C H A P T E R 3

ASSEMBLY LANGUAGE BASICS

Machine Language of a Computing System (CS) – the set of the machine instructions to which the processor directly reacts. These are represented as bit strings with predefined semantics.

Assembly Language – a programming language in which the basic instructions set corresponds with the machine operations and which data structures are the machine primary structures. This is a **symbolic language**. **Symbols - Mnemonics + labels.**

The basic elements with which an assembler works with are:

* **labels** – user-defined names for pointing to data or memory areas.

* **instructions** - mnemonics which suggests the underlying action. The assembler generates the bytes that codifies the corresponding instruction.

* **directives** - indications given to the assembler for correctly generating the corresponding bytes. Ex: relationships between the object 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 the generated bytes correspondingly with the instructions and the directives already met in that segment (the current offset inside that segment). The programmer can use this value (read-only access!) by specifying in the source code the '$' symbol.

NASM supports two special tokens in expressions, allowing calculations to involve the current assembly position: the $ and $$ tokens. $ evaluates to the assembly position at the beginning of the line containing the expression; so you can code an infinite loop using JMP $.
$$ evaluates to the start of the current section; so you can tell how far into the section are by using ($-$$).

Directive SECTION

```
section .data
   db 'hello'
   db  'h', 'e', 'l','l','o'
   data_segment_size  equ $-$$
```

$-$$ = the distance from the beginning of the segment AS A SCALAR (constant numerical value)!!!!!!!!!!

$ - is an offset = POINTER TYPE !!!! It is an address !!!!
$$ - is an offset =POINTER TYPE !!!! It is an address !!!!

$ means "address of here".
$$ means "address of start of current section".
So $-$$ means "current size of section".

For the example above, this will be 10, as there are 10 bytes of data given.

3.1. SOURCE LINE FORMAT

In the x86 assembly language the source line format is:

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

We illustrate the concept through some examples:

```
here:    jmp here          ; label + mnemonic + operand + comment
repz cmpsd                 ; prefix + mnemonic + comment
start:                     ; label + comment
                           ; just a comment (which could be missed)
a   dw  19872, 42h         ; label + mnemonic + 2 operands + comment

 len equ $-a  ; label + mnemonic + $-a (operand) + comment
```

The allowed characters for a *label* are:

- Letters: A-Z, a-z;
- Digits: 0-9;
- Characters _, $, $$, #, @, ~, . and ?

A valid variable name starts with a letter, _ or ?.
These rules are valid for all valid *identifiers* (symbolic names, such as variable names, label names, macros, etc).

All identifiers are *case sensitive*, the language making the distinction between upper and lower case letters while analyzing user defined identifiers. This means that the Abc identifier is different from the abc identifier. For implicit names which are part of the language (such as keywords, mnemonics, registers) there are no differences between upper and lower case letters (they are *case insensitive*).

The assembly language offers two categories of labels:

1). ***Code labels***, present at the level of instructions sequences for defining the destinations of the control transfer during a program execution. **They can appear also in data segments!**

2). ***Data labels***, which provide symbolic identification for some memory locations, from a semantic point of view being similar with the *variable* concept from the other programming languages. **They can appear also in code segments!**

***The value* associated with a label in assembly language is an integer number representing the address of the instruction or directive following that label.**

The distinction between accessing a variable's address or its associated content is made as follows:

- When specified in ***straight brackets, the variable name denotes the value of the variable***; for example, [p] specifies accessing the value of the variable, in the same way in which *p represents dereferencing a pointer (accessing the content indicated by the pointer) in C;
- In any other context, ***the name of the variable represents the address of the variable***; for example, p is always the address of the variable p;

Examples:

```
mov EAX, et      ; loads into EAX register the address (offset) of data or code starting at label et
mov EAX, [et]    ; loads into EAX register the content from address et (4 bytes)
lea eax, [v]     ; loads into EAX register the address (offset) of variable v (4 bytes)
```

(similar as effect with MOV eax, v)

As a generalization, ***using straight brackets always indicates accessing an operand from memory***. For example, mov EAX, [EBX] means the transfer of the memory content whose address is given by the value of EBX into EAX (4 bytes are taken from memory starting at the address specified in EBX as a pointer).

There are 2 types of *mnemonics*: *instructions names* and *directives names*. *Directives* guide the assembler. They specify the particular way in which the assembler will generate the object code. *Instructions* are actions that guide the processor.

