-]

iRegisters – storage capacities – very small in terms of sizes (8,16, 32 or 64 bits) but very fast as access speed used for temporary store the operands (data, commands codes, ADDRESSES !!!!!!!!) with which a processor currently.

3654

76532

1234876453

746464646409823409820984208349023840928304828342834 2438

Computer/processor “on N bits”:

a). software perspective – the size of the memory word = the size of the (majority of) the registers (in our case = 32 bits)

b).engineering perspective – the size of the communication buses (channels) - ABUS, CBUS, DBUS

A[i] ; I = INDEX

A[7] = *(A+7)   * = the DEREFERENCING operator in C

A – the name of an array in C is its starting address – it’s a pointer = THE BASE (the starting address)

Byte + byte (ADD) = byte ;

MULTIPLY op1(M positions) * op2 (N positions) → M+N positions

B*B → W(ord); W*W → DW ; DW*DW → QW (EDX:EAX)

(DX:AX)

Data structures – array, list, queue (FIFO), stack (LIFO)

WHY is the stack SO important ?????

RUN-TIME Mechanism of ANY program in Computer Science FOLLOWS ALWAYS THE LIFO ORDER of activating and running the involved programming units (subroutines = functions + procedures).

A user defined type in C is defined by TYPEDEF (which is INCORRECT, because typedef is in fact defining only the structure)

C, Java, VB, Pascal, Fortran – were IMPERATIVE language , because they rely as a central element on the INSTRUCTIONS (commands).

DATA TYPE = structure + associated OPERATIONS !!!!

(essential in this definition is ASSOCIATED – we did not have until OOP AN ENCAPSULATION mechanism)

- You also have in OOP inheritance + polymorphism;

OOP = DATA ORIENTED PROGRAMMING (everything is built having as the central figure the notion of DATA).

From the point of view of the mP – which is its understanding of DATA TYPE notion ?

DATA TYPE for the mP (and in Assembly language) – The size of representation of that element;

On 32 bits these can be – byte, word, dword and qword (these are the assembly language DATA TYPES); You can define variables/operands in the RAM memory by using DATA DEFINITION DIRECTIVES – DB, DW, DD, DQ.

=============

RAM (Random Access Memory) – who is RANDOM ?

- The access time at any given location from the RAM is THE SAME independently of the position (randomely far from the beginning of the memory…)

- In contrast from ROM (read only memories) a RAM supports/allows any number of R/W and in any ORDER (Randomely… reads and writes in a randomely order… The order in which R/W appear is RANDOM…)

==============

101101100101 – <u>representation</u> (in base 2) of what do I need as a base 10 value as a user/human being

So, we need a TRANSFORMATION in base 10 !!!

REPRESENTATION (in base 2) vs. <u>INTERPRETATION</u> (in base 10) !!!!!!!!

UNSIGNED vs SIGNED – how can we REPRESENT them ?

The answer – develop 2's complement representation !!!!!

Contents of a computer memory ---→ correct and consistent INTERPRETATION in base 10 !!!!!!

Base 2 – 1…… → the 2 interpretations (SIGNED and UNSIGNED) will be ALWAYS DIFFERENT !!!!

Base 2 – 0…… → the 2 interpretations (SIGNED and UNSIGNED) will be IDENTICAL !!!!

```
  0110 1100+            6ch + c4h =  130h
  1100 0100
1 0011 0000  =
1      3     0 h        AF=1
```

---

4 bits = 1 semioctet = 1 nibble = 1 hexadecimal digit

CF (*Carry Flag*) is 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. **CF signals overflow for the case of UNSIGNED interpretation.**

1001 0011 +	147+	93h +	- 109 +
0111 0011	115	73h	115
1 0000 0110	262	1 06h	06
CF=1	**(unsigned)**	**(hexa)**	**(signed) OF = 0**

byte + byte → byte (147 + 115 = 6 !!!!! – ADC) It follows that in the above example unsigned addition does not function correctly as an arithmetic operation in assembly language and that is why the processor reacts to this situation by setting CF=1 ! The signed interpretation addition functions correctly and that is why we have OF=0.

In such situations is advisable to take into account the value from CF using further ADC instructions if needed.

word + word → word dword + dword → dword
B+B = B (but what if it doesn't fit ??...) B+B = W

Both CF and OF are reffering to addition and subtraction ONLY.

OF (*Overflow Flag*) flags the **<u>signed overflow</u>**. If the result of the LPO (considered in the signed interpretation) didn't fit the operands reserved space (admissible representation interval), then OF will be set to 1 and will be set to 0 otherwise.

Base 2 – 1…… → the 2 interpretations (SIGNED and UNSIGNED) will be ALWAYS DIFFERENT !!!!

Base 2 – 0…… → the 2 interpretations (SIGNED and UNSIGNED) will be IDENTICAL !!!!

a). Express the number ab in each of the 2, 10 and 16 nummeration bases, in each of the two possible interpretations (signed and unsigned)   1001 0011b, 147, -109, 93h (4 values – NOT 6 !!!!!!)

b). Express the number 73h in each of the 2, 10 and 16 nummeration bases, in each of the two possible interpretations (signed and unsigned)

c). Express the number 129 in each of the 2, 10 and 16 nummeration bases, in each of the two possible interpretations (signed and unsigned)

d). Express the number -37 in each of the 2, 10 and 16 nummeration bases, in each of the two possible interpretations (signed and unsigned)

b,c,d – also 4 values each – NOT 6 !!!!!

Base 2 → Base 10 ← Base 16

(representation) INTERPRETATION (representation)

of base 2 representation

↙ ↘

Unsigned Signed

(absolute value) (positive and/or negative)

CF and OF express overflow situations ONLY for ADDITION and SUBTRACTION !!!!
(The 2 flags OF and CF are set exactly for letting you come up with a correction if you want…)

- If set to 1, both of them are expressing INCORRECT MATHEMATICAL OPERATIONS !!! (That is why in fact they are set to 1 !)

- What about MULTIPLICATION and DIVISION ?

  Op1 – m positions, op2 – n positions , the result og op1 * op2 will be represented on m+n positions. Fortunately, the assembly language IS ALWAYS providing multiplications with enough space for the result.

  Multiplication in assembly language WILL NEVER
  issue a REAL overflow !!!!

  Anyway, CF and OF will have a role in case of multiplication

  B*b = word; b*b=b (2*3=6) → OF=CF=0 (no "overflow")

  B*b = w (255 * 3 =…) → OF=CF=1 ("overflow")

DIVISION ???

