

# Appunti su Assembler

Gabriele Frassi

A.A 2020-2021 - Primo semestre

# Indice

<b>1 Unimap</b>	<b>5</b>
<b>I Lezioni di Stea</b>	<b>6</b>
<b>2 Martedì 29/09/2020</b>	<b>7</b>
2.1 Linguaggio assembler . . . . .	7
2.2 Struttura di un calcolatore . . . . .	8
2.3 Ripasso: rappresentazione dell'informazione . . . . .	9
2.3.1 Numeri naturali . . . . .	9
2.3.2 Numeri interi . . . . .	9
2.3.2.1 Diagramma della farfalla . . . . .	10
2.3.2.2 Tabella di conversione da $-8$ a $+7$ in $p = 4$ . . . . .	12
2.3.2.3 Rappresentazione esadecimale . . . . .	12
2.4 Osservazione sul calcolo di MB, KB o GB occupati . . . . .	12
2.5 Struttura del calcolatore vista da un programmatore ASS . . . . .	13
<b>3 Mercoledì 30/09/2020</b>	<b>16</b>
3.1 Codifica macchina e codifica mnemonica . . . . .	16
3.2 Struttura di un'istruzione . . . . .	16
3.3 Esempio di programma . . . . .	16
3.4 Memoria occupata da una certa istruzione . . . . .	17
3.5 Indirizzamento degli operandi . . . . .	17
3.5.1 Indirizzamento di registro . . . . .	17
3.5.2 Indirizzamento immediato . . . . .	18
3.5.3 Indirizzamento di memoria . . . . .	18
3.5.3.1 Indirizzamento di tipo diretto . . . . .	18
3.5.3.2 Indirizzamento di tipo indiretto . . . . .	18
3.5.3.3 Indirizzamento con displacement e registro di modifica . . . . .	19
3.5.3.4 Indirizzamento bimodificato senza displacement . . . . .	19
3.5.3.5 Indirizzamento bimodificato con displacement . . . . .	19
3.5.4 Indirizzamento delle porte di I/O . . . . .	19
3.6 Principali istruzioni . . . . .	20
3.6.1 Istruzioni di trasferimento . . . . .	20
3.6.1.1 MOVE . . . . .	20
3.6.1.2 LOAD EFFECTIVE ADDRESS . . . . .	21
3.6.1.3 EXCHANGE . . . . .	21

<b>4 Giovedì 01/10/2020</b>	<b>22</b>
4.1 Concludiamo con le istruzioni di trasferimento . . . . .	22
4.1.1 INPUT e OUTPUT . . . . .	22
4.1.1.1 ASSEMBLER non è linguaggio ortogonale . . . . .	23
4.1.1.2 Uscita da registro a porta . . . . .	23
4.1.2 Pila . . . . .	23
4.1.2.1 Pila come memoria temporanea . . . . .	24
4.1.3 PUSHAD e POPAD . . . . .	25
4.2 Istruzioni aritmetiche . . . . .	25
4.2.1 ADD e SUBTRACT . . . . .	25
4.2.2 INCREMENT e DECREMENT . . . . .	26
4.2.3 ADD WITH CARRY e SUBTRACT WITH BORROW . . . . .	27
4.2.4 NEGATE . . . . .	29
4.2.5 COMPARE . . . . .	29
4.2.6 INTEGER e INTEGER MULTIPLY . . . . .	30
4.2.7 DIVIDE e INTEGER DIVIDE . . . . .	31
4.2.8 Conclusioni . . . . .	33
<b>5 Venerdì 02/10/2020</b>	<b>34</b>
5.1 Continuiamo con le istruzioni aritmetiche . . . . .	34
5.1.1 Estensione di campo . . . . .	34
5.1.2 Naturali . . . . .	34
5.1.3 Interi . . . . .	34
5.1.4 CONVERT WORD TO DOUBLEWORD in EAX . . . . .	34
5.2 Istruzioni di traslazione . . . . .	35
5.2.1 SHIFT LOGICAL LEFT . . . . .	35
5.2.2 SHIFT ARITHMETIC LEFT . . . . .	36
5.2.3 SHIFT LOGICAL RIGHT . . . . .	36
5.2.4 SHIFT ARITHMETIC RIGHT . . . . .	36
5.3 Istruzioni di rotazione . . . . .	37
5.3.1 ROTATE LEFT . . . . .	37
5.4 Istruzioni logiche . . . . .	38
5.4.1 NOT . . . . .	38
5.4.2 AND . . . . .	38
5.4.3 OR . . . . .	38
5.4.4 XOR (OR esclusivo) . . . . .	39
5.4.5 Utilizzi di questi operatori . . . . .	39
5.5 Istruzioni di controllo . . . . .	40
5.5.1 JUMP . . . . .	40
5.5.2 JUMP if condition met . . . . .	40
5.5.3 Sottoprogrammi . . . . .	41
5.5.3.1 CALL . . . . .	41
5.5.3.2 RET . . . . .	41
5.5.3.3 NO OPERATION . . . . .	41
5.5.3.4 HALT . . . . .	42
5.6 Protezione ed istruzioni privilegiate . . . . .	42
<b>6 Martedì 06/10/2020</b>	<b>43</b>
6.1 Assemblatore . . . . .	43
6.2 Struttura di un programma assembler . . . . .	43
6.2.1 Assembler case-sensitive o case-insensitive? . . . . .	44
6.2.2 Cosa cambia rispetto agli esempi di programmi visti in passato? . . . . .	44
6.2.2.1 Attenzione al jump : confronto col C++ . . . . .	44

6.2.3	Riga di codice . . . . .	45
6.3	Direttive . . . . .	45
6.3.1	Dichiarazione di variabili . . . . .	45
6.3.1.1	Alternativa per dichiarare vettori (comando FILL) . . . . .	46
6.3.1.2	Codifica ASCII e caratteri speciali . . . . .	46
6.3.2	INCLUDE . . . . .	47
6.3.3	SET . . . . .	47
6.3.3.1	Calcolare memoria occupata . . . . .	47
6.3.3.2	Costanti numeriche . . . . .	48
6.4	Controllo di flusso . . . . .	48
6.4.1	if...then ...else . . . . .	49
6.4.2	for... . . . . .	49
6.4.3	do...while . . . . .	50
6.4.4	<i>Spaghetti-like coding</i> . . . . .	50
6.4.5	LOOP . . . . .	51
6.4.5.1	For discendente e for ascendente con LOOP . . . . .	51
6.4.6	LOOP condizionato . . . . .	51
6.4.7	Utilità dei LOOP e manipolazione stringhe . . . . .	52
6.5	Sottoprogrammi e passaggio dei parametri . . . . .	52
<b>7</b>	<b>Giovedì 08/10/2020</b>	<b>53</b>
7.1	Uso dei registri ed effetti collaterali . . . . .	53
7.2	Sottoprogramma principale . . . . .	54
7.3	Dichiarazione e allocazione di spazio per la pila . . . . .	54
7.4	Ingresso/uscita e sottoprogrammi di utilità . . . . .	55
7.5	Osservazioni sulla tabella ASCII . . . . .	55
7.5.1	I/O da tastiera e video . . . . .	56
7.5.2	File utility . . . . .	56
7.5.2.1	Sottoprogrammi di I/O (su cui si basano i sottoprogrammi della sezione successiva) . . . . .	56
7.5.2.2	Sottoprogrammi a livello più alto . . . . .	56
7.5.2.3	Sottoprogrammi per l'ingresso/uscita di numeri esadecimali . . . . .	57
7.5.2.4	Sottoprogrammi per l'ingresso/uscita di numeri decimali . . . . .	57
7.6	Istruzioni che manipolano le stringhe . . . . .	58
7.6.1	MOVE DATA FROM STRING TO STRING . . . . .	59
7.6.2	Istruzioni <i>Direction Flag</i> (STD e CLD) . . . . .	59
7.6.3	LODSsuf ( <i>load string</i> ) e STOSsuf ( <i>store string</i> ) . . . . .	60
7.6.4	[Privilegiate] Istruzioni stringa per l'I/O - INSsuf, OUTSsuf . . . . .	60
7.6.5	COMPARE STRINGS (CMPSsuf) - confronto memoria-memoria . . . . .	61
7.6.6	SCAN STRING (SCASsuf) - confronto registro-memoria . . . . .	61
7.6.7	Prefissi di ripetizione . . . . .	61
7.6.8	Utilità delle due direzioni . . . . .	62
<b>8</b>	<b>Venerdì 09/10/2020</b>	<b>63</b>
8.1	Conclusione su Assembler . . . . .	63
8.1.1	Differenze tra compilatore e assemblatore . . . . .	63
8.1.2	Tempo di esecuzione di un programma . . . . .	63
8.1.3	Lunghezza delle istruzioni e tempo di fetch . . . . .	64
8.1.4	Tempo di esecuzione delle istruzioni . . . . .	64
8.1.4.1	Come si evitano moltiplicazioni e divisioni? . . . . .	64

<b>II Esercitazioni di Zippo</b>	<b>66</b>
<b>9 Martedì 06/10/2020</b>	<b>67</b>
9.1 Assemblaggio . . . . .	67
9.1.1 File listato . . . . .	67
9.2 Debugging . . . . .	69
9.2.1 Comandi . . . . .	69
<b>10 Venerdì 16/10/2020</b>	<b>75</b>
<b>11 Venerdì 23/10/2020</b>	<b>81</b>
11.1 Esercizio sul fattoriale . . . . .	81
11.2 Calcolo del binomiale . . . . .	86

# Capitolo 1

## Unimap

1. **Mar 29/09/2020 11:45-13:45 (2:0 h)** lezione: Lezione: Introduzione al corso ed informazioni pratiche. Richiami sulla rappresentazione dei numeri naturali ed interi in base 2. Schema a blocchi del calcolatore: memoria, spazio di I/O, processore. I registri del processore, condizioni al reset. (GIOVANNI STEA)
2. **Mer 30/09/2020 10:45-12:45 (2:0 h)** lezione: Codifica macchina e codifica mnemonica delle istruzioni e primo esempio di programma in Assembler. Indirizzamento degli operandi nelle istruzioni operative. Istruzioni di trasferimento (inizio). (GIOVANNI STEA)
3. **Gio 01/10/2020 10:45-12:45 (2:0 h)** lezione: Istruzioni di trasferimento (fine). Istruzioni aritmetiche (inizio). (GIOVANNI STEA)
4. **Ven 02/10/2020 15:00-17:00 (2:0 h)** lezione: istruzioni aritmetiche (fine). Istruzioni di traslazione e rotazione. Istruzioni logiche. Istruzioni di controllo. (GIOVANNI STEA)
5. **Mar 06/10/2020 11:45-13:45 (2:0 h)** lezione: Programmare in linguaggio Assembler: struttura sintattica di un programma. Direttive in Assembler: dichiarazione di variabile e di costante. Strutture di controllo di flusso tipiche dei linguaggi ad alto livello e loro traduzione in linguaggio Assembler. (GIOVANNI STEA)
6. **Mer 07/10/2020 10:45-12:45 (2:0 h)** esercitazione: (in copresenza con Ing. R. Zippo) Esercitazione Assembler. Presentazione ambiente di sviluppo. Uso del debugger per verifica del programma e controllo degli errori. (GIOVANNI STEA)
7. **Gio 08/10/2020 10:45-12:45 (2:0 h)** lezione: Sottoprogrammi e passaggio dei parametri. Gestione della pila. I/O e sottoprogrammi di ingresso/uscita. Istruzioni stringa. (GIOVANNI STEA)
8. **Ven 09/10/2020 15:00-17:00 (2:0 h)** lezione: Note sulla programmazione Assembler. Generalità sulle reti logiche. Modello, limiti del modello e non contemporaneità. descrizione mediante tabelle di verita'. Esempi di reti combinatorie semplici. AND e OR a piu' di due ingressi. Algebra di Boole (GIOVANNI STEA)
9. **Ven 16/10/2020 15:00-17:00 (2:0 h)** esercitazione: Esercitazione: (in copresenza con Ing. R. Zippo) Esercitazione Assembler. Svolgimento di esercizi di programmazione. (GIOVANNI STEA)
10. **Ven 23/10/2020 15:00-17:00 (2:0 h)** esercitazione: (in copresenza con Ing. R. Zippo) Esercitazione Assembler. Svolgimento di esercizi di programmazione. (GIOVANNI STEA)

# **Parte I**

# **Lezioni di Stea**

# Capitolo 2

**Martedì 29/09/2020**

## 2.1 Linguaggio assembler

**Nome preciso** Il nome preciso è *Assembly*, ma lo chiameremo *Assembler* per ragioni storiche (a Pisa è sempre stato chiamato così)

**Differenza da altri linguaggi** Il linguaggio Assembler è di basso livello, in contrapposizione al C++ che è di alto livello. Con questo linguaggio saremo in grado di scrivere **istruzioni macchina** eseguite dal processore.

- Con linguaggio ad alto livello intendiamo un linguaggio i cui costrutti sono più vicini al ragionamento dell'essere umano
- Con linguaggio a basso livello intendiamo un linguaggio i cui costrutti sono più vicini al ragionamento della macchina.

Scrivere in assembler significa imparare a ragionare come la macchina!

**Sintassi simbolica** Con Assembler non scriveremo in linguaggio macchina (sequenze di zeri e uno incomprensibili per l'uomo), ma ricorremo a una sintassi simbolica. Qual è la differenza?

- In C++ il compilatore effettua una traduzione vera e propria. Basti pensare che dietro l'istruzione

`a=b;`

possono celarsi centinaia di istruzioni macchina.

- In Assembler abbiamo l'**assemblaggio**, svolto dall'assemblatore. Ponendo quanto segue

`MOV %AX, %BX`

rappresentiamo un numero binario. Questa è una traduzione 1 : 1!

## Ulteriori differenze rispetto al C++

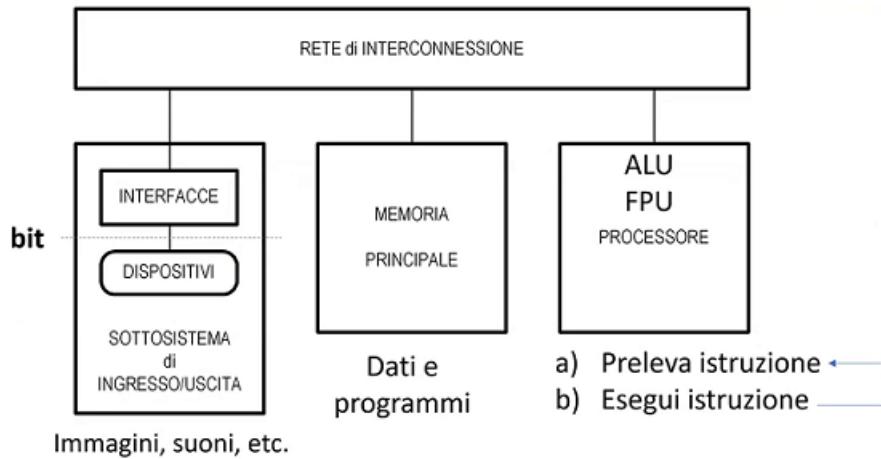
- Non sono presenti i cosiddetti *costrutti di flusso strutturato*: for, while, dowhile, ifthenelse, case. Saranno presenti istruzioni di salto *jump*.
- Le variabili non hanno un tipo. Avremo stringhe di bit. Cosa rappresenti una variabile è pensiero esclusivamente nostro, non della macchina

**Assembler processor-specific** L'assembler è specifico per ogni processore: i comandi possono cambiare. Segue che non sia un linguaggio, portabile, a differenza del C++ o dei linguaggi che impareremo simultaneamente a progettazione web. Utilizzeremo, in questo corso, assembler per *Intel x86*, visto la validità dei suoi principi per quasi tutte le versioni di Assembler. Non lo studieremo tutto, considerando la vastità del manuale (20 ore per 2000 pagine? impensabile)

**Utilità di Assembler** Assembler è utilizzato per *sistemi Embedded*. Al di là di questo un ingegnere informatico deve sapere come si comporta un processore!

## 2.2 Struttura di un calcolatore

Un calcolatore può essere immaginato come una serie di blocchi connessi da una *rete di interconnessione* detta **bus**. In particolare individuiamo nello schema funzionale:



1. **Sottosistema di ingresso/uscita:** area che si occupa di compiere la traduzione dal mondo esterno al calcolatore e viceversa. Immagini, suoni, qualunque contenuto vengono codificate in stringhe di bit. Nella stessa area abbiamo il dispositivo (che non è detto faccia sia input che output, pensiamo alla tastiera e al mouse): questo non è collegato direttamente al bus, ma passa per *interfacce*, reti logiche che adattano il dispositivo al bus (attraverso protocolli il calcolatore può interfacciarsi con i dispositivi senza conoscerne la struttura fisica)
2. **Memoria principale:** ospita i programmi (le istruzioni da eseguire) e i dati (non tutti, alcuni di questi potrebbero trovarsi nel sottosistema introdotto prima)
3. **Processore:** contiene al suo interno almeno le seguenti unità:

- la ALU (*Arithmetic Logic Unit*): unità che si occupa di eseguire istruzioni logiche (AND, OR e NOT) e aritmetiche (relativamente a numeri naturali e interi)
- la FPU (*Floating Point Unit*): unità che svolge istruzioni aritmetiche relative ai numeri reali

**Cosa fa il processore** Il processore, ciclicamente:

- preleva un'istruzione macchina dalla memoria
- la esegue

## 2.3 Ripasso: rappresentazione dell'informazione

### 2.3.1 Numeri naturali

Con  $N$  bit possiamo rappresentare  $2^N$  numeri naturali, precisamente quelli compresi nell'intervallo  $[0, 2^N - 1]$ . Come nella base 10 anche qua abbiamo un sistema basato sulla presenza di cifre più significative e cifre meno significative (rispettivamente MSB, *Most significant bit*, e LSB, *Least Significant Bit*). Un numero in base 2 avente la seguente forma

$$b_{N-1}, b_{N-2}, \dots, b_1, b_0$$

può essere rappresentato in base 10 mediante la seguente formula

$$X = \sum_{i=0}^{N-1} b_i 2^i$$

ricordiamo, inoltre, l'algoritmo **DIV&MOD** per convertire numeri dalla base 10 alla base 2 (vedere dispensa di Rappresentazione dell'informazione, FdP).

### 2.3.2 Numeri interi

Con  $N$  bit possiamo rappresentare  $2^N$  numeri interi, precisamente quelli compresi nell'intervallo  $[-2^{N-1}, 2^{N-1} - 1]$ . Osserviamo che la cosa non è simmetrica:

- Con  $N = 8$  abbiamo  $[-128, 127]$
- Con  $N = 16$  abbiamo  $[-32768, 32767]$

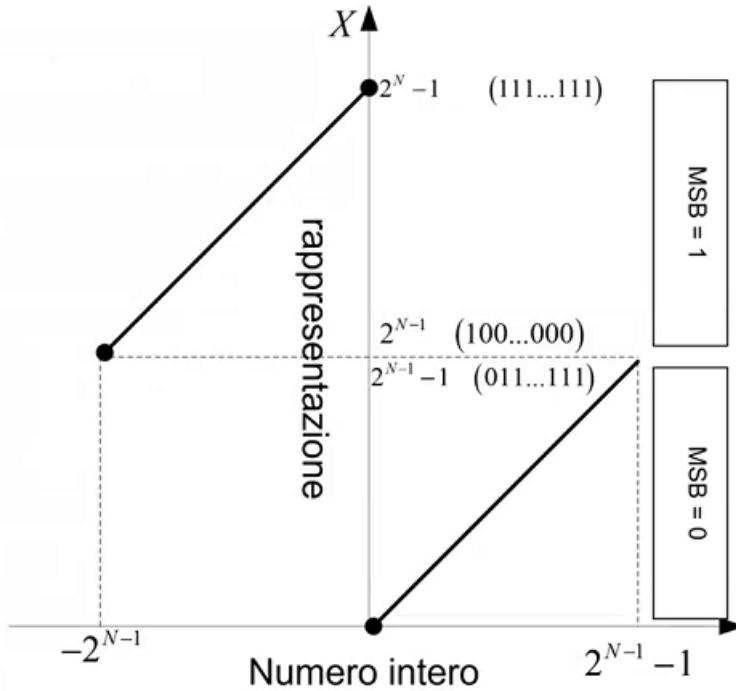
La rappresentazione che ci interessa è quella **in complemento a 2**. Per ottenere un numero da base 10 a base 2 procederemo identificando la seguente stringa di bit  $X$  (numeri naturali per rappresentare numeri interi)

$$X = \begin{cases} x & x \geq 0 \\ 2^N + x & x < 0 \end{cases} \quad \text{purchè si abbia } x \in [-2^{N-1}, 2^{N-1} - 1]$$

ricordiamo che la cosa deve essere verificata, il metodo ci porta ad ottenere una sequenza binaria in ogni caso. Se la condizione non è soddisfatta quella sequenza non rappresenta il numero da cui siamo partiti. Tutto questo può essere rappresentato anche così:  $X = |x|_{2^N}$ . Vedremo questa notazione nella parte di Aritmetica del corso.

