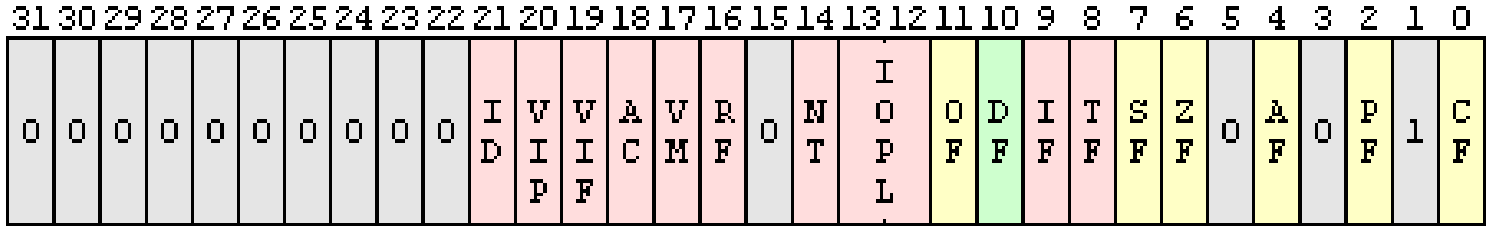
Computer Architecture and Assembly Language

Practical Session 2

EFLAGS - Flags Register (Status Register)

- **flag** is a single independent bit of (status) information
- each flag has a **two-letter symbol name**



Status Flags

CF = Carry flag

PF = Parity flag

AF = Auxiliary carry flag

ZF = Zero flag

SF = Sign flag

OF = Overflow flag

Control flag

DF = Direction flag

System flags

TF = Trap flag
IF = Interrupt flag
IOPL = I/O privilege level
NT = Nested task
RF = Resume flag
VM = Virtual 8086 mode
AC = Alignment check
VIF = Virtual interrupt flag
VIP = Virtual interrupt pending
ID = ID flag

CF - Carry Flag

CF gets ‘1’ if **result is larger than "capacity" of target operand**

$11111111_b + 00000001_b = 100000000_b = 0 \rightarrow CF = 1$

CF gets ‘1’ if **subtract larger number from smaller** (borrow out of “capacity” of target operand)

$00000000_b - 00000001_b = 11111111_b \rightarrow CF = 1$

$00000000_b - 00000001_b \equiv 100000000_b - 00000001_b \equiv 11111111_b$

otherwise CF gets ‘0’

- **In unsigned arithmetic, watch the carry flag to detect errors**.
- In signed arithmetic ignore CF flag.

$00000000_b - 00000001_b = 11111111$

unsigned arithmetic → 11111111_b = 255 (decimal) → got the wrong answer
signed arithmetic → 11111111_b = -1 (decimal) → ignore CF flag

OF – Overflow Flag

sign-bit-off operands + sign-bit-on result → OF gets ‘1’

$\mathbf{0}1000000_b + \mathbf{0}1000000_b = 10000000_b$ → OF = 1

sign-bit-on operands + sign-bit-off result → OF gets ‘1’

$\mathbf{1}0000000_b + \mathbf{1}0000000_b = 100000000_b = 0$ → OF = 1

otherwise OF gets ‘0’

$01000000_b + 00010000_b = 01010000_b$ → OF = 0
$01100000_b + 10010000_b = 11110000_b$ → OF = 0
$10000000_b + 00010000_b = 10010000_b$ → OF = 0
$11000000_b + 11000000_b = 10000000_b$ → OF = 0

- **In signed arithmetic, watch the overflow flag to detect errors** – addition of two positive numbers results negative, or addition two negative numbers results positive.
- In unsigned arithmetic ignore OF flag.

