Laborator 2 - Arithmetic Expressions in assembly language

Contents

1. Addition: quadword+quadword

```
bits 32 ; assembling for the 32 bits architecture
; declare the EntryPoint (a label defining the very first instruction of the program)
global start

; declare external functions needed by our program
extern exit             ; tell nasm that exit exists even if we won't be defining it
import exit msvcrt.dll    ; exit is a function that ends the calling process. It is defined in msvcrt.dll
                        ; msvcrt.dll contains exit, printf and all the other important C-runtime specific functions
; our data is declared here (the variables needed by our program)
segment data use32 class=data

    a dq 1122334455667788h
    b dq 0abcdef1a2b3c4d5eh
    r resq 1 ;  reserve 1 quadword in memory  ; r dq 0
            ;  to save the rezult
```

1 byte = 8biti

1 word = 2 bytes = 16 biti

1 doubleword = 2 worduri=4 bytes =32 biti

a

88	77	66	55	44	33	22	11	Bytes lui A
A+0	A+1	A+2	A+3	A+4	A+5	A+6	A+7	Adresele lui a

A in memory cf little-endian

```
; our code starts here
segment code use32 class=code
    start:

        ;11223344 55667788 h -> EDX : EAX
        ;   EDX  :   EAX
        mov edx, dword [a+4] ; pune in edx 4 bytes incepand cu adresa [a+4]

        mov eax, dword [a+0] ; pune in eax 4 bytes incepand cu adresa [a+0]
```

[De exemplu: x dd 1a2b3c4dh ; x definit in data segment -> dx:ax, dx = 1a2bh si in ax=3c4dh
X in memorie cf little-endian

4d	3c	2b	1a	Bytes lui X
X+0	X+1	X+2	X+3	Adresele lui X

x-> dx:ax
mov dx, word[x+2]
mov ax, word[x+0]

salvare registrii in variabila, dx:ax->aux, aux dd 0
mov word[aux+0], ax
mov word[aux+2], dx

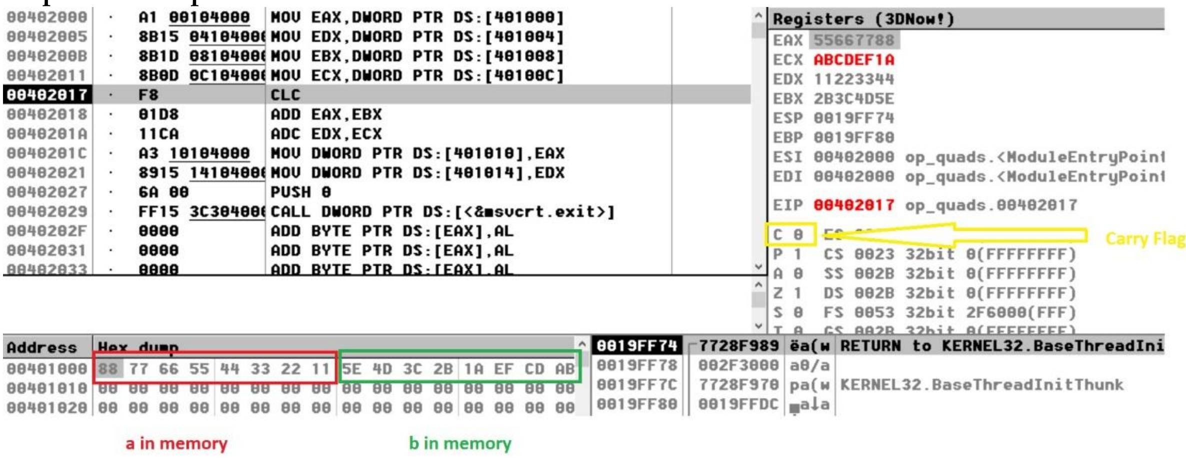
]