- WE DO HAVE overflow in case of division and IT IS THE WORST CASE OF ALL !!!

  w/b = byte ; 1002/3 = 334, but… 334 is a byte ??? No, it isn't and IT IS FATAL !!!
→ RUN TIME ERROR …. Divide overflow, Zero Divide, Division by zero

Number/0 = Zero divide – this is a forbidden operation ! Why this is a forbidden operation ???

Mathematical analysis tells us that:

Number/(epsilon that approaches zero) = a result that aproaches infinit !!! – this is a correct mathematical operations

Number/0 is assembly language will issue a DIVIDE OVERFLOW (ZERO DIVIDE, DIVISION by ZERO) because infinit doesn't fit a byte, a word, a doubleword

If I intend to make 1002/3 =… the assembly language will issue a DIVIDE OVERFLOW because 334 doesn't fit a byte. But from above the processor already considered that 3 messages equivalent… so, it still will issue the same things …. Why ? because from the mP point of view it the same…

From a technical point of view an INT 0 will be issued in ALL situations of this type !!

- **Do I really need 2 flags for overflow ?**
- What means IMUL and IDIV ? …
- Where are IADD and ISUB ?? … They do not exist because

IADD = ADD, ISUB = SUB … because they work exactly the same way in base 2. Addition and subtraction are VALID - UNDER BOTH INTERPRETATIONS SIMULTANEOUSLY !!!!

> When a base 2 addition or subtraction is performed, in fact 2 different simultaneously operations in base 10 are performed !! One for the unsigned interpretation (and that is why we need CF) and the other one for the signed interpretation (and that is why we need OF).
>
> What values will be associated with CF and OF in case of DIVISION ??
> In case of division CF and OF are UNDEFINED !!!

1 addition in base 2 = 2 different SIMULTANEOUS additions in base 10 !!

1 subtraction in base 2 = 2 different SIMULTANEOUS subtractions in base 10 !!

That is why CF and OF are NECESSARY to be TOGETHER present and associated with this operations

- Why DO I NEED IMUL AND IDIV ?

Because in contrast from addition and subtraction (which operate identically in base 2 independently of interpretation – signed or unsigned) multiplication and division function DIFFERENTLY ! Here there is mandatory to specify the interpretation of the operands (signed or unsigned) BEFORE the actual operation is performed… and the microprocessor distinguish that by using DIV vs IDIV and MUL vs. IMUL

In case of addition and subtraction there is no need to make such a distinction before performing it, so the decision regarding a possible interpretation (signed or unsigned) is taken only AFTER the actual operation is performed. That is why we do NOT need IADD or ISUB…

Set CF=1 ; No way !

Mov ax, [v] ; a NASM specific – using ‘[]’ = DEREFERENCING operator = ‘*’ from C

A[7] = *(A+7) ; [] = addition pointer arithmetic operator

Mov eax, [8:1000h];

- Using assembly language directives
  BITS 16
  ………….

  A[7] = *(A+7);   “+ = []”

```
Mov eax, [a1] ;

Mov eax, [8:1000h];


Segment code

    Start:

          Jmp REAL_START

          V db 17

           V1 dw 54321

          REAL_START:   Mov ax, [v]

            Add ebx, eax

            Mul [v1]

            ………
```

Flags instructions do NOT have explicit operands !

In 16 bits programming the address of a memory location is handled explicitly by the programming with total freedom (segment + offset). In 32 bits programming only offsets are allowed to be handled by the programmer.

Limit = size of a segment

At the level of a source code, a programmer can use only ADDRESS SPECIFICATIONS, NEVER final effective addresses ! These may only be handled EXCLUSIVELY by the ADR component from BIU.

In contrast with the segment starting address, on which we don't have any control as programmers in 32 bits programming, offsets must be handled by us !!

For running a program mandatory are the CODE segment and the STACK segment. In writing a source code the only mandatory segment is the CODE segment ! (because stack segment is automatically generated)

The binary form (machine code) of a .exe program does NOT contain ONLY the equivalent of the instructions to be run, but ALSO the WHOLE data segment generated correspondingly by the assembler/compiler !!!

In other words, the DATA segment is also saved together with the code segment on the disk as a part of the **.exe program !!**

November 4th

- main task of an assembler = generating the corresponding bytes

- at any given moment ONLY ONE segment of every type may be ACTIVE

- in 16 bits programming the segment registers CS, DS, SS, ES contained the STARTING ADDRESSES of the currently active segments

- in 32 bits programming the segment registers CS, DS, S S, ES contain the values of the SELECTORS of the currently active segments

- at any given moment during run time the CS:EIP combination of registers expresses /contain the address of the currently executed instruction

- these values are handled exclusively by BIU

- an assembly language instruction doesn't support/allow both of its explicit operands to be from the RAM memory

- that is because BIU may "bring" only one memory operand at a time (for 2 memory operands we would need 2 BIU, 2 segment registers sets etc)

$$\textbf{\textit{offset\_address}} = [\ \textbf{base}] + [\ \textbf{index} \times \textbf{scale}\ ] + [\ \textbf{constant}\ ]$$

(SIB) (displacement + immediate)

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

- the first 2 elements from the offset address computation formula (base and index*scale) are expressed by the SIB byte from the internal format formula

- the third element: the constant, if present, is expressed by the displacement and/or immediate fields

- SIB and displacement participate ONLY to the offset computation of the memory operand, if there is any

- "immediate" field may be also involved in offset computation, but it can also appear INDEPENDENTLY from a memory operand, expressing in such a case an immediate operand (mov eax, 7 ; 7 is "immediate" and no memory operand present in the instruction)

- if Modr/m tells us that we have a register operand the next 3 fields from the internal format formula are absent (because if the operand is a register it can NOT be in the same time also a memory operand or an immediate value)

- if Modr/m tells us that we have a memory operand => SIB byte is mandatory, followed MAYBE by displacement and/or immediate

- the field "immediate" may participate to the offset computation of a memory operand (providing the "constant" field from the offset computation formula) or may appear only by itself expressing the immediate value of an operand (example: mov ebx, 12345678h)

- the displacement field expresses the direct addressing memory access

- immediate field = numerical constants

- in the instructions used in our programs we will use almost exclusively only offsets,these being implicitely prefixed by one of the segment registers CS, DS,SS or ES. (ex. in debugger image - push variabila -> DS:[40100...])

- offset = an adress

- direct addressing means direct access to the memory operand based on its offset, without needing / specifying any register in the offset specification formula (so no base or index !)

- if registers appear in the offset computation formula (base or index) => indirect addressing

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

Mov eip, eax - forbidden

Jmp FAR somewhere ; CS and EIP will be both modified !

Jmp start1 ; NEAR jmp – only the offset will be modified, so EIP !

