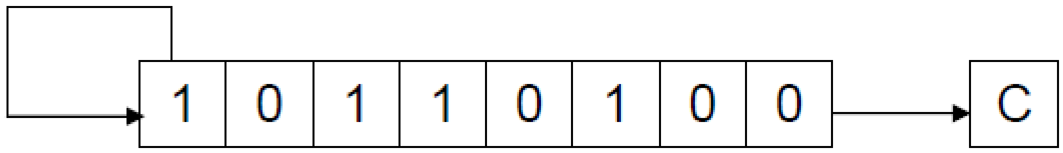
The shift logical left operation is the exact opposite of shift logical right. In this operation the zero bit is inserted from the right and every bit moves one position to its left with the most significant bit dropping into the carry flag. Shift arithmetic left is just another name for shift logical left. The operation is again exemplified with the following illustration of ball and pipes.



Shift Arithmetic Right (SAR)

A signed number holds the sign in its most significant bit. If this bit was one a logical right shifting will change the sign of this number because of insertion of a zero from the left. The sign of a signed number should not change because of shifting.

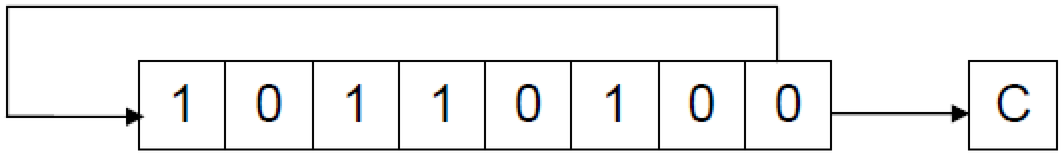
The operation of shift arithmetic right is therefore to shift every bit one place to the right with a copy of the most significant bit left at the most significant place. The bit dropped from the right is caught in the carry basket. The sign bit is retained in this operation. The operation is further illustrated below.



The left shifting operation is basically multiplication by 2 while the right shifting operation is division by two. However for signed numbers division by two can be accomplished by using shift arithmetic right and not shift logical right. The left shift operation is equivalent to multiplication except when an important bit is dropped from the left. The overflow flag will signal this condition if it occurs and can be checked with JO. For division by 2 of a signed number logical right shifting will give a wrong answer for a negative number as the zero inserted from the left will change its sign. To retain the sign flag and still effectively divide by two the shift arithmetic right instruction must be used on signed numbers.

Rotate Right (ROR)

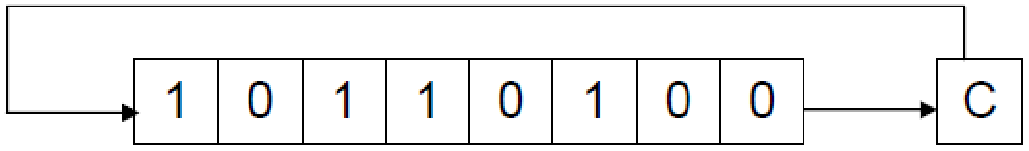
In the rotate right operation every bit moves one position to the right and the bit dropped from the right is inserted at the left. This bit is also copied into the carry flag. The operation can be understood by imagining that the pipe used for shifting has been molded such that both ends coincide. Now when the first ball is forced to move forward, every ball moves one step forward with the last ball entering the pipe from its other end occupying the first ball's old position. The carry basket takes a snapshot of this ball leaving one end of the pipe and entering from the other.



Rotate Through Carry Right (RCR)

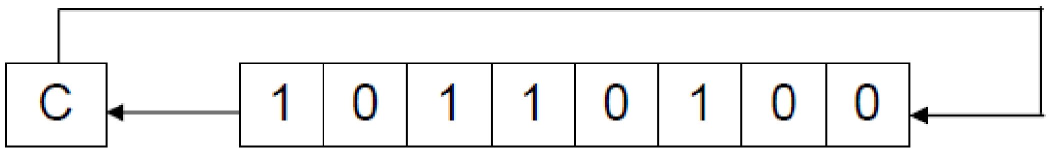
In the rotate through carry right instruction, the carry flag is inserted from the left, every bit moves one position to the right, and the right most bit is dropped in the carry flag. Effectively this is a nine bit or a seventeen bit rotation instead of the eight or sixteen bit rotation as in the case of simple rotations.

Imagine the circular molded pipe as used in the simple rotations but this time the carry position is part of the circle between the two ends of the pipe. Pushing the carry ball from the left causes every ball to move one step to its right and the right most bit occupying the carry place. The idea is further illustrated below.



Rotate Through Carry Left (RCL)

The exact opposite of rotate through carry right instruction is the rotate through carry left instruction. In its operation the carry flag is inserted from the right causing every bit to move one location to its left and the most significant bit occupying the carry flag. The concept is illustrated below in the same manner as in the last example.



Example 1:

Multiply the number by 4 using shift operator.

Let the number is 5.

```
mov al,5
shl al,2
```

Example 2:

Rotate right 3 times the value in register bx.

Let BX=0xEFCD

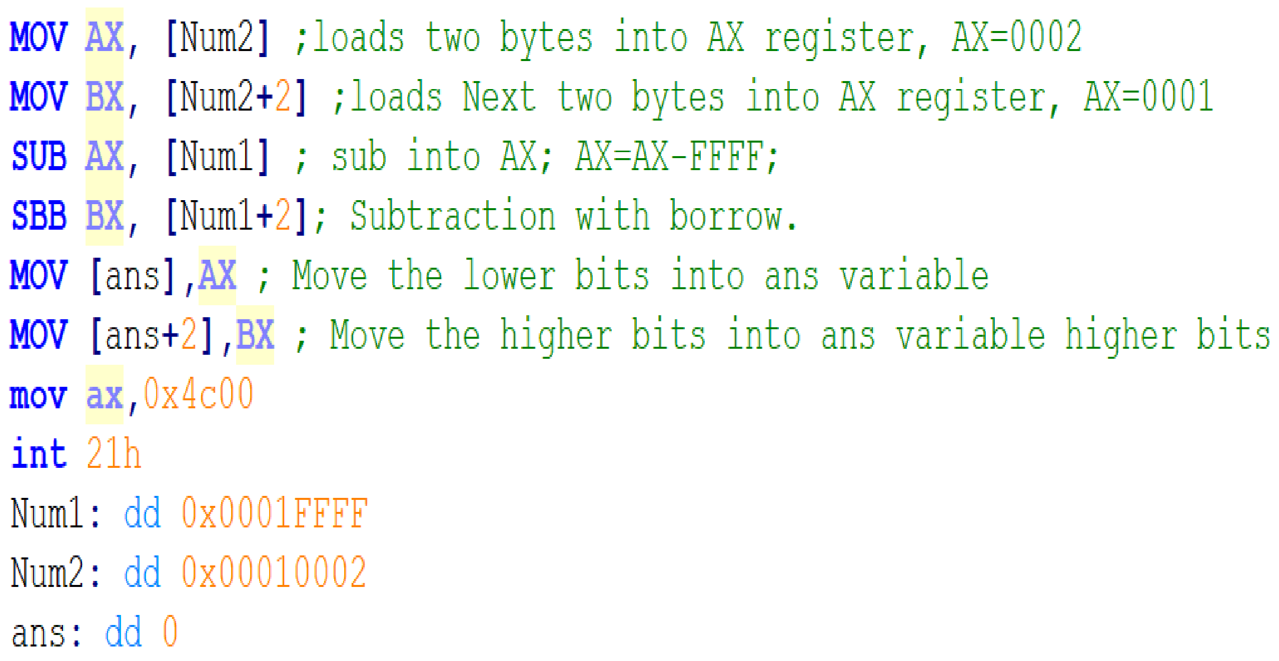
```
mov bx,0xEFCD
ror bx,3
```

Extended addition and subtraction

Example 3(Extended addition)

```
MOV AX, [Num1] ;loads two bytes into AX register, AX=FFFF
MOV BX, [Num1+2] ;loads Next two bytes into BX register, BX=0001
ADD AX, [Num2] ; adds into AX; AX=AX+0002;
ADC BX, [Num2+2]; Add with carry instruction
MOV [SUM],AX ; Move the lower bits into Sum variable
MOV [SUM+2],BX ; Move the higher bits into Sum variable higher bits
mov ax,0x4c00
int 21h
Num1: dd 0x0001FFFF
Num2: dd 0x00010002
SUM: dd 0
```

Example 4(Extended subtraction)



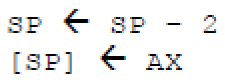
```
MOV AX, [Num2] ;loads two bytes into AX register, AX=0002
MOV BX, [Num2+2] ;loads Next two bytes into AX register, AX=0001
SUB AX, [Num1] ; sub into AX; AX=AX-FFFF;
SBB BX, [Num1+2]; Subtraction with borrow.
MOV [ans],AX ; Move the lower bits into ans variable
MOV [ans+2],BX ; Move the higher bits into ans variable higher bits
mov ax,0x4c00
int 21h
Num1: dd 0x0001FFFF
Num2: dd 0x00010002
ans: dd 0
```

Stack operations

Observe the values of SP, IP in each code after push, pop, call and ret instructions carefully.

PUSH

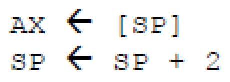
PUSH decrements SP (the stack pointer) by two and then transfers a word from the source operand to the top of stack now pointed to by SP. PUSH often is used to place parameters on the stack before calling a procedure; more generally, it is the basic means of storing temporary data on the stack. For example “push ax” will push the current value of the AX register on the stack. The operation of PUSH is shown below.



```
SP ← SP - 2
[SP] ← AX
```

POP

POP transfers the word at the current top of stack (pointed to by SP) to the destination operand and then increments SP by two to point to the new top of stack. POP can be used to move temporary variables from the stack to registers or memory. Observe that the operand of PUSH is called a source operand since the data is moving to the stack from the operand, while the operand of POP is called destination since data is moving from the stack to the operand. The operation of “pop ax” is shown below.



```
AX ← [SP]
SP ← SP + 2
```

CALL

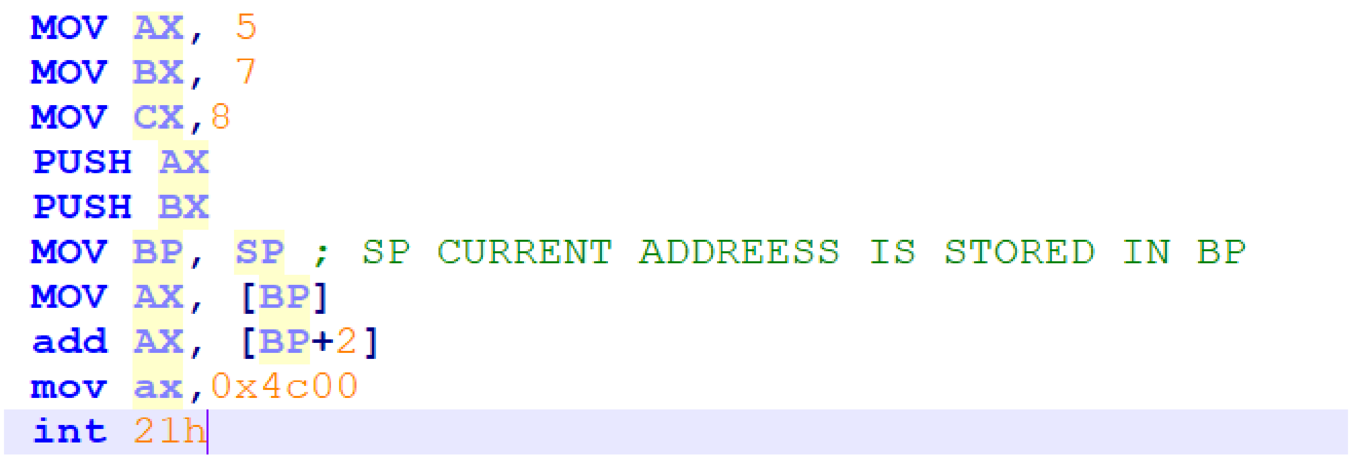
CALL activates an out-of-line procedure, saving information on the stack to permit a RET (return) instruction in the procedure to transfer control back to the instruction following the CALL. For an intra segment direct CALL, SP is decremented by two and IP is pushed onto the stack. The target procedure’s relative displacement from the CALL instruction is then added to the instruction pointer.

