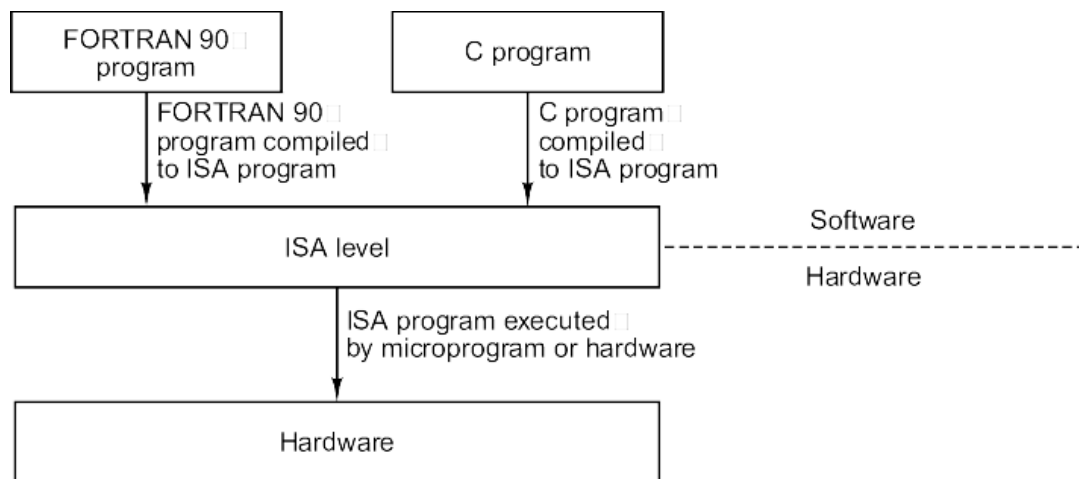


Il livello ISA

Instruction Set Architecture

Il livello **ISA** (Instruction Set Architecture) descrive l'architettura delle istruzioni che la **CPU** è in grado di eseguire in **Hardware** (**Firmware**). Ogni diversa CPU ha un proprio ISA e quindi istruzioni diverse spesso non compatibili tra loro.

Scrivere programmi complessi utilizzando direttamente istruzioni ISA è **difficile** e **spesso inutile**. In quasi tutti i calcolatori è possibile scrivere programmi utilizzando linguaggi di **alto livello** (più orientati all'uomo: esempio C) che vengano compilati (ovvero tradotti in istruzioni ISA) da programmi chiamati **Compilatori**.



Perché eseguire la compilazione e non progettare direttamente macchine in grado di comprendere linguaggi ad alto livello come il C?

I linguaggi di **alto livello** sono spesso molto **complessi** e la definizione di primitive ISA così articolate richiederebbe la realizzazione di **CPU troppo complicate e costose**.

Inoltre un programma in linguaggio di alto livello può essere (teoricamente) **compilato ed eseguito su CPU diverse** semplicemente utilizzando compilatori specifici per le diverse CPU (**portabilità**).

Perché compilare e non interpretare i programmi di alto livello?

Livello ISA e Assembler

Qual è la differenza tra **ISA**, **Assembly language**, **Assembler** e **linguaggio macchina**? Esistono pareri discordanti e notazioni diverse, ma l'importante è capirsi:

- Quando si parla di **Assembly language** si intende un linguaggio costituito da **codici mnemonici** corrispondenti alle istruzioni ISA. In realtà, il linguaggio Assembly fornisce altre **facilitazioni** al programmatore, quali **etichette simboliche** per variabili e indirizzi, primitive per **allocazione** in memoria di variabili, costanti, definizione di **macro**, ... che semplificano il compito al programmatore (vedi **Assembly language Inline**).

Frammento C	Assembly language	ISA (IA-32)
a = 10; b = 20; c = a + b;	mov [a], 0Ah mov [b], 14h mov eax, [a] add eax, [b] mov [c], eax	c7 45 f8 0a 00 00 00 c7 45 ec 14 00 00 00 8b 45 f8 03 45 ec 89 45 e0

- Un programma "semplice" detto **Assembler** (Assemblatore) **traduce** i codici mnemonici nei codici numerici corrispondenti alle istruzioni ISA. L'insieme di questi codici costituisce i programmi eseguibili (.EXE) che possiamo eseguire nei nostri PC.
- Assembler** (ovvero il programma traduttore) viene da molti usato come **sinonimo** di **Assembly language**: anche noi spesso useremo i due termini indifferentemente. **Linguaggio macchina** viene talvolta usato per indicare Assembly language, altre volte per istruzioni ISA.

Perché studiare ISA e Assembler?

- È importante per **capire** veramente il **funzionamento** di una CPU e di un sistema di elaborazione.
- Un programma scritto in linguaggio **Assembly** è solitamente **dalle 2 alle 3 volte più veloce** di un programma analogo scritto in C e compilato!
- L'**ottimizzazione** di piccole porzioni di codice, detta **tuning** (ad. esempio effettuata con Assembler Inline), è un'ottima tecnica per **migliorare radicalmente le prestazioni** di programmi con uno sforzo contenuto.

	Programmer-years to produce the program	Program execution time in seconds
Assembly language	50	33
High-level language	10	100
Mixed approach before tuning		
Critical 10%	1	90
Other 90%	9	10
Total	10	100
Mixed approach after tuning		
Critical 10%	6	30
Other 90%	9	10
Total	15	40

- L'**analisi del codice** prodotto automaticamente da un compilatore ci permette di verificare la presenza di **bug di compilazione** o di comprendere meccanismi complessi (es: passaggio parametri).
- L'Assembly è spesso l'**unico linguaggio** di programmazione per **sistemi industriali embedded** basati su micro-controllori ed è indispensabile per **applicazioni industriali run-time**.

IA-32: ISA dei sistemi x86 a 32 bit

D'ora in avanti ci concentreremo sullo studio dell'**ISA dei processori x86** a 32 bit, (es. **Intel Core**, **AMD Ryzen**), denominata **IA-32**.

Si tratta di un ISA frutto dell'**evoluzione di molte generazioni di CPU**: *mantiene il completo supporto per l'esecuzione di programmi scritti per l'8086 del 1978* e contiene addirittura rimembranze dell'8080, un processore a 8 bit, che a sua volta era basato sul 4004 a 4 bit del 1971!

- Il capostipite della famiglia x86 è l'**8086**, una CPU a 16 bit con 4 registri a 16 bit di uso più o meno generale (**AX, BX, CX, DX**) e altri 4 con utilizzo più specifico (**SI, DI, BP, SP**).
- Una versione più economica, l'**8088**, che differiva sostanzialmente per il bus a 8 bit invece che a 16, fu, come abbiamo visto, la **CPU del PC IBM**.
- Il successore, l'**80286**, era una CPU a 16 bit, che presentava come principale novità un maggiore spazio degli indirizzi di memoria.
- L'**80386** fu la **prima CPU a 32 bit** della famiglia x86: manteneva la compatibilità con le CPU precedenti a 16 bit e disponeva di 8 registri principali a 32 bit (**EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP**).

Tutti i processori successivi hanno sostanzialmente **la stessa ISA**, denominata **IA-32** da Intel. Una CPU IA-32 ha tre **modalità operative** (due delle quali per compatibilità con vecchi modelli a 16 bit):

- **real**: opera fisicamente **come un 8086** (16 bit); tutte le operazioni aggiunte a seguito dell'8088 sono inibite. Quando la CPU opera in questa modalità un errore blocca irrimediabilmente la macchina.
- **virtual-8086**: opera in **emulazione 8086**, ma il sistema operativo crea per ogni processo un ambiente isolato; anche in caso di errore è possibile terminare il processo responsabile senza compromettere il funzionamento del resto del sistema.
- **protected**: opera **come una CPU IA-32** e non come un costoso 8086. È la modalità utilizzata dai sistemi operativi a 32 bit.

IA-32: ISA dei sistemi x86 a 32 bit (2)

Gli *unici cambiamenti significativi* di IA-32 nel corso degli anni sono stati:

- Istruzioni **floating point**, grazie alla **FPU** incorporata a partire da alcune versioni del **80486**, che comunque prima erano disponibili installando un coprocessore esterno (**8087**, **80287**, ...).
- Nuovi **registri e istruzioni specializzate** per applicazioni **multimediali**, grazie alle estensioni **SIMD MMX**, **SSE**, **AVX**, **AVX-512**.
- Estensione **x86-64** (chiamata **AMD64** da AMD e **Intel64** da Intel): i **registri aumentano in ampiezza e numero**, si aggiunge una **nuova modalità operativa** (chiamata **IA-32e** da Intel e **Long** da AMD) utilizzata dai **sistemi operativi a 64 bit**. La nuova modalità consente di eseguire sia **programmi a 32 bit** in modo nativo (**compatibility mode**), sia **programmi a 64 bit** che utilizzano la nuova estensione e vedono uno spazio lineare di indirizzamento a 64 bit (**64-bit mode**).

Modalità operativa		Sistema Operativo	Applicazioni	Spazio indirizzi
Real		16 bit	16 bit	1 MB
Virtual-8086		32 bit	16 bit	1 MB
Protected			32 bit	4 GB
IA-32e / Long	Compatibility mode	64 bit	16 ^(*) / 32 bit	4 GB
	64-bit mode		64 bit	16 EB

(*) La CPU è in grado di eseguire le istruzioni macchina di applicazioni scritte per l'8086 anche in questa modalità operativa, ma in pratica è necessario anche il supporto del sistema operativo. Ad esempio, nel caso di Windows a 64 bit, non è possibile eseguire applicazioni a 16 bit in modo nativo perché il sistema operativo non lo consente.

IA-32: ISA dei sistemi x86 a 32 bit (3)

- Principali caratteristiche dell'ISA dei precursori di x86:

	Intel 4004	Intel 8008	Intel 8080
Anno	1971	1972	1974
Program counter	PC [12 bit]	PC [14 bit]	PC [16 bit]
Registri di base	A (accumulator), 16 index registers [4 bit]		A, B, C, D, E, H, L [8 bit]

- Principali caratteristiche dell'ISA x86 nelle sue evoluzioni da 16 a 64 bit:

	x86-16	x86 (IA-32)	x86-64 (x64, AMD64, Intel 64)
Anno	1978	1985	2003
Program counter	IP [16 bit]	EIP [32 bit]	RIP [64 bit]
Registri di base	AX, BX, CX, DX, SI, DI, BP, SP [16 bit]	EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP [32 bit]	RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8, R9, ..., R15 [64 bit]
Registri FP	FPR0, FPR1, ..., FPR7 [80 bit]		
Registri MMX (dal 1997)		MMX0, ..., MMX7 [64 bit]	
Registri XMM (dal 1999)		XMM0, ..., XMM7 [128 bit]	XMM0, ..., XMM15 [128 bit]
Registri YMM (dal 2011)		YMM0, ..., YMM7 [256 bit]	YMM0, ..., YMM15 [256 bit]
Registri ZMM (dal 2017)		ZMM0, ..., ZMM7 [512 bit]	ZMM0, ..., ZMM31 [512 bit]
Esempi di CPU	8086, 80286	80386, 80486, Pentium	Athlon 64, Core 2, Core i7, Ryzen

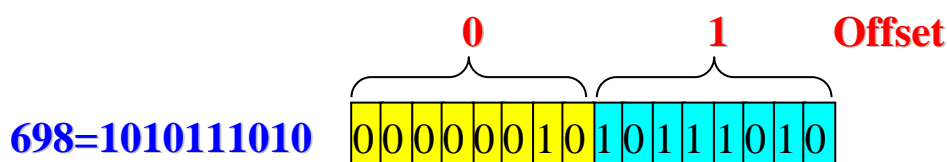
IMPORTANTE

Nelle CPU x86 le informazioni sono memorizzate in modalità **LITTLE ENDIAN** (prima byte meno significativo).