For an instruction there are 3 ways to express a required operand:
(*operands specification modes*)

- *register mode*, if the required operand is a register; mov ax, bx
- *immediate mode*, when we use directly the operand's value (not its address and neither a register holding it); mov eax,2
- *memory addressing mode*, if the operand is located somewhere in memory. In this case, its offset is computed using the following formula:

$$\textbf{\textit{offset\_address}} = \textbf{[base\_reg] + [index\_reg} \times \textbf{scale] + [constant]}$$
$$\underbrace{}_{\textbf{(SIB)}} \qquad \underbrace{}_{\textbf{(displacement+immediate)}}$$

constant = constant offset (displacement = direct addressing) or/and immediate value

```
mov edx, [var-5]
[EBX+ECX*2 +   v   -7] – ok
     SIB       depl.  const.
```

```
v db 19
add ebx, [EBX+ECX*2 +   v + (-7)] ;  - ok
```

sub ecx, [EBX+ECX*2 - v-7] – syntax error !! invalid effective address – impossible segment base multiplier

mov [EBX+ECX*2 + a+b-7], bx - not allowed ! syntax error ! because of "a+b" invalid effective address – impossible segment base multiplier

```
add [EBX+ECX*2 + a-b-7], ecx – ok !
     SIB         const.
```

So *offset_address* is obtained from the following (maximum) four elements:

- 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 or on a doubleword.

From here results the following modes to address the memory:

- *direct addressing*, when only the *constant* is present;
- *based addressing*, if in the computing one of the base registers is present;
- *scale-indexed addressing*, if in the computing one of the index registers is present.

These three mode of addressing could be combined. For example, it can be present direct based addressing, based addressing and scaled-indexed etc.

A NOT direct addressing mode is named INDIRECT addressing (based and/or indexed). So, an indirect addressing is that for which we have at least one register specified in square brackets ([]).

In the addressing system operations with pointers are performed. Which are the ARITHMETIC operations allowed with pointers in COMPUTER SCIENCE ?...

Answer: Any operation that makes sense... meaning any operation that expresses as a result a correct location in memory useful as an information for the programmer.

- adding a constant value to a pointer a[7] = *(a+7) – useful for going into memory forth and back relative to a starting address
- multiplying 2 pointers ? – No way ... no practical usage !
- subtracting ..... a[-4] , a(-4) ...
- dividing 2 pointers ? - No way ... no practical usage !
- adding/subtracting 2 pointers ?
- ADDING 2 pointers doesn't make sense !! – it is not allowed
- SUBTRACTING 2 pointers !! does makes sense... q-p = nr. Of elements (in C) = nr. Of bytes between these 2 addresses in assembly (this can be very useful for determine the length of a memory area).

Pointer arithmetic...? Contains ONLY 3 operations that are possible:

Adress – adress = ok (q-p = subtraction of 2 pointers = sizeof(array))
Address - offset = address – address

Adress + numerical constant (identification of an element by indexing – a[7]) , q+9
Adress - numerical constant - a[-4] , p-7

- subtraction of 2 pointers = SCALAR VALUE (constant)
- adding a constant to a pointer → a POINTER !!
- subtracting a constant from a pointer → a POINTER !!

POINTER ARITHMETIC OPERATIONS - ***Pointer arithmetic*** represents the set of arithmetic operations allowed to be performed with pointers, this meaning using arithmetic expressions which have addresses as operands.

- subtraction of 2 addresess – ok, is allowed, q-p = the number of bytes between those 2 addresses... !!!!

- adding a CONSTANT (INTEGER) to an address – a[7] = *(a+7)

- subtracting a CONSTANT (INTEGER) from an address – a[-4] = = *(a-4) - useful for reffering array elements

p+q = ???? (allowed in NASM...sometimes...) – but it doesn't mean in the end as we shall see a pointer addition

How do I make the difference between the address of a variable and its contents ?

Var – invoked like that it is an address (offset) ; [var] – is its contents

[] = the dereferencing operator !! (like *p in C)

Assignment: i:=i+1
Dereferencing is usually implicit depending on the context !
BLISS language i←*i+1

LHS – is always an address (left-value)
RHS - is a contents (is part of an expression)

Symbol := expression (usually in 99% of the cases)
Address_expression := value_expression !!! (the most general)

Address of I ← value of I + 1

LHS (Left Hand Side of an assignment = L-value = address) := RHS (Right Hand Side of an assignment = R-Value = CONTENTS !!)

Symbol := expression_value (99% of the cases…)
Address_computation_Expression := expression_value
In C++   f(a, b, 2) = x+y+z

Int& f(i,…) {….return v[i];} – f is a function that will return an L-value !!
f(88,…) = 79;   it means that v[88]=79 !!!

Int& j = i; // j becomes ALIAS for i

(a+2?b:c) = x+y+z ; - correct
(a+2?1:c) = x+y+z; - syntax error !!!   1:=n !!???

Case studies

Mov op_size_dest, op_SAME_size   b,b w,w dw,dw

Mov ax, ebx   - syntax error ! "invalid combination of opcode and operands"

Mov ebx, ch - syntax error ! "invalid combination of opcode and operands"

Mov eax, ebx - eax ← the contents of EBX

Mov eax, [ebx]   = mov eax, [ds:ebx] ; eax = the doubleword value from memory starting at the address DS:EBX

Mov ax, [ebx] = mov ax, [ds:ebx] ; ax = the word value from memory starting at the address DS:EBX

Mov edx, [eax+ebx] – EDX := the doubleword value from memory starting at the address [DS:EAX+EBX]

Mov edx, eax+ebx ; SYNTAX ERROR !! – see the diff. between the OPERATOR + and THE INSTRUCTION ADD !!!

Operators can ***perform computations*** ***only*** with ***constant values*** determinable at assembly time. The ***single exception*** to this rule is ***the offset specification/computation formula***. *(we have there the operator '+' which ha`ndles registers contents !)*

Mov edx, [ebx+eax] – EDX := the doubleword value from memory starting at the address [DS:EAX+EBX]

Mov edx, [esp+ecx] ; EDX := the doubleword value from memory starting at the address [SS:ESP+ECX] - ok ! ESP is always THE BASE !!

Mov edx, [ecx+esp]; - same effect as above ESP – BASE register ; ESP is always THE BASE !! It doesn't matter the order in which you write it !

