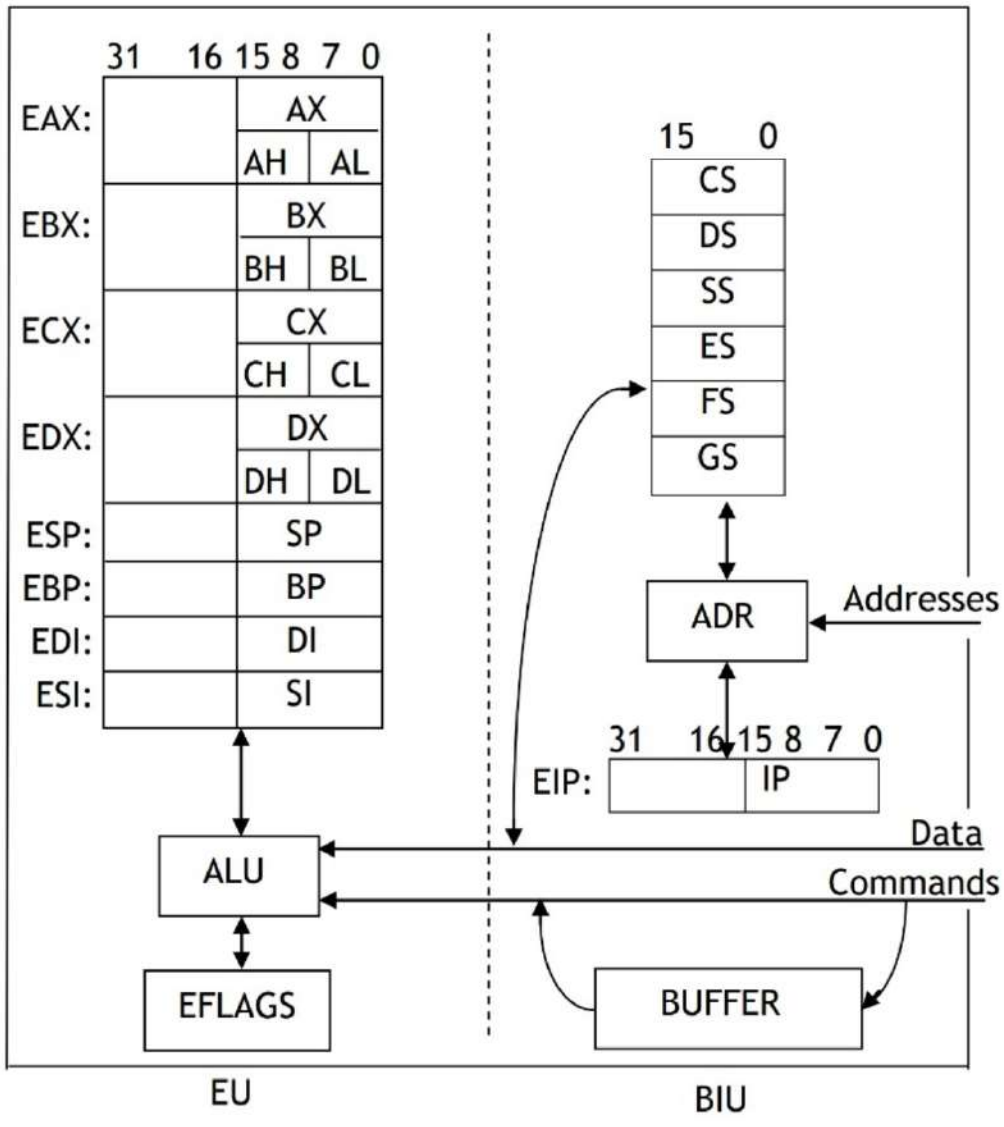
Notite ASC - curs + carte

Cuprins

I The x86 Microprocessor architecture



Registers

EAX - accumulator

EBX - base register

ECX - counter register - used mostly for repetitive loops

EDX - data register - mostly used with EAX when the result exceeds a doubleword

(folosit if we dd * dd -> dq)

