Computer Organization and Architecture

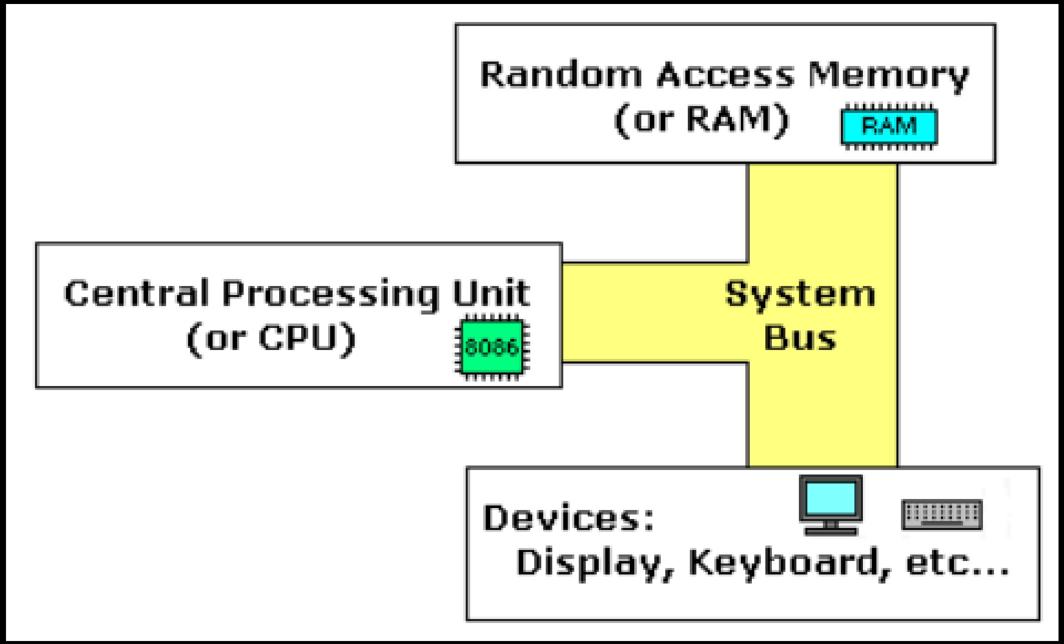
X86 Assembly

Content

Memory Access
Variables
Interrupts
Arithmetic and Logic Instructions
Program Flow Control
Procedures
Programs