Operands are parameters which define the values to be processed by the instructions or directives. They can be **registers, constants, labels, expressions, keywords** or **other symbols**. Their semantics depends on the mnemonic of the associated instruction or directive.

3.2. EXPRESSIONS

expression - operands + operators. *Operators* indicate how to combine the operands for building an expression. **Expressions are evaluated at assembly time** (their values are computable at assembly time, except for the operands representing registers contents, that can be evaluated only at run time – the offset specification formula).

3.2.1. Operands specification modes

Instructions operands may be specified in 3 different ways, called ***specification modes.***

The 3 operand types are: ***immediate operands, register operands*** and ***memory operands***. Their values are computed at assembly time for the immediate operands and for the direct addressed operands (the offset part only!), at loading time for memory operands in direct addressing mode (as a complete FAR address – segment address is determinable here so the whole FAR address is known now !) – this step involves a so called ADDRESS RELOCATION PROCESS (adjusting an address by fixing its segment part), and at run time for the registers operands and for indirectly accessed memory operands.

??:offset (assembly time) 0708:offset (loading time)

3.2.1.1. Immediate operands

Immediate operands are constant numeric data computable at assembly time.

Integer constants are specified through binary, octal, decimal or hexadecimal values. Additionally, the use of the _ (underscore) character allows the separation of groups of digits. The numeration base may be specified in multiple ways:

- Using the H or X suffixes for hexadecimal, D or T for decimal, Q or O for octal and B or Y for binary; in these cases the number must start with a digit between 0 and 9, to eliminate confusions between constants and symbols, for example 0ABCH is interpreted as a hexadecimal number, but ABCH is interpreted as a symbol.
- Using the C language convention, by adding the 0x or 0h prefixes for hexadecimal, 0d or 0t for decimal, 0o or 0q for octal, and 0b or 0y for binary.

Examples:

- the hexadecimal constant B2A may be expressed as: 0xb2a, 0xb2A, 0hb2a, 0b12Ah, 0B12AH

- the decimal value 123 may be specified as: 123, 0d123, 0d0123, 123d, 123D, ...
- 11001000b, 0b11001000, 0y1100_1000, 001100_1000Y represent various ways of expressing the binary number 11001000

The offsets of data labels and code labels are values computable at assembly time and they remain constant during the whole program's run-time.

```
mov eax, et     ; transfer into the EAX register the offset associated to the et label
```

will be evaluated at assembly time as (for example):

```
mov eax, 8          ; 8 bytes „distance" relative to the beginning of the data segment
mov eax, [var] – in OllyDBG you will find   mov eax, DWORD PTR [DS:004027AB]
```

These values are constant because of the allocation rules in programming languages in general. These rules state that the memory allocation order of declared variables (more precisely the distance relative to the start of the data segment in which a variable is allocated) as well as the distances of destination jumps in the case of **goto** - style instructions are constant values during the execution of a program.

In other words, a variable once allocated in a memory segment will never change its location (i.e. its position relative to the start of that segment). This information is determinable at assembly time based upon the order in which variables are declared in the source code and due to the dimension of representation inferred from the associated type information.

3.2.1.2. Register operands

Direct using - **mov eax, ebx**
Indirect usage and addressing – used for pointing to memory locations - **mov eax, [ebx]**

3.2.1.3. Memory addressing operands

There are 2 types of memory operands: *direct addressing operands* and *indirect addressing operands*.

The *direct addressing operand* is a constant or a symbol representing the address (segment and offset) of an instruction or some data. These operands may be *labels* (for ex: jmp et), *procedures names* (for ex: call proc1) or *the value of the location counter* (for 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***. The actual physical address is computed ***at loading time***.

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** 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;

Explicit segment register specification is done using the segment prefix operator "**:**"
ES can be used only in explicit specifications (like ES:[Var] or ES:[ebx+eax*2-a]) or IN CERTAIN STRING INSTRUCTIONS (MOVSB)

JMP FAR CS:…
JMP FAR DS:… or JMP FAR [label2]

3.2.1.4. Indirect addressing operands

Indirect addressing operands use registers for pointing to memory addresses. Because the actual registers values are known only at run time, indirect addressing is suited for dynamic data operations.

The general form for indirectly accessing a memory operand is given by the offset computing formula:

[base_register + index_register* scale + constant**]**

Constant is an expression which value is computable at assembly time. For ex. [ebx + edi + table + 6] denotes an indirect addressed operand, where both *table* and *6* are constants.

