The length of a string

The length is computed using the location counter $ (sentinel character) and the directive EQU.

The EQU directive in .asm allows to assign a value to a variable in data segment without using a memory location to store that value (is a declaration without a definition in memory).

We perform a subtraction between the $ and the initial address of a string.

The $ - the number of generated bytes in memory until $ in encountered in code

data segment

byte

```
s db 1, 2, 3, 4, 5
lens equ $-s
;lens equ 5-0 =5
```

word

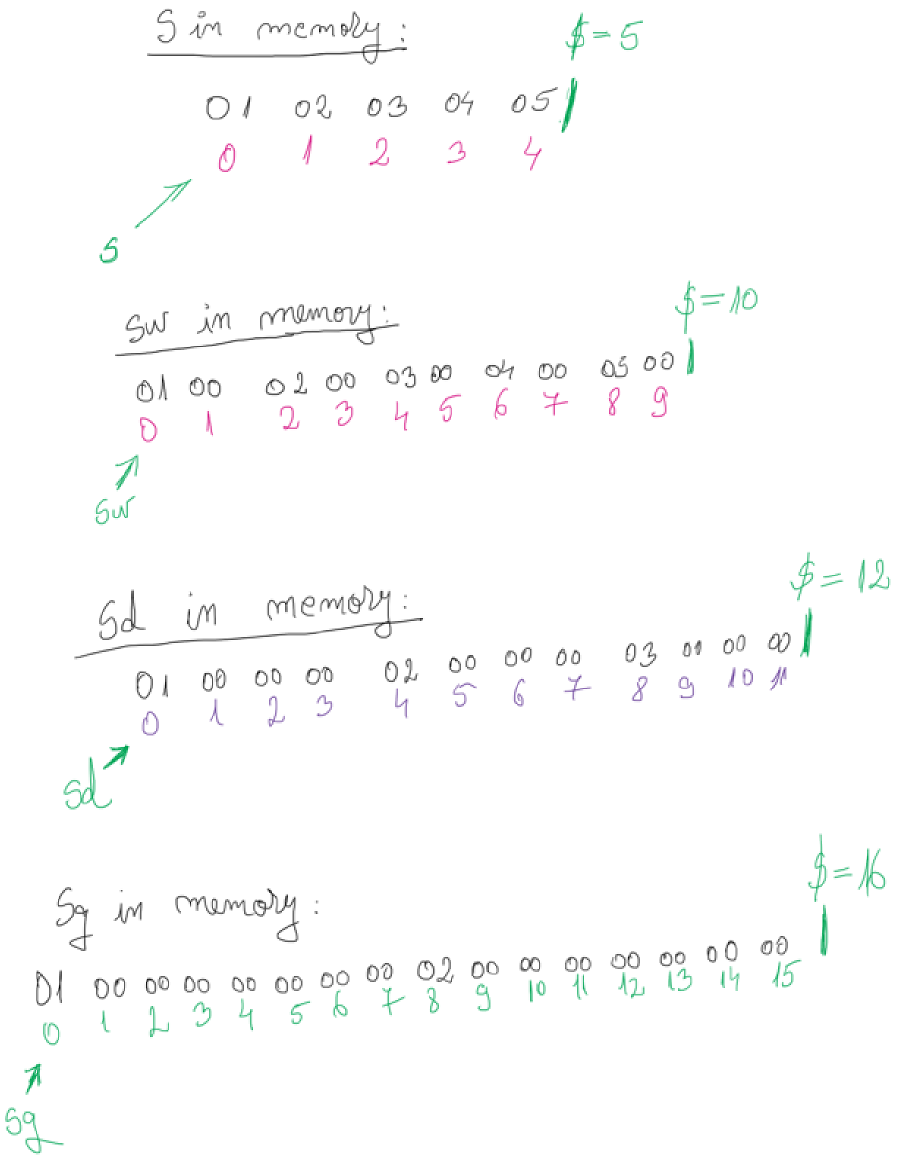
```
sw dw 1, 2, 3, 4, 5
lensw equ($ - sw)/2
;lensw equ (10 - 0) /2 =5
```

doubleword

```
sd dd 1, 2, 3
lensd equ ($-sd)/4
;lensd equ (12-0)/4 =3
```

quadword

```
sq dq 1, 2
lensq equ ($-sq)/8
;lensq equ (16-0)/8 =2
```



s in memory:

$=5

01	02	03	04	05
0	1	2	3	4

s

sw in memory:

$=10

01	00	02	00	03	00	04	00	05	00
0	1	2	3	4	5	6	7	8	9

sw

sd in memory:

$=12

01	00	00	00	02	00	00	00	03	00	00	00
0	1	2	3	4	5	6	7	8	9	10	11

sd

sq in memory:

$=16

01	00	00	00	00	00	00	00	02	00	00	00	00	00	00	00
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

sq

The space allocated in memory

- The space allocated in memory for each string is strict dependent of the type of string s.
  - string of **N bytes** defined in data segment has in memory **N byes allocated**
  - string of **N words** defined in data segment has in memory N words = **N*2 byes allocated**
  - string of **N doublewords** defined in data segment has in memory N doublewords = **N*4 byes allocated**
  - string of **N quadwords** defined in data segment has in memory N quadwords = **N*8 byes allocated**

Space for a string

- For a source string (input string) the space is allocated step by step, base on each element from the string.

- For a destination string (output string), we have to define the name, the type, the length and the initial values:
- There are three ways (we assume the **lens** is a constant with value **10**):
  - Reserve each byte: D DB/dw/dd/dq 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
  - Using RES directive: D RESB/resw/resd/resq lens
  - Using TIMES directive: D times lens DB 0 or **D times lens DB/dw/dd/dq -1**

Word – one variable

1 word = 2 bytes

date segment: a dw 1234h

high byte: 12

low byte: 34

register: mov ax, [a] ; ax =

high byte	low byte
12	34
ah	al

a in memory (little-endian)

34	12
0	1

low byte is at small address: 0

high byte is at a larger address: 1

String of words

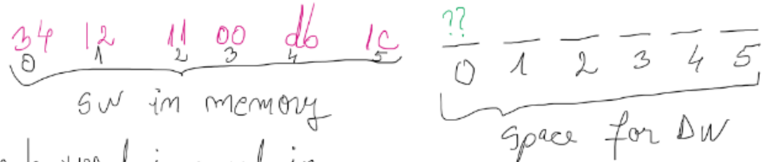
data segment:

Sw dw 1234h, 11h, 1cdbh

lenSw equ ($-Sw)/2

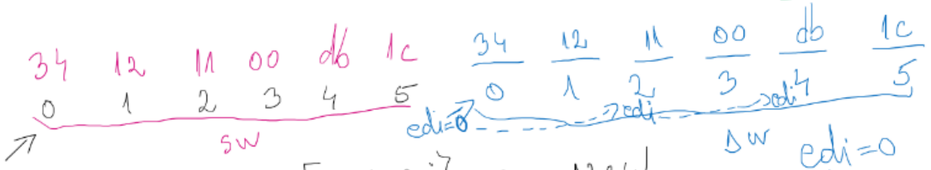
Dw resw lenSw

Sw and Dw in memory:



(each word is saved in memory according little endian: bytes of each word in reversed order)

Access a word from the string: mov bx, [Sw+0] ; bx = 1234h
mov cx, [Sw+2] ; cx = 0011h
mov ax, [Sw+5] ; ax = ?? 1ch



esi=0 => mov ax,[Sw+esi] ; ax=1234h
esi = esi+2 = 2
mov ax, [Sw+esi] ; ax=0011h

esi = esi+2 = 4
mov ax, [Sw+esi], ax=1cdbh

edi=0
mov [dw+edi],ax
edi = edi+2
mov [dw+edi], ax
edi = edi+2
mov [dw+edi], ax

For a word:

esi } modified with 2
edi

(because a word has 2 memory locations)

Doubleword – one variable

1 doubleword = 4 bytes

data segment : $b\ dd\ \underbrace{\boxed{1234}}_{\text{high word}}\overset{\text{low word}}{\boxed{5678}}h$

