`gcc` `gdb`

`powershell`

`gcc`

`gdb`

`build-essential`

`gcc-multilib`

`gdb`

`powershell`

C:/reti_logiche assembler.code-workspace C:/reti_logiche/assembler

WSL: Ubuntu-22.04 test-ambiente.s

```
.include "./files/utility.s"

.data
messaggio: .ascii "Ok.\r"

.text
_main:
   nop
   lea messaggio, %ebx
   call outline
   ret
```

\+ pwsh

```
./assemble.ps1 ./test-ambiente.s
```

test-ambiente ./test-ambiente Ok.

`mov` `mov src, %eax` `mov %eax, dest` `%eax`

`utility.s` `\r`

```
.include "./files/utility.s"
```

`_main` `ret` `nop` `.data` `.text` `.data` `.text`

```
.include "./files/utility.s"

.data
messaggio: .ascii "Ok.\r"

.text
_main:
    nop
    lea messaggio, %ebx
    call outline
    ret
```

> `.include "./files/utility.s"` `utility.s` `\r` `_main` `ret`

```
1. Leggere messaggio da terminale.
2. Convertire le lettere minuscole in maiuscolo.
3. Stampare messaggio modificato.
```

`inline` `outline`

```
.data

msg_in: .fill 80, 1, 0
msg_out: .fill 80, 1, 0
```

```
mov $80, %cx
lea msg_in, %ebx
call inline
```

```
lea msg_out, %ebx
call outline
```

`msg_in msg_out`

`\r`

```
char[] msg_in, msg_out;
...
int i = 0;
char c;
do {
    c = msg_in[i];
    msg_out[i] = c;
    i++;
} while (c != '\r')
```

```
    lea msg_in, %esi
    lea msg_out, %edi
    mov $0, %ecx
loop:
    movb (%esi, %ecx), %al
    movb %al, (%edi, %ecx)
    inc %ecx
    cmp $0x0d, %al
    jne loop
```

`movb (%esi, %ecx), %al` `offset(%base, %indice, scala)` `offset + %base + (%indice * scala)` `(%esi, %ecx)` `0(%esi, %ecx, 1)` `b` `%al` `char` `char c` `c >= 'a' && c <= 'z'` `'a' 'A'` $32 = 2^5$ `and` `or` `xor`

```
char[] msg_in, msg_out;
...
int i = 0;
char c;
```

```
do {
    c = msg_in[i];
    if(c >= 'a' && c <= 'z')
        c = c & 0xdf;
    msg_out[i] = c;
    i++;
} while (c != '\r')
```

`0xdf 1101 1111 and`

> `0x20`

```
    lea msg_in, %esi
    lea msg_out, %edi
    mov $0, %ecx
loop:
    movb (%esi, %ecx), %al
    cmp $'a', %al
    jb post_check
    cmp $'z', %al
    ja post_check

    and $0xdf, %al      # 1101 1111 -> l'and resetta il bit 5

post_check:
    movb %al, (%edi, %ecx)
    inc %ecx
    cmp $0x0d, %al
    jne loop
```

```
.include "./files/utility.s"

.data
msg_in: .fill 80, 1, 0
msg_out: .fill 80, 1, 0

.text
_main:
    nop
punto_1:
    mov $80, %cx
    lea msg_in, %ebx
    call inline
    nop
punto_2:
    lea msg_in, %esi
    lea msg_out, %edi
    mov $0, %ecx
loop:
    movb (%esi, %ecx), %al
    cmp $'a', %al
    jb post_check
```

```
    cmp $'z', %al
    ja post_check
    and $0xdf, %al
post_check:
    movb %al, (%edi, %ecx)
    inc %ecx
    cmp $0x0d, %al
    jne loop
punto_3:
    lea msg_out, %ebx
    call outline
    nop
fine:
    ret
```

`punto_1 punto_2 punto_3 fine nop call gdb`

`gdb debug.ps1`

```
./debug.ps1 nome-eseguibile
```

`_main rr qq`

```
Breakpoint 1, _main () at /mnt/c/reti_logiche/assembler/lezioni/1/maiusc.s:9
9           nop
(gdb)
```