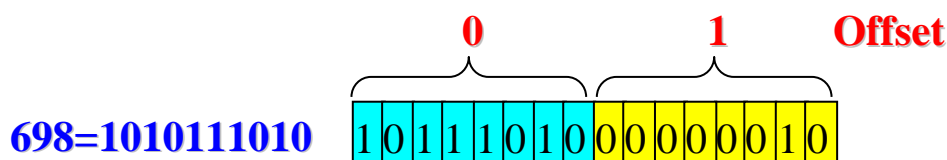
Ordinamento dei byte

Quando la parola contiene più di un byte si pone il problema di come enumerare i byte al suo interno e quindi di come rappresentare i numeri binari che sono memorizzati su più byte.

Big endian: il byte **più significativo (big)** del numero è memorizzato nel byte della parola con **offset minore**. Questa rappresentazione è utilizzata, tra gli altri, dai processori SPARC e Motorola.



Little endian: il byte **meno significativo (little)** del numero è memorizzato nel byte della parola con **offset minore**. Questa rappresentazione è utilizzata, tra gli altri, dai processori Intel e Alpha RISC.



Entrambe le rappresentazioni sono internamente consistenti, il problema si pone quando è necessario scambiare dati tra macchine utilizzando sistemi diversi.

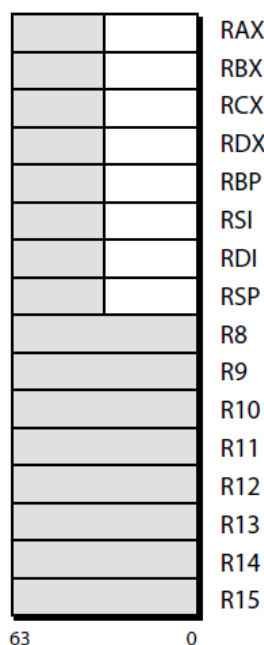
0000001010111010 = 698_{BE}

47618_{LE} = 0000001010111010

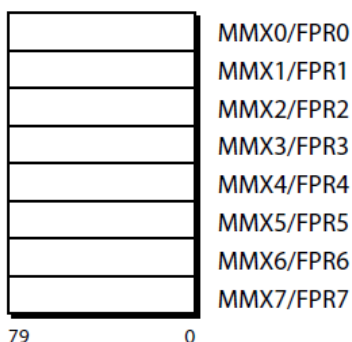
Il problema si avrebbe ugualmente se i dati fossero scambiati in formato ASCII anziché binario?

Registri di IA-32: visione d'insieme

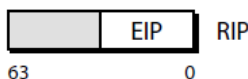
Registri di
uso generale



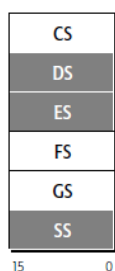
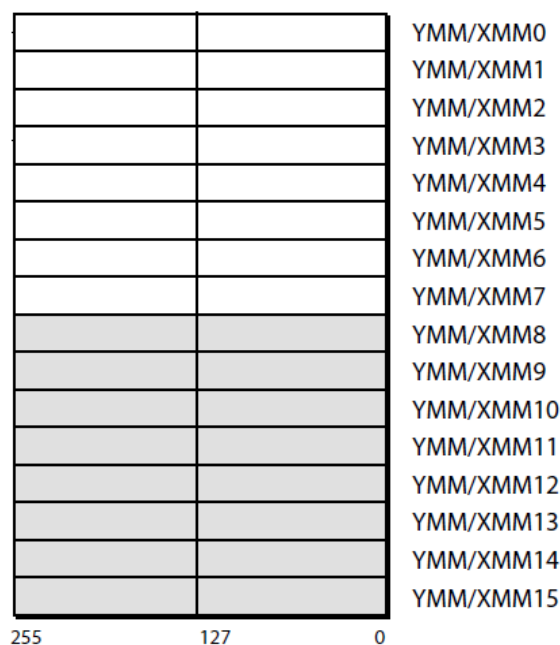
Registri FPU / MMX



Program counter



Registri SSE

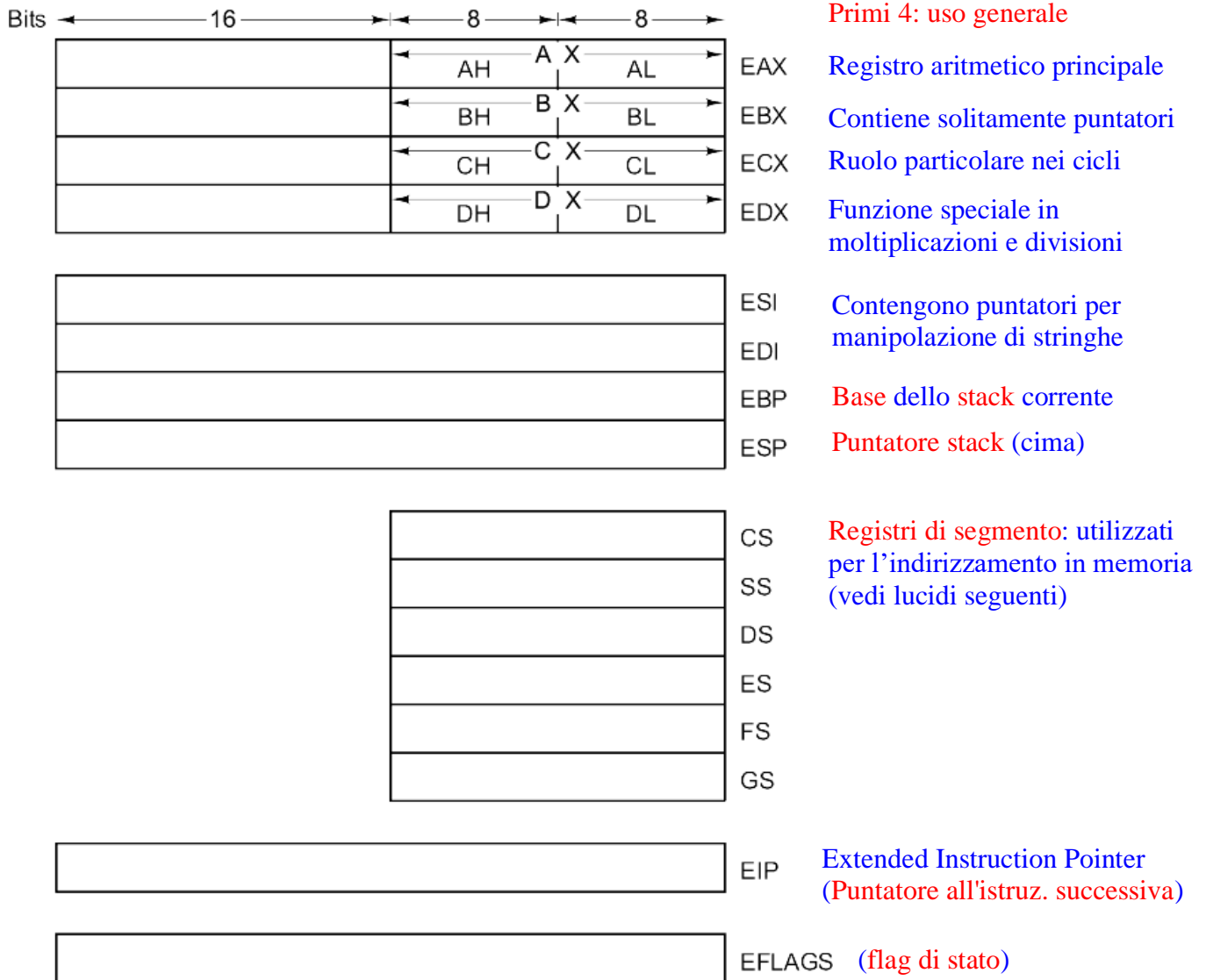


Registri
di segmento

- Disponibili solo in 64 bit mode
- Ignorati dall'hardware in 64 bit mode

N.B. Ulteriori registri sono disponibili con l'estensione AVX-512.

IA-32: registri di base



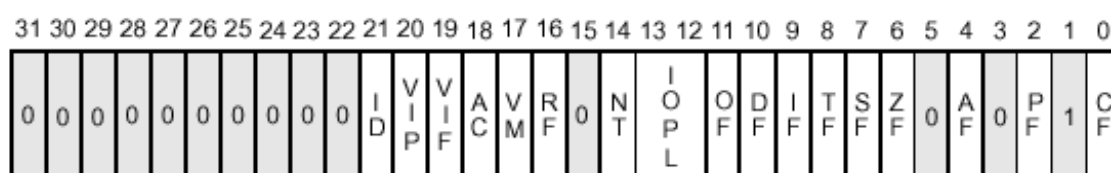
Sono disponibili 8 registri di utilizzo più o meno generale (un numero piuttosto limitato), ma le cose migliorano se si considerano le varie estensioni SIMD e x86-64.

Poter disporre di un numero elevato di registri (vedi RISC) consente di velocizzare l'esecuzione di programmi in quanto è possibile conservare nei registri molte variabili evitando accessi in RAM (che sono più lenti e sprecano cicli).

I **primi 4 registri** possono essere utilizzati a **8** bit (es: AL, AH), a **16** bit (es: AX) o a **32** bit (es: EAX).

IA-32: registri di base (2)

- **EIP** (**I**nstruction **P**ointer): contiene l'indirizzo della prossima istruzione da eseguire. Viene incrementato **automaticamente** durante il fetch delle istruzioni e **modificato** dalle **istruzioni di salto**.
- **EFLAGS** (**b**it **d**i **s**tato): questo registro contiene diversi bit utili sia alla CPU sia al programmatore. I bit principali determinano i cosiddetti **condition code**: vengono scritti a ogni ciclo dell'ALU e riflettono il risultato dell'operazione più recente. Come vedremo, **le istruzioni di saldo condizionale** utilizzano i condition code per determinare se saltare oppure no. I **flag più comuni** sono i seguenti: (**fra parentesi il nome visualizzato nel Visual Studio in debug**):
 - **CF** (**CY**): attivo quando il risultato ha determinato riporto (**carry**).
 - **PF** (**PE**): attivo quando il byte meno significativo del risultato ha "**parità pari**", ovvero numero di "uni" o "zeri" pari.
 - **AF** (**AC**): attivo quando il risultato ha determinato riporto intermedio sul bit 3 (**auxiliary carry**); utile in codifica BCD.
 - **ZF** (**ZR**): attivo quando il risultato è **zero**.
 - **SF** (**PL**): bit **segno**, attivo quando il risultato è negativo.
 - **OF** (**OV**): attivo quando il risultato ha causato **overflow** con operazioni in aritmetica intera **con segno**.
 - **DF** (**UP**): la direzione nelle istruzioni di manipolazione stringhe.