\xlord - 16 bits
dxlord - 32 bits
qxword - 64 bits

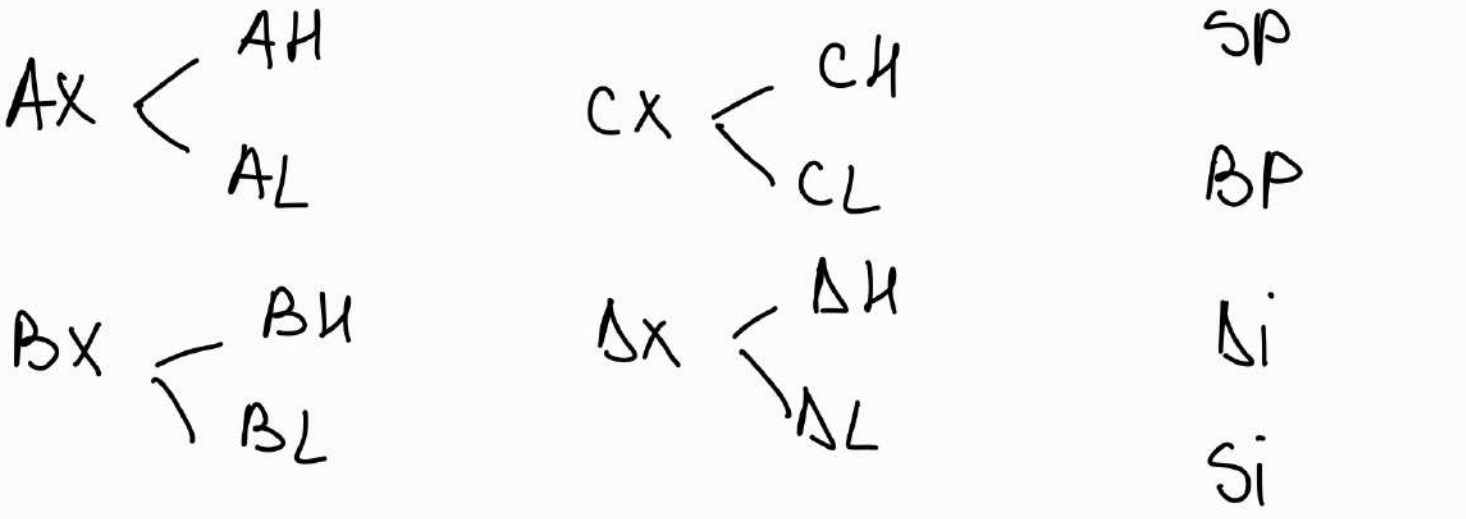
ESP - (stack pointer) - points to the last element put on the stack (top)

EBP - (base pointer) - points to the first elem. put on the stack (base

ESP + EBP = stack registers (LIFO memory area)

EDi - dustination index } used for accesing
ESi - source index } element =>
index registers

! All of them are dxlord registers (32 Bits)
=> ther are also 16 bits registers which contains (each) other 2 registers that are 8 bits



a[1042] = *(a + 1042) => (x86 add pointers)

adresa — baza — index (deplasament)

x = x + 1 → continutul lui x din memorie

adresa sa

Flags

CF - (carry flag) = the transport flag

set to 1 = (unsigned overflow) când este o cifră în plus (în afara dom. de repr.)

set to 0 = otherwise

1001 0011 +	147 +
0111 0011	115
1 0000 0110	262

→ carry digit => CF = 1

93h +	-109 +
73h	115
106h	06

$93h = (10010011)_2$ { 147 unsigned; -109 signed

PF - (parity flag) - if the representation of the last significant byte has an odd number of 1's => PF = 0, otherwise PF = 1 (nr. par de 1)

AF - (auxiliary flag) - dacă există a transport carry de la bitul 3 la bitul 4

ZF - (zero flag) -

```
  0000 0101
  0000 0001
  ---------
  0000 0110   => non-zero result
```

=> ZF = 0

```
  0000 0011
  1111 1101   => zero result
  ---------
  0000 0000
```

ZF = 1

SF - (sign flag) - the most significant bit is 1 => SF = 1, otherwise SF = 0

```
 1000   0000
 1111   1111
-------------
 0111   1111
```

=> SF = 0

TF (trap flag) - debugging flag

IF (interrupt flag)

DF (direction flag) - for operating string functions

set to 0 - string parsing in ascending order

set to 1 - string parsing in descending order

OF (overflow flag) - signed overflow if the result (signed interpretation) didn't

fit the reserved space OF = 1 otherwise OF = 0

Flags categories

a) set as direct effect of the LPO (last performed operation) - CF, PF, AF, ZF, SF, OF

b) flags set by the programmes to influence the next instructions - CF, DF, TF, IF

⇓

this are set by **special** instructions

CLC - CF = 0 STC - CF = 1 CMC - complements the value of CF

CLD - DF = 0 STD - DF = 1

CLI - IF = 0 STI - IF = 1 => nu merge pe 32 biți programming

Flag = indicator represented on one bit. A configuration of the flag briefing for every instructions execution

Flag register = 16 bits, but only 9 are used

II Operations performed by ALU

ALU = arithmetic logic unit

1. Arithmetic operators (*, :)
2. bit-shifting operators
3. logical operations (AND, OR, xOR, NOT)

→ it sets the flags

CF = 1

```
  1001 0011 +
  0111 0011
 ------------
1 0000 0110
```

↘ SF = 0

nibble = semioctet

- A representation in base 2 has always two possible interpretation in base 10

p byte : signed [-128, 127]

unsigned [0, 127]