`_main nop`

`info registers x`

`break`

`step next continue`

```
Breakpoint 1, _main () at /mnt/c/reti_logiche/assembler/lezioni/1/maiusc.s:9
9           nop
(gdb) step
punto_1 () at /mnt/c/reti_logiche/assembler/lezioni/1/maiusc.s:11
11          mov $80, %cx
(gdb) s
12          lea msg_in, %ebx
```

```
(gdb) s
13          call inline
(gdb)
```

`gdb step s call info registers i r`

```
(gdb) i r
eax            0x66                102
ecx            0x50                80
edx            0x2d                45
ebx            0x56559066          1448448102
esp            0xffffc06c          0xffffc06c
ebp            0xffffc078          0xffffc078
esi            0xf7fb2000          -134537216
edi            0xf7fb2000          -134537216
eip            0x5655676e          0x5655676e <punto_1+10>
eflags         0x282               [ SF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
(gdb)
```

```
(gdb) i r cx ebx
cx             0x50                80
ebx            0x56559066          1448448102
(gdb)
```

`call inline utility.s step next step next nop parte_2 next punto_1 next nop punto_1`

```
13          call inline
(gdb) n
questo e' un test
14          nop
(gdb)
```

`inline break continue`

```
(gdb) b loop
Breakpoint 2 at 0x56556785: file /mnt/c/reti_logiche/assembler/lezioni/1/maiusc.s, line 20.
(gdb) c
Continuing.

Breakpoint 2, loop () at /mnt/c/reti_logiche/assembler/lezioni/1/maiusc.s:20
20          movb (%esi, %ecx), %al
(gdb) i r ecx
ecx            0x0                 0
(gdb) c
Continuing.

Breakpoint 2, loop () at /mnt/c/reti_logiche/assembler/lezioni/1/maiusc.s:20
```

```
20          movb (%esi, %ecx), %al
(gdb) i r ecx
ecx            0x1                 1
(gdb) c
Continuing.

Breakpoint 2, loop () at /mnt/c/reti_logiche/assembler/lezioni/1/maiusc.s:20
20          movb (%esi, %ecx), %al
(gdb) c
Continuing.

Breakpoint 2, loop () at /mnt/c/reti_logiche/assembler/lezioni/1/maiusc.s:20
20          movb (%esi, %ecx), %al
(gdb) i r ecx
ecx            0x3                 3
(gdb)
```

`x`

- `N`
- `F d c`
- `U b h w`
- `addr`

`x/NFU addr N F U / addr`

- `x 0x56559066`
- `& x &msg_in`
- `$ x $esi`
- `x (int*)&msg_in+$ecx`

```
(gdb) x/20cb &msg_in
0x56559066:     113 'q' 117 'u' 101 'e' 115 's' 116 't' 111 'o' 32 ' '  101 'e'
0x5655906e:     39 '\'' 32 ' '  117 'u' 110 'n' 32 ' '  116 't' 101 'e' 115 's'
0x56559076:     116 't' 13 '\r' 10 '\n' 0 '\000'
(gdb) x/20cb &msg_out
0x565590b6:     81 'Q'  85 'U'  69 'E'  0 '\000'        0 '\000'        0 '\000'        0 '\000'      0 '\000'
0x565590be:     0 '\000'        0 '\000'        0 '\000'        0 '\000'        0 '\000'     0 '\000' 0 '\000'
0x565590c6:     0 '\000'        0 '\000'        0 '\000'        0 '\000'
(gdb) x/20cb $esi
0x56559066:     113 'q' 117 'u' 101 'e' 115 's' 116 't' 111 'o' 32 ' '  101 'e'
0x5655906e:     39 '\'' 32 ' '  117 'u' 110 'n' 32 ' '  116 't' 101 'e' 115 's'
0x56559076:     116 't' 13 '\r' 10 '\n' 0 '\000'
```

`esi ecx (%esi, %ecx) movb (%esi, %ecx), %al lea mov lea`

```
    lea (%esi, %ecx), %ebx
    movb (%ebx), %al
```

`ebx x/1cb $ebx layout regs r q qq rr`

1. Leggere messaggio da terminale.
2. Convertire le lettere minuscole in maiuscolo, usando le istruzioni stringa.
3. Stampare messaggio modificato.

```
    lea msg_in, %eax
    lea msg_out, %ebx
    mov $0, %edx
loop:
    movb (%eax, %edx), %cl
    ...
```