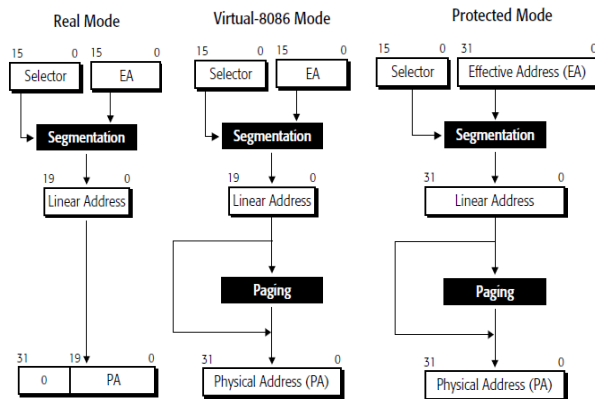


Altri bit di EFLAGS (vedi figura) sono dedicati alla modalità operativa (**real**, **virtual-8086**, **protected**) e a particolari modalità di funzionamento per operazioni di debugging di programmi (esecuzione **step by step**, **interrupt**, ...).

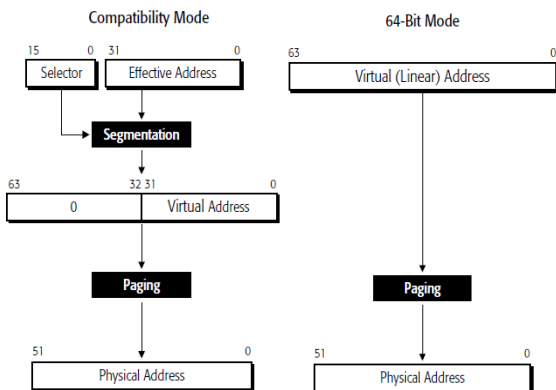
- Esistono inoltre altri **registri di sistema** GDTR, IDTR, LDTR, TR che contengono i puntatori a tabelle di sistema importanti (**es: IDTR = Interrupt Description Table Register**), e altri registri utilizzati per il **debugging** di programmi e per il **supporto** della **cache**.

IA-32: Organizzazione della memoria

Approfondimento: costruzione indirizzi nelle diverse modalità operative

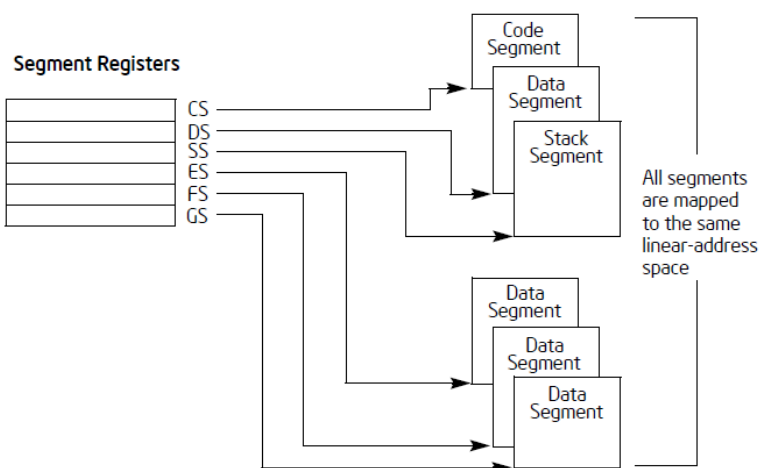


- **Real:** un **registro di segmento** a 16 bit (**CS**, **DS**, ...) è combinato con un registro a 16 bit per ottenere un **indirizzo a 20 bit**. La paginazione non è supportata.
- **Virtual-8086:** l'indirizzo a 20 bit viene determinato come nella modalità Real, ma il sistema operativo può trasformarlo in un **indirizzo a 32 bit** e utilizzare la **paginazione**.
- **Protected:** un registro di segmento a 16 bit è combinato con un registro a 32 bit per ottenere un **indirizzo a 32 bit**. Supporta **modello di memoria segmentato** o **flat** con eventuale **paginazione**.

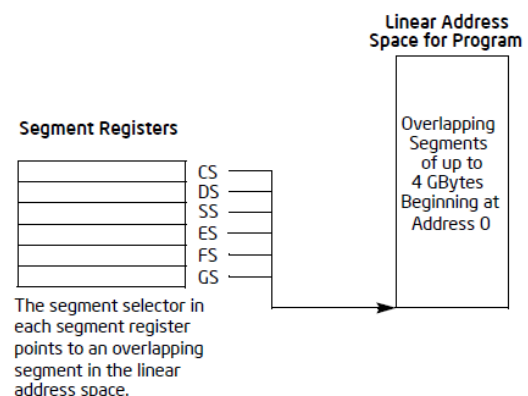


- **x64 – Compatibility Mode:** l'indirizzo a 32 bit viene determinato come nella modalità Protected e convertito a **64 bit** semplicemente considerando i 32 bit alti pari a zero. Gli indirizzi virtuali a 64 bit sono convertiti in indirizzi fisici (con un massimo di 52 bit) mediante **paginazione**.
- **64-bit Mode:** i principali **registri di segmento** (**CS**, **DS**, **ES**, **SS**) sono sostanzialmente **ignorati** e il modello di memoria è **flat**. Gli indirizzi virtuali a 64 bit sono convertiti in indirizzi fisici (con un massimo di 52 bit) mediante **paginazione**.

Possibili modelli di memoria nella modalità Protected

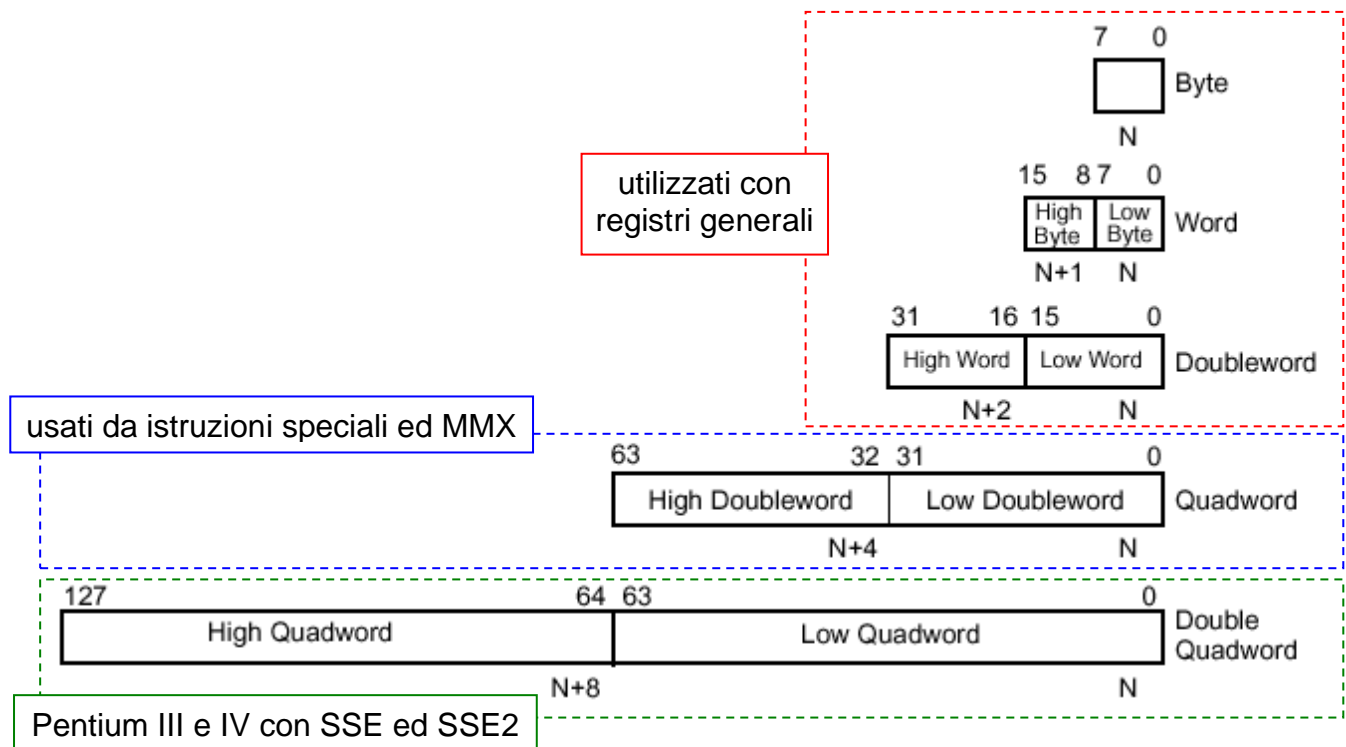


Modello Segmentato

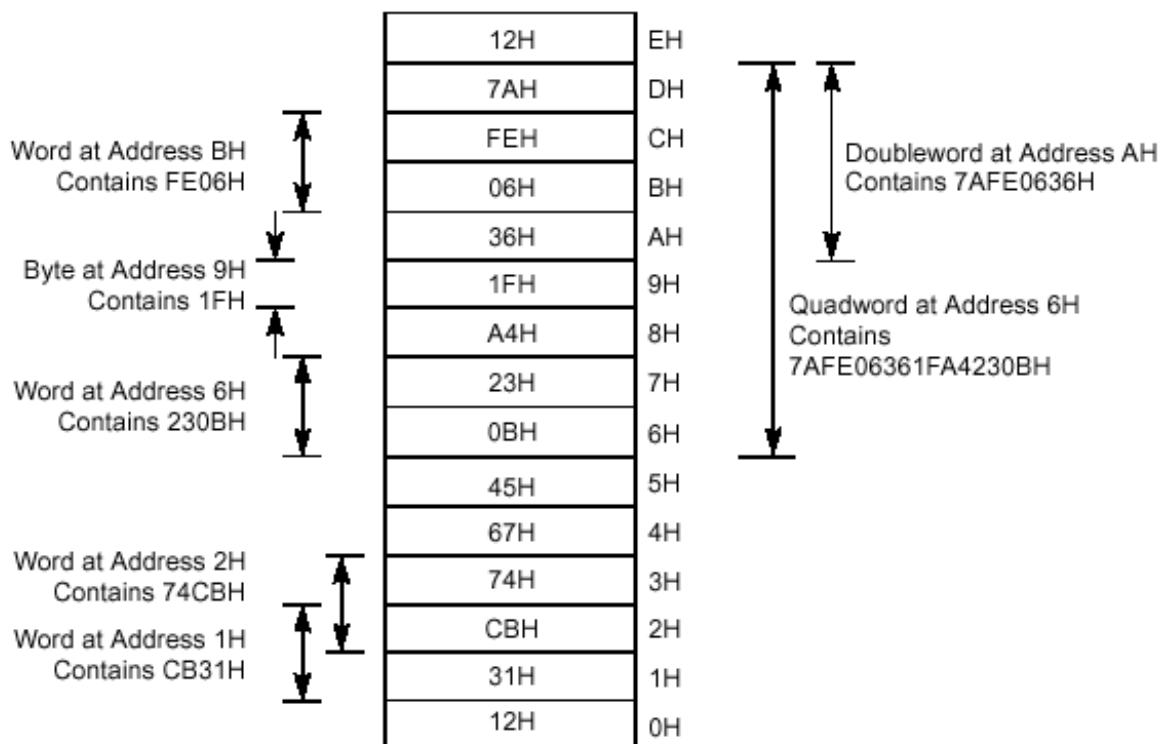


Modello Flat (es. Windows/Linux 32bit)

IA-32: tipi di dati (1)



I dati in memoria: **attenzione, sono CPU Little Endian!**

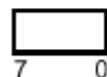


IA-32: tipi di dati (2)

Attenzione alla differenza tra **aritmetica**:

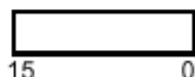
- **unsigned** (solo interi positivi)
- **signed** (interi positivi e negativi memorizzati in complemento a 2)

da 0 a 255



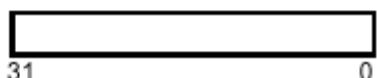
Byte Unsigned Integer

da 0 a 65.535



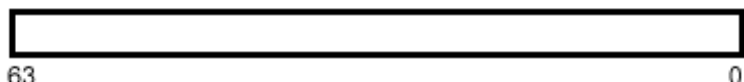
Word Unsigned Integer

da 0 a $2^{32}-1 = 4,294,967,295$



Doubleword Unsigned Integer

da 0 a $2^{64}-1$



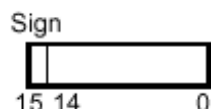
Quadword Unsigned Integer

da -128 a +127



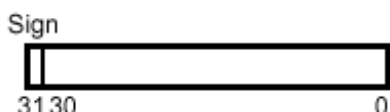
Byte Signed Integer

da -32.768 a 32.767



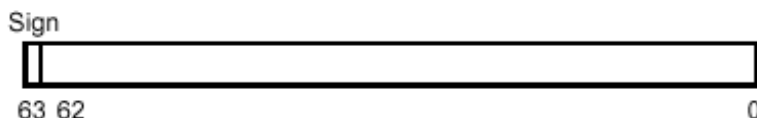
Word Signed Integer

da -2^{31} a $2^{31}-1$



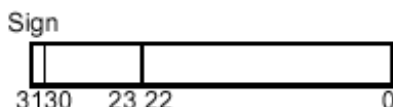
Doubleword Signed Integer

da -2^{63} a $2^{63}-1$



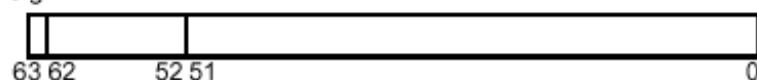
Quadword Signed Integer

da $\pm 1,18 \cdot 10^{-38}$ a $\pm 3,4 \cdot 10^{38}$



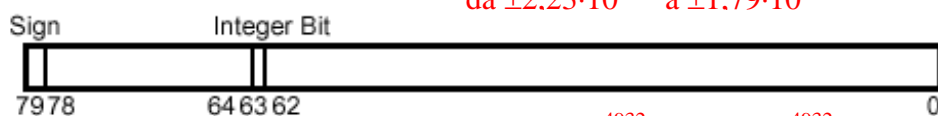
Single Precision Floating Point

Sign



Double Precision Floating Point

da $\pm 2,23 \cdot 10^{-308}$ a $\pm 1,79 \cdot 10^{308}$



Double Extended Precision Floating Point

da $\pm 3,37 \cdot 10^{-4932}$ a $\pm 1,18 \cdot 10^{4932}$

IA-32: modi di indirizzamento (1)

Gran parte delle istruzioni ISA consentono di caricare/salvare i dati attraverso i registri e la memoria. Le **modalità di reperimento dei dati** sono definite dai **modi di indirizzamento**.

Consideriamo l'istruzione **MOV** che viene utilizzata per copiare un valore da una **sorgente** a una **destinazione**:

```
MOV DST, SRC
```

DST e **SRC** vengono chiamati **operandi** dell'istruzione; esistono istruzioni senza operandi, istruzioni con un solo operando e istruzioni a due o più operandi.

Un **operando può specificare elementi diversi**: un registro, una costante, un indirizzo di memoria semplice, un indirizzo di memoria al quale è sommato uno scostamento, ...

Esempi:

```
MOV EAX,10           // immediato
MOV EAX,EBX          // registro
MOV EAX,DS:[10345467h] // memoria
MOV EAX,[ECX]         // memoria
MOV EAX,[ECX+2]       // memoria
...
```

IA-32 dispone di vari **modi di indirizzamento**, ciascuno dei quali specifica le modalità di reperimento di operandi da parte della CPU.

Grazie al programma assembler, possiamo anche utilizzare **nomi simbolici** per **variabili ed indirizzi**; pertanto, se l'indirizzo 10345467h precedente fosse l'indirizzo della variabile **pippo**, potremmo caricare il valore di **pippo** in **EAX** scrivendo:

```
MOV EAX,pippo        // indirizzamento in memoria
```

IA-32: modi di indirizzamento (2)

Indirizzamento immediato

L'operando contiene **direttamente un valore** costante; la lunghezza del valore (1, 2, o 4 byte) dipende dal tipo di operazione e dai registri coinvolti.

```
MOV AL,10           // carica il numero 10 in AL
MOV AH,10h          // carica il numero esadecimale 10 in AH
MOV AH,10100101b    // carica il numero binario 10100101 in AH
MOV AX,0d3c5h       // carica il numero esadecimale d3c5 in AX
MOV EAX,104ed3c5h   // carica il numero esadec. 104ed3c5 in EAX

MOV AX,0d3c5001ah   // ERRORE!!!
```

Per **caricare la costante 0** in un registro (azzeramento di un registro) invece di scrivere

```
MOV EAX,0
```

è preferibile:

```
XOR EAX,EAX
```

in quanto l'operazione XOR (XOR bit a bit) non richiede il caricamento di alcun operando dalla memoria.

Indirizzamento dei registri

L'operando contiene l'indicazione di uno specifico **registro** della CPU.

```
MOV AL,AH           // carica in AL il contenuto di AH
MOV CX,AX            // carica in CX il contenuto di AX
MOV ECX,EBX          // carica in ECX il contenuto di EBX
```

Attenzione alle dimensioni degli operandi (devono essere le stesse)!

IA-32: modi di indirizzamento (3)

Indirizzamento in memoria: l'operando **specifica un indirizzo** di memoria che viene indicato come segue.

$$\text{Indirizzo} = \text{Segmento} : \text{Offset}$$

Il **Segmento** può essere specificato con uno dei registri di segmento (**CS**, **DS**, **ES**, **SS**, **FS**, **GS**). Se non è indicato esplicitamente, nella maggior parte dei casi viene implicitamente utilizzato **DS**.

L'**Offset** (chiamato anche **Effective Address**) è calcolato al momento dell'esecuzione dell'istruzione, utilizzando uno o più dei componenti della formula seguente:

$$\text{Offset} = \text{Base} + (\text{Index} \times \text{Scale}) + \text{Displacement}$$

dove:

- **Displacement** è una costante; può essere omessa.
- **Base**, se presente, è un registro.
- **Index**, se presente, è un registro.
- **Scale** è una costante (2, 4 o 8) moltiplicata per **Index**; può essere omessa (**Scale** = 1).

Base		Index		Scale		Displacement
EAX EBX ECX EDX ESP EBP ESI EDI	+	(EAX EBX ECX EDX EBP ESI EDI)	*	(1 2 4 8)	+	None 8-bit 16-bit 32-bit

Alcuni esempi

Nell'istruzione seguente, l'**Offset** utilizza solo la componente **Displacement**: questo **indirizzamento** viene anche chiamato **assoluto** o **diretto**.

```
MOV AL,DS:[104532a0h] // carica in AL il byte all'indirizzo
                      // DS:104532a0
```

Nell'istruzione seguente, l'**Offset** utilizza solo la componente **Base**: questo **indirizzamento** viene anche chiamato **indiretto**.

```
MOV EAX,[ECX] // carica in EAX la doubleword all'indirizzo
              // DS:<valore contenuto in ECX>, DS è implicito)
```

Attenzione: in caso di omissione delle [], l'istruzione è comunque valida ma la semantica completamente differente!

Indirizzamento in memoria

- Come già visto, con l'assemblatore è possibile utilizzare **nomi simbolici**.

```
MOV AX, pippo // carica in AX la word contenuta nella
               // variabile pippo
```

Supponendo che la variabile sia memorizzata a partire dall'indirizzo DS:12345678h, l'assemblatore trasforma l'istruzione precedente in:

```
MOV AX, DS:[12345678h]
```

- Nell'esempio seguente, l'indirizzo di memoria è determinato a partire da un **valore assoluto** (ad esempio l'**indirizzo iniziale di un vettore**), a cui viene sommato il contenuto di un **registro** che è usato come **indice**.

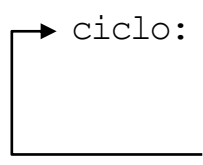
```
MOV EAX, [1000h][ECX] // copia in EAX la doubleword
                      // dall'indirizzo di memoria
                      // DS:1000h + <contenuto di ECX>
```

Se l'indirizzo DS:1000h corrisponde al nome simbolico Vettore, l'assemblatore consente di scrivere la seguente istruzione più leggibile:

```
MOV EAX, Vettore[ECX] // copia in EAX la doubleword
                      // dall'indirizzo di memoria
                      // DS:Vettore + <contenuto di ECX>
```

In questo modo, ad esempio, è possibile accedere a tutti gli elementi di un vettore di byte:

```
      XOR ECX, ECX
ciclo: MOV AL, Vettore[ECX]
      ...
      INC ECX
      JMP ciclo
```



Come fare ad accedere agli elementi di un vettore di doubleword?

Indirizzamento in memoria (2)

- La combinazione di **Index** e **Scale** consente di accedere efficientemente a vettori con **elementi di 2/4/8 byte**. Nell'esempio seguente si accede a un vettore di **doubleword**:

```
        XOR ECX, ECX
ciclo:  MOV EAX, Vettore[ECX*4]
        ...
        INC ECX
        JMP ciclo
```

In alternativa potrei pensare di non utilizzare Scale e di incrementare ECX di 4 unità con un'istruzione ADD ECX, 4. Perché non conviene farlo?

- E se l'**indirizzo** del primo elemento del vettore non è noto a priori, ma è contenuto a sua volta in una **variabile** (es. un **puntatore in C**)? Si può utilizzare **Base**. Nell'esempio seguente supponiamo che la variabile **punt_vet** contenga l'indirizzo del vettore:

```
        MOV EBX, punt_vet
        XOR ECX, ECX
ciclo:  MOV EAX, [EBX][ECX*4]
        ...
        INC ECX
        JMP ciclo
```

Copia in EBX l'indirizzo del vettore

Utilizza EBX come Base, ECX come Index

- La combinazione **Displacement + Base + Index × Scale** può anche essere utilizzata per accedere agli elementi di una **matrice**: supponiamo che una matrice di **50x100 doubleword** (50 righe, 100 colonne) sia dichiarata come variabile globale **matrix** in un C e si voglia accedere all'elemento che si trova alla **riga** il cui indice è in **EDX** e alla **colonna** in **ECX**. La Base **EBX** contiene il numero di byte da “saltare” per posizionarsi all'inizio della riga desiderata (**dimensione di una riga: 100 elementi x 4 byte = 400 byte**).

```
MOV EDX, ...    EDX contiene l'indice della riga (0-based)
MOV ECX, ...    ECX contiene l'indice della colonna (0-based)
```

```
IMUL EBX, EDX, 400 EBX = EDX*400 (quanti byte “saltare” per
                   posizionarsi all'inizio della riga EDX)
```

```
MOV EAX, matrix[EBX][ECX*4] Copia in EAX il valore dell'elemento
                              alla riga EDX e colonna ECX
```

Istruzioni IA-32

Una CPU IA-32, come in genere tutte le CPU di categoria CISC, è dotata di **molte istruzioni** diverse che possono essere classificate in:

- Copie e spostamento di valori
- Aritmetica intera
- Operazioni logiche e spostamento di bit
- Istruzioni di Test e di Salto
- Manipolazione di stringhe
- Unità Floating Point (aritmetica in virgola mobile)
- MMX
- Supporto sistema operativo
- Controllo I/O
- ...