Mov edx, [esp+2*ecx] ; correct!; ESP – base register; ECX – index ; 2=scale; EDX ← the doubleword taken from memory address given by the [SS:ESP+2*ECX]

Mov edx, [ecx+2*esp] ; syntax error ! ESP can be only a base register

mov dh, [edx + ecx * 4 + 3] ; DH← from memory address DS:edx+ecx*4+3 ONE byte is taken and transferred into DH

mov dx, [edx + ecx * 4 + 3] ; DX← from memory address DS:edx+ecx*4+3 ONE word is taken and transferred into DX
mov eax, [eax*3] = mov eax, [eax+eax*2] – CORRECT !

mov eax, [ebx*9 + 12] = mov eax, [DS:ebx + ebx*8+12]
mov eax, [esp*5] – syntax error ! ESP cannot be an INDEX register !

mov ax, [a] ; constant is the ADDRESS of the variable a, NOT its contents !!!! So, that is why the operand [a] OBBEYS the offset specification formula !!!

Mov reg, [var] – in which of the operands specification modes does it belong ?

Mov eax, [a] - ??? what value has a ??
a = its address ! but when specifying [a] this means THE CONTENTS of a
[] = dereferencing operator in assembly !!

We can access memory values in 2 ways:

- by means of variable names (mov eax, [a])
- or by computing address values applying the offset spec. formula (mov eax, [ebx + 2 * ecx-7] in assembler or var1=*(p-8) in C)

Var d? ….

Mov eax, var   ; EAX ←offset (var) – which is ALWAYS a value on 32 bits !!!
Mov eax, [var]   ; EAX ← 4 bytes from address DS:var – its CONTENTS !!

In TASM and MASM we DO NOT have the dereferencing operator and whenever we define a variable the DATA TYPE inferred by the data definition directive is associated strongly to that variable (db, dw, dd really means associating the corresponding data type with that symbol). This will result in

Mov eax, var ; SYNTAX ERROR in TASM AND MASM !!! if var is NOT a dd

If we need in TASM /MASM to transfer the address of an operand we must use the OFFSET operator:

Mov eax, OFFSET var – transfers the offset of var into EAX in TASM/MASM
Mov eax, var – transfers the CONTENTS of var into EAX in TASM/MASM (no need of the dereferencing operator – dereferencing is implicit in TASM/MASM)

In NASM it is a BIG difference !!!

Mov ax, var   ; IS ALLOWED with a WARNING ! (16 bits reloc of 32 bits value) only 16 bits will be taken (the inferior word from the offset) – it is allowed because of 16 bits addressing mode whih has to still be valid in 32 bits programming also

Mov ax, [var] ; AX ← 2 bytes from address DS:var

Mov ah, var   ; syntax error ! (OBJ file can only handle 16- or 32 bits values) – no offsets on 8 bits are allowed !!!

Mov ah, [var]   ; ok; AH ← 1 byte from address DS:var

Var db 17, 18, 19, 29, 2ah, 0x2a, -3
Mov [var], eax;   the contents of EAX will overwrite the first 4 bytes from var; [] = means the CONTENTS of var ([] = dereferencing operator)

A db 17
B db 19
C db 21
D db 23, 87, 9h

Mov eax, [A]; 4 bytes taken in order (17, 19, 21, 23) and transferring them in EAX

Mov eax, [B-1] = mov eax, [C-2] = mov eax, [D-3]

Mov eax, [D+ebx*2]

Mov ah, ebx ; - syntax error !

Mov ah, [ebx] ; - 1 byte from [DS:EBX] into AH

Mov ax, [ebx]; - 2 bytes from [DS:EBX] into AX

Mov eax, [ebx] ; - 4 bytes from [DS:EBX] into EAX

Offset_spec16 = [BX|BP] + [SI|DI] + [constant] – the offset specification formula on 16 bits !!!

Mov ah, [bx] ;   AH:= 1 byte from DS:[BX]

Mov ax, [bx] ;   AX:= 2 bytes from DS:[BX]

BX is A PART of EBX !!! it means that EBX = 0000000 BX

Mov eax,[bx] ; EAX:= 4 bytes from DS:[BX]

Mov ah, [bh]   ; syntax error !!!! because BH isn't accepted as an indirect register specification (we can use either EBX in spec32 or BX in spec16) !

a db …
b dw…
c dd….

The task of the data definition directives in NASM is NOT to specify an associated data type for the defined variables, but ONLY to generate the corresponding bytes to those memory areas designated by the variables accordingly to the chosen data definition directive and following the little-endian representation order.

So, a is NOT a byte – but only an offset and that is all… a symbol representing the start of a memory area WITHOUT HAVING AN ASSOCIATED DATA TYPE !

So, b is NOT a word – but only an offset and that is all… a symbol representing the start of a memory area WITHOUT HAVING AN ASSOCIATED DATA TYPE !

So, c is NOT a doubleword – but only an offset and that is all… a symbol representing the start of a memory area WITHOUT HAVING AN ASSOCIATED DATA TYPE !

- their task is only to allocate the required space AND to specify the way in which they have to be initialized !!!!

The ***name of a variable*** is ***associated*** in assembly language ***with its offset relative to the segment*** in which its ***definition appears***. The ***offsets of the variables*** defined in a program are ***always constant*** values, determinable at assembly/compiling time.

Assembly language and ***C*** are ***value oriented languages***, meaning that everything is reduced in the end to a numeric value, this is a low level feature.

In a ***high-level programming language***, the ***programmer can access the memory only*** by using ***variable names***, in contrast, in ***assembly language***, the ***memory is/can/must be accessed*** **ONLY** by using the ***offset computation formula***(formula de la doua noaptea) where ***pointer arithmetic*** is also used (pointer arithmetic is also used in C !).

mov ax, [ebx] *– the source operand **doesn't** have an **associated data type** (it represents only a start of a memory area) and because of that, in the case of our MOV instruction the **destination operand** is the one that **decides the data type of the transfer (a word in this case)**, and the transfer will be made accordingly to the little endian representation.*

„„Location Counter

Segment data
a db 17, -2, 0ffh, ‘xyz’,…
db ….
db….
;lga db $-a (mov [lga],….) ok – lga is a variable
;lga EQU $-a (mov [lga],… - illegal !!!) – lga is a CONSTANT
;lga dw $-$$; ok ! – IF a is THE FIRST identifier to be defined in the data segment !!!!
;lga dw lga-a ; !!!!!!!

b EQU 27
c dd 12345678h
lg dw b-a ; NO !!! b is NOT an address !!! syntax error
lg db c-a ; OK !!!
lga dw $-a-4 ; ok !!!
lg dw $-a ; length (a) + 4 !!!

If no SECTION directive is explicitly used, The $$ symbol will evaluate implicitly to the offset of the beginning of the current segment.