III Signed and unsigned representation

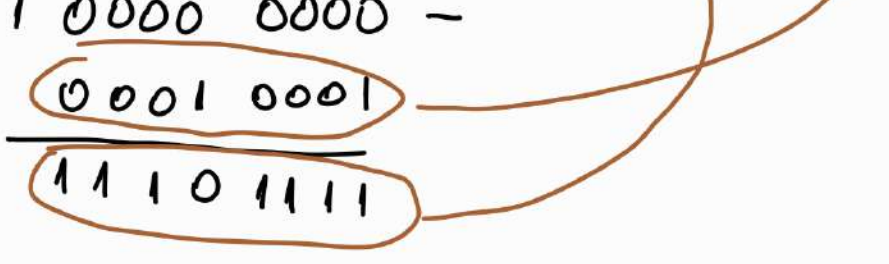
unsigned : can represent only positive numbers

the unsigned repres. of a pos. number = the number in base 2

signed : can represent positive and negative

17 on 8 bits : 0001 0001

-17 on 8 bits : 1110 1111



Interpretation

unsigned :

$1110\ 1111 = 1\cdot 2^7 + 1\cdot 2^6 + 1\cdot 2^5 + 0\cdot 2^4 + 1\cdot 2^3 + 1\cdot 2^2 + 1\cdot 2 + 1\cdot 2^0$

$= 239$

signed :

(1)110 111(1) -> we keep this and inverse the rest

>> it is a negative number

(1)110 1111 = 0001 0001 $= 1\cdot 2^4 + 1\cdot 2^0 = 17$

=> -17

2's complement

"we interpret representations and represent interpretations"

8 bits $2^8 = 256$ => [0, 255] (uns byte) or [-128, 127] (sig byte)

16 bits $2^{16} = 65536$ => [0, 65535] (uns w) or [-32768, + 32767] (sig w)

1001 0011 => 0110 1101 = 109 = -109

93h = 1001 0011 0110 1101 = 109 = -109

- A number that starts with 0 in base 2 has the same value in base 10 in both signed and unsigned

The minimum of bits to represent

On "n" bits we may represent 2^n values : unsigned values $[0 \dots 2^n - 1]$

: signed values $[-2^{(n-1)}, 2^{(n-1)} - 1]$

on 2 bits => $[0, 3]$, $[-2, +1]$ } we can represent 3 on 2 bits, but not -3

=> -3 can be represented on 3 bits

IV Overflow concept

Overflow = situation / condition which expresses the fact that the result of the last operation performed didn't fit the reserved space for it or does not belong to the admissible representation interval for that size or that operation is a mathematical nonsense in that particular interpretation (signed or unsigned)

1001 0011 +	147 +	93h +	-109 +
1011 0011	179	B3h	- 77
1 0100 0110	**326**	1 46h	**- 186** !!!!

both OF = 1 CF = 1

OF - adding two positive numbers and getting a negative result
- adding two negative numbers and

getting a positive result

- when an operation exceeds the signed number range

```
 1001  0011 +
 0111  0011
-----------
10000  0110
```

$1001\ 0011 = 1\cdot2^7 + 1\cdot2^4 + 1\cdot2 + 1\cdot2^0 = 147$

$\Rightarrow$

$0110\ 1101 = 1\cdot2^6 + 1\cdot2^5 + 1\cdot2^3 + 1\cdot2^2 + 1\cdot2^0 = -109$

$0111\ 0011 = 1\cdot2^6 + 1\cdot2^5 + 1\cdot2^4 + 1\cdot2 + 1\cdot2^0 = 115$

$\Rightarrow$ signed = unsigned 115

CF is set to 1

OF is set to 0 (0000 0110 = 6 which fits on $[-128, 127]$)

```
 0110  0010 -   (98)
 1100  1000     (200)
-------------
 1001  1010
```

1001 1010 = 154 unsigned

= -102 signed

0110 0010 = 98 unsigned

= 98 signed

1100 1000 = 200 unsigned

= -56 signed

unsigned

98 -

200

-102 => not ok => CF = 1

it is diff. from 154

signed

98 -

(-56)

154 => not ok => OF = 1

it is diff. from -102

The multiplication operand does not produce overflow => the space reserved is enough for both interpretations

Anyway, $b*b=b$, $w*w=w$, $d*d=d$ } $CF=OF=0$

$b*b=w$, $w*w=d$, $d*d=$ qword } $CF=OF=1$

! the worst effect is in the case of **division** : if the quotient (câtul) does not fit in the reserved space (xl/b=b, w = dd/w, qw/dd=dd) => division overflow will signal „Run-time error"

Why do we need CF and OF simultan?

