- Machine code = is any low-level programming language, consisting of machine language instructions, which are used to control a computer's central processing unit (CPU). Each instruction causes the CPU to perform a very specific task, such as a load, a store, a jump or an arithmetic logic unit (ALU) operation on one or more units of data in the CPU's registers or memory.

Why do PC's use 0 and 1 (binary)?

Computers speak in binary because of how are they built. A computer is a vast collection of switches (1 - repr. ON (TRUE) phase, 0 - OFF (FALSE phase).

- Because of a physical limitation on how computers work (microtransistors - switches on/off repr. in comp. by binary digits
  ↳ bit
- Base 16 (hexa) = zipped version of base 2
  ↳ it is just an interpretation for better understanding and efficency (computers don't use hexa for numbers

BIT = basic unit for repr. info. inside a computer

- The memory of a computer is ~~organized~~ constructed of bits, but it is organized in bigger elements called bytes.

! Data segment is separate because what is written in high level programming language is sent to a data segment. Data segment is not executable, it is "READ"

Addresses are naming memory locations when creating a variable. An address points to a location where data may be accessed.

BYTE = smallest unit of information that can be accesed and addressed by a microprocessor.

- ALU (Arithmetic and logic unit) → component of CPU
  → it can do arith. and logic operations
  → it can apply the instruction code and activate a circuit and utilizes an already fixed program.

→ The native architecture doesn't work with real numbers, only integers.
→ The task to use real numbers → by mathematical co-processor
→ works with : addition, subtraction, multiplication, division

→ WORK ONLY WITH ADDRESSES, NO VALUES OR COMMANDS

BACKWARDS COMPATIBILITY (based on what was created before)

For not being in danger of mixing everything, any kind of information flows on diff channels, (like highways). An instruction goes on a "command bus".

For being able to acces the operators and operands we need the BUS INTERFACE UNIT (BIU)

BIU
→ responsible for grabbing from memory the operands
→ it needs addresses to work

REGISTERS
→ storage capacities
→ very small in terms of size (8, 16, 32, 64, 128 bits)
→ very fast as the access speed
→ used for temporary store (commands, codes, data, addresses)

Why are bits numbered from right to left?
→ arithmetic operations ($\begin{array}{r} 0011 \\ 1101 \\ \hline 10000 \end{array}$ pune ordine, că în dr→st)

DATA TYPE = meaning of ~~buj~~ trying to interpret smth
→ from a constructie and logical point of view, data type is a couple of structure + associated operations. (specific fa that type)

- interface = variables + functions
- implementation = variables + functions that are private

RAM (Raudom Acces Memory)

→ volatile memory (needs constant power in order to retain data)

→ the acces time to one memory area is THE SAME independently of the location of that memory area (the memory area can be randomly far from the beggining of the memory)

→ you can read & write (ROM - only read) RAM accepts reads & writes in any order (raudom how many r & w and raudom order)

~~There are 8 general registers~~

! x86 microprocessor → EU (Executive Unit)
→ BIU (Bus Interface Unit)

→ EU - run the machine instructions by means of ALU component

→ BIU - prepares the execution of very machine instruction.

- reads an ~~isntr~~ instruction from memory → decodes it → computes the memory address (if any)

- the output configuration is stored in a 15 bytes * BUFFER, from where EU will take it.

* BUFFER = region of memory used to temporarily hold data while is it being moved from one place to another

EU and BIU work in parallel - while EU runs the current instruction, BIU prepares the next one. These 2 actions are synchronized - the one that ends first waits after the other.

EXECUTIVE UNIT (registers, ALU, EFLAGS)

- REGISTERS

  - There are 8 general registers: EAX, EBX, ECX, EDX, ESP, EBP, EDI, ESI (stack: ESP, EBP; strings: EDI, ESI) (E-extended)

  - They are called general registers because the programmer has access to them and he can uses in whatever way he wants. On the other hand every one of them was introduced on 16 bits having in mind a specific role/functionality

- EAX = Accumulator Register
  - most used register in any architecture
  - it is usually used as one of the operands
  - used for multiplication (final result in AX)
  - used for division (dividend usually in (E)AX, quotient in (AX)AL)
- EBX = Base Register
  - a base could be the beggining of an array
- ECX = Counter Register
  - mostly used as numerical upper limit for instructions that need repetitive runs.
  - a so called "iteration variable" (not a variable!)
  - (E)CX = 0 ⇒ finish repetitive instruction
- EDX = Data Register
  - frequently used with EAX, when the result exceeds a dword (EDX:EAX)

All these 4 registers can be accessed and are the LOWER PART. There is no direct way to access the HIGH PART. They can be accessed in halves (through bytes) - access their low/high part

- The use of a stack data structure, rather than a queue is more efficient for the execution of programs because it allows the processor to keep track of function calls and handle recursion.

↗ points to the LAST element put on the stack (element from the top of the s

- ESP - Stack Pointer } registers destinated to work with the
- EBP - Base Pointer } stack

↳ points to the FIRST element put on the stack (points to the stack's basis)

STACK = data structure characterized by a certain discipline over access

- ESI - Source Index
- EDI - Destination Index } usually used for accessing elements from bytes and words strings

AN EXECUTION TAKES PLACE IN THE FOLLOWING ORDER:

→ Usually there exists a main program

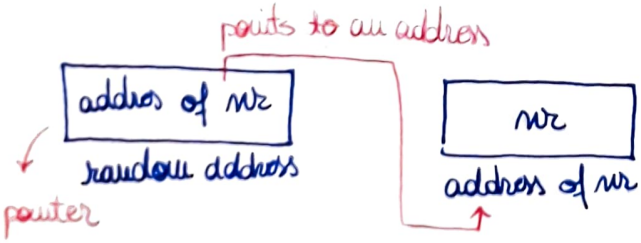
→ When the main program starts, the microprocessor associates ~~with~~ to the the main programming unit the "main program stack frame/activation record" (all of this is done by the compiler)

→ This takes place until a new function appears. The execution ends here, it is all saved and a new stack frame is created. Therefore, the old stack frame creates the new one that is now the main stack frame

→ Everything will be called from the top to the bottom

Ⓐ A pointer variable is a special variable in the sense that it is used to store an address of another variable. To differentiate it from other variables that do not store an address, we use * symbol in declaration

↓ dereferencing operator



ex: *(a+34) ⟶ a[34] (for programmer = array element

↙ pointer arithmetic

for processor = content of a+34 (enough for processor to know if a is a word, dword, etc. the rest is pointer arithmetic to it)

The name of an array in C = its starting location

The name of a variable in assembly is the address of the value.

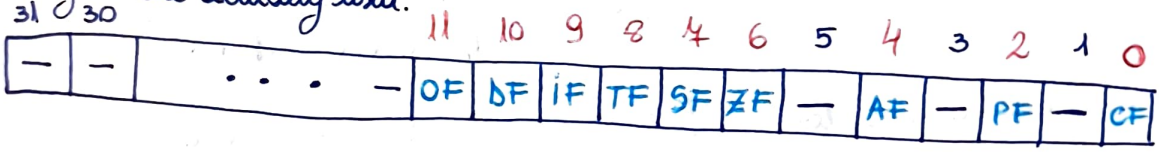
The refference to a certain element is made through syntactics. [] - operator helps (2 sum formula)

• FLAGS

- A FLAG is an indicator represented on 1 bit. prescurtare sintactica

– A configuration of the FLAGS register shows a synthetic overview of the execution of the each instruction. (starea din registrul FLAGS reflecta efectele instructiunilor)

– For x86 the EFLAGS register (the status register) has 32 bits, but only 9 are actually used.



~ EFLAGS REGISTER ~

CF (Carry Flag) = the transport flag. It will be set to 1 if in the LPO there was a transport digit outside the representation domain of the obtained result and set to 0 otherwise.

Ex:

$$\begin{array}{r} {\scriptstyle 1\ \ 1} \\ 1001\ 0011 \\ +\ 0111\ 0011 \\ \hline (1)\ 0000\ 0010 \end{array}$$

CF = 1

the obtained result didn't fit so the extra 1 goes in CF

But why? We are in binary logic. Because everything is binary ⇒ any instruction has max 2 operands. Binary numbers ⇒ only 1 and 0. Since 1 is a bit ⇒ it doesn't need to sacrifice such bigger (byte, word, etc), so for marking that there is an extra 1 ⇒ it will use a flag.

Why no operations with 3 operands? Because the possibilty of having a transport digit becomes 2 (impossible - not binary)

! CF flags the UNSIGNED OVERFLOW

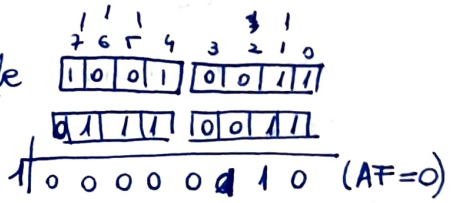
PF (Parity Flag) - indicates if the number of set bits is odd or even in the binary representation of the least significant byte of the result of the LPO (last performed operation).

ex: res. of LPO = 26 (11010b) ⇒ PF = 0 (3 ones)
res of LPO = 10 (1010b) ⇒ PF = 1 (2 ones)

AF (Auxiliary Flag) - shows the transport value from bit 3 to bit 4 of the LPO's result.

→ AF is set to 1, if for example, during an "add" operation there is a carry from the low nibble to the high nibble, or a borrow from the high nibble to the low nibble (subtraction).

