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:

***offset\_address* = [ base\_reg] + [ index\_reg × scale ] +**

**[constant]**

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 mosde 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.

Pointer arithmetic...?

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

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
* adding a constant to a pointer 🡪 a POINTER !!
* subtracting a constant from a pointer 🡪 a POINTER !!

POINTER ARITHMETIC OPERATIONS

* 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

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 !!!

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]

Mov edx, [ecx+esp]; - same effect as above ESP – BASE register

Assignment: i:=i+1

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(…) {….return v[i];}

F(a,7) = 79; inseamna ca v[i]=79 !!!

Int& j = i; // j devine ALIAS pt i

(a+2?b:c) = x+y+z ; - correct

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

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

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

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)

Mov ah, [var] ; 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

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] = move ax, [D-3]

Mov op\_size\_dest, op\_SAME\_size b,b w,w dw,dw

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]

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[7] = \*(a+7) = \*(7+a) = 7[a] !!!!!!

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 !

Instructions – addressed to the PROCESSOR

Directives – addressed to the ASSEMBLER

**Location Counter**

**Segment data**

**……**

**a db 17, -2, 0ffh, ‘xyz’,…**

**dw ….**

**dd….**

**lg dw $-a //pointer arithmetic – subtracting 2 pointers = scalar (numeric constant) - lg=memory variable (mov [lg], bx)**

**lg db $-a**

**lg EQU $-a //lg = constanta !!! (mov [lg],… - illegal !)**

**lg dw $-$$ ; OK !!!**

**lg dw $-data ; syntax error – “Expression is not simple or relocatable”**

**lg dw lg-a ; !!!!!! OK !!!!!!!**

Segment data

A db 17,-2

B dw -20345, “x”

Start2:

C dd 12345678h

Segment code

……

Jmp start2

……