56 - high byte of low word
78 - low byte of low word

12 - high byte of high word
34 - low byte of high word

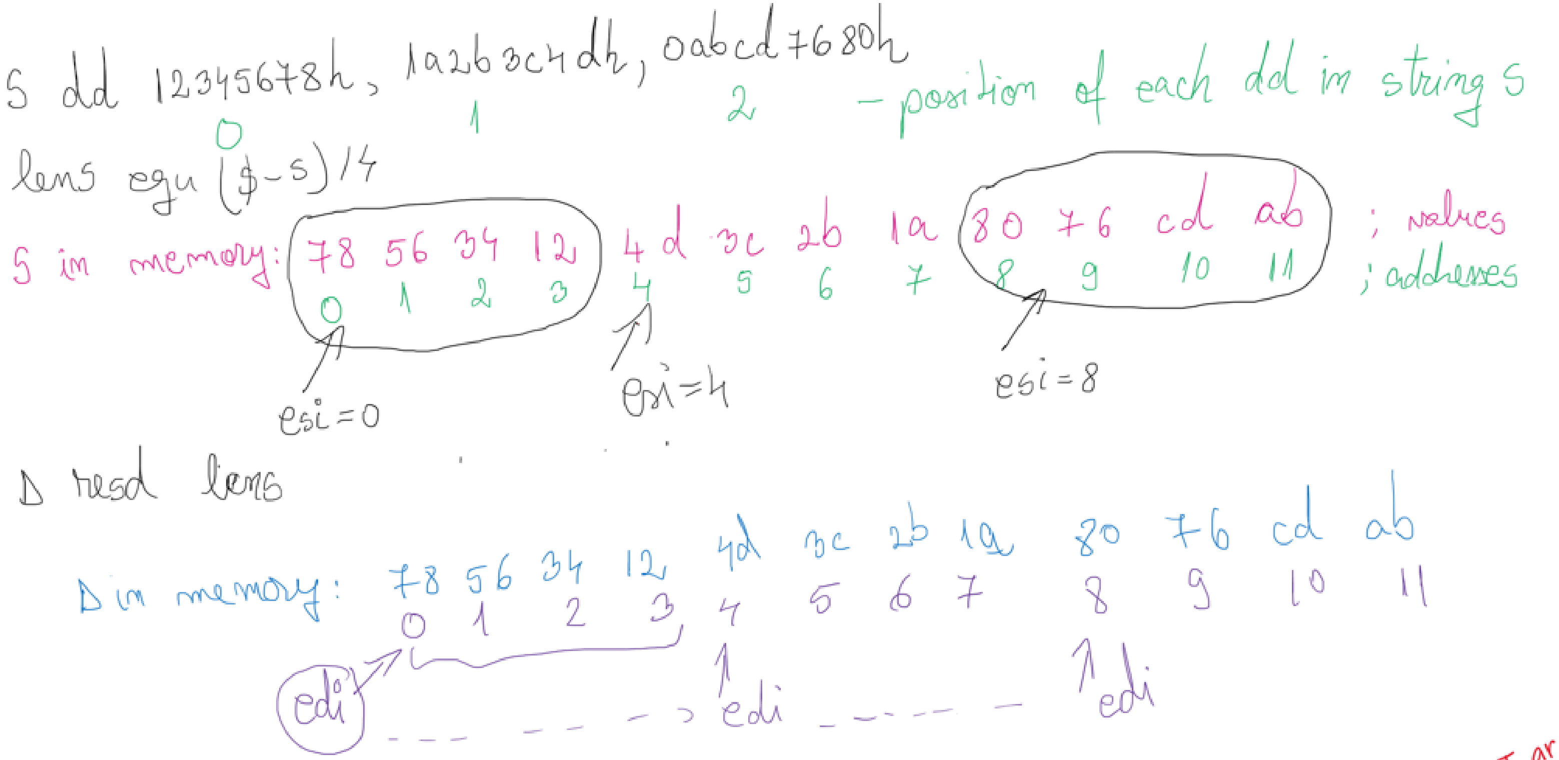
Registers : mov eax, [b] ; eax = 12345678h

mov dx, word [b+2] ;
or
mov ax, word [b+0] ;

$\left.\right\}$ $dx:ax = \underbrace{1234}_{dx}\ \underbrace{5678}_{ax}h$

b in memory: 78 56 34 12 ; value of each byte from dd b.
0 1 2 3 ; address of each byte from dd b.

String of doublewords



```
S dd 12345678h, 1a2b3c4dh, 0abcd7680h
        0           1           2        – position of each dd in string S
lens equ ($-s)/4
```

S in memory:

78	56	34	12	4 d	3c	2b	1a	80	76	cd	ab	; values
0	1	2	3	4	5	6	7	8	9	10	11	; addresses

```
D resd lens
```

D in memory:

78	56	34	12	4d	3c	2b	1a	80	76	cd	ab
0	1	2	3	4	5	6	7	8	9	10	11

Acces a doubleword from a string S

```
mov ESI, 0
mov EAX, [s+ESI] ; EAX = 12345678h

add ESI, 4 ; go to next doubleword
```

Save a doubleword in string D

```
mov EDI, 0
mov [d+EDI], EAX

add EDI, 4
```

For a Doubleword string, the ESI and EDI are modified with 4 (because a doubleword has 4 memory locations)