RET

RET (Return) transfers control from a procedure back to the instruction following the CALL that activated the procedure. RET pops the word at the top of the stack (pointed to by register SP) into the instruction pointer and increments SP by two. If RET is used the word at the top of the stack is popped into the IP register and SP is incremented by two. If an optional pop value has been specified, RET adds that value to SP. This feature may be used to discard parameters pushed onto the stack before the execution of the CALL instruction.

Stack Example:

ADD Two Numbers that are pushed in stack without POP



```
MOV AX, 5
MOV BX, 7
MOV CX,8
PUSH AX
PUSH BX
MOV BP, SP ; SP CURRENT ADDREESS IS STORED IN BP
MOV AX, [BP]
add AX, [BP+2]
mov ax,0x4c00
int 21h
```

Implementation of subroutine

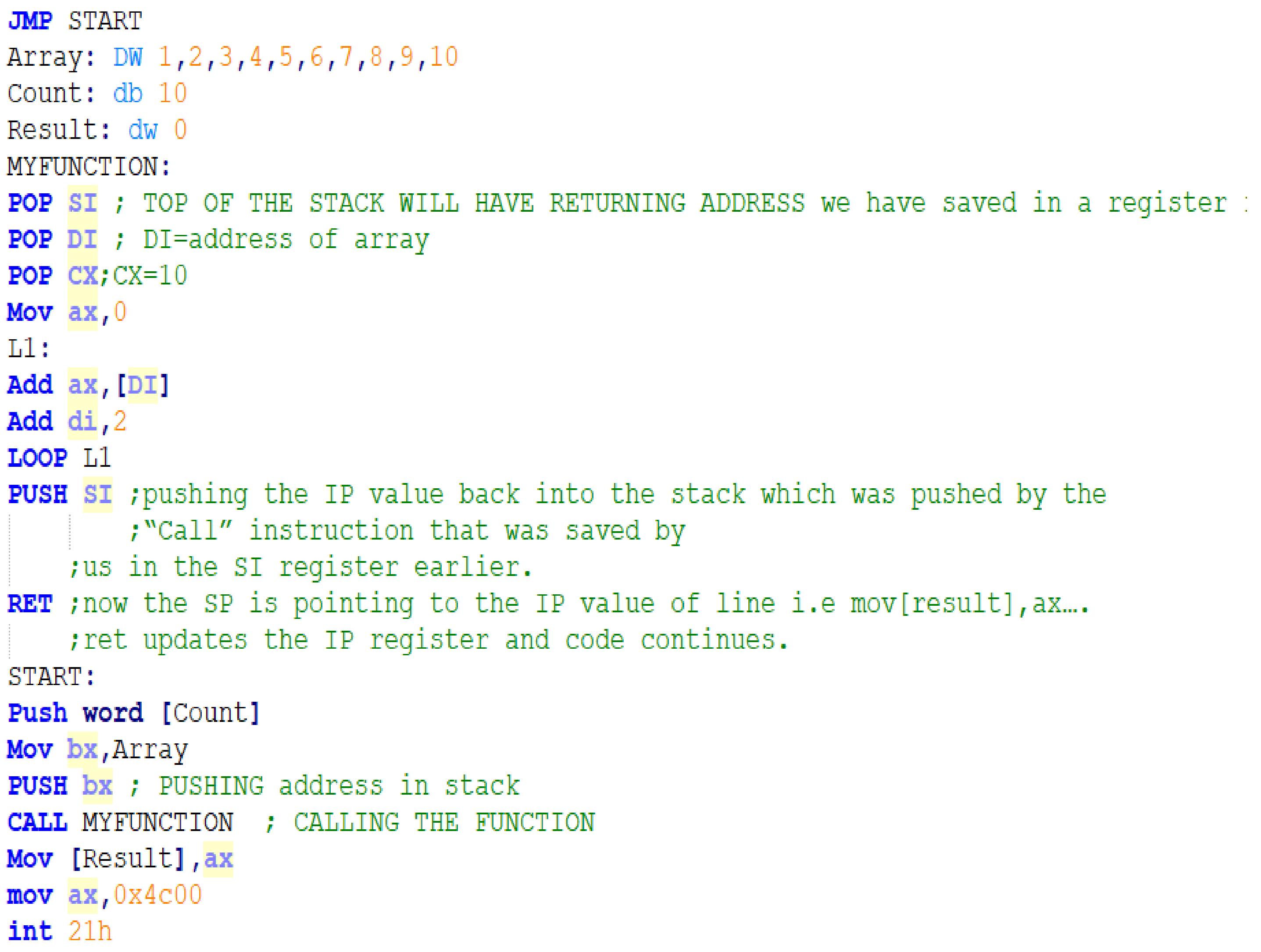
Subroutine Example 1:

USING BP (Base Pointer)

```
JMP START
Array: DW 1,2,3,4,5,6,7,8,9,10
Count: dw 10
Result: dw 0
MYFUNCTION:
MOV BP,SP ; TOP OF THE STACK WILL HAVE RETURNING ADDRESS.
MOV DI,[BP+2] ; DI=address of array
MOV CX,[BP+4] ;CX=10
Mov ax,0
L1:
Add ax,[DI]
Add di,2
LOOP L1
RET
START:
Push word [Count]
Mov bx,Array
PUSH bx ; PUSHING address in stack
CALL MYFUNCTION  ; CALLING THE FUNCTION
Mov [Result],ax
mov ax,0x4c00
int 21h
```

Subroutine Example 2:

WITHOUT USING BP (Base Pointer)

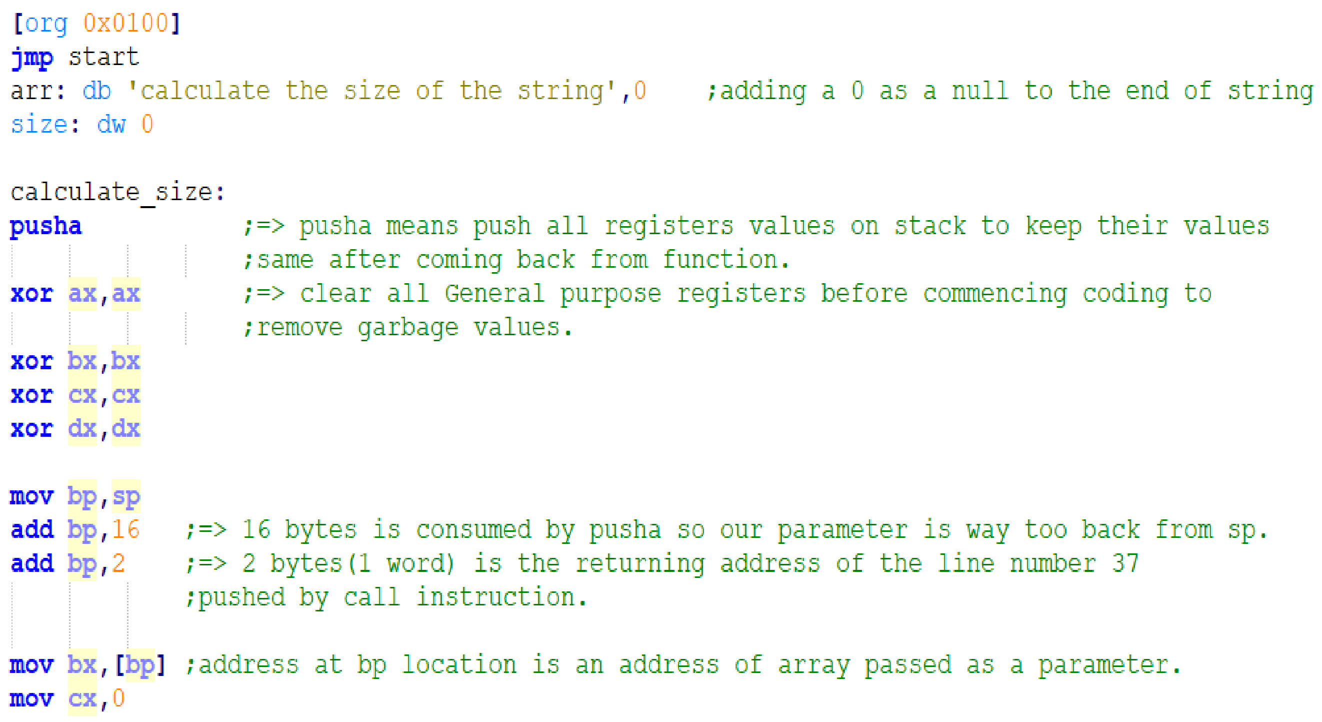


```
JMP START
Array: DW 1,2,3,4,5,6,7,8,9,10
Count: db 10
Result: dw 0
MYFUNCTION:
POP SI ; TOP OF THE STACK WILL HAVE RETURNING ADDRESS we have saved in a register
POP DI ; DI=address of array
POP CX;CX=10
Mov ax,0
L1:
Add ax,[DI]
Add di,2
LOOP L1
PUSH SI ;pushing the IP value back into the stack which was pushed by the
        ;"Call" instruction that was saved by
    ;us in the SI register earlier.
RET ;now the SP is pointing to the IP value of line i.e mov[result],ax….
    ;ret updates the IP register and code continues.
START:
Push word [Count]
Mov bx,Array
PUSH bx ; PUSHING address in stack
CALL MYFUNCTION  ; CALLING THE FUNCTION
Mov [Result],ax
mov ax,0x4c00
int 21h
```