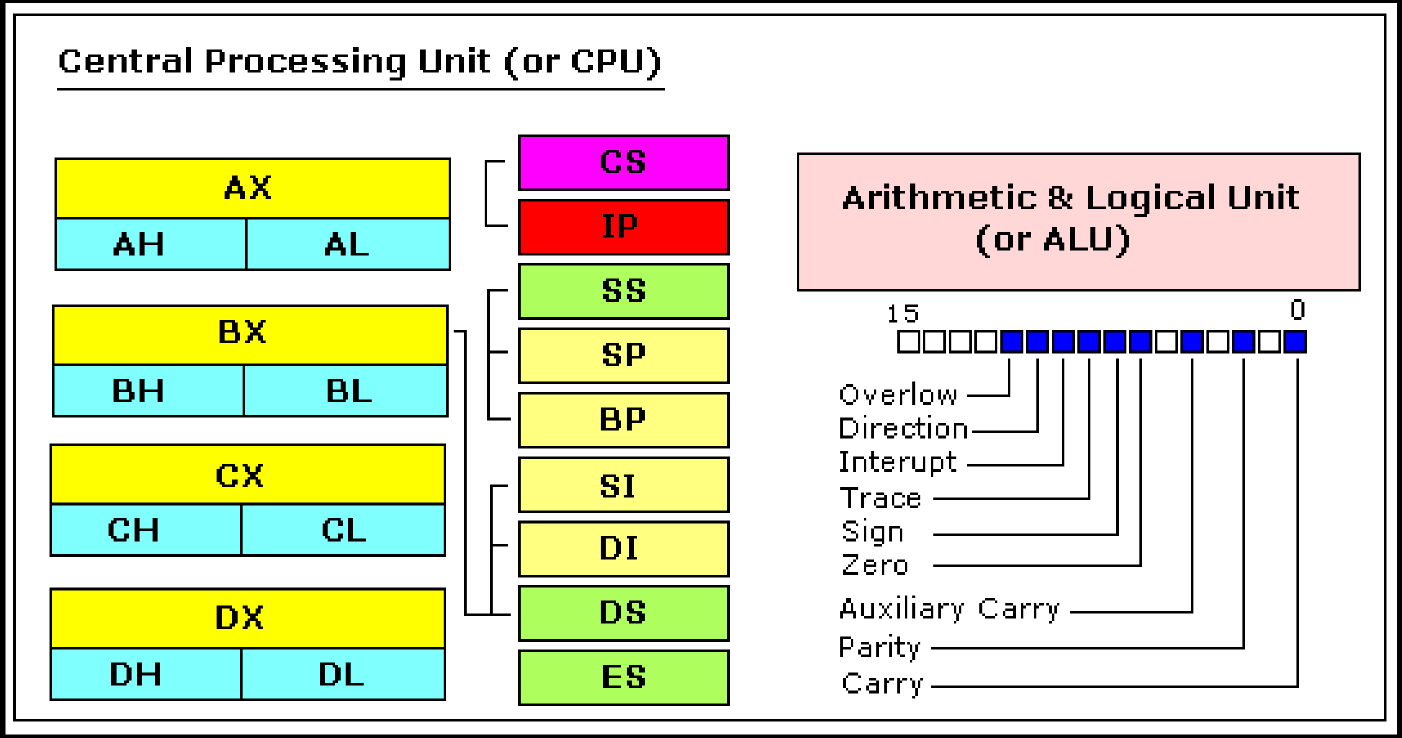
Memory Access

- The simple computer model is as follows
  - The system bus connects the various components of a computer.
  - The CPU is the heart of the computer, most of computations occur inside the CPU.
  - RAM is a place to where the programs are loaded in order to be executed.