### 2.3.2.1 Diagramma della farfalla

Quanto detto prima può essere tradotto sfruttando la seguente immagine



Lungo l'asse delle ascisse abbiamo gli interi  $x$ , lungo l'asse delle ordinate le stringhe di bit  $X$  (naturali, capiremo per bene nella parte di Aritmetica). Osserviamo che

- Nel semiasse positivo abbiamo una retta con coefficiente  $m = 1$  dove  $X = x$ . Quest'area consiste nei numeri positivi ed è quella con MSB = 0
- Nel semiasse negativo abbiamo una retta parallela alla precedente traslata di  $2^N$  ( $X = x + 2^N$ ). Quest'area consiste nei numeri negativi ed è quella con MSB = 1

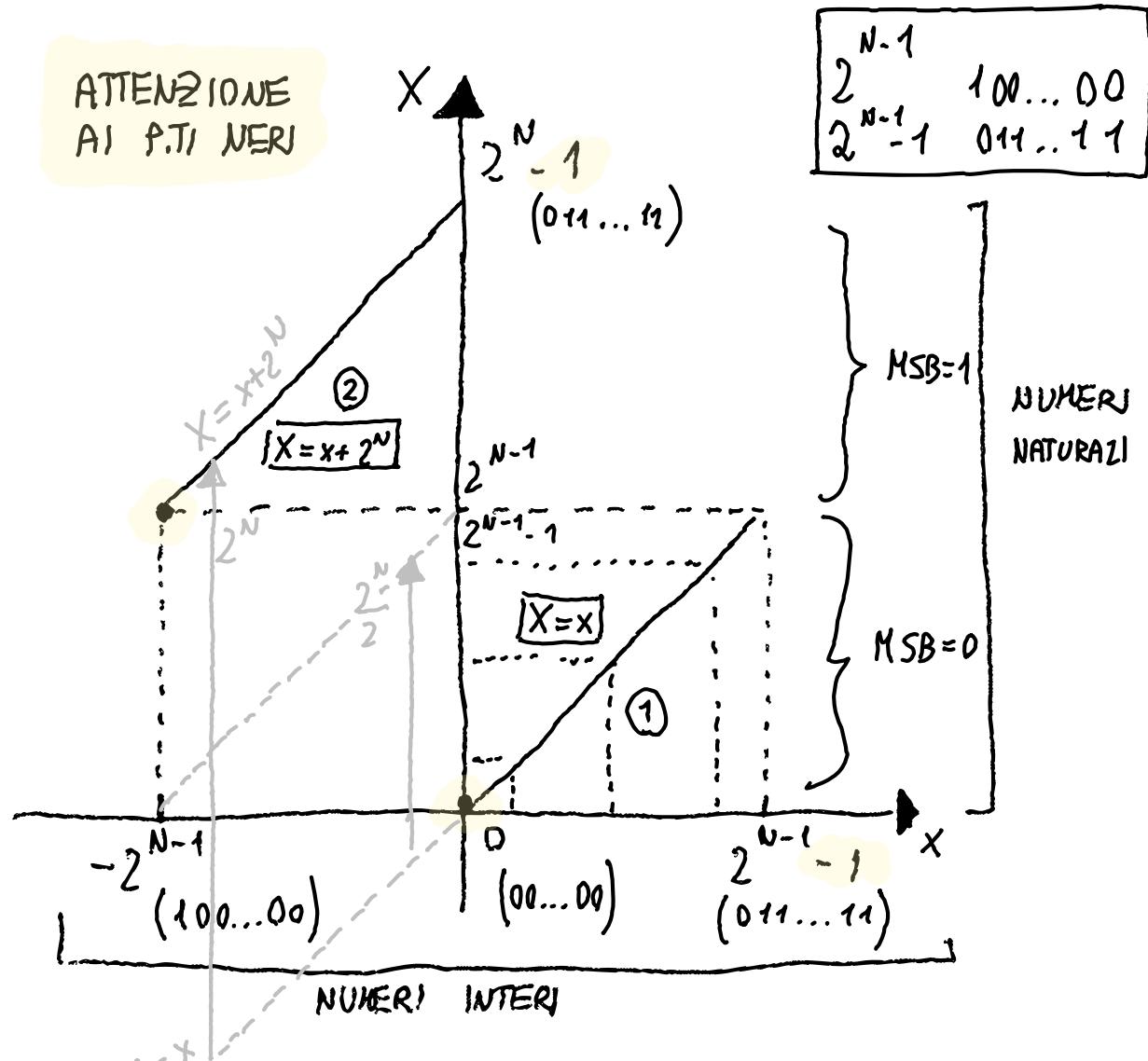
Ricordiamo che l'ultimo numero con  $MSB = 0$  ( $2^{N-1} - 1$ ) sarà del tipo  $011\dots11$ , mentre l'ultimo numero con  $MSB = 0$  ( $2^N - 1$ ) sarà del tipo  $111\dots11$ . Dallo stesso disegno possiamo capire il processo inverso per passare da base 2 a base 10

$$x = \begin{cases} X & X_{N-1} = 0 \\ -(\bar{X} + 1) & X_{N-1} = 1 \end{cases}$$

ottengo il naturale *complementando* i bit della rappresentazione (cioè invertendo gli zeri e gli uno). Spiegheremo questa cosa nella parte di Aritmetica del corso! Prendiamo il seguente esempio:

$$\begin{aligned} & 10110101 \\ & -(01001010 + 1) \\ & -(01001011) = -(64 + 8 + 2 + 1) = -75 \end{aligned}$$

# DISEGNO DELLA FARFALLA (IN C2)



$$X = \begin{cases} x & x \geq 0 \\ x + 2^N & x < 0 \end{cases}$$

① CHIARAMENTE UNA RETTA  
CON  $m = 1$

$$[0, 2^{N-1}-1]$$

② RETTA TRASLATA

### 2.3.2.2 Tabella di conversione da $-8$ a $+7$ in $p = 4$

N	A	a
0	0000	+0
1	0001	+1
2	0010	+2
3	0011	+3
4	0100	+4
5	0101	+5
6	0110	+6
7	0111	+7
8	1000	-8
9	1001	-7
10	1010	-6
11	1011	-5
12	1100	-4
13	1101	-3
14	1110	-2
15	1111	-1

### 2.3.2.3 Rappresentazione esadecimale

Molto spesso le sequenze binarie sono lunghe e difficili da comprendere. Potremo compattare convertendo in base 10, ma questo richiede dei calcoli. Proprio per questo andiamo ad adottare la **notazione esadecimale**. 4 bit consistono in un numero compreso tra 0 e 15, sfruttando una scorciatoia (descritta nella dispensa di rappresentazione dell'informazione, FdP) possiamo sostituire in modo rapido leggendo blocchi di cifre da destra verso sinistra.

Quadr. Simb.      Quadr. Simb.      Quadr. Simb.      Quadr. Simb.

0000	0	0100	4	1000	8	1100	C
0001	1	0101	5	1001	9	1101	D
0010	2	0110	6	1010	A	1110	E
0011	3	0111	7	1011	B	1111	F

**Esempio 1**  $\underbrace{1011}_{B} \mid \underbrace{1001}_9 \Rightarrow 0xB9$

**Esempio 2**  $\underbrace{1100}_{C} \mid \underbrace{0001}_1 \Rightarrow 0xC1$

## 2.4 Osservazione sul calcolo di MB, KB o GB occupati

Ricordarsi che

$$2^{10} = 1 \text{ KB} \quad 2^{20} = 1 \text{ MB} \quad 2^{30} = 1 \text{ GB}$$

**Esempio** Definiamo lo spazio di memoria come l'insieme di  $2^{32}$  locazioni.

$$2^{32} = 2^{30} \cdot 2^2 = 4 \text{ GB}$$

## 2.5 Struttura del calcolatore vista da un programmatore ASS

Cosa vede un programmatore quando programma un calcolatore? Essenzialmente tre cose: spazio di memoria, spazio di I/O e processori.

**Spazio di memoria** Con spazio di memoria intendiamo una sequenza lineare e contigua di locazioni: ciascuna di esse ha capacità di un byte ed è identificata da un numero naturale a 32bit detto indirizzo. Se abbiamo 32 bit significa che potremo esprimere  $2^{32}$  indirizzi diversi, quindi avremo  $2^{32}$  locazioni possibili (si va da  $0x000\dots00$  a  $0x111\dots11$ , se ho  $2^{32}$ ). Solitamente si accede alla memoria per le seguenti motivazioni:

- Prelevare istruzioni
- Prelevare gli operandi delle istruzioni (la maggior parte si trova in memoria, molto spesso nel processore - in particolare nella ALU)

può svolgere accessi di lettura o di scrittura, possibili su singola locazione (operando a 8 bit), doppia locazione (operando a 16 bit) e quadrupla locazione (32 bit).

**Memoria del computer** La memoria del computer si divide in

- **RAM**, *Random Access Memory*. Questa memoria è di tipo volatile, cioè le informazioni vengono mantenute finché c'è tensione. C'è un problema: cosa fa il processore al momento dell'accensione se l'accesso è appunto casuale? Questo non può succedere e richiede l'introduzione della...
- **ROM**, *Read Only Memory*. Memoria di sola lettura, non volatile, che contiene il programma eseguito dal processore al momento dell'accensione.

**Tipi di accessi** Accessi di tipo diverso, dato lo stesso indirizzo, restituiscono un contenuto diverso. Negli accessi a 16/32bit si utilizza l'indirizzo più piccolo delle 2/4 locazioni. L'indirizzo più grande contiene i bit più significativi, segue quanto presente nell'immagine

	b7	b0			
0x3F65B432		0x1C	Lettura a	Indirizzo	Contenuto
0x3F65B433		0x39	8 bit (byte)	0x3F65B432	0x1C
0x3F65B434		0xA2	16 bit (word)	0x3F65B432	0x391C
0x3F65B435		0xC6	32 bit (double word)	0x3F65B432	0xC6A2391C

**Dimensione dello spazio di memoria** Che dimensione hanno  $2^{32}$  bit? 4GB di memoria! Come faccio a sapere se posso riferire un indirizzo (in un calcolatore non sono presenti solo io coi miei programmi, c'è anche la ROM, per esempio)? Non è un problema nostro: gli indirizzi in Assembler sono simbolici, ci pensa l'assemblatore a mappare la variabile su una cella utilizzabile.

**Spazio di I/O** Lo spazio di Input/Output consiste in  $2^{16} = 64k$  locazioni (dette anche porte). Ciascuna porta ha una capacità pari ad un byte ed è indirizzabile mediante un indirizzo a 16bit. Possiamo accedere allo spazio di I/O usando due particolari istruzioni (IN e OUT). Contrariamente alle celle dello spazio di memoria le porte di I/O non sono intercambiabili: a un certo indirizzo abbiamo la tastiera, a un altro il mouse. Segue la necessità di conoscere gli indirizzi! Generalmente

- si eseguono istruzioni IN per prelevare uno o più byte da un dispositivo: dati da elaborare o semplicemente informazioni sullo stato del dispositivo.
- si eseguono istruzioni OUT per trasmettere uno o più byte a un dispositivo: dati elaborati o informazioni per modificare lo stato del dispositivo.

**Processore** Il processore consiste in una collezione di registri. Un registro consiste in una piccola locazione di memoria, avente 32 bit. Si hanno due tipi di registri:

- **Registri generali.** Si osserva che tutti i registri sono introdotti dalla lettera E: questa sta per *Extended*. Precedentemente un registro occupava solo 16bit, successivamente sono stati estesi a 32 bit. La E risolve anche questioni non banali di compatibilità: è possibile richiamare l'intero registro con i suoi 32 bit col nome *EXY*, ma possiamo riferire la parte bassa dei registri col nome vecchio (*XY*)!



Risulta possibile richiamare la prima parte bassa e la seconda parte bassa con *XH* ed *YL*, dove *H* sta per *High* ed *L* sta per *Low*.

I nomi dei registri sono i seguenti:

- **EAX: Accumulator.** È il registro che serve per fare calcoli aritmetici. Alcune istruzioni aritmetiche lo usano per contenere operandi e risultati
- **EBX: Base.** Veniva spesso usato come indirizzo di base per l'accesso in memoria
- **ECX: Counter.** Viene usato come contatore nei cicli (la variabile "i" del for viene spesso mappata sul registro CX o parti di esso)
- **EDX: Data.** Anche questo viene usato come operando da istruzioni aritmetiche.
- **ESI: Source Index.** Veniva usato come registro indice per accessi in memoria
- **EDI: Destination Index.** Come sopra
- **EBP: Base pointer.** Veniva usato come registro "base" per accessi in memoria.

Alcuni di questi sono utilizzati per particolari funzioni:

- EAX è detto registro accumulatore ed è usato da alcune istruzioni aritmetiche (per esempio per variabili incremento)

- ESI, EDI, EBX ed EBP sono detti registri puntatore, dove  $B$  sta per *base* ed  $I$  sta per *indice*.
- ESP, utilizzato per indirizzare la pila. Solitamente si utilizza per gestire sottoprogrammi

Per comprendere l'utilità di alcuni registri leggere sull'*indirizzamento di memoria*.

- **Registri di stato.** I registri di stato sono due:

- EIP, cioè *Instruction Pointer register* (detto anche *program counter*). Consiste nel registro contenente la locazione di memoria a partire dalla quale sarà prelevata la prossima istruzione. Il suo contenuto è fissato al momento dell'accensione del processore! Raffiniamo quanto detto all'inizio: il processore
  1. preleva dalla memoria, precisamente all'indirizzo EIP, la nuova istruzione
  2. incrementa EIP del numero di byte dell'istruzione che ha prelevato (si parla di istruzioni contigue, ovviamente la cosa cambia se si incontrano salti)
  3. esegue l'istruzione e ritorna allo step (1)

EIP inizialmente vale 0xFFFF0000: la prima istruzione si trova lì. Ovviamente le celle successive dovranno essere implementate in ROM.

- EF, cioè *Extended Flag register*. Il registro consiste in 32 elementi detti *flag*. Ci interessano, in particolare, i seguenti flag:
  - \* OF (Overflow): elemento impostato ad 1 quando il risultato generato dall'ultima istruzione *trabocca*
  - \* SF (Sign): elemento impostato ad 1 quando il risultato generato dall'ultima istruzione restituisce un qualcosa con *MSB = 1*
  - \* ZF (Zero): elemento impostato ad 1 quando il risultato generato dall'ultima istruzione restituisce un qualcosa con tutti bit nulli
  - \* CF (Carry): elemento impostato ad 1 quando il risultato generato dall'ultima istruzione genera un riporto o un prestito.

Per gli interi ci serviranno OF, SF, ZF (CF è inutile), per i naturali CF e ZF (OF ed SF sono inutili). Nel registro dei flag quelli che ci interessano valgono 0, inizialmente.

# Capitolo 3

**Mercoledì 30/09/2020**

## 3.1 Codifica macchina e codifica mnemonica

Presente a pagina 7 della dispensa di Assembler.

## 3.2 Struttura di un'istruzione

Le istruzioni sono formate da tre campi:

- Il **codice operativo**, il nome dell'istruzione che vogliamo eseguire
- Il **suffisso di lunghezza** (lunghezza degli operandi): sono ammesse come lunghezze B (byte, 8bit), W (word, 16bit), L (long, 32bit). Non dobbiamo porre il suffisso obbligatoriamente: ci pensa l'assemblatore se la lunghezza degli operandi può essere intuita (per esempio quando almeno uno degli operandi è un registro).
- **Operandi**: le istruzioni ammettono 0/2 operandi (sono poche le istruzioni con 0 operandi). Nelle istruzioni con due operandi si distingue l'operando **sorgente** dall'operando **destinatario**. Se si ha un solo operando quello è detto **source** (destinatario è implicito) o **destinatario** (se l'istruzione lavora con un solo operando e pone nell'operando stesso il risultato del suo lavoro). I due operandi (salvo rare eccezioni) hanno la stessa lunghezza! Gli operandi possono essere in registri o porte I/O, ma possono essere anche costanti!

Un esempio di istruzione è la seguente

`ADD %BX, pippo`

il contenuto presente all'indirizzo di una locazione di memoria (*pippo*) viene incrementato del valore contenuto nella libreria BX. Il linguaggio Assembler consente al programmatore di riferire le locazioni di memoria con nomi simbolici che l'assemblatore tradurrà in indirizzi. L'assemblatore si incaricherà di sostituire lo stesso indirizzo tutte le volte che trova scritto *pippo*.

## 3.3 Esempio di programma

Presente un esempio di programma, accompagnato da spiegazione, da pagina 8 a pagina 11 della dispensa di Assembler.

## 3.4 Memoria occupata da una certa istruzione

Sia sul libro di Corsini che sulla dispensa di Stea si definisce un'istruzione come una stringa che occupa da 1 a 14 byte. Tenendo conto delle cose appena introdotte osserviamo che

- Il numero di operandi è uno degli indicatori dello spazio occupato da una certa istruzione (è ovvio che un'istruzione senza operandi possa occupare meno di un'istruzione con un operando, stessa cosa se mettiamo a confronto istruzioni con uno e due operandi, rispettivamente)
- Lo spazio occupato dagli operandi dipende dall'indirizzamento adottato: nel caso di indirizzamento diretto abbiamo una costante; in tutti gli altri casi andiamo a salvare l'indirizzo dell'area di memoria in cui si trova un certo operando.
- Ricordiamo che un processore, nell'ordine:
  - estraе da EIP l'indirizzo dell'istruzione successiva
  - incrementa EIP in modo tale che questa punti alla nuova istruzione successiva
  - esegue l'istruzione estratta poco fa ed eventualmente preleva dalla memoria gli operandi necessari per eseguire l'istruzione (usando gli indirizzi salvati)
- Ricordarsi le dimensioni di un'istruzione: sono allocati al più 4byte per Displacement! Segue che non è possibile, in un'istruzione a due operandi, fare indirizzamenti di memoria in entrambi gli operandi.

## 3.5 Indirizzamento degli operandi

Ricordiamo la struttura di un'istruzione

`OPCODEsize source, destination`

### 3.5.1 Indirizzamento di registro

L'indirizzamento **NON** è possibile con i registri di stato. Possiamo scegliere tra i seguenti registri generali:

- 8 registri a 32bit (EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP)
- 8 registri a 16 bit (AX, BX, CX, DX, SI, DI, SP , BP)
- 8 registri ad 8 bit (AH, BH, CH, DH, AL, BL , CL, DL)

operandi di registro possono essere sia sorgente che destinatario

`OPCODE %DI`  
`OPCODE %EAX, %EBX`  
`OPCODE %AH, %CL`

la prima istruzione lavora su un operando a 16bit, la seconda su operando a 32bit, la terza su operandi ad 8 bit. Noi non sappiamo quali sono le istruzioni (e quindi la dimensione che occuperanno queste istruzioni), ma sappiamo già che potremo omettere il suffisso di dimensione (grazie alla presenza di almeno un operando con indirizzamento di registro).

### 3.5.2 Indirizzamento immediato

Questo tipo di indirizzamento può essere usato solo nell'operando sorgente: scriviamo una costante direttamente nell'istruzione (per ovvie ragioni non ha senso porre una costante nel destinatario).

```
OPCODE $0x20, %AL  
OPCODE $0x5683a20b, %ECX
```

nella prima lavora su operandi ad 8 bit, nella seconda su operandi a 32bit.

### 3.5.3 Indirizzamento di memoria

L'indirizzamento di memoria è complicato, ma fondamentale: il processore, per la maggior parte del tempo, copia pezzi di memoria da una parte a un'altra. L'indirizzamento di memoria è possibile sia con l'operando sorgente che con quello destinatario, ma non è possibile farlo in entrambi in una stessa istruzione (l'assemblatore da errore se ci proviamo).

**Caso più generale** Il caso più generico di indirizzo è il seguente

$$\text{Indirizzo} = |\text{base} + \text{indice} \times \text{scala} \pm \text{displacement}|_{\text{modulo}_{2^{32}}}$$

cioè

```
OPCODEsfx +-disp(base,indice,scala)
```

- la base e l'indice consistono in registri generali a 32 bit. Precedentemente era obbligatorio porre un registro B in base e un registro I in indice: oggi si offre maggiore flessibilità e si possono utilizzare tutti i registri generali.
- scala è una costante che può avere per valore 1 (valore default se non indicato), 2, 4, 8.
- displacement è una costante intera.