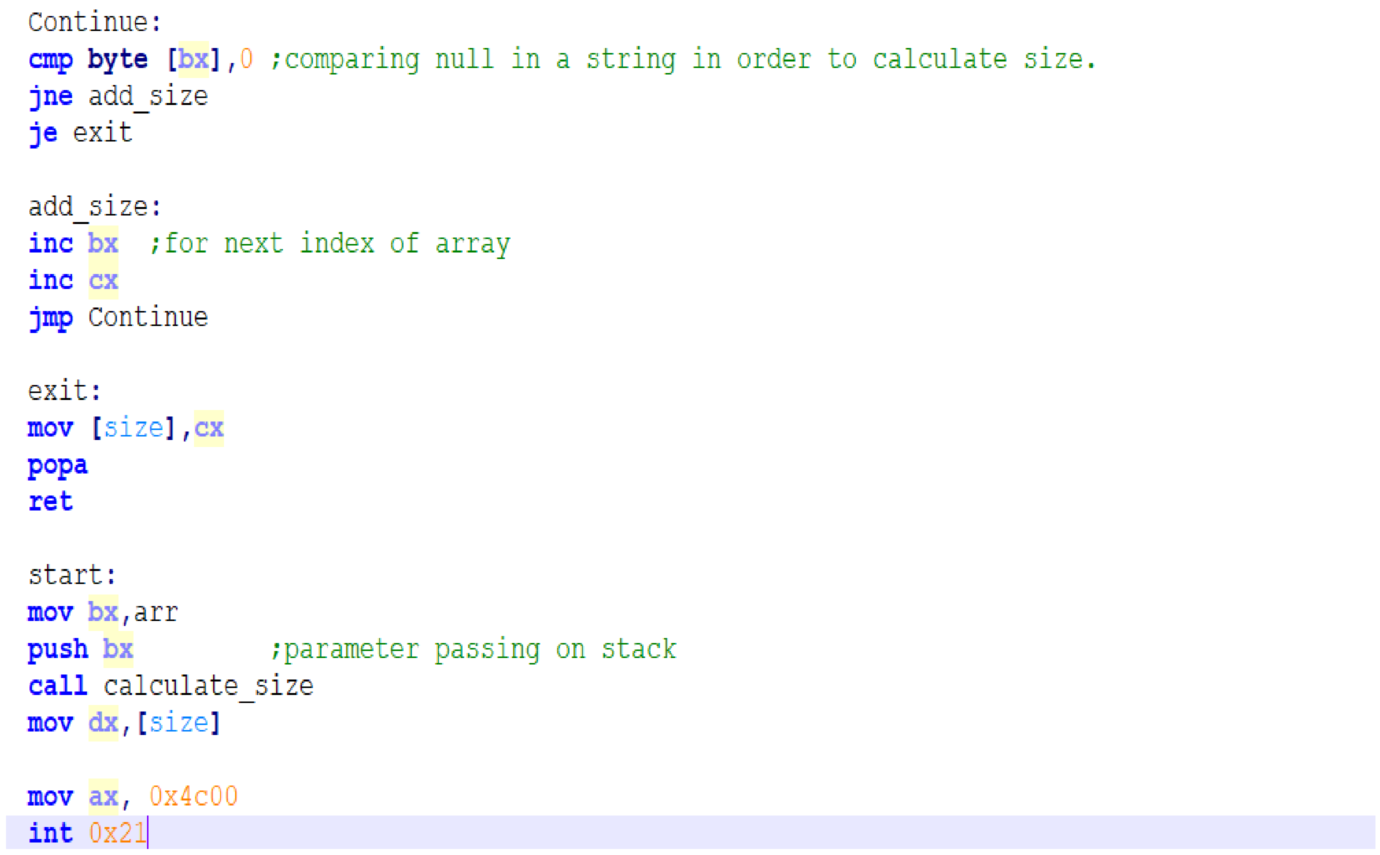
Simple Reading a character ascii example:

```
jmp start
sampleword: db 'UCP FALL 2017'
character: db 0
start:
mov bx,sampleword

mov al,[bx]      ;55 in hex is the ascii of ‘U’
mov cl,[bx+12] ;37 in hex is the ascii of ‘7’
mov [character],cl

mov ax,0x4c00
int 21h
```

Subroutine Example 3:



```
[org 0x0100]
jmp start
arr: db 'calculate the size of the string',0    ;adding a 0 as a null to the end of string
size: dw 0

calculate_size:
pusha               ;=> pusha means push all registers values on stack to keep their values
                    ;same after coming back from function.
xor ax,ax           ;=> clear all General purpose registers before commencing coding to
                    ;remove garbage values.
xor bx,bx
xor cx,cx
xor dx,dx

mov bp,sp
add bp,16       ;=> 16 bytes is consumed by pusha so our parameter is way too back from sp.
add bp,2        ;=> 2 bytes(1 word) is the returning address of the line number 37
                ;pushed by call instruction.

mov bx,[bp] ;address at bp location is an address of array passed as a parameter.
mov cx,0
```



```
Continue:
cmp byte [bx],0 ;comparing null in a string in order to calculate size.
jne add_size
je exit

add_size:
inc bx  ;for next index of array
inc cx
jmp Continue

exit:
mov [size],cx
popa
ret

start:
mov bx,arr
push bx         ;parameter passing on stack
call calculate_size
mov dx,[size]

mov ax, 0x4c00
int 0x21
```

VIDEO MEMORY

Console Display:

Note : Each cell represents a word (2 byte).

Row 1,Col 1	**Row 1,Col 2**	**….**			**Row 1,Col 80**
Row 2,Col 1	**Row 2,Col 2**	**…**			**Row 2,Col 80**
…	**…**	**…**			**…**
…	**…**	**…**			**…**
…	**…**	**…**			**…**
…	**…**	**…**			**…**
…	**….**	**…**			**…**
…	**…**	**…**			**…**
Row 25,Col 1	**Row 25,Col 2**	**…**	**…**	**…**	**Row 25,Col 80**

; if you change the second byte, you can change the color of the character.

; character attribute is 8 bit value,

; high 4 bits set background color and low 4 bits set foreground color.

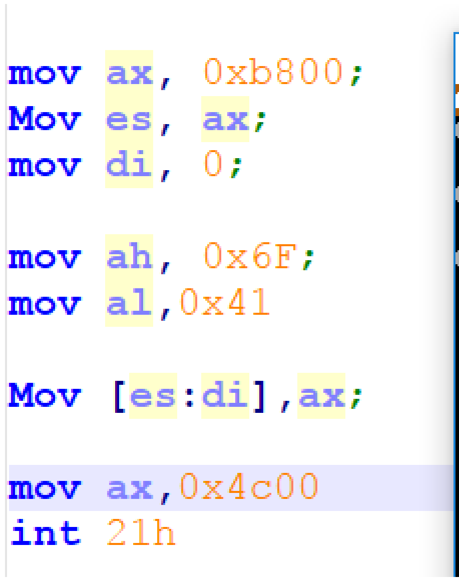
LET AX have 16 bits with character 'A' as a value byte and Brown background with white foreground color.

Blinking of the foreground color	Attribute byte							Value byte							
	Background			Foreground											
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	1	1	0	1	0	0	0	0	0	1

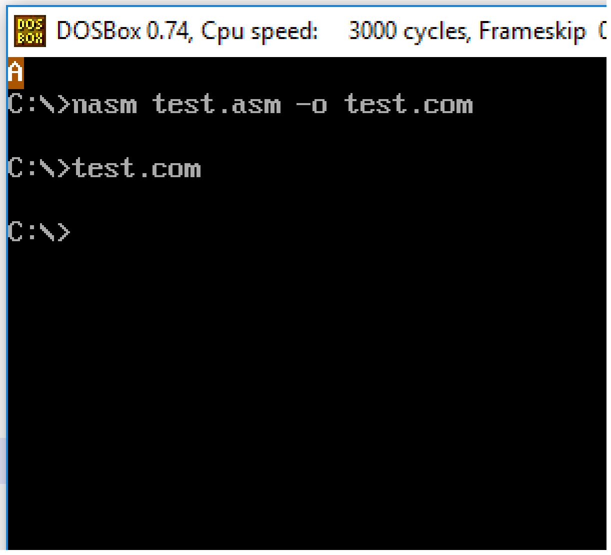
```
; hex   bin     color
; 0     0000    black
; 1     0001    blue
; 2     0010    green
; 3     0011    cyan
; 4     0100    red
; 5     0101    magenta
; 6     0110    brown
; 7     0111    light gray
; 8     1000    dark gray
; 9     1001    light blue
; a     1010    light green
; b     1011    light cyan
; c     1100    light red
; d     1101    light magenta
; e     1110    yellow
; f     1111    white
```

$\Big\}$ possible background colors (0–7)

$\Big\}$ possible foreground color (0–f)



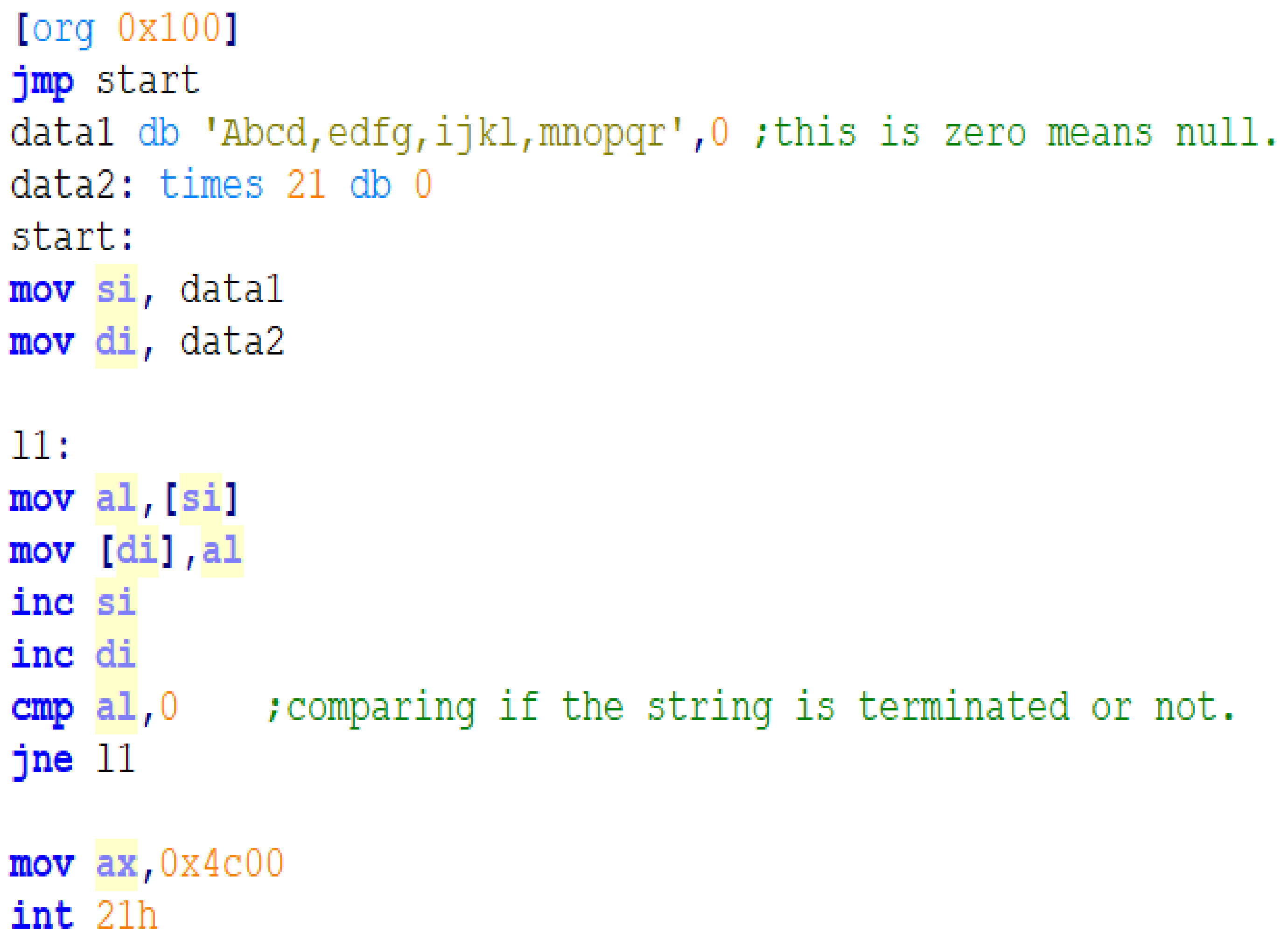
```
mov ax, 0xb800;
Mov es, ax;
mov di, 0;

mov ah, 0x6F;
mov al,0x41

Mov [es:di],ax;

mov ax,0x4c00
int 21h
```



DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip

```
A
C:\>nasm test.asm -o test.com

C:\>test.com

C:\>
```

Copy character array from one to another.