`esi edi ecx esi edi eax ecx lods stos punto_2`

```
...
punto_2:
    lea msg_in, %esi
    lea msg_out, %edi
loop:
    movb (%esi), %al
    inc %esi
    cmp $'a', %al
    jb post_check
    cmp $'z', %al
    ja post_check
    and $0xdf, %al
post_check:
    movb %al, (%edi)
    inc %edi
    cmp $0x0d, %al
    jne loop
...
```

`ecx esi edi inc add $2, %esi add $4, %esi`

```
...
punto_2:
    lea msg_in, %esi
    lea msg_out, %edi
    cld
loop:
    lodsb
```

```
    cmp $'a', %al
    jb post_check
    cmp $'z', %al
    ja post_check
    and $0xdf, %al
post_check:
    stosb
    cmp $0x0d, %al
    jne loop
...
```

`cld b`

`.data numero array`

```
.include "./files/utility.s"

.data
array:      .word 1, 256, 256, 512, 42, 2048, 1024, 1, 0
array_len:  .long 9
numero:     .word 1
```

```
.include "./files/utility.s"

.data
array:      .word 1, 256, 256, 512, 42, 2048, 1024, 1, 0
array_len:  .long 9
numero:     .word 1

.text

_main:
    nop
    mov $0, %cl
    mov numero, %ax
    mov $0, %esi

comp:
    cmp array_len, %esi
```

```
    je fine
    cmpw array(%esi), %ax
    jne poi
    inc %cl

poi:
    inc %esi
    jmp comp

fine:
    mov %cl, %al
    call outdecimal_byte
    ret
```

```
# leggere 2 numeri interi in base 10, calcolarne il prodotto, e stampare il risultato.

# lettura:
# come primo carattere leggere il segno del numero, cioè un '+' o un '-'
# segue il modulo del numero, minore di 256

# stampa:
# stampare prima il segno del numero (+ o -), poi il modulo in cifre decimali
```

```
.include "./files/utility.s"

mess1:  .asciz "inserire il primo numero intero:\r"
mess2:  .asciz "inserire il secondo numero intero:\r"
mess3:  .asciz "il prodotto dei due numeri e':\r"
a:      .word 0
b:      .word 0

_main:
    nop
    lea mess1, %ebx
    call outline
    call in_intero
    mov %ax, a

    lea mess2, %ebx
    call outline
    call in_intero
    mov %ax, b

    mov a, %ax
    mov b, %bx
    imul %bx
```

```
    lea mess3, %ebx
    call outline
    call out_intero
    ret

# legge un intero composto da segno e modulo minore di 256
# ne lascia la rappresentazione in complemento alla radice base 2 in ax
in_intero:
    push %ebx
    mov $0, %bl
in_segno_loop:
    call inchar
    cmp $'+', %al
    je in_segno_poi
    cmp $'-', %al
    jne in_segno_loop
    mov $1, %bl
in_segno_poi:
    call outchar
    call indecimal_word
    call newline
    cmp $1, %bl
    jne in_intero_fine
    neg %ax
in_intero_fine:
    pop %ebx
    ret

# legge la rappresentazione di un numero intero in complemento alla radice base 2 in eax
# lo stampa come segno seguito dalle cifre decimali
out_intero:
    push %ebx
    mov %eax, %ebx
    cmp $0, %ebx
    ja out_intero_pos
    jmp out_intero_neg
out_intero_pos:
    mov $'+', %al
    call outchar
    jmp out_intero_poi
out_intero_neg:
    mov $'-', %al
    call outchar
    neg %ebx
    jmp out_intero_poi
out_intero_poi:
    mov %ebx, %eax
    call outdecimal_long
    pop %ebx
    ret
```

`.data numero array`

```
.include "./files/utility.s"

.data
array:      .word 1, 256, 256, 512, 42, 2048, 1024, 1, 0
array_len:  .long 9
numero:     .word 1
```

```
.include "./files/utility.s"

.data
array:      .word 1, 256, 256, 512, 42, 2048, 1024, 1, 0
array_len:  .long 9
numero:     .word 1

.text

_main:
    nop
    mov $0, %cl
    mov numero, %ax
    mov $0, %esi

comp:
    cmp array_len, %esi
    je fine
    cmpw array(%esi), %ax
    jne poi
    inc %cl
```

```
poi:
    inc %esi
    jmp comp

fine:
    mov %cl, %al
    call outdecimal_byte
    ret
```

