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.

3.1. SOURCE LINE FORMAT

In the 8086 assembly language the source line format is:

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

The allowed characters for a *label* are A - Z a - z _ @ $? 0 - 9

1). ***Code labels***, present at the level of instructions sequences (code segments) for defining the destinations of the control transfer during a program execution.

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.

***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 difference between accessing the address and the contents of a named memory location is made **depending on the context of usage**. Example:

```
lea ax, v     ; loads in the ax register the address of the v variable
mov ax, v     ; loads in the ax  register the contents of the v variable
```

There are 2 *mnemonics cathegories*: *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 the operands representing registers contents and can be evaluated only at run time).

3.2.1. Addressing modes

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

The 3 operand types are: ***immediate operands, register operands*** and ***memory operands***. Their values are computed at assembly time for the immediate operands, at loading time for memory operands in direct addressing mode and at run time for the registers operands and for indirectly accessed memory operands.

3.2.1.1. Immediate operands

Immediate operands are constant numeric data computable at assembly time.

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.

```
lea ax, v     ; transfer in the AX register the offset of the v variable
```

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

```
lea ax, 0008      ; 8 bytes „distance" relative to the beginning of the data segment
```

In a similar way, jmp et will be evaluated as jmp [0004]↓ meaning "jump 4 bytes lower from the current position".

3.2.1.2. Register operands

Direct addressing - **mov ax,bx**
Indirect addressing – used for pointing to memory locations - **mov ax,[bx]**
(*offset* = [BX | BP] + [SI | DI] + [*constant*])

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 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***.

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 operating data in a dynamic way.

The only registers that may be used in indirect addressing are: BX, BP (*base registers*), DI and SI (*index registers*). Any attempt of using other registers in indirect addressing will result in an syntax error (*Illegal addressing mode*).

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

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

Constant is an expression which value is computable at assembly time. For ex. [bx + di + table + 6] denotes an indirect addressed operand.

Regarding the *implicit rules for computing the segment address corresponding to a given offset* these are : if for computing the offset, BX is used or no base register is specified, then the processor will use **DS** as the implicit segment register. If BP is used, then the implicit segment register will be **SS**.

Any operator for addition (+,[],.) may be used for combining the offset with the base or index registers. For example, the following address specifications are all of them equivalent:

```
table [bx] [di] + 6        [bx+6][di] + table
6 + table [bx+di]          table [di] [bx] + 6
[table+bx+di] + 6          bx + di + table[6]
[bx][di].table + 6         di + table + bx[6]
```

If 2 registers are used in the specification, one of them must be a base register and the other one must be an index register. The following two instructions will result in a „syntax error”:

```
mov  ax, table [bx] [bp]        ;illegal - 2 base registers !
mov  ax, table [di] [si]        ;illegal - 2 index registers !
```

For indirect addressing, it is essential to specify between square brackets [ ] at least one register valid for computing the offset.

3.2.2. 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 values computable at assembly time. Instructions perform computations with values that may remain unknown (and most of them are in this cathegory) until run time. For example: the addition operator (+) may perform the addition only at assembly time; the ADD instruction performs the addition operation during run time.

The operators set in 8086 assembly language is (operators on one line have the same priority):

maximal	
1	**(), [], <>, LENGTH, SIZE, WIDTH, MASK**
2	**.(selector pentru membru al unei structuri)**
3	**HIGH,LOW**
4	**+,- (unar)**
5	**: (explicit segment specification)**
6	**PTR, OFFSET, SEG, TYPE, THIS**
7	***, /, MOD, SHL, SHR**
8	**+,- (binar)**
9	**EQ, NE, LT, LE, GT, GE**
10	**NOT**
11	**AND**
12	**OR, XOR**
13	**SHORT, .TYPE, SMALL, LARGE**
minimal	

3.2.2.1. Arithmetic operators

OPERATOR	SYNTAX	SEMANTICS

+	+ expression	positive (unary)
-	- expression	negative (unary)
*	expression1*expression2	multiplication
/	expression1/ expression2	integer division
MOD	expr1 MOD expr2	remainder (modulo)
+	expression1+expression2	addition
-	expression1- expression2	subtraction

3.2.2.2. Indexing operator

The indexing operator (**[]**) shows an addition. It is similar in semantics to the addition operator (+). Its syntax is [*expresie_1*] [*expresie_2*]

3.2.2.3. Bits shifting operators

expression **SHR** *how_many* and *expression* **SHL** *how_many*

```
mov ax, 01110111b  SHL  3 ; puts in AX the value  10111000b
add  bx, 01110111b  SHR  3 ; adds to the value in BX  the value  00001110b
```

3.2.2.4. Bitwise operators

Bitwise operators perform logical operations for every bit of the given operand. The resulting expressions have constant values.