#### 3.5.3.1 Indirizzamento di tipo diretto

Nelle cose viste fino ad ora abbiamo fatto indirizzamenti di memoria diretti, cioè indirizzamenti con solo il displacement.

```
OPCODEW 0x00002001
```

#### 3.5.3.2 Indirizzamento di tipo indiretto

Specifico un solo registro, precisamente un registro puntatore.

**Registro base** Poniamo la cosa nella seguente forma

```
OPCODEL (%EBX)
```

dove EBX consiste nel registro contenente l'indirizzo. Attenzione: è necessario indicare il suffisso di lunghezza, non abbiamo un indirizzamento di registri.

**Registro indice** Se volessi indicare un registro indice pongo

OPCODEL (,%ESI, 4)

la scala moltiplica solo l'indice e non la base.

### 3.5.3.3 Indirizzamento con displacement e registro di modifica

OPCODEW 0x002A3A2B (%EDI)

Indirizzo un operando a 16bit, che si trova nella doppia locazione il cui indirizzo si ottiene sommando (modulo  $2^{32}$ ) il displacement e il contenuto di EDI. Questa cosa è molto versatile per i vettori

### 3.5.3.4 Indirizzamento bimodificato senza displacement

OPCODEW (%EBX, %EDI)

OPCODEW (%EBX, %EDI, 8)

utilizzo due registri puntatori ponendo, eventualmente, la scala.

### 3.5.3.5 Indirizzamento bimodificato con displacement

In questo caso utilizzeremo tutte le armi a nostra disposizione

OPCODEB 0x002F9000 (%EBX, %EDI)

OPCODEB -0x9000 (%EBX, %EDI)

## 3.5.4 Indirizzamento delle porte di I/O

L'indirizzo di I/O può avvenire sia con la sorgente che col destinatario, ma mai con entrambi. L'indirizzamento può essere diretto o indiretto con registro puntatore

- Possibili porre solo indirizzi su 8bit nell'indirizzamento diretto (perchè la macchina fornisce solo 8bit)
- L'indirizzamento indiretto è possibile solo usando il registro DX

Vediamo i seguenti esempi

```
IN 0x001A, %AL  
IN (%DX), %AX  
OUT %AL, 0x003A  
OUT %AL, (%DX)
```

La prima istruzione preleva un operando a 8bit dalla porta di I/O 0x001A e pone il contenuto presente a quell'indirizzo nel registro AL, la terza pone il contenuto del registro AL nella porta di I/O avente indirizzo 0x003A.

## 3.6 Principali istruzioni

Si distinguono

- Istruzioni operative
  - Istruzioni di trasferimento
  - Istruzioni aritmetiche
  - Istruzioni di traslazione/rotazione
  - Istruzioni logiche
- Istruzioni di controllo
  - Istruzioni di salto
  - Istruzioni per la gestione di sottoprogrammi

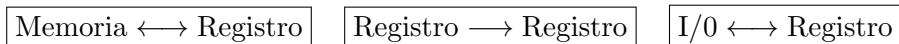
Per ogni istruzione diremo:

- il formato
- cosa fa
- se ci sono aggiornamenti di flag
- le modalità di indirizzamento ammesse per gli operandi

le istruzioni sono spiegate in modo dettagliato nella bibbia di Corsini.

### 3.6.1 Istruzioni di trasferimento

Ribadiamo che gli spostamenti possibili sono i seguenti



non è possibile fare trasferimenti da memoria a memoria. Le istruzioni di trasferimento, inoltre, non modificano i flag.

#### 3.6.1.1 MOVE

- **FORMATO:**      `MOV source, destination`
- **AZIONE:**    Sostituisce l'operando destinatario con una copia dell'operando sorgente
- **FLAG di cui viene modificato il contenuto:** Nessuno.

Operandi	Esempi
Memoria, Registro Generale	<code>MOV 0x00002000,%EDX</code>
Registro Generale, Memoria	<code>MOV %CL,0x12AB1024</code>
Registro Generale, Registro Generale	<code>MOV %AX,%DX</code>
Immediato, Memoria	<code>MOVB \$0x5B,(%EDI)</code>
Immediato, Registro Generale	<code>MOV \$0x54A3,%AX</code>

### 3.6.1.2 LOAD EFFECTIVE ADDRESS

- **FORMATO:** LEA source, destination
- **AZIONE:** Sostituisce l'operando destinatario con l'espressione indirizzo contenuta nell'operando sorgente.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

Operandi	Esempi
Memoria, Registro Generale a 32 bit	LEA 0x00002000, %EDX LEA 0x00213AB1(%EAX, %EBX, 4), %ECX

```
MOV 0x00213AB1 (%EAX, %EBX, 4), %ECX  
LEA 0x00213AB1 (%EAX, %EBX, 4), %ECX
```

Col primo esempio intendiamo *copia l'indirizzo 0x00002000 nel registro EDX*. col secondo intendiamo *copia il risultato dell'operazione di indirizzamento di memoria nel registro ECX*.

**Differenza tra MOV e LEA** La MOV copia il contenuto posto a quell'indirizzo, la LEA copia l'indirizzo!

### 3.6.1.3 EXCHANGE

- **FORMATO:** XCHG source, destination
- **AZIONE:** Sostituisce all'operando destinatario una copia dell'operando sorgente e viceversa.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

Operandi	Esempi
Memoria, Registro Generale	XCHG 0x00002000, %DX
Registro Generale, Memoria	XCHG %AL, 0x000A2003
Registro Generale, Registro Generale	XCHG %EAX, %EDX

- La XCHG **modifica il sorgente**. È l'unica istruzione che lo fa.
- Fatto curioso (per adesso): in Assembler si possono scambiare due operandi con una sola istruzione (**non trasparenza** dei registri)

# Capitolo 4

**Giovedì 01/10/2020**

## 4.1 Concludiamo con le istruzioni di trasferimento

### 4.1.1 INPUT e OUTPUT

- **FORMATO:**

```
IN indirizzo, %AL  
IN indirizzo, %AX  
IN (%DX), %AL  
IN (%DX), %AX
```

- **AZIONE:** Sostituisce il contenuto del registro destinatario (AL, AX) con il contenuto di un adeguato numero di porte consecutive.

L'indirizzo della prima (ed eventualmente unica) porta è specificato direttamente nell'istruzione (primi due formati) o è contenuto nel registro DX (ultimi due formati); i primi **due** formati possono essere utilizzati solo per individuare porte con indirizzo inferiore a 256.

- **FLAG** di cui viene modificato il contenuto: Nessuno.

- **FORMATO:**

```
OUT    %AL, indirizzo  
OUT    %AX, indirizzo  
OUT    %AL, (%DX)  
OUT    %AX, (%DX)
```

- **AZIONE:** Copia il contenuto del registro sorgente (AL, AX) in un adeguato numero di porte consecutive. L'indirizzo della prima (ed eventualmente unica porta) è specificato direttamente nell'istruzione (primi due formati) o è contenuto nel registro DX (ultimi due formati); i primi tre formati possono essere utilizzati solo per individuare porte con indirizzo inferiore a 256.

- **FLAG** di cui viene modificato il contenuto: Nessuno.

Gli operandi nell'I/O si possono trasferire solo nei/dai registri generali. Non si fanno operazioni sulle porte! Inoltre, gli unici registri utilizzabili sono AL, AX (come sorgente o destinatario) e DX (come registro puntatore).

#### 4.1.1.1 ASSEMBLER non è linguaggio ortogonale

Contrariamente al C++ Assembler non può essere definito un linguaggio ortogonale (in C++ posso mettere qualunque variabile se la sintassi mi consente di scrivere una variabile). Cose che sono fatte con un certo registro generale non è detto possano essere fatte con altri registri.

#### 4.1.1.2 Uscita da registro a porta

Supponiamo di voler passare il contenuto del registro BX (si possono effettuare trasferimenti solo dai registri AL e AX) alla porta con indirizzo 0x3142. Scriveremo le seguenti istruzioni

```
MOV %BX, %AX  
MOV $0x3142, %DX  
MOV %AX, (%DX)
```

ricordiamo che non è possibile fare trasferimenti da memoria a memoria. Inoltre, nei formati dove la sorgente (input) o il destinatario (output) sono indirizzi è necessario che l'indirizzo sia inferiore a 256.

#### 4.1.2 Pila

Abbiamo già visto la pila a Fondamenti di programmazione: una struttura dati dove il contenuto è gestito secondo la regola LIFO (*Last in First Out*, cioè l'ultimo elemento inserito è il primo ad andarsene). Al di là della questione C++, la pila è essenziale per il funzionamento del calcolatore, precisamente per annidare sottoprogrammi (L'assembler stesso è organizzato per sottoprogrammi)

- Quando avvio un sottoprogramma salvo l'indirizzo di ritorno, cioè quello dell'istruzione successiva e lo pongo nella pila (*push*)
- Quando termine l'esecuzione del sottoprogramma faccio *pop*, cioè estraggo dalla pila l'ultimo indirizzo inserito (quello della prossima istruzione da eseguire).

La pila permette di chiamare sottoprogrammi all'interno di altri sottoprogrammi.

**Puntatore al top della pila** Il registro ESP (*Extended stackpointer*) mi permette di puntare all'elemento più alto della pila.

**push value** Decremento ESP in modo tale che punti alla zona di memoria immediatamente superiore, copio un certo valore in %ESP

**push dest** Copio il contenuto di %ESP nel destinatario e incremento ESP in modo tale da puntare alla zona di memoria immediatamente inferiore.

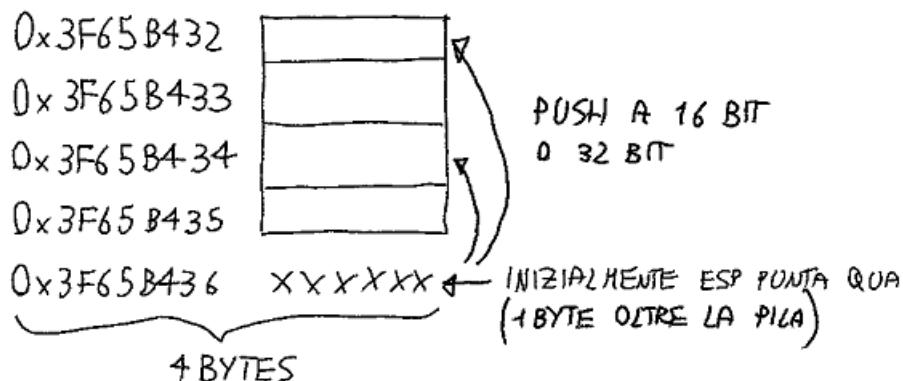
**ESP deve essere inizializzato** Ogni programma che utilizza la pila deve inizializzare ESP assegnando un valore sensato.

- **FORMATO:** PUSH source
- **AZIONE:** Salva nella pila corrente una copia dell'operando sorgente (che deve essere a 16 o a 32 bit). Più in dettaglio, compie le seguenti azioni: i) decrementa l'indirizzo contenuto nel registro ESP di due o di quattro; ii) memorizza una copia dell'operando sorgente nella doppia o quadrupla locazione il cui indirizzo è contenuto in ESP.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

Operandi	Esempi
Memoria	PUSHW 0x3214200A
Immediato	PUSHL \$0x4871A000
Registro Generale	PUSH %BX

- **FORMATO:** POP destination
- **AZIONE:** Rimuove dalla pila corrente una word o un long e la sostituisce all'operando destinatario. Più in dettaglio, compie le seguenti azioni: i) sostituisce all'operando destinatario una copia del contenuto della doppia o della quadrupla locazione il cui indirizzo è contenuto in ESP, ii) incrementa di due o di quattro l'indirizzo contenuto in ESP, rimuovendo in tal modo dalla pila la word o il long copiato.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

Operandi	Esempi
Memoria	POPW 0x02AB2000
Registro Generale	POP %BX



Gli operandi salvati ed ottenuti dalla pila devono essere a 16 o a 32 bit. Se non si ha un registro generale come operando è necessario indicare il suffisso di lunghezza (per forza W o L, come già detto).

#### 4.1.2.1 Pila come memoria temporanea

La pila può essere usata come *parcheggio di dati*. I registri generali sono pochi e in alcune istruzioni sono gli unici operandi possibili. In alcune circostanze, addirittura, che debba usare un certo registro per due scopi diversi in momenti diversi (e se volessi recuperare il dato trovato nel primo momento?). Supponiamo che mi serva un dato da una porta:

```

PUSH %EAX
IN 0x001A, %AL
...
POP %EAX

```

#### 4.1.3 PUSHAD e POPAD

**Significato** Per il prof A dovrebbe stare per ALL e *D* per double. Non è sicuro.

- **FORMATO:** PUSHAD
- **AZIONE:** Salva nella pila corrente una copia del contenuto degli 8 registri generali a 32 bit, rispettando il seguente ordine: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI.
- **FLAG** di cui viene modificato il contenuto: Nessuno.
  
- **FORMATO:** POPAD
- **AZIONE:** Rimuove dalla pila 8 long e con essi rinnova il contenuto degli 8 registri generali a 32 bit, rispettando il seguente ordine: EDI, ESI, EBP, ESP, EBX, EDX, ECX, EAX.
- **FLAG** di cui viene modificato il contenuto: Nessuno
  - Si noti che ESP **non viene sovrascritto** (sarebbe un problema se lo fosse).

Chiaramente l'ESP verrà modificato non perchè poniamo noi un valore, ma perchè eseguiamo l'operazione POP (quindi L'ESP sarà incrementato quanto necessario).

## 4.2 Istruzioni aritmetiche

Introduzione a pagina 13 sulle dispense di Assembler

### 4.2.1 ADD e SUBTRACT

Spiegazioni con algoritmo e flag coinvolte da pagina 13 a pagina 17 della dispensa di Assembler.

$$\boxed{\text{dest} + = \text{src}} \quad \boxed{\text{dest} - = \text{src}}$$

#### Osservazioni

- Gli algoritmi, sia per l'addizione che per la sottrazione, sono gli stessi sia nei naturali che negli interi (ovviamente negli interi solo con rappresentazione in complemento a 2).
- Le proprietà tipiche della rappresentazione in C2 (spiegate nella parte relativa all'aritmetica) chiariscono perchè si possa utilizzare la stessa circuiteria, quindi svolgere le operazioni su naturali e interi utilizzando la stessa istruzione.
- La differenza tra lo svolgere operazioni su interi o naturali sta nei flag da osservare: la circuiteria modifica, in qualunque caso, tutti i flag relativi. Solo NOI sappiamo se quel numero è naturale o intero, segue che siamo noi (con le operazioni successive) a scegliere quali flag leggere (e quindi se trattare il risultato come un intero o un naturale).

- Precisamente:
  - Nei naturali controlliamo i flag CF, SF e ZF. In particolare se  $CF = 1$  significa che l'ultimo riporto da una cifra in più al risultato finale, quindi siamo usciti dall'intervallo di rappresentazione.
  - Negli interi controlliamo i flag OF, SF e ZF. Il CF può avere valore uguale ad 1 come prima, ma non è più indicatore della validità del risultato. Andremo a vedere la Overflow flag, che
    - \* è uguale ad 1 se nell'addizione si è avuto riporto con operandi di segno concorde.
    - \* è uguale a 0 se gli operandi dell'addizione sono di segno discordi (il riporto si ignora)
    - \* è uguale ad 1 se nella sottrazione si è avuto prestito con operandi di segno discordi
    - \* è uguale a 0 se gli operandi della sottrazione sono di segno concorde (il prestito si ignora)
- In caso di Overflow negli interi il MSB indicherà un segno sbagliato.

Relativamente alla sottrazione se scriviamo l'operazione possiamo ricondurci ai casi dell'addizione (somma di elementi concordi, non sempre rappresentabile, o somma di elementi discordi, sempre rappresentabile).

#### 4.2.2 INCREMENT e DECREMENT

Queste due istruzioni esistono per ragioni soprattutto storiche: molti anni fa la circuiteria più lenta era quella che svolgeva somme e sottrazioni. Avere una funzione di incremento e decremento permetteva di rendere più veloci certe operazioni. Oggi non è più la stessa cosa parlare di costi di operazione: i processori, in molte situazioni, eseguono operazioni in parallelo.

- **FORMATO:** INC destination
- **AZIONE:** Equivale all'istruzione ADD \$1, destination, con la sola differenza che il contenuto del flag CF non viene modificato.
- **FLAG** di cui viene modificato il contenuto: OF, SF e ZF.

Operandi	Esempi
Memoria	INCB (%ESI)
Registro Generale	INC %CX

- **Più compatta**, in quanto nella ADD andrebbe comunque specificata una costante sul numero di bit del destinatario (8, 16, 32)
- Tanti (tanti) anni fa, era anche più veloce (vedremo perché)

- **FORMATO:** DEC destination
- **AZIONE:** Equivale all'istruzione SUB \$1, destination con la sola differenza che il contenuto del flag CF non viene modificato.
- **FLAG** di cui viene modificato il contenuto: OF, SF e ZF.

Operandi	Esempi
Memoria	DECB (%EDI)
Registro Generale	DEC %CX

#### 4.2.3 ADD WITH CARRY e SUBTRACT WITH BORROW

Informazioni presenti a pagina 65 e 66 della dispensa di Assembler.

- HO OPERANDI A 64 BIT
- LA ADD LAVORA FINO A 32 BIT
- DIVIDO GLI OPERANDI IN 2 PARTI

$$\begin{array}{r}
 \begin{array}{l} 1^{\circ} \text{ OPERANDO} \\ 2^{\circ} \text{ OPERANDO} \end{array} \quad \begin{array}{r}
 \begin{array}{r}
 \overline{5\ 6\ A\ 9\ C\ 2\ D\ 4} \\
 + \overline{4\ 4\ B\ 9\ A\ 5\ A\ 4} \\
 \hline 9\ B\ 6\ 3\ 6\ 8\ 7\ 9
 \end{array} \quad \left| \begin{array}{r}
 67\ A\ 4\ 3\ B\ 5\ F\ + \\
 A\ 6\ B\ 4\ C\ 5\ 5\ A\ = \\
 \hline 0\ E\ 5\ 9\ 0\ 9\ B\ 9
 \end{array} \right.
 \end{array}
 \end{array}$$

POI QUESTE DUE ↗ PRIMA SOMMA QUESTE DUE PARTI

AFFINCHÉ LA COSA ABBIA SENSO DEVO SOMMARE, NECCA SECONDA SOMMA, ANCHE IL RIPORTO. QUINDI:

- NELLA PRIMA SOMMA USO IL SOLITO ADD
- NELLÀ SECONDA USO ADC: SOMMA COME IN ADD, PIÙ IL RIPORTO FINALE DELLA PRECEDENTE SOMMA

#### 4.2.4 NEGATE

- **FORMATO:** NEG destination
- **AZIONE:** Interpreta l'operando destinatario come un numero intero e lo sostituisce con il suo opposto. Qualora l'operazione non sia possibile, mette ad 1 il contenuto del flag OF. Mette ad 1 il contenuto del flag CF eccetto quando l'operando è zero: in tal caso lo mette a 0.
- **FLAG** di cui viene modificato il contenuto: Tutti.

Operandi	Esempi
Memoria	NEGB (%EDI)
Registro Generale	NEG %CX

**Algoritmo** Con la NEG si ottiene l'opposto (ovviamente la cosa va fatta con un destinatario concepito come intero)

- Complementando bit a bit il destinatario e
- aggiungendo uno

**Attenzione all'overflow** Abbiamo che l'intervallo di rappresentabilità in C2, dato un numero  $N$  di bit, è il seguente

$$[-2^{N-1}; 2^{N-1} - 1]$$

non possiamo calcolare l'opposto dell'estremo negativo ( $2^{N-1}$  non appartiene all'intervallo)! In questo caso l'OF sarà uguale ad uno.

#### 4.2.5 COMPARE

- **FORMATO:** CMP source, destination
- **AZIONE:** Verifica se l'operando destinatario è maggiore, uguale o minore dell'operando sorgente, sia interpretando gli operandi come numeri naturali che come numeri interi. Aggiorna poi il contenuto dei flag tenendo conto del risultato della verifica (i due operandi rimangono inalterati). L'esatto algoritmo di aggiornamento del contenuto dei flag è noioso da descriversi: l'aggiornamento è comunque consistente con l'interpretazione che del contenuto dei flag sarà data dall'istruzione di salto condizionato (vedi avanti), che in un programma sensato segue sempre l'istruzione CMP.
- **FLAG** di cui viene modificato il contenuto: Tutti.

Operandi	Esempi
Memoria, Registro Generale	CMP 0x00002000, %EDX
Registro Generale, Memoria	CMP %CL, 0x12AB1024
Registro Generale, Registro Generale	CMP %AX, %DX
Immediato, Memoria	CMPB \$0x5B, (%EDI)
Immediato, Registro Generale	CMP \$0x45AB54A3, %EAX

75

Presente a pagina 17 della dispensa di Assembler l'algoritmo. Si considera che

- Con ZF = 1 si ha destination = source
- Con SF = 1 si ha risultato della sottrazione negativa, quindi destination < source

- Con SF = 0 si ha risultato della sottrazione positiva, quindi destination > source

Noi non controlleremo direttamente i flag ma utilizzeremo delle JUMP CONDIZIONATE. Nella dispensa e nel libro di Corsini sono presenti tutti i JUMP possibili.