“:” are mandatory to be present when DEFINING CODE labels (ex: “start:”) but they must be absent when defining DATA labels (ex: defining variables “a db 17”)

The format of a source line isn’t specific only to the code segment, but is general applicable for ant source line independently of the type of that segment (inclusively a data segment)

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

The offset of any label is a constant value determinable at assembly time. In any programming language the location of an allocated variable (its address) remain fixed; that is why the offsets of variables represents constant values determinable at assembly/compile time.

The SEGMENT address is also fixed but determinable ONLY at loading time.

Any offset used only by itself in a program (without the segment part) will be finally completed BY THE ASSEMBLER to a FAR address by prefixing it with a corresponding segment value. This IMPLICIT value will be always one of the CS, DS or SS segment registers and the rules for these implicit associations are:

- **CS** for 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* or *scale*);
- **DS** for the rest of data accesses;

Examples - implicit rules for prefixing an offset with the corresponding segment register
(material discussed and developed together with you)

Mov eax, [ebx+esp] ; - ESP – base , EBX – index ;…SS
Mov eax, [esp + ebx] ; - ESP – base , EBX – index ;…SS

Mov eax, [ebx+esp*2] ; - syntactic error !!
Mov eax, [ebx+ebp*2] ; - ok ! …DS

Mov eax, [ebx+ebp] ; ok ! …DS
Mov eax, [ebp+ebx] ; ok ! …SS

Mov eax, [ebx*2+ebp] ; ok ! - …SS

Mov eax, [ebx*1+ebp] ; …SS
Mov eax, [ebp*1+ebx] ; …DS

Mov eax, [ebx*1+ebp*1] ; ok !...SS
Mov eax, [ebp*1+ebx*1] ; … DS

Mov eax, [ebx*1+ebp*2] ; ????
Mov eax, [ebp*1+ebx*2] ; ????

Jmp et1 ; …CS:et1…

Jmp eax;

Jmp [DS:et1] ; 4 bytes (short / NEAR jmp) will be taken from [DS:et1] and will be considered as a POINTER to which the jmp will be made IN THE SAME CODE SEGMENT… The jmp will be made to CS:[DS:et1]

JMP FAR [et1]

Jmp 5 ; syntax error because it does not follow the syntax : JMP label/register/memory address

- **CS** for 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* or *scale*);
- **DS** for the rest of data accesses;

Examples - implicit rules for prefixing an offset with the corresponding segment register (material prepared by me in advance) – I left them both here for being parsed and analyzed comparatively if it helps you…

Mov eax, [ebx+esp] ; ESP – base… EBX – index ;EAX ← …SS:…
Mov eax, [esp + ebx] ; ESP – base… EBX – index ;EAX ← …SS:…

Mov eax, [ebx+esp*2] ; syntactic error BECAUSE ESP can be ONLY a base register !
Mov eax, [ebx+ebp*2] ; mov eax, DWORD PTR [DS:EBX+EBP*2]

Mov eax, [ebx+ebp] ;   …DS…
Mov eax, [ebp+ebx] ;   …SS…

Mov eax, [ebx*2+ebp] ; …SS…

Mov eax, [ebx*1+ebp]   ;…SS…
Mov eax, [ebp*1+ebx]   ; …DS…

Mov eax, [ebx*1+ebp*1] ; ;…SS…
Mov eax, [ebp*1+ebx*1] ; …DS…

Jmp et1 ; …CS:et1…

Jmp [et1] ; JMP short [DS:0f6795B4]   - I have to take 4 bytes as the needed correct offset to be referred to the current CS !!! JMP DWORD PTR [DS…] – to be performed at CS:the correct identified offset ; JMP CS:correct_offset (taken relatively to DS) will result usually in "Access violation" run-time error ! (to be checked by you!)

- I go in memory to the address DS:0f6795B4 , because of [] I will take THE CONTENTS from this address (for example 0BA2F5C4) and BECAUSE of JMP this contents will be THE TARGET OFFSET to which I (the processor) will perform this JMP (this offset being relative to the current CS). So, the JMP will be made to the address CS: 0BA2F5C4 !!!!

What you will be as programmers confronted within your checkings will be that DS=ES=SS=GS, a different value for CS and a different value for FS (due to the FLAT MEMORY MODEL).

Jmp 5 ; syntax error BECAUSE it does not obey the JMP syntax , 5 is not a label, nor a register and nor a memory address !!! - Relative call to absolute address not supported by OBJ format

- **CS** for 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* or *scale*);
- **DS** for the rest of data accesses;

[eax+ebx] – indirect addressed operand ; [v] – direct addressed operand (the contents !!!)
V – is determinable at assembly time as an offset !

Bitwise operations and operators

Attention to the difference between operators and instructions !!!

Mov ah, 01110111 << 3 ; AH :=10111000b Vs.

Mov ah, 01110111
Shl ah, 3

& - bitwise AND operator	x AND 0 = 0	; x AND x = x
AND – instruction	x AND 1 = x	; x AND ~x = 0

Operation useful for FORCING THE VALUES OF CERTAIN BITS TO 0 !!!!

| - bitwise OR operator	x OR 0 = x	; x OR x = x
OR – instruction	x OR 1 = 1	; x OR ~x = 1

Operation useful for setting the values of some bits to 1 !!!

^ - bitwise EXCLUSIVE OR operator;	x XOR 0 = x ;	x XOR x = 0
XOR – instruction	x XOR 1 = ~x;	x XOR ~x = 1

Operation useful for COMPLEMENTING the value of some bits !

XOR ax, ax ; AX=0 !!! = 00000000 0000000b

Reported Error types in Computer Science

- **Syntax error – diagnosed by assembler/compiler !**
- **Run-time error (execution error) – program crashes – it stops executing**
- **Logical error = program runs until its end or remains blocked in an infinite loop … if it functions until its end, it functions LOGICALLY WRONG obtaining totally different results/output then the envisioned ones**
- **Fatal: Linking Error !!! (for example in the case of a variable defined multiple times in a multimodule program … if we have 17 modules, a variable must be defined ONLY in a SINGLE module ! If it is defined in 2 or more modules , a "Fatal: Linking Error !!! – Duplicate definition for symbol …." Will be obtained.**

Operators ! and ~ usage (page illustrating the way in which we TOGETHER obtained the results !)

In C - !0 = 1 (0 = false, anything different from 0 = TRUE, but a predefined function will set TRUE =1)

In ASM - !0 = ? It the same mechanism as in C

! Logic Negation: !X = 0 when X ≠ 0, otherwise = 1 (X-bit)
~ 1's Complement: mov al, ~0 => mov AL, 0ffh

a is defined… RESB

Mov eax, ![a] - Expression syntax error ! [a] – is not a SCALAR value…
Mov eax, [!a] - !a is NOT a SCALAR – is a POINTER !!! a is an offset, it is determinable at assembly time, but IT IS NOT A SCALAR !!!!