`%cl %cl numero array %ax numero %esi array_len array %esi %ax numero array %esi numero %ax cmp %cl`

```
int cl = 0;
for(int esi = 0; esi < array_len; esi++){
    if(array[esi] == numero)
        cl++;
}
```

`cmpw array %ax numero cmpw`

```
.include "./files/utility.s"

.data
array:      .word 1, 256, 256, 512, 42, 2048, 1024, 1, 0
array_len:  .long 9
numero:     .word 1

.text

_main:
    nop
    mov $0, %cl
    mov numero, %ax
    mov $0, %esi

comp:
    cmp array_len, %esi
    je fine
    movw array(%esi), %bx
    cmpw %bx, %ax
    jne poi
    inc %cl

poi:
    inc %esi
    jmp comp

fine:
    mov %cl, %al
    call outdecimal_byte
    ret
```

`break 20 continue i r ax bx cl esi`

```
(gdb) i r ax bx cl esi
ax             0x1                 1
bx             0x1                 1
cl             0x0                 0
esi            0x0                 0
```

`%ax numero %bx array %cl %esi step jne step`

```
(gdb) i r ax bx cl esi
ax             0x1                 1
bx             0x0                 0
cl             0x1                 1
esi            0x1                 1
```

`%bx array array` $naaa + 1a + (n-1)$ `array` $aaa + 1(a+1, a)$ `movw a, %bx` $a + 1$ `%bh` a `%bl` $array + 2$ $array + 4$

```
comp:
    cmp array_len, %esi
    je fine
    movw array(%esi), %bx
    cmpw %bx, %ax
    jne poi
    inc %cl

poi:
    inc %esi
    jmp comp
```

`movb array(%esi), %bx %esi array %esi array(%esi) %esi array_len array(, %esi, 2)` $array + 2 * esi$ `%esi array[esi]`

```
.include "./files/utility.s"

.data
array:      .word 1, 256, 256, 512, 42, 2048, 1024, 1, 0
array_len:  .long 9
numero:     .word 1

.text

_main:
    nop
    mov $0, %cl
    mov numero, %ax
    mov $0, %esi

comp:
    cmp array_len, %esi
    je fine
    cmpw array(, %esi, 2), %ax
    jne poi
    inc %cl

poi:
    inc %esi
```

```
    jmp comp

fine:
    mov %cl, %al
    call outdecimal_byte
    ret
```

```
# leggere 2 numeri interi in base 10, calcolarne il prodotto, e stampare il risultato.

# lettura:
# come primo carattere leggere il segno del numero, cioè un '+' o un '-'
# segue il modulo del numero, minore di 256

# stampa:
# stampare prima il segno del numero (+ o -), poi il modulo in cifre decimali
```

```
.include "./files/utility.s"

mess1:  .asciz "inserire il primo numero intero:\r"
mess2:  .asciz "inserire il secondo numero intero:\r"
mess3:  .asciz "il prodotto dei due numeri e':\r"
a:      .word 0
b:      .word 0

_main:
    nop
    lea mess1, %ebx
    call outline
    call in_intero
    mov %ax, a

    lea mess2, %ebx
    call outline
    call in_intero
    mov %ax, b

    mov a, %ax
    mov b, %bx
    imul %bx

    lea mess3, %ebx
    call outline
    call out_intero
    ret

# legge un intero composto da segno e modulo minore di 256
```

```
# ne lascia la rappresentazione in complemento alla radice base 2 in ax
in_intero:
    push %ebx
    mov $0, %bl
in_segno_loop:
    call inchar
    cmp $'+', %al
    je in_segno_poi
    cmp $'-', %al
    jne in_segno_loop
    mov $1, %bl
in_segno_poi:
    call outchar
    call indecimal_word
    call newline
    cmp $1, %bl
    jne in_intero_fine
    neg %ax
in_intero_fine:
    pop %ebx
    ret

# legge la rappresentazione di un numero intero in complemento alla radice base 2 in eax
# lo stampa come segno seguito dalle cifre decimali
out_intero:
    push %ebx
    mov %eax, %ebx
    cmp $0, %ebx
    ja out_intero_pos
    jmp out_intero_neg
out_intero_pos:
    mov $'+', %al
    call outchar
    jmp out_intero_poi
out_intero_neg:
    mov $'-', %al
    call outchar
    neg %ebx
    jmp out_intero_poi
out_intero_poi:
    mov %ebx, %eax
    call outdecimal_long
    pop %ebx
    ret
```

```
inserire il primo numero intero:
+30
Segmentation fault
```

```
Program received signal SIGSEGV, Segmentation fault.
                                                    _main () at /mnt/c/reti_logiche/assembler/lezioni/2/imul_debug.s:14
14          mov %ax, a
```

```
a .data  .text  .text
```

```
.include "./files/utility.s"

.data
mess1:  .asciz "inserire il primo numero intero:\r"
mess2:  .asciz "inserire il secondo numero intero:\r"
mess3:  .asciz "il prodotto dei due numeri e':\r"
a:      .word 0
b:      .word 0

.text
_main:
...
```

```
inserire il primo numero intero:
+30
inserire il secondo numero intero:
+20
il prodotto dei due numeri e':
+600
```

```
inserire il primo numero intero:
+255
inserire il secondo numero intero:
+255
il prodotto dei due numeri e':
+65025
```

```
inserire il primo numero intero:
-255
inserire il secondo numero intero:
+255
il prodotto dei due numeri e':
+511
```