→ work like a CF, but reporting to a nibble (half a byte)



ZF (Zero Flag) is set to 1 if the result of the LPO was 0 and set to 0 otherwise.

ZF = 1, if LPO = 0
ZF = 0, if LPO ≠ 0

SF (Sign Flag) is set to 1 if the result of the LPO is strictly negative and set to 0 otherwise.

• What means to represent negative numbers? It is needed to reserve one bit for the sign, so only 7 bits are used for the absolute value.

→ Mathematicians invented a mechanism in order to keep the importance of the sign bit while it is still participating at the construction of the value. (2's complement)

→ SF marks the last bit (most significant bit)

ex: 0010 0110 ⇒ SF = 0

→ ! For any architecture at computer science, ZERO IS A POSITIVE NUMBER

Why? Because when interpreting zero as a signed number, using 2's complement, the MSB is zero (positive) - the SF = 0

TF (Trap Flag) – is a debugging flag. If set to 1 – the machine stops after way execution. If set to 0 – let everything flow.

→ This is why programmers can use debuggers were step by step.

IF (Interrupt Flag) – If set to 1 interrupts are allowed. If set to 0 interrupts will not be handled.

- CRITICAL SECTION = section of code that cannot be interrupted. When IF = 0, the code cannot be interrupted. 1 → opposite. The time between the switch of IF from 0 to 1 is called a time critical section

DF (Direction Flag) – for operating string instructions

– If set to 0 ⇒ string parsing will be performed in ascending ~~way~~ order (start → end)

– If set to 1 ⇒ — 11 — descending order (end → start)

NOT SET BY THE PROCESSOR

OF (Overflow flag) – flags the SIGNED OVERFLOW. If the result of the LPO (considered in SIGNED INTERPRETATION) didn't fit the reserved space, then OF = 1, else OF = 0

→ like CF, but SIGNED INTERPRETATION

→ never able to have an overflow on multiplication (because the result will be 2 * operand_size)

FLAGS CATEGORIES

A) reporting the status of the LPO (having a previous effect):

- (CF), PF, AF, ZF, SF, OF
- ADC (Conditional jumps: JA=JNBE; JG=JNLE; JZ ...) 23

B) flags to be set by the programmer (having a future effect) on the next instructions:

(CF), TF, IF, DF

using the SPECIAL instructions (4):

CLC - effect: CF = 0
STC - effect: CF = 1
CMC - effect: complements the value of CF
→ 3 instr (CF)

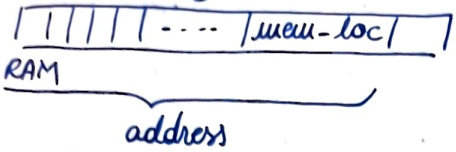
CLD - effect: DF = 0
STD - effect: DF = 1
→ 2 instr (DF)

CLI - effect: IF = 0
STI - effect: IF = 1
→ 2 instr (IF) (only on 16 bits or 32 bits OS restricts the access)

! there are NO instructions to directly access TF, because of the risk of accidentaly setting the value from TF and also because its absolutely special role is to develop debuggers.

Address registers and address computation

Address of a memory location = nr of consecutive bytes from the beggining of the RAM memory ~~and~~ to the beggining of that memory location



! 1 MB = 2^{20} bits ⇒ 20 bits addresses

- An uninterrupted sequence of memory locations, used for similar purposes during a program execution, represents a segment.

→ So a segment represents a logical section of a program's memory, featured by its basic address (beggining), by its limit (size) and by ~~its~~ its type (code, data, stack)

→ Both basic address and segment's size have 32 bits value representations.

When the people wanted to build a computer using intel with 1 MB of RAM the developers had to create a processing mechanism that could compute up with 20 bits addresses. How can you do that on 16 bits? It is needed a logic mechanism to use more than one register. Therefore, you need a 2-step mechanism. They created a memory segment. This segment was initiated for every address and after that, you can specify how far from the beggining of the segment you go. Any beggining of the memory segment must also be a 20 bits address, so they proposed that any segment will start only at a multiple of 16 bytes. The last 4 binary digits are 0, so a 16 bit entity is enough to repr. the most significant digits.

addressing mechanism on 16 bits.

This is only an address specification. Who is counting the final address? It is used a formula - an address computation: only use the first 16 bits from the segment as a number, multiply it by 16 and then add the offset

[segment - code] * 16 + offset

↓

how far from the beggining of the segment you go.

Addressing mechanism on 32 bits

$1\ GB = 2^{30}$

$4\ GB = 2^{32}$ bytes ⇒ need 32 digits in base 2

- If this design could be started then, the 2 step mechanism wouldn't be needed because 32 bits would have been enough. From a logical point of view, a single structure would have been needed.

MEMORY FLAT MODEL : no memory segments with start finish, only one long memory segment

In 8086-processors, the term segment has 2 meanings

1) block of memory of diserete size = physical segment
   - nr of bytes in a physical memory segment:
     - → 64K ⟶ 16 bit processors
     - → 4GB ⟶ 32 bit processors
2) variable-sized block of memory = logical segment
   - occupied by a program's:
     - → code
     - → data

- An offset = address of a location relative to the beggining of the segment
  = the nr of bytes between the beggining of that segment and that particular memory location (valid only if doesn't exceeds the segment's limit which is reffering to)

The flat memory model means that ~~two~~ now there are no longer starting addresses of memory segment. It is a very protective way of memory processing, because it is hidden by the operating system which is hiding all the segments that are possible to be handed to the programmer. So, the OS is managing a SELECTOR TABLE.

Address specification = pair of a segment selector and an offset

Segment selector = numeric value of 16 bits that selects uniquely the accessed segment and his features.

↳ defined and provided by the OS

→ In hexadecimal, an address specification can be written as:

$S_3 S_2 S_1 S_0 : \sigma_7 \sigma_6 \sigma_5 \sigma_4 \sigma_3 \sigma_2 \sigma_1 \sigma_0$

selector (selects DS, CS, SS, ES) — offset (base adr, limit) ← 16 bits ← 16 bits

↳ has these obtained by the processor after the segmentation process

! $\sigma_7 \sigma_6 \sigma_5 \sigma_4 \sigma_3 \sigma_2 \sigma_1 \sigma_0 \leq l_4 l_6 l_5 l_4 l_3 l_2 l_1 l_0$ (condition to give access to a specific location)

The actual segmentation - address computation :

* $a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0 := b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 + \sigma_7 \sigma_6 \sigma_5 \sigma_4 \sigma_3 \sigma_2 \sigma_1$ (≤ limit)

computed address (hexa)

↳ linear address / segmentation address

→ address specification = FAR address

→ - spec. only by offset = NEAR address

Concrete ex. of an address specification : 8 : 1000h

To compute the linear address, the processor will:

① check if the segment with value 8 was defined by the OS and blocks the access if such a segment wasn't defined (memory violation error)

② it extracts the base address (B) and the segment's limit (L), for ex : B = 2000h, L = 4000h

③ it verifies if the offset doesn't exceed the limit : 1000h > 4000h (?). If it does, the access will be blocked

④ it adds up the offset to the base address. This computation = made by ADR component of BIU.

* segmentation → segmented addressing model

(def. the start of the segment and then def. the offset)

- segment start from 0 ($B = 0$)
- maximum possible size ($L = 4GB$)

$\Rightarrow \underbrace{s_3 s_2 s_1 s_0 = \sigma_7 \sigma_6 \sigma_5 \sigma_4 \sigma_3 \sigma_2 \sigma_1 \sigma_0}_{\text{logical address}}$

$\xrightarrow[\text{into}]{\text{turn}} a_7 \dots a_0 := 00000000 + \sigma_7 \dots \sigma_0 \Rightarrow \boxed{a_7 \dots a_0 = \sigma_7 \dots \sigma_8}$

[This is a particular mode of using segmentation (used by modern OS)
↳ FLAT MEMORY MODEL (access only by offset)

The x86 processors also have a memory access control mechanism called paging (independent of address segmentation)

Paging = dividing the virtual memory into pages which are associated (translated) to the physical memory (1 page = 2^{12} bytes = 4096 bytes)

The configuration, control of segmentation and paging are performed by the OS. Only segmentation interferes with addresses, paging = transparent relative to user programs.