OPERATOR	SINTAXA	SEMANTICS

NOT	NOT expression	complements every bit
AND	expr1 AND expr2	AND bit by bit
OR	expr1 OR expr2	OR bit by bit
XOR	expr1 XOR expr2	exclusive OR bit by bit

Examples (assuming that the expression is represented on 1 byte):

```
NOT          11110000b          ; gives 00001111b
01010101b  AND  11110000b    ; gives 01010000b
01010101b  OR  11110000b      ; the output result is 11110101b
01010101b  XOR 11110000b     ; the output result is 10100101b
```

3.2.2.5. Relational operators

A relational operator compares (with sign!) 2 expressions and returns as result the value -1 (TRUE) when the specified condition is met or 0 (FALSE) otherwise. The truth value -1 is represented in complementary code as a string of bits, all having the value 1. The evaluated expressions have constant values. The name and semantics of these operators are taken from the FORTRAN programming language. These are: **EQ** (***EQ**ual*), **NE** (*Not **E**qual*), **LT** (***L**ess **T**han*), **LE** (***L**ess or **E**qual*), **GT** (***G**reater **T**han*) şi **GE** (***G**reater or **E**qual*). Examples:

```
4 EQ 3  ; false (0)          - 4 LT 3  ; true  (-1)
4 NE 3  ; true  (-1)         - 4 GT 3  ; false (0)
```

3.2.2.6. The segment specification operator

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

```
ss:[bx+4]   ; offset relative to SS
```

es:082h ; offset relative to ES
date:var ; the segment address is the starting address of the segment named *date*, the offset being the value of the *var* label.

3.2.2.7. Type operators

They specify or analyze the types of some expressions or operands stored in memory.

The PTR operator

The **PTR** operator specifies the type (memory size) for a variable or code label. Its syntax is:

type **PTR** *expression*

The PTR operator forces *expression* to be used as having the specified *type* (temporary type coercion) without destructively modifying its value. So, PTR is a *non-destructive and temporary conversion operator*. For the stored memory operands, *type* may be **BYTE**, **WORD**, **DWORD** having the size 1, 2 and 4 bytes respectively. For the code labels it can be **NEAR** (2 bytes address) or **FAR** (4 bytes address).

Expression **byte ptr** A will point to the first byte from the address specified by A.
In a similar way, **dword ptr** A specifies the doubleword starting at A.

Operator THIS creates an operand which offset and segment's values are the actual value of the location counter : **THIS** *type*

So, the specification **THIS** *type* is similar to *type* **PTR $**.

Such forms are usually specified for initializing some symbols with the size of an array. For example:

lg dw this word – a is a definition equivalent to

```
lg      dw    word  ptr  $ – a
```

Operators HIGH and LOW return the most significant byte and the least significant byte respectively of a word sized constant expression. The syntax is:

HIGH *expression* and **LOW** *expression*

Operators SEG and OFFSET

The syntax of these two operators are:

SEG *expression* and **OFFSET** *expression*

where *expression* addresses directly a memory location.

The **SEG** operator returns the segment address of the referenced memory location. The value returnd by the **OFFSET** operator is a constant representing the number of bytes between the beginning of the segment and the referenced memory location. The values returned by these two operators are computed at program's loading time, remaining unchanged during run time. Example: let's suppose that the V label has the FAR address 5AFDh:0003. Then we have : SEG (V+5) = 5AFDh and OFFSET (V+5) = 0008.

Because direct addressing in the registers case means using directly their contents, as a special case of SEG and OFFSET operators we have :

```
SEG register = 0                Ex:  mov bx, SEG ax       ;bx:=0
OFFSET register = register           mov bx, OFFSET ax    ;bx:=ax
```

Also, because the name of a segment is a label having as value the starting address of that segment, we have:

```
SEG segment_name = segment_name     Ex:  mov bx, SEG data      ;mov bx,data
```

OFFSET segment_name = 0 mov bx, OFFSET data ;mov bx,0

3.3. DIRECTIVES

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

3.3.1. Standard directives for segments definition

There are 2 types of segment directives: *standard* segment directives (**SEGMENT, ENDS, ASSUME**) and *simplified* segment directives.