Ogni istruzione, può essere utilizzata in **modalità diverse** a seconda dei modi di indirizzamento. Esistono inoltre **limitazioni** che impediscono l'utilizzo di certi registri con determinate istruzioni, o che **impongono** un certo ordine di esecuzione di istruzioni.

Il "**bravo programmatore**", utilizza come riferimento i manuali del SET di istruzioni ISA messi a disposizione dal fornitore. Nel nostro caso specifico, Intel mette a disposizione (anche on-line) i manuali IA-32. Si tratta di documentazione completa di tutti i possibili dettagli e quindi abbastanza complessa ... **ma sicuramente molto utile e spesso insostituibile!**

Copie e spostamento di valori

MOV DST, SRC	Copia SRC in DST
PUSH SRC	Mette SRC sulla cima dello stack
POP DST	Preleva una parola dalla cima dello stack
XCHG DS1, DS2	Scambia DS1 e DS2
LEA DST, SRC	Carica l'indirizzo di SRC in DST
CMOV DST, SRC	Copia condizionata di un valore

Istruzioni di copia e spostamento

MOV DST, SRC: copia un valore dall'operando **SRC** all'operando **DST**. Abbiamo già visto alcuni esempi nei lucidi precedenti.

- **SRC** e **DST** devono avere la **stessa dimensione**;
- Per **SRC** si può utilizzare indirizzamento **immediato**, **registro**, o **memoria**;
- Per **DST** si può utilizzare indirizzamento **registro** o **memoria**, ma *non è possibile utilizzare un indirizzamento in memoria per entrambi gli operandi nella stessa istruzione*, ad esempio non è scrivere un'istruzione del tipo **MOV pippo,pluto** che copia valori da memoria a memoria.

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
88 /r	MOV r/m8, r8	MR	Valid	Valid	Move r8 to r/m8.
REX + 88 /r	MOV r/m8 ¹ , r8 ¹	MR	Valid	N.E.	Move r8 to r/m8.
89 /r	MOV r/m16, r16	MR	Valid	Valid	Move r16 to r/m16.
89 /r	MOV r/m32, r32	MR	Valid	Valid	Move r32 to r/m32.
REX.W + 89 /r	MOV r/m64, r64	MR	Valid	N.E.	Move r64 to r/m64.
8A /r	MOV r8, r/m8	RM	Valid	Valid	Move r/m8 to r8.
REX + 8A /r	MOV r8 ¹ , r/m8 ¹	RM	Valid	N.E.	Move r/m8 to r8.
8B /r	MOV r16, r/m16	RM	Valid	Valid	Move r/m16 to r16.
8B /r	MOV r32, r/m32	RM	Valid	Valid	Move r/m32 to r32.
REX.W + 8B /r	MOV r64, r/m64	RM	Valid	N.E.	Move r/m64 to r64.
8C /r	MOV r/m16, Sreg ²	MR	Valid	Valid	Move segment register to r/m16.
8C /r	MOV r16/r32/m16, Sreg ²	MR	Valid	Valid	Move zero extended 16-bit segment register to r16/r32/m16.
REX.W + 8C /r	MOV r64/m16, Sreg ²	MR	Valid	Valid	Move zero extended 16-bit segment register to r64/m16.
8E /r	MOV Sreg, r/m16 ²	RM	Valid	Valid	Move r/m16 to segment register.
REX.W + 8E /r	MOV Sreg, r/m64 ²	RM	Valid	Valid	Move lower 16 bits of r/m64 to segment register.
A0	MOV AL, moffs8 ³	FD	Valid	Valid	Move byte at (seg:offset) to AL.
REX.W + A0	MOV AL, moffs8 ³	FD	Valid	N.E.	Move byte at (offset) to AL.
A1	MOV AX, moffs16 ³	FD	Valid	Valid	Move word at (seg:offset) to AX.
A1	MOV EAX, moffs32 ³	FD	Valid	Valid	Move doubleword at (seg:offset) to EAX.
REX.W + A1	MOV RAX, moffs64 ³	FD	Valid	N.E.	Move quadword at (offset) to RAX.
A2	MOV moffs8, AL	TD	Valid	Valid	Move AL to (seg:offset).
REX.W + A2	MOV moffs8 ¹ , AL	TD	Valid	N.E.	Move AL to (offset).
A3	MOV moffs16 ³ , AX	TD	Valid	Valid	Move AX to (seg:offset).
A3	MOV moffs32 ³ , EAX	TD	Valid	Valid	Move EAX to (seg:offset).
REX.W + A3	MOV moffs64 ³ , RAX	TD	Valid	N.E.	Move RAX to (offset).
B0+ rb ib	MOV r8, imm8	OI	Valid	Valid	Move imm8 to r8.
REX + B0+ rb ib	MOV r8 ¹ , imm8	OI	Valid	N.E.	Move imm8 to r8.
B8+ rw iw	MOV r16, imm16	OI	Valid	Valid	Move imm16 to r16.
B8+ rd id	MOV r32, imm32	OI	Valid	Valid	Move imm32 to r32.
REX.W + B8+ rd io	MOV r64, imm64	OI	Valid	N.E.	Move imm64 to r64.
C6 /0 ib	MOV r/m8, imm8	MI	Valid	Valid	Move imm8 to r/m8.
REX + C6 /0 ib	MOV r/m8 ¹ , imm8	MI	Valid	N.E.	Move imm8 to r/m8.
C7 /0 iw	MOV r/m16, imm16	MI	Valid	Valid	Move imm16 to r/m16.
C7 /0 id	MOV r/m32, imm32	MI	Valid	Valid	Move imm32 to r/m32.
REX.W + C7 /0 id	MOV r/m64, imm32	MI	Valid	N.E.	Move imm32 sign extended to 64-bits to r/m64.

Istruzioni di copia e spostamento (2)

PUSH e **POP**: mettono e tolgono parole dalla cima dello stack.

Lo stack è una parte della memoria utilizzata in genere per:

- la valutazione di **espressioni aritmetiche**
- la memorizzazione di **variabili locali**
- la chiamata di **sottoprogrammi**

Attenzione lo STACK **cresce verso il basso** (ovvero verso indirizzi più piccoli), pertanto una PUSH causa (oltre alla copia) anche il decremento di **ESP** e una POP l'incremento di ESP.

Esempio:

```
...                // ESP = 0x0023F7D8
PUSH EAX           // ESP = 0x0023F7D4
PUSH BX            // ESP = 0x0023F7D2
PUSH WORD PTR 0x10 // ESP = 0x0023F7D0
POP AX             // ESP = 0x0023F7D2, AX = 0x0010

ADD ESP, 6         // Necessario ripristinare SP = 0x0023F7D8
                  // prima del termine programma
```

Istruzioni di copia e spostamento (3)

- **XCHG DS1, DS2**: scambia il contenuto di DS1 e DS2 in un'unica operazione.

```
XCHG EAX,EBX      // scambia il contenuto di EAX con EBX  
  
XCHG EBX,pippo    // scambia il contenuto di EBX con quello  
                  della variabile pippo
```

Esempio:

```
MOV  AX, 5         // AX = 5  
MOV  BX, 4         // BX = 4  
XCHG AX,BX        // AX = 4 e BX = 5
```

Quante operazioni MOV sono necessarie per implementare XCHG?

- **LEA DST, SRC**: carica in DST (un registro di solito a 32 bit) l'Offset di SRC (un indirizzamento in memoria del tipo Segment:Offset, nell'assembly inline spesso si utilizza il nome di una variabile). LEA è l'acronimo di *Load Effective Address*.

```
LEA EAX,[10456de4h] // EAX = 10456de4  
LEA EAX,pippo       // EAX = offset dell'indirizzo di pippo
```

Esempio:

```
LEA EAX,pippo       // Carica in EAX l'indirizzo di pippo  
MOV [EAX],10        // pippo = 10
```

Che effetto ha l'istruzione seguente?

```
LEA EAX, [EBX*2+10]
```

Con un'unica istruzione esegue: $EAX = EBX \times 2 + 10$!

Infatti, è come se LEA **eliminasse le parentesi quadre dal secondo operando**. Questo strano costrutto è utilizzato talvolta per ottimizzare al massimo il codice; si sfruttano cioè le peculiarità del modo di indirizzamento con offset per eseguire operazioni aritmetiche.

Istruzioni di copia e spostamento (4)

- **CMOV_{cc} DST, SRC**: come MOV ma la copia viene eseguita **solo se** la condizione **cc** è vera. La condizione **cc** viene determinata a partire dal valore dei bit (**flag**) del registro **EFLAGS**. Si faccia riferimento alle istruzioni di salto condizionale (riportate nel seguito).

Esempio:

```
CMP  AX,BX          // Confronta AX e BX se sono uguali -> ZF = 1
CMOVZ CX,DX         // Se ZF=1 (AX era uguale a BX) -> CX = DX
```

Questa istruzione risulta talvolta **molto utile per evitare di utilizzare salti condizionali** che in genere deteriorano le prestazioni in quanto rendono **inefficace il pre-fetching** (come vedremo nel seguito).

Aritmetica intera

ADD DST, SRC	Somma SRC a DST
SUB DST, SRC	Sottrae SRC a DST
MUL SRC	Moltiplica EAX per SRC (senza segno)
IMUL SRC	Moltiplica EAX per SRC (con segno)
DIV SRC	Dividi EDX:EAX per SRC (senza segno)
IDIV SRC	Dividi EDX:EAX per SRC (con segno)
INC DST	Incrementa DST di 1
DEC DST	Decrementa DST di 1
NEG DST	Nega DST; $DST = 0 - DST$

- **ADD DST, SRC**: esegue la somma di DST e SRC; il risultato è in DST il cui valore iniziale viene quindi sovrascritto.

Esempio:

```
MOV  EAX,5          // Carica 5 in EAX
ADD  EAX,pippo      // EAX = EAX + pippo
```

In base al risultato sono **impostati i flag**: **OF**, **SF**, **ZF**, **AF**, **CF**, e **PF**

Istruzioni aritmetiche

- **SUB DST, SRC**: esegue la **sottrazione** DST-SRC e memorizza il risultato in DST il cui valore iniziale viene quindi sovrascritto.

Esempio:

```
MOV EAX, 15          // Carica 15 in EAX
SUB EAX, 20          // EAX = -5 (in complemento a 2)
NEG EAX              // EAX = 5
```

In base al risultato sono **impostati i flag**: **OF**, **SF**, **ZF**, **AF**, **CF**, e **PF**

- **MUL SRC**: esegue una **moltiplicazione** senza segno.

Instruction	Description
MUL <i>r/m8</i>	Unsigned multiply ($AX \leftarrow AL * r/m8$)
MUL <i>r/m16</i>	Unsigned multiply ($DX:AX \leftarrow AX * r/m16$)
MUL <i>r/m32</i>	Unsigned multiply ($EDX:EAX \leftarrow EAX * r/m32$)

Come mostrato nella tabella sopra riportata questa operazione si comporta **in modo diverso a seconda della dimensione dell'operando SRC**:

Se l'operando è di **8 bit**, la moltiplicazione è eseguita tra AL e SRC e il risultato copiato in AX; se l'operando è di **16 bit** AX viene moltiplicato per SRC e il risultato (32 bit) è memorizzato in DX:AX il che significa che in DX sono contenuti i 16 bit più significativi e in AX i 16 bit meno significativi; infine se l'operando è di **32 bit** EAX viene moltiplicato per SRC e si fa uso oltre che di EAX anche di EDX per memorizzare il risultato (64 bit).