ZF, SF, PF, and AF Flags

ZF – **Zero Flag - s**et if a result is zero; cleared otherwise

add 0, 0 → ZF = 1

SF – **Sign Flag – set if a result is negative** (i.e. MSB of the result is 1)

sub 00000000_b, 00000001_b → 11111111_b → SF = 1

PF – **Parity Flag - s**et if low-order eight bits of result contain an even number of "1" bits; cleared otherwise

add 11010000_b, 1000_b (result is 11011000_b → 4 bits are '1' → PF = 1)
add 11000000_b, 1000_b (result is 11001000_b → 3 bits are '1' → PF = 0)

AF – **Auxiliary Carry Flag** (or **Adjust Flag**) is set if there is a carry from low nibble (4 bits) to high nibble or a borrow from a high nibble to low nibble

$1111_b + 0001_b = 10000_b$ → AF = 1 (first nibble, second nibble)

$0000_b - 0001_b = 10000_b - 0001_b = 1111_b$ → AF = 1

Jcc: Conditional Branch

Instruction	Description	Flags
JO	Jump if overflow	OF = 1
JNO	Jump if not overflow	OF = 0
JS	Jump if sign	SF = 1
JNS	Jump if not sign	SF = 0
JE **JZ**	Jump if equal Jump if zero	ZF = 1
JNE **JNZ**	Jump if not equal Jump if not zero	ZF = 0
JB **JNAE** **JC**	Jump if below Jump if not above or equal Jump if carry	CF = 1
JNB **JAE** **JNC**	Jump if not below Jump if above or equal Jump if not carry	CF = 0
JBE **JNA**	Jump if below or equal Jump if not above	CF = 1 or ZF = 1
JA **JNBE**	Jump if above Jump if not below or equal	CF = 0 and ZF = 0
JL **JNGE**	Jump if less Jump if not greater or equal	SF <> OF
JGE **JNL**	Jump if greater or equal Jump if not less	SF = OF
JLE **JNG**	Jump if less or equal Jump if not greater	ZF = 1 or SF <> OF
JG **JNLE**	Jump if greater Jump if not less or equal	ZF = 0 and SF = OF
JP **JPE**	Jump if parity Jump if parity even	PF = 1
JNP **JPO**	Jump if not parity Jump if parity odd	PF = 0
JCXZ **JECXZ**	Jump if CX register is 0 Jump if ECX register is 0	CX = 0 ECX = 0

Instructions seen till now – affecting flags

MOV NOT JMP*

does not affect flags

NEG

The CF flag set to 0 if the source operand is 0; otherwise it is set to 1. The OF, SF, ZF, AF, and PF flags are set according to the result.

AND OR

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined

DEC INC

The CF flag is not affected. The OF, SF, ZF, AF, and PF flags are set according to the result

ADD ADC SUB SBB

The OF, SF, ZF, AF, PF, and CF flags are set according to the result.

CMP

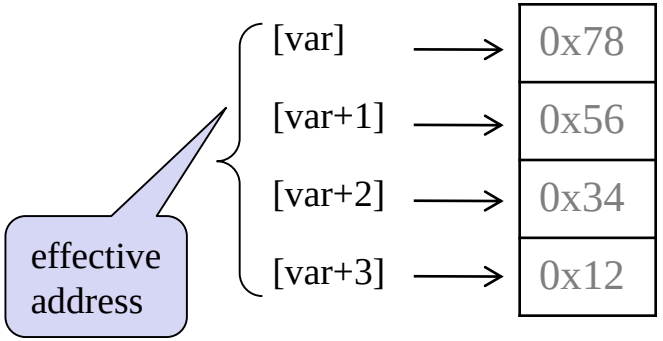
Modify status flags in the same manner as the SUB instruction

The full set of instructions can be found here.

Byte Order - Little Endian

Little Endian - stores the least significant byte in the smallest address.

numeric into memory → reversed order
```
var:  dd   0x12345678 ; 0x78 0x56 0x34 0x12
```
numeric into register → source order
```
mov EAX, 0x12345678 ; 0x12 0x34 0x56 0x78
```



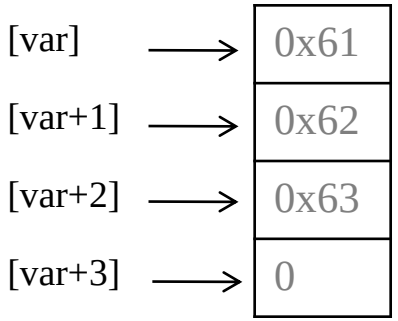
characters into memory → source order
```
var: dw  'abc'   ; 0x61 0x62 0x63 0x00
```
characters into register → reversed order
```
mov eax, 'abc'   ; 0x00 0x63 0x62 0x61
```

EAX register

0x12	0x34	0x56	0x78

register into memory → reversed order
```
mov [buffer], eax ; 0x61 0x62 0x63 0x00
```

memory into register → reversed order
```
mov eax, [buffer] ; 0x00 0x63 0x62 0x61
```



EAX register

0	0x63	0x62	0x61

Sections

Every process consists of sections that are accessible to the process during its execution.