3.3.1.1. The SEGMENT directive

The beginning of a program's segment is defined by the **SEGMENT** directive and its end is defined by the **ENDS** directive. The general syntax of a segment definition is:

name **SEGMENT** [*alignment*] [*combination*] [*usage*] [*'class'*]

[*instructions*]
name **ENDS**

The name of the segment is defined by the *name* label. This name has as its associated value the segment address (16 bits) corresponding to the memory position of the segment during run time.

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

3.3.1.2. The ASSUME directive and the management of the memory segments

The **ASSUME** directive declares which are the active segments at a given moment :

ASSUME CS:*name1*, SS:*name2*, DS:*name3*, ES:*name4*

The main task of the ASSUME directive is to specify to the assembler the segment registers to be used for the effective address computation of data and code labels used in the program.

Explicit prefixing labels and variables with the name of a segment register provides in an immediate way the information relative to the segment address corresponding to that offset, overriding therefore the association provided by the ASSUME directive.

Specifying ASSUME is not necessary if the program doesn't use labels and variables (but such programs are obvious very rare and unusual....).

Keep in mind that ASSUME directive **DOES NOT ALSO** load the segment registers with the corresponding addresses !!!!!!! It does only specify the corresponding association in the implicit case.

```
ASSUME CS:c                ;associates the segment register CS with the code segment c
c   segment
start: jmp far ptr etd     ;unconditional far jump to the etd label from the d segment code
                           . . .   ;(we don't need ASSUME here because etd is forced to be considered
                           FAR)
       etc:    jmp  x      ;unconditional NEAR jump to the local label x (we need ASSUME here
                           ;for correctly compose the physical address CS:x)
       . . .
       x:      .  .  .
c      ends
```

ASSUME CS:d
;performs a new association of the CS segment register, this time with the d segment. The old one is canceled.

```
d  segment
        etd:   jmp y          ;unconditional NEAR jump to the y local label (we need ASSUME here
                              ;for correctly compose the physical address CS:y)
        . . .
        y:     jmp far ptr etc   ; unconditional far jump to the etc label from the c segment code
d  ends
end start
```

Remark. If the code segment doesn't have local labels refferals, ASSUME is no longer needed.

```
c    segment
start:          jmp far ptr etd  ; unconditional far jump to the etd label from the d segment code
        etc:    .   .   .        ;ASSUME is not needed here because the local label etc is referred only as
                                  a FAR label from the d segment
c    ends

d    segment
        etd:    jmp far ptr etc    ; unconditional far jump to the etc label from the c segment code
                .   .   .
d    ends
end  start
```

Regarding the loading of the segment registers with the appropriate values, we have:

- CS is automatically loaded

- SS is also automatically loaded, but if the programmer wants to change the stack segment, then he must load explicitly the new segment value in SS
- DS and ES, in **.EXE** programs must be loaded explicitly by the programmer.

Any assembly language program must finish with the **END** directive (the end of the source code). Any further lines will be ignored by the assembler. Its syntax is

END [*start_address*]

3.3.2. Data definition directives

Data definition = *declaration* (attributes specification) + *allocation* (reserving the required memory space).

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

The general form of a data definition source line is:

[*name*] *data_type* *expression_list* [**;***comment*]

or

[*name*] *data_type factor* **DUP (***expression_list***)** [**;***comment*]

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

factor is a number which shows how many times the *expression_list* is DUPlicated in the data generation process.

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)

For example, the sequence below defines and initializes 3 memory variables:

```
data  segment
      varb  DB   'd'     ;1 byte
      varw DW   101b ;2 bytes = 1 word
      vard  DD   2bfh  ;4 bytes = 2 words = 1 doubleword
data ends
```

Assembly language does not have arrays as data structures. But the basic mechanism of address computation offers „indexing" (a[i] – memory location addressing based on the starting address of a certain memory area – a C language concept also – the name of an array represents the starting address of its allocated memory area), so programmers can in fact use arrays, even the assembly language is not aware of such a data structure.

```
Tabpatrate DD   0, 1, 4, 9, 16, 25, 36    ; "Array" definition
           DD   49, 64, 81                ; a[4] = 16, a[9] = 81, a[11]=121
           DD   100, 121, 144, 169
```