`.data .text -255`

```
(gdb) b 16
Breakpoint 2 at 0x56556774: file /mnt/c/reti_logiche/assembler/lezioni/2/imul_debug.s, line 16.
(gdb) c
Continuing.
inserire il primo numero intero:
-255

Breakpoint 2, _main () at /mnt/c/reti_logiche/assembler/lezioni/2/imul_debug.s:16
16          mov %ax, a
(gdb) i r ax
ax             0xff01              -255
(gdb)
```

`imul %ax %bx %dx_%ax`

```
Breakpoint 3, _main () at /mnt/c/reti_logiche/assembler/lezioni/2/imul_debug.s:25
25          imul %bx
(gdb) i r ax bx
ax             0xff01              -255
bx             0xff                255
(gdb) s
27          lea mess3, %ebx
(gdb) i r dx ax
dx             0xffff              -1
ax             0x1ff               511
(gdb)
```

`0xffff01ff %ax gdb 0xffffffffffff01ff f out_intero`

```
# legge la rappresentazione di un numero intero in complemento alla radice base 2 in eax
# lo stampa come segno seguito dalle cifre decimali
```

`%eax imul %dx_%ax call out_intero`

```
...
    mov a, %ax
    mov b, %bx
    imul %bx

    shl $16, %edx
    movw %ax, %dx
    movl %edx, %eax

    lea mess3, %ebx
    call outline
    call out_intero
...
```

```
inserire il primo numero intero:
-255
inserire il secondo numero intero:
+255
il prodotto dei due numeri e':
+4294902271
```

`call out_intero %eax`

```
Breakpoint 2, _main () at /mnt/c/reti_logiche/assembler/lezioni/2/imul_debug.s:33
33          call out_intero
(gdb) i r eax
eax            0xffff01ff          -65025
(gdb)
```

`out_intero_pos out_intero_neg neg out_intero_poi outdecimal_long step out_intero`

```
(gdb) s
out_intero () at /mnt/c/reti_logiche/assembler/lezioni/2/imul_debug.s:62
62          push %ebx
(gdb) s
```

```
63          mov %eax, %ebx
(gdb) s
64          cmp $0, %ebx
(gdb) i r ebx
ebx            0xffff01ff          -65025
(gdb) s
65          ja out_intero_pos
(gdb) s
out_intero_pos () at /mnt/c/reti_logiche/assembler/lezioni/2/imul_debug.s:68
68          mov $'+', %al
(gdb)
```