#### 4.2.6 INTEGER e INTEGER MULTIPLY

Questa istruzione è concepita diversamente dalle altre operazioni:

- Il risultato di una somma sta su  $N$  o  $N + 1$  cifre
- Il prodotto di due numeri a  $N$  cifre sta su  $2N$  cifre. Questo dettaglio pone un problema non secondario: contrariamente alla somma non posso comparare fattori e risultato. Segue che un operando non potrà essere utilizzato sia come fattore che come risultato.

Potrei pensare a un'istruzione a tre operandi, ma cose del genere in assembler non esistono. Si risolve specificando un solo operando (l'operando sorgente): l'altro operando e la sede del risultato sono impliciti. Si osserva che i risultati delle moltiplicazioni, in 16 e 32bit, vengono divisi tra due registri. In 32bit la cosa è necessaria, ma in 16bit a cosa serve? Non potrei fare...?

**EAX=AX\* source**

Anche in questo caso le motivazioni sono storiche: i registri precedentemente erano a 16bit, e il ragionamento che si faceva era lo stesso fatto con gli operandi a 32bit. Con l'estensione dei registri si è mantenuta la cosa per compatibilità. Un trucco per porre il risultato in un registro a 32bit è il seguente

```
PUSH %DX
PUSH %AX
POP %EAX
```

Pongo nella pila prima la parte alta (cifre più significative), poi la parte bassa (cifre meno significative), poi le estraggo dalla pila insieme (nei comandi push abbiamo indicato due registri a 16bit, col comando pop indichiamo un registro a 32bit).

- **FORMATO:** MUL source
- **AZIONE:** Considera l'operando sorgente come un moltiplicando e l'operando destinatario (implicito) come un moltiplicatore ed effettua l'operazione di moltiplicazione, interpretando gli operandi come numeri naturali.

Se l'operando sorgente (moltiplicando) è:

- 1) ad 8 bit, allora carica in AX il prodotto di AL e source
- 2) a 16 bit, allora carica in DX\_AX il prodotto di AX e source
- 3) a 32 bit, allora carica in EDX\_EAX il prodotto di EAX e source

- **FLAG** di cui viene modificato il contenuto: CF, OF: vengono messi a 1 se il risultato non sta sul numero di bit di source, e a zero altrimenti. SF e ZF sono **indefiniti**.

Operandi	Esempi
Memoria	MULB (%ESI)
Registro Generale	MUL %ECX

- **FORMATO:** IMUL source
- **AZIONE:** Considera l'operando sorgente come un moltiplicando e l'operando destinatario (implicito) come un moltiplicatore ed effettua l'operazione di moltiplicazione, interpretando gli operandi come numeri intei.
- Se l'operando sorgente (moltiplicando) è:
  - 1) ad 8 bit, allora carica in AX il prodotto di AL e source
  - 2) a 16 bit, allora carica in DX\_AX il prodotto di AX e source
  - 3) a 32 bit, allora carica in EDX\_EAX il prodotto di EAX e source
- **FLAG** di cui viene modificato il contenuto: CF, OF, SF (non attendibile) e ZF (non attendibile).

Operandi	Esempi
Memoria	IMULB (%ESI)
Registro Generale	IMUL %ECX

### Quindi

- Uno dei due operandi è un registro, l'altro è indicato nell'istruzione (source)
- L'operando source determina il numero di bit che andremo ad utilizzare per il risultato, quindi i registri dove saranno salvati i risultati.
- Se source è a 16 o 32 bit i risultati saranno divisi tra due registri: con 32bit è necessario, in 16bit si ha questa cosa per ragioni storiche.

**Non attendibile?** Molto difficile da capire cosa succeda coi flag. Inutile guardare, non ci dovrebbero servire (cit.Stea)

### 4.2.7 DIVIDE e INTEGER DIVIDE

La divisione presenta qualche problemino ulteriore rispetto alla moltiplicazione:

- I risultati sono due: quoziente e resto
- l'operazione non è fattibile se il divisore vale zero.

Dato un dividendo  $X$  e un divisore  $Y$  individuiamo che  $0 \leq R \leq Y - 1$  (può essere grande quanto il divisore) e  $0 \leq Q \leq X$  (può essere grande quanto il dividendo).

**Istruzione** Il risultato è diviso tra due registri in tutte le versioni possibili. Ricordiamo che dobbiamo salvare quoziente e divisore.

**Osservazione** Risulta ovvio che la dimensione del divisore (source) risulta inferiore al dividendo.

- Se il divisore è di 8bit il dividendo sarà di 16bit
- Se il divisore è di 16bit il dividendo sarà di 32bit
- Se il divisore è di 32bit il dividendo sarà di 64bit

negli ultimi due casi il dividendo sarà preso dai due registri divisi.

**Wait** Ma non si era detto che il quoziente può stare al massimo nel numero di bit del dividendo? Cosa succede se il quoziente non è rappresentabile sul registro designato? Se il quoziente della divisione non sta sul numero di bit previsto dal formato viene sollevata un'eccezione (la stessa eccezione che partirebbe con una divisione per 0 - il programma si inchioda). Per evitare problemi del genere dobbiamo scegliere un'adeguata versione della divisione

**Esempio di divisione** Supponiamo di voler fare 15.000 per 3 (abbiamo un dividendo di 16bit e un divisore di 8bit). Se noi dividiamo otteniamo come risultato 5000. Questo è problematico se adottiamo la divisione con source ad 8bit, segue che le seguenti istruzioni non vanno bene!

```
MOX $3, %CL
MOV $15000, %AX
DIV %CL
```

La cosa può essere risolta così:

```
MOX $3, %CX
MOV $15000, %AX
MOV $0, %DX <-----
DIV %CL
```

Abbiamo modificato il contenuto dei registri *AX* e *DX*: questo è sufficiente per scegliere la divisione con source a 16bit. La terza istruzione è vitale, e molto spesso viene dimenticata (cit.)

- **FORMATO:** DIV source
- **AZIONE:** Considera l'operando sorgente come un divisore e l'operando destinatario (implicito) come un dividendo ed effettua l'operazione di divisione, interpretando gli operandi come numeri naturali. Più in dettaglio, compie le azioni che seguono.
  - i) Se l'operando sorgente (divisore) è:
    - 1) ad 8 bit, allora divide AX per il sorgente, mettendo il quoziente in AL ed il resto in AH
    - 2) a 16 bit, allora divide DX\_AX per il sorgente, mettendo il quoziente in AX ed il resto in DX
    - 3) a 32 bit, allora divide EDX\_EAX per il sorgente, mettendo il quoziente in EAX ed il resto in EDX
  - ii) Se il quoziente non è esprimibile su un numero di bit pari a quello del divisore, allora genera un'*interruzione interna* in conseguenza della quale viene messo in esecuzione un opportuno sottoprogramma. Il contenuto dei flag e dei registri destinati a contenere il quoziente ed il resto è, in tal caso, non significativo.
- **FLAG** di cui viene modificato il contenuto: Tutti, ma in modo non attendibile.

Operandi	Esempi
Memoria	DIVB (%ESI)
Registro Generale	DIV %ECX

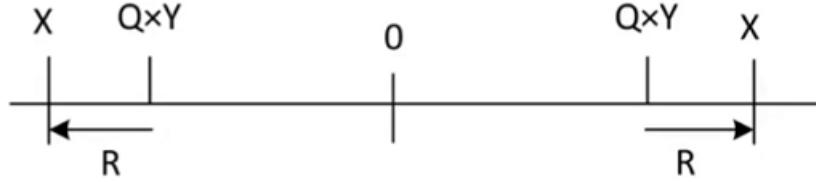
- **FORMATO:** IDIV source
- **AZIONE:** Considera l'operando sorgente come un divisore e l'operando destinatario (implicito) come un dividendo ed effettua l'operazione di divisione, interpretando gli operandi come numeri intei [...]
- **FLAG** di cui viene modificato il contenuto: Tutti, ma in modo non attendibile.

Operandi	Esempi
Memoria	IDIVB (%ESI)
Registro Generale	IDIV %ECX

- Speculare alla precedente, lavora su operandi intei

## Divisione intera

- Nella divisione intera il resto ha sempre il segno del dividendo e in modulo è minore rispetto a quello del divisore.



```
-7 idiv 3 : quoziante -2, resto -1
7 idiv -3: quoziante -2, resto +1
```

- Ciò significa che il quoziante viene approssimato per troncamento, cioè viene sempre approssimato all'intero più vicino allo zero. In entrambi gli esempi ottengo che  $Q \times Y$ , cioè il prodotto tra quoziante e divisore, è più vicino allo zero rispetto al dividendo.
- La cosa è inconsistente con le nozioni di algebra che conosciamo: per capire la differenza pensiamo alla definizione di resto.

$$|-4|_3 = |3 \cdot (-2) + 2|_3 = |2|_3 = 2 \quad |4|_3 = 1$$

### 4.2.8 Conclusioni

- Scegliere con cura la versione della divisione (basandoci sulle ipotesi in mano nostra)
- I registri devono essere azzerati prima del calcolo, altrimenti i risultati potrebbero essere inconsistenti (chi ce lo dice che il registro con le cifre più significative presenti bit nulli?)
- Ricordarsi che il contenuto di DX o EDX viene modificato dalle operazioni

Per la moltiplicazione:

- Se l'unico operando esplicito (un fattore) è a 8bit allora il risultato della moltiplicazione (l'altro fattore è un registro a 8bit) sarà posto in un registro a 16bit.
- Se l'unico operando esplicito (un fattore) è a 16bit allora il risultato della moltiplicazione (l'altro fattore è un registro a 16bit) sarà diviso in due registri a 16 bit (abbiamo quindi un risultato a 32bit).
- Se l'unico operando esplicito (un fattore) è a 32bit allora il risultato della moltiplicazione (l'altro fattore è un registro a 32bit) sarà diviso in due registri a 32 bit (abbiamo quindi un risultato a 64bit).

Per la divisione:

- Se l'unico operando esplicito (divisore) è a 8bit allora quoziante e resto saranno memorizzati su registri a 8bit (il dividendo è a 16bit).
- Se l'unico operando esplicito (divisore) è a 16bit allora quoziante e resto saranno memorizzati su registri a 16bit (il dividendo è a 32 bit, diviso su due registri).
- Se l'unico operando esplicito (divisore) è a 32bit allora quoziante e resto saranno memorizzati su registri a 32bit (il dividendo è a 64bit, diviso su due registri).

# Capitolo 5

Venerdì 02/10/2020

## 5.1 Continuiamo con le istruzioni aritmetiche

### 5.1.1 Estensione di campo

Con estensione di campo intendiamo un'operazione con cui si rappresenta un numero usando più cifre.

### 5.1.2 Naturali

Nei numeri naturali l'operazione è banale, mi basta aggiungere gli zeri a sinistra. Segue una cosa del seguente tipo

$$100110 \longrightarrow \boxed{0}100110$$

### 5.1.3 Interi

Nei numeri interi non possiamo fare la stessa cosa: il MSB indica il segno, quindi non posso aggiungere zeri (andrei ad alterare il segno del numero). Il metodo adottato consiste nel ripetere il bit più significativo

$$100110 \longrightarrow \boxed{1}100110$$

Il motivo di questo metodo è chiaro quando passiamo da rappresentazione in base 2 di un intero negativo alla base 10.

### 5.1.4 CONVERT WORD TO DOUBLEWORD in EAX

- **FORMATO:** CWDE
- **AZIONE:** Interpreta il contenuto di AX come un numero intero a 16 bit, rappresenta tale numero su 32 bit e quindi lo memorizza in EAX.
- **FLAG** di cui viene modificato il contenuto: Nessuno
  
- **Esempio:** voglio sommare due interi, uno si trova in AX ed uno in EBX.

```
MOV $-5, %AX
MOV $100000, %EBX
CWDE
ADD %EAX, %EBX
```

## 5.2 Istruzioni di traslazione

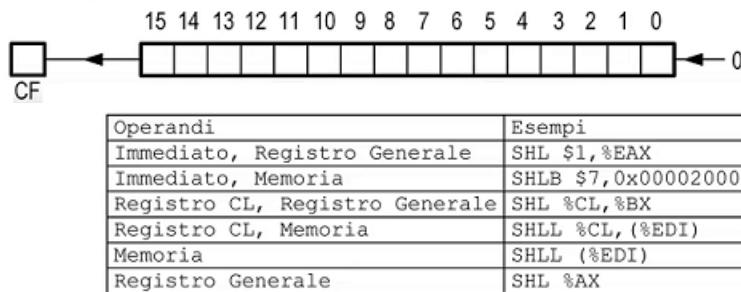
Le istruzioni di traslazione permettono la variazione dell'ordine di bit in un operando destinatario. Solitamente si hanno due formati

```
OPCODE src, dest
OPCODE dest
```

col primo formato indichiamo, attraverso src (immediato o registro CL, di cui si considerano solo i 5bit più bassi), quante volte vogliamo ripetere l'operazione; col secondo ci limitiamo ad eseguire l'operazione una sola volta (è come se ponessi src = 1). Ovviamente src deve valere al più 31: porre un src > 31 ha poco senso.

### 5.2.1 SHIFT LOGICAL LEFT

- **FORMATO:**
  - SHL source, destination
  - SHL destination
- **AZIONE:** Interpreta l'operando sorgente come un numero naturale  $n$  e, per  $n$  volte, sostituisce il bit contenuto in CF e ciascun bit dell'operando destinatario con il bit che gli è immediatamente a destra, considerando il bit più significativo dell'operando destinatario immediatamente a destra del bit contenuto in CF e sostituendo con 0 il bit meno significativo dell'operando destinatario. Pone infine a 0 il contenuto del flag OF.
- **FLAG di cui viene modificato il contenuto:** Tutti.



- Lo shift sinistro  $n$  volte equivale a **moltiplicare il destinatario per  $2^n$** 
  - Vale in base 2 come in base 10
- Non è segno di grande furbizia usare una MUL quando si può usare una SHL
  - SHL è più veloce, e per un programmatore è estremamente più semplice
- CF viene sovrascritto  $n$  volte (difficile capire se il risultato è corretto)

**Attenzione** In alcuni casi lo shift può essere utile sul piano dell'efficienza (occupa meno spazio con l'operazione ed eseguo un'operazione più veloce). In altri casi può essere una condanna a morte: quando si utilizza questa istruzione ci si sbarazza ogni volta di un bit, quindi si perde informazione. Segue che una moltiplicazione per  $2^3 = 8$  sia possibile con lo shift left solo se il risultato non esce dal numero di bit a disposizione.

### 5.2.2 SHIFT ARITHMETIC LEFT

- **FORMATO:** SAL source, destination  
SAL destination

**AZIONE:** la stessa cosa della SHL

- Sulla dispensa sono riportate alcune differenze minori, ma è un errore (SHL e SAL hanno lo stesso OPCODE sul manuale Intel)
- La SAL moltiplica per  $2^{src}$  un operando **intero**
  - Se MSB cambia valore almeno una volta, OF=1 (risultato non attendibile)

### 5.2.3 SHIFT LOGICAL RIGHT

- **FORMATO:** SHR source, destination  
SHR destination

- **AZIONE:** Interpreta l'operando sorgente come un numero naturale  $n$  e, per  $n$  volte, sostituisce il bit contenuto in CF e ciascun bit dell'operando destinatario con il bit che gli è immediatamente a sinistra, considerando il bit meno significativo dell'operando destinatario immediatamente a sinistra del bit contenuto in CF e sostituendo con 0 il bit più significativo dell'operando destinatario. Pone infine a 0 il contenuto del flag OF. Il numero naturale  $n$  non deve superare 31. Se l'operando sorgente non è presente, compie l'operazione una sola volta.

- **FLAG** di cui viene modificato il contenuto: Tutti.

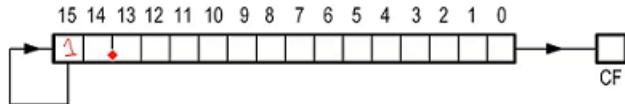


- SHR src, %EAX
  - Corrisponde a dividere EAX per  $2^{src}$ , approssimando il quoziente per difetto
  - Purché EAX sia interpretato come **operando naturale** (vengono inseriti zeri in testa)

### 5.2.4 SHIFT ARITHMETIC RIGHT

Non è possibile applicare lo stesso algoritmo agli interi: abbiamo già detto che negli interi il MSB rappresenta il segno del numero, segue che non posso fare entrare zeri a sinistra come in SHIFT LOGICAL RIGHT

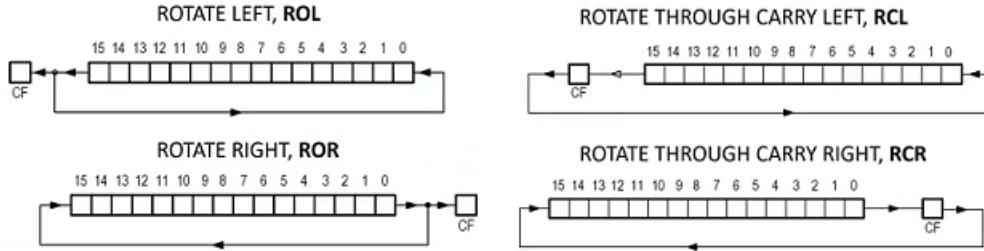
- **FORMATO:** SAR source, destination
- SAR destination
- **AZIONE:** [...].
- Guardando all'operando destinatario come ad un numero intero, questa istruzione divide tale numero per  $2^n$  e lo sostituisce con il quoziente (approssimato per difetto).



**Attenzione al quoziente** SAR e IDIV restituiscono lo stesso quoziente quando il dividendo è positivo oppure quando la divisione da resto nullo. Casi diversi sono dovuti al fatto che la SAR approssima sempre a sinistra, mentre la IDIV approssima per troncamento (verso lo 0).

### 5.3 Istruzioni di rotazione

Le istruzioni di rotazione ruotano i bit (verso destra o verso sinistra) coinvolgendo il CF. Hanno tutte lo stesso formato (per questo segue l'introduzione della sola ROTATE LEFT). Si osserva che nelle ROTATE THROUGH CARRY sposteremo la cifra più significativa (o quella meno significativa) nel CF, ponendo come cifra meno significativa (o più significativa) il contenuto del CF presente prima della modifica.



#### 5.3.1 ROTATE LEFT

- **FORMATO:** ROL source, destination  
ROL destination
- **AZIONE:** Interpreta l'operando sorgente come un numero naturale  $n$  e, per  $n$  volte, [...]. Il numero naturale  $n$  non deve superare 31. Se l'operando sorgente non è presente, compie l'operazione una sola volta.
- **FLAG di cui viene modificato il contenuto:** CF, OF.

Operandi	Esempi
Immediato, Registro Generale	ROL \$1,%EAX
Immediato, Memoria	ROLB \$7,0x00002000
Registro CL, Registro Generale	ROL %CL,%BX
Registro CL, Memoria	ROLL %CL,%EDI
Memoria	ROLL (%EDI)
Registro Generale	ROL %AX

## 5.4 Istruzioni logiche

Le istruzioni logiche permettono l'applicazione degli operatori dell'*Algebra di Boole*. Bisogna tenere conto di eventuali flag modificati (generalmente vengono modificati). Le seguenti operazioni applicano BIT a BIT le operazioni appena dette:

### 5.4.1 NOT

- **FORMATO:** NOT destination
- **AZIONE:** Modifica l'operando destinatario applicando a ciascuno dei suoi bit l'operazione logica *not*.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

Operandi	Esempi
Memoria	NOTL (%ESI)
Registro Generale	NOT %CX

### 5.4.2 AND

- **FORMATO:** AND source, destination
- **AZIONE:** Sostituisce ciascun bit dell'operando destinatario con il risultato dell'operazione logica *and* tra il bit stesso ed il corrispondente bit dell'operando sorgente; mette a 0 il contenuto dei flag CF ed OF.
- **FLAG** di cui viene modificato il contenuto: Tutti.

Operandi	Esempi
Memoria, Registro Generale	AND 0x00002000,%EDX
Registro Generale, Memoria	AND %CL,0x12AB1024
Registro Generale, Registro Generale	AND %AX,%DX
Immediato, Memoria	ANDB \$x5B,(%EDI)
Immediato, Registro Generale	AND \$0x45AB54A3,%EAX

### 5.4.3 OR

- **FORMATO:** OR source, destination
- **AZIONE:** Sostituisce ciascun bit dell'operando destinatario con il risultato dell'operazione logica *or* tra il bit stesso ed il corrispondente bit dell'operando sorgente; mette a 0 il contenuto dei flag CF ed OF.
- **FLAG** di cui viene modificato il contenuto: Tutti.