=> when performing add / sub operation in base 2, in fact 2 operations are actually performed simultaneously In base 10: one in signed version and the other in unsigned

BUT

- unlike add and sub, which work the same in base 2 (regardless of the signed / unsigned), multiplication and division work differently in the signed and unsigned case

Multiplication and division do not work the same binary in the 2 interpretations => this is why we need to specify before

the operation how we want the operands to be interpreted

=> mul / div (unsigned)

imul / idiv (signed)

also, we have ADC (add with carry) that is used when we want to add things bigger than 8 bits

- the same for sbb (substract with borrow)

V Address registers
Address computation

- Address of a memory location - nr. of consecutive bytes from the beginning of the RAM memory and the beginning of that memory location

- segment - a logical section of a program's memory (featured by its basic address (beginning), by its limit (size) and by its type)

Segment
- block of memory (physical segment)
- variable-sized block of memory (logical segment) occupied by a program's code or data

offset = the number of bytes between the beginning of that segment and that particularly memory location

segment selector = numeric value of 16 bits which selects uniquely the accessed segment and his features

segment : offset

16 bits 32 bits

programming → segment selector issued by OS

→ the starting address of that segment

linear address

Based on such a specification the actual segmentation address computation will be performed as:

$a_7a_6a_5a_4a_3a_2a_1a_0 := b_7b_6b_5b_4b_3b_2b_1b_0 + 0_70_60_50_40_30_20_10_0$

computed address in hexa — base address — address specification

An address specification is also named FAR address

The x86 processors have a memory access control mechanism called paging, which is independent of address segmentation.

Paging implies dividing the virtual memory into pages, which are associated (translated) to the physical memory

The x86 architecture allows 4 types of segments:

address registers
segment registers
- CS - code segment
- DS - data segment
- SS - stack segment
- ES - extra segment
- FS
- GS
- EIP

} logical segments

the needed information for identifing the current active segment selectos

pointer arithmetic

a[7] = * (a + 7)

↓

dereferencing operator

- FS, GS = extended, the appeared on the 32 bits programming
- EIP = extension of IP (instruction pointer)

CS:EIP → the address of the current executing instruction

- size of an instruction = 1-15 bits

At any given moment during run time there is only one active

segment of any type

Far addresses and Near addresses

- to address a RAM memory location two values are needed
  1) to indicate the segment
  2) to indicate the offset inside the segment

Near address = an address for which only the offset is specified (the segment address being implicitly taken from a segment register)

! always inside one of the 4 active segments

Far address = an address for which the programmer explicitly specifies a

segment selector and it can be specified in one of the following 3 ways - offset specification : $S_3 S_2 S_1 S_0$

- segment register : CS, DS, SS, ES FS or GS

- FAR [variable], where variable is of type qword and contains the 6 bytes representing the FAR address

! There are 3 ways of expressing an operand :

i) register mode : mov eax, 17

↳ specified in register mode

ii) immediate mode : mov eax, 17

giving you right now the constant

iii) memory addresing mode: mov eax, [a+4]
(every variable is in memory, but not everything from the memory is a variable)

OFFSET ADDRESS

[base] + [index * scale] + [constant]

- [base]: 1 of the 8 registers
- [index * scale]: one of the 7 registers (cannot use ESP)
- scale: a constant
- [constant]: direct addressing mode

the content of one of the registers EAX, EBX, ECX, EDX, EBP, ESI, EDI or ESP as **base**;
the content of one of the registers EAX, EBX, ECX, EDX, EBP, ESI or EDI as **index**;
scale to multiply the value of the index register with 1, 2, 4 or 8;
the value of a numeric constant, on a byte, word or on a doubleword.

[ ] - optionality (not all 3 of them are needed)

Modes to address the memory

- direct addressing -> only the constant is present

indirect addressing

- based addressing -> if there is on of the base registes
- scale-indexed addressing -> if there is one of the index registes present

indirect addressing <=> we have at last one registe between []

jmp instructions => relative addressing

indicates the position of the next instruction to be run relative to the current position

VII Pointers arithmetic

= the set of arithmetic operations allowed to be performed with pointers, this meaning using arithmetic expressions which have addresses as operands

Only 3 operations

① substracting two addresses

address - address = OK (g - p = substs of 2 pointers) => scalar value

② adding a numerical value to a pointer => a pointer

address + numerical constant (identification of an element by indexing - a[7]) g + 9

③ substracting a numerical constant from a pointer => a pointer

address - numerical constant

var - this is an address

[var] - is its contents

L-value vs R-value. LHS vs RHS of an assignment

i := i + 1

- i → LHS → L-value → ⇓ the address
- i → RHS → R-value → ⇓ the contents from the address i

mov eax, [esp*5] - syntax error
↳ cannot be index

mov eax, ebx ok!

mov eax, [ebx * 3] ok!

mov eax, [ebx + ebx * 3]

mov eax, [ebp + 7] - syntax errors

VII Assembly language basics

an assembler => generates bytes

an assembler works with:

labels = user-defined names for pointing to data or memory areas

instructions = mnemonics which suggests the underlying action

directives = indications given to the assembler for correctly generating the corresponding bytes

$ location counter - the size of an array = an offset

$ - $$ = the distance from the beginning of the segment as a scalar = current size of section

$ = an offset = address = pointer type address of here

$$ = an offset = address = pointer type
address of start of current section

Source line format

[label [:]] [prefixes] [mnemonic] [operands] [; comments]

ex: jmp here ; label + mnemonic + operand + comm
repz cmpsd ; prefix + mnemonic + comment

Categories of labels

1) **code labels** => at the level of instructions sequence for defining the destinations of the control transfers during a program execution