Esempio:

```
MOV EAX, 80000000h    // Carica 80000000h in EAX
MOV EBX, 2h           // Carica 2h in EBX
MUL EBX               // EDX:EAX = EAX * EBX = 100000000
                      // -> EDX = 1
                      // -> EAX = 0
```

NOTA: non è possibile usare un operando immediato per SRC.

Istruzioni aritmetiche (2)

aperta parentesi

*Nel caso in cui l'operando SRC indichi un indirizzo di memoria, **come specificare la dimensione 8, 16 o 32 bit dell'operando?***

Fino ad ora infatti la dimensione è stata sempre implicitamente determinata dai registri coinvolti, ma in questo caso **non è possibile** ...

Il programma assembler accetta davanti agli indirizzi **i seguenti modificatori di tipo:**

BYTE PTR
WORD PTR
DWORD PTR

che indicano rispettivamente che l'indirizzo fornito specifica un operando byte (8 bit), word (16 bit) o double word (32 bit).

Esempio:

```
WORD pippo = 0x0102; // dichiarazione in linguaggio C (0x  
                      indica numero esadecimale)
```

...

```
MOV EAX, 2h           // Carica 2h in EAX  
MUL BYTE PTR pippo    // AX = 4h (pippo è memor. little endian)
```

```
MOV EAX, 2h           // Carica 2h in EAX  
MUL WORD PTR pippo    // DX:AX = (0:204h)
```

```
MOV EAX, 2h           // Carica 2h in EAX  
MUL pippo              // Se non specifico un modificatore in  
                      questo caso l'assembler guardando  
                      la dimensione della variabile pippo si  
                      comporta come se avessi specificato  
                      WORD. In generale questo non è  
                      possibile infatti l'accesso alla  
                      memoria potrebbe avvenire in una zona  
                      "non strutturata".
```

chiusa parentesi

Istruzioni aritmetiche (3)

- **IMUL**: esegue moltiplicazione intera con segno (gli operandi sono in complemento a 2). A differenza di MUL il cui formato prevede un solo operando, **IMUL prevede tre formati**:

1. **IMUL SRC**
2. **IMUL DST, SRC** // $DST = DST * SRC$
3. **IMUL DST, SRC1, SRC2** // $DST = SRC1 * SRC2$

Nel primo caso il funzionamento è analogo a MUL per quanto riguarda i registri utilizzati; nel secondo caso SRC può essere **anche un valore immediato**; nel terzo caso SRC2 è **obbligatoriamente un valore immediato**.

Attenzione: possono non essere sufficienti n bit per memorizzare il risultato della moltiplicazione di due operandi a n bit! **Controllare il valore del flag OF (overflow)**!

Esempio:

```
MOV EAX,10000000h // Carica 10000000h in EAX
IMUL EBX,EAX,16    // risultato = 100000000h, EBX = 0, OF = 1!
```

- **DIV SRC**: **divisione senza segno**; analogamente a MUL si comporta in modo diverso in base alla dimensione dell'operando SRC (**divisore**). In particolare il **divisore**, il **quoziente** e il **resto** sono prelevati/scritti diversamente in base alla dimensione 8, 16 o 32 bit di SRC (vedi tabella).

Instruction	Description
DIV <i>r/m8</i>	Unsigned divide AX by <i>r/m8</i> , with result stored in AL ← Quotient, AH ← Remainder
DIV <i>r/m16</i>	Unsigned divide DX:AX by <i>r/m16</i> , with result stored in AX ← Quotient, DX ← Remainder
DIV <i>r/m32</i>	Unsigned divide EDX:EAX by <i>r/m32</i> , with result stored in EAX ← Quotient, EDX ← Remainder

NOTA: su **divisione per 0** → errore run-time (eccezione **INT 0**),
se il risultato è fuori range, errore run-time **overflow** (eccezione **INT 4**).

Istruzioni aritmetiche (4)

- **IDIV**: equivalente a DIV ma gli operandi sono con segno:

Esempio:

```
MOV EAX,100    // Carica 100 in EAX
CDQ            // Converte (estendendo il segno) la DWORD EAX
               // nella QWORD EDX:EAX
MOV EBX,-3     // Carica -3 in EBX
IDIV EBX       // EAX = -33 (quoziente) , EDX = 1 (resto)
```

- **INC DST**: incrementa di 1 il valore specificato da DST (senza alterare il flag CF). Utilizzato in genere nei cicli.

```
INC EAX        // EAX = EAX + 1
INC pippo      // pippo = pippo + 1
```

Lo stesso risultato si otterrebbe con **ADD DST,1** ma **INC DST** è più efficiente perché non richiede di caricare operandi immediati.

NOTA: se DST ha raggiunto il valore massimo (es. `EAX = ffffffffh`) l'istruzione di incremento causa traboccamento e quindi la destinazione assume valore 0.

D'altro canto, il flag OF non viene impostato in questo caso, esattamente come avviene con ADD (si veda la definizione di overflow nella somma fra numeri binari). Siccome nemmeno il flag CF viene alterato, l'unico flag utilizzabile per verificare il traboccamento è ZF.

- **DEC DST**: decrementa di 1 il valore specificato da DST (senza alterare il flag CF). Utilizzato in genere nei cicli.

```
DEC EAX        // EAX = EAX - 1
DEC pippo      // pippo = pippo - 1
```

Lo stesso risultato si otterrebbe con **SUB DST,1** ma **DEC DST** è più efficiente perché non richiede di caricare operandi immediati.

Per verificare underflow (traboccamento sotto lo zero) può essere utilizzato il flag di segno SF.

Operazioni sui bit

AND DST, SRC	AND bit a bit tra SRC e DST
OR DST, SRC	OR bit a bit tra SRC e DST
XOR DST, SRC	XOR bit a bit tra SRC e DST
NOT DST	Nega bit a bit DST
SAL/SAR DST, #	Shift aritm. a sinistra/destra di # bit in DST
SHL/SHR DST, #	Shift logico a sinistra/destra di # bit in DST
ROL/ROR DST, #	Rotazione a sinistra/destra di # bit in DST

- **AND/OR/XOR DST, SRC**: AND/OR/XOR logico bit a bit; il risultato viene sovrascritto su DST.

Esempi:

```
AND EAX, 00001111h    // EAX = EAX AND 00001111h
AND pippo, EBX         // pippo = pippo AND EBX
OR EAX, Vettore[EBX*2+4] // EAX = EAX OR double word
                        all'indirizzo Vettore+EBX*2+4
XOR EAX, EAX           // EAX = EAX AND EAX -> AZZERA EAX
XOR EBX, 0a0b0c0dh    // EBX = EBX XOR 0a0b0c0d
```

AND e **OR** sono ampiamente utilizzati per operazione di **mascheratura** e **impostazioni** di bit. **XOR** molto utilizzato per **crittografia** ...

Esempio: eseguire EBX=pippo se almeno uno dei bit 2 o 4 in AL è 1:

```
AND AL, 00010100b    // Maschera tutti i bit tranne 2 e 4
CMOVNZ EBX, pippo    // Assegna EBX=pippo se ZF è zero,
                        ovvero se AL dopo la mascheratura
                        contiene qualche bit a 1
```

Esempio: imposta a 1 i bit 0 e 4 in AL, e a 0 il bit 1 di AH:

```
OR AL, 00010001b
AND AH, 11111101b
```

Operazioni sui bit (2)

- **SAL/SAR DST, #**: shift **aritmetico** bit a bit a sinistra/destra in DST di un numero di bit specificato dal secondo operando. *Aritmetico significa equivalente a una moltiplicazione per 2 (SAL) o divisione per 2 (SAR).*

può essere un **valore immediato a 8 bit** (solo i valori da 0 a 31 sono ammessi) oppure il **registro CL**.

- Nel caso di shift a sinistra (**SAL**), per ogni shift atomico (1 posizione), il bit **meno significativo** assume valore **0**, mentre il bit **più significativo** (che fuoriesce) finisce in **CF**.
- Nel caso di shift a destra (**SAR**), per ogni shift atomico (1 posizione), il bit **meno significativo** fuoriesce e finisce in **CF**, mentre il bit più significativo MSB estende il segno (stesso valore del precedente MSB).

Esempi:

```
MOV EAX, 20
SAL EAX, 2           // EAX = 80
MOV EAX, -9
SAR EAX, 1           // EAX = -5; se non avessi esteso il
                    // segno che valore avrei ottenuto?
```

- **SHL/SHR DST, #**: shift **logico** bit a bit a sinistra/destra in DST di un numero di bit specificato dal secondo operando. *Logico significa scorrimento puro senza estensione del segno.*

può essere un **valore immediato a 8 bit** (compreso tra 0 e 31) oppure il **registro CL**.

SHL opera in modo identico a SAL (hanno lo stesso OP-CODE), mentre **SHR a differenza di SAR non estende il bit di segno** ma pone a 0 l'MSB entrante.

Esempio:

```
MOV AL, 01001011b
SHR AL, 1           // AL = 00100101
```


Operazioni sui bit (3)

- **ROL/ROR DST,#**: rotazione logica bit a bit a sinistra/destra in DST di un numero di bit specificato dal secondo operando.

può essere un **valore immediato a 8 bit** (solo i valori da 0 a 31 sono ammessi) oppure il **registro CL**.
 - Nel caso di rotazione a sinistra (**ROL**), per ogni rotazione atomica (1 posizione), il bit **più significativo** fuoriesce ma rientra a destra diventando il nuovo bit **meno significativo**.
 - Nel caso di rotazione a destra (**ROR**), per ogni rotazione atomica (1 posizione), il bit **meno significativo** fuoriesce ma rientra a sinistra divenendo il bit **più significativo**.

Esempio:

```
MOV AL, 01010101b
```

```
ROR AL, 1           // AL = 10101010
```

Istruzioni di Test e Salto

TEST SRC1, SRC2	Imposta i flag sulla base di SRC1 AND SRC2
CMP SRC1, SRC2	Imposta i flag sulla base di SRC1-SRC2
JMP Addr	Salto incondizionato a Addr
Jcc Addr	Salto condizionale a Addr
LOOPcc	Cicla fino a che la condizione è vera
CALL Addr	Chiamata di procedura all'indirizzo Addr
RET	Ritorno da procedura

Le istruzioni di test e salto costituiscono un **insieme molto importante** di istruzioni che devono essere ben comprese al fine di una corretta programmazione in linguaggio assembly.

In generale, in tutti gli ISA esistono **salti incondizionati**, **salti condizionali** che vengono intrapresi se certe condizioni sono vere e meccanismi per la **chiamata di sottoprogrammi**..

Istruzioni di Test e di Salto

- **TEST SRC1, SRC2**: esegue l'AND logico di SRC1 e SRC2; il risultato non viene scritto da nessuna parte ma viene utilizzato per l'impostazione dei flag **SF**, **ZF** e **PF** nel registro EFLAGS.
 - **SF** viene impostato al valore del bit più significativo del risultato.
 - **ZF** viene impostato se il risultato è 0.
 - **PF** viene impostato se il byte meno significativo del risultato ha parità pari.

A cosa serve?

