Arithmetic Instructions

- ADD, SUB, MUL, IMUL, DIV, IDIV…
  - ADD/SUB: add dest, src ; left one is source and destination

```
add eax, ebx    ; eax <- eax+ebx
add [esp], eax ; eax on top of the stack
add eax, [esp] ; top of the stack in eax
add eax, 4      ; immediate 4 added in eax
```

  - DIV/IDIV: div divisor ; dividend always in eax: result in eax and rest in edx

```
mov eax, 65 ; immediate 65 in eax
mov ecx, 4      ; 4 in ecx
div ecx      ; eax<- eax/ecx  edx<- eax%ecx
```

  - MUL/IMUL:

```
mul value       ; eax<-eax*value
mul dest, val1, val2
mul dest, val
```

Bitwise opeartions

- AND, OR, XOR, NOT
  - AND/OR/XOR dst, src
  - NOT eax

Branching

- JMP, JE, JLE, JNZ, JZ, JBE, JGE… :
  - sintax: `jmp address` ; EIP <- OFFSET address
  - Jump could depend on current status of condition codes, i.e. result of previous operations
    - suffix E jump if ZF==1; LE if ZF==1 ||SF==1; NZ if ZF==0; Z if ZF==1; …
    - Condition Codes: ZF – zero flag, SF – signed flag, OF – overflow flag, CF – carry flag
  - ADD/SUB: set ZF, SF, OF, CF
  - AND: OF = CF = 0, set ZF, SF
  - CMP: “cmp dest, src" only purpose is to set flags for following JMP instruction
  - JLE vs JBE : Less or Egual vs Below or Egual
    - signed vs unsigned

Data Moving

- MOV, MOVS, MOVSB, MOVSW, MOVZX, MOVSX, LEA… :
  - MOV dst, src
    - moves src in dst
    - reg to reg / mem to reg / reg to mem (NO mem to mem)
  - MOVS.. Move string from the ESI address to EDI one

Instruction	Description
MOVSB	Move byte at address DS:(E)SI to address ES:(E)DI
MOVSW	Move word at address DS:(E)SI to address ES:(E)DI
MOVSD	Move doubleword at address DS:(E)SI to address ES:(E)DI

    - se DF (Decrement Flag) è 0/1 EDI e ESI sono ***properly*** incrementati/decreased
    - Explicit operand can be given
    - Can be prefixed by REP to move ECX bytes/words/double words

`REP MOVSB ;` moves ECX bytes from ESI to EDI

Loop-ing

- LOOP dst : jump to destination until ECX is zero

```
    mov ecx, 5 ; ecx stands for extended counter
_proc:
    dec ecx ; decrements ecx
    loop _proc ; loops back to _procs
```

  - In NASM *loop label, reg_cunter*

- REP/REPE/REPZ/REPNE/REPNZ : like loop but for string management

```
mov esi, str1
mov edi, str2
mov ecx, 10h
rep cmps ; stops after 16 bytes or strings differ
```

  - `REPE CMPS m8, m8 ;` Find nonmatching bytes in ES:[(E)DI] and DS:[(E)SI].
  - `REPNE CMPS m16, m16;` Find matching words in ES:[(E)DI] and DS:[(E)SI].

Scan string (SCAS) example

- **SCAS/SCASB/SCASW/SCASD**
  - SCASB: Compare AL with byte at ES:(E)DI and set status flags.

- find the length of a NUL-terminated string:

```
MOV DI, DX       ;Starting address in DX (assume ES = DS)
MOV AL, 0        ;Byte to search for (NUL)
MOV CX, -1       ;Start count at FFFFh
CLD              ;Clear Direction Flag
                 ;DI=0 =>Increment DI after each character
REPNE SCASB ;Scan string for AL,inc CX for each char
MOV AX, -2       ;CX=-2 for len 0, -3 for len 1, ...
SUB AX, CX       ;Length in AX
```

Data Allocation

[variable-name] define-directive initial-value [,initial-value],...

- Variable-name: identify the storage space allocated.
- Define-directive:

Data directive

Directive	Description of Initializers
BYTE, DB (byte)	Allocates unsigned numbers from 0 to 255.
SBYTE (signed byte)	Allocates signed numbers from –128 to +127.
WORD, DW (word = 2 bytes)	Allocates unsigned numbers from 0 to 65,535 (64K).
SWORD (signed word)	Allocates signed numbers from –32,768 to +32,767.
DWORD, DD (doubleword = 4 bytes),	Allocates unsigned numbers from 0 to 4,294,967,295 (4 megabytes).
SDWORD (signed doubleword)	Allocates signed numbers from –2,147,483,648 to +2,147,483,647.
FWORD, DF (farword = 6 bytes)	Allocates 6-byte (48-bit) integers. These values are normally used only as pointer variables on the 80386/486 processors.
QWORD, DQ (quadword = 8 bytes)	Allocates 8-byte integers used with 8087-family coprocessor instructions.
TBYTE, DT (10 bytes),	Allocates 10-byte (80-bit) integers if the initializer has a radix specifying the base of the number.
REAL4	Short (32-bit) real numbers
REAL8	Long (64-bit) real numbers
REAL10	10-byte (80-bit) real numbers and BCD numbers