2) **data labels** => symbolic identification for some memory location

[v] => the value of the variable
v => the address of the variable

lx: mov EAX, et => the address (offset)

mov EAX, [et] => the content from the address et

lea eax, [v] => the address (offset)

mnemonics - instruction names
- directives names

- instruction - guides the processor
- directives - guides the assembler

Expressions

expression - operands + operators

Operands specification modes

1) immediate operands -> the value is computed at assembly time
2) memory operands -> the value is computed at loading time

3) registers operands -> the value is computed at run-time

① immediate operands = constant numeric
- binary, octal, decimal, hexa values

② register operands
- direct using mov eax, ebx
- indirect usage and addressing mov eax, [ebx]

③ memory addressing operands
direct addressing operand = constant or symbol representing the address
(labels, procedures names, value of the loc counter)

- its offset is computed at assembly time
- the effective address always refers to a segment register

CS - for code labels
SS - when using EBP or ESP as base

DS - for the rest of data accesses

indirect addressing operands -> use registers for pointing to memory address

[base_register + index_register * scale + constant]

exemple for DS, SS

```
mov eax, [ebx]        DS:[ebx]
mov eax, [ebp]        SS:[ebp]
mov eax, [ebp*2] <=> mov eax, [ebp+ebp] SS
mov eax, [ebp*3] <=> mov eax, [ebp+ebp*3] SS
mov eax, [ebp*4]  DS  (ebp nu mai e baza)

mov eax, [ebx+esp]   } SS
mov eax, [esp+ebx]   }

mov eax, [ebx+esp*2]   DS
          L> base
mov eax, [ebx+ebp]     DS
mov eax, [ebp+ebx]     SS
```

Using operators

operators = used for combining, comparing, modifying and analyzing the operands

- + - performs addition at assembly time
- ADD - performs addition during run-time

Bit-shifting operators

expression >> how many

expression << how many

```
mov ax, 01110111b << 3;  AX = 00000011 10111000b
add bh, 01110111b >> 3;  the source operator is 00001110b
```

Bitwise operators

OPERATOR	SYNTAX	MEANING
~	~ expresie	Bits complement
&	expr1 & expr2	Bitwise AND
|	expr1 | expr2	Bitwise OR
^	expr1 ^ expr2	Bitwise XOR

Examples (we assume that the expression is represented on a byte):

```
~11110000b                  ; output result is   00001111b
01010101b & 11110000b       ; output result is   01010000b
01010101b | 11110000b       ; output result is   11110101b
01010101b ^ 11110000b       ; output result is   10100101b
```

! – logical negation (similar with C) ; !0 = 1 ; !(anything different from zero) = 0

& - and

1 - 0 -> 0

0 - 1 -> 0

1 - 1 -> 1

0 - 0 -> 0

~ not

0 -> 1

1 -> 0

| or

1 - 0 ->

0 - 1 -> 1

1 - 1 -> 1

0 - 0 -> 0

^ xor

1 - 0 -> 1

0 - 1 -> 1

1 - 1 -> 0

0 - 0 -> 0

xor ax, ax => ax = 0

The segment specification operator

(:) perform the far address computation of a variable or label

segment : expression

ex: [ss : ebx + 4] offset relative to ss
[es : 082h] offset relative to es

Type operators

- they specify the type of some expressions or operands

byte [A]
dword [A]

byte / word / dword / qword specifiers have always only the task of clarifying an ambiguity

ex: mov [v], 0 => syntax error
- operation size not specified

examples where a data size specifier is needed

mov [mem], 12

(i) div [mem]

(i) mul [mem]

push [mem]

pop [mem]

Directives

1) the segment directive = allows targeting the bytes of code or data

- the numeric value assigned to the segment name is the segment address (32 bits)

- the optional arguments give to the link-editor and the assembles the necessary info. regarding the way in which segments must be loaded and combined in memory

the [type] allows selecting the usage mode of the segment

- code (or text) - the segment will contain code
- data (or bss) - data segment
- rdata - def. of constant data

Data definition derivatives

data definition = declaration + allocation

VIII Assembly language instructions

MOV *d,s*	<d> <-- <s> (b-b, w-w, d-d)
PUSH *s*	ESP = ESP - 4 and transfers („pushes") <s> in the stack (s – doubleword)
POP *d*	Eliminates („pops") the current element from the top of the stack and transfers it to d (d – doubleword) ; ESP = ESP + 4
XCHG *d,s*	<d> ↔ <s> ; s,d – have to be L-values !!!
[reg_segment] XLAT	AL ← < DS:[EBX+AL] > or AL ← < segment:[EBX+AL] >
CMOVcc d, s	<d> ← <s> if cc (conditional move) is true
PUSHA / PUSHAD	Pushes in the stack EAX, ECX, EDX, EBX, ESP, EBP, ESI and EDI
POPA / POPAD	Pops EDI, ESI, EBP, ESP, EBX, EDX, ECX and EAX from stack
PUSHF	Pushes EFlags in the stack
POPF	Pops the top of the stack and transfers it to Eflags
SETcc d	<d> ← 1 if cc is true, otherwise <d> ← 0 (byte set on condition code)

push s
pop d
} s,d must be dwords

```
push eax  ⇔   sub esp, 4        ; prepare (allocate) space in order to store the value
              mov [esp], eax    ; store the value in the allocated space

pop eax   ⇔   mov eax, [esp]    ; load in eax the value from the top of the stack
              add esp, 4        ; clear the location
```