Data

.bss - holds uninitialized data; occupies no space

.data and .data1 - holds initialized data

.rodata and .rodata1 - holds read-only data

Text

.text – holds executable instructions of a program

Stack – contains information + local variables that is saved for each function call

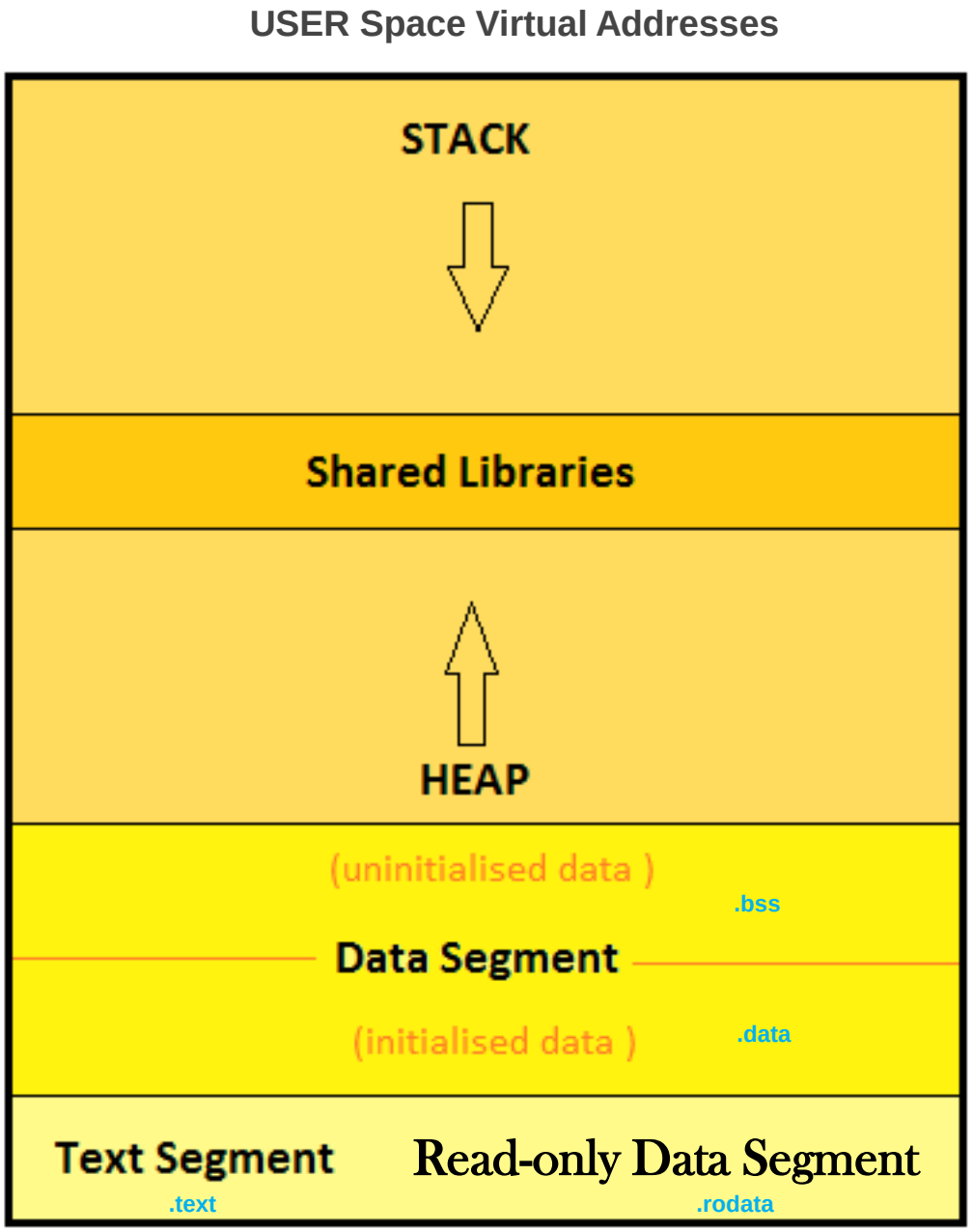
Heap – contains dynamically allocated memory