Come sarà chiaro tra un attimo tutte le istruzioni di salto condizionato operano sulla base del valore dei flag. Tramite questa istruzione è **ad esempio** possibile decidere di saltare quando alcuni bit di un certo registro o variabile in memoria sono impostati a 1 o a 0; in questo caso **SRC2 viene utilizzato come maschera** (valore immediato).

```
TEST AL,00000011b
JNZ Addr           // Salta ad Addr se uno dei bit 0 o 1
                   // in AL è impostato ad 1
```

Analogo risultato può essere ottenuto con:

```
AND AL,00000011b
JNZ Addr           // Qual'è la differenza?
```

- **CMP SRC1, SRC2**: esegue la sottrazione SRC1-SRC2; il risultato non viene scritto da nessuna parte ma viene utilizzato per l'impostazione dei flag **CF**, **SF**, **ZF**, **PF**, **OF**, **AF** nel registro EFLAGS.

```
CMP AL,20
JE Addr           // Salta ad Addr se AL = 20
```

Nei lucidi successivi è riportato l'elenco dei **condition codes** utilizzati dalle istruzioni di **salto condizionale** e altre istruzioni tipo **CMOV**, **LOOP**.

Istruzioni di Test e di Salto (2)

- **JMP Addr**: esegue un salto incondizionato a Addr; il salto viene in pratica eseguito caricando in **EIP** (Extended Instruction Pointer) l'indirizzo **Addr**.

Il programma assembler permette di utilizzare **etichette simboliche** che verranno poi sostituite con **indirizzi relativi all'istruzione corrente** (a 8, 16 o 32 bit) a tempo di compilazione del programma.

```
JMP Fine      // Salta all'indirizzo Fine
...
Fine: MOV AX,20
```

È anche possibile specificare **indirizzi assoluti**, utilizzando in modo indiretto registri o memoria:

```
JMP [EDX]     // Salta all'indirizzo di memoria indicato dalla
               DWORD all'indirizzo specificato da EDX
```

- **Jcc Addr**: salta all'indirizzo Addr **se e solo se** la **condition code cc** determinata a partire dai flag impostati con l'istruzione (solitamente) precedente **è vera**.

```
CMP EAX,ECX
JE Addr       // Salta ad Addr se EAX = ECX

CMP EAX,ECX
JB Addr       // Salta ad Addr se EAX < ECX (unsigned)

CMP EAX,ECX
JA Addr       // Salta ad Addr se EAX > ECX (unsigned)

CMP EAX,ECX
JNE Addr      // Salta ad Addr se EAX <> ECX
...
```

Istruzioni di Test e di Salto (3)

L'elenco dei **condition code**, inclusivo dei rispettivi codici mnemonici e corrispondenza in termini di flag è riportato nel lucido successivo. Nella pratica, l'utilizzo dei **codici mnemonici** consente spesso di "ignorare" il funzionamento in termini di flag. Bisogna però fare attenzione e **distinguere** operazioni in **aritmetica unsigned** (solo positivi) e in **aritmetica signed** (numeri negativi in complemento a due).

Infatti, quando viene **caricato un valore immediato** in un registro non si indica al sistema se questo è **signed** o **unsigned**; alcune operazioni (es: **MUL** e **IMUL**) esplicitamente operano su un solo tipo, altre (es. **ADD** o **SUB**) non fanno differenza e solo attraverso il modo in cui flag vengono settati siamo in grado di capire ad esempio se siamo incorsi in una situazione di traboccamento ...

Istruzioni di Test e di Salto (4)

EFLAGS Condition Codes

Mnemonic (cc)	Condition Tested For	Status Flags Setting
O	Overflow	OF = 1
NO	No overflow	OF = 0
B NAE C	Below Neither above nor equal	CF = 1
NB AE NC	Not below Above or equal	CF = 0
E Z	Equal Zero	ZF = 1
NE NZ	Not equal Not zero	ZF = 0
BE NA	Below or equal Not above	(CF OR ZF) = 1
NBE A	Neither below nor equal Above	(CF OR ZF) = 0
S	Sign	SF = 1
NS	No sign	SF = 0
P PE	Parity Parity even	PF = 1
NP PO	No parity Parity odd	PF = 0
L NGE	Less Neither greater nor equal	(SF XOR OF) = 1
NL GE	Not less Greater or equal	(SF XOR OF) = 0
LE NG	Less or equal Not greater	((SF XOR OF) OR ZF) = 1
NLE G	Neither less nor equal Greater	((SF XOR OF) OR ZF) = 0

Senza segno

Con segno

Istruzioni di Test e di Salto (5)

Esistono **altre due versioni** di **Jcc** dove cc non si riferisce ai condition code determinati dai flag: **JCXZ** e **JECXZ**

JCXZ Addr: salta ad Addr se CX = 0

JECXZ Addr: salta ad Addr se ECX = 0

NOTA: Addr può essere specificato solo come **indirizzo relativo a 8 bit**; pertanto se l'etichetta utilizzata si trova distante dal punto di salto l'assemblatore può non essere in grado di generare un indirizzo compreso in -128 .. +127; In questo caso siamo costretti a utilizzare altre forme di Jcc.

- **LOOP/LOOPcc Addr**: si tratta di un'istruzione compatta e ottimizzata per l'esecuzione di cicli dove per la variabile contatore viene usato obbligatoriamente il registro **ECX**. Addr può essere solo un indirizzo relativo a 8 bit.
 - **LOOP Addr**: il registro ECX viene **decrementato automaticamente** di un'unità, il valore di ECX viene **controllato**, se ECX è diverso da 0 salta ad Addr.

Esempio: *somma in EAX gli elementi di un vettore di double word di lunghezza 10 (offset da 0 a 9, ogni elemento 4 byte):*

```
MOV ECX,10      // Valore iniziale di ECX
XOR EAX,EAX
Ciclo: ADD EAX, Vettore[ECX*4-4]
      LOOP Ciclo
```

Lo stesso risultato (**meno efficiente**) può essere ottenuto con:

```
MOV ECX,10      // Valore iniziale di ECX
XOR EAX,EAX
Ciclo: ADD EAX, Vettore[ECX*4-4]
      DEC ECX
      JNZ Ciclo
```

D'altro canto **LOOP** **costringe a contare all'indietro** e ad utilizzare **obbligatoriamente ECX**.

Istruzioni di Test e di Salto (6)

Esiste una variante di LOOP che oltre a controllare quando ECX diviene 0, controlla anche le 4 **condition code** **E**, **Z**, **NE**, **NZ** legate al flag ZF:

LOOPcc Addr: continua a ciclare (saltare ad Addr) fino a quando ECX è diverso da 0 e la condizione **cc** è vera. Pertanto due eventi possono causare l'uscita dal ciclo (è sufficiente che se ne verifichi uno):

- **ECX** = 0
- **cc** falsa (da notare che LOOP non altera i flag e quindi ZF deve essere in questo caso impostato da qualche istruzione interna al ciclo).

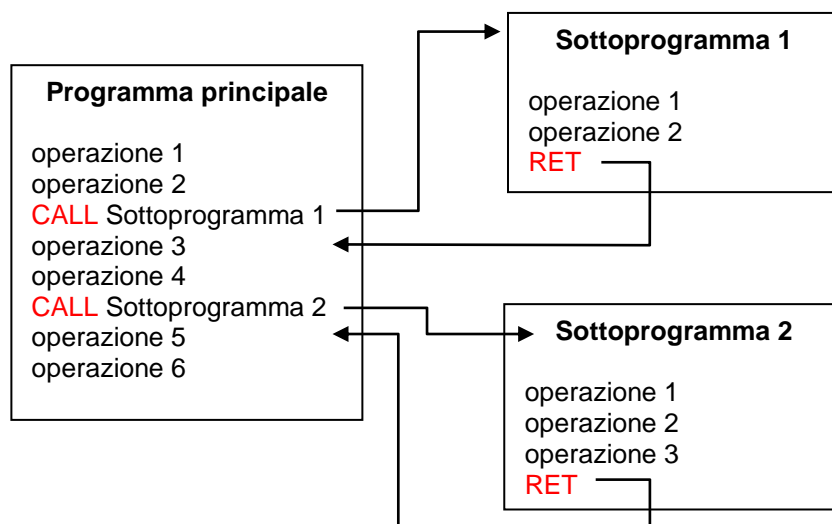
```
MOV ECX,10      // Valore iniziale di ECX
XOR EAX,EAX
Ciclo: ADD EAX, Vettore[ECX*4-4]
      CMP EAX,100
      LOOPNE Ciclo
```

NOTA: quando si utilizza LOOP/LOOPcc bisogna **fare attenzione** a entrare nel ciclo con **ECX > 0**; in caso contrario infatti la prima volta che il valore viene decrementato si passa a 0xffffffff e quindi il ciclo verrà eseguito 2^{32} volte. L'istruzione **JECXZ** consente di eseguire un semplice controllo in entrata:

```
MOV ECX,pippo   // Non sono sicuro del valore iniziale
JECXZ Fine
XOR EAX,EAX
Ciclo: ADD EAX, Vettore[ECX*4-4]
      CMP EAX,100
      LOOPNE Ciclo
Fine:
```


Istruzioni di Test e di Salto (7)

- **CALL** Addr e **RET**: esecuzione di un **sottoprogramma** a partire dall'indirizzo Addr. L'utilizzo di sottoprogrammi è un concetto fondamentale nella programmazione; si tratta di demandare l'esecuzione di una funzione a un insieme di istruzioni logicamente separate dal programma principale.



Il sottoprogramma termina con un'istruzione **RET**, a seguito della quale il controllo ritorna al programma chiamante che continua l'esecuzione **all'istruzione successiva rispetto alla chiamata**. Si noti che non è necessario specificare l'indirizzo di ritorno...

Ciò comporta una serie di vantaggi:

- Se la funzione eseguita dal sottoprogramma deve essere eseguita più volte, non è necessario **replicare il codice**.
- Utilizzo di **parametri** (per valore e indirizzo)
- I sottoprogrammi possono essere **raccolti in librerie** e riutilizzati per lo sviluppo di applicazioni diverse. Le librerie del sistema operativo vengono sempre invocate sotto forma di sottoprogrammi (a parte il caso di **Interrupt** le cui procedure di risposta sono comunque analoghe ai sottoprogrammi).

Istruzioni di Test e di Salto (8)

Esempio chiamata di procedura: programma che somma in DX i 10 elementi di un vettore di WORD, eseguendo di ogni parola la conversione in Big Endian prima di sommarla in DX.

```
JMP Main
```

```
// Sottoprogramma che trasforma in big endian la WORD in AX
```

```
Swap:  MOV BL,AH  
        SHL AX,8  
        MOV AL,BL  
        RET
```

```
// Programma principale
```

```
Main:  XOR ECX,ECX  
        XOR DX,DX  
Ciclo: MOV AX,Vettore[ECX*2]  
        CALL Swap  
        ADD DX,AX  
        INC ECX  
        CMP ECX,10  
        JNE Ciclo
```

Come viene gestito l'indirizzo di ritorno?

L'istruzione **CALL** prima di eseguire il salto memorizza il **valore di EIP** (che punta all'istruzione successiva `ADD DX,AX`) **nello stack**, ovvero esegue un **PUSH EIP** sullo stack. *Ciò può essere verificato notando che il valore di ESP cambia a seguito di CALL e sullo stack viene caricata una DWORD equivalente a EIP.*

Quando il sottoprogramma termina (**RET**), un'istruzione **POP EIP** causa il ritorno al punto desiderato.

NOTA: non è possibile manipolare EIP direttamente con istruzioni del tipo `MOV EIP, EAX`.

Istruzioni di Test e di Salto (9)

Disassembly chiamata standard del C: **CDECL**

- I parametri sono messi sullo stack dal chiamante nell'ordine Right-to-Left
- La funzione chiamata sa dove andare a reperire i parametri
- Il valore di ritorno è sempre in EAX
- La funzione chiamante ripristina (pulisce) lo stack.