XCHG op1, op2 => interchange variables

[reg-segment] XLAT

- daca in ebx ai sirul iti pune in al ebx+al

ex: lbx a b c d e f } =>
al 2

=> în al o să pună codul ascii al lui c

LEA gen_reg, content of mem_op

lea eax, [v] => loads the offset of v

<=> mov eax, v

pushf transfers all the flags on top of the stack

popf

Type conversion instructions

CBW	converts the byte from AL to the word in AX (sign extension)
CWD	converts the word from AX to the doubleword in DX:AX (sign extension)
CWDE	converts the word from AX to the doubleword in EAX (sign extension)
CDQ	converts the doubleword from EAX to the quadword in EDX:EAX (sign extension)
MOVZX d, s	loads in d (REGISTER !), which must be of size larger than s (reg/mem), the UNSIGNED contents of s (zero extension)
MOVSX d, s	load in d (REGISTER !), which must be of size larger than s (reg/mem), the SIGNED contents of s (sign extension) http://www.c-jump.com/CIS77/ASM/DataTypes/T77_0270_sext_example_movsx.htm !!!!!!!!!!!

Shifts and rotates

Bit shifting instructions can be classified in the following way:

- Logic shifting instructions
  - left - **SHL**
  - right - **SHR**
- Arithmetic shifting instructions
  - left - **SAL**
  - -right - **SAR**

Bit rotating instructions can be classifed in the following way:

- Rotating instructions without carry
  - left - **ROL**
  - right - **ROR**
- Rotating instructions with carry
  - left - **RCL**
  - right - **RCR**

For giving a suggestive definition for shifts and rotates let's consider as an initial configuration one byte having the value X = abcdefgh, where a-h are binary digits, h is the least significant bit, bit 0, a is the most significant one, bit 7, and k is the actual value from CF (CF=k). We then have:

```
SHL X,1 ;has the effect X = bcdefgh0 and CF = a
SHR X,1 ;has the effect X = 0abcdefg and CF = h
SAL X,1 ; identically to SHL
SAR X,1 ;has the effect X = aabcdefg and CF = h
ROL X,1 ;has the effect X = bcdefgha and CF = a
ROR X,1 ;has the effect X = habcdefg and CF = h
RCL X,1 ;has the effect X = bcdefghk and CF = a
RCR X,1 ;has the effect X = kabcdefg and CF = h
```

Branching, jumps, loops

jmp operand - unconditional jump

call operand - transfer control to the procedure identified by operand

ret [n] - transfer control to the first instruction after CALL

jmp operand

label / registers / mem. address

```
            mov ax,1
            jmp AdunaDoi
AdunaUnu:   inc  ax
            jmp urmare
AdunaDoi:   add  ax,2
urmare:     .    .    .
```

Conditional jump instructions

cmp d, s - fictious subst d - s

test d, s - d AND s

OF, SF, ZF, AF, PF and CF

OF = 0
CF = 0

SF, ZF, PF - modified

Repetitive instructions

LOOP -> repetitive run of instruction block, as loong as the value of the

CX registes is different from 0

CALL and RET instructions

a CALL instruction may be
- a procedure name
- the name of a registes containing an address
- a memory address

MNEMONICA	SEMNIFICAȚIE (salt dacă..<<relație>>)	Condiția verificată
JB **JNAE** **JC**	este inferior nu este superior sau egal există transport	CF=1
JAE **JNB** **JNC**	este superior sau egal nu este inferior nu există transport	CF=0
JBE **JNA**	este inferior sau egal nu este superior	CF=1 sau ZF=1
JA **JNBE**	este superior nu este inferior sau egal	CF=0 și ZF=0
JE **JZ**	este egal este zero	ZF=1
JNE **JNZ**	nu este egal nu este zero	ZF=0
JL **JNGE**	este mai mic decât nu este mai mare sau egal	SF≠OF
JGE **JNL**	este mai mare sau egal nu este mai mic decât	SF=OF
JLE **JNG**	este mai mic sau egal nu este mai mare decât	ZF=1 sau SF≠OF
JG **JNLE**	este mai mare decât nu este mai mic sau egal	ZF=0 și SF=OF
JP **JPE**	are paritate paritatea este pară	PF=1
JNP **JPO**	nu are paritate paritatea este impară	PF=0
JS	are semn negativ	SF=1
JNS	nu are semn negativ	SF=0
JO	există depășire	OF=1
JNO	nu există depășire	OF=0