Quadword – one variable

1 quadword = 8 bytes

data segment: a dq 11 22 33 44 55 66 77 88h

Registers:

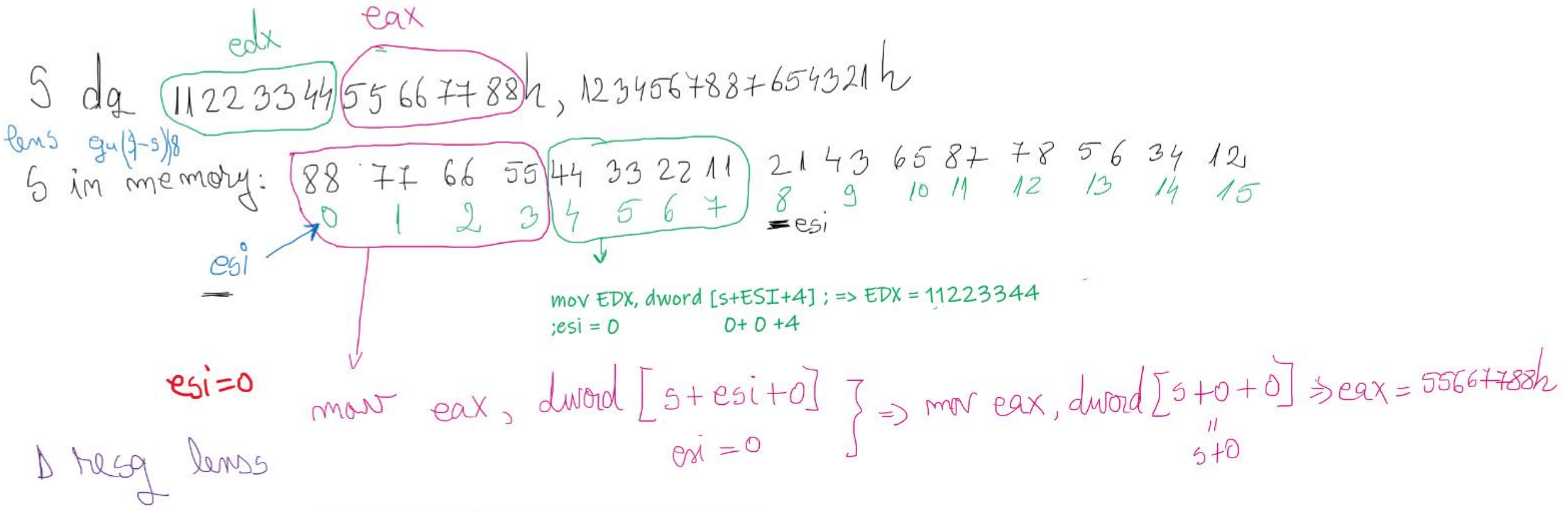
a => edx: eax
or
a => ecx : ebx

```
MOV  eax, dword [a+0] ; eax=55667788h
MOV  edx, dword [a+4] ; edx=11223344h
        or
MOV  ebx, dword [a+0] ; ebx=55667788h
MOV  ecx, dword [a+4] ; ecx=11223344h
```

a-quadword in memory:

```
88 77 66 55 44 33 22 11 - values of bytes from quadword a
0  1  2  3  4  5  6  7  - addresses of bytes from quadword a
```

String of quadword



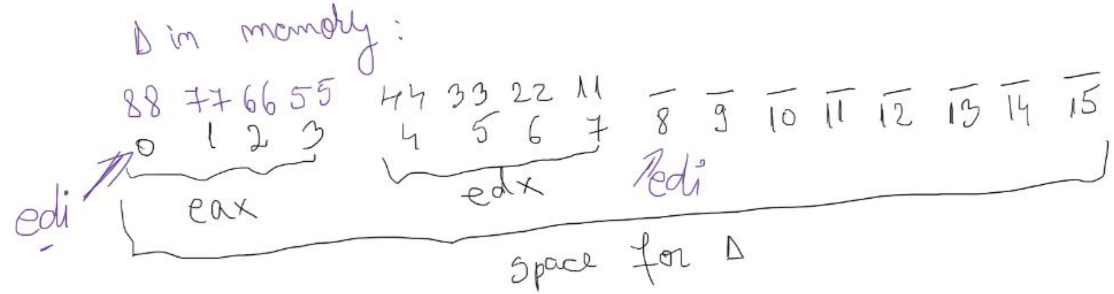
edx
eax
S dq 1122334455667788h, 1234567887654321h
lens equ ($-s)/8
S in memory: 88 77 66 55 44 33 22 11 21 43 65 87 78 56 34 12
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
esi
esi
esi=0
mov eax, dword [s+esi+0] esi = 0 } => mov eax, dword [s+0+0] => eax = 55667788h
s+0
D resq lens