The operands *base_register* and *index_register* are generally used to indicate a memory address referring to an array. In combination with the scaling factor, the mechanism is flexible enough to allow direct access to the elements of an array of records, with the condition that the byte size of one record to be 1, 2, 4 or 8. For example, the upper byte of the DWORD element with the index given in ECX, part of a record vector which address (of the vector) is in edx can be loaded in dh by using the instruction

```
mov dh, [edx + ecx * 4 + 3]
```

From a syntactic point of view, when the operand is not specified by the complete formula, some of the components missing (for example when "* scale" is not present), the assembler will solve the possible ambiguity by an analysis process of all possible equivalent encoding forms, choosing the shortest finally. For example, having

```
push dword [eax + ebx]   ; saves on the stack the doubleword from the address eax+ebx
```

the assembler is free to consider eax as the base and ebx as an index or vice versa, ebx as the basis and eax as index.

In a similar way, for

```
pop DWORD [ecx] ; restores the top of the stack in the variable which address is given in ecx
```

the assembler can interpret ecx either as a base or as an index. What is really important to keep in mind is that all codifications considered by the assembler are equivalent and its final decision has no impact on the functionality of the resulted code.

Also, in addition to solving such ambiguities, the assembler also allows non-standard expressions, with the condition to be in the end transformable into the above standard form. Other examples:

```
lea eax, [eax*2]          ; load in eax the value of eax*2 (which is, eax becomes 2*eax)
```

In this case, the assembler may decide between coding as base = eax + index = eax and scale = 1 or index = eax and scale = 2.

```
lea eax, [eax*9 + 12]     ; eax will be eax * 9 + 12
```

Although the scale cannot be 9, the assembler will not issue an error message here. This is because it will notice the possible encoding of the address like: base = eax + index = eax with scale = 8, where this time the value 8 is correct for the scale. Obviously, the statement could be made clearer in the form

```
lea eax, [eax + eax * 8 + 12]
```

For indirect addressing it is essential to specify between square brackets at least one of the components of the offset computation formula.

3.2.2. Using operators

Operators – used for combining, comparing, modifying and analyzing the operands. Some operators work with integer constants, others with stored integer values and others with both types of operands.

It is very important to understand the difference between operators and instructions. **Operators perform computations only with constant SCALAR values computable at assembly time (scalar values = constant immediate values), with the exception of adding and/or subtracting a constant from a pointer (which will issue a pointer data type) and with the exception of the offset computation formula (which supports the '+' operator).** Instructions perform computations with values that may remain unknown (and this is generally the case) until run time. Operators are relatively similar with those in C. Expression evaluation is done on 64 bits, the final results being afterwards adjusted accordingly to the sizeof available in the available usage context of that expression.

For example the addition operator (+) performs addition at assembly time and the ADD instruction performs addition during run time. We give below the operators that are used by the x86 assembly language expressions in NASM !.

Priority	Operator	Type	Result
7	-	unary, prefix	Two's complement (negation): -X = 0 – X
7	+	unary, prefix	No effect (provides simetry to „-"): +X = X
7	~	unary, prefix	One's complement: mov al, ~0 => mov AL, 0xFF
7	!	unary, prefix	Logic negation: !X = 0 when X ≠ 0, else 1
6	*	Binary, infix	Multiplication: 1 * 2 * 3 = 6
6	/	Binary, infix	Result (quotient) of unsigned division: 24 / 4 / 2 = 3
6	//	Binary, infix	Result (quotient) of signed division: -24 // 4 // 2 = -3 **(-24 / 4 / 2 ≠ -3!)**
6	%	Binary, infix	Remainder of unsigned division: 123 % 100 % 5 = 3
6	%%	Binary, infix	Remainder of signed division: -123 %% 100 %% 5 = -3
5	+	Binary, infix	Sum: 1 + 2 = 3
5	-	Binary, infix	Subtraction: 1 – 2 = -1

4	<<	Binary, infix	Bitwise left shift: 1 << 4 = 16
4	>>	Binary, infix	Bitwise right shift: 0xFE >> 4 = 0x0F
3	&	Binary, infix	AND: 0xF00F & 0x0FF6 = 0x0006
2	^	Binary, infix	Exclusive OR: 0xFF0F ^ 0xF0FF = 0x0FF0
1	|	Binary, infix	OR: 1 | 2 = 3

The indexing operator has a widespread use in specifying indirectly addressed operands from memory. The role of the [] operator regarding indirect addressing has been explained in Paragraph 3.2.1.