Tabelul 4.1. Instrucțiunile de salt condiționat

String contents

```
a6 dd '123', '345','abcd'    ; 3 doublewords are defined, their contents being
                               31 32 33 00|33 34 35 00|61 62 63 64|

a6 dd '1234'  ; 31 32 33 34

a6 dd '12345' ;   31 32 33 34|35 00 00 00|
```

codurile ascii

```
a7 dd '2345'          ;   |32 33 34 35|
a8 dd 12345678h       ;   |78 56 34 12|
```

-> No little endian

=> little endian

Conversions classification

a). Destructive – cbw, cwd, cwde, cdq, movzx, movsx, mov ah,0; mov dx,0; mov edx,0

Non-destructive – Type operators: byte, word, dword, qword

b). Signed - cbw, cwd, cwde, cdq, movsx

Unsigned – movzx, mov ah,0; mov dx,0; mov edx,0, byte, word, dword, qword

c). by enlargement – all the destructive ones ! + word, dword, qword

by narrowing – byte, word, dword

d). implicit vs explicit conversions

e = a+b+c e,b = float , a,c – integer

i=c //only in C NOT in C++ !

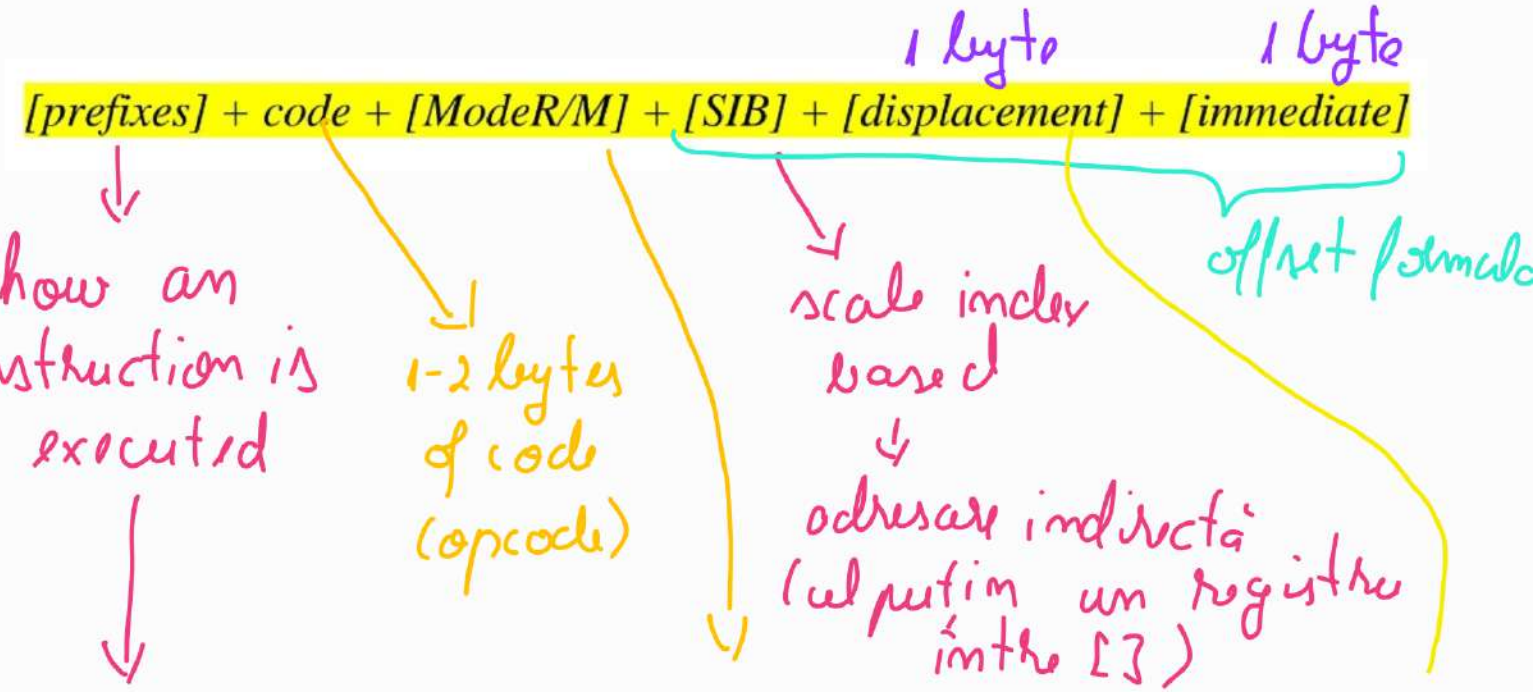
IX Machine instructions representation

- a x86 machine instruction = represents a sequence of 1 to 15 bytes
  = has maximum 2 operands (source / destination operands)

⇓

from these 2 operands, only one may be stored in the RAM memory. The other one must be either one EU registers, either an integer constant

internal format of an instruction



- instruction
- address size
- operand size
- segment override

specify for some instructions the nature and the storage of operands (register or memory)

displacement + immediate = constant

Instruction prefix

- string manipulation instruction prefixes (explicite, provided de programators)

F3h = REP, REPE
F2h = REPNE

where

- REP repeats instruction the number of times specified by *iteration count* ECX.
- REPE and REPNE prefixes allow to terminate loop on the value of **ZF** CPU flag.