`%ebx` `out_intero_pos` `ja` `%ebx` `jg`

```
    cmp $0, %ebx
    jg out_intero_pos
    jmp out_intero_neg
```

```
inserire il primo numero intero:
-255
inserire il secondo numero intero:
+255
il prodotto dei due numeri e':
-65025
```

```
.include "./files/utility.s"

.data
mess1:  .asciz "inserire il primo numero intero:\r"
mess2:  .asciz "inserire il secondo numero intero:\r"
mess3:  .asciz "il prodotto dei due numeri e':\r"
a:      .word 0
b:      .word 0

.text
_main:
    nop
    lea mess1, %ebx
    call outline
    call in_intero
    mov %ax, a

    lea mess2, %ebx
    call outline
    call in_intero
    mov %ax, b

    mov a, %ax
    mov b, %bx
    imul %bx

    shl $16, %edx
    movw %ax, %dx
```

```
    movl %edx, %eax

    lea mess3, %ebx
    call outline
    call out_intero
    ret

# legge un intero composto da segno e modulo minore di 256
# ne lascia la rappresentazione in complemento alla radice base 2 in ax
in_intero:
    push %ebx
    mov $0, %bl
in_segno_loop:
    call inchar
    cmp $'+', %al
    je in_segno_poi
    cmp $'-', %al
    jne in_segno_loop
    mov $1, %bl
in_segno_poi:
    call outchar
    call indecimal_word
    call newline
    cmp $1, %bl
    jne in_intero_fine
    neg %ax
in_intero_fine:
    pop %ebx
    ret

# legge la rappresentazione di un numero intero in complemento alla radice base 2 in eax
# lo stampa come segno seguito dalle cifre decimali
out_intero:
    push %ebx
    mov %eax, %ebx
    cmp $0, %ebx
    jg out_intero_pos
    jmp out_intero_neg
out_intero_pos:
    mov $'+', %al
    call outchar
    jmp out_intero_poi
out_intero_neg:
    mov $'-', %al
    call outchar
    neg %ebx
    jmp out_intero_poi
out_intero_poi:
    mov %ebx, %eax
    call outdecimal_long
    pop %ebx
    ret
```

`matrice[i][j]` a $[0,3]$ b $[4,7]$ `1 2` $ij \in [0,3] i*4+j$ `in_lettera in_numero %al`

```
# Sottoprogramma per la lettura della lettera, da 'a' a 'd'
# Lascia l'indice corrispondente (da 0 a 3) in AL
in_lettera:
    call inchar
    cmp $'a', %al
    jb in_lettera
    cmp $'d', %al
    ja in_lettera
    call outchar
    sub $'a', %al
    ret
```

```
    call in_lettera
    mov %al, %cl
    shl $2, %cl # cl = cl * 4, ossia la dimensione di ogni riga
    call in_numero
    add %al, %cl # cl contiene l'indice (da 0 a 15) della posizione bersagliata
```

`%ax[%cl] and`

```
    mov $1, %ax
    shl %cl, %ax # ax contiene una maschera da 16 bit con 1 nella posizione bersagliata
    and %dx, %ax # se abbiamo colpito qualcosa, ax rimane invariato. altrimenti varra' 0
    jz mancato
```

`and xor`

```
colpito:
    lea msg_colpito, %ebx
    call outline
    xor %ax, %dx # togliamo il bersaglio colpito
    jmp ciclo_partita_fine
```

`inword`

```
# Leggere una riga dal terminale, che DEVE contenere almeno 2 caratteri '_'
# Identificare e stampa la sottostringa delimitata dai primi due caratteri '_'
```

```
questa e' una _prova_ !!
prova
```

```
.include "./files/utility.s"

.data

msg_in: .fill 80, 1, 0

.text
_main:
    nop
    mov $80, %cx
    lea msg_in, %ebx
    call inline

    cld
    mov $'_', %al
    lea msg_in, %esi
    mov $80, %cx

    repne scasb
    mov %esi, %ebx
    repne scasb
    mov %esi, %ecx
    sub %ebx, %ecx
    call outline

    ret
```

```
# Leggere una riga dal terminale
# Identificare e stampa la sottostringa delimitata dai primi due caratteri '_'
# Se un solo carattere '_' e' presente, assumere che la sottostringa cominci
# ad inizio stringa e finisca prima del carattere '_'
# Se nessun carattere '_' e' presente, stampare l'intera stringa
```

