Laboratory Work 3

3 Intel x86 CPU instruction set

3.1Objectives

In this laboratory work, variable declarations and some assembly language instructions are described. First, the general syntax of an instruction is assembly is presented. The exercises in this laboratory work focus on using the presented instructions.

3.2Variable declaration

In assembly language, a variable is a memory location. By declaring a variable, a specific space is reserved in the memory and a symbolic name is attached to the physical address of the variable. The syntax is the following:

<variable_name>   DB|DW|DD|DQ <value1>, [<value2>, <value3>...]

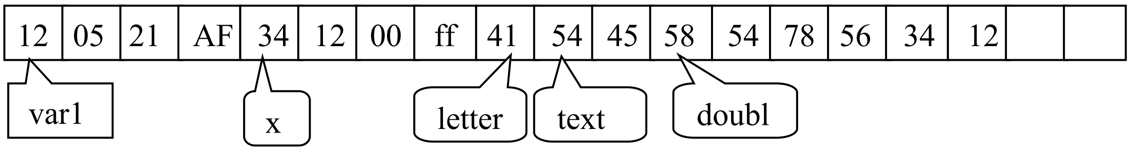
where:

<variable_name> - symbolic name of the variable
DB – Data Byte – 8 bit size
DW – Data Word – 16 bit size or 2 bytes
DD – Data Double – 32 bit size or 4 bytes
DQ – Data Quadruple – 64 bit size or 8 bytes
<value$_i$> - a constant value

Examples:

```
var1   db     12h,5, 33, 10101111b
x      dw     1234h, 0ff00h
letter db     'A'
text   db     "TEXT"
double dd     12345678h
```

Following the above declarations, the memory contains the following:



12	05	21	AF	34	12	00	ff	41	54	45	58	54	78	56	34	12		

Another way of reserving a memory block and initializing it with a value or a series of values is the following:

<variable_Name>   DB|DW|DD|DQ <number> DUP(<value1>[,<value2>..] |?)

where: <number> - indicates how many locations are reserved
<value$_i$> - initializing values
? – a zone which is not initializing

Examples:

```
bloc    db      100 dup(0)      ; 100 bytes of value 0
xx      dw      20 dup(0ffffh) ; 20 memory locations, of size word (16 bits) and initialized with
a value of FFFFh
buffer  dd      5 dup(?)        ; 5 double words (32 bits each) are reserved
```

Constants are declared in the same manner as a variable, but some specific rules apply:

```
<constant_name>         EQU   <value>
```

where: <constant_name> - symbolic name of the constant
EQU – mnemonic for constants ("equals")
<value> - a constant value

Examples:

```
        one     equ     1
        true    equ     0ffh
        false   equ     0
        adr_io  equ     300h            ; input/output port address
        mask    equ     00100000b       ; mask for selection of bit D5
        .....
        array   db      1,2,3,....
        array_length    equ     $-tablou        ; $-variable is a construction that calculates the
size of the memory location
```

3.2 Assembly language instruction syntax

One instruction occupies one program line and it contains several fields (the brackets indicate that a field can be optional):

```
[<label>:]      [<mnemonic> [<parameter_1> [,<parameter_2>]]    [;<comment>]
```

- <Label> - symbolic name or identifier given in front of an instruction, can have letters, numbers and special characters.
- <Mnemonic> - combination of letters that symbolizes a specific instruction, they are reserved words, cannot be used for other purpose.
- <parameter_1> - the first operand of an instruction and also the destination of the result, can be a register, memory address, variable or an expression that generates a memory address.
- <parameter_2> - the second operand of an instruction, can be anything as described above, in the first parameter, plus it can also be an immediate value/constant.
- <Comment> - ignored by the compiler, a comment symbol is valid for the entire code line.

3.2.1 Instruction types

3.2.1.1 Transfer instructions

These instructions transfer data between two registers, one register and a memory location or an immediate value is stored in a register or variable. Memory-to-memory transfers are not allowed and both parameters must have the same size.