mov EDX, dword [s+ESI+4] ; => EDX = 11223344
;esi = 0 0+ 0 +4

For next quadword from the string: ESI = ESI +8

To save a quadword in a string of quadwords:

```
mov EDI, 0
MOV [D+EDI+0], EAX
MOV [D+EDI+4], EDX
 ADD EDi, 8
```



String Characteristics

Type of the elements

- Bytes (B)
- Words (W)
- Doublewords (D)

Address of the first element

- in DS:ESI - for the source string
- in ES:EDI - for the destination string

The parsing direction

- from small addresses to large addresses -> DF=0 <-> CLD
- from large addresses to small addresses -> DF=1 <-> STD

The number of elements

Instructions for strings processing

LODS (Load from string)	Load memory addressed by ESI into the register
STOS (Store string data)	Store the register contents into memory addressed by EDI
MOVS (Move string data)	Copy data from memory addressed by ESI to memory addressed by EDI
CMPS (Compare strings)	Compare the contents of two memory locations addressed by ESI and EDI
SCAS (Scan string)	Compare the register to the contents of memory addressed by EDI

LODS - Load memory addressed by ESI into the accumulator register

LODS**B** - The byte from the address <DS:ESI> is loaded in AL

- If DF=0 then inc(ESI), else dec(ESI)

LODS**W** - The word from the address <DS:ESI> is loaded in AX

- if DF=0 then ESI:=ESI+2, else ESI:=ESI-2

LODS**D** - The double word from the address <DS:ESI> is loaded in EAX

- If DF=0 then ESI:=ESI+4, else ESI:=ESI-4

STOS - Store the accumulator register contents into memory addressed by EDI

STOS**B** - Store AL into the byte from the address <ES:EDI>

- If DF=0 then inc(EDI), else dec(EDI)

STOS**W** - Store AX into the word from the address <ES:EDI>

- If DF=0 then EDI:= EDI+2, else EDI:= EDI-2

STOS**D** - Store EAX into the double word from the address <ES:EDI>

- If DF=0 then EDI:= EDI+4, else EDI:= EDI-4

MOVS - Copy data from memory addressed by ESI to memory addressed by EDI

MOVS**B** - Copy the byte from the address <DS:ESI> to the address <ES:EDI>

- If DF=0 then inc(ESI), inc(EDI), else dec(ESI), dec(EDI)

MOVS**W** - Copy the word from the address <DS:ESI> to the address <ES:EDI>

- If DF=0 then ESI:= ESI+2, EDI:= EDI+2, else ESI:= ESI-2, EDI:= EDI-2

MOVS**D** - Copy the doubleword from the address <DS:ESI> to the address <ES:EDI>

- If DF=0 then ESI:= ESI+4, EDI:= EDI+4, else ESI:= ESI-4, EDI:= EDI-4

CMPS - Compare the contents of two memory locations addressed by ESI and EDI

CMPS**B** - Compare a byte from <DS:ESI> with a byte from <ES:EDI>

- If DF=0 Then inc(ESI), inc(EDI), Else dec(ESI), dec(EDI)

CMPS**W** - Compare a word from <DS:ESI> with a word from <ES:EDI>

- If DF=0 Then ESI:= ESI+2, EDI:= EDI+2, Else ESI:= ESI-2, EDI:= EDI-2

CMPS**D** - Compare a doubleword from <DS:ESI> with a doubleword from <ES:EDI>

- If DF=0 Then ESI:= ESI+4, EDI:= EDI+4, Else ESI:= ESI-4, EDI:= EDI-4

SCAS - Compare the accumulator register to the contents of memory addressed by EDI

SCAS**B** - Compare AL with a byte from <ES:EDI>

- If DF=0 then inc(EDI), else dec(EDI)