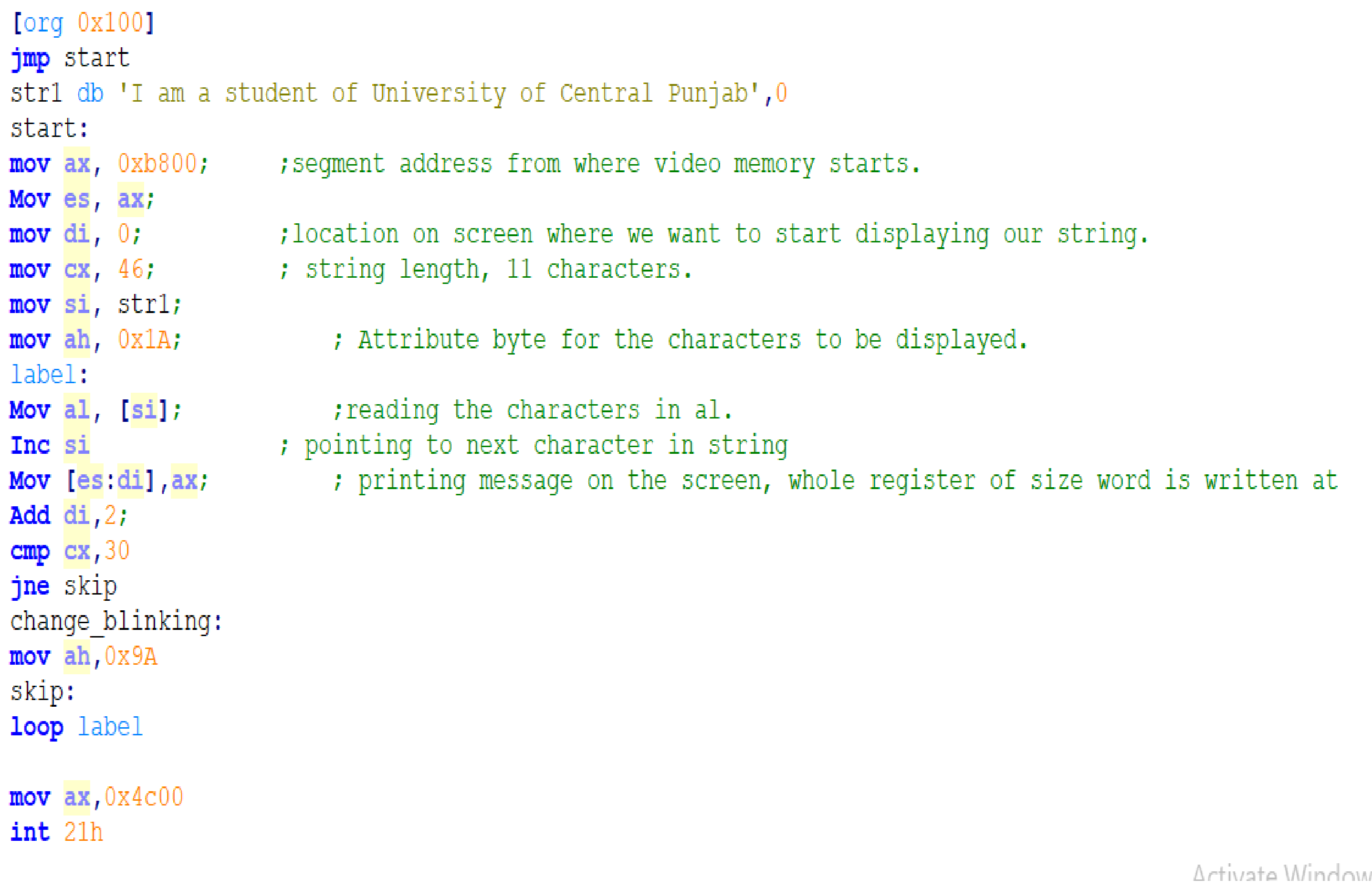


```
[org 0x100]
jmp start
data1 db 'Abcd,edfg,ijkl,mnopqr',0 ;this is zero means null.
data2: times 21 db 0
start:
mov si, data1
mov di, data2

l1:
mov al,[si]
mov [di],al
inc si
inc di
cmp al,0    ;comparing if the string is terminated or not.
jne l1

mov ax,0x4c00
int 21h
```

“To run code without debugging simply type test.com instead of afd test.com”

Type cls then enter before running the following codes.

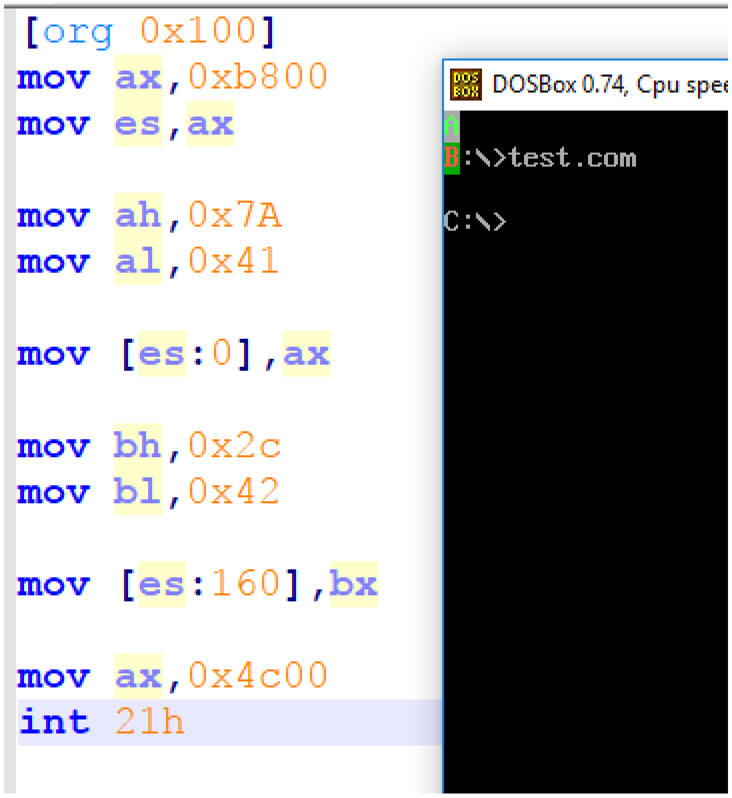
Display string on screen



```
[org 0x100]
jmp start
str1 db 'I am a student of University of Central Punjab',0
start:
mov ax, 0xb800;     ;segment address from where video memory starts.
Mov es, ax;
mov di, 0;          ;location on screen where we want to start displaying our string.
mov cx, 46;         ; string length, 11 characters.
mov si, str1;
mov ah, 0x1A;           ; Attribute byte for the characters to be displayed.
label:
Mov al, [si];           ;reading the characters in al.
Inc si              ; pointing to next character in string
Mov [es:di],ax;         ; printing message on the screen, whole register of size word is written at
Add di,2;
cmp cx,30
jne skip
change_blinking:
mov ah,0x9A
skip:
loop label

mov ax,0x4c00
int 21h
```

For example:

- Different attribute values of each word
- Different locations can be accessed for the display.



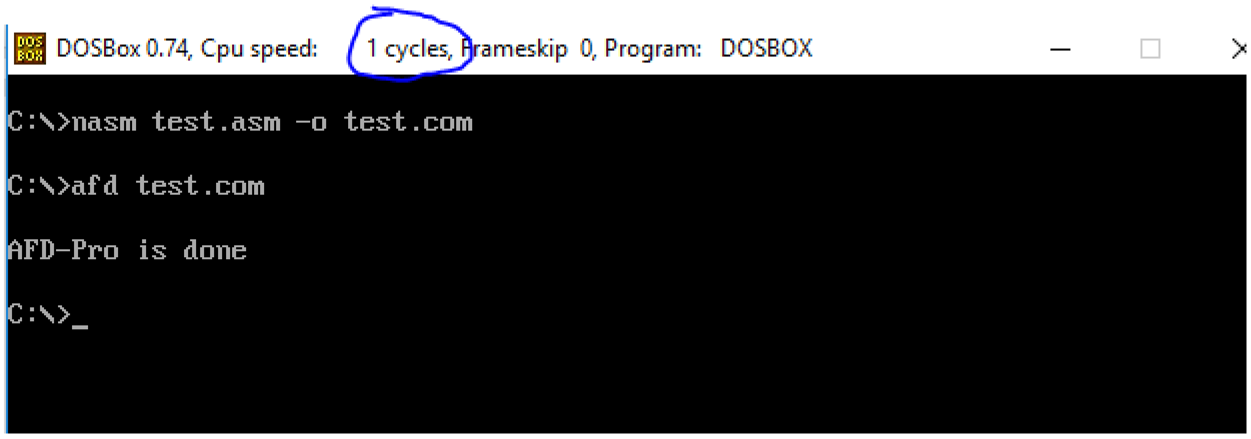
```
[org 0x100]
mov ax,0xb800
mov es,ax

mov ah,0x7A
mov al,0x41

mov [es:0],ax

mov bh,0x2c
mov bl,0x42

mov [es:160],bx

mov ax,0x4c00
int 21h
```

```
AB:\>test.com

C:\>
```

Display code which writes and clears the string from screen.

USE CTRL+F11 to reduce cycles / sec or CTRL+F12 to increase the speed of dosbox.

Slow down the speed of dosbox by press and hold ctrl and press F11 till 1 cycle



DOSBox 0.74, Cpu speed: 1 cycles, Frameskip 0, Program: DOSBOX

```
C:\>nasm test.asm -o test.com

C:\>afd test.com

AFD-Pro is done

C:\>_
```

```
[org 0x100]
jmp start
str1 db 'HELLO WORLD'
start:
mov ax, 0xb800;
Mov es, ax;
mov di, 500;
mov cx, 11; ; string length, 11 characters.
mov si, str1;
mov ah, 0x1A;    ; Attribute byte, use any number
l1:
Mov al, [si];
Inc si; pointing to next character in string
Mov [es:di],ax; ; printing message on the screen;
Add di,2;
loop l1

mov cx, 2000;    ; total screen locations.
mov ax, 0x0720; Attribute byte (07) and (20h) ASCII for space character.
mov di, 0;  ; start from top left
l2:
Mov [es:di],ax; ; writing blank spaces on whole screen
Add di,2;
loop l2
mov ax,0x4c00
int 21h
```