MOV instruction

```
 [<label>:]      MOV    <parameter_1>, <parameter_2>     [;<comment>]
```

where:

<parameter_1> = <register>|<reg_segment>|<offset_address>|<variable_name>|<expression>
<parameter_2> = <parameter_1>|<constant>
<register> = EAX|EBX|.....ESP|AX|BX|....SP|AH|AL|....DL
<expression> = **[**[<index_register>][+<base_register >][+<offset>]**]**
; the bold brackets are mandatory
<index_register> = SI| DI |ESI | EDI
<base_register> = BX|BP |EBX| EBP
<offset> = <constant>

Examples:

```
        mov ax,bx                   et1:    mov ah, [si+100h]
        mov cl, 12h                         mov al, 'A'
        mov dx, var16                       mov si, 1234h
        mov var32,eax               sf:     mov [si+bx+30h], dx
        mov ds, ax                          mov bx, cs
```

Syntax error examples:

```
        mov ax, cl              ; different sizes of the operands
        mov var1, var2          ; both operands are memory locations
        mov al, 1234h           ; the immediate value/constant is higher than the size of the first
                   operand
```

LEA instruction

These instructions allow loading in a register a variable address. The first instruction LEA (Load Effective Address) loads in a register (the first parameter) the offset address of the variable from the 2nd parameter.

Examples:

```
        lea  si, var1           ; SI<= offset(var1)
        lea di, [bx+100]        ; DI<= BX+100
```

XCHG instruction

Exchange the contents of the 2 operands.

```
XCHG <parameter_1>, <parameter_2>
```

Note: the second parameter cannot be a constant.

Examples:

```
xchg al, bh
xchg ax,bx
```

PUSH and POP instructions

These 2 instructions work by default with the top level of the stack. PUSH adds an operand on the stack, while POP extract the top level value from the stack and stores is in the specified operand. Please note that the stack for x86 works with sizes of 16 and 32 bit.

```
PUSH <parameter_1>
POP <parameter_1>
```

Examples:

```
push ax          push var16
pop bx           pop var16
```

3.2.1.2 Arithmetic instructions

ADD and ADC instructions

These 2 instructions perform the addition of 2 operands and store the result in the first operand. The ADC instruction also takes into account the carry flag value.

```
ADD <parameter_1>,<parameter_2>
ADC <parameter_1>,<parameter_2>
```

Examples:

```
add ax, 1234h
add bx, ax
adc dx, var16
```

SUB and SBB instructions

SUB – subtract the second operand from the first, store the result in the first parameter
SBB – subtract with borrow, also subtracts the carry flag value.

```
SUB   <parameter_1>,<parameter_2>
SBB   <parameter_1>,<parameter_2>
```

MUL and IMUL instructions

These instructions perform multiplication for unsigned numbers (MUL) and for signed numbers (IMUL). These instructions have only one parameter because, by default, register EAX is used as the first operand (aka. also stores the result).

```
MUL <parameter_2>
IMUL <parameter_2>
```

Examples:

```
mul dh        ; AX<=AL*DH
mul bx        ; DX:AX<= AX*BX    DX is the extension of AX register
imul var8     ; AX<=AL*var8
```

DIV and IDIV

These instructions perform division for unsigned and signed numbers and, by default, use register EAX for division and result storage.

```
DIV <parameter_2>
IDIV <parameter_2>
```

Examples:

```
div   cl      ; AL<=AX/CL and AH<=AX modulo CL (residue of the division)
div   bx      ; AX<= (DX:AX)/BX and DX<=(DX:AX) modulo BX
```

INC and DEC

Perform increase or decrease of the specified parameter (addition or subtraction with a value of one).

```
INC <parameter>
DEC <parameter>
```

Examples:

```
inc   si      ; SI<=SI+1
dec  cx       ; CX<=CX-1
```

CMP instruction

Compares the two operands by performing subtraction, but the result is not stored, ony the flags are modified. This instruction is usually followed by a conditional jump instruction.

```
CMP <parameter_1>, <parameter_2>
```

Example:

```
cmp ax, 50h
```

3.2.1.3 Logical instructions

These instructions implement the basic Boolean algebra, they are performed at bit level and the result is stored in the first operand.

AND, OR, NOT, XOR instructions

```
AND <parameter_1>,<parameter_2>
OR <parameter_1>, <parameter_2>
NOT <parameter_1>
XOR <parameter_1>,<parameter_2>
```

Examples:

```
and   al, 0fh
or    bx, 0000111100001111b
and   al,ch
```

```
xor   ax,ax                 ; clear register AX
```

TEST instruction

Logical AND between the 2 operands, but the result is not stored. The purpose is to modify the condition indicators and to avoid the destruction of contents from the first operand.

```
TEST <parameter_1>,<parameter_2>
```

Examples:

```
test al, 00010000b          ;checks if bit D4 from AL is zero or not
test bl, 0fh                ;checks if the first hexa digit of BL is 0
```

3.3Exercises

1. Please follow the online document at: http://users.utcluj.ro/~madalin/SM/labs/W3.pdf