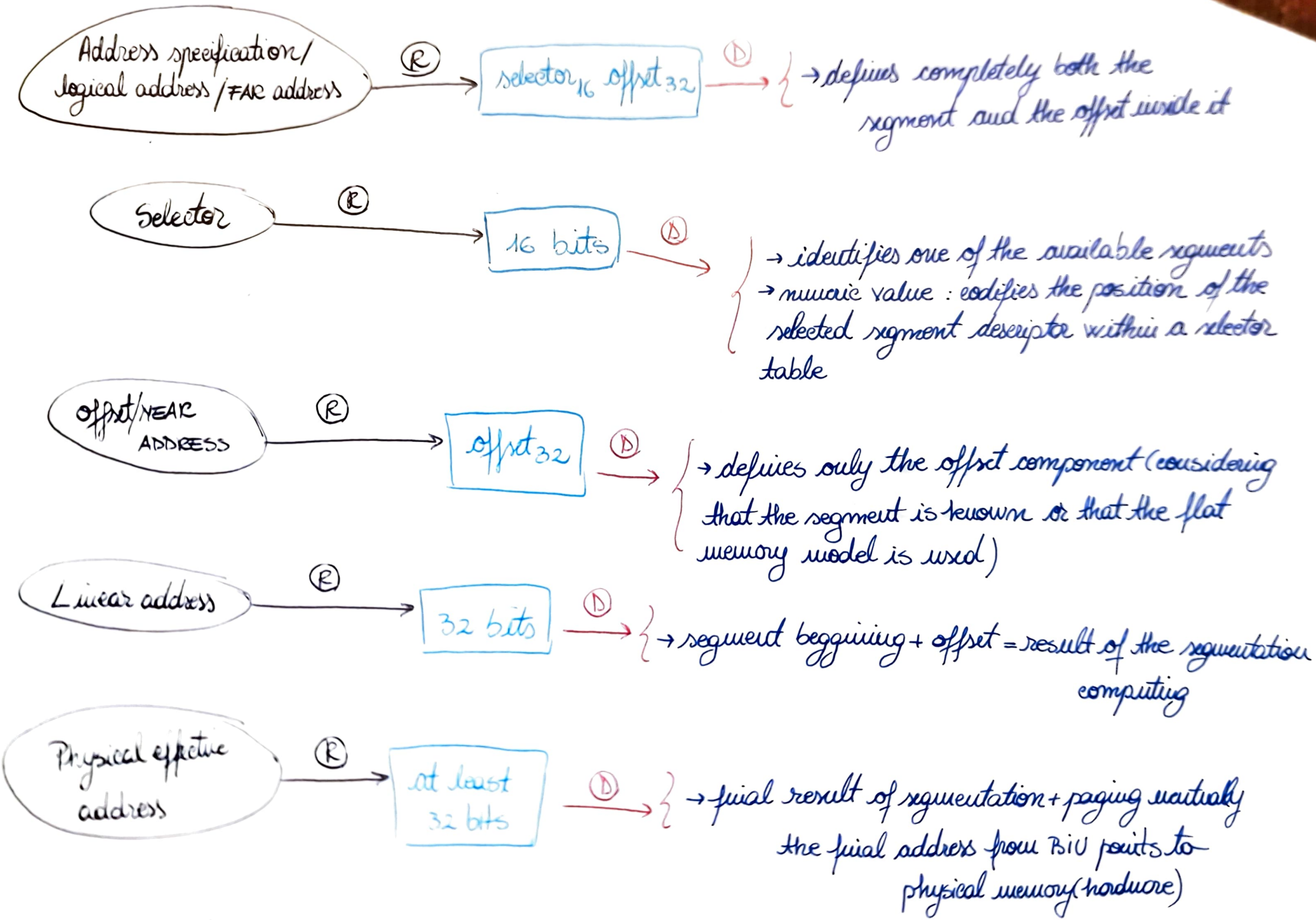
Protected mode on 32 bits — using paging and segmentation

x86 architecture allows 4 types of segments:

- CS (code segment)
- DS (data segment)
- SS (stack segment)
- ES (Extra segment)

→ Every program – composed by one or more segments. At any given moment there is (only) at most one active segment of any type (not synchron). Registers CS, DS, SS, ES, ~~FS, GS~~ from BIU contain the values of the selectors of the active segments correspondingly to every type. So CS, DS, SS, ES determine the starting addresses (unique base) and the dimensions (limit) of these 4 active segments.

→ EIP contains the offset of the current instruction inside the current code segment (like an index through the current segment) - register managed by BIU


Address specification/ logical address / FAR address
R
selector16 offset32
D
→ defines completely both the segment and the offset inside it
Selector
R
16 bits
D
→ identifies one of the available segments
→ numeric value : codifies the position of the selected segment descriptor within a selector table
offset/NEAR ADDRESS
R
offset32
D
→ defines only the offset component (considering that the segment is known or that the flat memory model is used)
Linear address
R
32 bits
D
→ segment beggining + offset = result of the segmentation computing
Physical effective address
R
at least 32 bits
D
→ final result of segmentation + paging unitary the final address from BIU points to physical memory (hardware)

- A x86 machine instruction represents a sequence of 1 to 15 bytes – these values specifying an operation to be run, the operands to which it will be applied and also possible supplimentary modifiers.

A x86 machine instr. has max 2 operands. For most instr. they are called source and destination.

From these 2 operands, only one may be stored in the RAM memory. The other one = EU register / constant.

↳ instruction_name destination, source
general form instruction

The internal format (memory repr.) of an instruction varies between 1-15 bytes and has the following general form:

[prefixes] + code + [#mode R/M] + [SIB] + [displacement] + [immediate]

- PREFIXES → control how an instruction is executed
  - → optional (0 to max 4) (occupy 1 byte each)
  - → ex: - may request repetitive execution of the current instr
    - may block the address BUS during execution to not allow concurrent access to operands and results
- CODE (opcode) → the operation to be run
  - → identified by 1 to 2 bytes
  - → mandatory bytes
  - → ex: add EAX, EBX; "add" → opcode
    - EAX = A, EBX = B
    - "add memory location B to mem. loc. A"
- MODE R/M → register / memory mode
  - → 1 byte
  - → specifies for some instructions the nature and the exact storage of operands (reg/mem)
  - → allows the specification of ~~an offset to be~~ a register / memory location described by an offset

[base] + [index x scale] + [constant]

SIB (displacement + immediate)

scale index base

Offset specification formula

(this formula is a value) not an address

[ ] → dereferencing operator

↳ an indication to the processor that you as a programmer need the value not the address

- Base : EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP
- Index : EAX, EBX, ECX, EDX, EBP, ESI, EDI

⚠ ESP never index ⚠

- Scale : 1 / 2 / 4 / 8

As a consequence of the impossibility of opearing more than one Mode R/M, SIB and displacement fields in one instruction →

the x86 architecture doesn't allow encoding of two memory addresses in the same instruction

- With the immediate value → define an operand as a numeric constant on 1/2/4 bytes

- mov eax, ebx ; EAX =

- mov eax, [ebx]

  ↙

  - [ebx] - will be used as an offset into the memory
    - the value will go into the memory and will transfer the value from there to EAX (.indirection)
    - basically, EBX - pointer value (goes to go into memory ~~location~~ first and take the address from there
    - obeys the offset spec. formula ⇒ correct

mov [v], [ebx] - both memory addressing operands
- ebx will act as a memory address
- invalid combination of opcode and operands

mov eax, [ecx + 2*edx - 4] - NU (?)

mov eax, 23 - YES

mov eax, [23] ; - go into offset 23 and take the value from there
- the instruction will appear disassembled (give the rep in base 2)

mov eax, [ecx+2*esp+4] - syntax error (ESP ≠ index)

mov eax, [2*ebx] = [ebx+ebx] - yes
(base, index)

mov eax, [ebx*3] = [ebx+ebx*2] - yes
(also works for 9 (1+8ebx)

For an instruction there are 3 ways to express a required operand:

- Register mode (the required register operand = reg)

ex: mov eax, 17

mov eax, ebx ; both reg → no formula

- Immediate mode (use directly the operand's value, no adr a reg holding it)

mov eax, 17

- Memory addressing mode (with offset spec. formula)

mov eax, [v]

The offset of a ~~formula~~ variable can always be determined at assembly time. This is why if you are using at least one register between [] it will be called indirect addressing.

If it is used only the name of the variable → from the point of off. sp. f. it will be seen as a constant

Modes to access memory:

→ direct addressing - when only constant is present

→ based addressing - if in the computing one of the base reg. is present

→ scaled - indexed addressing - if in the computing one the index reg is present

→ can be combined

indirect addressing (based and/or indexed) - one for which there is at least one register in []

- jump instructions → relative addressing

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

→ this "distance" = expressed as the nr. of bytes to jump over

Conversions clasification

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

TWO'S COMPLEMENT

Mathematically, the 2's complement representation of a NEGATIVE number is the value $2^n - V$, V - absolute value of the represented value.

ex: 1001 0011 = 93h = 147 ⇒ UNSIGNED(1001 0011) = 147

1001 0011 starts with "1" so in SIGNED INTERPRETATION ⇒ negative number

The value: **-(2's complement of 1001 0011)**

! The value of a binary number starting with 1 is:

-(2's complement of the initial binary configuration)

Variants of computing the 2's complement:

→ VARIANT 1 (classic): subtracting the binary content from 100...0 (where the nr. of zeros are exactly the same ≠ with the nr of binary digits that need to be complemented)

ex:
```
  1 0000 0000
   1001 0011
-------------
  0110 1101   = 6Dh = 96+13 = 109 (⇒ the 2's compl of 147 on 8bits = 109)
```

⇒ the value of 1001 0011 in SIGNED interpretation = -109

⊕ number = 1001 0011

- UNSIGNED(1001 0011) = 144
- SIGNED(1001 0011) = -109

→ VARIANT 2 (faster): reversing the values of the bits from the initial binary configuration, after which it is added 1.

ex: 1001 0011 $\xrightarrow[\text{bits}]{\text{rev}}$ 0110 1100 $\xrightarrow{+1}$ 0110 1101 = 109

⇒ SIGNED(1001 0011) = -109

→ VARIANT 3 (much faster): there are left unchanged the values of the bits, starting from the right, until the first 1 bit inclusive, then all the other bits are inversed.

1001 0011 ⟶ 0110 1101 (109) ⇒ SIGNED(1001 0011) = -109

→ VARIANT 4 (most faster) (ONLY IF INTERESTED IN THE ABSOLUTE VALUE IN BASE 10 OF THE 2'S COMPLEMENT)

Rule: The sum of the absolute values of the 2 complementary values is the cardinal of the set representable on that size.