ASCII CODES

HEX format

00:	null	20:	spa	40:	@	60:	`	80:	Ç	A0:	á	C0:	└	E0:	α
01:	☺	21:	!	41:	A	61:	a	81:	ü	A1:	í	C1:	┴	E1:	ß
02:	☻	22:	"	42:	B	62:	b	82:	é	A2:	ó	C2:	┬	E2:	Γ
03:	♥	23:	#	43:	C	63:	c	83:	â	A3:	ú	C3:	├	E3:	π
04:	♦	24:	$	44:	D	64:	d	84:	ä	A4:	ñ	C4:	─	E4:	Σ
05:	♣	25:	%	45:	E	65:	e	85:	à	A5:	Ñ	C5:	┼	E5:	σ
06:	♠	26:	&	46:	F	66:	f	86:	å	A6:	ª	C6:	╞	E6:	µ
07:	beep	27:	'	47:	G	67:	g	87:	ç	A7:	º	C7:	╟	E7:	τ
08:	back	28:	(	48:	H	68:	h	88:	ê	A8:	¿	C8:	╚	E8:	Φ
09:	tab	29:	)	49:	I	69:	i	89:	ë	A9:	⌐	C9:	╔	E9:	Θ
0A:	newl	2A:	*	4A:	J	6A:	j	8A:	è	AA:	¬	CA:	╩	EA:	Ω
0B:	♂	2B:	+	4B:	K	6B:	k	8B:	ï	AB:	½	CB:	╦	EB:	δ
0C:	♀	2C:	,	4C:	L	6C:	l	8C:	î	AC:	¼	CC:	╠	EC:	∞
0D:	cret	2D:	-	4D:	M	6D:	m	8D:	ì	AD:	¡	CD:	═	ED:	φ
0E:	♫	2E:	.	4E:	N	6E:	n	8E:	Ä	AE:	«	CE:	╬	EE:	ε
0F:	☼	2F:	/	4F:	O	6F:	o	8F:	Å	AF:	»	CF:	╧	EF:	∩
10:	►	30:	0	50:	P	70:	p	90:	É	B0:	░	D0:	╨	F0:	≡
11:	◄	31:	1	51:	Q	71:	q	91:	æ	B1:	▒	D1:	╤	F1:	±
12:	↕	32:	2	52:	R	72:	r	92:	Æ	B2:	▓	D2:	╥	F2:	≥
13:	‼	33:	3	53:	S	73:	s	93:	ô	B3:	│	D3:	╙	F3:	≤
14:	¶	34:	4	54:	T	74:	t	94:	ö	B4:	┤	D4:	╘	F4:	⌠
15:	§	35:	5	55:	U	75:	u	95:	ò	B5:	╡	D5:	╒	F5:	⌡
16:	▬	36:	6	56:	V	76:	v	96:	û	B6:	╢	D6:	╓	F6:	÷
17:	↨	37:	7	57:	W	77:	w	97:	ù	B7:	╖	D7:	╫	F7:	≈
18:	↑	38:	8	58:	X	78:	x	98:	ÿ	B8:	╕	D8:	╪	F8:	°
19:	↓	39:	9	59:	Y	79:	y	99:	Ö	B9:	╣	D9:	┘	F9:	∙
1A:	→	3A:	:	5A:	Z	7A:	z	9A:	Ü	BA:	║	DA:	┌	FA:	·
1B:	←	3B:	;	5B:	[	7B:	{	9B:	¢	BB:	╗	DB:	█	FB:	√
1C:	∟	3C:	<	5C:	\	7C:	¦	9C:	£	BC:	╝	DC:	▄	FC:	ⁿ
1D:	↔	3D:	=	5D:	]	7D:	}	9D:	¥	BD:	╜	DD:	▌	FD:	²
1E:	▲	3E:	>	5E:	^	7E:	~	9E:	₧	BE:	╛	DE:	▐	FE:	■
1F:	▼	3F:	?	5F:	_	7F:	⌂	9F:	ƒ	BF:	┐	DF:	▀	FF:	res

ASCII CODES

Decimal format

000:	null	032:	spa	064:	@	096:	`	128:	Ç	160:	á	192:	└	224:	α
001:	☺	033:	!	065:	A	097:	a	129:	ü	161:	í	193:	┴	225:	β
002:	☻	034:	"	066:	B	098:	b	130:	é	162:	ó	194:	┬	226:	Γ
003:	♥	035:	#	067:	C	099:	c	131:	â	163:	ú	195:	├	227:	π
004:	♦	036:	$	068:	D	100:	d	132:	ä	164:	ñ	196:	─	228:	Σ
005:	♣	037:	%	069:	E	101:	e	133:	à	165:	Ñ	197:	┼	229:	σ
006:	♠	038:	&	070:	F	102:	f	134:	å	166:	ª	198:	╞	230:	µ
007:	beep	039:	'	071:	G	103:	g	135:	ç	167:	º	199:	╟	231:	τ
008:	back	040:	(	072:	H	104:	h	136:	ê	168:	¿	200:	╚	232:	Φ
009:	tab	041:	)	073:	I	105:	i	137:	ë	169:	⌐	201:	╔	233:	Θ
010:	newl	042:	*	074:	J	106:	j	138:	è	170:	¬	202:	╩	234:	Ω
011:	♂	043:	+	075:	K	107:	k	139:	ï	171:	½	203:	╦	235:	δ
012:	♀	044:	,	076:	L	108:	l	140:	î	172:	¼	204:	╠	236:	∞
013:	cret	045:	-	077:	M	109:	m	141:	ì	173:	¡	205:	═	237:	φ
014:	♫	046:	.	078:	N	110:	n	142:	Ä	174:	«	206:	╬	238:	ε
015:	☼	047:	/	079:	O	111:	o	143:	Å	175:	»	207:	╧	239:	∩
016:	►	048:	0	080:	P	112:	p	144:	É	176:	░	208:	╨	240:	≡
017:	◄	049:	1	081:	Q	113:	q	145:	æ	177:	▒	209:	╤	241:	±
018:	↕	050:	2	082:	R	114:	r	146:	Æ	178:	▓	210:	╥	242:	≥
019:	‼	051:	3	083:	S	115:	s	147:	ô	179:	│	211:	╙	243:	≤
020:	¶	052:	4	084:	T	116:	t	148:	ö	180:	┤	212:	╘	244:	⌠
021:	§	053:	5	085:	U	117:	u	149:	ò	181:	╡	213:	╒	245:	⌡
022:	▬	054:	6	086:	V	118:	v	150:	û	182:	╢	214:	╓	246:	÷
023:	↨	055:	7	087:	W	119:	w	151:	ù	183:	╖	215:	╫	247:	≈
024:	↑	056:	8	088:	X	120:	x	152:	ÿ	184:	╕	216:	╪	248:	°
025:	↓	057:	9	089:	Y	121:	y	153:	Ö	185:	╣	217:	┘	249:	∙
026:	→	058:	:	090:	Z	122:	z	154:	Ü	186:	║	218:	┌	250:	·
027:	←	059:	;	091:	[	123:	{	155:	¢	187:	╗	219:	█	251:	√
028:	∟	060:	<	092:	\	124:	¦	156:	£	188:	╝	220:	▄	252:	ⁿ
029:	↔	061:	=	093:	]	125:	}	157:	¥	189:	╜	221:	▌	253:	²
030:	▲	062:	>	094:	^	126:	~	158:	₧	190:	╛	222:	▐	254:	■
031:	▼	063:	?	095:	_	127:	⌂	159:	ƒ	191:	┐	223:	▀	255:	res

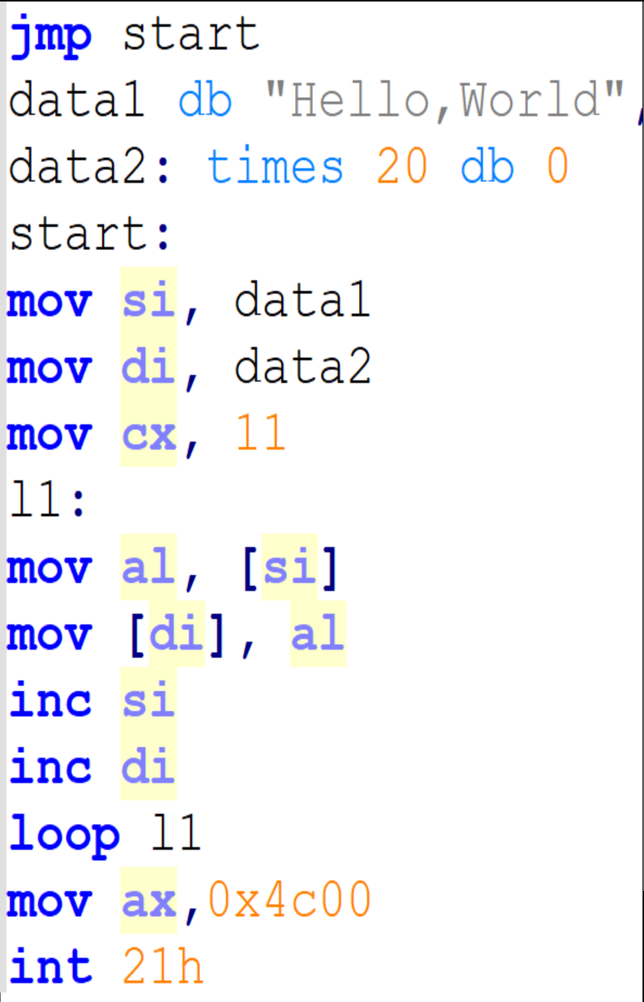
String Instructions

Instruction	Functionality actually performed	
movsb	1. Mov [ES:DI],[DS:SI] 2. Inc si 3. Inc di	Invalid instruction (memory to memory)
movsw	1. Mov [ES:DI],[DS:SI] 2. Add si,2 3. Add di,2	Invalid instruction (memory to memory)
scasb	1. Cmp al,[ES:DI];ZF=1 if same 2. Inc DI	
scasw	1. Cmp ax,[DI];ZF=1 if same 2. Add DI,2	
cmpsb	1. Cmp [DS:SI],[ES:DI];ZF=1 if same 2. Inc SI 3. Inc DI	Invalid instruction (memory to memory)
cmpsw	1. Cmp [DS:SI],[ES:DI];ZF=1 if same 2. Add si,2 3. Add di,2	Invalid instruction (memory to memory)
lodsb	1. Mov al,[DS:SI] 2. Inc si	
lodsw	1. Mov ax,[DS:SI] 2. Add si,2	
stosb	1. Mov [ES:DI],al 2. Inc di	
stosw	1. Mov [ES:DI],ax 2. Add di,2	
Rep	It repeats the instruction cx times.	
Repe	It executes the instruction cx times or until zf remains 1.	
Repne	It executes the instruction cx times or exit when zf becomes 1.	

Note: All yellow highlighted instructions will depend upon direction flag(cld, std) see second last example.

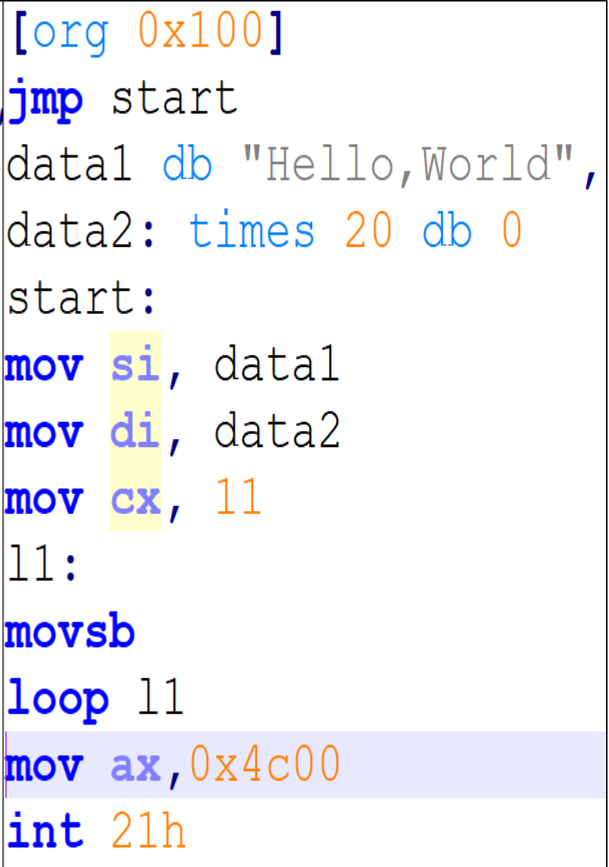
String Examples

Simple String(Example)