Mov eax, !a - !a is NOT a SCALAR – syntax error

Mov eax, !(a+7) - !(a+7) is NOT a SCALAR – syntax error

Mov eax, !(b-a) – OK !!! because the difference of 2 pointers IS A SCALAR !!! (usually you will obtain a zer !)

Mov eax, ![a+7] - Expression syntax error !
Mov eax, !7 - EAX = 0
Mov eax, !0 – EAX = 1

Mov eax, ~7 ; 7 = 00 00 00 07h = … 00000111b, so ~7 = 0 ff ff ff f8h

Mov eax, !ebx ; syntax error !

aa equ 2
mov ah, !aa ; AH = 0

Mov AH, 17^(~17) ; AH = 0 ffh = -1
Mov ax, value ^ (~value); eax= 0 ff ffh = -1
Mov eax, value ^ (~value); eax= 0 ff ff ff ffh = -1
(in the general case we can say that we obtain -1)

Operators ! and ~ usage (examples prepared for me in advance – with previously completed answers…)

In C   - !0 = 1 (0 = false, anything different from 0 = TRUE, but a predefined function will set TRUE =1)

In ASM - !0 = same as in C, so
!   Logic Negation: !X = 0 when X ≠ 0, otherwise = 1 (X-bit)

~   1's Complement: mov al, ~0 => mov AL, 0ffh (bitwise operator !)
(because a 0 in asm is a binary ZERO represented on 8, 16, 32 or 64 bits the logical BITWISE negation – 1's complement - will issue a binary 8 of 1's, 16 of 1's, 32 of 1's or 64 of 1's…)

Mov eax, ![a]   - because [a] is not something computable/determinable at assembly time, this instr. will issue a syntax error ! – (expression syntax error)

Mov eax, [!a] - ! can only be applied to SCALAR values !!

Mov eax, !a - ! can only be applied to SCALAR values !!

Mov eax, !(a+7) - ! can only be applied to SCALAR values

Mov eax, !(b-a) – ok !
Mov eax, ![a+7] - expression syntax error
Mov eax, !7 - EAX = 0
Mov eax, !0 – EAX = 1

Mov eax, ~7 ; 7 = 00000111b , so ~7 = 11111000b = 0f8h,
EAX=0 ff ff ff f8h

Mov eax, !ebx ; syntax error !

aa equ 2
mov ah, !aa ; AH=0

Mov AH, 17^(~17) ; AH = 11111111b = 0ffh = -1
Mov ax, value ^ ~value ax=11111111 11111111 = 0ffffh = 1

Operands data type (discussed with you…)

Push v – stack ← offset v (32 bits)

Push [v] - Syntax error ! – Operation size not specified !
Push byte [v] – syntax error !
Push word [v] – ok !
Push dword [v] – ok !
Push qword [v] – syntax error !

Mov eax,[v] - ok ! EAX=dword ptr [v] = mov eax, dword ptr [DS:v]

Push [eax] - Syntax error ! – Operation size not specified !
Push word/dword [eax] ; ok !
…? – is it a correct, valid and accessible address [DS:EAX] ?? Possible run-time error "Memory violation"… but this is something decided at run-time based on the value from EAX…

Push 15 – PUSH DWORD 15 – ok !

Pop [v] - Syntax error ! – Operation size not specified !
Pop word/dword [v] – ok !

Pop v ; v is an address !! BUT… it is a CONSTANT address… You can not change a CONSTANT address !! It would be exactly like attempting to write 2=3 !!!... v is NOT a L-value !!

Pop [eax] ; Syntax error ! – Operation size not specified !
Pop word/dword [eax] ; ok !

Pop 15 - 15 is NOT a L-value !! (15 = 3 !!!)
Pop [15] - Syntax error ! – Operation size not specified !
Pop word/dword [15] = [DS:15] – most probably will issue a run –time error …

Mov [v],0 - Syntax error ! – Operation size not specified !
Mov byte [v], 0 ; OK !!
Mov [v], byte 0 ; OK !!

Div [v] – syntax error
Div byte/word/dword [v] – OK !!!

Imul [v+2] – syntax error
Imul byte/word/dword [v+2] – OK !!!

a dd...
b dw...

Mov a,b – …error

Mov [a], b – syntax error – Op.size not specified !
Mov [a], word b - ok
Mov dword [a], b – ok
Mov byte [a], b – syntax error ! (similar to mov ah, b type of error…)
Mov qword [a], b ; syntax error !

Mov a,[b] – a NOT a L-value !!

Mov [a], [b] – NO asm instruction can have both operands from memory !!
Mov word [a], [b] - NO asm instruction can have both operands from memory !!

Mul v – MUL reg/mem – syntax error because it doesn't follow the syntax of MUL !

Mul [v] – Op.size not specified !
Mul byte/word/dword [v] - ok

Mul eax ; ok !
Mul [eax] ; Op.size not specified !
Mul byte/word/dword [eax] - ok

MUL 15 ; MUL reg/mem – syntax error because it doesn't follow the syntax of MUL !

Operands data type (examples + answers prepared by me in advance…)

Push v – stack←offset v

Push [v] - Syntax error ! – Operation size not specified !! (a PUSH on a 32 bits programming stack accepts both 16 and 32 bits values as stack operands) ;

Push dword [v] - ok
Push word [v] - ok

Mov eax,[v] - ok ; EAX = dword ptr [v], in Olly dbg "mov eax, dword ptr [DS:v]"

Push [eax] - Syntax error ! – Operation size not specified !!
Push word/dword [eax]

Push 15 – PUSH DWORD 15
Pop [v] - Syntax error ! – Operation size not specified !! (a POP from the stack accepts both 16 and 32 bits values as stack operands) ;

Pop word/dword [v];

Pop v ; Invalid combination of opcode and operands , because v is an offset (R-value) and a R-value CANNOT be the destination of an assignment ! (like attempting 2=3)

Pop [eax] – Op size not specified !

Pop 15 - Invalid combination of opcode and operands , because v is an offset (R-value) and a R-value CANNOT be the destination of an assignment ! (like attempting 2=3)

Mov [v],0 - op size not spec.
Mov byte [v],0 ; ok !!!
Mov [v], byte 0 ; ok !!!!

Div [v] – Op. size not spec. – 3 possibilities …
Imul [v+2] - Op. size not spec

a d?...
b d?...