- 0xF3 is called REP when used with MOVS/LODS/STOS/INS/OUTS (instructions which don't affect flags)
  0xF3 is called REPE or REPZ when used with CMPS/SCAS
  0xF2 is called REPNE or REPNZ when used with CMPS/SCAS, and is not documented for other instructions.
  Intel's insn reference manual REP entry only documents F3 REP for MOVS, not the F2 prefix.

Related string manipulation instructions are:

- MOVS, move string ; STOS, store string
- SCAS, scan string ; CMPS, compare string, etc.

- segment override prefix => memory access to use specified segment instead of default segment (provided by the programmer)

2Eh = CS	**26h = ES**
36h = SS	**64h = FS**
3Eh = DS	**65h = GS**

Definition

Instruction prefixes are assembly language constructs that appear optionally in the composition of a source line (explicit prefixes) or in the internal format of an instruction (prefixes generated implicitly by the assembler in two cases) and that modify the standard behavior of those instructions (in the case of explicit prefixes) or which signals the processor to change the default representation size of operands and/or addresses, sizes established by assembly directives (BITS 16 su BITS 32).

- main task of an assembler - generating bytes

- at any given moment only one segment of every type may be active
- in 32 bits programming => the segment registers CS, DS, SS, ES contain the values of the selectors of the currently active segments
- CS:EIP -> contain the address of the currently executed instruction
- an assembly language instruction doesn't allow both of its explicit operands to be from the memory

[prefixes] + code + [ModeR/M] + [SIB] + [displacement] + [immediate]

1 2 3 4 5 6

3 - can express register-type operands and/or indirect addressing memory-type operands in which only [base] appears

if [index * scale] also appears => SIB (4) is also needed

3+4 => base + index * scal

5+6 => constant

if "3" tells us that we have a register operand => 4,5,6 are absent

5 -> expresses the direct addressing memory access

6 -> numerical constants

- direct addressing = direct access to the memory operand based on its offset, without needing / specifying any registers in the offset specification formula

CS:EIP – The FAR (complete, full) address of the currently executing instruction

EIP – automatically incremented by the current execution

CS – contains the segment selector of the currently active segment and it can be changed only if the execution will switch to another segment

Mov cs, [var] – forbidden - syntactic ok (illegal instruction in OllyDbg...)
Mov eip, eax - syntax error – symbol 'eip' undefined

Jmp FAR somewhere ; CS and EIP will be both modified !
Jmp start1 ; NEAR jmp – only the offset will be modified, so only EIP !

X Multi-module programming

Volatile vs. non-volatile resources

- **volatile** resources are represented by those registers that **the calling convention is defining them as belonging to the called subroutine**, thus, the caller being responsible **as part of the call code** to save their values (if the called subroutine is using them) and after that, at the end of the call to restore the initial (old) values. So: who is saving the volatile resources ? **The caller** (as part of the call code) . Who is restoring in the end those values ? Also **the caller** but NOT as part of a certain call/entry or exit code. Just restore them after the call in the regular code as a mandatory responsibility.

- **non-volatile** resources are any memory addresses or registers which do not belong explicitly to the called subroutine, but if this one needs to modify those resources, it is necessary that the called subroutine to save them at the entry as part of the entry code and restore them back at exit, as part of the exit code. So: who is saving the non-volatile resources ? **The callee** (apelatul = the **called** subroutine, as part of the entry code) . Who is restoring in the end these values ? Also **the callee** (as part of the exit code).

Call code (THE CALLER):

a). Saving the volatile resources (EAX, ECX, EDX, EFLAGS)
b). Passing parameters
c). Saving the returning address and performing the call

Entry code (THE CALLEE – called subroutine):

a). Building the new stackframe PUSH EBP,
MOV EBP, ESP

b). Allocating space for local variables SUB ESP, nr_bytes
c). Saving non-volatile resources exposed to be modified

Exit code (THE CALLEE):

a). Restoring non-volatile resources
b). Freeing the space allocated for local variables [ADD ESP, nr_bytes_locals] – mentioned here just as a reverse for the above b) from the entry code, but not really necessary because deallocating the stackframe (mov esp, ebp) includes this action anyway from a practically point of view.

c). Deallocating the stackframe MOV ESP, EBP (if we know exactly the size of the stackframe , ADD ESP, sizeof(stackframe) solves similarly...)
and restoring the base of the caller stackframe (old EBP) POP EBP
(a, b c – the reverse of the entry code)

d). Returning from the subroutine (RET) and deallocating passed parameters (if we have a STDCALL function) - (reverse of b + c from the call code)

It is still to be done the reverse of a) from call code. It is the task of the CALLER to do it together with a possible parameters take out from the stack (if it is a CDECL function).