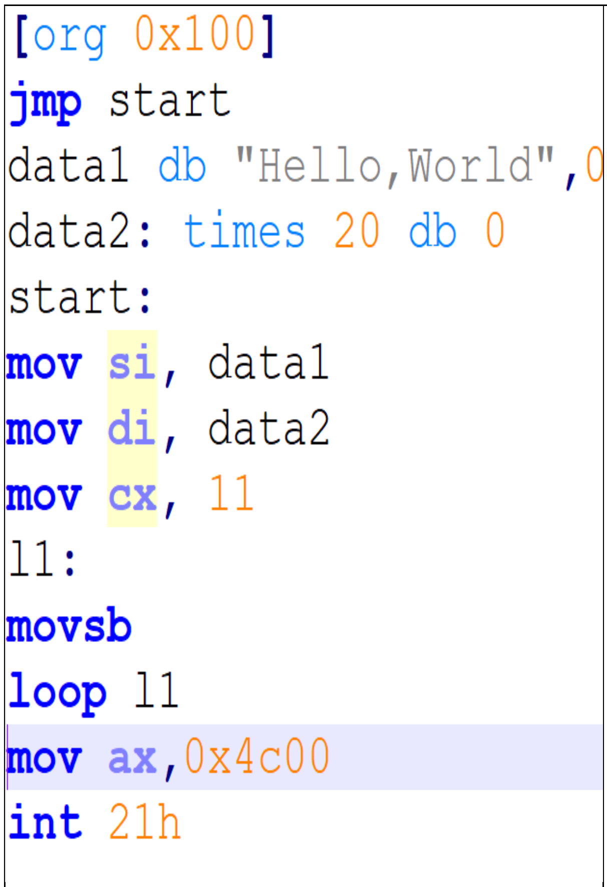
```
jmp start
data1 db "Hello,World",
data2: times 20 db 0
start:
mov si, data1
mov di, data2
mov cx, 11
l1:
mov al, [si]
mov [di], al
inc si
inc di
loop l1
mov ax,0x4c00
int 21h
```

movsb(Example)



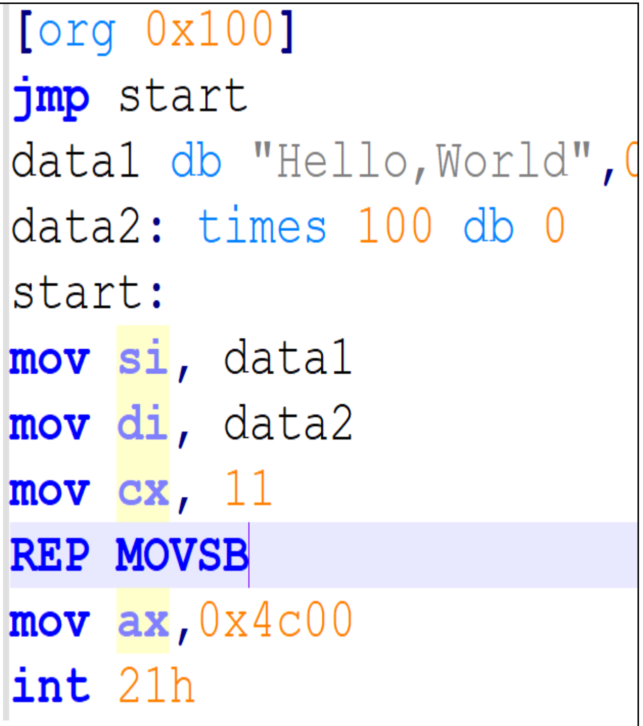
```
[org 0x100]
jmp start
data1 db "Hello,World",
data2: times 20 db 0
start:
mov si, data1
mov di, data2
mov cx, 11
l1:
movsb
loop l1
mov ax,0x4c00
int 21h
```

Using loop instruction(Example)



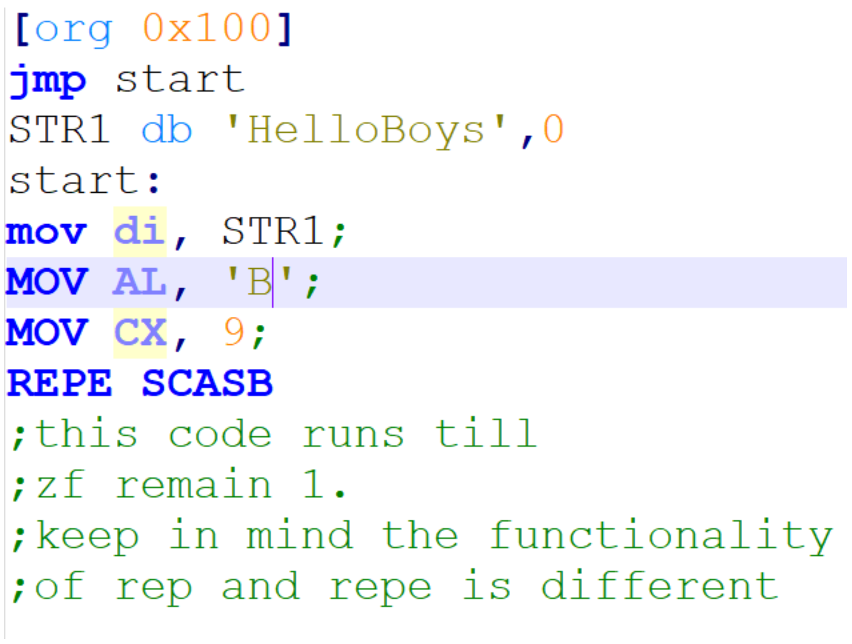
```
[org 0x100]
jmp start
data1 db "Hello,World",0
data2: times 20 db 0
start:
mov si, data1
mov di, data2
mov cx, 11
l1:
movsb
loop l1
mov ax,0x4c00
int 21h
```

Using REP instruction(Example)



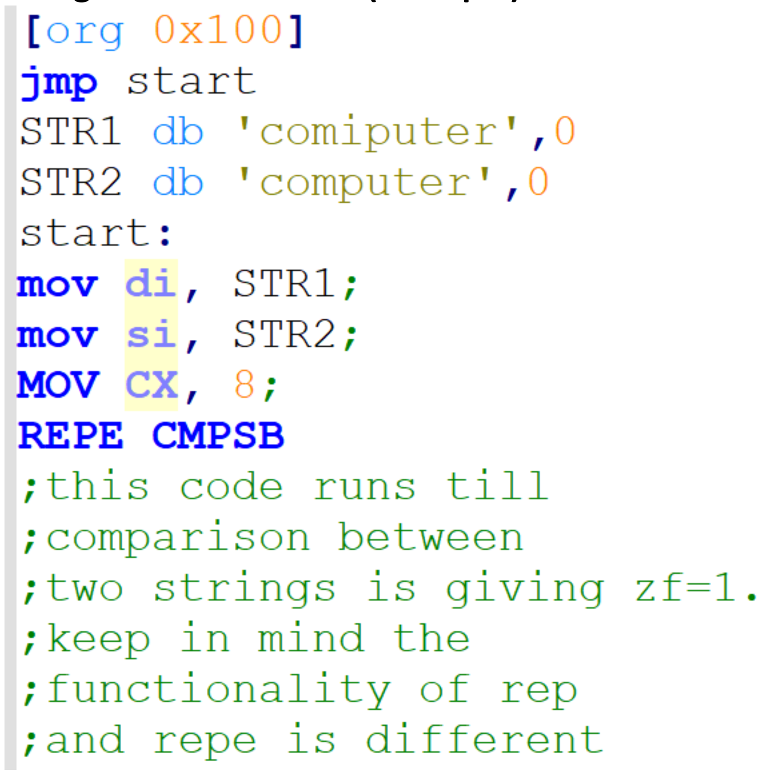
```
[org 0x100]
jmp start
data1 db "Hello,World",0
data2: times 100 db 0
start:
mov si, data1
mov di, data2
mov cx, 11
REP MOVSB
mov ax,0x4c00
int 21h
```

Using SCAS instruction(Example)



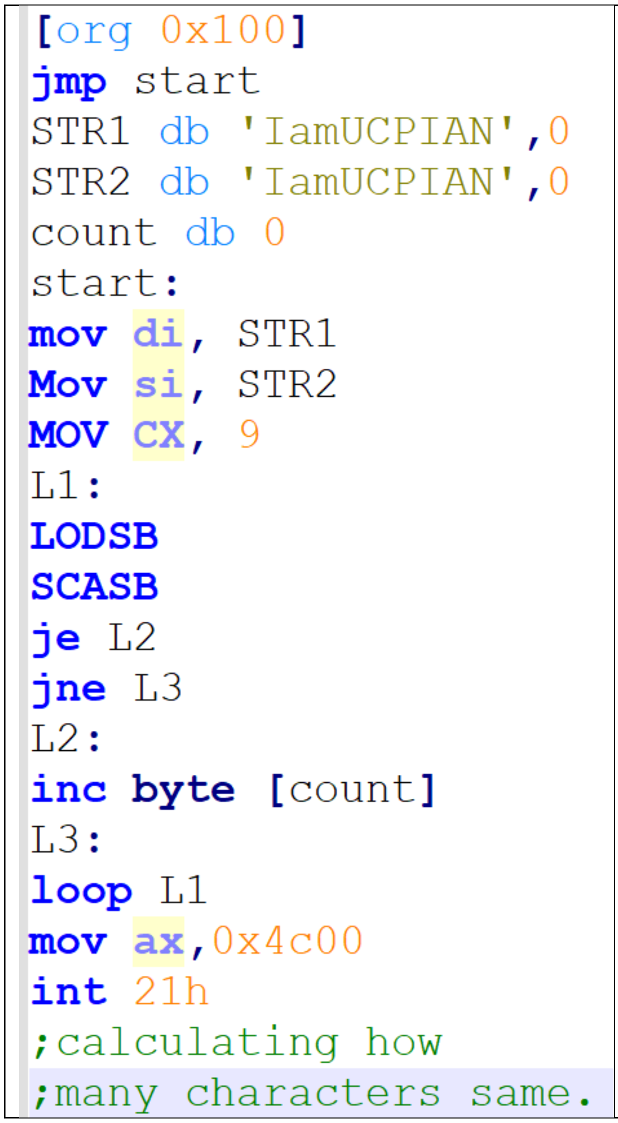
```
[org 0x100]
jmp start
STR1 db 'HelloBoys',0
start:
mov di, STR1;
MOV AL, 'B';
MOV CX, 9;
REPE SCASB
;this code runs till
;zf remain 1.
;keep in mind the functionality
;of rep and repe is different
```

Using CMPS instruction(Example)



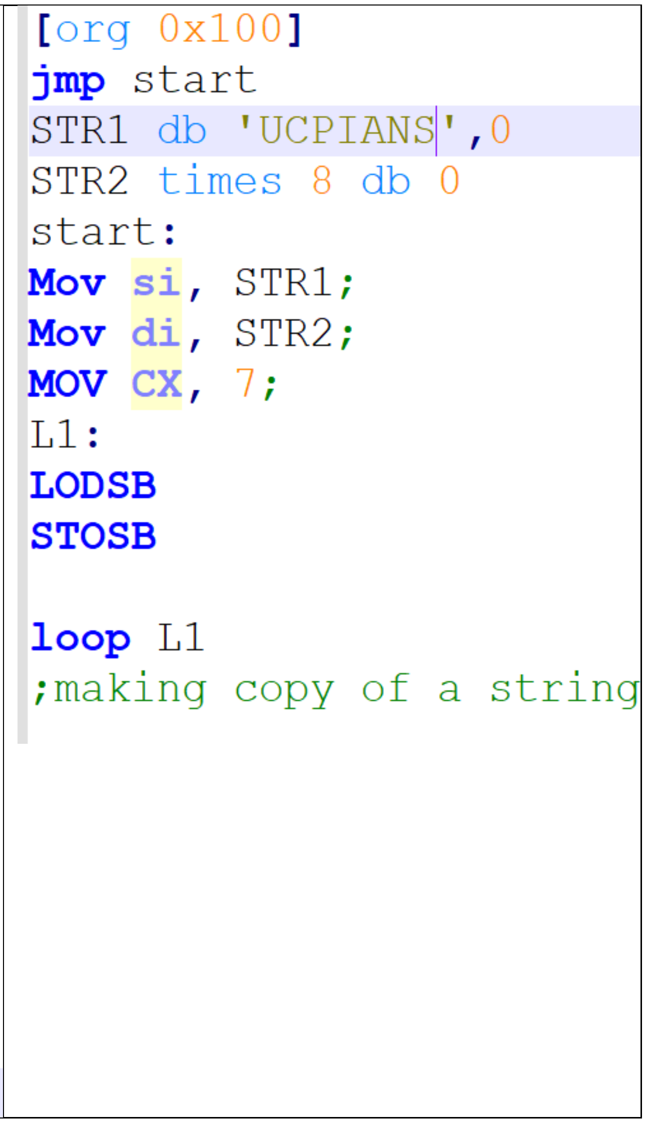
```
[org 0x100]
jmp start
STR1 db 'comiputer',0
STR2 db 'computer',0
start:
mov di, STR1;
mov si, STR2;
MOV CX, 8;
REPE CMPSB
;this code runs till
;comparison between
;two strings is giving zf=1.
;keep in mind the
;functionality of rep
;and repe is different
```