Mov a,b – Invalid combination of opcode and operands , because a is an offset (R-value) and a R-value CANNOT be the destination of an assignment ! (like attempting 2=3)

Mov [a], b – Op. size not spec.
Mov word [a], b or mov [a], word b - the lower word from the offset of b will be transferred into the first 2 bytes starting at offset a !

Mov dword [a], b or… - the offset of b will be transferred into the first 4 bytes starting at offset a !

Mov byte [a], b or…. – SYNTAX ERROR ! because AN OFFSET is EITHER a 16 bits value or a 32 bits value, NEVER an 8 bit value !!!!!
(the same effect as mov ah, v)

Mov a,[b] - Invalid combination of opcode and operands , because a is an offset (R-value) and a R-value CANNOT be the destination of an assignment ! (like attempting 2=3)

Mov [a], [b] - Invalid combination of opcode and operands, BECAUSE asm doesn't allow both explicit operands to be from memory !!!

Mul v – Invalid combination of opcode and operands, BECAUSE syntax is MUL reg/mem

Mul [v] – op size not spec.

Mul eax ; ok !
Mul [eax] ; op size not spec.

MUL 15 ; Invalid combination of opcode and operands, BECAUSE syntax is MUL reg/mem

Pop byte [v] - Invalid combination of opcode and operands
Pop qword [v] – Instruction not supported in 32 bit mode !

Mov eax,0
Idiv eax; run-time error ! Zero divide…

Eroare de asamblare / assembly error = syntax error !

The need for XLAT emerged from situations like this:

How to generate the STRING of digits corresponding to a numeric value ?

fa26h in AX → ‘fa26’
3 + ‘0’ = Ascii code of CHARACTER ‘3’
I+’0’ = ascii code of whatever I is… 0..9…

If the value is between 10..16 → i+’a’-10

Data definition directives (discussed together)

Always your data segment starts at offset   00401000 in OllyDbg

Segment data

a1 db 0,1,2,’xyz’   ;   00 01 02 ‘x’ ‘y’ ‘z’   ; offset(a1- determinat la incarcarea lui OllyDbg)=00401000; offset(a1) determinat la asamblare de catre NASM = 0 !!!
78 79 7A

db 300, “F”+3   ;   2C ascii code for ‘F’ + 3 49
a2 TIMES 3 db 44h ; 44 44 44
a3 TIMES 11 db 5,1,3 ; 05 01 03…11 times (total 33 bytes)

a41 db a2+1 – syntax error !!!
a4 dw a2+1, ‘bc’ ; offset(a2)=00401008h; a2+1=1009h; 09 00?
a44 dw 1009h ; 09 10 (correct, BUT… this particular 10h value it is only finally computable after LOADING !!!)

a5 dd a2+1, ‘bcd’ ; 09 10 40 00| 62 63 64 00
a6 TIMES 4 db ‘13’ ; equiv with a6 TIMES 4 db ‘1’,’3’ ; 31 33 31 33 31 33 31 33
a6bis TIMES 4 dw ‘13’ ; 31 33 31 33 31 33 31 33 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!

d dd [eax] ; syntax error !!
d dd eax ; syntax error !

a7 db a2   ; Syntax error !! Why ? (similar mov ah,a2)

a8 dw a2 ; 08 10
a9 dd a2   ; 08 10 40 00
a10 dq a2 ; 08 10 40 00 00 00 00 00

a11 db [a2] ; syntax error ! A CONTENTS of a memory area or the contents of a registers are NOT values accessible at assembly time ! These are accessible ONLY at run-time !!

a12 dw [a2] ; syntax error !

a13 dd dword [a2] ; syntax error !
a14 dq [a2] ; syntax error !

mov ax, v   ; Warning…

Data definition directives (prepared by me in advance…)

Always your data segment starts at offset   00401000 in OllyDbg

```
Segment data
 a1 db 0,1,2,'xyz'   ; 00 01 02 'x' 'y' 'z' (ascii codes for these chars)
                                            78 79 7A
     db 300, "F"+3   ;   2C 'ascii code for F + 3"; Warning – byte data (300) exceeds bounds !

a2 TIMES 3 db 44h ; 44 44 44
a3 TIMES 11 db 5,1,3 ; 05 01 03… 11 times (33 bytes)

a4 dw a2+1, 'bc'  ;  offset a2 = 00401008    ; 09 10  'b' 'c'

a5 dd a2+1, 'bcd'  ; 09 10 40 00 62 63 64 00
a6 TIMES 4 db '13'  ; 31 33 31 33 31 33 31 33
a6bis TIMES 4 dw '13'  ; 31 33 31 33 31 33 31 33

a7 db a2   ; syntax err. – OBJ format can only handle 16 or 32 bits relocation (equiv. mov ah,a2)

a8 dw a2  ; 08 10
a9 dd a2   ; 08 10 40 00
a10 dq a2  ; 08 10 40 00 00 00 00 00

a11 db [a2]  ; expression syntax error !
a12 dw [a2]  ; expression syntax error !
a13 dd dword [a2] ; expression syntax error !
a14 dq [a2] ; expression syntax error !
mov ax, v   ; Warning – 32 bit offset in 16 bit field !
```

The steps followed by a program from source code to run-time:

- Syntactic checking (done by assembler/compiler/interpreter)
- OBJ files are generated by the assembler/compiler
- Linking phase (performed by a LINKER = a tool provided by the OS, which checks the possible DEPENDENCIES between this OBJ files/modules); The result → .EXE file !!!
- You (the user) are activating your exe file by clicking or enter-ing…
- The LOADER of the OS is looking for the required RAM memory space for your EXE file. When finding it, it loads the EXE file AND performs ADDRESS RELOCATION !!!!
- In the end the loader gives control to the processor by specifying THE PROGRAM's ENTRY POINT (ex: the start label) !!! The run-time phase begins NOW…

Mark Zbirkowski – semnatura EXE = 'MZ'

Segment code (starts always at offset 00402000 – WHO DECIDES THAT ?)