Operandi	Esempi
Memoria, Registro Generale	OR 0x00002000,%EDX
Registro Generale, Memoria	OR %CL,0x12AB1024
Registro Generale, Registro Generale	OR %AX,%DX
Immediato, Memoria	ORB \$0x5B,(%EDI)
Immediato, Registro Generale	OR \$0x45AB54A3,%EAX

#### 5.4.4 XOR (OR esclusivo)

- **FORMATO:** XOR source, destination
- **AZIONE:** Sostituisce ciascun bit dell'operando destinatario con il risultato dell'operazione logica *xor* tra il bit stesso ed il corrispondente bit dell'operando sorgente; mette a 0 il contenuto dei flag CF ed OF.
- **FLAG** di cui viene modificato il contenuto: Tutti.

Operandi	Esempi
Memoria, Registro Generale	XOR 0x00002000,%EDX
Registro Generale, Memoria	XOR %CL,0x12AB1024
Registro Generale, Registro Generale	XOR %AX,%DX
Immediato, Memoria	XORB \$0x5B,(%EDI)
Immediato, Registro Generale	XOR \$0x45AB54A3,%EAX

#### 5.4.5 Utilizzi di questi operatori

- Singoli bit di un operando possono essere resettati usando l'operatore AND (usando una maschera formata da soli 1 tranne nel bit che si vuole resettare).
- Singoli bit di un operando possono essere settati usando l'operatore OR (usando una maschera formata da soli 0 tranne nel bit che si vuole settare)
- Singoli bit di un operando possono essere invertiti rispetto all'attuale valore usando l'operatore XOR (usando una maschera formata da soli 0 tranne nel bit che si vuole invertire). Ricordiamo che

$$\alpha \text{ XOR } 0 = \alpha \quad \alpha \text{ XOR } 1 = \bar{\alpha}$$

- **Applicazioni pratiche:**

- L'operatore AND può essere usato per estendere operandi naturali. Supponiamo di voler sommare due numeri naturali: uno in AL e un altro in EBX.

```
MOV $5, %AL
MOV $100000, %EBX
AND $000000FF, %EAX
ADD %EAX, %EBX
```

- XOR può essere usato per resettare un registro: uso la XOR (ponendo EAX sia come source che come dest) invece della MOV.

```
XOR %EAX, %EAX
```

Nel confronto bit a bit ho sempre due bit uguali, quindi ottengo 0 ogni volta! Ulteriore nota: la MOV, come istruzione, occupa maggiore spazio.

## 5.5 Istruzioni di controllo

Il flusso del programma normalmente è sequenziale: le istruzioni stanno in memoria consecutive e il processore, normalmente, incrementa EIP puntano l'area di memoria immediatamente successiva. Questa regola non vale più quando intervengono istruzioni di controllo: queste scrivono un nuovo valore in EIP alterando il normale flusso. Abbiamo due tipi di istruzioni:

- Istruzioni di salto (Jmp, Jcon). Nel salto non si memorizzano informazioni relative al punto da cui si è compiuto il salto (neanche implicitamente).
- Istruzioni per la gestione di sottoprogrammi (CALL, RET). Si memorizzano informazioni relative al salto (attraverso la pila)

### 5.5.1 JUMP

- **FORMATO:**  $\text{JMP } \%EIP \pm \text{displacement}$   
 $\text{JMP } *\text{extended\_register}$   
 $\text{JMP } *\text{memory}$
- **AZIONE:** Calcola un indirizzo di salto e lo immette nel registro EIP.
- **FLAG** di cui viene modificato il contenuto: Nessuno.
- In Assembler si usa **un nome simbolico** per indicare l'istruzione dove si deve saltare, quindi non è necessario aver presente il formato di indirizzamento (ci pensa l'assemblatore a tradurre in uno dei formati di sopra, in genere il primo)

### 5.5.2 JUMP if condition met

- **FORMATO:**  $\text{J}\cancel{\text{con}} \ %EIP \pm \text{displacement}$
- **AZIONE:** Esamina il contenuto dei flag. Se da questo esame risulta che la condizione *con* è soddisfatta, allora si comporta come l'istruzione  $\text{JMP } \%EIP \pm \text{displacement}$ ; in caso contrario termina senza compiere alcuna azione.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

Tabella completa con le condizioni a pagina 22 del libro di Corsini.

#### Cose di cui tenere conto

- Le JUMP condizionate si basano sui valori di certi flag, quindi vengono eseguite a seguito di CMP o operazioni aritmetiche.
- Il soggetto è sempre l'operando destinatario: se io dico JA significa che voglio verificare se l'operando destinatario è maggior dell'operando sorgente (naturali).
- Si distinguono:

- JUMP condizionate utilizzabili in qualunque caso (CMP, operazioni aritmetiche...);
- JUMP condizionate utilizzabili solo a seguito di operazioni CMP, precisamente...
  - \* JUMP per confronti tra numeri naturali (*Above* e *Below*), e
  - \* JUMP per confronti tra numeri interi (*Greater* e *Less*).

### 5.5.3 Sottoprogrammi

Le istruzioni coinvolte sono due:

- CALL, salto ad un sottoprogramma
- RET, ritorno al programma chiamante

entrambe fanno riferimento, come già detto, alla pila.

#### 5.5.3.1 CALL

- **FORMATO:**     CALL %EIP  $\pm$  \$displacement  
                       CALL \*extended\_register  
                       CALL \*memory
- **AZIONE:** Effettua la *chiamata* di un *sottoprogramma*. Più precisamente, salva nella pila corrente il contenuto del registro EIP e poi modifica il contenuto di tale registro in modo del tutto simile a come fa l'istruzione JMP.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

#### 5.5.3.2 RET

- **FORMATO:**     RET
- **AZIONE:** Effettua il *ritorno* da un sottoprogramma. Più precisamente, rimuove un long dalla pila e con esso rinnova il contenuto di EIP.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

#### 5.5.3.3 NO OPERATION

- **FORMATO:**     NOP
- **AZIONE:** Termina senza compiere alcuna azione.
- **FLAG** di cui viene modificato il contenuto: Nessuno.
- Serve a perdere tempo

#### 5.5.3.4 HALT

- **FORMATO:** HLT
- **AZIONE:** Provoca la cessazione di ogni attività del processore.

Il processore alterna la fase di fetch (in cui preleva un'istruzione) con quella di esecuzione dell'istruzione che ha prelevato. Quando preleva un'istruzione HLT, smette di fare qualunque cosa, e si pone in una condizione dalla quale si può uscire solo resettando la macchina.

## 5.6 Protezione ed istruzioni privilegiate

- Il processore può funzionare in due modalità: utente e sistema.
- La modalità sistema permette l'esecuzione di tutte le istruzioni, la modalità utente permette l'esecuzione di una sola parte di queste istruzioni.
- Delle operazioni viste le seguenti sono istruzioni privilegiate non eseguibili in modalità utente: HLT, IN, OUT. Se chiamate va in esecuzione un'eccezione di protezione (comportamento diverso da sistema a sistema).

**Relativamente all'I/O** Faremo IN e OUT attraverso sottoprogrammi di servizio (li vedremo più avanti). IN e OUT sono istruzioni privilegiate poiché è facile portarle in stato inconsistente.

# Capitolo 6

Martedì 06/10/2020

## 6.1 Assemblatore

Utilizzeremo l'assemblatore **GAS** (*GNU Assembler*): le informazioni per scaricarlo sono presenti sul sito del docente. Si raccomanda massimo rispetto della procedura di installazione.

## 6.2 Struttura di un programma assembler

Un programma Assembler si articola in due sezioni:

- Sezione dati. Contiene le dichiarazioni delle variabili, cioè nomi simbolici per indirizzi di memoria.
- Sezione codice. Istruzioni del programma

**Direttive e istruzioni** In un programma sono presenti **direttive** (per esempio le dichiarazioni di variabili o di costanti) e **istruzioni** (ciò di cui abbiamo parlato fin dalla prima lezione). Ciascuna istruzione e direttiva è contenuta in una riga: alla fine è sempre presente il ritorno carrello *CR* (anche nell'ultima riga di tutto il codice). In un codice ASS individueremo, in conclusione:

- Una prima parte contenente le direttive

```
.GLOBAL _main  
.DATA  
....
```

la prima istruzione ci permette di stabilire quale sottoprogramma eseguire quando viene avviato l'applicativo. Nell'area .DATA indicheremo, come già detto, le variabili da utilizzare.

- Una parte contenente le istruzioni

```
.TEXT  
_main : NOP
```

```

...
RET
```

la RET deve essere posta in fondo ad ogni sottoprogramma, incluso quello principale. Relativamente alla NOP capiremo più avanti l'utilità nel porla all'inizio del sottoprogramma.

**Ordine degli elementi** L'ordine degli elementi descritto non è l'unico possibile: quanto detto è consigliato per ottenere un programma leggibile.

### 6.2.1 Assembler case-sensitive o case-insensitive?

Assembler è:

- **case-insensitive** relativamente ai caratteri delle keyword. Si consiglia di porle in stampatello maiuscolo
- **case-sensitive** relativamente alle etichette (nomi di variabili e nomi di sottoprogrammi). Significa che

$$\_main \neq \_Main$$

### 6.2.2 Cosa cambia rispetto agli esempi di programmi visti in passato?

<pre> # conteggio bit a 1 in un long  .GLOBAL _main .DATA dato:    .LONG 0x0F0F0101 conteggio: .BYTE 0x00  .TEXT _main:    NOP           MOVB \$0x00, %CL           MOVL dato, %EAX comp:     CMPL \$0x00, %EAX           JE fine           SHRL %EAX           ADCB \$0x00, %CL           JMP comp fine:     MOVB %CL, conteggio           RET</pre>	<pre> 0x00000100 0x00000101 0x00000102 0x00000103 0x00000104  ... 0x00000200  MOVB \$0x00, %CL 0x00000202  MOVL 0x00000100, %EAX 0x00000207  CMPL \$0x00000000, %EAX 0x0000020A  JE %EIP+\$0x07 0x0000020C  SHRL %EAX 0x0000020E  ADCB \$0x00, %CL 0x00000211  JMP %EIP-\$0x0C 0x00000213  MOVB %CL, 0x00000104 0x00000218  ...  </pre>
---	---

Il codice risulta decisamente più ordinato ma soprattutto andrà a richiamare aree di memoria usando nomi simbolici (e non più solo displacement). Queste etichette, che poniamo nella parte DATA, rappresentano simbolicamente l'indirizzo di quell'area di memoria. La cosa ci semplifica la vita non poco: le cose viste nelle lezioni precedenti saranno gestione dell'assemblatore.

#### 6.2.2.1 Attenzione al jump : confronto col C++

Si individua nel codice richiami ad etichette dichiarate più avanti: questa cosa è impensabile in C++ ma possibile in Assembler. Questo perchè l'assemblatore

1. Controlla tutti i nomi
2. traduce

l'approccio è necessario per poter svolgere salti in avanti, ma può essere anche mortale. Assembler, a differenze del C++, offre maggiori spazi di libertà: dobbiamo usare questi spazi con estrema cautela. Il suggerimento, in generale, è di continuare a rispettare la prassi del C++ (unica eccezione i salti in avanti).

**Altro esempio di porcheria** La libertà fornita da Assembler potrebbe portarci a scrivere quanto segue

```
nome1:  
nome2: CMP  
...  
...  
JMP nome1  
JMP nome2
```

queste cose, pur essendo valide (di fatto abbiamo creato due alias per un sottoprogramma), sono porcherie da evitare.

### 6.2.3 Riga di codice

Le righe di codice presentano la seguente struttura

```
nome: KEYWORD operandi #commento [/CR]
```

l'unico elemento obbligatorio in una riga è il carattere di ritorno carrello.

## 6.3 Direttive

**Attenzione** Ricordarselo come l'ave maria

Direttive  $\neq$  Istruzioni

Le direttive presentano la seguente struttura

```
.KEYWORD operandi [\CR]
```

### 6.3.1 Dichiarazione di variabili

Abbiamo già detto che dichiareremo variabili all'inizio del nostro programma. La cosa è utile non tanto per dire ad assembler su cosa stiamo lavorando ma per stabilire in modo agile quanta memoria ci serve. I tipi di variabili possibili sono:

- .WORD (2byte, 16bit)
- .BYTE (1byte, 8bit)
- .LONG (4byte, 32bit)

```

.WORD _____ # scalare, 2 byte, valore 0x0000
.BYTE 0x30    # scalare, 1 byte, valore 0x30
.BYTE 0x30, 0x31 # vett., 2 componenti da 1 byte
.WORD 0x1020, 0x32AB# vett., 2 componenti da 2 byte
.LONG var3+2   # scalare, 4 byte

```

Individuiamo che

- è ammesso dichiarare variabili senza inizializzarle. Formalmente il valore inizializzato dovrebbe essere quello indicato in var0, ma la cosa non è certa e il comportamento della macchina imprevedibile. Si consiglia di inizializzare in qualunque caso
- possiamo dichiarare variabili scalari, ma anche variabili vettoriali (attraverso una lista)
- Possiamo fare riferimento ad elementi di un vettore attraverso la seguente sintassi

`nome_variabile_vettoriale+K`

dove  $K$  consiste nel numero di byte. Se abbiamo un vettore di tipo WORD porremo  $K = 2$  per passare dal primo al secondo elemento.

#### 6.3.1.1 Alternativa per dichiarare vettori (comando FILL)

Assembler offre la direttiva FILL per dichiarare vettori. Abbiamo la seguente sintassi

`.FILL numero, dim, espressione`

- **numero** si utilizza per indicare il numero di elementi del vettore
- **dim** si utilizza per indicare il tipo di vettore. Può assumere solo i valori 1, 2, 4 (per indicare, rispettivamente, byte, word e long)
- **espressione** si utilizza per indicare il valore con cui inizializzare ogni elemento del vettore. Se omesso è uguale a zero (in questo caso possiamo fidarci senza problemi). Segue che FILL è utile quando non vogliamo inizializzare una variabile.

Abbiamo utilizzato questa cosa per salvare stringhe da porre nel buffer: avremo come numero 80, come dim 1 (codifica ASCII, con 7bit posso codificare tutte le lettere necessarie per una stringa) e come espressione un valore iniziale. Per porre una stringa all'interno di questa area di memoria si utilizza la LEA e la funzione inline!

#### 6.3.1.2 Codifica ASCII e caratteri speciali

Anche in Assembler possiamo immaginare il salvataggio di stringhe come array, precisamente un array di byte. Possiamo porre, tra singoli apici, i caratteri, o direttamente la codifica ASCII

```

var5: .BYTE 'C','i','a','o'
var5: .BYTE 0x43, 0x69, 0x61, 0x6F

```

Possibile inizializzare vettori di BYTE anche attraverso l'istruzione ASCII e ASCIZ

```
var6: .ASCII "messaggio"  
var7: .ASCIZ "messaggio"
```

con la prima istruzione creiamo un vettore composto da 9 elementi (ciascuno 1byte), mentre con la seconda creiamo un vettore composto da 9 elementi più uno aggiuntivo (backslash 0, NUL).

**Caratteri speciali** Caratteri speciali come ritorno carrello, tabulazione, etc... possono essere indicati attraverso le stesse sequenze di escape viste in C++.

### 6.3.2 INCLUDE

La direttiva include permette di includere file sorgente:

```
.INCLUDE "./path"
```

essa può essere posta in cima o in fondo (si consiglia in cima). Il percorso del file sorgente incluso, inoltre, deve essere per forza incluso tra doppi apici.

### 6.3.3 SET

Con la direttiva SET possiamo creare costanti simboliche, richiamabili all'interno di istruzioni (ovviamente precedute dal dollaro)

```
.SET nome, espressione
```

- **nome**, cioè l'identificativo della costante
- **espressione**, cioè il contenuto della costante

#### Esempio

```
.SET dimensione, 4  
.SET n_iter, (100*dimensione)  
...  
MOV $n_iter, %CX #op. immediato, ci vuole '$'
```

è il discorso di *pippo* fatto all'inizio.

#### 6.3.3.1 Calcolare memoria occupata

Il seguente codice permette di calcolare il numero di byte occupati in memoria

```
dato1: .FILL 1024, 4  
dato2: .FILL 100, 2  
...  
datoN : .FILL 350, 2  
foo: .BYTE 1  
      .SET occupazione, (foo-dato1)  
      ...  
      MOV $occupazione, %ECX
```

dichiaro una variabile foo che non serve a niente, ma permette di fare la differenza tra indirizzi. Questo mi permette di individuare il numero di byte occupati!

### 6.3.3.2 Costanti numeriche

#### Differenza tra naturali e interi

- I naturali non sono preceduti da segno, vengono convertiti nella loro rappresentazione in base 2.
- Gli interi sono preceduti da un segno (positivo o negativo), e vengono convertiti nella loro rappresentazione in complemento a bit sul numero di bit opportuno.

La presenza del segno permette all'assemblatore di capire se stiamo parlando di naturali o interi.

**Basi numeriche** I numeri possono essere scritti in base 2, 8, 10, 16. Abbiamo già visto che si identifica la base delle costanti numeriche mediante prefissi:

- I numeri in base 2 iniziano con 0b
- I numeri in base 8 iniziano con 0
- I numeri in base 10 non iniziano per 0
- I numeri in base 16 iniziano per 0x

**Attenzione all'intervallo di rappresentazione** Supponiamo di voler porre 128 come contenuto del registro *AL*

```
MOV $128, %AL  
MOV $+128, %AL
```

scrivere la seconda istruzione invece della prima potrebbe dare problemi. Ricordiamo che gli estremi dell'intervallo di rappresentazione degli interi sono asimmetrici  $[-128, +127]$ . Il numero 128, in 8bit, è rappresentabile come naturale ma non come intero!

**Come si comporta l'assemblatore?** Quando le rappresentazioni non sono della dimensione giusta vengono

- troncate se troppo lunghe (solitamente l'assemblatore avverte)
- estese se di dimensione minore (solitamente l'assemblatore non ce lo dice)

## 6.4 Controllo di flusso

Abbiamo già detto che in assembler non sono presenti i costrutti legati al controllo di flusso, precisamente

- if...then...else
- for...
- while...
- do...while

tuttavia con le istruzioni di salto viste nelle scorse lezioni e le notazioni simboliche introdotte oggi possiamo ottenere la stessa azione di quei costrutti. L'approccio adottato, cioè imparare prima il C++ con questi costrutti e poi l'assembler, aiuta moltissimo.

#### 6.4.1 if...then ... else

```

if (%AX<variabile)      // naturali
    {ist1; ...; istN;}
else
    {istN+1; ...; istN+M;}
ist_nuova;

```

- Invertire ramo **then** e ramo **else**

```

        CMP variabile, %AX
        JB ramothen
ramoelse:   istN+1
        ...
        istN+M
        JMP segue
ramothen:   ist1
        ...
        istN
segue:      ist_nuova

```

- Invertire la **condizione**

```

        CMP variabile, %AX
        JAE ramoelse
ramothen:   ist1
        ...
        istN
        JMP segue
ramoelse:   istN+1
        ...
        istN+M
segue:      ist_nuova

```

- Verifico la condizione
- Se è vera eseguo il then-statement
- Se è falsa eseguo l'else-statement (se presente)

#### 6.4.2 for...

```

for (int i=0; i<var; i++)      // "var" variabile o costante
{ist1; ...; istN}

        MOV $0, %CX
ciclo:   CMP var, %CX
        JE fuori
        ist1
        ...
        istN
        INC %CX      // NON ADD $1, %CX (sta su 3 byte)
        JMP ciclo
fuori:   ...

```

- Inizializzo la variabile di controllo
- Verifico la condizione dove è presente la variabile di controllo
- Se è vera eseguo il contenuto del body del for. Dopo aver eseguito il body eseguo lo step (incremento la variabile di controllo)
- Se è falsa passo alle istruzioni successive al for.

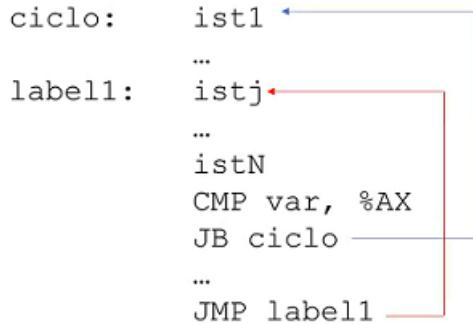
### 6.4.3 do...while

```
do
    {ist1; ...; istN;}
    while (AX<var);

ciclo:    ist1
...
istN
CMP var, %AX
JB ciclo
```

- Eseguo il contenuto del body. All'interno del body aggiorno la variabile di controllo.
- Verifico la condizione dove è presente la variabile di controllo
- Se è vera eseguo nuovamente il contenuto del body.
- Se è falsa passo alle istruzioni successive al do-while.