Using LODSB instruction(Example)



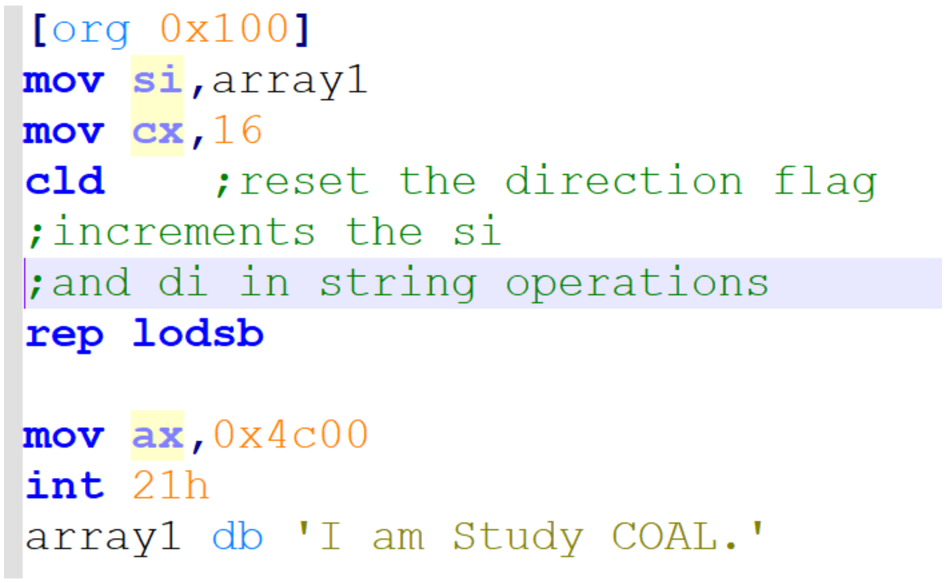
```
[org 0x100]
jmp start
STR1 db 'IamUCPIAN',0
STR2 db 'IamUCPIAN',0
count db 0
start:
mov di, STR1
Mov si, STR2
MOV CX, 9
L1:
LODSB
SCASB
je L2
jne L3
L2:
inc byte [count]
L3:
loop L1
mov ax,0x4c00
int 21h
;calculating how
;many characters same.
```

Using STOSB instruction(Example)



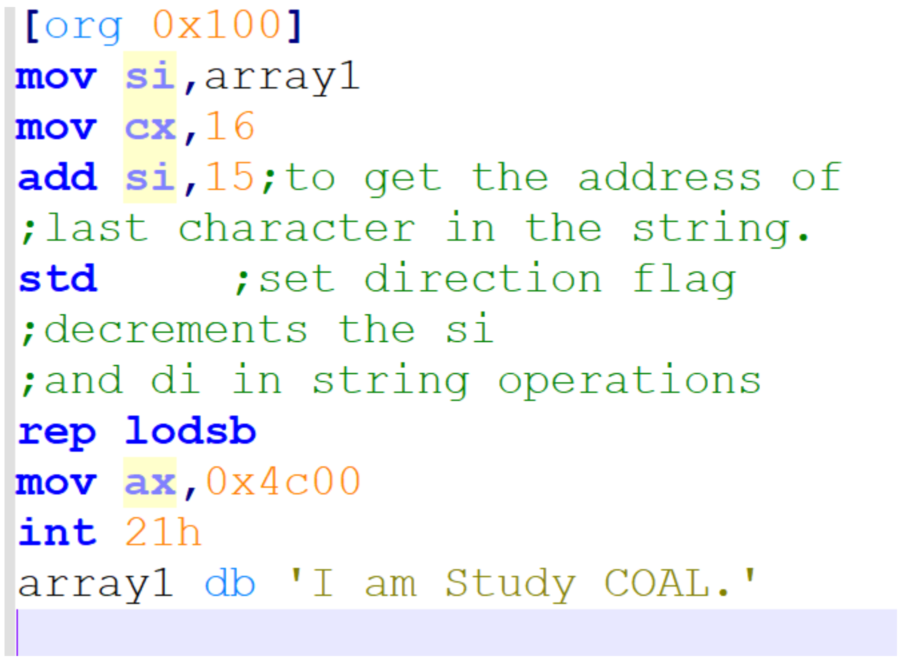
```
[org 0x100]
jmp start
STR1 db 'UCPIANS',0
STR2 times 8 db 0
start:
Mov si, STR1;
Mov di, STR2;
MOV CX, 7;
L1:
LODSB
STOSB

loop L1
;making copy of a string
```

Traversing array from left to right



```
[org 0x100]
mov si,array1
mov cx,16
cld     ;reset the direction flag
;increments the si
;and di in string operations
rep lodsb

mov ax,0x4c00
int 21h
array1 db 'I am Study COAL.'
```

Traversing array from right to left



```
[org 0x100]
mov si,array1
mov cx,16
add si,15;to get the address of
;last character in the string.
std     ;set direction flag
;decrements the si
;and di in string operations
rep lodsb
mov ax,0x4c00
int 21h
array1 db 'I am Study COAL.'
```

Using string operations with video memory.

```
[org 0x100]
jmp start
data1 db "HELLOUCPIAN";
data2: times 11 db 0
start:
mov si, data1
mov di, data2
mov cx, 11
l1:
movsb
loop l1
mov cx,21
mov ax,0xb800
mov es,ax
mov si,data1
mov di,0
mov ah,0x3f
label1:
    lodsb
    stosw
    loop label1
mov ax,0x4c00
int 21h
```

Software Interrupts

Example 1: Printing Character Using Interrupt

```
[org 0x100]
start:  mov ah, 0
int 16h ; wait for any key....
cmp al, 27  ; if key is 'esc' then exit.
je stop
;al contains ascii of pressed key
mov ah, 0Eh ; print it.
int 10h
jmp start
stop:
mov ax,0x4c00
int 21h
```

Example 2: Printing String Using Interrupt

```
[org 0x100]
mov al, 1;update curser after every character printing
mov bh, 0;page 0, means first page
mov bl, 00111011b;attribules
mov cx, 15 ; message size
mov dl, 10 ;row
mov dh, 7  ;col
push cs
pop es
mov bp, msg1
mov ah, 13h
int 10h
mov ax,0x4c00
int 21h
msg1 db " hello, world! "
```

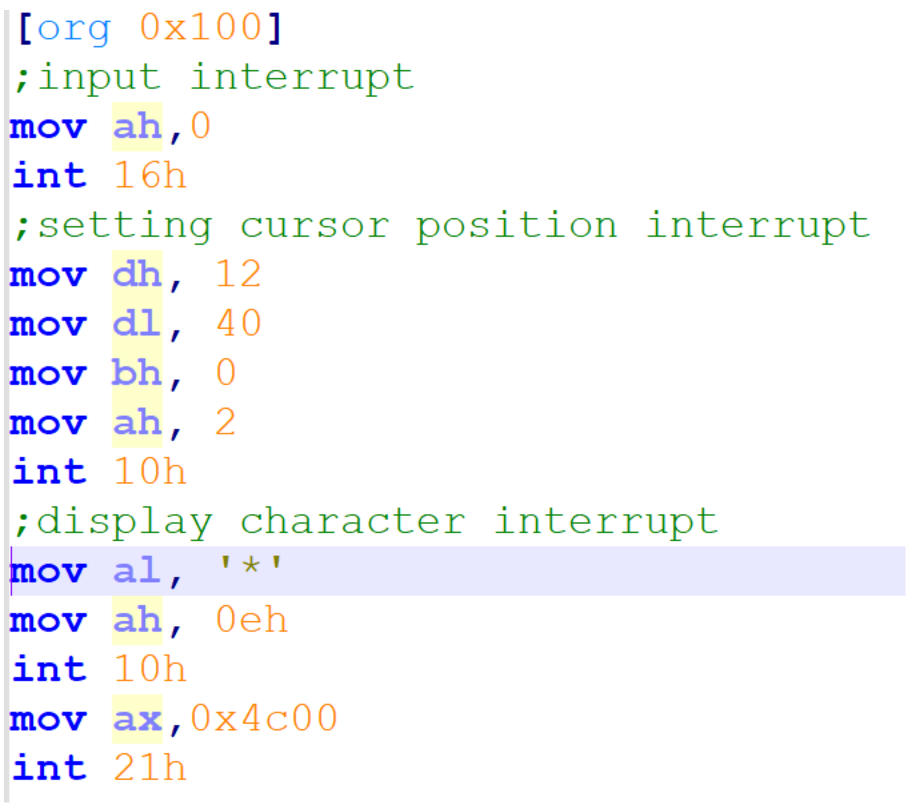
Example 3: (Taking Input from User and display)

```
[org 0x100]
MOV AX, 0xB800
MOV ES, AX; Initializing ES with video memory address

MOV AH, 0; service number
INT 0x16; calling interrupt number 16h
; When you call interrupt 16h with service number 0,
;processor waits for keyboard input.
;When a key is pressed, its ASCII value is stored in AL register.
; Printing the character on screen.
MOV DI, 0; screen location di=0 top left.
MOV AH,07h; attribute byte
STOSW    ; displaying on screen
Mov ax,0x4c00
Int 21h
```

Example 4: (Take input from User until they press Esc)

```
[org 0x100]
MOV AX, 0xB800
MOV ES, AX; Initializing ES with video memory address
XOR DI, DI; screen location di=0 top left.
again:
MOV AH, 0; service number
INT 0x16; calling interrupt number 16h
; When you call interrupt 16h with service number 0,
;processor waits for keyboard input.
;When a key is pressed, its ASCII value is stored in AL register.
; Printing the character on screen.
MOV AH,07h  ; attribute byte
STOSW    ; displaying on screen
cmp al, 0x1b
jne again
mov ax,0x4c00
int 21h
```

Example 5: (Taking Input from User setting cursor position display the character)

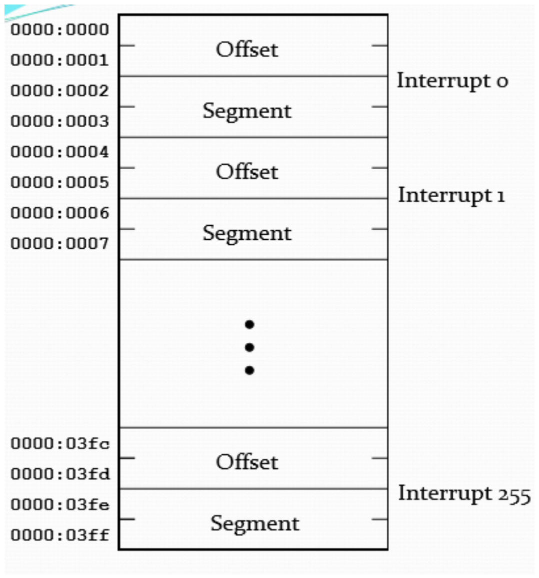


```
[org 0x100]
;input interrupt
mov ah,0
int 16h
;setting cursor position interrupt
mov dh, 12
mov dl, 40
mov bh, 0
mov ah, 2
int 10h
;display character interrupt
mov al, '*'
mov ah, 0eh
int 10h
mov ax,0x4c00
int 21h
```