3.2.2.3. Bit shifting operators

expression >> how_many *and* *expression << how_many*

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

3.2.2.4. Bitwise operators

Bitwise operators perform bit-level logical operations for the operand(s) of an expression. The resulting expressions have constant values.

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

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

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

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

3.2.2.6. The segment specification operator

The segment specifier operator (:) performs the FAR address computation of a variable or label relative to a certain segment. Its syntax is: **segment:expression**

```
[ss: ebx+4]  ; offset relative to SS;
[es:082h]    ; offset relative to ES;
10h:var      ; the segment address is specified by the 10h selector,
               the offset being the value of the var label.
```

3.2.2.7. Type operators

They specify the types of some expressions or operands stored in memory. Their syntax is:

type expression

where the type specifier is one of the following: **BYTE**, **WORD**, **DWORD**, **QWORD** or **TWORD**

This syntactic form causes ***expression*** to be treated temporarily (limited to that particular instruction) as having ***„type"*** sizeof without destructively modifying its initial value. That is why these type operators are also called „non-destructive temporary conversion operators". For memory stored operators, *type* may be **BYTE**, **WORD**, **DWORD, QWORD or TWORD** having the size of 1, 2, 4, 8 and 10 bytes respectively. For code labels *type* is either **NEAR** (4 bytes address) or **FAR** (6 bytes address).

For example, **byte [**A] takes only the first byte from the memory location designated by A. Similar, **dword [**A] will consider the doubleword starting at address A.

3.3. DIRECTIVES

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

3.3.1.1. The SEGMENT directive

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

SEGMENT *name* [*type*] [ALIGN=*alignment*] [*combination*] [*usage*] [CLASS=*class*]

The numeric value assigned to the segment name is the segment address (32 bits) corresponding to the memory segment's position during run-time. For this purpose, NASM offers the special symbol $$ which is equal with the current segment's address, this having the advantage that can be used in any context, without knowing the current segment's name.

Except the name, all the other fields are optional both regarding their presence or the order in which they are specified.

The optional arguments *alignment*, *combination*, *usage* and *'class'* give to the link-editor and the assembler the necessary information regarding the way in which segments must be loaded and combined in memory.

The type allows selecting the usage mode of the segment, having the following possible values:

- **code** (or **text**) - the segment will contain code, meaning that the content cannot be written but it can be executed
- **data** (or **bss**) - data segment allowing reading and writing but not execution (implicit value).
- **rdata** - the segment that it can only be read, containing definitions of constant data

The optional argument *alignment* specifies the multiple of the bytes number from which that segment may start. The accepted alignments are powers of 2, between 1 and 4096.

If *alignment* is missing, then it is considered implicitly that ALIGN=1, i.e. the segment can start from any address.

The optional argument *combination* controls the way in which similar named segments from other modules will be combined with the current segment at linking time. The possible values are:

- **PUBLIC -** indicates to the link editor to concatenate this segment with other segments with the same name, obtaining a single segment having the length the sum of concatenated segments' lengths.
- **COMMON -** specifies that the beginning of this segment must overlap with the beginning of all segments with the same name, obtaining a segment having the length equal to the length of the larger segment with the same name.
- **PRIVATE** - indicates to the link editor that this segment cannot be combined with others with the same name.
- **STACK** - the segments with the same name will be concatenated. During run time the resulted segment will be the stack segment.

Implicitly, if no combination method is specified, any segment is PUBLIC.

The argument *usage* allows choosing another word size than the default 16 bits one.

The argument *'class'* has the task to allow choosing the order in which the link editor puts the segments in memory. All the segments that have the same class will be placed in a contiguous block of memory whatever their order in the source code. No implicit value exists, it being undefined when its specification is missing, leading though to NOT concatenating all the program's segments defined so in a continous memory block.

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

3.3.2. Data definition directives

Data definition = *declaration* (attributes specification) + *allocation* (reserving required mem. space).
(UNIQUE !!!) (it is NOT unique !!!) (UNIQUE !!!!)

In C – 17 modules (separate files) ; A1- global variable (int A1 + 16 data declarations extern int A1)
LINKER is responsible for checking the DEPENDENCIES between the modules !

The structure of a Variable = ([name], set_of_attributes, [address/reference, value])

Dynamic variable DOES NOT HAVE A NAME !!!!!