### 6.4.4 Spaghetti-like coding



In Assembler, contrariamente a C++, C e Pascal, è possibile saltare nel mezzo di un ciclo. La cosa può causare non pochi problemi: si parla di *spaghetti-like* coding poichè il programma, se simile a un piatto di spaghetti, può avere risultati imprevedibili. I linguaggi strutturati (quelli citati poco fa) sono stati pensati apposta per evitare questo stile di programmazione

Unico punto di ingresso, unico punto di uscita

Nel C++ esiste il costrutto *goto* (rammentato sul libro di Domenici e Frosini, ma mai affrontato a FdP): quel costrutto, dice Stea, serve solo per far danni e non conviene usarlo assolutamente!

## 6.4.5 LOOP

Il comando LOOP decrementa ogni volta il contenuto di ECX, e salta finchè il valore di ECX non sarà uguale a 0.

```
MOV $5, %ECX
ciclo: ist1
...
istN;
LOOP ciclo
```

La cosa è utilissima per riprodurre le stesse azioni di un for.

**Attenzione** Il contenuto di ECX non deve essere modificato durante il ciclo.

### 6.4.5.1 For descendente e for ascendente con LOOP



## 6.4.6 LOOP condizionato

Le LOOP condizionali, precisamente le istruzioni LOOPE e LOOPNE, permettono di determinare l'uscita da un LOOP attraverso una istruzione CMP, posta all'interno del ciclo.

- Poniamo in ECX il numero massimo di iterazioni. Il ciclo si conclude col raggiungimento del numero massimo di iterazioni, o se la condizione della CMP non risulta più vera. **Condizioni presenti:** *Loop if equal, loop if zero, loop if not equal, loop if not zero.*

- LOOPE, LOOPNE (LOOPZ, LOOPNZ)
- Sensate soltanto **dopo** una CMP
- Il salto avviene se:
  1. La condizione e' vera
  2. Dopo il decremento, ECX!=0
- Si scrive in ECX il numero **massimo** di iterazioni
  - Il ciclo puo' terminare prima, se la condizione diventa falsa

```
MOV $10, %ECX
ciclo: ist1
...
istN
CMP <src>, <dest>
LOOPcond ciclo
```

#### 6.4.7 Utilità dei LOOP e manipolazione stringhe

I LOOP sono comodi ma non indispensabili. Possono essere posti in modo alternativo, per esempio attraverso una istruzione compare e una jump (con condizione). Andando avanti vedremo istruzioni per la manipolazione delle stringhe: sono presenti prefissi di ripetizione che permettono l'implementazione di cicli, quindi accessi sequenziali in memoria.

### 6.5 Sottoprogrammi e passaggio dei parametri

Abbiamo già detto che le istruzioni per i sottoprogrammi sono due: la CALL e la RET. Per la prima abbiamo un unico operando (l'indirizzo del sottoprogramma), mentre la RET non prevede operandi. Per passare parametri ai sottoprogrammi (e viceversa) ci si affida a delle convenzioni:

- si utilizzano locazioni di memoria condivise tra il programma chiamante e il programma chiamato (il programma chiamante mette da qualche parte certi valori, il programma chiamato lo sa e li va a pescare lì)
- si utilizzano i registri (il programma chiamante pone qualcosa in un registro, il programma chiamato lo sa e va a leggere lì; la cosa vale anche al contrario)
- usare la pila (non affronteremo questo metodo, è un casino e ce ne occuperemo a Calcolatori elettronici - questo metodo è utilizzato proprio dai compilatori).

**Osservazione** In Assembler non esiste il concetto di variabile locale: la memoria principale è indirizzabile da qualunque sottoprogramma.

**Regola** Convenzione buona e giusta è specificare i parametri di ingresso e di uscita del sottoprogramma attraverso commenti.

```
MOV ..., %AX          # preparazione dei parametri
MOV ..., %EBX         # per la chiamata di sottoprogramma
CALL sottoprg
MOV %CX, var          # utilizzo del valore di ritorno

# sottoprogramma "sottoprg", [descrizione]
# ingresso: %AX, [descrizione]
#           %EBX, [descrizione]
# uscita:   %CX, [descrizione]

sottoprg: ...
...
MOV $..., %CX # preparazione del valore di ritorno
RET
```

ogni parametro deve essere accompagnato da descrizione (contenuto, scopo...)

# Capitolo 7

## Giovedì 08/10/2020

### 7.1 Uso dei registri ed effetti collaterali

I registri che non contengono valori di ritorno non devono essere sporcati da un sottoprogramma.  
Precisamente:

- I registri usati da un sottoprogramma devono essere salvati in pila
- Attenzione ai registri toccati, non è detto che si vedano nel codice (dobbiamo controllare la documentazione)
- La pila deve essere tenuta in ordine: per ogni push dovrà esserci una POP. La cosa avviene nel seguente modo

```
sottoprog: PUSH x1  
          PUSH x2  
          PUSH x3  
          ...  
          POP x3  
          POP x2  
          POP x1  
          RET
```

Se la RET, quando pesca l'indirizzo di ritorno dalla pila, trova valori casuali il programma si inchioda.

## 7.2 Sottoprogramma principale

Il `_main` è il sottoprogramma principale.

- Deve terminare con una RET.
- Solitamente ci si aspetta un valore nel registro EAX:
  - 0 se tutto è ok
  - un valore  $\neq 0$  se ci sono errori (e il valore consiste nel codice errore)

quanto detto consiste in una convenzione che non è obbligatoria all'esame.

**Prassi usata spesso** Se vogliamo rispettare la condizione ci basta mettere la seguente istruzione subito prima della RET

`XOR %EAX, %EAX`

## 7.3 Dichiarazione e allocazione di spazio per la pila

Abbiamo lasciato in sospeso un discorso relativo alla pila: essa esiste perché qualcuno la dichiara e la inizializza (allocando spazio). Dobbiamo:

- dichiarare lo stack
- inizializzare il registro ESP affinchè esso punti alla cella successiva al fondo dello stack (non in fondo, ma alla prima locazione successiva all'area riservata alla pila).

La dichiarazione avviene nella parte DATA

```
.DATA  
mystack: .FILL 1024, 4  
.SET initial_esp, (mystack + 1024* 4)  
  
.TEXT  
_main: NOP  
      MOV $initial_esp, %ESP
```

La dimensione dello stack è problema del programmatore (sappiamo noi cosa andiamo a fare e quindi quanto spazio ci serve).

**Tuttavia** Nel nostro ambiente, ma non in Assembler in generale, possiamo omettere la dichiarazione dello stack. Ci pensa l'ambiente secondo regole sue.

## 7.4 Ingresso/uscita e sottoprogrammi di utilità

**Ricordiamo** Abbiamo già detto che le istruzioni IN e OUT sono istruzioni privilegiate, normalmente non utilizzabili. L’assemblatore assembra ma al momento dell’esecuzione viene lanciata un’eccezione di protezione.

**Come facciamo?** Ricorriamo a sottoprogrammi offerti dal sistema DOS.

- DOS offre sottoprogrammi che girano in modalità sistema e che possono usare le IN/OUT
- Tuttavia questi sottoprogrammi sono molto primitivi: si ha l’ingresso o l’uscita di un solo carattere.
- A partire da questi sottoprogrammi i docenti Stea e Corsini hanno costruito altri sottoprogrammi che permettono ingresso e uscita a più alto livello (non troppo). Si specifica che
  - Gli ingressi avvengono da tastiera
  - Le uscite avvengono su video, precisamente su una finestra DOS di 80x25 caratteri.

## 7.5 Osservazioni sulla tabella ASCII

Prendiamo una tabella riepilogativa della codifica ASCII (che ricordiamo, da FdP, essere su 7bit). Ogni combinazione consiste in un particolare carattere stampabile. Osserviamo che:

- I caratteri numerici sono consecutivi, stessa cosa i caratteri maiuscoli e quelli minuscoli (nella tabella individuiamo prima le maiuscole delle minuscole).
- Le sequenze binarie associate ai caratteri numerici presentano la seguente struttura

0011XXXX

La parte rimanente della sequenza consiste nella rappresentazione binaria del corrispondente numero naturale. Questo ci permette di capire, in modo agile, se una certa sequenza rappresenta un certo carattere numerico.

- Le sequenze binarie associate alle lettere presentano la seguente struttura

01XYYYYY

dove  $X$  può assumere come valore 0 o 1. Precisamente, se si ha 0 il carattere è maiuscolo, se si ha 1 il carattere è minuscolo. Segue che dato un carattere maiuscolo (o viceversa) possiamo ottenere il corrispondente carattere minuscolo (o viceversa) attraverso la modifica del bit  $X$  (con una maschera).

- Ci interessano tre caratteri speciali in particolari
  - il backspace. Si ritorna indietro di una posizione nella riga
  - il line feed, o avanzamento di riga. Si scorre di uno le righe
  - il carriage return, o ritorno carrello. Si riporta il cursore all’inizio di una riga.

### 7.5.1 I/O da tastiera e video

Si hanno, sia in ingresso che in uscita, codifiche ASCII di singoli caratteri. Non abbiamo l'I/O tipica del C++. Prendiamo ad esempio il numero 32:

- stampo il primo carattere ASCII 0x33 (3)
- stampo il secondo carattere ASCII 0x32 (2)

**Esempio** Supponiamo di voler fare ingresso da tastiera di un numero naturale a 2 cifre in base 10

- Memorizzo due codifiche ASCII:  $c_1, c_0$
- Calcolo le singole cifre decimali:  $a_1, a_0$  (a partire dalle codifiche  $c_1, c_0$ )
- Ricostruisco il numero digitato:  $10 \cdot a_1 + a_0$  (tengo conto delle cifre più significative e quelle meno significative)

### 7.5.2 File utility

```
.INCLUDE "C:/amb_GAS/utility",
```

attraverso la direttiva andiamo a includere un pack di sottoprogrammi creato da Stea e Corsini per l'ingresso e l'uscita di caratteri e stringhe.

#### 7.5.2.1 Sottoprogrammi di I/O (su cui si basano i sottoprogrammi della sezione successiva)

- **inchar**: metto nel registro AL la codifica ASCII del tasto premuto (solo questo, non stampo)
- **outchar**: metto sul video la codifica ASCII contenuta in AL
- **newline**: per andare a capo (stampo due caratteri: ritorno carrello e line feed)
- **pauseN**: metto in pausa il programma stampando il seguente messaggio

```
Checkpoint number N. Press any key to continue.
```

dove  $N$  è una cifra decimale.

#### 7.5.2.2 Sottoprogrammi a livello più alto

- **inline**: consente di portare una stringa di massimo 80 caratteri in un buffer da memoria (digitando da tastiera con eco su video).
  - il registro EBX consiste nell'indirizzo di memoria del buffer
  - il registro CX contiene il numero di caratteri da leggere (massimo 80, una riga di DOS)

La lettura da tastiera termina dopo 78 caratteri (78 + carriage return + line feed) o se premiamo invio. Cosa molto comoda è l'interpretazione del backspace come ordine di cancellare dal buffer e dal video l'ultimo carattere digitato.

- **outline:** stampa a video massimo 80 caratteri. Si ferma prima se trova un carattere di ritorno carrello ed eventualmente stampa i caratteri per andare a capo
  - il registro EBX consiste nell'indirizzo di memoria del buffer. Se vediamo simboli esoterici significa che il registro contiene sequenze di zeri e uno che non hanno a che vedere con la codifica ASCII.
- **outmess:** stessa funzione del sottoprogramma precedente
  - il registro EBX consiste nell'indirizzo di memoria del buffer.
  - il registro CX contiene il numero di caratteri da stampare a video

#### 7.5.2.3 Sottoprogrammi per l'ingresso/uscita di numeri esadecimali

- **inbyte, inword, inlong:** prelevano da tastiera (facendo eco su video) 2, 4 o 8 caratteri (non fino a x caratteri). La sequenza ottenuta in ingresso viene interpretata come un numero esadecimale. Ignorano qualunque altro carattere premuto.
  - il numero esadecimale viene posto in *AL, AX, EAX* (in base al sottoprogramma scelto).
- **outbyte, outword, outlong:** stampano sul video, rispettivamente, 2, 4 o 8 caratteri corrispondenti a cifre esadecimale.
  - Vengono estratte interpretando il contenuto di *AL, AX, EAX* (in base al sottoprogramma scelto) come un numero naturale.

#### 7.5.2.4 Sottoprogrammi per l'ingresso/uscita di numeri decimali

- **indecimal\_byte, indecimal\_word, indecimal\_long:** prelevano da tastiera fino a 3,5 o 10 cifre decimali. La sequenza è interpretata come un numero decimale. Tutti gli altri caratteri vengono ignorati.
  - il numero decimale viene posto in *AL, AX, EAX* (in base al sottoprogramma scelto).
- Se il numero è troppo grande il numero viene troncato.
- **outdecimal\_byte, outdecimal\_word, outdecimal\_long:** stampano sul video il contenuto di *AL, AX, EAX* (in base al sottoprogramma scelto) interpretato come un numero naturale in base 10 sul numero di cifre strettamente necessario.

## 7.6 Istruzioni che manipolano le stringhe

In Assembler non esistono tipi di dati né strutture dati. Esistono soltanto byte, word e long. L'unica cosa simile a una struttura dati è quella dei vettori:

- possiamo dichiarare vettori di variabili di una certa direzione
- possiamo fare indirizzamento con displacement + registri base/indice

introduciamo le cosiddette istruzioni stringa, che permettono di copiare interi buffer di memoria (quindi interi blocchi di memoria). Queste istruzioni, oltre ad essere comodee, sono efficienti!

- il registro ESI è puntatore a sorgente
- il registro EDI è puntatore a destinazione

**Copia di un vettore** Supponiamo di voler copiare un vettore. Ho un ciclo con molte istruzioni

```
vett_sorg: .FILL 1000, 4
vett_dest: .FILL 1000, 4
ciclo:
    MOV $1000, %ECX
    LEA vett_sorg, %ESI
    LEA vett_dest, %EDI
    MOV (%ESI), $EAX
    MOV %EAX, (%EDI)
    ADD $4, %ESI
    ADD $4, %EDI
    LOOP ciclo
```

usiamo le istruzioni stringa

```
MOV $1000, %ECX
LEA vett_sorg, %ESI
LEA vett_dest, %EDI
REP MOVSL
```

come vediamo la cosa è estremamente semplificata (una sola istruzione è sintomo di maggiore efficienza). REP consiste in un prefisso di ripetizione, mentre MOVSL rappresenta un'istruzione stringa.

### 7.6.1 MOVE DATA FROM STRING TO STRING

Istruzione che permette di copiare vettori

- il suf è obbligatorio (NON ABBIAMO OPERANDI ESPLICITI) e stabilisce di quanti bit dobbiamo copiare (B, W, o L).
- Si utilizzano i puntatori di memoria ESI (puntatore sorgente) ed EDI (puntatore destinatario), che vengono incrementati o decrementati.
- L'istruzione, senza REP, copia il numero di byte specificato dal suffisso dall'indirizzo di memoria puntato da ESI all'indirizzo di memoria puntato da EDI.
- Se è incluso il prefisso REP le azioni appena dette vengono replicate per il numero di volte specificato in ECX. ECX viene decrementato finché non avrà come valore 0.

la copia, attenzione, può essere sia in avanti che indietro: utilizzeremo un nuovo registro non ancora visto, il *Direction Flag*.

- Se DF = 0 si copia in avanti. ESI ed EDI vengono incrementati dopo aver copiato il numero di byte indicato.
- Se DF = 1 si copia all'indietro. ESI ed EDI vengono decrementati dopo aver copiato il numero di byte indicato.

Ovviamente ESI ed EDI, con DF = 1, saranno inizializzati puntando all'ultimo byte, piuttosto che al primo.

**Attenzione** DF non vale 0 di default.

- **FORMATO:** MOVSSuf REP MOVSSuf
- **AZIONE:** Copia il numero di byte specificato dal suffisso *suf* dall'indirizzo di memoria puntato da ESI all'indirizzo di memoria puntato da EDI. Se DF=0, somma ad ESI e ad EDI il numero di byte specificato dal suffisso. Se DF=1, sottrae da ESI e da EDI il numero di byte specificato dal suffisso.
- Se viene premesso il prefisso REP, allora le azioni indicate sopra vengono ripetute per il numero di volte specificato in ECX, che viene decrementato fino a zero.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

### 7.6.2 Istruzioni *Direction Flag* (STD e CLD)

Abbiamo due istruzioni per gestire questo flag:

- STD (*Set direction flag*), si imposta DF = 1
- CLD (*Clear direction flag*), si imposta DF = 0

queste istruzioni dovranno essere poste prima della MOVE DATA.

### 7.6.3 LODSsuf (*load string*) e STOSsuf (*store string*)

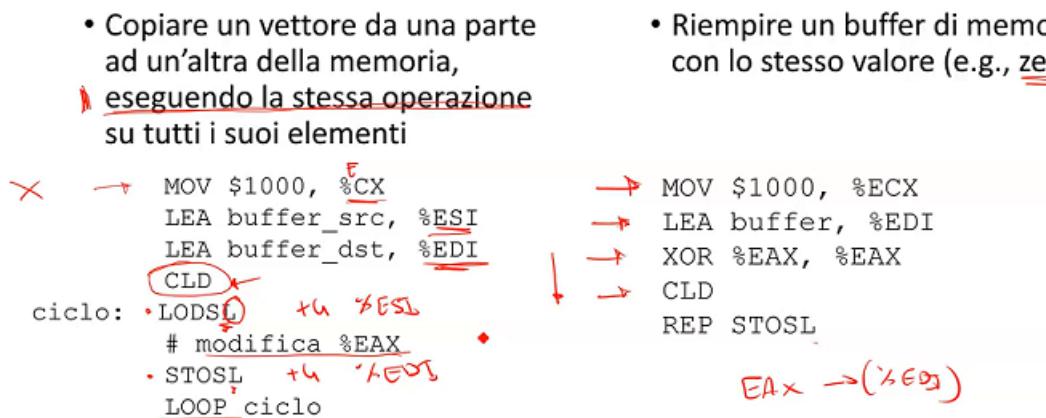
- **LODSsuf:**

- copia in **AL, AX, oppure EAX** (a seconda del suffisso) il contenuto della memoria all'indirizzo puntato da **ESI**.
- A seconda del valore del flag DF, incrementa o decrementa di **1, 2, 4 ESI**.

- **STOSsuf:**

- copia il registro **AL, AX, oppure EAX** (a seconda del suffisso) in memoria all'indirizzo puntato da **EDI**.
- A seconda del valore del flag DF, incrementa o decrementa di **1, 2, 4 EDI**.

**Osservazione** Non ha senso utilizzare i prefissi di ripetizione con LODS: se io sposto il contenuto nei registri significa che voglio utilizzarlo per fare qualcosa. Ripetere l'operazione  $n$  volte consecutive mi impedisce di fare ciò.



### 7.6.4 [Privilegiate] Istruzioni stringa per l'I/O - INSsuf, OUTSsuf

**Attenzione** Sono istruzioni privilegiate. Ne ripareremo a *Calcolatori elettronici*.

- **INSsuf:** fa ingresso di uno, due, quattro byte dalla porta di I/O il cui offset è contenuto in **DX**. L'operando viene inserito in memoria a partire dall'indirizzo di memoria contenuto in **EDI**. A seconda del valore del flag DF, incrementa o decrementa di 1, 2, 4 il contenuto di **EDI**.
- **OUTSsuf:** copia uno, due, quattro byte, contenuti in memoria a partire dall'indirizzo di memoria contenuto in **ESI**, alla porta di I/O il cui offset è contenuto in **DX**. A seconda del valore del flag DF, incrementa o decrementa di 1, 2, 4 il contenuto di **ESI**.



### 7.6.5 COMPARE STRINGS (CMPSsuf) - confronto memoria-memoria

- **CMPSsuf:** confronta il contenuto delle locazioni (doppi locazioni, quadruple locazioni) indirizzate da **ESI** (sorgente) ed **EDI** (destinatario). A seconda del valore del flag DF, incrementa o decrementa di 1, 2, 4 il contenuto di **ESI**, **EDI**.



### 7.6.6 SCAN STRING (SCASsuf) - confronto registro-memoria

- **SCASsuf:** Confronta il contenuto del registro **AL**, **AX**, oppure **EAX** (a seconda del suffisso) con la locazione (doppia locazione, quadrupla locazione) di memoria indirizzata da **EDI**. L'algoritmo usato nel confronto è identico a quello della CMP. A seconda del valore del flag DF, incrementa o decrementa di **1, 2, 4** il contenuto di **EDI**.