- 8 bits ⇒ 2^8 values = 256 values ($[0,255]$, $[-128,127]$)
- 16 bits ⇒ 2^{16} values = 65536 values ($[0,65535]$, $[-32768, 32767]$)
- 32 bits ⇒ - - - -

number = 1001 0011 = 93h = 147

2's complement = 2^8 - 147 = 256 - 147 = 109

⇒ SIGNED(1001 0011) = -109

Admissible representation intervals UNSIGNED/SIGNED (the same nr of values, only the interpretation matters)

(2 bits) 1 BYTE ⇒
- $[0, 2^8-1]$ - UNSIGNED
- $[-2^7, 2^7-1]$ - SIGNED

(16 bits) 1 WORD ⇒
- $[0, 2^{16}-1]$ - UNSIGNED
- $[-2^{15}, 2^{15}-1]$ - SIGNED

Admissible representation intervals for n bits

→ $[0, 2^n-1]$ - UNSIGNED

→ $[-2^{n-1}, 2^{n-1}-1]$ - SIGNED

Types of questions to ask starting from a given binary configuration

a) REPRESENTATION = $\overline{0xxx..x}$
UNSIGNED $(\overline{0xx..x}) = +\overline{abc}$
$\Rightarrow$ SIGNED $(\overline{0xx..x}) = \overline{+abc}$
(A number that begins with 0 in base 2 is the same in both interpretations!)

b) REPRESENTATION = $\overline{0xx..x}$
value $(\overline{0xx..x}) = +abc$ (UNSIGNED + SIGNED)
$\Rightarrow -\overline{abc}$ = the 2's complement of the initial binary configuration

ex: 109 = 0110 1101
2's comp (0110 1101) = 1001 0011 $\Rightarrow$ -109 = 1001 0011

A complementary value of an integer that starts with „0" will always start with „1" (exception: 0) and it will fit in the admissible representation interval (both values are the same size: bytes, words, etc — -109 and 109 bytes)

c) REPRESENTATION = $\overline{1xx..x}$
~~(~~ $\overline{1xx..x}) = +abc$ (UNSIGNED)
$\Rightarrow$ SIGNED $(\overline{1xx..x}) = -$(2's complement of the initial binary configuration)

ex: 147 = 1001 0011
2's comp (1001 0011) = 0110 1101 = 109 $\Rightarrow$ SIGNED (1001 0011) = -109

d) REPRESENTATION = $\overline{1xx..x}$
value $(\overline{1xx..x}) = +abc$
$-\overline{abc}$ = also the 2's complement of the initial binary config.

BUT performing this, the result in binary will start with a „0" (and we expect a negative number) $\Rightarrow$ it will provide a natural number in binary from the same interval (or of the same representation size)

$\Rightarrow$ Starting from value $(\overline{1xx..x}) = +abc$ it will not obtain the value $-abc$ ON THE SAME REPRESENTATION SIZE !!!

WHY ISN'T THIS (2's comp) A BIDIRECTIONAL THING?

Looking at the definition, which says: „Mathematically, the 2's complement representation of a NEGATIVE number is the result of $2^n - V$, V - absolute value of that number." So, we can relate to this formula only when our binary representation starts with 1 (in our mind this means that we have a negative number, so we want to find out its value ⇒ we use the -(2's compl. of the binary config). {Logically, this is why "2's complement was invented. Because of computing the negative value of a binary number, without limitating the possible binary digits and decreasing the value („1" means "-" so the rest = value)

Ex: is -147 complement for 109?

109 = 0110 1101 ⇒ 2's comp(0110 1101) = 1001 0011 = 147

But this in both (SIGNED and UNSIGNED) interpretations means a positive number ⇒ there is no point in finding the 2's comp.

Why numbers that start with „0" are always (considered) positive (s.p.v)

UNSIGNED: $[0, 2^n - 1]$

SIGNED: $[-2^{n-1}, 2^{n-1} - 1]$ } ∩ ⇒ $[0, 2^{n-1} - 1]$ (only positives)

⇒ The signed and unsigned interpretations of any binary configuration starting with „1" WILL ALWAYS BE DIFFERENT and they will NEVER be parts of the same admissible interval!

The absolute values of the 2 interpretations represent 2 complementary values

$147 = 1001\ 0011$

$147 \in [0, 255]$ UNSIGNED int

$-147 \notin [-128, 127]$ SIGNED int

$\Rightarrow -147$ NOT representable on 1 byte even if 147 is

CORRECT WAY OF COMPUTING $-\overline{abc}$, where value $(\overline{1xx..x}) = \overline{abc}$:

1) $147 = 1001\ 0011$

$-147 \notin \underbrace{[-128, 127]}_{\text{byte}}$, but $-147 \in \underbrace{[-2^{15}, 2^{15}-1]}_{\text{word}}$

2) representation on a word of 147 : 0000 0000 1001 0011 (1)

(starting with zero $\Rightarrow$ visually look like a positive nr)

-147 = "2's complement of 147"

2's comp of 147 = 1111 1111 0110 1101 (2)

the sum of this abs values is $2^{16} \Rightarrow$ it verifies the condition

The involvement of "2's complement" is UNIDIRECTIONAL !!!

↳ makes sense to use it starting only from the signed interpretation and representation on negative numbers.

OVERFLOW CONCEPT ANALYSIS

At the level of the assembly language an overflow is a situation /condition which expresses the fact that the result of the ALPO didn't fit the reserved space for it OR does not belong to the admissible representation interval for that size OR the operation is a mathematical non sense in that particular interpretation (SIGNED/UNSIGNED)

- An addition is seen like this $\begin{cases} b+b=b \\ w+w=w \end{cases}$

$\begin{cases} CF=0 & \text{the UNSIGNED INTERPRETATION in base 10 of the base 2 result addition is correct} \\ CF=1 & \text{the UNSIGNED INTERPRETATION in base 10 of the base 2 result addition is incorrect} \end{cases}$

$\begin{cases} OF=0 & \text{the SIGNED INTERPRETATION in base 10 of the base 2 result addition is correct} \\ OF=1 & \text{the SIGNED INTERPRETATION in base 10 of the base 2 result addition is incorrect} \end{cases}$

Overflow for addition

When?

1) positive + positive = negative
2) negative + negative = positive

$$\begin{array}{r} 0\ldots+ \\ 0\ldots \\ \hline 1\ldots+ \\ 1\ldots \\ \hline 0\ldots \end{array} \quad \Big\} CF=1$$

$$\begin{array}{r} 0101\ 0011+ \\ 0111\ 0011 \\ \hline 1100\ 0110 \end{array} \qquad \begin{array}{r} 83+ \\ 115 \\ \hline 198 \end{array}$$

CF=0 (no carry)

$$\begin{array}{r} 83+ \\ 115 \\ \hline -56 \end{array}$$

(looking SIGNED) makes no sense $\Rightarrow OF=1$

Overflow for subtraction

1) positive − negative = negative

```
0...  -
1....
-----
1
```

2) negative − positive = positive

```
1...  -
0...
-----
0....
```

borrowed

```
1 0110 0010 -
1100 1000
---------
1001 1010
```

```
  98-
 200
-----
 154
```

CF=1 (need borrow)

(~~incorrect~~)

```
  98-
 -56
-----
-102 (incorrect)
```

OF=1

Overflow for multiplication

The multiplication does not produce overflow (the reserved space is enough for both interpretations)

BUT in case of multiplication : (CF=OF!!!)

→ CF = OF = 0 if b*b = b / w*w = w

→ CF = OF = 1 if b*b = w / w*w = d

Overflow for division

The worst effect in case of overflow is in the case for the division operation : the quotient doesn't fit in the reserved space, then the division overflow will signal a "Run-time error" and the OS will stop the program issuing one of these 3 messages - "Divide overflow"

- "Division by zero"
- "Zero divide"

in the case of a correct division, CF and OF - undefined

Ex overflow

100 + 50 = 150 (OF signed)

100 = 64h = 0110 0100

50 = 32h = 0011 0010

150 = 96h = 1001 0110

$150 \notin [-128, 127]$

but $-106 \in [-128, 127]$ but it is not accepted, because (0...) (0...) 100, 50 ⇒ 100+50 should be positive ⇒ we cannot consider the signed interpretation

⇒ **OF = 1**

CF = 0 (no transport digit outside the repr. space)

```
  1001 0110 +
  1000 0010
-------------
1 0001 1000
```

↳ CF=1 (the result seen is out – it doesn't provide the right one

⇒ CF + result on 8 bits = correct result

Necesitatea transcrierii = depășire !!!

!!! În cazul în care adunăm 2 nr de semne diferite ⇒ never overflow

Dacă adunăm 2 nr de același semn – rezultatul are același semn ⇒ no overflow

necesitatea împrumutului ⇒ CF = 1

(*) $a - b \in [-127, 127] \Rightarrow b - a \in [-127, 127] \Rightarrow$ cmp a,b and cmp b,a (same value for OF)

```
mov al, v
shl al, 1 ; pune în CF = 1
jc negative ; if CF = 1 → jump
```

} verifies if the nr. V is < 0

— Assembly language = programing language in which the basic instructions corresponds with the machine operations and which data structures are the machine primary structures.