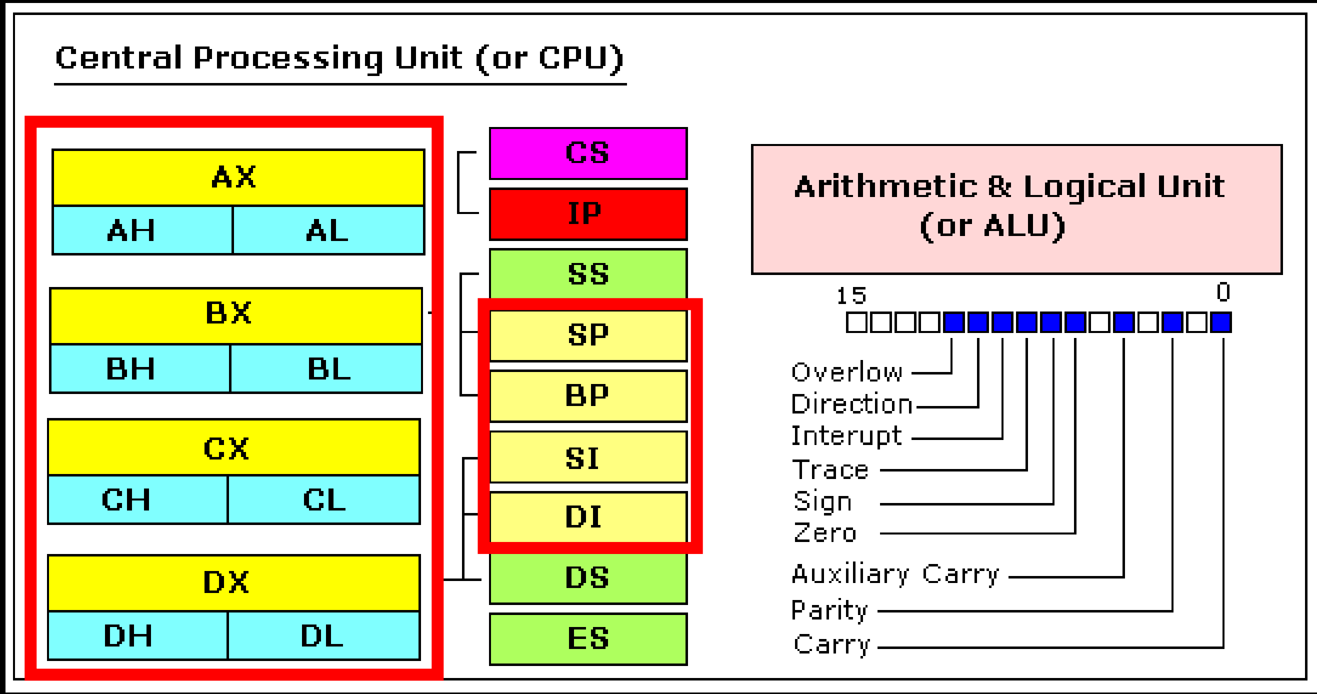
Memory Access

- Inside the CPU



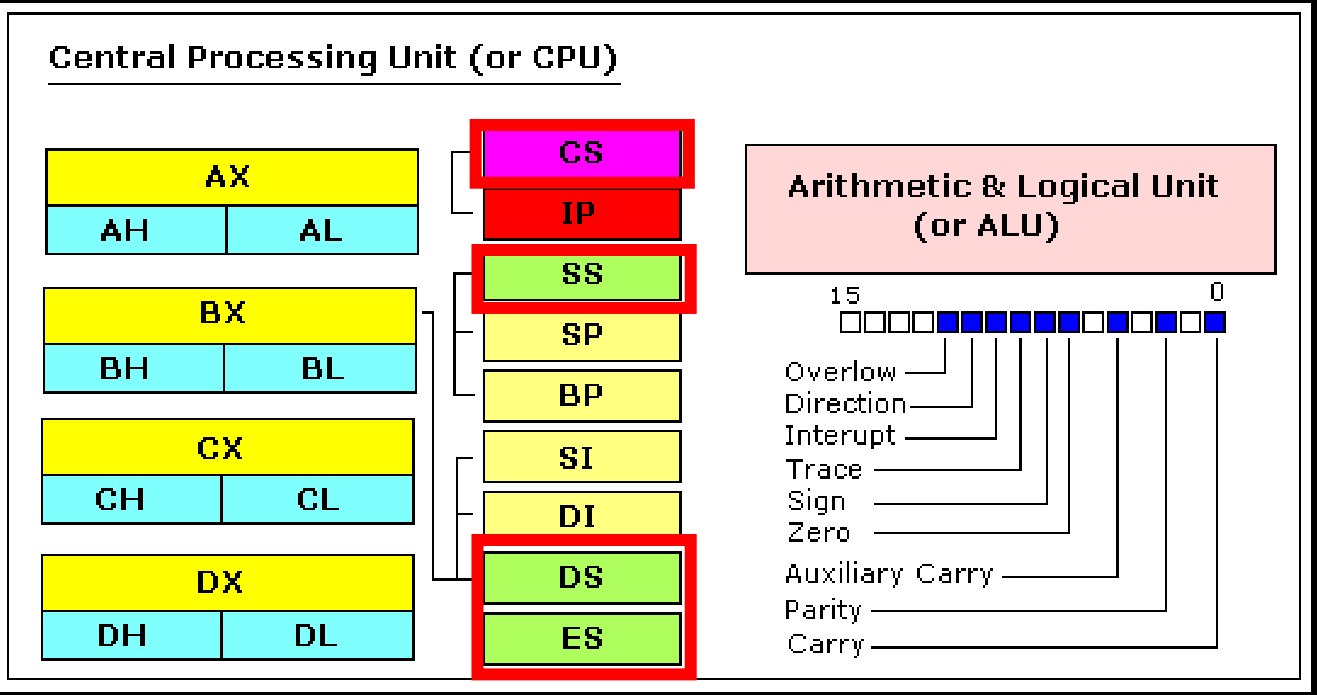
Memory Access

- General purpose registers
  - AX - the accumulator register.
  - BX - the base address register.
  - CX - the count register.
  - DX - the data register.
  - SI - source index register.
  - DI - destination index register.
  - BP - base pointer.
  - SP - stack pointer.



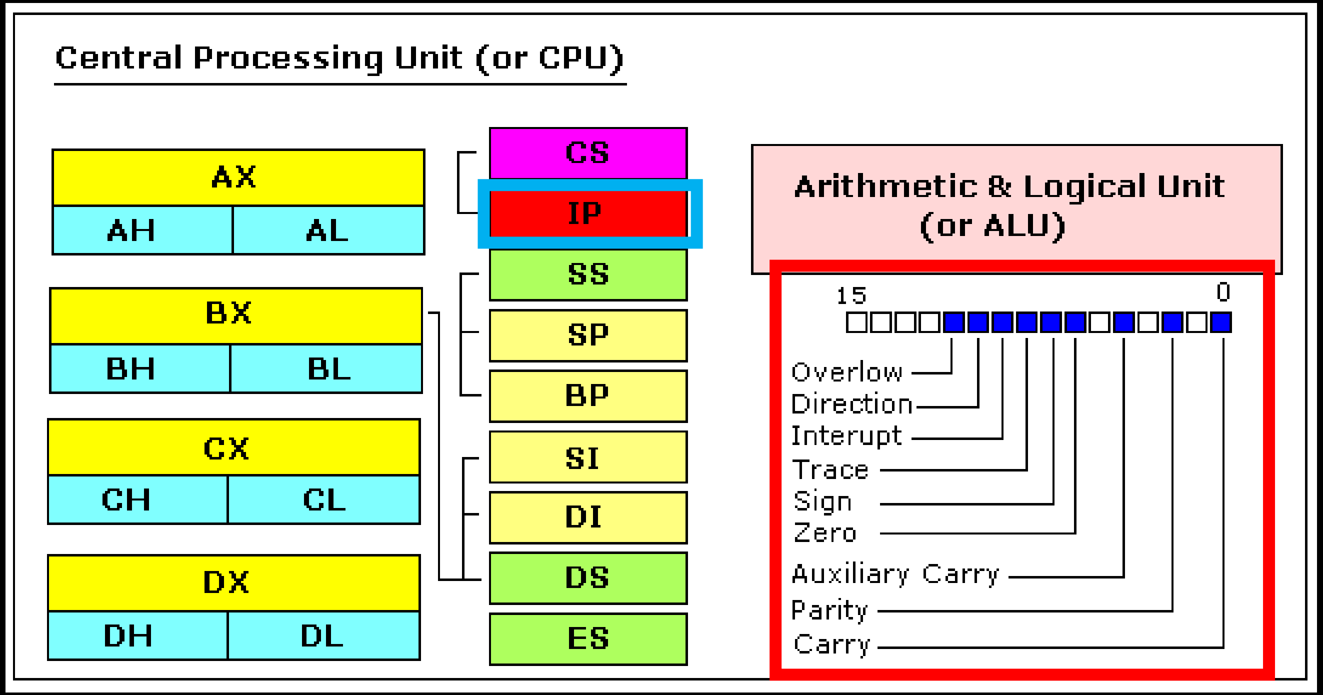
Memory Access

- segment registers
  - CS - points at the segment containing the current program.
  - DS - generally points at segment where variables are defined.
  - ES - extra segment register, it's up to a coder to define its usage.
  - SS - points at the segment containing the stack.