Serve a trovare un elemento di valore noto dentro a un vettore

- DF = 0: cerca la prima occorrenza
- DF = 1: cerca l'ultima occorrenza



### 7.6.7 Prefissi di ripetizione

#### • REP

- Puo' essere usato con MOVS, DODS, STOS, INS, OUTS.
- Utilizzo con LODS è del tutto privo di senso, in quanto si finisce a sovrascrivere  $N$  volte lo stesso registro.
- Si applica ad una istruzione (non ad un blocco di codice)

#### • REPE / REPNE

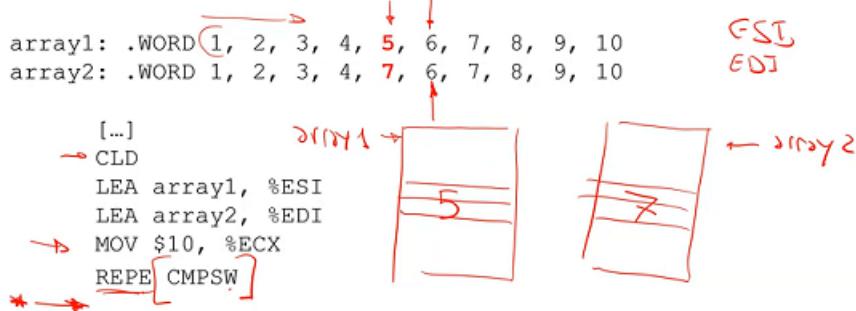
- Puo' essere usato con CMPS, SCAS.
- Si applica ad una istruzione (non ad un blocco di codice)
- Si fanno al massimo ECX ripetizioni, finche' la condizione specificata e' vera

DF=0 ↓ ↗ MOV \$100, ECX  
REPE SCASL  
REPNE CMPSL

### 7.6.8 Utilità delle due direzioni

- Trovare prima o ultima occorrenza di un dato in un vettore.

- Trovare il primo elemento differente tra due vettori



- **Copia di buffer parzialmente sovrapposti.** Attraverso un po' di algebra, e conoscendo le dimensioni dei buffer, possiamo notare se questi sono sovrapposti o meno. Se sono disgiunti possiamo copiare nel modo che preferiamo, se invece sono sovrapposti dobbiamo analizzare il tipo di sovrapposizione e procedere di conseguenza. Le situazioni possibili sono le seguenti:

- prima il buffer destinatario e poi quello sorgente;
- prima il buffer sorgente e poi quello destinatario



Dalle immagini deduciamo qual è la direzione di copia da assumere



in entrambi i casi le prime cose copiate sono quelle presenti in entrambi i buffer.

# Capitolo 8

**Venerdì 09/10/2020**

## 8.1 Conclusioni su Assembler

### 8.1.1 Differenze tra compilatore e assemblatore

Abbiamo già detto che...

- un compilatore C++ ottimizza il codice per il sistema su cui gira, mentre...
- un assemblatore traduce le istruzioni una per una.

Il compilatore ottimizza il nostro codice, mentre l'assemblatore traduce 1 : 1. Questo significa che l'efficienza del programma dipenderà esclusivamente da quanto scritto da noi.

### 8.1.2 Tempo di esecuzione di un programma

Il tempo di esecuzione non può essere determinato a partire dalle istruzioni: ciò che si misura non è il programma ma un processo<sup>1</sup>. Il tempo dipende

- dai dati
- dallo stato del sistema
- dipende da chi altri sta usando il processore, e da cosa ci fa.

Segue che se misuro  $n$  volte l'esecuzione dello stesso programma otterrò tempi molto diversi. La velocità di esecuzione è assolutamente imprevedibile. Inoltre

- il clock non va a velocità costante (dipende dal carico)
- il processo non gira necessariamente su un solo core
- la presenza di mille altri meccanismi che da una parte rendono i processi più veloci, ma da un'altra rendono i tempi più efficienti:
  - Memoria cache. Area posta tra CPU e RAM: molto piccola, ma in grado di ridurre i passaggi dalla RAM aumentando la velocità. Nel 98% dei casi i dati che mi servono stanno nella cache. Chiaramente se i dati non sono in cache dovremo passare dalla memoria RAM.

---

<sup>1</sup>Noi non assaggiamo la ricetta, ma la torta, cioè il prodotto della ricetta (cit.Stea)

- Code di prefetch.
- Esecuzione in pipeline
- Esecuzione non sequenziale
- Branch prediction.

Sono questioni che saranno affrontate alla magistrale o in corsi come *Calcolatori elettronici*.

### 8.1.3 Lunghezza delle istruzioni e tempo di fetch

Le istruzioni occupano un certo spazio in memoria. Ciò dipende:

- dall'OPCODE (quindi dall'istruzione)
- da dove sono gli operandi (in larga parte)

Se gli operandi sono registri le istruzioni stanno su un byte (normalmente), in caso di operandi immediati questi dovranno essere codificati nell'istruzione (e occuperanno 1,2, 4 byte).

#### Esempio

```
MOV $0, %EAX
XOR %EAX, %EAX
```

La MOV non è sbagliata, ma lo XOR occupa minore memoria e il fetch è più veloce.

### 8.1.4 Tempo di esecuzione delle istruzioni

Il tempo di esecuzione dipende molto dall'architettura del processore (anche all'interno della stessa famiglia, per esempio la Intel). In generale:

- le istruzioni operative della ALU (tranne divisione e moltiplicazione) costano poche:  $O(1)$  clock
- divisione e moltiplicazione costano molto:  $O(10)$  clock
- Istruzioni trascendenti della FPU (sin, cos) costano addirittura di più:  $O(100)$  clock
- Le istruzioni di controllo hanno un costo alto, ma per altri motivi.

#### 8.1.4.1 Come si evitano moltiplicazioni e divisioni?

Moltiplicazioni e divisioni possono essere scomposte in catene di shift. Tenendo conto della dimensione degli operandi possiamo utilizzare le operazioni di shift al posto di quelle della divisione. Si osservi, inoltre, che la LEA può essere utilizzata per fare conti (ricordiamo la struttura generica di un indirizzo)

**Compilatore C++** Il compilatore utilizza trucchi di questo tipo. Dire  $a = b \cdot c$  non significa automaticamente tradurre con (I)MUL. Attraverso delle tabelle di corrispondenza il compilatore sceglie la traduzione più efficiente.

**All'esame** La cosa non è vitale: i programmi eseguiti sono "banali", segue che adottare un metodo o un altro non sia significativo nel determinare la velocità.

$$ES: \quad z = 16 \cdot x + 3 \cdot y - 13500$$

OPCODE + displacement (base, indice, scala)

$$\text{Indirizzo} = \left| \text{base} + \text{indice} \times \text{scala} \pm \text{displacement} \right|_2^{32}$$

MOV Y, %EAX

LEA -13500(%EAX,%EAX,2), %EAX

MOV X, %EBX

SHL \$4, %EBX

[ADD %EBX,%EAX]

MOV %EAX, z

$$\left| \frac{Y + Y \cdot 2 - 13500}{Y \cdot 3} \right|_2^{32}$$

$$16 \cdot x = 2^4 \cdot x$$

SOMMO LE DUE PARTI E OTTENGO z

## Parte II

# Esercitazioni di Zippo

# Capitolo 9

Martedì 06/10/2020

## 9.1 Assemblaggio

Dalla finestra di DOSBox poniamo

ASSEMBLE.BAT PERCORSO\FILE.S

saranno generati due file:

- il file eseguibile assemblato (nella stessa cartella dove si trova il file .s)
- il file listato.txt (che si trova nella cartella WORK)

### 9.1.1 File listato

La parte più interessante del file listato è quella relativa alla traduzione del codice. Possiamo vedere

- la riga di codice
- l'indirizzo di memoria (locazione di memoria dove è presente il dato o l'istruzione). Si capisce, da certi indirizzi, la dimensione delle variabili (si veda riga 5 e riga 6 come esempio)
- la relativa traduzione in bit

```
3          .DATA
4          # Abbiamo bisogno di un long per contare i dati e uno per svolgere il cont
5 0000 01010F0F  dato:  .LONG 0x0F0F0101
6 0004 00      conteggio:  .BYTE 0x00 # Inizializzato a 0
7
8 0005 00000000  .TEXT
8 00000000
8 000000
9          # Inizializziamo i registri
10         # Definiamo un ciclo poichè dovremo analizzare tutta la variabile.
11         # Si fa il confronto per verificare se EAX è nullo. A un certo punto avremo
12         # Incrementiamo di 1 con ADD WITH CARRY (con la shift spostiamo sempre il
```

```
13 0000 90      _main : NOP
14 0001 B100    MOVB $0x00, %CL
15 0003 A1000000 MOVL dato, %EAX
15      00
16
17 0008 83F800  comp: CMPL $0x00, %EAX
18 000b 7407    JE fine
19
20 000d D1E8    SHRL %EAX
21 000f 80D100  ADCB $0x00, %CL
22 0012 EBF4    JMP comp
23
24 0014 880D0400 fine: MOVBL %CL, conteggio
24      0000
25 001a C3909090 RET
25      9090
```

Altra cosa utile, forse la più utile per noi che debuggiamo, è la lista dei simboli definiti e non definiti

#### DEFINED SYMBOLS

[...]

```
ESERCIZI\PRIMO.S:13      .text:00000000 _main
ESERCIZI\PRIMO.S:5      .data:00000000 dato
ESERCIZI\PRIMO.S:6      .data:00000004 conteggio
ESERCIZI\PRIMO.S:17      .text:00000008 comp
ESERCIZI\PRIMO.S:24      .text:00000014 fine
```

#### NO UNDEFINED SYMBOLS

in fondo avremo la lista di simboli non definiti in caso di errore.

## 9.2 Debugging

Come debugger utilizziamo GDB. Eseguiamo DEBUG.BAT con il percorso del file eseguibile

```
011> C:\WORK>DEBUG.BAT ESERCIZI\PROVA2.EXE
GNU gdb 6.1.1
Copyright 2004 Free Software Foundation, Inc.
This GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions,
see the COPYING file for details.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "--host=i386-pc-msdosdjgpp --target=djgpp"...
Breakpoint 1 at 0x1e60: file ESERCIZI\PROVA2.S, line 10.

Breakpoint 1, main () at ESERCIZI\PROVA2.S:10
warning: Source file is more recent than executable.

10      _main : NOP
Current language: auto; currently asm
(gdb)
```

come vedremo GDB è molto scomodo da utilizzare. Non appena viene aperto il debugger si pone un breakpoint all'inizio del programma in modo da poter decidere come muoverci.

### 9.2.1 Comandi

Si distinguono due tipi di comandi: comandi per controllare l'esecuzione e comandi per controllare lo stato del programma. Essi sono:

- **list source:**

```
[list|l] X
```

mostra uno spezzone del file sorgente. *X*, facoltativo, consiste in un numero identificativo di una riga. Se omesso mi mostra la parte di file sorgente eseguita fino ad ora, se non omesso mi mostra la parte di file sorgente fino al rigo *X*.

```
.ls
(gdb) 1
un5      .DATA
6      dato: .LONG 0b1010101 # Ho inizializzato ponendo come valore 5 (notazione
e decimale)
7      risultato: .BYTE 0x00
8
9      .TEXT
10     _main : NOP
11     MOVB dato, %EAX
12     MOUL $0x00, %CL
13
14     comp : CMPL $0x00, %EAX
(gdb)
```

- **insert breakpoint:**

```
[break|b] nome_istruzione
```

inserisco un breakpoint all'istruzione individuata dal nome indicato. Si può indicare il numero di riga al posto del nome dell'istruzione

```
10      _main : NOP
Current language: auto; currently asm
(gdb) break comp
Breakpoint 2 at 0x1e68: file ESERCIZI\PROVA2.S, line 14.
(gdb) break fine
Breakpoint 3 at 0x1e74: file ESERCIZI\PROVA2.S, line 20.
(gdb) _
```

- **info breakpoints:**

```
[info breakpoints|i b]
```

mostra i breakpoint inseriti. Per ogni breakpoint abbiamo un numero identificativo e l'istruzione a cui è associato

```
(gdb) break 3
Breakpoint 3 at 0x1e74: file ESERCIZI/PROVAZ.S, line 20.
(gdb) i b
Num Type      Disp Enb Address   What
1  breakpoint  keep y  0x00001e60 ESERCIZI/PROVAZ.S:10
              breakpoint already hit 1 time
2  breakpoint  keep y  0x00001e68 ESERCIZI/PROVAZ.S:14
3  breakpoint  keep y  0x00001e74 ESERCIZI/PROVAZ.S:20
(gdb)
```

- **delete breakpoint:**

```
[delete breakpoint|d b] NUMBREAKPOINT
```

il numero del breakpoint è facoltativo: se omesso comporta l'eliminazione di tutti i breakpoint inseriti.

```
(gdb) d b 2
(gdb) i b
Num Type      Disp Enb Address   What
1  breakpoint  keep y  0x00001e60 ESERCIZI/PROVAZ.S:10
              breakpoint already hit 1 time
3  breakpoint  keep y  0x00001e74 ESERCIZI/PROVAZ.S:20
(gdb)
```

- **step:**

```
[step | s] n
```

permette di procedere avanti di un certo numero di istruzioni. *n* può essere omesso, in quel caso è uguale ad 1

```
3  breakpoint  keep y  0x00001e74
(gdb) step
11             MOVB dato, %EAX
(gdb) step
12             MOVL $0x00, %CL
(gdb) step 2
15             JE fine
(gdb) step
16             SHRL %EAX
(gdb)
```

- **continue**

```
[continue| c]
```

mette in esecuzione il programma dall'istruzione successiva fino al prossimo breakpoint

```
(gdb) continue
Continuing.

Breakpoint 3, fine () at ESERCIZI/PROVAZ.S:20
120  fine : MOVB %CL, risultato
(gdb) _
```

- **info registers:**

```
[info registers| i r]
```

mostra il contenuto di tutti i registri

```
Breakpoint 3, fine () at ESERCIZI/PROVAZ.S:20
20      fine : MOVB %CL, risultato
(gdb) i r
eax          0x0      0
ecx          0x4      4
edx          0x33f    831
ebx          0x102e   4142
esp          0x8dd44  0x8dd44
ebp          0x8dd58  0x8dd58
esi          0x54     84
edi          0xd60    56672
eip          0x1e74   0x1e74
eflags        0x3046  12358
ott
cs           0xe7     231
ss           0xef     239
ds           0xef     239
es           0xef     239
fs           0xdf     223
gs           0xff     255
(gdb) _
```

- **print:**

```
[print| p]/k $reg
```

mostra il contenuto del registro *reg*. *K* può essere omesso e permette di ottenere una rappresentazione in particolare del valore.

- *k* = *x*, rappresentazione esadecimale
- *k* = *t*, rappresentazione binaria
- *k* = *d*, stampa come decimale interpretato come naturale
- *k* = *u*, stampa come decimale interpretato come intero.

- **examine memory:**

```
x /numerotipo nome_di_una_variabile
```

visualizza il valore della variabile riferita.

- numero consiste nel numero di componenti della variabile da esaminare. Se omesso numero equivale a 1
- tipo consiste nel tipo della variabile. Gli unici valori possibili sono *b*, *w* ed *l*. Se omesso tipo sarà *l*.

Ovviamente non è possibile omettere entrambi i parametri. Con le etichette simboliche è necessario porre la & commerciale poco prima. Se lascio vuoto la locazione implicita consiste in quella dell'istruzione appena eseguita.

```
rs          0xdf    223
gs          0xff    255
(gdb) x
0x1e74 <fine>: 0x9ee40d88
(gdb) x &dato
0x9ee0 <dato>: 0x00000055
(gdb) _
```

- **quit:** per uscire dal debugger ed eseguire nuovi comandi sul DOS.

```
1 #conteggio dei bit a 1 in un long
2
3 .GLOBAL _main
4
5 .DATA
6 dato:      .LONG 0x0F0F0101
7 conteggio:  .BYTE 0x00
8
9
10 .TEXT
11 _main:   NOP
12         MOVB $0x00, %CL
13         MOVL dato, %EAX
14
15 comp:    CMPL $0x00, %EAX    # while(eax != 0) {
16         JE fine
17
18         SHRL %EAX          # bit = eax[0]
19         ADCB $0x00, %CL      # if(bit == 1) cl++;
20
21         JMP comp            # }
22
23 fine:   MOVB %CL, conteggio
24         RET
25
```

```

1 # conteggio del numero di occorrenze di una lettera in una stringa
2
3 .GLOBAL _main
4
5 .DATA
6 stringa:    .ASCII "Questa e' la stringa di caratteri ASCII che usiamo come esempio"
7 lettera:    .BYTE 'e'
8 conteggio:   .BYTE 0x00
9
10 .TEXT
11 _main:      NOP
12             MOV $0x00, %CL
13             LEA stringa, %ESI
14             MOV lettera, %AL
15
16 comp:       CMPB $0x00, (%ESI)  # while (c != '\0') {
17             JE fine
18
19             CMP (%ESI), %AL
20             JNE poi          # if(c == 'e') {
21             INC %CL          #     CL++
22             #
23
24 poi:        INC %ESI          # c = stringa[i++]
25             JMP comp         # }
26
27
28 fine:      MOV %CL, conteggio
29             RET
30

```

```

1 #conteggio dei bit a 0 in un long
2
3 .GLOBAL _main
4
5 .DATA
6 # Abbiamo bisogno di un long per contare i dati e uno per svolgere il conteggio
7 dato:      .LONG 0x0F0F0101
8 conteggio:   .BYTE 0x00
9
10 .TEXT
11 # Inizializziamo i registri
12 # Definiamo un ciclo poichè dovremo analizzare tutta la variabile.
13 # Si fa il confronto per verificare se EAX è nullo. A un certo punto avremo solo bit
14 # uguali a 0
15 # Utilizzo il registro DL per salvare tutte le volte il valore del CF (con la shift
16 # spostiamo sempre il bit "espulso" nel carry flag): resetto DL e incremento con ADC
17 # (ovviamente mi interessa solo il CF).
18 # Se il valore del flag è diverso da 0 (e quindi uguale ad 1) salto l'incremento
19
20 _main:  NOP
21         MOVB $0x00, %CL
22         MOVL dato, %EAX
23
24 comp:   CMPL $0x00, %EAX    # while(eax != 0) {
25         JE fine
26
27         SHRL %EAX          # bit = eax[0]
28         MOVB $0x00, %DL      # DL = 0
29         ADCB $0x00, %DL      # DL = DL + bit
30         CMPB $0x00, %DL      # if(DL == 0) cl++;
31         JNE step
32         INC %CL
33 step:   JMP comp           # }
34
35 fine:   MOVB %CL, conteggio
36         RET
37

```

**Capitolo 10**

**Venerdì 16/10/2020**

```

1 # Conteggio del numero di occorrenze di un numero in un array
2
3 .GLOBAL _main
4 .INCLUDE "C:/amb_GAS/utility"
5
6 .DATA
7 # Inizializzo l'array attraverso una lista di numeri. In una seconda variabile indico
# il numero di elementi presenti nell'array
8 # Inizializzo ulteriori variabili per indicare il numero di cui voglio trovare le
# occorrenze e salvare il risultato finale del conteggio.
9 array:    .WORD 1, 256, 256, 512, 42, 2048, 1024, 1, 0
10 array_len: .LONG 9
11
12 numero:   .WORD 1
13 conteggio: .BYTE 0x00
14
15 .TEXT
16 # Inizializzo CL, che sarà il mio contatore.
17 # Inizializzo AX, dove pongo il numero di cui voglio trovare le occorrenze.
18 # Inizializzo ESI, che mi servirà per scorrere il vettore (Registro source index) e
# verificare se ho visto tutti gli elementi
19 _main:      NOP
20         MOV $0, %CL
21         MOV numero, %AX
22         MOV $0, %ESI          # esi = 0
23
24 # Confronto ESI con la lunghezza dell'array, se le lunghezze coincidono ho finito
25 # Confronto AX, dove è presente il numero di cui vogliamo le occorrenze, con
# l'elemento dell'array in posizione ESI
26 # Se non c'è uguaglianza salto l'incremento di CL
27 comp:      CMP array_len, %ESI      # while (esi != array_len ) {
28         JE fine
29
30         CMPW array(, %ESI, 2), %AX    # if( array[esi] == numero )
31         JNE poi
32
33         INC %CL
34
35 poi:       INC %ESI          # esi++
36         JMP comp           # }
37
38 # Sposto il risultato in AL per poter effettuare la stampa del risultato con il
# sottoprogramma
39 fine:      MOV %CL, %AL
40         CALL outdecimal_byte
41         RET
42