; b = abcdef1a 2b3c4d5e h -> ECX : EBX
; ECX : EBX
B in memory

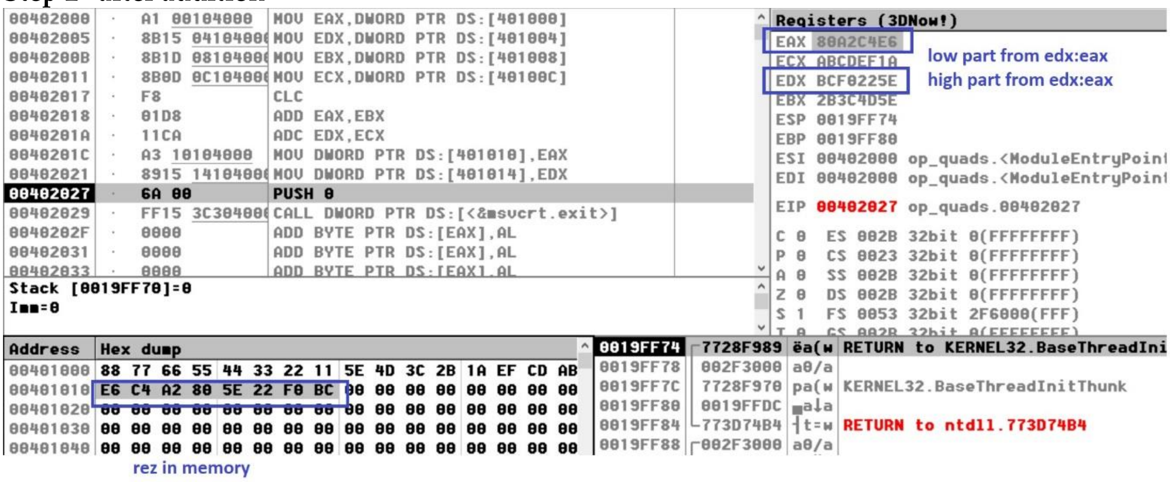
5e	4d	3c	2b	1a	ef	cd	ab	Bytes lui B
B+0	B+1	B+2	B+3	B+4	B+5	B+6	B+7	Adresele lui b

```
    mov ebx, dword [b+0] ; pune in ebx 4 bytes incepand cu adresa [b+0]
    mov ecx, dword [b+4]  ; pune in ecx 4 bytes incepand cu adresa [b+4]
; ecx:ebx si edx:eax

    ;a + b
    ; +tr
    ; edx : eax +
    ; ecx : ebx
    clc         ; clear Carry Flag (punem 0 in CF)
    add eax, ebx  ; eax= eax+ebx si se pune 1 in CF daca exista trasnport
    adc edx, ecx  ; edx = edx+ecx + CF
                ;(CF se seteaza daca add eax, ebx produce un transport)

   ;edx:eax -> r
    mov dword [r+0], eax
    mov dword [r+4], edx
    push    dword 0      ; push the parameter for exit onto the stack
   call    [exit]       ; call exit to terminate the program
```