The **DUP** operator is used for defining memory blocks initialized repetitively with a certain value. For example:

```
Tabzero     DW   100h  DUP (0)
```

allocates 256 words starting from *Tabzero* offset initializing them with 0 and

```
Tabchar     DB    80 DUP ('a')
```

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

If we want just memory allocation without initialization :

```
Tabzero   DW  100h DUP (?)
Tabchar   DB  80 DUP (?)
```

The assembler considers similar declarations as:

```
sirchar       DB  'a','b','c','d'
sirchar       DB  'abcd'
```

The initialization value may be also an expression evaluated to a constant , as for example:

```
vartest   DW  (1002/4+1)
```

The actual value of the location counter may be accessed (always in ”read-only” mode) using the **$** symbol or the expression **this** <*type*>. So, we may have the following equivalent sequences:

```
tabcuv    DW  50 DUP (?)          tabcuv    DW  50 DUP (?)
lungtab   DW  $-tabcuv            lungtab   DW  this word-tabcuv

tabcuv    DW  50 DUP (?)          tabcuv    DW  50 DUP (?)
lungtab   DW  lungtab-tabcuv      lungtab   EQU this word-tabcuv
```

3.3.3. LABEL, EQU, = directives

LABEL directive allows naming a location without allocating memory space for it or generating bytes for this purpose and allows accessing data using a different data type than the defining one. The syntax is

name **LABEL** *type*

where *name* is a symbol not defined previously in the source code and *type* describes the size of the symbol and if it reffers to code or data.

'name' will have as its value the value of the location counter. *Type* may be:

BYTE	**NEAR**	**FAR**
WORD	**QWORD**	**PROC**
DWORD	**TBYTE**	**UNKNOWN**

```
data segment                          code segment
      .     .     .                         .     .     .
      varcuv LABEL WORD                           mov ax, varcuv
            DB 1,2
      .     .     .                         .     .     .
data ends
```

UNKNOWN type declares an unknown type and it is used when intended to access a memory variable in different ways during the same run (similar to **void** in C).

```
data segment                          code segment
      tempvar LABEL UNKNOWN                     .     .     .
            DB ?,?                    mov  tempvar,ax       ;used as a word
      .     .     .                   .     .     .
data ends                             add  dl,tempvar       ;used as a byte
                                      .     .     .
```

Another way of accessing data with another type than that in definition (conversion, coercion) is the PTR operator (see 3.2.2.7).

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 syntax is

name **EQU** *expression*

Examples:

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

Similar to symbolic constants in high level programming languages, the source code may become though more readable.

The **"="** directive is similar to **EQU** with the following 2 differences:

- labels defined with EQU can't be redefined , while those defined by **"="** can.
- the operands used with **"="** must provide in the end a numeric value, assigning strings to labels being forbidden.

3.3.4. The PROC directive.

In the 8086 assembly language a subroutine definition starts with a **PROC** directive:

<*procedure_name*> **PROC** [*call_type*]

where *procedure_name* is a label denoting the procedure name and *call_type* is **NEAR** or **FAR**. If *call_type* is missing then the implicit value is NEAR.

ENDP directive marks the end of a subroutine started with **PROC**. Its syntax is

<*nume_procedură*> **ENDP**

3.3.5. Repetitive blocks.

A *repetitive block* is a construction by which the assembler is required to generate in a repetitive manner a certain bytes configuration. There are 3 types of repetitive blocks: **REPT**, **IRP** and **IRPC**.

A repetitive block is delimited by **REPT** and **ENDM** directives and has the syntax:

REPT *counter*
sequence
ENDM

meaning that *sequence* will be assembled *counter* times. For example the sequences

```
                                dw   0
REPT      5                     dw   0
      dw  0      and            dw   0
ENDM                            dw   0
                                dw   0
```

generate the same source code, action that may be accomplished also with : dw 5 DUP (0)

The following example does not have such a simple equivalent. It generates 5 consecutive memory locations containing , in order, the values from 0 to 4. We use for this purpose the **"="** directive:

```
Intval = 0              which will generate   dw   0
REPT      5                                   dw   1
      dw  Intval                              dw   2
      Intval = Intval + 1                     dw   3
ENDM                                          dw   4
```

Also, repetitive blocks may be nested. The sequence

```
REPT      5
```

```
REPT    2
   sequence
ENDM
ENDM
```

generates 10 times the specified sequence.