Examples :

```
char a[10];                    ----> .bss
int a = 1;                     ----> .data
const int i = 10;              ----> .rodata
main()
{
    int aL=1, cL;              ----> local variables on Stack
    char * ptr = malloc(4);    ----> allocated memory in Heap
    cL= a+aL+i;                ----> .text
}
```

```
section .bss
    BUFFLEN equ 1024
    buff: resb BUFFLEN

section .rodata
    LC0: db 'Hello world!'

section .data

    an: dd 0

section .text
    mov ax, 2
    ...
```

Memory layout for Linux



program (.exe file) creates its own memory space in the RAM → **process**

Pseudo-instructions - **RES<size>** - declare uninitialized storage space

Examples:

```
buffer:     resb 64          ; reserve 64 bytes
wordVar:    resw 1           ; reserve a word
realArray:  resq 10          ; array of ten quadwords (8 bytes)
```

Note: you can **not make any assumption** about content of a storage space cells.

Pseudo-instructions - **EQU** - define constant

Example:

```
foo:  EQU   1   ; foo = 1
```

Note: the value of foo **cannot be changed**.

Pseudo-instructions - **TIMES** - repeat (instruction or data)

TIMES prefix causes the instruction to be assembled multiple times

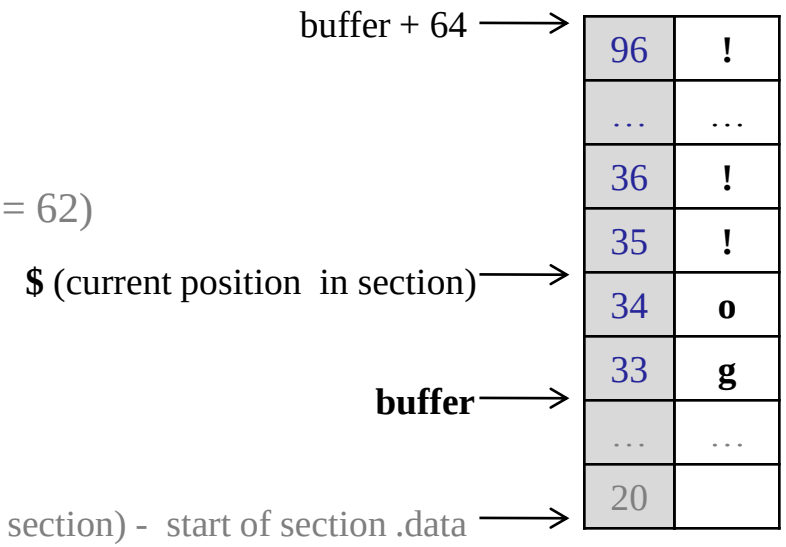
```
zeroBuf:   times 64 db 0      ; 64 bytes initialized to 0
```

TIMES can be applied to ordinary instructions, so you can code trivial loops

```
mov EAX, 0
times 100 inc EAX             ; EAX =100 => loop
```

the argument to **TIMES** is not just a numeric constant, but a numeric expression

```
buffer:   db 'go'
          times 64-$+buffer db '!'
          ; (64 - $ + buffer = 64 – 35 + 33 = 64 – 2 = 62)
```



Advanced Instructions - multiplication

MUL r/m - unsigned integer multiplication

IMUL r/m - signed integer multiplication

Multiplicand	Multiplier	Product
AL	*r/m8*	AX
AX	*r/m16*	DX:AX
EAX	*r/m32*	EDX:EAX

MUL r/m8

```
mov bl,5  ; multiplier
mov al,9  ; multiplicand
mul bl                ; ax ← 2D_h
```

MUL r/m16

```
mov bx, 8000_h
mov ax, 2000_h
mul bx    ; dx:ax ← 1000:0000_h
```

MUL r/m32

```
mov ebx, 80008000_h
mov eax, 20002000_h
mul ebx   ; edx:eax ← 10002000:10000000_h
```