Step 1 – before perform the addition



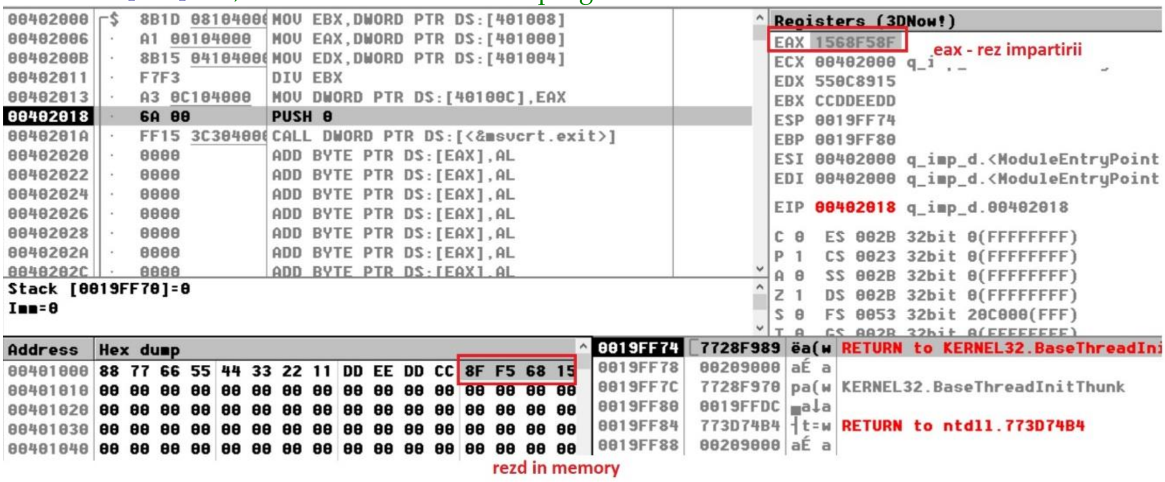
Step 2 –after addition



2. Division: quadword/doubleword

```
bits 32 ; assembling for the 32 bits architecture

; declare the EntryPoint (a label defining the very first instruction of the program)
global start

; declare external functions needed by our program
extern exit             ; tell nasm that exit exists even if we won't be defining it
import exit msvcrt.dll   ; exit is a function that ends the calling process. It is defined in msvcrt.dll
                         ; msvcrt.dll contains exit, printf and all the other important C-runtime specific functions

; our data is declared here (the variables needed by our program)
segment data use32 class=data

    m dq 1122334455667788h
    n dd 0ccddeeddh
    rezd resd 1  ; sau rezd dd 0
```

```
; our code starts here
segment code use32 class=code
    start:
        mov ebx, [n]

        ;11223344 55667788 h -> EDX : EAX
        ;  EDX  :   EAX
         mov eax, dword [m+0]
         mov edx, dword [m+4]

         div ebx   ; edx:eax/ebx=eax cat si edx rest

         mov dword[rezd], eax

        ; exit(0)
        push    dword 0      ; push the parameter for exit onto the stack
        call    [exit]       ; call exit to terminate the program
```



3. Transfer content from DX:AX in EAX

Method 1	Method 2
;stiva – memory area based on LIFO principle	;with aux variable
; stiva = dim standard 32 biti	aux dd 0 ; sau aux resd 1
Push dx	mov [aux+0], ax
Push ax	mov [aux+2], dx
Pop eax	
	mov eax, [aux]

4. Transfer content from EAX in DX:AX

Method 1	Method 2
;stiva – memory area based on LIFO principle	;with aux variable
	aux dd 0 ; sau aux resd 1
Push eax	mov [aux], eax
Pop ax	mov dx, word[aux+2]
Pop dx	mov ax, word[aux+0]]

Homework:

Solve the following expression in signed representation:

1. ((a+b-c)*2 + d-5)*d data types: a,b,c - byte, d - word and
second expression: c+(a*a-b+7)/(2+a), a-byte; b-doubleword; c-qword
2. d*(d+2*a)/(b*c) data types: a,b,c - byte, d - word and
second expression: 2/(a+b*c-9)+e-d; a,b,c-byte; d-doubleword; e-qword
3. [-1+d-2*(b+1)]/a data types: a,b,c - byte, d - word and
second expression: (8-a*b*100+c)/d+x; a,b,d-byte; c-doubleword; x-qword
4. –a*a + 2*(b-1) – d data types: a,b,c - byte, d - word and
second expression: (a*2+b/2+e)/(c-d)+x/a; a-word; b,c,d-byte; e-doubleword; x-qword
5. [d-2*(a-b)+b*c]/2 data types: a,b,c - byte, d - word and
second expression: (a+b/c-1)/(b+2)-x; a-doubleword; b-byte; c-word; x-qword
6. [2*(a+b)-5*c]*(d-3) data types: a,b,c - byte, d - word and
second expression: x+a/b+c*d-b/c+e; a,b,d-byte; c-word; e-doubleword; x-qword
7. [100*(d+3)-10]/d data types: a,b,c - byte, d - word and
second expression: (a-2)/(b+c)+a*c+e-x; a,b-byte; c-word; e-doubleword; x-qword
8. (100*a+d+5-75*b)/(c-5) data types: a,b,c - byte, d - word and
second expression: 1/a+200*b-c/(d+1)+x/a-e; a,b-word; c,d-byte; e-doubleword; x-qword
9. 3*[20*(b-a+2)-10*c]+2*(d-3) data types: a,b,c - byte, d - word and
second expression: (a-b+c*128)/(a+b)+e-x; a,b-byte; c-word; e-doubleword; x-qword
10. 3*[20*(b-a+2)-10*c]/5 data types: a,b,c - byte, d - word and
second expression: d-(7-a*b+c)/a-6+x/2; a,c-byte; b-word; d-doubleword; x-qword
11. [(d/2)*(c+b)-a*a]/b data types: a,b,c - byte, d - word and
second expression: (a+b)/(2-b*b+b/c)-x; a-doubleword; b,c-byte; x-qword
12. a*[b+c+d/b]+d data types: a,b,c - byte, d - word and
second expression: (a*b+2)/(a+7-c)+d+x; a,c-byte; b-word; d-doubleword; x-qword
13. [(a*b)-d]/(b+c) data types: a,b,c - byte, d - word and
second expression: x-(a+b+c*d)/(9-a); a,c,d-byte; b-doubleword; x-qword
14. (d-b*c+b*2)/a data types: a,b,c - byte, d - word and
second expression: x+(2-a*b)/(a*3)-a+c; a-byte; b-word; c-doubleword; x-qword
15. (a*2)+2*(b-3)-d-2*c data types: a,b,c - byte, d - word and
second expression: x-(a*b*25+c*3)/(a+b)+e; a,b,c-byte; e-doubleword
16. (a+b)/2 + (10-a/c)+b/4 data types: a,b,c - byte, d - word and
second expression: x/2+100*(a+b)-3/(c+d)+e*e; a,c-word; b,d-byte; e-doubleword; x-qword
17. [(a+b)*3]/c-d data types: a,b,c - byte, d - word and
second expression: (a+b)/(3-c)-d+2+x; a,b,c-byte; d-doubleword; x-qword
18. 200-[3*(c+b-d/a)-300] data types: a,b,c - byte, d - word and
second expression: (a+b*c+2/c)/(2+a)+e+x; a,b-byte; c-word; e-doubleword; x-qword
19. [(a-b)*3+c*2]-d data types: a,b,c - byte, d - word and
second expression: (a+a+b*c*100+x)/(a+10)+e*a; a,b,c-byte; e-doubleword; x-qword
20. (50-b-c)*2+a*a+d data types: a,b,c - byte, d - word and
second expression: x-b+8+(2*a-b)/(b*b)+e; a-word; b-byte; e-doubleword; x-qword
21. d-[3*(a+b+2)-5*(c+2)] data types: a,b,c - byte, d - word and
second expression: (a*a/b+b*b)/(2+b)+e-x; a-byte; b-word; e-doubleword; x-qword
22. [(10+d)-(a*a-2*b)]/c data types: a,b,c - byte, d - word and
second expression: a/2+b*b-a*b*c+e+x; a,b,c-byte; e-doubleword; x-qword
23. [(a+b)*3-c*2]+d data types: a,b,c - byte, d - word and
second expression: (a*b-2*c*d)/(c-e)+x/a; a,b,c,d-byte; e-word; x-qword
24. (10*a-5*b)+(d-5*c) data types: a,b,c - byte, d - word and
second expression: a-(7+x)/(b*b-c/d+2); a-doubleword; b,c,d-byte; x-qword
25. [100-10*a+4*(b+c)]-d data types: a,b,c - byte, d - word and