`repne scasb %al _`

`rep %ecx %cx %ecx`

`scasb %edi %esi`

`repne scasb _ _ _`

`outline \r outmess`

```
.include "./files/utility.s"

.data

msg_in: .fill 80, 1, 0

.text
_main:
    nop
    mov $80, %cx
    lea msg_in, %ebx
    call inline

    cld
    mov $'_', %al
    lea msg_in, %edi
    mov $80, %cx

    repne scasb
    mov %edi, %ebx
    repne scasb
    mov %edi, %ecx
    sub %ebx, %ecx
```

```
    dec %ecx
    call outmess

    ret
```

`%ecx`

`_ repne scasb %ecx %ecx repne scasb _ print_all _ print_from_start print_substr print_all outline print_from_start msg_in %edi repne scasb`

```
.include "./files/utility.s"

.data

msg_in: .fill 80, 1, 0

.text
_main:
    nop
    mov $80, %cx
    lea msg_in, %ebx
    call inline

    cld
    mov $'_', %al
    lea msg_in, %edi
    mov $80, %ecx

    repne scasb
    cmp $0, %ecx
    je print_all

    mov %edi, %ebx
    repne scasb
    cmp $0, %ecx
    je print_from_start

print_substr:
    mov %edi, %ecx
    sub %ebx, %ecx
    dec %ecx
    call outmess
    ret

print_from_start:
    mov %ebx, %ecx
    lea msg_in, %ebx
    sub %ebx, %ecx
    dec %ecx
    call outmess
    ret

print_all:
```

```
    lea msg_in, %ebx
    call outline
    ret
```

`inchar outchar` $N = 9k = 91 + 2 + ... + n\frac{n(n+1)}{2}9 \cdot 10/2 = 459 \cdot 9 = 8181/2418145j(j-1) \cdot 9 + 1$
$40 \cdot 9 + 1 = 36180 \cdot 9 + 1 = 72144 \cdot 9 + 1 = 397$

```
short c = 1; // word da 16 bit
for(int i = 0; i < n; i++) {
    for(int j = 0; j < i + 1; j++) {
        outdecimal_word(c);
        outchar(' ');
        c += k;
    }
    outline()
}
```

`inline` c `$'a' $'f'` $c -' a' + 10O(n_{cifre})O(n_{stringa}) + O(n_{cifre})O(1)O(n_{cifre} \cdot n_{stringa})n$

`[bwl]` `r` `mov %eax, %ebx` `m` `mov numero, %eax` `mov (%esi), %eax` `mov matrice(%esi, %ecx, 4)` `i` `mov $0, %eax` `movl x, y` `mov (%eax), (%ebx)` `movs`

			ZF
			%al %ax
			%ax %eax

			CF OF
			CF OF
			CF CF OF
			CF CF OF
			CF
			CF
			OF

`%dx_%ax` `%dx %ax`

			`%al %ax CF OF`
			`%ax %dx_%ax CF OF`
			`%eax %edx_%eax CF OF`
			`%al %ax CF OF`
			`%ax %dx_%ax CF OF`
			`%eax %edx_%eax CF OF`

			`%ax %al %ah`
			`%dx_%ax %ax %dx`
			`%edx_%eax %eax %edx`
			`%ax %al %ah`
			`%dx_%ax %ax %dx`
			`%edx_%eax %eax %edx`

`and`

			`CF %cl`
			`OF %cl`
			`CF %cl`
			`CF %cl`
			`CF %cl`
			`CF %cl`

			`CF CF %cl`
			`CF CF %cl`

			`ret`
			`call`

`cmp cmp cmp cmp`

			`ZF`
			`ZF`
			`CF`
			`CF`
			`OF`
			`OF`
			`SF`
			`SF`

`cmp cmp loop dec cmp`

			%ecx
			%ecx %ecx
			%ecx %ecx
			%ecx %ecx ZF
			%ecx %ecx ZF

`set`

			ZF
			ZF
			CF
			CF
			OF
			OF
			SF
			SF

			mov
			mov
			mov
			mov
			mov
			mov
			mov
			mov
			mov
			mov ZF

			`mov ZF`
			`mov CF`
			`mov CF`
			`mov OF`
			`mov OF`
			`mov SF`
			`mov SF`

			`DF`
			`DF`
			`%esi %al %ax %eax DF %esi`
			`%al %ax %eax %edi DF %edi`
			`%esi %edi DF %edi`
			`%esi %edi cmp`
			`%al %ax %eax %edi cmp`