The linker makes such decisions. The base address for loading PEs, at least the default one (set by the Microsoft linkeditor and not only) is 0x400000 for executables (respectively 0x10000000 for libraries). Alink complies with this convention and fills in the ImageBase field of the IMAGE_OPTIONAL_HEADER structure in the P.E. newly built value 0x400000. As each “segment” / section of the program can provide different access rights (the code is executable, we can have read-only segments etc ...), these are planned to start each one at the address of a new memory page (4KiB, so multiple of 0x1000), each memory page can be configured with specific rights by the program loader. In the case of small programs, the implication is that you will get the following map of the program in memory (at run time):

- the program is planned to be loaded into memory at exactly the address 0x400000 (but here will reach the metadata structures of the file, not the code or the data of the program itself)
- the first "segment" will be loaded at 0x401000 (quotation marks because it is not a segment itself but only a logical division of the program, the "segment" of a segment register is not directly associated - for this reason the name is often preferred section instead of segment)
- the second "segment" will be loaded at 0x402000 (the segment that the processor will use for segmentation starts at address 0 and has a limit of 4GiB, regardless of the addresses and section sizes)
- will be prepared "segment" (section) of imports, "segment" of exports and "segment" of stack in the order decided by the likeditor (and of dimensions also provided by him), segments that will be loaded from 0x403000, 0x404000 and so on (increments of 0x1000 as long as they are small enough, otherwise need to be used for increment the smallest multiple of 0x1000 which allows enough space for the contents of the entire segment)

According to the decision logic of the start addresses of the sections, we can conclude that here we have a section (probably data) before the code, containing less than 0x1000 bytes, which is why the code starts immediately after, from 0x402000, the program map being at the end : metadata (headers) from 0x400000, data to 0x401000 and code to 0x402000 (followed of course by other "segments" for stack, imports and, optionally, exports).

Linkeditorul ia deciziile de acest tip. Adresa de baza pentru incarcarea PE-urilor, cel putin cea implicita (setata de catre linkeditorul de la Microsoft si nu numai) este 0x400000 in cazul executabilelor (respectiv 0x10000000 pentru biblioteci). Alink respecta aceasta conventie si completeaza in campul ImageBase al structurii IMAGE_OPTIONAL_HEADER din fisierul P.E. nou construit valoarea 0x400000. Cum fiecare “segment”/sectiune din program poate prevedea drepturi diferite de acces (codul este executabil, putem avea segmente read-only etc...), acestea sunt planificate sa inceapa fiecare la adresa cate unei noi pagini de memorie (4KiB, deci multiplu de 0x1000), fiecare pagina de memorie putand fi configurata cu drepturi specifice de catre incarcatorul de programe. In cazul unor programe de mici dimensiuni, implicatia este ca se va obtine urmatoarea harta a programului in memorie (la executare):

- programul este planificat a fi incarcat in memorie la exact adresa 0x400000 (insa aici vor ajunge structurile de metadate ale fisierului, nu codul sau datele programului in sine)
- primul “segment” va fi incarcat la 0x401000 (pun ghilimele deoarece nu este un segment propriu-zis ci doar o diviziune logica a programului, nu este asociat direct “segmentul” unui registru de segment – din aceasta pricina se prefera de multe ori denumirea de sectiune in loc de cea de segment)
- al doilea “segment” va fi incarcat la 0x402000 (segmentul pe care il va folosi procesorul pentru segmentare incepe la adresa 0 si are limita de 4GiB, indiferent de adresele si dimensiunile sectiunilor)
- va fi pregatit “segment” (sectiune) de importuri, “segment” de exporturi si “segment” de stack in ordinea decisa de catre likeditor (si de dimensiuni prevazute tot de catre acesta), segmente ce vor fi incarcate de la 0x403000, 0x404000 si asa mai departe (incremente de 0x1000 cat timp au dimensiune suficient de mica, in caz contrar fiind nevoie a se folosi pentru increment cel mai mic multiplu de 0x1000 care permite suficient spatiu pentru continutul intregului segment)

Conform logicii de decizie a adreselor de inceput ale sectiunilor, putem concluziona ca aici avem o sectiune (de date probabil) inaintea celei de cod, continand sub 0x1000 octeti, motiv pentru care codul porneste imediat dupa, de la 0x402000, harta programului fiind la final: metadate (antete) de la 0x400000, date la 0x401000 si cod la 0x402000 (urmat bineinteles de alte “segmente” pentru stiva, importuri si, optional, exporturi).

Segment code (starts always at offset 00402000)

```
Start:
        Jmp Real_start    (2 bytes)  - 00402000
        a db 17                      - 00402002
        b dw 1234h                   - 00402003
        c dd  12345678h              - 00402005

Real_start:
         …….
         Mov eax, c ; eax = 402005
         Mov edx, [c]   ; mov edx, DWORD PTR DS:[402005]; in mod normal asta
inseamna ca in EDX will be assigned with the doubleword of offset 00402005 taken
from DS !!!!
         ……
         Mov edx, [CS:c]   ; mov edx, DWORD PTR CS:[402005]
         Mov edx, [DS:c]   ; mov edx, DWORD PTR DS:[402005]
         Mov edx, [SS:c]   ; mov edx, DWORD PTR SS:[402005]
         Mov edx, [ES:c]   ; mov edx, DWORD PTR ES:[402005]
```

The output will be in all of the 5 cases the same EDX:=12345678h   WHY ??

The explanation is directly related to the flat memory model - all segments actually describe the entire memory, from 0 to the end of the first 4GiB of memory. As such, [CS: c] or [DS: c] or [SS: c] or [ES: c] will access the same memory location but with different access rights. Although all selectors indicate identical segments in address and size, they may differ in how other control and access fields of the segment descriptors indicated by them are completed.

The flat model assures us that the segmentation mechanism is transparent to us, we do not notice differences between segments and, as such, we completely get rid of the segmentation worry, but we are interested in the logical division into segments of the program, which is why we use separate sections / "segments". for data code). This is true but only as long as we limit ourselves to CS / DS / ES and SS! The FS and GS selectors point to special segments that do not follow the flat pattern (reserved for the interaction of the program with the S.O.), more precisely, [FS: c] does not indicate the same memory as [CS: c]!

Explicatia este direct legata de modelul de memorie flat – toate segmentele descriu in realitate intreaga memorie, incepand de la 0 si pana la capatul primilor 4GiB ai memoriei. Ca atare, [CS:c] sau [DS:c] sau [SS:c] sau [ES:c] vor accesa aceeasi locatie de memorie insa cu drepturi de acces potential diferite. Desi toti selectorii indica segmente identice ca adresa si dimensiune, acestia pot avea diferente in cum le sunt completate alte campuri de control si de acces ale descriptorilor de segment indicati de catre ei.

Modelul flat ne asigura ca mecanismul de segmentare este transparent pentru noi, noi nu sesizam diferente intre segmente si, ca atare, scapam complet de grija segmentarii (insa ne intereseaza impartirea logica in segmente a programului, motiv pentru care folosim sectiuni/"segmente" separate pentru cod date). Acest lucru este valabil insa doar cat timp ne limitam la CS/DS/ES si SS! Selectorii FS si GS indica inspre segmente speciale care nu respecta modelul flat (rezervate interactiunii programului cu S.O-ul), mai precis, [FS:c] sau [GS:c] NU indica aceeasi memorie ca si [CS:c]!