↳ symbolic language

Symbols – mnemonics + labels

The basic elements with which an assembler works are:

* labels = user-defined names for pointing to data or memory areas
* instructions = mnemonics which sugest the underlyind action
  ↳ the assembler generates the bytes that codifies the corespouding instruction
* directives
  ↳ indications given to the assembler for correctly generating the corespouding bytes

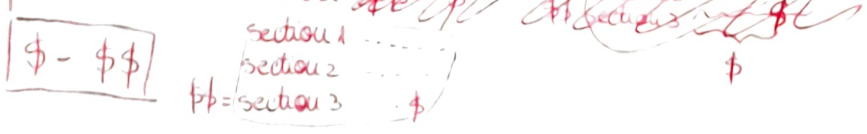
ex: - relationship obj modules
- segment definitions
- conditional assembling
- data definition directives

* location counter = an integer number managed by the assembler for every separate memory segment.
  → at any given moment, the value of the location counter is the number of ~~bytes~~ generated bytes correspondingly with the instructions and the directives already met in that segment (current offset inside that segment)
  → read-only access for the programmer – $ symbol

$ – evaluates to the assembly position at the beggining of the line containing the expression (JMP $ = loop)

$$ – evaluates to the start of the current section → $ $

how far into the section $$ – $



$ - $$ = distance (scalar)

$, $$ = offsets, pointer type, ADDRESS

$ means "address of here"
$$ means "address of the start of the current section"
→ $ - $$ = current size of section

SOURCE LINE FORMAT ← how does a line of code look like in asm

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

here: jmp here ; label + mnemonic + operand + comment
repz cmpsd ; prefix + mnemonic + comment
a dw 1984, 42h ; label + mnemonic + 2 operands + comment
len equ $-a ; label + mnemonic + $-a = operand + comm

2 categories of labels:

→ Code labels = present at the level of instructions sequences for defining the destinations of the control transfer during a program execution
→ can appear in data labels

→ Data labels = provide symbolic identification for some memory locations
↳ from a semantic point of view ≙ variables
→ can appear in code segments.

! The value associated with a label in assembly language is an integer number representing the address of the instruction/directive following that label. ex: JMP ; jump will have the
mov EAX, 2 value = address(mov eax, 2)

EXPRESSION = operands + operators

! Expressions are evaluated at assembly time
(their values = computable at assembly time)

exception : operands = register contents → evaluated at run time !
(offset specification formula)

Operands specification modes

- immediate operands
- direct addressed operands } assembly time
(offset part only)

- memory operands in direct addressing mode (as a complete FAR address
- segment address is determinable here so the whole FAR address is known now) } loading time

↳ address relocation process (adjusting an address by fixing its segment)

- register operands
- indirectly accessed memory op } run time

Immediate operands (computable at assembly time)

→ H/X - hexadecimal
→ D/T - decimal
→ Q/O - octal
→ B/Y - binary

0ABCH - hexa number
ABCH - symbol

ex : B2A : 0xB2A , 0xb2A, - -
↳ hexa

- [straight brackets] ⇒ the variable name denotes the value of the variable

ex: [p] → specifies accessing the value from the address of p (*p)

- any other context ⇒ the variable name = address of the variable [no str. br]

ex: p → address of variable p

Examples

mov EAX, et ; loads into EAX the address (offset) of data/code starting at label et

mov EAX, [et] ; loads into EAX the content from that address (4 bytes)

lea EAX, [v] ; loads into EAX the address (offset) of var v (4 bytes)

(=) mov EAX, v

Using [] → will always indicate accessing an operand from memory

mov EAX, [EBX] → transfer of memory content whose address is given by the value of EBX into EAX

(off spec. ↑ gives an address)

(4 bytes are taken from memory starting at the address specified in EBX as a pointer)

mnemonics → instruction names (guide the processor)
→ directive names (guide the assembler)

operands - parameters which define the values to be processed by the instr/dir

- registers
- constants
- labels
- expressions
- keywords
- other symbols

The offsets of data labels and code labels are values computable at assembly time and they remain [constant] during the whole program's runtime

- A variable once allocated in memory segment will never change its location (its position relative to the start of that segment)

This information ("the position") is determinable at assembly time based on the order in which variables are declared ~~and~~ in the source code, and due to the dimension of representation inferred from the associated type information.

(a db 5
b dw 10, 20 ⇒ mem. seg. : 05 (a) 0A 00 14 00 (b))

Register operands
→ direct using mov eax, ebx
→ indirect usage and addressing mov eax, [ebx] (for pointing memory locations)

- Memory addressing operands — direct
  — indirect

① DIRECT ADDRESSING OPERAND = constant or symbol-rep the address (segment and offset) for an instr/data
→ labels (ex: jmp et)
→ procedures names (ex: call proc1)
→ value of the location counter (ex: b db $-a)

(*) The offset of a direct addressing operand is computed at assembly time. The address of every operand relative to the executable program's structure (establishing the segments to which the computed offsets are relative to) is computed at linking time (obj file, ...) The actual physical address is computed at loading time.

Dacă nu e specificat, e ales implicit pe baza unor reguli
→ The effective address always refers to a segment register.

This register can be EXPLICITLY specified by the PROGRAMMER or otherwise a segment register is IMPLICITLY associated by the ASSEMBLER

The implicit rules for performing this association with an EXPLICIT SPECIFIED OFFSET OPERAND are:

- CS – code labels target of the control transfer instructions (jmp, call, ret, jz, etc)
- SS – in SIB addressing when using EBP or ESP as base (no matter of index of scale) (it ref. to the stack so makes sense to go there)
- DS – for the rest of data accesses

ES – only explicit (ex: ES:[var], ES:[ebx+eax*2-a])
- certain string instr (MOVSB)

(2) INDIRECT ADDRESSING OPERANDS
↳ use registers for pointing to memory addresses
The actual register values → known at runtime ⇒
⇒ indirect addressing – suited for dynamic data operations (move memory)

base_register + index * scale + constant → general form for indirect addressing
↓
comp at as time

When the operand is not specified by the complete formula (some components missing) the assembler will solve the ambiguity by choosing the shortest encoding.

ex: push [eax+ebx] ; may consider eax = base, ebx = index or ↔
pop [ecx] ; restore the top of the stack in the variable which address is given by ecx.
(ecx – base/index)

! All codifications considered by the assembler are equivalent and its final decision doesn't affect the functionality of the resulted code.

ex: lea eax, [eax*2] ; load in eax the val. from eax*2

assembler decide between coding as: base = eax ~~+ index~~ eax
index = eax
scale = 1

or

index = eax
scale = 2

lea eax, [9*eax+12] ⟶ lea eax, [eax + 8*eax+12]

- Operators - perform computations only with constant SCALAR values computable at assembly time

(* ! exception : → adding/subtracting a constant from a pointer
→ offset computation formula (allows "+" operator)

- Expression evaluation is done on 64 bits, the final results being adjusted to the sizeof available in the available usage context of that expression.

→ (+) - performs addition at assembly time
→ (ADD) - performs addition at runtime

Bitwise operators: ~, &, |, ^

~	1's complement
&	AND
|	OR
^	XOR

! - logical negation
! 0 = 1
! (with ≠ 0) = 0

- Segment specification operator (:) performs the FAR address computation of a variable or label relative to a certain segment.

segment : expression

(synthax)

ex: [SS : ebx+4] ; offset relative to SS

[ES : 082h] ; —"— ES

10h : var → offset = value from var label

↓ selector

- Type operators

type expression

synthax

WHAT IT DOES → causes expression to be treated temporarily (limited to that particular instruction) as having "type" sizeof without destructively modifying its initial value (also called "non-destructive temporary conversion operators")

→ BYTE, WORD, DWORD, QWORD, TWORD → (memory stored operands)

1 2 4 8 10 bytes

FAR (6 bytes address) NEAR (4 bytes address) → (code labels)

! IMPORTANT

BYTE/WORD/DWORD/QWORD specifiers have always the task to clarify an ambiguity.

(inclusive when we talk about a memory variable

mov byte [v], 0
mov dword [v], 0

} NASM doesn't associate a data type to a variable (v is not a byte/word/dword v = just a label without an associated data type)

Ex when it needs a data type specifier

- mov [mem], 12
- (i) div [mem]
- (i) mul [mem]
- push [mem]
- pop [mem]
- push 15 - it's ok - generates push dword 15

- IMPLICIT OPERANDS:

mul dword [v]; EDX:EAX = EAX · [v]

div dword [v]; EDX:EAX % v

```
a db 17, -1, 'xyz'
b dw 10001, -128, '13'
```

(SE) - syntax error

```
mov eax, ![a]
mov eax, [!a]
mov eax, !a
mov eax, !(a+7)
```
(SE)

here all of the (SE) are because the ! operator can be applied only to operands computable at assembly time constants/scalars

```
mov eax, !(b-a)   (b-a) = scalar ⇒ OK
mov eax, ![a+4]   (SE)
mov eax, !7 ; eax = 0
mov eax, !0 ; eax = 1
mov eax, ~4 ; eax = 11 ... 110
mov eax, !EBX (SE)
aa equ 2
mov AH, !aa  OK (constant)
mov AH, 14^(~17); AH = FFh
```

v db ...
a dw ...
b dd ....