- Examples
- letter_c DB 'c' ; Allocate a single byte of memory, and initialize it to the letter 'c'.
- an_integer DD 12425 ; Allocate memory for an integer (4-bytes), and initialize it to 12425.
- a_float REAL4 2.32 ; Allocate memory for a float, and initialize it to 2.32
- message DB 'Hello',13,0 ; Allocate memory for a null terminated string "Hello\n"
- marks DW 0, 0, 0, 0 ; Both allocates memory for an array of 4 * 2 bytes, and ; initialize all elements to zero.
- marks DW 4 DUP (0) ; DUP allows multiple initializations to the same value
- name DB 30 DUP(?) ; Allocate memory for 30 bytes, uninitialized.
- matrix QW 12*10 ; Allocate memory for a 12*10 quad-bytes matrix

FITNESS

FAULT AND INTRUSION TOLERANT NETWORKED SYSTEMS

Laboratorio 1

- *Scrittura, assemblaggio ed esecuzione di un programma x86_64*
- *Debugging mediante gdb/peda*
- *Introduzione ai principali comandi GDB*

Esample: array_sum.asm

```
SECTION .data

valori    db   1,2,3
n         db   $-valori
somma  db 1

SECTION .text

global _start

_start:
          mov rax,0    ; rax = 0
          mov rcx,0
          mov cl, byte [n]
          lea rbx,[valori]

_loop:
          add al, byte [rbx]
          inc bl
          loop _loop, ecx

          mov [somma],rax
```

Execution (continue)

- *nasm -f elf64 -o as.o array_sum.asm*
- *ld -o as as.o*

- *gdb as*
  - To start debugging the program
- *break start*
- *r*
- …

```
[----------------------------------code---------------------
   0x4000aa:    and    BYTE PTR [rax],al
   0x4000ac:    add    BYTE PTR [rax],al
   0x4000ae:    add    BYTE PTR [rax],al
=> 0x4000b0 <_start>:    mov    eax,0x0
   0x4000b5 <_start+5>: mov    ecx,0x0
   0x4000ba <_start+10>:        mov    cl,BYTE PTR ds:0x6000e7
   0x4000c1 <_start+17>:        lea    rbx,ds:0x6000e4
   0x4000c9 <_loop>:    add    al,BYTE PTR [rbx]
```

Execution (continue)

- *disassemble _loop*

```
gdb-peda$ disassemble _loop
Dump of assembler code for function _loop:
   0x00000000004000c9 <+0>:     add    al,BYTE PTR [rbx]
   0x00000000004000cb <+2>:     inc    bl
   0x00000000004000cd <+4>:     addr32 loop 0x4000c9 <_loop>
   0x00000000004000d0 <+7>:     mov    QWORD PTR ds:0x6000e8,rax
   0x00000000004000d8 <+15>:    mov    eax,0x3c
   0x00000000004000dd <+20>:    mov    edi,0x0
   0x00000000004000e2 <+25>:    syscall
End of assembler dump.
```

- *break *0x00000000004000d0*
- *c* (continue until next break 0x4000d0)

```
gdb-peda$ info register rax
rax            0x6      0x6
```

1+2+3

Execution (continue)

- La prossima istruzione scirve il risultato in *somma (0x6000e8)*

```
0x4000d0 <_loop+7>:   mov     QWORD PTR ds:0x6000e8,rax
```

- Leggiamo il contenuto prima dell'istruzione (help x):

```
gdb-peda$ x/dw 0x6000e8
0x6000e8:       1
gdb-peda$ x/xw 0x6000e8
0x6000e8:       0x00000001
```

- Eseguiamo l'istruzione: `si` (step into: esegue una istruzione eventualmente entrando nella prossima funzione; in alternativa `next` esegue senza entrare in sottofunzioni)
- Infine leggiamo nuovamente il contenuto di somma

```
gdb-peda$ x/xw 0x6000e8
0x6000e8:       0x00000006
gdb-peda$ x/4xb 0x6000e8
0x6000e8:       0x06    0x00    0x00    0x00
```

Assignment 1

- Preparare un programma per nasm che confronta due stringhe contando il numero di caratteri uguali a partire dal primo e fino al primo diverso
- Eseguire una sessione di debug con gdb
- Preparare una relazione che spieghi programma e debugging

- Tempo: 1w
- Modalità: lavorare in coppie
- Punti: 8pt se tutto fatto bene, extra 2pt alla coppia che resolve il problema con meno linee di codice