second expression: (a*a+b+x)/(b+b)+c*c; a-word; b-byte; c-doubleword; x-qword
26. d+[(a+b)*5-(c+c)*5] data types: a,b,c - byte, d - word and
second expression: (a*a+b/c-1)/(b+c)+d-x; a-word; b-byte; c-word; d-doubleword; x-qword
27. d/[(a+b)-(c+c)] data types: a,b,c - byte, d - word and
second expression: (100+a+b*c)/(a-100)+e+x/a; a,b-byte; c-word; e-doubleword; x-qword
28. d+10*a-b*c data types: a,b,c - byte, d - word and
second expression: x-(a*100+b)/(b+c-1); a-word; b-byte; c-word; x-qword
29. [d-(a+b)*2]/c data types: a,b,c - byte, d - word and
second expression: (a+b)/(c-2)-d+2-x; a,b,c-byte; d-doubleword; x-qword
30. [(a-b)*5+d]-2*c data types: a,b,c - byte, d - word and
second expression: a*b-(100-c)/(b*b)+e+x; a-word; b,c-byte; e-doubleword; x-qword
31. 300-[5*(d-2*a)-1] data types: a,b,c - byte, d - word and
second expression: x-(a*a+b)/(a+c/a); a,c-byte; b-doubleword; x-qword

Sinteza instructiuni pentru expresii aritmetice (sau toate instructiunile invatate pana acum)

Arithmetic Expressions (Expresii Aritmetice) in assembly language

<table>
<tr><th>Unsigned Representation (Reprezentare fara semn)</th><th>Signed Representation (Reprezentare cu semn)</th></tr>
<tr><td colspan="2">Instructions to perform standard operations</td></tr>
<tr><td colspan="2">ADD op1, op2
; op1 = op1+op2
; op1, op2 – same size/same data type (aceasi dimensiune/acelasi tip de data)
; op1 or op2 should be a register (op1 sau op2 trebuie sa fie registru)
;while both operand can be registers, at most one operand can be a variable
;(in timp ce ambii operanzi pot fi regiștri, cel mult un operand poate fi o variabila)</td></tr>
<tr><td colspan="2">ADC op1, op2
(Add with Carry)
; op1 = op1+op2+Carry Flag
; op1, op2 – same size/same data type (aceasi dimensiune/acelasi tip de data)
; op1 or op2 should be a register (op1 sau op2 trebuie sa fie registru)
;while both operand can be registers, at most one operand can be a variable
;(in timp ce ambii operanzi pot fi regiștri, cel mult un operand poate fi o variabila)</td></tr>
<tr><td colspan="2">SUB op1, op2
; op1 = op1-op2
; op1, op2 – same size/same data type (aceasi dimensiune/acelasi tip de data)
; op1 or op2 should be a register (op1 sau op2 trebuie sa fie registru)
;while both operand can be registers, at most one operand can be a variable
;(in timp ce ambii operanzi pot fi regiștri, cel mult un operand poate fi o variabila)</td></tr>
<tr><td colspan="2">SBB op1, op2
(Subtraction with borrow)
; op1 = op1-op2-Carry Flag
; op1, op2 – same size/same data type (aceasi dimensiune/acelasi tip de data)</td></tr>
</table>

<table>
<tr><td colspan="2">; op1 or op2 should be a register (op1 sau op2 trebuie sa fie registru)
;while both operand can be registers, at most one operand can be a variable
;(in timp ce ambii operanzi pot fi regiștri, cel mult un operand poate fi o variabila)</td></tr>
<tr><td>MUL op
op is called explicit operand

op8 - byte = mul op8 ; AX ← AL * op8
op16 - word = mul op16 ; DX:AX ← AX * op16
op32 - doubleword = mul op32 ; EDX:EAX ← EAX * op32

the explicit operand can be a register or a variable, but it cannot be a constant (operandul explicit poate sa fie registru sau variabila, dar nu poate sa fie o constanta)</td><td>IMUL op
op is called explicit operand

op8 - byte = imul op8 ; AX ← AL * op8
op16 - word = imul op16 ; DX:AX ← AX * op16
op32 - doubleword = imul op32 ; EDX:EAX ← EAX * op32