(ic 00) – invalid combination of opcode and operands

(OSNS) – operation size not specified

push v OK (v - offset of 32 bits)

pop byte [v] (ic00) (should be word/dword)

push [v] (OSNS)

mov eax, [v] OK

push [EAX] (OSNS)

push 15 OK

push [15] (OSNS)

pop [v] (OSNS)

pop v (ic00)

pop [EAX] (OSNS)

pop 15 (ic00)

mov [v], 0 (OSNS)

mov a, b (ic00)

mov [a], b (OSNS)

mov AH, (b) → offset (32 bits)

mov AX, b OK

mov EAX, b OK

mov a, [b] (ic00)

mov [a], [b] (ic00) (no instr with 2 explicit operands)

Location counter and pointer arithmetic

a db 1,2,3,4 ; 01 02 03 04 (offset(a)=0)

lg db $-a ; 04

db a-$; -5 = FB (pt. că lg a fost definit ⇒ $ s-a schimbat)

↓ offset of this = 5

~~c equ A-$~~

lg db $-data → not an offset

→ not det at assembly time ⇒ (SE) $ increments after each is put in memory.

offset(data) = 00401000

lg db data-$ - (SE)

lg2 dw data-a ; NOT (SE)

e equ a-$; -6 ; FA

c equ a-$; 0-6=-6 = FA (there are 6 bytes generated before)

d equ a-$; 0-6 = -6 = FA

e db a-$; 0-6 = -6 = FA

⚠ offsets of variables can be determined at assembly time

⚠ offsets of segments cannot be determined at assembly time.

(known only at loading time)

db lg-a ; 4

db a-lg ; -4

db [$-a] - the content of a memory variable ~~are~~ is not a value accessible at assembly time (SE)

db [lg-a] ; (SE)

equ → saves on constant table

lg1 equ lg1

x dw x ; so it puts the offset ✓ x = offset(x) (09 00 la asamblare)

defined ⇒ has an offset ($)

09 • 10

x dw x – ✓

x db x – (SE) (an offset cannot fit a byte)

lg1 EQU lg1 ; lg1=0

lg1 equ lg1-a ; lg1=0 (because offset(a) = 0)

if offset(a) ≠0 ⇒ NASM BUG

b dd a-start ; (SE) (a-[here(in DS)], start(somewhere else))

dd start-a ; OK!!! WORKS (start-def here, a-such else)

dd start-start1 ; same segment (✓) OK (nr. of bytes between 2 labels)

mov bh, c

Normally, c would be an offset (32 bits), BUT c (EQU...) so it is a constant and it fits !!!

mov ch, lg ; (SE) (lg's offset)

mov ch, lg-a ; ch = 4 (2 addresses subtr.) OK

mov ch, [lg-a] ; mov BYTE PTR DS:[4] (wrong v. error)

mov ch, lg-a ; OK

mov cx, [lg-a] ; mov WORD PTR DS:[4]

mov cx, $-a ($ - CODE SEGMENT's loc. counter
a - such else

WORKS only DESCĂZUT – SCĂZĂTOR

HERE

SMTH ELSE

SMTH ELSE

HERE

Error types in Computer Science

→ Synthax error - diagnosed by assembler/compiler

→ Run-time error (execution error) - program crashes (stops executing)

→ Logical error = program runs until its end or remains blocked in an infinite loop

→ Fatal: linking error (ex: in case of a variable defined multiple times in a multimodule program (1 variable defined only in a single module)

- rcr - bits stored there are rotated number positions to the right (a bit from right → CF → left side)
- sar - shift arithmetic right, the leftmost bits → filled with the bit sign

STRING INSTRUCTIONS

DF = 0/1

I DATA TRANSFER

LODS(B/W/D) : byte from <DS:ESI> loaded in AL ⊕ ESI ± 1
: word from <DS:ESI> loaded in AX ⊕ ESI ± 2
: dword from <DS:ESI> loaded in EAX ⊕ ESI ± 4

STOS(B/W/D): AL-byte is stored in <ES:EDI> ⊕ EDI ± 1
AX-word is stored in <ES:EDI> ⊕ EDI ± 2
EAX-dword is stored in <ES:EDI> ⊕ EDI ± 4

MOVSB = LODSB + STOSB (byte/word/dword from <DS:ESI> to <ES:EDI>)
EDI, ESI ± 1/2/4

<DS ESI> = source string
<ESI:EDI> = destination string

(*) Because of flat memory model OS initializes DS=ES

II DATA COMPARISON

- SCASB (=) { cmp AL, <ES:EDI>
  W
  D { EDI ± 1
- CMPSB (=) ~~cmp(ES:EDI), <DS:~~
  W
  D { cmp <ES EDI>, <DS:ESI>
  { EDI, ESI ± 2

LODS, STOS, MOVS don't change any flags

SCAS, CMPS - change flags (cmp instr)

III PREFIXES FOR REPETITIVE INSTRUCTIONS

rep - prefix string - instruction

REP, REPE } until ECX = 0 OR ZF = 0

REPNE, REPNZ } until ECX = 0 OR ZF = 1

— Data transfer instructions —

MOV dest, sourse (b-b, w-w, d-d)

PUSH s ; ESP = ESP - 4 and transfers <s> in the stack (s-dword)

POP d ; "eliminates" the current element from the top of the stack and trans to d(dword) ESP+=4

XCHG dest, souree ; <dest> ⇔ (souce) - have to be ≠ values

[reg-segment] XLAT ; AL ← <DS: [EBX+AL]> or AL ← <segment: [EBX+AL]>

CMOVcc d, s ; <d> ← <s> if cc = True (conditional mov)

PUSHA/PUSHAD ; pushes in the stack EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI

POPA/POPAD ; pops EDI, ESI, EBP, ESP, EBX, EDX, ECX, EAX from the 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 = True, otherwise <d> ← 0 (byte set on condition code)

(*) PUSH s | POP d

→ Operands s and d must always be doublewords, because the stack is organized on doublewords.

→ The stack grows from big addresses to small addresses, 4 bytes at a time ESP pointing always at the top of the stack

push eax (=) 1) sub esp, 4 ; prepare (allocate) space in order to store the value
2) 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

PUSH/POP ESP

PUSH ESP — the top of the stack = one position higher

POP ESP — the top of the stack moves one position lower

XCHG instruction allows interchanging the contents of 2 operands of the same size (byte, word, dword)
→ at least one operand = register
→ the other one = register / memory address
* because both operands must be L-values

*L-value = any object that has an address

XCHG op1, op2

(good for numerical value → string) (Syntax)

XLAT "translates" the byte from AL to another byte, using for that purpose a user-defined correspondence table called translation table

[reg-segment] XLAT

* translation table = direct address of a string of bytes

The instruction requires the FAR address of the translation table provided:
→ DS:EBX (implicit, if the segment is missing)
→ segment_register:EBX (if the seg-reg specified)

```
mov ebx, Table
mov al, 6
ES xlat   AL ← <ES:[EBX+6]>
```

translates a decimal number between 0-15 into the ASCII CODE of the hexa digit

```
segment data use32
TabHexa db '0123456789ABCDEF'
segment code use32
mov ebx, TabHexa
mov al, numar
xlat ; AL ← <DS:[EBX+AL]>
ES xlat, AL ← <ES:[EBX+AL]>
```

The definition of a variable

→ declaring its attributes (can be done MANY times)

→ allocation (can be done ONCE)

The definition must always be unique (that's why the linkeditor gives an error)

Allocation = means to associate the space necesary for that var.

TIMES = directive that can be also applied for instructions

EQU = directive that allows assigning a numeric value or a string during assembly time to a label without allocating any memory space

```
a1 db 0,1,2,'xyz' ; 00 01 02 78 79 80
a1 db 300,'F'+3 ; warning: byte data exceeds bounds
a2 TIMES 3 db 44h ; 44 44 44
a3 TIMES 11 db 5,1,3 ; 33 bytes : 05 01 03;...
a4 db a2+1 ;
a41 dw a2+1,'be'
a42 dw 1009h
a5 dd a2+1,'bed'
a6 TIMES 4 db '13'
a61 TIMES 4 dw '13'
a7 db a2
a8 dw a2
a9 dd a2
a10 dq a2
a11 db (a2)
a12 dw (a2)
a13 dd dword (a2)
```

Concept of a variable

Variable → we express the fact that the content of that item is modifiable

→ a variable is not a symbol, not just a name

→ a variable = pair of the memory address of some location and the contents

! the symbol = name, but in assembly it denotes the memory address of that location

From a structural point of view, a variable is 4 things that represent its structure

- NAME
- SET OF ATRIBUTES
  - → Type (main attribute, in assembly = size)
    - → defines the domain of values
  - → Visibility domain (scope)
    - → interval of source program (measured in the distance in which that variable is accessible)
  - → Lifetime variable
    - → how much does it live (from where you give it birth to where it dies)
    - → the space attributed for a variable is cleared ⇒ var = dead
    - → a variable is born when it's allocated
    - → measured in seconds
  - → Memory class
    how a variable is allocated
- ADDRESS (address) - optional to formal variables
- VALUE - You have a refference ⇒ you have a value

POINTER ARITHMETIC

Arithmetic operations allowed for pointers (only operations that express as a result a correct location in memory useful as information for the programmer/processor.