Memory Access

- special purpose registers
  - IP - the instruction pointer.
  - flags register - determines the current state of the microprocessor.
- IP register always works with CS register and it points to currently executing instruction.
- Flags register is modified automatically by CPU after mathematical operations.



Memory Access

- Segment registers work with general purpose register to access any memory value.
- For example, if we would like to access memory at the physical address 12345h, we should set the DS = 1230h and SI = 0045h.
  - Shift the DS to the left – DS = 12300h
  - Add the SI to the DS

$$\begin{array}{r} 12300 \\ + \\ 0045 \\ = 12345 \end{array}$$

- The address formed with 2 registers is called an effective address.

Content

Memory Access
Variables
Interrupts
Arithmetic and Logic Instructions
Program Flow Control
Procedures
Programs

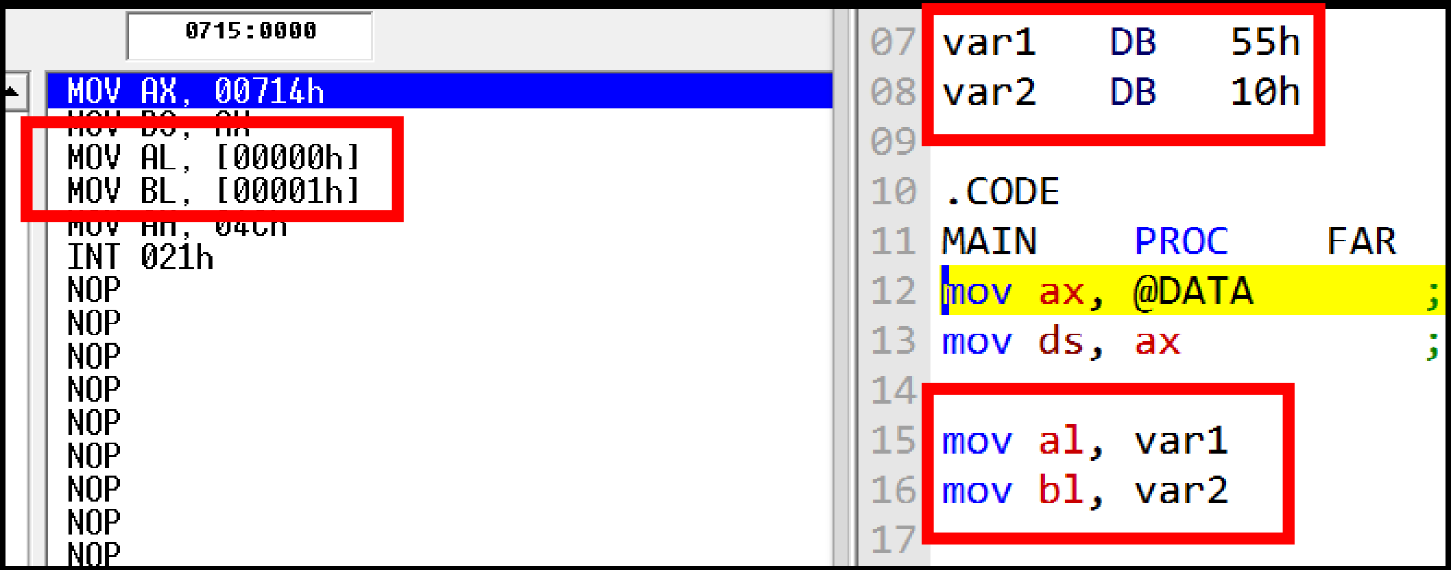
Variables

- Syntax for a variable declaration:

  name **DB** value

  name **DW** value

- When we define variables and use them in the code, the actual assembly code refers to their memory address in [ ] symbol.

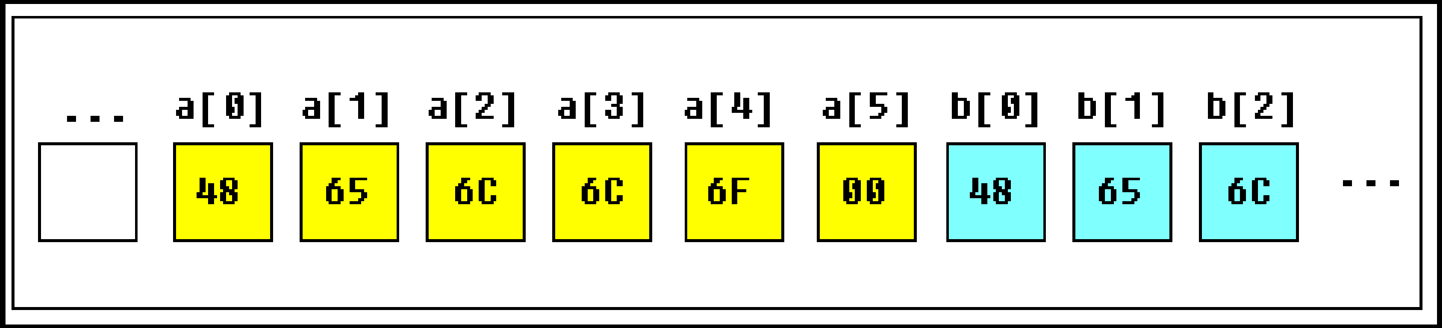


Variables

- Arrays can be seen as chains of variables.
    - A text string is an example of a byte array, each character is presented as an ASCII code value.
- Examples:

$$a \quad DB \quad 48h, 65h, 6Ch, 6Ch, 6Fh, 00h$$
$$b \quad DB \quad 'Hello', 0$$

- b is an exact copy of the a array ,when compiler sees a string inside quotes it automatically converts it to set of bytes.