the explicit operand can be a register or a variable, but it cannot be a constant (operandul explicit poate sa fie registru sau variabila, dar nu poate sa fie o constanta)</td></tr>
<tr><td>DIV op
op is called explicit operand

div op8 ; AL ← AX / op8 , AH ← AX % op8
div op16 ; AX ← DX:AX / op16 , DX ← DX:AX % op16
div op32 ; EAX ← EDX:EAX / op32 , EDX ← EDX:EAX % op32

the explicit operand can be a register or a variable, but it cannot be a constant (operandul explicit poate sa fie registru sau variabila, dar nu poate sa fie o constanta)</td><td>IDIV op
op is called explicit operand

idiv op8 ; AL ← AX / op8 , AH ← AX % op8
idiv op16 ; AX ← DX:AX / op16 , DX ← DX:AX % op16
idiv op32 ; EAX ← EDX:EAX / op32 , EDX ← EDX:EAX % op32

the explicit operand can be a register or a variable, but it cannot be a constant (operandul explicit poate sa fie registru sau variabila, dar nu poate sa fie o constanta)</td></tr>
<tr><td colspan="2">INC op; op ← op + 1
The operand op can be a register or a variable, but it cannot be a constant (operandul op poate sa fie registru sau variabila, dar nu poate sa fie o constanta)</td></tr>
<tr><td colspan="2">DEC op; op ← op - 1
The operand op can be a register or a variable, but it cannot be a constant (operandul op poate sa fie registru sau variabila, dar nu poate sa fie o constanta)</td></tr>
<tr><td colspan="2">NEG op; op ← 0 - op
The operand op can be a register or a variable, but it cannot be a constant (operandul op poate sa fie registru sau variabila, dar nu poate sa fie o constanta)</td></tr>
<tr><td colspan="2">Conversions (Conversii)</td></tr>
<tr><td>Byte to Word (unsigned representation)

mov AH,0 ; for converting AL → AX
- there is no specific instruction to convert a byte in word in the unsigned representation</td><td>Byte to Word (signed representation)
CBW - converts the byte AL to the word AX in the signed interpretation, meaning that we extend the representation from 8 bits to 16 bits, by filling AH with the sign bit of AL</td></tr>
</table>

- in the unsigned representation, conversion is realized by filling with 0 the high byte	- CBW instruction does not have any explicitly specified operands because it is always converting AL → AX
Word to Doubleword (unsigned representation) mov DX,0 ; for converting AX → DX:AX - there is no specific instruction to convert a word in doubleword in the unsigned representation - in the unsigned representation, conversion is realized by filling with 0 the high word or doubleword	**Word to Doubleword (signed representation)** **CWD -** converts the word AX to the doubleword DX:AX in the signed interpretation, meaning that we extend the representation from 16 bits to 32 bits, by filling DX with the sign bit of AX - CWD instruction does not have any explicitly specified operands because it is always converting AX → DX:AX
Word to Doubleword Extended (unsigned representation) mov EAX, 0 mov AX, [value that we need to convert] - there is no specific instruction to convert a word in doubleword in the unsigned representation - first, we have to fill with 0 the EAX register and the we save in the word AX the value that we want to convert	**Word to Doubleword Extended (signed representation)** **CWDE -** converts the word AX to the doubleword EAX in the signed interpretation, , meaning that we extend the representation from 16 bits to 32 bits, by filling the high word of EAX with the sign bit of AX - CWDE instruction does not have any explicitly specified operands because it is always converting AX → EAX
Doubleword to Quadword (unsigned representation) mov EDX,0 ; for converting EAX → EDX:EAX - there is no specific instruction to convert a word in doubleword in the unsigned representation - in the unsigned representation, conversion is realized by filling with 0 the high doubleword	**Doubleword to Quadword (unsigned representation)** **CDQ** - converts the doubleword EAX to the qword EDX:EAX in the signed interpretation meaning that we extend the representation from 32 bits to 64 bits, by filling EDX (the high doubleword) with the sign bit of EAX - CDQ instruction does not have any explicitly specified operands because it is always converting EAX → EDX:EAX