SCAS**W** - Compare AX with a word from <ES:EDI>

- If DF=0 then EDI:= EDI+2, tlse EDI:= EDI-2

SCAS**D** - Compare EAX with a doubleword from <ES:EDI>

- If DF=0 then EDI:= EDI+4, else EDI:= EDI-4

```
sir dq 1122334455667788h, 1a2b3c4d5e6faabbh
lung_sir equ ($-sir)/8  ;dd / 4  ; dw / 2
r times lung_sir dq 0
```

sir in memorie: cf little-endian

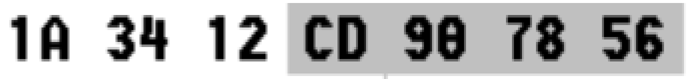
88	**77**	**66**	**55**	**44**	**33**	**22**	**11**
sir+0	sir+1	sir+2	sir+3	sir+4	sir+5	sir+6	sir+7

bb	**aa**	**6f**	**5e**	**4d**	**3c**	**2b**	**1a**
sir+8	sir+9	sir+10	sir+11	sir+12	sir+13	sir+14	sir+15

```
CLD ; left->right (small adresses to large adresses)
Mov esi, sir
Lodsb ; al=88, esi=esi+1
Lodsw; ax=6677, esi=esi+2
Lodsd; eax=22334455, esi=esi+4
```

```
mov edi, r
CLD
Mov al, 1Ah
Stosb ; [r+edi]=1A, edi = edi+1
Mov ax, 1234h
Stosw ; [r+edi]=1234h, edi = edi+2
Mov eax, 567890cdh
Stosd ;[r+edi]=567890cdh, edi=edi+4
```

r in memory:



```
1A 34 12 CD 90 78 56
```

Repeat Prefix for string instructions

REP	Repeat while ECX > 0
REPZ, REPE	Repeat while the ZF=1 and ECX > 0
REPNZ, REPNE	Repeat while the ZF=0 and ECX > 0

Examples

A string of words (unsigned representation) is given in data segment. Copy the content in second string of words.

Without string instructions	With string instruction		
segment data	*segment data*	*segment data*	*segment data*
source_str dw 1234h, 5678h	source_str dw 1234h, 5678h	source_str dw 1234h, 5678h	source_str dw 1234h, 5678h
len_str EQU ($-source_str)/2	len_str EQU ($-source_str)/2	len_str EQU ($-source_str)/2	len_str EQU ($-source_str)/2
dest_str times len_str dw 0	dest_str times len_str dw 0	dest_str times len_str dw 0	dest_str times len_str dw 0
segment code	*segment code*	*segment code*	*segment code*
mov ECX, len_str	mov ECX, len_str	mov ECX, len_str	mov ECX, len_str
mov ESI, 0	mov ESI, source_str	mov ESI, source_str	mov ESI, source_str
mov EDI, 0	mov EDI, dest_str	mov EDI, dest_str	mov EDI, dest_str
myRepeat:			
mov AX, word[source_str+ESI]	**CLD**	**CLD**	**CLD**
mov word[dest_str+EDI], AX	myRepeat:	myRepeat:	
add ESI, 2	LODSW	MOVSW	rep MOVSW
add EDI, 2	STOSW	LOOP myRepeat	
LOOP myRepeat	LOOP myRepeat		

A string of bytes (signed representation) is given in data segment.

Create two strings:

- first string to contain only positive values from the initial string
- second string to contain only negative values from the initial string.