`rep`

			`%ecx opcode %ecx lods stos movs`
			`%ecx %ecx opcode %ecx cmps scas`
			`%ecx %ecx opcode %ecx cmps scas`

			`%eip`

			`%dx %edi`
			`%esi %dx`

`utility.s`

`\n 0x0A \r 0x0D`

`inchar`	`%al`
`outchar`	`%al`
`inbyte inword inlong`	`%al %ax %eax`
`outbyte outword outlong`	`%al %ax %eax`
`indecimal_byte indecimal_word indecimal_long`	`\r %al %ax %eax`
`outdecimal_byte outdecimal_word outdecimal_long`	`%al %ax %eax`
`outmess`	v `%ebx` n `%cx` v
`outline`	v `%ebx` v `\r`
`inline`	v `%ebx` n `%cx` v `\r` $n-2$ `\r\n`
`newline`	`\r\n`

`gdb`

```
help
```

			`call`
			`call ret nop call`
			`ret call`

`gdb`

`next next punto_1 call next continue`

```
punto_1:
    ...
    call newline
punto_2:
    ...
```

`nop call nop punto_1`

```
punto_1:
    ...
    call newline
    nop
punto_2:
    ...
```

			%ax
			i r eax %ax

`gdb layout layout regs i r l`

			/

`x gdb`

`x`

`d`

`c`

`t`

`s 0x00`

`gdb x`

`b`

`h`

`w`

`gdb h`

`x 0x56559066`

`x &label`

`x $esi`

`x (char*)$esi + $ecx`

`$ecx (char*) (short*) (int*) movb (%esi, %ecx), %al lea (%esi, %ecx), %ebx movb (%ebx), %al x $ebx`

`info b`

`x`

`cond 2 $al==5 al`

`cond 2 (short *)$edi==-5 edi`

`cond 2 (int *)&count!=0 count`

```
gdb cond 2 $al==128 0x80 -128 128
```

```
gdb
```


`x watch $eax eax`

			\0
			\n
			\r
			!
			"
			#
			$
			%
			&
			'
			(
			)
			*
			+
			,
			-
			.
			/
			0
			1
			2
			3
			4
			5
			6
			7

			8
			9
			:
			;
			<
			=
			>
			?
			@
			A
			B
			C
			D
			E
			F
			G
			H
			I
			J
			K
			L
			M
			N
			O
			P
			Q
			R
			S
			T
			U
			V
			W
			X
			Y
			Z
			[
			\
			]
			^
			_
			`
			a
			b
			c
			d
			e

			f
			g
			h
			i
			j
			k
			l
			m
			n
			o
			p
			q
			r
			s
			t
			u
			v
			w
			x
			y
			z
			{
			|
			}
			~

`assemble.ps1 debug.ps1 run-test.ps1 run-tests.ps1`

```
PS /mnt/c/reti_logiche/assembler> ./assemble.ps1 mio_programma.s
```

`./files/main.c`

```
PS /mnt/c/reti_logiche/assembler> ./debug.ps1 mio_programma
```

`./files/gdb_startup qquit rrun _main ./files/main.c`

```
PS /mnt/c/reti_logiche/assembler> ./run-test.ps1 mio_programma input.txt output.txt
```

```
PS /mnt/c/reti_logiche/assembler> ./run-tests.ps1 mio_programma cartella_test
```

`in_*.txt out_*.txt out_*.txt out_ref_*.txt \r\n \n`

`assemble.code-workspace` `Ubuntu` `wsl --list -v`

```
PS C:\Users\raffa> wsl --list -v
  NAME                   STATE           VERSION
* Ubuntu                 Stopped         2
  Ubuntu-22.04           Stopped         2
```

`NAME` `assemble.code-workspace` `wsl+ubuntu` `wsl+NOME-DELLA-DISTRO` `Ubuntu-22.04` `wsl+Ubuntu-22.04`

`assemble.code-workspace`

- `assembler`
- `pwsh`

`assemble.code-workspace` `"remoteAuthority":` `"uri":` `"uri": "/home/raff/reti_logiche/assembler"`

`./assemble.ps1 programma.s ./programma`

`bash pwsh pwsh`

`assemble.ps1`

```
gcc -m32 -o ...
```

```
gcc -m32 -no-pie -o ...
```