* a[7] = *(a+4) = *(4+a) = 4[a]

POINTER ARITHMETIC OPERATIONS

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

contains only (3) operations that are possible:

1. Subtracting 2 addresses (address - address = ok)

   q-p = subtraction of 2 pointers = the number of bytes between these 2 addresses

   (constant) scalar value

2. Adding a numerical constant to a pointer (address + numerical constant)
   * identification of an element by indexing (a[7], q+9)

3. Subtracting a numerical constant from a pointer
   (address - numerical constant) - a[-4], p-4
   * *(a-4) - useful for reffering to array elements

! If you take the last possible memory address - which is repr. on a dword ⇒ go outside a dword ⇒ "memory access violation"

* p+q is supoted in NASM. But p+q doesn't mean addition of 2 pointers (it is an expression with pointers, but not pointer arithmetic)
  → adding 2 addresses makes no sense, because you have no idea what you would point to.

variable var

→ var = address (offset)

→ [var] = its contents (dereferencing operator)

like *(var)

v db 17

add EDX, [EBX + ECX*2 + v - 7] OK

mov EBX, [EBX + ECX*2 - v - 7] (SE) invalid effective address – impossible segment base multiplier

mov [EBX + ECX*2 + a+b - 7], bx (SE) a+b invalid effective address

sub [EBX + ECX*2 + a-b - 7], eax OK a-b correct pointer operation (it depends on a-b-7 → memory access violation)

L-value vs R-value

→ LHS (Left Hand Side of an assignment) → contains the address

→ RHS (Right Hand Side of an assignment) → contains the content

---

a db 17, -2, 0FFh, 'xyz'

b db 232, -112

c db -3, 20h, 7, -2, 432

lga dw $-a OK
lga dw $-$$ OK } usually not the same, but in this case yes

lga equ $-a OK from the point of view of $-a (EQU-directive - so a command addressed to the assembler

mov [lga], ax (SE) no memory area associated with lga

lga dw lga-a } the moment it is write "lga dw" the variable is already defined so it can be used and it is exactly where $ is.

LEA (Load Effective Address) – transfers the offset of the memory operand into the destination register

faster

LEA general_reg, contents of a memory operand

syntax

{general_register ← offset (mem_operand)}

effect

lea eax, [v] (=) mov eax, v

! LEA has the advantage that the source operand may be an addressing expression (unlike mov which allows as a source operand only a variable with direct addressing)

lea eax, [ebx+v-6] ≠ mov eax, ebx+v-6

syntactically incorrect

ebx+v-6 cannot be det at asm time

PUSHF – transfers all the flags on top of the stack (the content of EFLAGS on top of the stack)

POPF – extracts the word from top of the stack and transfer its contents into the EFLAGS register

CLC → CF = 0

CMC → CF = ~CF

STC → CF = 1

CLD → DF = 0

STD → DF = 1

DIRECTIVES

→ direct the way in which code and data are generated during assembling

SEGMENT DIRECTIVE → allows targeting the bytes of code or data emitted by an assembler to a given segment, having a name and some specific characteristics

```
SEGMENT name [type][ALIGN=alignement][combination]
[usage][CLASS=class]
```

→ numeric value assigned to the segment name is the segment address (32 bits) corresponding to the memory's segment's position during runtime

$$ = current segment address (without knowing the current segment name)

* optional (arguments): alignement, combination, usage, class → give info to the link editor and assembler regarding the way in which segments must be loaded and combined in memory

Type - allows selecting the usage mode of the segment:

→ code (text) – segment = code (! the content cannot be written, only executed)

→ data – data segment allows reading & writing, no execution

→ rdata – segment → only read, containing definitions of const data

Alignement → specifies the multiple of bytes number from which that segment may start (only powers of 2)

* alignement = missing ⇒ implicit ALIGN = 1, the segment can start from any address

Combination → controls the way in which similar named segments will be combined with the current segment at linking time

- PUBLIC : tells to link-editor to concatenate this seg with other seg with the same name ⇒ a single segment (length = sum of concat. seg) (implicit)
- COMMON: larger segment ⇒ will include the current one with same name
- PRIVATE : segment cannot be combined with others with the same name
- STACK : segments with same name = concatenated during run-time ⇒ stack segment

Usage - allows choosing another word size than 16 bits (default)

Class - allows choosing the order in which the link-editor puts the segments in memory. all seg with the same class will be placed in a contiguous block of memory.
- no implicit value

segment code use32 class = CODE segment data use32 class = DATA

DATA DEF DIRECTIVES

Data definition = declaration (attrib spec) + allocation (reserving required memory space)
(UNIQUE) (NOT UNIQUE) (UNIQUE)

- data type = size of representation
- Value = address of its first byte
- allocation type = ~~uninit~~ uninitialized data reservation directive (RESB, RESW, RESD, & RESQ)

Times directive allows repeated assembly of an instr/data def

EQU directive - allows assigning a numeric value or a string during assembly time to a label without allocating any memory space or bytes generation

Instruction prefix bytes

x86 instruction can have up to 4 prefixes

each prefix adjusts interpretation of the opcode

String manipulation (instr prefixes - provided explicitly by the programmer)

F3h = REP, REPE

F2h = REPNE

- REP - repeats instruction the nr. of times specified by iteration count ECX

REPE/REPNE allow to terminate loop on the value of ZF CPU flag

0xF3 = REP when ~~using~~ used with MOVS/LODS/STOS/INS/OUTS (instr which do not affect flags

0xF3 = REPE or REPZ when used with CMPS/SCAS

0xF2 = REPNE or REPNZ when used with CMPS/SCAS (not documented for other instructions)

- MOVS move str
- STOS store str
- SCAS scan str
- CMPS compare str

Segment override prefix causes memory access to use specified segment instead of default segment for instruction operand

2Eh = CS

36h = SS

3Eh = DS

26h = ES

64h = FS

65h = GS

- Operand override 66h → changes size of data expected by default mode of the instruction
  (ex: change 16 bit → 32 bit (vice-versa))
- Address override 67h → changes size of address expected by default mode of the instruction
  (ex: 32 bit address → 16 bit address (vice-versa))

- these 2 prefixes appear as a result of some particular ways of using instructions. NOT PROVIDED EXPLICITLY

cbw ; 66:98 → result on AX (16 bits)

cwd ; 66:99 → result on AX:DX (2 reg of 16 bits)

cwde ; 99 → result on EAX (default mode on 32 bits)

push AX ; 66:50 → 16 bits value loaded into the stack (stack org. on 32 bits)

push eax ; 50 - ok - consistent usage with default mode

mov ax, a ; 66:B8 0010 → res on 16 bits

Address size prefix - 0x67

Bits 32

mov eax, [bx] ; 67:8B07 — DS:[BX] - 16 bits addressing

Bits 16

mov bx, [eax] ; 67:8B18 — DS:[EAX] - 32 bits addressing

Bits 16

push dword [ebx] ; 66:67:FF33 (here, def mode = 16 bits)

(66: ebx = 32 bits so switch) (67: address on 32b)

67 : 8B07 mov eax, bx ; offset 16 bits = (BX|BP) + (SI|DI) + (eaux8)

Bits 16

cbw ; 98 ok (reg ou 16 bits)

cwd ; 99 ok (res. of 2 reg ou 16 bits)

~~registres ou 16 bits~~

cwde ; 66 : 98 (16 bits default → res in eax 32 bits)

NEAR/FAR addresses

seg data
off(aici)=0 → aici dd here ; aici := offset (here)

seg code

mov eax, [aici] ; EAX := offset(here)
mov ebx, aici ; EBX := offset (aici) = 0 (computed by the assembler)
jmp [aici] ; jump offset(here) (content of aici) but in the mod 0040 1000h

jmp here → direct access of the target code label

jmp eax ; jump to offset of here (jump here) (content of eax)

jmp [ebx] ; jump to the address stored in the memory location having the address the contents of ebx (indirect register address)

jmp [ebp] ; jmp DWORD SS:[EBP] ↓ don't know exactly what it does, mem_acc_Violation offset (there)

here :

mov eax, 0FFh ← this is because of flat memory model (allows prefixing, but that's it won't use them here)

! It doesn't matter, when you perform a jump, what the segment code in that address specifies. The jump will take only the offset and relate it to the current code segment. !

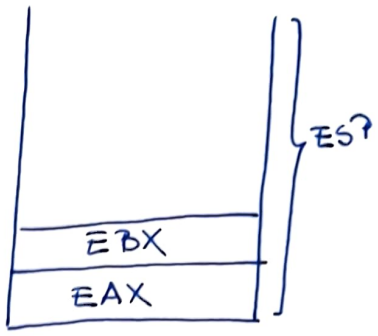
ex: jmp [SS:ebp+12] is the same as jmp [ebp+12] (near jump)
(jumps to CS:EIP, EIP = offset value taken from the stack)
ONLY IF YOU SPECIFY THE KEYWORD "FAR" takes into consideration the code segment from the address
↓ FAR type operator

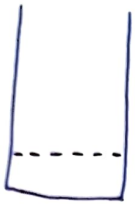
jmp FAR [SS ebx+12] ⇒ CS:EIP ← the far address (48 bits) taken from the stack segment
16-segment + 32-offset

jmp [GS:EBX+ESI*8-1023] (=) " mov EIP, [GS:EBX+ESI*8-1023] "
dword
↳ is a syntax error, but only as explanation

push EAX → ESP decremented by 4
→ [ESP] = EAX

push EBX → ESP decremented by 4
→ [ESP] = EBX





→ stack, when push something onto the stack is like filling the stack.