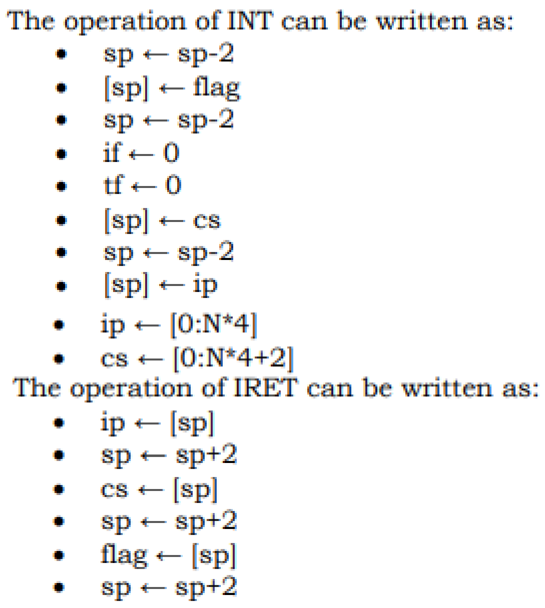
Interrupts hooking and unhooking

Interrupt vector table-address mapping

- Offset: n*4 ; offset address of nth interrupt
- Segment: n*4+2 ; base address of nth interrupt



If N is the interrupt number then following operations are executed by the INT and IRET by the processor.



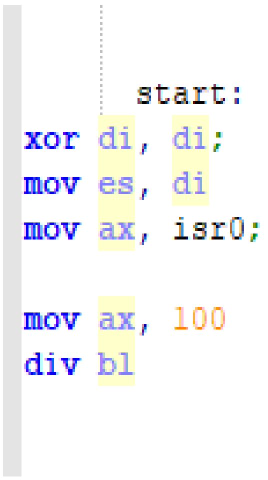
The operation of INT can be written as:

- $sp \leftarrow sp-2$
- $[sp] \leftarrow flag$
- $sp \leftarrow sp-2$
- $if \leftarrow 0$
- $tf \leftarrow 0$
- $[sp] \leftarrow cs$
- $sp \leftarrow sp-2$
- $[sp] \leftarrow ip$
- $ip \leftarrow [0:N*4]$
- $cs \leftarrow [0:N*4+2]$

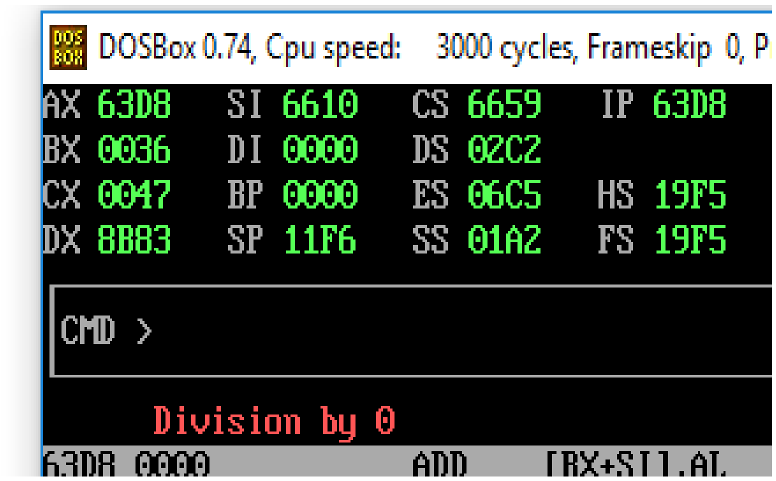
The operation of IRET can be written as:

- $ip \leftarrow [sp]$
- $sp \leftarrow sp+2$
- $cs \leftarrow [sp]$
- $sp \leftarrow sp+2$
- $flag \leftarrow [sp]$
- $sp \leftarrow sp+2$

Interrupt zero: INT 0



```
    start:
xor di, di;
mov es, di
mov ax, isr0;

mov ax, 100
div bl
```



```
DOSBox 0.74, Cpu speed:   3000 cycles, Frameskip 0, P
AX 63D8    SI 6610    CS 6659    IP 63D8
BX 0036    DI 0000    DS 02C2
CX 0047    BP 0000    ES 06C5    HS 19F5
DX 8B83    SP 11F6    SS 01A2    FS 19F5

 CMD >

      Division by 0
63D8 0000           ADD     [BX+SI],AL
```

Example 6: ISR(Interrupt Service Routine) hooking-interrupt zero

```
[org 0x100]
jmp start
message db 'Your message for divide overflow Interupt',0;
isr0:
pop ax;pop the IP of div instruction
push continue;push the IP of next instruction after "DIV"
mov ax, 0xb800
mov es, ax;
mov si, message;
mov ah,7
nextchar:
lodsb;
cmp al, 0
je skip
stosw
jmp nextchar
skip:
iret
```

```
start:
xor di, di;
mov es, di
mov ax, isr0;
mov [es:0h*4],ax;
mov [es:0H*4+2], cs;
mov ax, 100
div bl ;when div interrupt is called it pushes the IP value of itself
;instead of the next instruction from where our code
; should continue after returning from interrupt.
continue:
mov ax,0x4c00
int 21h
```

Example 7: Another Interrupt hooking

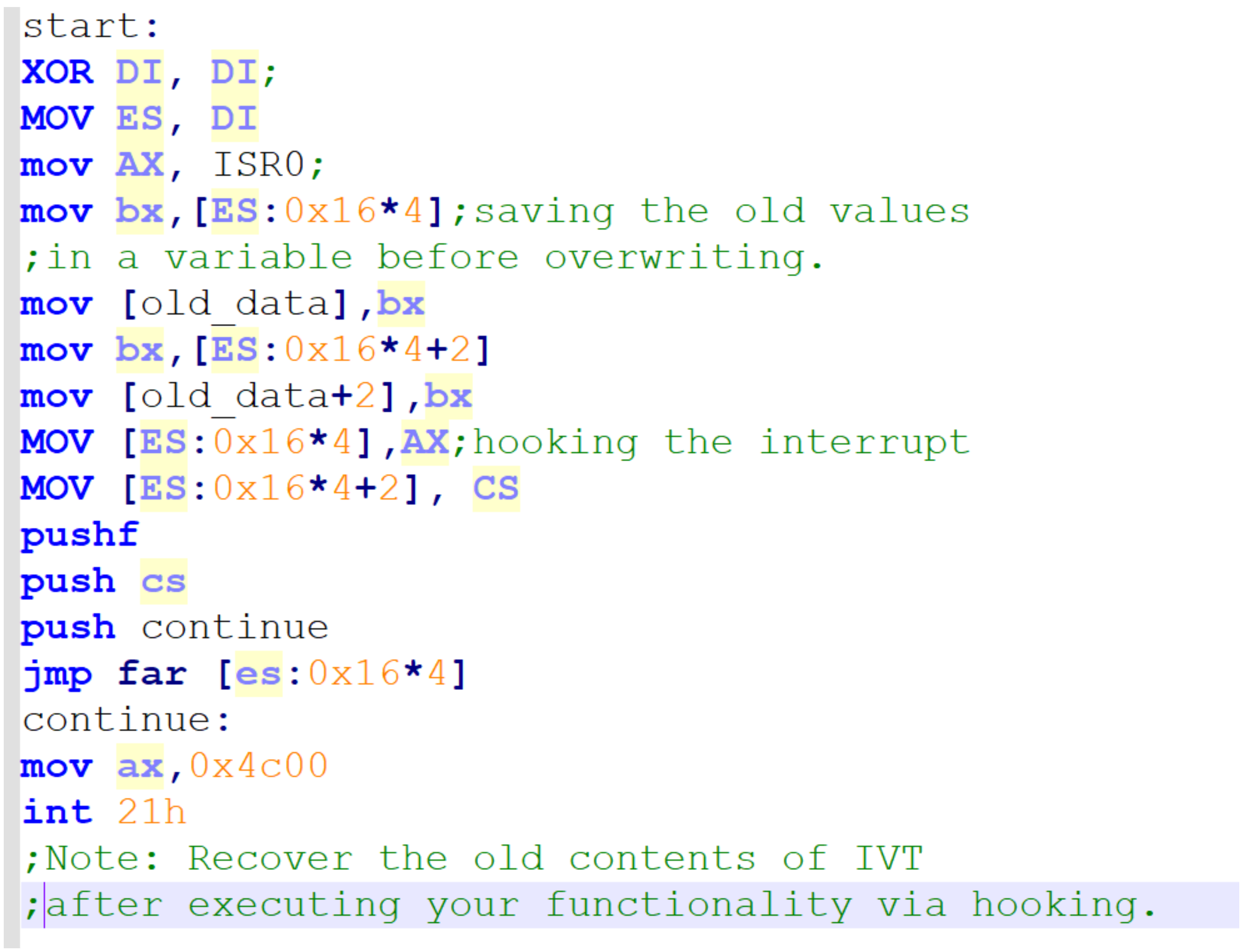
```
[org 0x100]
jmp start
ISR0:
MOV AX, 0XB800
MOV ES, AX;
MOV word [ES:0], 0X0741;
IRET
start:
XOR DI, DI;
MOV ES, DI
mov AX, ISR0;
MOV [ES:16h*4],AX;
MOV [ES:16h*4+2], CS;
mov ah,0;
int 0x16;
mov ax,0x4c00
int 21h
;Note: After executing this interrupt, the contents of IVT against
;int 0x16 has been overwritten so the keyboard will not work properly.
```

Example 8: Interrupt hooking without using INT instruction

```
[org 0x100]
jmp start
ISR0:
MOV AX, 0XB800
MOV ES, AX;
MOV word [ES:2], 0X0741;
IRET
start:
XOR DI, DI;
MOV ES, DI
mov AX, ISR0;
MOV [ES:17h*4],AX;
MOV [ES:17h*4+2], CS;
Pushf;push flag register
push cs ;push code segment
push continue   ;push IP (address of next instruction where to return)
jmp far [es:17h*4]  ;calling interrupt
continue:
mov ax,0x4c00
int 21h
```

Example 9: Interrupt unhooking.

```
[org 0x100]
jmp start
old_data: dd 0
ISR0:
MOV AX, 0XB800
MOV ES, AX;
MOV word [ES:0], 0X0741
mov ax,0
mov es,ax
mov bx,[old_data]
mov [ES:0x16*4],bx   ;saving the old values
;in a variable before overwritting.
mov bx,[old_data+2]
mov [ES:0x16*4+2],bx
IRET
```



```
start:
XOR DI, DI;
MOV ES, DI
mov AX, ISR0;
mov bx,[ES:0x16*4];saving the old values
;in a variable before overwriting.
mov [old_data],bx
mov bx,[ES:0x16*4+2]
mov [old_data+2],bx
MOV [ES:0x16*4],AX;hooking the interrupt
MOV [ES:0x16*4+2], CS
pushf
push cs
push continue
jmp far [es:0x16*4]
continue:
mov ax,0x4c00
int 21h
;Note: Recover the old contents of IVT
;after executing your functionality via hooking.
```