```

```

1 # Conteggio del numero di occorrenze di un numero in un array
2 # Differenze rispetto al programma fatto con Zippo:
3 # - Indico in ingresso gli elementi dell'array
4 # - Indico in ingresso il numero di cui voglio verificare le occorrenze
5 # Il numero di elementi da controllare è indicato nella costante numero_elementi
6
7 .GLOBAL _main
8 .INCLUDE "C:/amb_GAS/utility"
9
10 .DATA
11 .SET numero_elementi, 9
12 array:    .FILL numero_elementi, 2 # Allocazione array: elementi da 2byte
13 conteggio: .BYTE 0x00 # Conteggio delle corrispondenze
14
15 .TEXT
16 # Inizializzo CL, che sarà il mio contatore.
17 # AX è il registro che conterrà i numeri decimali inseriti in ingresso con la
18 # indecimal_word
19 # ESI lo utilizzo per scorrere l'array e individuare se ho visitato tutte le
20 # posizioni
21
22 _main:      NOP
23             MOV $0, %CL
24             MOV $0, %ESI      # esi = 0
25
26 popolamento: CMP $numero_elementi, %ESI
27             JE numdacontrollare
28
29             CALL indecimal_word
30             # Caldamente consigliata, situa orrenda senza
31             CALL newline
32
33             MOV %AX, array(, %ESI, 2)
34
35             INC %ESI
36             JMP popolamento
37
38 # Si estrae il numero di cui si vogliono trovare le corrispondenze. Anche in questo
39 # caso si guarda AX
40 # Confronto AX, dove è presente il numero di cui vogliamo le occorrenze, con
41 # l'elemento dell'array in posizione ESI
42 # Se non c'è uguaglianza salto l'incremento di CL
43 # NB: La ESI è adeguatamente resettata dopo l'uso precedente, idem la AX che viene
44 # sovrascritta con la indecimal_word
45
46 numdacontrollare: CALL indecimal_word
47             CALL newline
48             MOV $0, %ESI
49
50 reset:       CMP $numero_elementi, %ESI
51             JE fine
52
53 comp:        CMPW array(, %ESI, 2), %AX
54             JNE poi
55
56             INC %CL
57
58 poi:         INC %ESI
59             JMP comp

```

```
56 # Sposto il risultato in AL per poter effettuare la stampa del risultato con il
57 sottoprogramma
58 fine:      MOV %CL, %AL
59             CALL outdecimal_byte
60             # Pongo per evitare la stampa di "Program exited with code X"
61             XOR %EAX, %EAX
62
63             RET
64
```

```

1 # Leggere messaggio da terminale, di sole lettere
2 # Convertire in minuscolo
3 # Stampare messaggio modificato
4 #
5 # Bonus: usare istruzioni stringa
6
7 .GLOBAL _main
8 .INCLUDE "C:/amb_GAS/utility"
9
10 .DATA
11 msg_in:    .FILL 80, 1, 0  # numero, dimensione, valore
12 msg_out:   .FILL 80, 1, 0
13
14 .TEXT
15 _main:     NOP
16
17         # lettura da terminale
18 MOV $80, %CX
19 LEA msg_in, %EBX
20 CALL inline
21
22 CLD
23 LEA msg_in, %ESI
24 LEA msg_out, %EDI
25
26 inizio:    LODSB           # do {
27                         # al = *esi
28
29         # elaborazione
30 CMP $0x41, %AL          # if ( al >= 'A' && al <= 'Z')
31 JB poi
32 CMP $0x5A, %AL
33 JA poi
34
35 OR $0x20, %AL
36
37 poi:       STOSB
38
39         CMP $0x0D , %AL      # while ( al != '\r' )
40 JNE inizio
41
42 fine:      LEA msg_out, %EBX
43 CALL outline
44 RET
45
46

```

```

1 # Leggere messaggio da terminale, di sole lettere
2 # Convertire in maiuscolo
3 # Stampare messaggio modificato
4 #
5 # Bonus: usare istruzioni stringa
6
7 .GLOBAL _main
8 .INCLUDE "C:/amb_GAS/utility"
9
10 .DATA
11 msg_in:    .FILL 80, 1, 0  # numero, dimensione, valore
12 msg_out:   .FILL 80, 1, 0
13
14 .TEXT
15 _main:     NOP
16
17         # lettura da terminale
18 MOV $80, %CX
19 LEA msg_in, %EBX
20 CALL inline
21
22 CLD
23 LEA msg_in, %ESI
24 LEA msg_out, %EDI
25
26 inizio:   LODSB           # do {
27                     # al = *esi
28
29         # elaborazione
30 CMP $0x61, %AL          # if ( al >= 'a' && al <= 'z')
31 JB poi
32 CMP $0x7A, %AL
33 JA poi
34
35 AND $0xDF, %AL
36
37 poi:      STOSB
38
39         CMP $0x0D , %AL        # while ( al != '\r' )
40 JNE inizio
41
42 fine:     LEA msg_out, %EBX
43 CALL outline
44 RET
45
46

```

# Capitolo 11

**Venerdì 23/10/2020**

## 11.1 Esercizio sul fattoriale

L'esercizio richiede di inserire in ingresso un numero che sia tra 0 e 9. Dato un valore  $n$  in ingresso vogliamo calcolare  $n!$  e stamparlo.

**Realizzazione del programma** Il programma può essere realizzato in due modi:

- Per incremento da 2 ad al più 9

```
n = indecimal_byte();
fattoriale = 1;
for(int i = 2; i <= n; i++)
    fattoriale = fattoriale*i;
```

- Per decremento da al più 9 fino a 0.

```
n = indecimal_byte();
fattoriale = 1;
for(int i = n; i <= 2; i--)
    fattoriale = fattoriale*i;
```

**Istruzione MUL** Risulta ovvio che dovremo utilizzare la MUL per numeri naturali. I dubbi che sorgono riguardano il dimensionamento: quanto spazio ci serve?

- Ricordiamo che nell'istruzione MUL il dimensionamento viene deciso dall'operando sorgente indicato: l'altro operando,隐式的, avrà la stessa dimensione dell'operando sorgente esplicito.
- Ricordiamo che nella MUL a 8 bit il risultato viene salvato in unico registro da 16bit, mentre nella MUL a 16 bit e in quella a 32 bit il risultato viene posto in due registri separati (due registri, rispettivamente, da 16 e 32 bit).
- Ricordiamo che con 8 bit possiamo rappresentare i numeri da 0 a 255 (dovreste già saperlo, quanto avete preso a Fondamenti? cit. Corsini), mentre con 16 bit i numeri da 0 a 65535.

- Il risultato più alto che possiamo ottenere, con entrambi gli algoritmi, è 362880 (contenibile in un registro a 32 bit). Tuttavia si osserva che
  - nell'algoritmo con incremento ci serve la MUL a 16 bit;
  - nell'algoritmo con decremento ci serve la MUL a 32 bit!
 entrambe le strategie sono praticabili.

**Spiegazione codice** Lungo il testo del codice sono presenti annotazioni.

- Se adottiamo il fattoriale con decremento utilizziamo la MUL a 32 bit. Sapendo che (*ECX* è l'operando esplicito)

$$EDX\_EAX = EAX * ECX$$

posso porre l'istruzione

**MUL %ECX**

il risultato finirà tutte le volte in *EAX*. Non ho bisogno di interpellare *EDX* poichè il valore più alto possibile (362880) è rappresentabile in 32 bit.

- Se adottiamo il fattoriale con incremento utilizziamo la MUL a 16 bit. Sapendo che (*CX* è l'operando esplicito)

$$DX\_AX = AX * CX$$

posso porre l'istruzione

**MUL %CX**

il risultato finirà tutte le volte in *AX*. Non ho bisogno di interpellare *DX* durante il ciclo: lo farò alla fine quando avrò l'unico risultato rappresentabile con più di 16bit. A quel punto attuo la strategia presente nel codice e spiegata attraverso le note.

```

1 # Leggere numero naturale, sia n, da input
2 # Controllare che sia tra 0 e 9
3 # Calcolare e stampare in output il fattoriale n!
4 #
5 # Extra: organizzare il codice di calcolo del fattoriale come sottoprogramma
6
7 # n! = n * n - 1 * .... * 1
8
9 # n = indecimal_byte()
10 # # if( n > 9 )
11     # return;
12 #
13 # fattoriale = 1;
14 #
15 # for( int i = 2; i <= n; i++)
16     # fattoriale = fattoriale * i;
17 #
18 # 8 bit: da 0 a 255
19 # 16 bit: da 0 a 65535
20 #
21 # 2      x  1      =  2      (8 x 8 -> 16)
22 # 3      x  2      =  6      (8 x 8 -> 16)
23 # 4      x  6      = 24     (8 x 8 -> 16)
24 # 5      x 24     = 120    (8 x 8 -> 16)
25 # 6      x 120    = 720    (8 x 8 -> 16)
26 # 7      x 720    = 5040   (16 x 16 -> 32)
27 # 8      x 5040   = 40320  (16 x 16 -> 32)
28 # 9      x 40320  = 362880 (16 x 16 -> 32)
29 #
30 # for( int i = n; i >= 2; i--) 32 Bit
31     # fattoriale = fattoriale * i;
32 #
33 # 9      x  1      =  9      (8 x 8 -> 16)
34 # 8      x  9      = 72     (8 x 8 -> 16)
35 # 7      x 72     = 504    (8 x 8 -> 16)
36 # 6      x 504    = 3024   (16 x 16 -> 32)
37 # 5      x 3024   = 15120  (16 x 16 -> 32)
38 # 4      x 15120  = 60480  (16 x 16 -> 32)
39 # 3      x 60480  = 181440 (16 x 16 -> 32)
40 # 2      x 181440 = 362880 (32 x 32 -> 64)
41 #
42 # outdecimal(fattoriale); 32 Bit
43 #
44
45 .GLOBAL _main
46 .INCLUDE "C:/amb_GAS/utility"
47
48 .DATA
49
50 n: .BYTE 0
51 risultato: .LONG 1
52
53 msg_1:      .ASCII "Inserire naturale n da tra 0 e 9:\r"
54 msg_2:      .ASCII "Il fattoriale di n (n!) e':\r"
55
56 .TEXT
57
58 _main:        NOP
59
60             LEA msg_1, %EBX

```

```

61      MOV $80, %ECX   STAMPA MSG_1
62      CALL outline
63
64      CALL indecimal_byte
65      CALL newline
66      MOV %AL, n
67
68      MOV $0, %ECX   → PULISCO (VIALE PER QUANTO FARÀ DOPO)
69      MOVB n, %CL    → PONGO L'INPUT E USO PIÙ AVANTI
70
71  SOTOPROGR. [ CALL factorial_inc
72      MOV %EAX, risultato
73
74 fine:          LEA msg_2, %EBX } → STAMPO MSG_2
75      MOV $80, %ECX
76      CALL outline
77
78      MOV risultato, %EAX } STAMPO IL RISULTATO
79      CALL outdecimal_long
80      RET
81
82 # sottoprogramma fattoriale, da n a 2
83 # input: ECX naturale da 0 a 9
84 # output: EAX fattoriale del numero (1 se invalido)
85 # sporca: EDX
86 factorial_dec:
87         MOV $1, %EAX  # farà da risultato e moltiplicando
88
89         # controllo validità
90         CMP $2, %ECX
91         JB fine_factorial_dec  ESCO SE ECX & [2,9]
92         CMP $9, %ECX
93         JA fine_factorial_dec
94
95 ciclo_factorial_dec:
96         CMP $1, %CL  # while( cl > 1 ) {
97         JE fine_factorial_dec
98
99         MUL %ECX  | # edx_eax = eax * ecx
100        DEC %CL
101        JMP ciclo_factorial_dec  #
102
103 fine_factorial_dec:
104         RET
105
106
107 # sottoprogramma fattoriale, da 2 a n
108 # input: ECX naturale da 0 a 9
109 # output: EAX fattoriale del numero (1 se invalido)
110 # sporca: EDX, BX
111 factorial_inc:
112         MOV $1, %AX  # farà da risultato e moltiplicando
113         MOV $0, %DX
114
115 STESSI CONFRONTI [ # controllo validità
116
117         CMP $2, %ECX
118         JB fine_factorial_inc
119         CMP $9, %ECX
120         JA fine_factorial_inc

```

NON MI SERVE CALCOLARE IL FATTORIALE DI 1.

- RIHOANGO NEL CICLO, FINCHÉ CL NON SARÀ UGUALE AD 1.

- OGNI VOLTA APPLICO LA MUL A 32 BIT.

PERCHÉ?

- OGNI VOLTA IL RISULTATO VAI IN BX

- NON TI DEDICO SÌ INTERPELLARE EDX

- USO ECX CHE HA LO STESSO VALORE DI CL (ECX È STATO PULITO ALL'INIZIO)

```

121      MOV $2, %BX
122
123 ciclo_factorial_inc:
124     # do {
125
126     MUL %BX    # dx_ax = ax * bx INITIALIZED IN VARIABLE  
OF INCREMENTATION
127
128 CONFRONTO  
DOPO COME  
NEL DO-WHILE [ CMP %BX, %CX
129         JE fine_factorial_inc
130
131         INC %BX
132         JMP ciclo_factorial_inc # } while( cl > 1 ) APPLICO LA XUL A 16 BIT
133
134 fine_factorial_inc:
135     # edx = ?_rh
136     # eax = ?_rl
137     SHL $16, %EDX    # edx = rh_0
138     MOV %AX, %DX    # edx = rh_rl
139
140     MOV %EDX, %EAX  # eax = rh_rl
141
142

```

- FINO ALL'ULTIMO STEP MI  
BASTA UN SOLO REGISTRO (AX)  
PER GESTIRE I RISULTATI  
PRECEDENTI.

- DX LO INTERPELLO SOLO ALLA  
FINE, COL RISULTATO FINALE

- EDX È OCCUPATO AL PIÙ NELLE PRIME 16 CIFRE  
MENO SIGNIFICATIVE
- EFFETTUO SHIFT E LIBERO LE CIFRE MENO  
SIGNIFICATIVE PER AX
- SPOSTO AX IN DX (DX EQUIVALENTE DELLE PRIME  
16 CIFRE MENO SIGNIFICATIVE DI EDX)
- ADESSO HO TUTTO IL RISULTATO IN EDX
- SPOSTO IN EAX (ABBIANO STABILITO DI PORRE)  
IL RISULTATO LI

## 11.2 Calcolo del binomiale

Il calcolo del binomiale

$$\frac{n!}{k!(n-k)!} \quad n \geq k$$

può essere fatto sfruttando il codice scritto nel problema precedente. Nel codice che segue abbiamo utilizzato la versione del fattoriale con incremento (non per ragioni particolari). Le strategie adottate per il problema sul fattoriale relativamente ai registri (utilizzo del registro con le cifre meno significative) è valida anche per il binomio. Osserviamo che

- Il numeratore è rappresentabile in 32bit.
- Il denominatore è rappresentabile in 32bit

Useremo la DIV a 32 bit: abbiamo un dividendo a 64 bit diviso tra due registri e un divisore esplicito a 32 bit. Il risultato sarà a 32 bit, posto in un unico registro!

```

1 # Leggere due numeri naturali, a e b, da input
2 # Controllare che siano tra 0 e 9, e a >= b
3 # Calcolare e stampare in output il coeff. binomiale ( a b ) = a! / ( b! * (a - b) )
4
5 # a, b => 8 bit
6 # a!, b!, (a - b)! => 32 bit
7 # denom. =>
8     # (a-b) in N
9     # a! >= denom.
10    # => denom. su 32 bit
11
12 # divisore 32 bit => dividendo 64 bit
13
14 #     a_fatt = fattoriale(a)
15 #     b_fatt = fattoriale(b)
16 #     ab_fatt = fattoriale(a - b)
17 #     denom = b_fatt * ab_fatt
18 #     risultato = a_fatt / denom
19
20 .GLOBAL _main
21 .INCLUDE "C:/amb_GAS/utility"
22
23 .DATA
24 a:      .BYTE 0
25 b:      .BYTE 0
26 a_fatt: .LONG 0
27 b_fatt: .LONG 0
28 ab_fatt: .LONG 0
29 denom:  .LONG 0
30 risultato: .LONG 0
31
32 msg_1: .ASCII "Inserire i due naturali a e b, da 0 a 9:\r"
33 msg_2: .ASCII "Il coefficiente binomiale (a b) e':\r"
34 msg_err: .ASCII "Input invalidi\r"
35
36 .TEXT
37
38 _main:
39     NOP
40
41     # lettura a e b
42     LEA msg_1, %EBX
43     MOV $80, %ECX
44     CALL outline
45
46     CALL indecimal_byte
47     CALL newline
48     MOV %AL, a
49
50     CALL indecimal_byte
51     CALL newline
52     MOV %AL, b
53
54     # controllo validita' a e b
55     CMPB $9, a
56     JA wrong_input
57     CMPB $9, b
58     JA wrong_input
59

```

INPUT IN 8 BIT  
- FATTORIALI IN 32BIT (QUINDI ANCHE NUMERATORE)  
- DENOMINATORE IN 32BIT

STessa strategia del problema precedente (GUARDO UN SOLO REGISTRO)

INPUT → SALVO I DATI  
CALCOLATI NEL TEMPO → OUTPUT

\* - SO CHE IL NUM. STA IN 32bit  
- SO CHE IL NUM È MAGGIOR E O UGUALE AL DENOMINATORE  
- SEGUI CHE IL DENOMINA TORE STA IN 32BIT =

STAMPO MSG-1  
CHIEDO a  
CHIEDO b

CONTROLLO CHE LE CONDIZIONI RICHIESTE SIANO RISPETTATE

```

60      MOV B a, %AL
61      MOV B b, %BL
62      CMP %AL, %BL
63      JA wrong_input
64
65      # calcolo dei fattoriali
66      MOV $0, %ECX
67      MOV B a, %CL
68      CALL factorial_inc
69      MOV %EAX, a_fatt
70
71      b!
72      MOV $0, %ECX
73      MOV B b, %CL
74      CALL factorial_inc
75      MOV %EAX, b_fatt
76
77      (a-b)!!
78      MOV $0, %ECX
79      MOV B a, %CL
80      SUB b, %CL # cl -= b => cl = a - b
81      CALL factorial_inc
82      MOV %EAX, ab_fatt
83
84      b!|(a-b))|[ # calcolo del denominatore
85      MOV b_fatt, %EAX
86      MOV ab_fatt, %EBX
87      MUL %EBX # edx_eax = eax * ebx ] MUL A 32 BIT
88
89      # divisione
90      MOV $0, %EDX
91      MOV a_fatt, %EAX
92      MOV denom, %EBX
93      DIV %EBX # EAX = EDX_EAX / EBX ] ISTRUZIONE DIV
94      MOV %EAX, risultato
95
96      # stampa del risultato
97      LEA msg_2, %EBX
98      MOV $80, %ECX
99      CALL outline
100
101      MOV risultato, %EAX
102      CALL outdecimal_long
103      RET
104
105      wrong_input:
106          LEA msg_err, %EBX
107          MOV $80, %ECX
108          CALL outline
109          RET
110
111
112      # sottoprogramma fattoriale, da 2 a n
113      # input: ECX naturale da 0 a 9
114      # output: EAX fattoriale del numero (1 se invalido)
115      # sporca: EDX, BX
116      factorial_inc:
117          MOV $1, %AX # fara' da risultato e moltiplicando
118          MOV $0, %DX
119

```

SE QUALcosa NON È RISAIETATO  
FACCIO JUMP.

CALCOLIAMO I FATTORIALI  
COL CODICE DEL PROBLEMA  
PRECEDENTE

NUKEATORE: a!

b! | a-b))| ISTRUZIONE DIV  
DIV A 32 BIT (DIVISORE A 32 BIT)

STAMPO MSG\_2

STAMPO IL RISULTATO

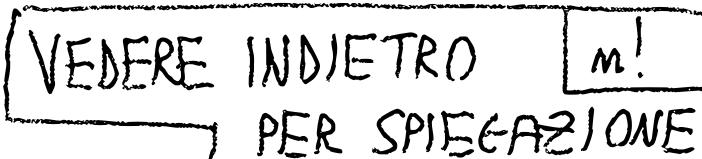
STAMPO MSG\_ERR

CI SI FERMA

```

120         # controllo validita'
121         CMP $2, %ECX
122         JB fine_factorial_inc
123         CMP $9, %ECX
124         JA fine_factorial_inc
125
126         MOV $2, %BX
127
128 ciclo_factorial_inc:
129         # do {
130
131         MUL %BX      # dx_ax = ax * bx
132
133         CMP %BX, %CX
134         JE fine_factorial_inc
135
136         INC %BX
137         JMP ciclo_factorial_inc  # } while( cl > 1 )
138
139 fine_factorial_inc:
140         # edx = ?_rh
141         # eax = ?_rl
142         SHL $16, %EDX      # edx = rh_0
143         MOV %AX, %DX      # edx = rh_rl
144
145         MOV %EDX, %EAX    # eax = rh_rl
146         RET
147

```


 m!