# Type operators and operands data types

***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 handles registers contents !)*

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

v d? ….
a d?...
b d?...

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 ; The syntax is POP destination, where destination is a L-value !!
"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 dword b ; syntax error !

Pop [eax] – Op size not specified !
Pop (d)word [eax] ; ok!

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); 15 is NOT a L-value !

Pop [15] ; syntax error - Op size not specified
Pop dword [15] ; syntactic ok , cel mai probabil run-time error deoarece probabil [DS:15] va provoca Access violation !!

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 …
Div word [v]; ok!

Imul [v+2] - Op. size not spec
Imul word [v+2]; DX:AX = AX*word de la adresa v+2

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

Mov a,b – Invalid combination of opcode and operands , because a is NOT a L-value, but 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 mov [a], dword b - the offset of b will be transferred into the first 4 bytes starting at offset a !

Mov byte [a], b or mov [a], byte b – 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 word v - syntax error – MUL reg/mem

Mul [v] – op size not spec.

Mul dword [v]; ok !

Mul eax ; ok !
Mul [eax] ; op. size not specified
Mul byte [eax] ; ok !!!

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 !

## 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.**

**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'

## Operands data types

Data definition directives in NASM are NOT data types definition mechanisms !!

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 TO THE ASSEMBLER the way in which they have to be initialized !!!!

**So… IN NASM !... a DATA DEFINITION DIRECTIVE is NOT a mechanism for associating DATA TYPES to variables, but ONLY an INITIALIZATION method for the named memory areas !!!**

**Known things, but good to be remembered:**

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.*