Variables

- You can access the value of any element in array using square brackets, for example: MOV AL, a[3]

```
   .DATA
arr DB  55h, 12h, 11h, 10h

   .CODE
MAIN    PROC    FAR
mov ax, @DATA
mov ds, ax
;-------------------
   mov al, arr[0]
   mov bl, arr[3]
;-------------------
MOV AH,4CH
INT 21H
MAIN ENDP
    END    MAIN
```

Variables

- You can get the address of a variable using LEA or $OFFSET$ operators.
  - To know the difference between them see this.

```
   .DATA
arr     DB  55h, 12h, 11h, 10h
var1    DB  88h
   .CODE
MAIN    PROC    FAR
mov ax, @DATA
mov ds, ax
;----------------------
   lea bx, arr[2]
   lea ax, arr
   lea cx, var1
;----------------------
MOV AH,4CH
INT 21H
MAIN ENDP
    END    MAIN
```

```
   .DATA
arr     DB  55h, 12h, 11h, 10h
var1    DB  88h
   .CODE
MAIN    PROC    FAR
mov ax, @DATA
mov ds, ax
;----------------------
   mov bx, offset arr[2]
   mov ax, offset arr
   mov cx, offset var1
;----------------------
MOV AH,4CH
INT 21H
MAIN ENDP
    END    MAIN
```

Variables

- You can define constants using EQU operator: name **EQU** value
    - Constants cannot be changed.

```
   .DATA
k   EQU     5h
   .CODE
MAIN    PROC    FAR
mov ax, @DATA
mov ds, ax
;---------------------
   mov al, 3h
   mov [k], 1h
   add al, k
;---------------------
MOV AH,4CH
INT 21H
MAIN ENDP
    END    MAIN
```

The result will be 8h, because k is EQU

```
   .DATA
k   DB     5h
   .CODE
MAIN    PROC    FAR
mov ax, @DATA
mov ds, ax
;---------------------
   mov al, 3h
   mov [k], 1h
   add al, k
;---------------------
MOV AH,4CH
INT 21H
MAIN ENDP
    END    MAIN
```

The result will be 4h, because k is DB

Content

Memory Access
Variables
Interrupts
Arithmetic and Logic Instructions
Program Flow Control
Procedures
Programs

Interrupts

- Interrupts can be seen as functions to do a specific task.
- To make an interrupt: $INT\ value$
  - Where value can be a number between 0 to 255 (or 0 to 0FFh)
  - Each interrupt may have sub-functions. To specify a sub-function AH register should be set before calling interrupt.

Interrupts

- Example, write “Hello” to console.
  - Use interrupt $INT\ 10h\ /\ AH\ =\ 0Eh$
- The interrupt $INT\ 10h\ /\ AH = 0EH$ is a teletype output – move the cursor after print.
- The interrupt $INT\ 21h\ /\ AH = 2$ writes one character only.

```
MAIN    PROC    FAR
   mov ax, @DATA
   mov ds, ax

   mov ah, 0eh

   mov al, 'H'
   int 10h

   mov al, 'E'
   int 10h

   mov al, 'L'
   int 10h

   mov al, 'L'
   int 10h

   mov al, 'O'
   int 10h


   MOV AH,4CH
   INT 21H
MAIN ENDP
    END    MAIN
```

Interrupts

- Example, create a new folder named “mydir”.
  - Use the interrupt $INT\ 21h\ \ /\ \ AH = 39h$
  - Any String in the assembly must end by 0.
  - The emulator has virtual hard drives located at *c:\emu8086\vdrive*

```
   .MODEL SMALL
   .STACK 64
   .DATA

filepath DB "C:\mydir", 0  ; path to be created.

   .CODE
MAIN   PROC   FAR
mov ax, @DATA
mov ds, ax

    mov dx, offset filepath
    mov ah, 39h
    int 21h

MAIN ENDP
    END   MAIN
```

Content

Memory Access
Variables
Interrupts
Arithmetic and Logic Instructions
Program Flow Control
Procedures
Programs

Arithmetic and Logic Instructions

- There are 3 groups of instructions.