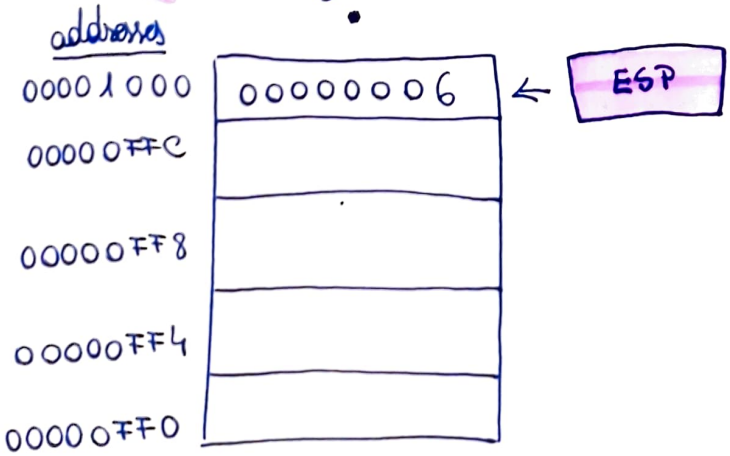
→ so at first the stack points at the bottom

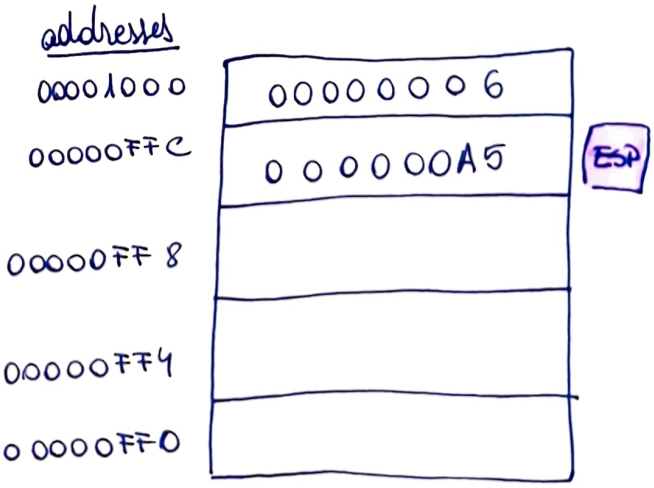
→ when we push EAX, we fill the stack with 4 bytes, so this means, ~~we push~~ in order to place EAX in the stack we have to "make room" (take 4 bytes up) ⇒ decrement ESP by 4, where you can insert now EAX (new start of the adress)

⇒ (THIS is WHY) we decrement

→ the same with increment (when we pop): we free the stack, empty the stack, so it's bigger and you can place/push stuff starting from a lower level (closer to the bottom)

ESP is a register that points to the last adress of the stack.





this stack is ordering opposite to that stack of plates, because the runtime stack grows downward in memory, from highest addresses to lower addresses (BEFORE THE PUSH : ESP = 00001000h
AFTER THE PUSH : ESP = 00000FFCh)

ABOUT FILE OPERATIONS

file descriptor = numerical value that identifies an open file
↳ is for a file
↳ it is used to reference the file when reading from or writing to it

; printf("Number is %d \n", n)

▽ arguments are placed on the stack (in reverse order) from right to left.

push dword [n]
push dword formatp } arguments for the function
call [printf]

→ when 'call' is executed, it pushes the address of the instruction immediately after itself (the return address) (WHY? so when 'ret' is executed to pop that address and the program to know where should be after the call)

call print_hello (stores the location where the program should continue executing once the procedure is finished
print_hello :
; ----- .
ret

→ when is finished, it uses 'ret' instruction to pop the return address off the stack and transfers control back to the instruction after the 'call' instruction

▽ Implicit result returned by the functions are stored in E

The function doesn't empty the stack, it is the responsabi

of the programmer to pop the arguments after the function call

(add ESP, 4*(nr_of_args))

; scanf ("%d", &n) → address of n

arguments from right to left:

push dword n → address of n (so no [ ])

push formats

call [scanf]

add ESP, 4*2

; fopen ("magie.txt", "w")

push dword acces_mode

push dword file_name

call [fopen]

add ESP, 4*2

; if the file was opened successfully, EAX will contain the file descriptor. if an error occurs, fopen will set EAX=0.

cmp EAX, 0

je end

mov [fd], EAX

; fprintf (fd, "Number is %d", n)

push dword [n]

push dword formatp

push dword [fd]

call [fprintf]

add ESP, 4*3

resb
resw
resd
} similar to declaring a variable and specifying its size (C++)

```
; fclose(fd)
  push dword [fd]
  call [fclose]

;
; fread(buffer, 1, len, fd)
buela:
  push dword [fd]
  push dword len
  push dword 1
  push dword buffer
  call [fread]
  add ESP, 4*4
; eax = nr of chars read
  cmp eax, 0
  je cleanup
  mov [nr_of_chars_read], eax
  mov [buffer + EAX], byte 0 ; in case we have less than the nr. of chars declared (256, 100, ...)

;printf("Read from file %d characters: %s", eax, buffer)
  push dword buffer
  push dword eax
  push dword format
  call [printf]
  add ESP, 4*3
  jmp buela
```

Multimodule programming

Modularization = split of subproblems
(for reusability)

Separate compilation = the possibility offered by a programming language to compile separate files at different moments in time and link them together sometimes in the future as the final steps for obtaining the executable program.

global = export
extern = import

→ everything is treated in C

→ restriction → static (static var local to that module)

→ can reuse names as long as they are local

! 2 diff. modules in assembly language may communicate by means of:

1) import/export mechanism global and extern
2) general registers (that are shared between these 2 modules)
3) the stack - also a shared structure between 2 modules

- asm+ high-level programming
  - call phase (call a procedure)
  - Entry phase (
  - Exit phase (restore everything)

→ stages normally implicit on high-level languages, but in asm → hardcode it

CDECL - C convention

→ specific to c prog. language
→ any nr. of params
→ the parameters are passed by pushing them on the stack
→ any type of parameters but extended to a dword
→ the order of passing the parameters: right → left
→ volatile resources: EAX, ECX, EDX, EFlags (can be overwritten/modify)
→ results on EAX/EDX: EAX/ST0
→ cleanup = caller

STD CALL convention

→ specific to windows
→ fixed nr. of params
→ cleanup performed by the callee

Subroutine call

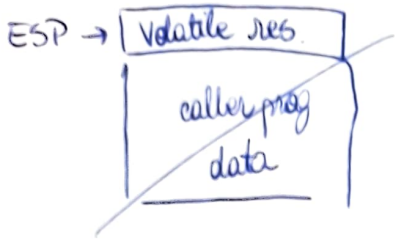
Steps:

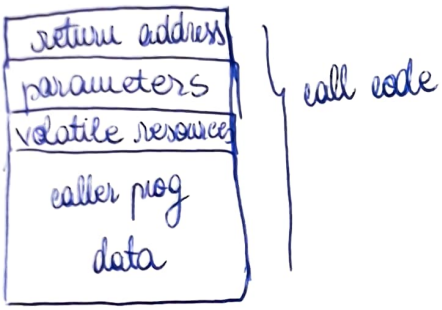
1) Call code: call preparation and execution
2) Entry code: procedure entry and preparation of execution
3) Exit code: returning and freeing up out-of date resources

Call code

Tasks:

1) save the volatile resources in use: push register
2) assure the compliance with constraints (aligned ESP, DF=0)
3) prepare arguments (stack, by convention): push
4) call execution: call
   - call subroutine - sub - statically linked
   - call [subroutine] - sub - dinamically linked (pref)
   - call register or call [variable] (jmp label, reg, var...)





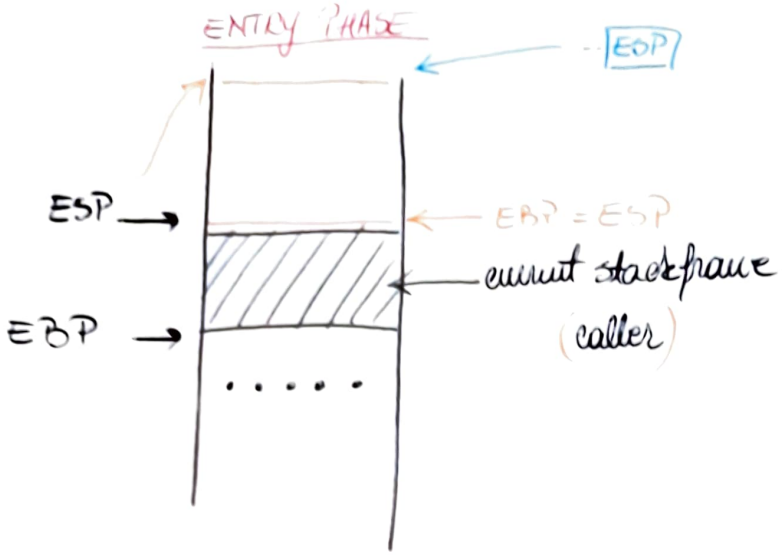
- Entry code

Tasks.

how to create new stackframe → push ebp, mov ebp, esp

1) configuring a stack frame
2) preparing local variables
3) saving a copy of the non-volatile resources that are modified: push register (any reg. except the volatile ones)

- Stack frame = data structure stored in stack of fixed dimension (for a given subroutine) and containing:
  - → the params prepared by the caller
  - → return address (to the instr that follows the call instr)
  - → copies the non-volatile resources used by the subroutine
  - → local variables



→ involves the creation of a NEW STACKFRAME for the called subroutine :

push EBP ; for restoring the base of the CURRENT STACKFRAME when returning

mov EBP, ESP ; This is the birth of the new stackframe (initial sizeof = 0)

ESP will start to grow by currently performed pushes

Exit code

Tasks

1) restore altered non volatile resources
2) release the local variables of the function
3) deallocating the stack frame
4) returning from the function and releasing arguments

→ PRE - call step