The **IRP** directive has the syntax

IRP *parameter*, <*arg1* [,*arg2*]...>
sequence
ENDM

It repetitively performs the assembling of the specified sequence, once for every argument from the arguments list, textually replacing every parameter presence in the sequence with the current argument. Arguments may be strings, symbols or numeric values. For example,

```
    IRP  param,<0,1,4,9,16,25>                    db   0
         db param                            db   1
    ENDM                                     db   4
                  generates the sequence     db   9
                                             db   16
                                             db   25
```

```
and IRP  reg,<ax,bx,cx,dx>                    mov  ax,di
         mov  reg,di                         mov  bx,di
    ENDM                                     mov  cx,di
                  generates the sequence     mov  dx,di
```

IRPC has a similar effect, accomplishing the textual replacement of the parameter, in turns, with every character from a given string. Its syntax is

```
IRPC    parameter,string
        sequence
ENDM
```

For example
```
IRPC nr,1375
    db nr
ENDM
```

will generate 4 bytes having the values 1, 3, 7 and 5 respectively.

3.3.6. The INCLUDE directive

Syntax: **INCLUDE** *filename*

Ex:
```
code segment
     mov  ax,1
     INCLUDE  INSTR2.ASM
     push ax
.    .    .
```

The included files may contain other INCLUDE directives and so on until any level of inclusion, the included source lines being inserted correspondingly for building in the end a single source file.

3.3.7. Macros

A *macro* is a parameterized text with a name assigned to it. Any time that name is invoked the assembler inserts in the source code the corresponding updated text. This operation is called *macro expansion*. The mechanism is similar to INCLUDE, but the big difference is that macros have in addition a great flexibility allowing **parameter passing** and **local labels**.

A macro is defined by the **MACRO** and **ENDM** directives:

name **MACRO** [*parameter* [,*parameter*]...]
instructions
ENDM

For example, interchanging the values of two word variables may be accomplished by the macro:

```
swap MACRO  a,b
     mov  ax,a
     mov  a,b
     mov  b,ax
ENDM
```

For multiplying by 4 the value of a word variable (the output result being stored in DX:AX) we may write the following macro:

```
inmcu4  MACRO  a
     mov  ax,a
     sub   dx,dx
     shl    ax,1
     rcl    dx,1
     shl    ax,1
     rcl    dx,1
ENDM
```

The usage **inmcu4 varm** will generate the sequence

```
mov  ax,varm
sub   dx,dx
shl    ax,1
rcl    dx,1
shl    ax,1
rcl    dx,1
```

Macros may contain repetitive blocks. We may rewrite the above macro as following:

```
inmcu4  MACRO  a
        mov  ax,a
        sub   dx,dx
      REPT  2
        shl    ax,1
        rcl    dx,1
      ENDM
ENDM
```

A potential problem refers to defining a local label inside a macro. If so and the program calls more than once that macro, then the SAME defined label will be generated at every macro expansion, causing a „label redefinition" syntax error. Example:

```
scade  MACRO
       jcxz  Etich
       dec   cx
Etich:.....
ENDM
.      .      .
scade              ;label Etich generated here
.      .      .
scade              ;label Etich generated here ALSO !!!!!
.      .      .
```

The solution is given by the **LOCAL** *directive*, which forces the local labels to have as their scope (visibility domain) only that macro. For the above example, the correct version is therefore:

```
scade  MACRO
       LOCAL Etich
       jcxz   Etich
       dec    cx
Etich:.....
```

```
        ENDM
```

the two consecutive above calls being translated to

```
.       .       .
        jcxz  ??0000
        dec   cx
??0000:
.       .       .
        jcxz  ??0001
        dec   cx
??0001:
```

Ex:

```
push_reg   MACRO  rlitera
        push  &rlitera&x
ENDM
.       .       .
push_reg b
```

will generate **push bx**. The substitution operator **&** may be used also with IRP or IRPC. For example:

```
                                        push ax
IRP rlitera,<a,b,c,d>                   push bx
        push &rlitera&x                 push cx
ENDM                    generates       push dx
```

As we saw, macros may contain repetitive blocks. Also, macros may call other macros:

```
push_reg   MACRO   register
        push  register
ENDM
.       .       .
push_all_reg  MACRO
        IRP   reg,<ax,bx,cx,dx,si,di,bp,sp>
```

```
            push_reg  reg
        ENDM
    ENDM
```

Macro **push_all_reg** contains a repetitive block, which also contains a call to **push_reg** macro.