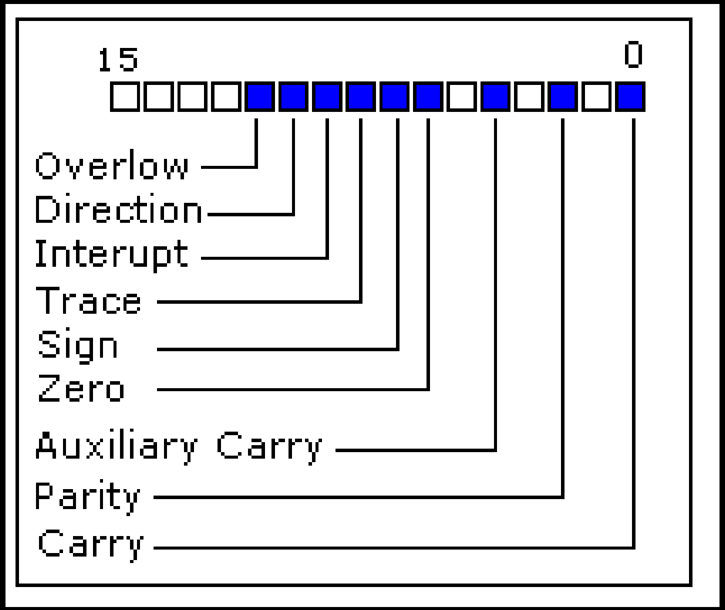
ADD	Add second operand to first.
SUB	Subtract second operand to first.
CMP	Subtract second operand from first for flags only.
AND	Logical AND between all bits of two operands.
OR	Logical OR between all bits of two operands.
XOR	Logical XOR between all bits of two operands.

MUL	Unsigned multiply
DIV	Unsigned divide

INC	Increment by 1
DEC	Decrement by 1
NOT	Reverse each bit of operand.
NEG	Make operand negative (two's complement).

Arithmetic and Logic Instructions

- Arithmetic and Logic Instructions affect the processor status register (Flags).
  - **Carry Flag (CF)** - set to 1 when there is an unsigned overflow. For example when you add bytes 255 + 1 (result is not in range 0...255).
  - **Zero Flag (ZF)** - set to 1 when result is zero.
  - **Sign Flag (SF)** - set to 1 when result is negative.
  - **Overflow Flag (OF)** - set to 1 when there is a signed overflow. For example, when you add bytes 100 + 50 (result is not in range -128...127).



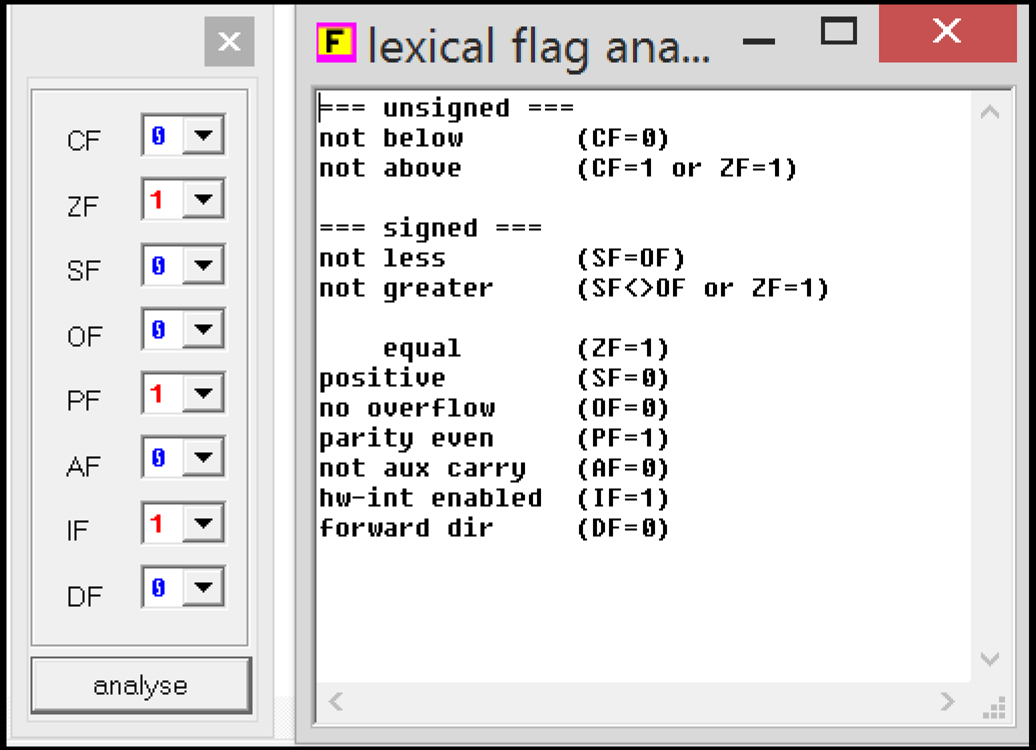
Arithmetic and Logic Instructions

- The CMP instruction subtract second operand from first operand and compare the operands by changing the flags only.
- Example

```
org 100h

mov al, 5
mov bl, 5
cmp al, bl

ret
```