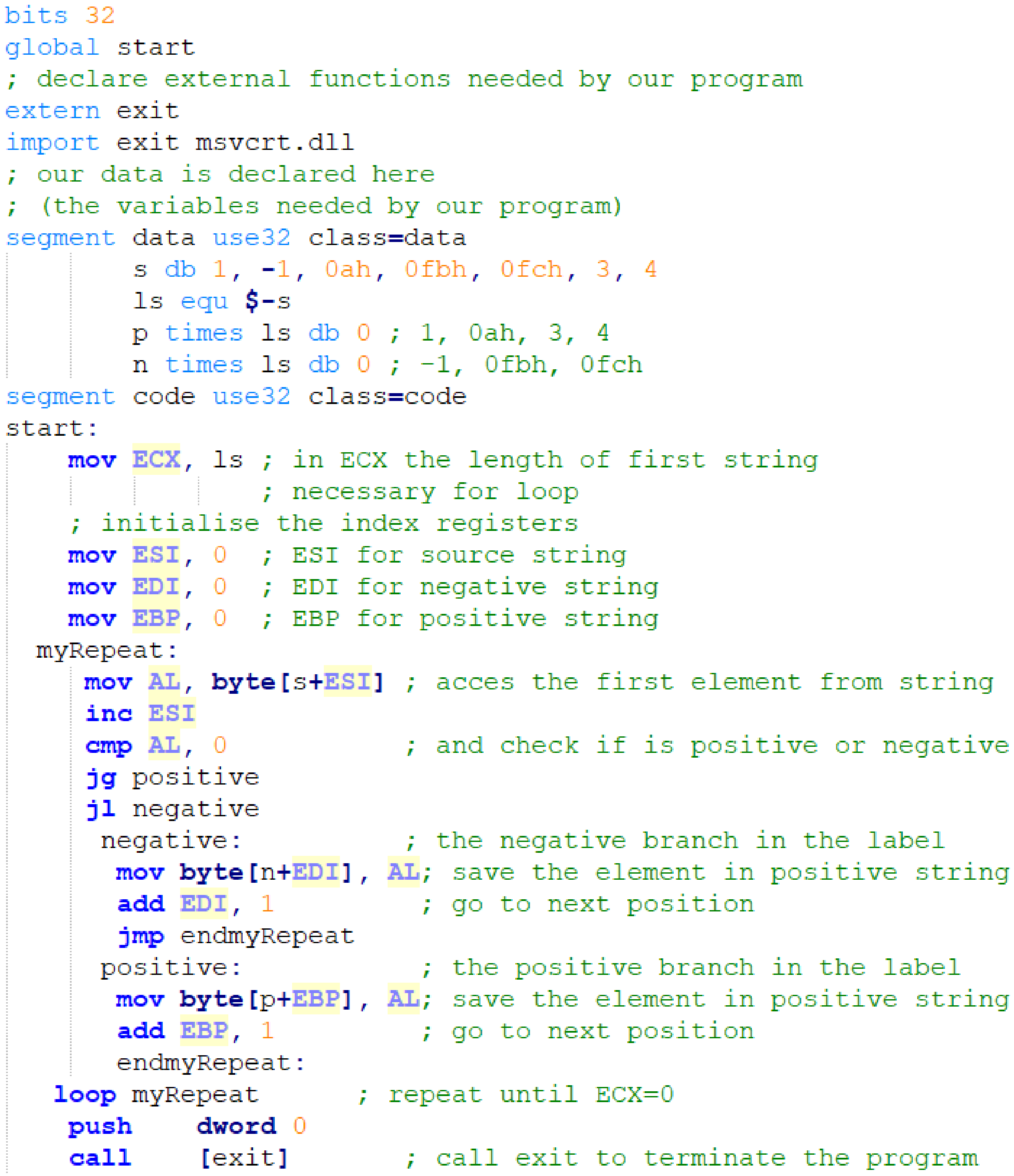
Eg: if initial string = 1, -1, 0ah, 0fbh, 0fch, 3, 4

then:

p = 1, 0ah, 3, 4

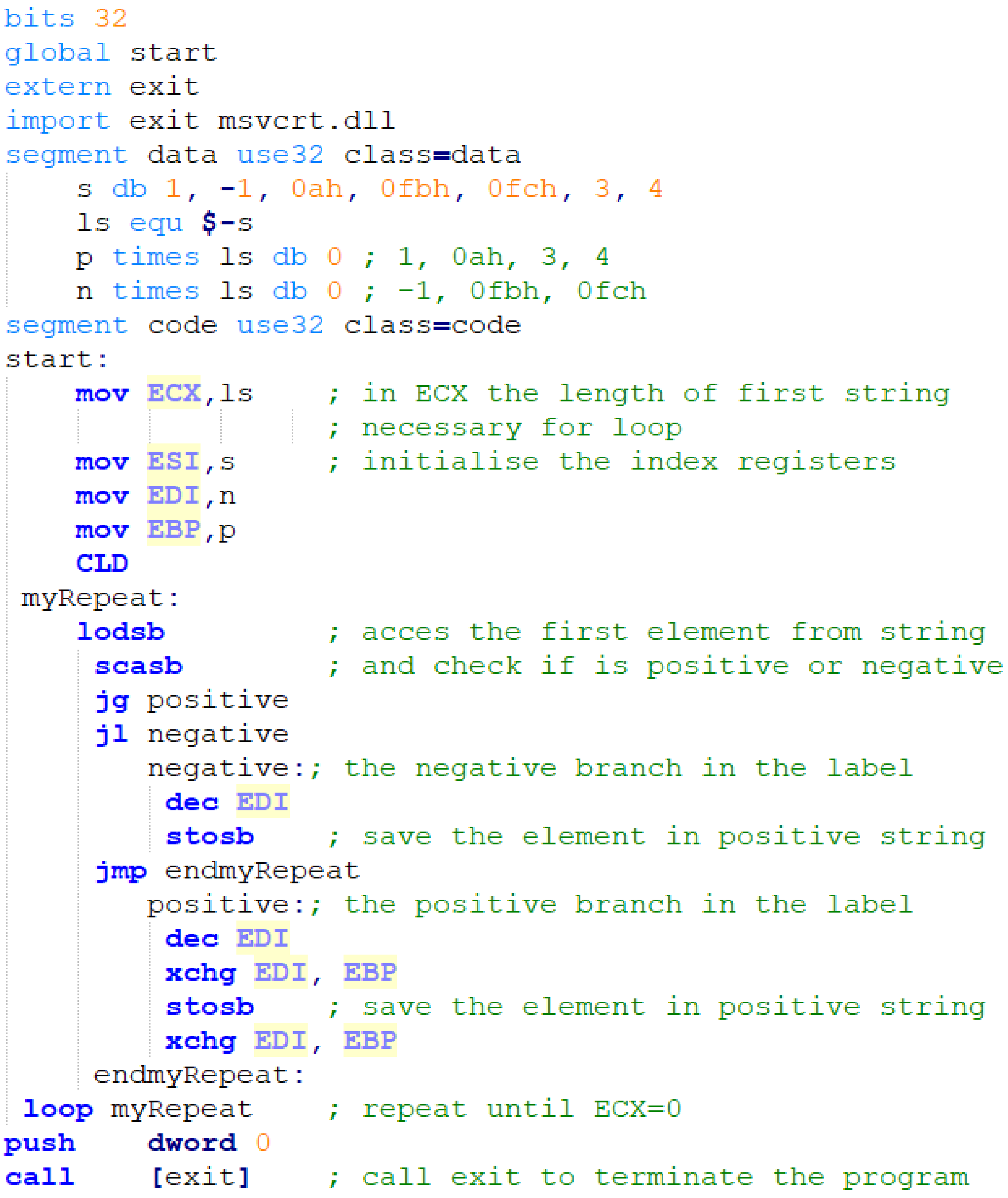
n = -1, 0fbh, 0fch

Without string instructions



```
bits 32
global start
; declare external functions needed by our program
extern exit
import exit msvcrt.dll
; our data is declared here
; (the variables needed by our program)
segment data use32 class=data
        s db 1, -1, 0ah, 0fbh, 0fch, 3, 4
        ls equ $-s
        p times ls db 0 ; 1, 0ah, 3, 4
        n times ls db 0 ; -1, 0fbh, 0fch
segment code use32 class=code
start:
    mov ECX, ls ; in ECX the length of first string
                ; necessary for loop
    ; initialise the index registers
    mov ESI, 0  ; ESI for source string
    mov EDI, 0  ; EDI for negative string
    mov EBP, 0  ; EBP for positive string
  myRepeat:
     mov AL, byte[s+ESI] ; acces the first element from string
     inc ESI
     cmp AL, 0           ; and check if is positive or negative
     jg positive
     jl negative
      negative:          ; the negative branch in the label
       mov byte[n+EDI], AL; save the element in positive string
       add EDI, 1         ; go to next position
       jmp endmyRepeat
      positive:           ; the positive branch in the label
       mov byte[p+EBP], AL; save the element in positive string
       add EBP, 1         ; go to next position
       endmyRepeat:
   loop myRepeat       ; repeat until ECX=0
    push    dword 0
    call    [exit]       ; call exit to terminate the program
```

With string instructions



```
bits 32
global start
extern exit
import exit msvcrt.dll
segment data use32 class=data
    s db 1, -1, 0ah, 0fbh, 0fch, 3, 4
    ls equ $-s
    p times ls db 0 ; 1, 0ah, 3, 4
    n times ls db 0 ; -1, 0fbh, 0fch
segment code use32 class=code
start:
    mov ECX,ls       ; in ECX the length of first string
                     ; necessary for loop
    mov ESI,s        ; initialise the index registers
    mov EDI,n
    mov EBP,p
    CLD
 myRepeat:
    lodsb            ; acces the first element from string
     scasb           ; and check if is positive or negative
     jg positive
     jl negative
        negative:; the negative branch in the label
         dec EDI
         stosb       ; save the element in positive string
     jmp endmyRepeat
        positive:; the positive branch in the label
         dec EDI
         xchg EDI, EBP
         stosb       ; save the element in positive string
         xchg EDI, EBP
     endmyRepeat:
 loop myRepeat       ; repeat until ECX=0
push    dword 0
call    [exit]       ; call exit to terminate the program
```