P=new(…); p=malloc(…); …free… (Diferenta between POINTER and DYNAMIC variables !!!!)

(Name, set_of_atrributes) = Formal parameters !!!!

Set_of_attributes = (type, Domeniu_de_vizibilitate (scope), lifetime (extent), memory class)
Memory class (in C) = (auto, register, static, extern)

data type = *size of representation* – byte, word, doubleword or quadword

The general form of a data definition source line is:

[*name*] *data_type* *expression_list* [**;***comment*]
or
[*name*] *allocation_type* *factor* [**;***comment*]
or
[*name*] **TIMES** *factor* *data_type* *expression_list* [**;***comment*]

where *name* is a label for data referral. The data type is the size of representation and its value will be the address of its first byte.

factor is a number which shows how many times the *expression_list* is repeated.

Data_type is a data definition directive, one of the following:

DB - byte data type (BYTE)
DW - word data type (WORD)
DD - doubleword data type (pointer - DWORD)
DQ - 8 bytes data type (QWORD – 64 bits)
DT - 10 bytes data type (TWORD – used to store BCD constants or real constants for extended precision)

For example, the following sequence defines and initializes 5 memory variables:

```
segment data
      var1  DB   'd'     ;1 byte
        .a  DW   101b ;2 bytes
      var2  DD   2bfh  ;4 bytes
        .a  DQ   307o ;8 bytes (1 quadword)
        .b  DT   100   ;10 bytes
```

Var1 and var2 variables are defined using common labels, visible in the entire source code, while .a and .b are local labels, the access to these local variables being restricted in the sense that:

- these variables can be accessed with the local name, i.e. .a or .b, until another common label is defined (they are local to preceding label);
- they can be accesed from anywhere by their complete name: var1.a, var2.a or var2.b.

The initialization value may be also an expression, as for example vartest DW (1002/4+1)

After a data definition directive may appear more than one value, thereby allowing declaration and initialization of arrays. For example, the declaration:

```
Tablou      DW  1,2,3,4,5
```

creates an array with 5 integers represented as words and having their values 1,2,3,4,5. If the values supplied after the directive don't fit on a single line, we can add as many lines as necessary, which shall contain only the directive and the desired values. Example:

```
Tabpatrate DD   0, 1, 4, 9, 16, 25, 36
           DD   49, 64, 81
           DD   100, 121, 144, 169
```

allocation_type is a uninitialized data reservation directive:

RESB - byte data type (BYTE)
RESW - word data type (WORD)
RESD - doubleword data type (DWORD)
RESQ - 8 bytes data type (QWORD - 64 bits)
REST - 10 bytes data type (TWORD – 80 bits)

factor is a number showing how many times the specified data type allocation is repeated.

For example Tabzero RESW 100h reserves 256 words for *Tabzero* array.

NASM does not support the MASM/TASM syntax of reserving uninitialized space by writing DW ?. The operand to a RESB-type pseudo-instruction is a **critical expression (all operands involved in computations must be known at expression evaluation time)**. Ex:

```
buffer:         resb    64      ; reserve 64 bytes
wordvar:        resw    1       ; reserve a word
realarray       resq    10      ; array of ten reals
```

TIMES directive allows repeated assembly of an instruction or data definition.

TIMES *factor data_type expression*

For example Tabchar TIMES 80 DB ‘a’

creates an ”array” of 80 bytes, every one of them being initialized with the ASCII code of 'a'.

matrice10x10 times 10*10 dd 0

will provide 100 doublewords stored continuously in memory starting from address associated with *matrice10x10* label.

TIMES can also be applied to instructions:

TIMES 32 add eax, edx ; having as effect EAX = EAX + 32*EDX

3.3.3. EQU directive

EQU directive allows assigning a numeric value or a string during assembly time to a label without allocating any memory space or bytes generation. The EQU directive syntax is:

name **EQU** *expression*

Examples:

```
END_OF_DATA      EQU '!'
BUFFER_SIZE      EQU 1000h
INDEX_START      EQU (1000/4 + 2)
VAR_CICLARE      EQU i
```

By use of such equivalence, the source code may become more readable.

You can see the similarity between the labels defined by the EQU directive and the symbolic constants defined in high level programming languages.

The expressions used when defining labels by EQU may also contain labels defined by EQU:

```
TABLE_OFFSET      EQU 1000h
INDEX_START       EQU (TABLE_OFFSET + 2)
DICTIONAR_STAR    EQU (TABLE_OFFSET + 100h)
```