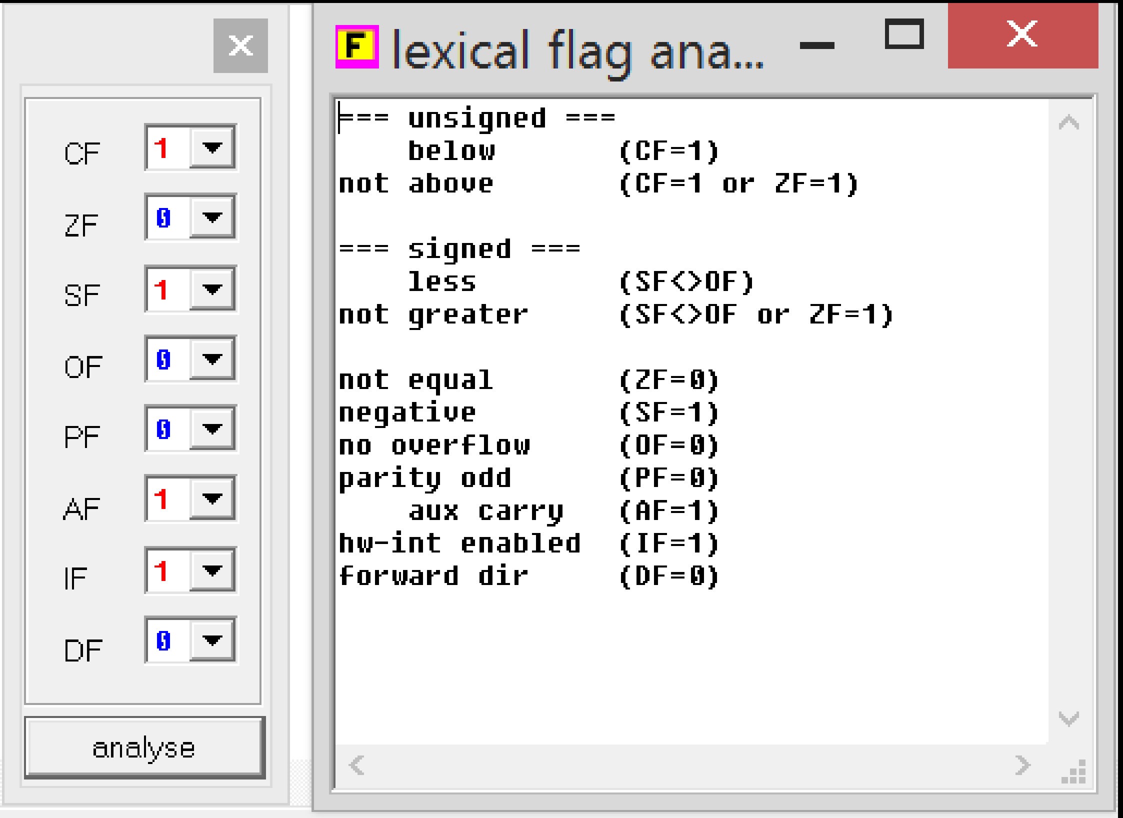


Arithmetic and Logic Instructions

```
org 100h

mov al, 3
mov bl, 5
cmp al, bl

ret
```



Content

Memory Access
Variables
Interrupts
Arithmetic and Logic Instructions
Program Flow Control
Procedures
Programs

Program Flow Control

- Unconditional jump is achieved via JMP instruction.

  JMP <u>label</u>

- To declare a label in your program, just type its name and add ":" to the end.
  - Labels cannot start with a number.
  - example

```
label1:
label2:
a:
```

Program Flow Control

- Example

```
MAIN    PROC    FAR
mov ax, @DATA
mov ds, ax
;--------------------------
   mov    ax, 5
   mov    bx, 2

   jmp    calc

 back:  jmp stop

 calc:
   add    ax, bx
   jmp    back

stop:
MOV AH,4CH
INT 21H
MAIN ENDP
    END    MAIN
```

Program Flow Control

- Conditional jumps are achieved via:
  - JZ, JE – Jump if Zero (Equal), exectued when the value of the ZF = 1.

    Algorithm:

    if ZF = 1 then jump

  - JNZ, JNE – Jump if not Zero (Not Equal), executed when the value of the ZF = 0.

    Algorithm:

    if ZF = 0 then jump

  - $LOOP$ - Decrease CX, jump to label if CX not zero.

    Algorithm:

    - CX = CX - 1
    - if CX <> 0 then
      - jump

      else
      - no jump, continue

Program Flow Control

In the program we wrote last time to compute the sum of the numbers.

When CX reaches 0, the ZF becomes 1, hence, the JNZ instruction is not executed.

```
        mov cx, 05
        mov bx, OFFSET DATA_IN
        mov al, 0
AGAIN:  add al, [bx]
        inc bx
        dec cx
        jnz AGAIN
        mov SUM, al
```

Thus, the JNZ instruction does not check the value of the CX, but the value of the ZF.

The ZF is changed when the CX becomes 0.

Program Flow Control

- JZ example, subtract 3 from AL.
  - **Check if the AL = 5.**
  - **If AL = 5, the ZF becomes 1.**
  - If AL != 5, the ZF stills 0.
  - **When ZF is 1, the JZ is executed, then subtract 3 from AL.**
  - When ZF is 0, the JZ is not executed, then add 3 to AL.
- *ZF = 1 means the two values are equal.*
- *ZF = 0 means the two values are different.*

```
   .CODE
MAIN   PROC   FAR
mov ax, @DATA
mov ds, ax
;-----------------------
   MOV AL, 5
   CMP AL, 5

   JZ  label1
   add al, 3
   JMP exit

   label1:
   sub al, 3
;-----------------------
exit:
MOV AH,4CH
INT 21H
MAIN ENDP
    END   MAIN
```

Program Flow Control

- JNZ example, add 3 to AL.
  - **Check if the AL = 5.**
  - If AL = 5, the ZF becomes 1.
  - **If AL != 5, the ZF stills 0.**
  - When ZF is 1, the JZ is executed, then subtract 3 from AL.
  - **When ZF is 0, the JZ is not executed, then add 3 to AL.**
- *ZF = 1 means the two values are equal.*
- *ZF = 0 means the two values are different.*

```
   .CODE
MAIN    PROC    FAR
mov ax, @DATA
mov ds, ax
;------------------------
   MOV AL, 5
   CMP AL, 5

   JNZ  label1
   add al, 3
   JMP exit

   label1:
   sub al, 3
;------------------------
exit:
MOV AH,4CH
INT 21H
MAIN ENDP
    END    MAIN
```

Program Flow Control

- LOOP example, increment AL by 1 five times.
  - By the end of the program, CX = 0 and AL = 6.

```
   .CODE
MAIN   PROC   FAR
mov ax, @DATA
mov ds, ax
;------------------------
   MOV cl, 5
   mov al, 1

sum_1:
   inc al
   LOOP sum_1
;------------------------
exit:
MOV AH,4CH
INT 21H
MAIN ENDP
    END   MAIN
```

Content

Memory Access
Variables
Interrupts
Arithmetic and Logic Instructions
Program Flow Control
Procedures
Programs

Procedures

- Procedure is a part of code that can be called to make some specific task.

The syntax for procedure declaration:

```
name PROC

      ; here goes the code
      ; of the procedure ...

name ENDP
```

- name is the procedure name, the same name should be in the top and the bottom, this is used to check correct closing of procedures.
- RET instruction is used to return to caller.

Procedures

- The program calls the sum procedure to compute AL + CL.
- Then calls subtract procedure to compute AL – CL.
- Notice that the sum and subtract are defined after $MAIN\ ENPD$ and before $END\ MAIN$.
- We can ignore the NEAR directive.

```
MAIN    PROC    FAR
mov ax, @DATA
mov ds, ax
;-------------------------------------------------------;
   mov cl, 5
   mov al, 8

   call sum
   call subtract

;-------------------------------------------------------;
MOV AH,4CH
INT 21H
MAIN ENDP

sum     PROC    NEAR
   add al, cl
   RET
sum     ENDP

subtract    PROC    NEAR
   sub al, cl
   RET
subtract    ENDP

    END    MAIN
```

Content

Memory Access
Variables
Interrupts
Arithmetic and Logic Instructions
Program Flow Control
Procedures
Programs

Programs

- Calculate the sum of the numbers from 1 to 15.

```
   .DATA
result  DW  ?

   .CODE
MAIN   PROC   FAR
mov ax, @DATA
mov ds, ax
;--------------------
   mov ax, 0d
   mov bx, 1d
   mov cx, 50d

sum:
   add ax, bx
   inc bl
   LOOP sum

mov result, ax
;---------------------
MOV AH,4CH
INT 21H
MAIN ENDP
    END   MAIN
```

Programs

- Print ‘*’ 100 times

```
  .DATA
  .CODE
MAIN   PROC   FAR
mov ax, @DATA
mov ds, ax
;----------------------
  mov cx, 100

  prnt:
    mov ax, '*'
    mov ah, 0Eh
    int 10h
    LOOP prnt



;----------------------
MOV AH,4CH
INT 21H
MAIN ENDP
   END   MAIN
```

Programs

- Print the following pattern

```
*       *
 *     *
  *   *
   * *
    *
```

Programs

Algorithm

1. Define a variable to maintain line break: $new\_line\ db\ 13, 10, "\$“$
2. Define a variable to hold number of left spaces: $crnt\_dl\ \ db\ ?$
3. Define a variable to hold number of middle spaces: $crnt\_bl\ db\ ?$
4. Set a counter to left spaces: $mov\ dl, 0$
5. Set a counter to right spaces: $mov\ bl, 10$
6. Set a counter to the number of lines: $mov\ cx, 6$
7. Move the value of the DL to its variable: $mov\ crnt\_dl\ , dl$
8. Move the value of the BL to its variable: $mov\ crnt\_bl,\ bl$

Programs

Algorithm

9. If DL != 0:
    a. print left spaces
    b. decrement DL
    c. repeat until DL = 0
10. If DL = 0
    a. print a star
11. If BL != 1
    a. print a space
    b. decrement DL
    c repeat until BL=1
12. Print a star
13. Print a new line
14. Update counters:
    a. increment crnt_DL
    b. move crnt_DL to DL
    c. decrement BL twice
    d. move crnt_BL to BL
15. Go to step 1 until CX = 0

Instructions Summary

- mov
- add
- sub
- xor
- int
- inc
- dec
- jnz
- mul
- div
- or
- lea
- $test$
- neg
- $call$
- $loop$

TASK

- Explain the behavior of the following program.

```
org 100h

mov al, 00100010b
mov bl, 00001111b
cmp al, bl

ret
```

- What is $TEST$ instruction used for?