<pre>int somma(int v1, int v2) { return v1 + v2; }</pre> <pre>... a = 10; b = 20; c = somma (a , b); ...</pre>	<pre>_somma: push ebp // salvo temporaneamente mov ebp, esp // uso di ebp come base mov eax, [ebp + 8] // valore di v1 = a mov edx, [ebp + 12] // valore di v2 = b add eax, edx // risultato di ritorno in EAX pop ebp // ripristino valore ebp ret ... mov dword ptr [a], 0Ah mov dword ptr [b], 14h mov eax, [b] push eax // passaggio b (per primo) mov ecx, [a] push ecx // passaggio a call _somma // implicitamente push eip add esp, 8 // ripristina lo stack (clean) ...</pre>
---	---

NOTA: Il primo parametro si trova all'indirizzo **ebp + 8** a causa dei due push successivi al passaggio dei parametri: quello esplicito di ebp del chiamato e quello implicito di eip della call dal chiamante.

Esistono altre modalità di chiamata in C e in altri linguaggi: affinché sia possibile da un linguaggio chiamare funzioni di una libreria scritta in altro linguaggio occorre che le convenzioni di chiamata (a livello ISA) siano compatibili.

Manipolazione di stringhe

LODS	Leggi stringa
STOS	Scrivi stringa
MOVS	Copia stringa
CMPS	Confronta due stringhe
SCAS	Esamina stringa

Le stringhe sono **sequenze contigue di caratteri** (byte), molto utilizzate in tutti i linguaggi di programmazione. Risulta spesso necessario eseguire operazioni su stringhe, quali: **copia**, **confronto**, **concatenazione**, **inversione**, **ricerca di un carattere in stringa**, ... *L'approccio tradizionale consiste nel trattare le stringhe come vettori di byte e accedere a essi con le istruzioni comuni.*

Sono disponibili **istruzioni ottimizzate** per la manipolazione di stringhe. In realtà queste istruzioni operano indipendentemente dalla rappresentazione ASCII dei caratteri e trattano gli elementi come byte; pertanto, sarebbe più appropriato parlare di "**manipolazione di blocchi contigui di memoria**".

Le operazioni su stringhe utilizzano obbligatoriamente due **registri dedicati**: **ESI** (**E**xtended **S**ource **I**ndex) e **EDI** (**E**xtended **D**estination **I**ndex) che vengono utilizzati come **indirizzo** dell'elemento corrente nella stringa sorgente o destinazione rispettivamente.

Il prefisso **REP** o **REPcc** anteposto ad una delle istruzioni sopraelencate consente di eseguire un **ciclo sulla stringa**:

- la lunghezza della stringa deve essere specificata dal registro **ECX**. Il registro ECX viene **automaticamente decrementato** durante il ciclo.
- il registro ESI o EDI (o entrambi) viene **automaticamente incrementato o decrementato** (a seconda del **flag DF** in EFLAGS).
- il ciclo continua fino a che **ECX > 0** e, nel caso di **REPcc**, fino a che la condizione **cc è vera**. (REPcc viene usato solo con CMPS e SCAS alle quali è concesso impostare i flag).

Manipolazione di stringhe (2)

L'istruzione **STD** imposta il **flag DF a 1** (scorrimento stringa **indietro**)

L'istruzione **CLD** imposta il **flag DF a 0** (scorrimento stringa **in avanti**)

Esempio 1: Azzerare il blocco di memoria di 256 byte a partire dall'indirizzo IndBlocco.

```
CLD                // Scorrimento in avanti
MOV ECX,256        // Numero di elementi
LEA EDI,IndBlocco  // Indirizzo di partenza in EDI
XOR AL,AL          // Valore da scrivere con STOS
REP STOS           // Scrive AL su [EDI], decrementa ECX,
                    // Incrementa EDI, ripete fino a che ECX > 0
```

Esempio 2: Copiare il blocco di memoria di 256 byte a partire dall'indirizzo IndBlocco su blocco il cui indirizzo di partenza è IndBlocco2.

```
CLD                // Scorrimento in avanti
MOV ECX,256        // Numero di elementi
LEA ESI,IndBlocco  // Indirizzo di partenza in ESI
LEA EDI,IndBlocco2 // Indirizzo di partenza in EDI
REP MOVSB          // Scrive [ESI] su [EDI], decrementa ECX,
                    // Incrementa EDI ed ESI, ripete fino a che
                    // ECX > 0
```

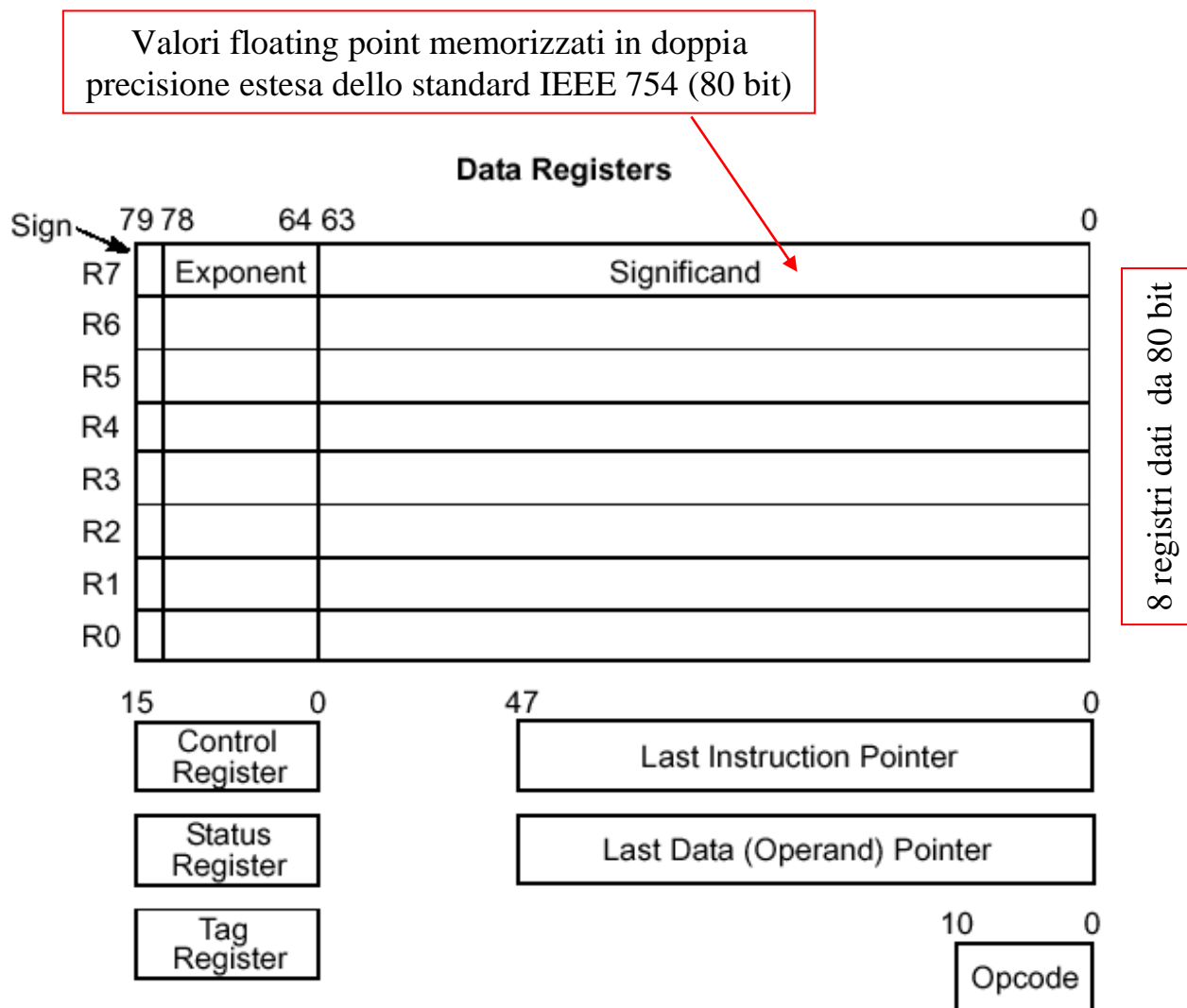
Esempio 3: Confrontare 2 blocchi di memoria lunghi 256 byte; IndBlocco e IndBlocco2 sono gli indirizzi di partenza.

```
CLD                // Scorrimento in avanti
MOV ECX,256        // Numero di elementi
LEA ESI,IndBlocco  // Indirizzo di partenza in ESI
LEA EDI,IndBlocco2 // Indirizzo di partenza in EDI
REPE CMPS          // Confronta [ESI] con [EDI] e setta il flag
                    // di conseguenza, decrementa ECX, Incrementa
                    // EDI ed ESI, ripete fino a che ECX > 0 e il
                    // flag ZF è 1.
```

All'uscita del ciclo è sufficiente controllare il flag ZF (se ZF=1 i blocchi sono uguali, altrimenti il ciclo è stato interrotto prematuramente).

Unità Floating Point (cenni)

L'unità floating point (incorporata all'interno del chip della CPU) utilizza una serie di **registri aggiuntivi** rispetto a quelli fino ad ora introdotti:



Gli 8 registri **R0..R7**, benché **accessibili singolarmente** senza nessuna restrizione sull'ordine, **vengono trattati come uno stack** sul quale le operazioni di caricamento e prelevamento aggiungono o rimuovono valori rispetto al **TOP** dello stack (memorizzato nel registro Status Register).

ST(0) si riferisce al registro 0 a partire da TOP (non necessariamente **R0**).
ST(1)..ST(7) sono i successivi registri sullo stack.

Unità Floating Point (2)

L'unità floating point fornisce **molte istruzioni** (più di 50) che possono essere raggruppate sulla base delle loro funzioni in:

- **Trasferimento di valori:** per caricare (**FLD**), salvare (**FST**), spostare, ... valori nei registri **ST(0)..ST(7)**.
- **Aritmetiche di base:** somma (**FADD**), sottrazione (**FSUB**), moltiplicazione (**FMUL**), divisione (**FDIV**), radice quadrata (**FSQRT**), ...
- **Confronto:** non è possibile confrontare con la tradizionale CMP valori floating point; sono quindi necessarie operazioni di confronto dedicate come (**FCOM**).
- **Funzioni trascendenti:** seno (**FSIN**), coseno (**FCOS**), logaritmo (**FYL2X**), esponenziale (**F2XM1**), ...
- **Caricamento di costanti note:** carica costanti quali 0, 1, π , e , ... nei registri senza dover caricarne il valore dalla memoria.
- **Controllo dell'FPU:** inizializzazione (**FINIT**), sincronizzazione, ...

La comprensione del funzionamento puntuale dell'FPU **non è cosa semplice** e non è pretesa di questo corso entrare nei dettagli.

Nel seguito viene fornito **un esempio** che calcola la semplice espressione $(a+b) \times (c-d)$:

```
FINIT          // Inizializza l'FPU
FLD    a       // Carica a in ST(0) che punta a R0
FADD    b       // R0 = R0 + b = (a+b)
FLD    c       // Carica c in ST(0) che punta a R1
FSUB    d       // R1 = R1 - d
FMUL           // R0 = R0 * R1
```

Si faccia attenzione all'utilizzo dei registri come stack; le operazioni aritmetiche quando sono dotate di operandi (es. **FADD b**) eseguono l'operazione tra l'operando e il registro **ST(0)**; se invece utilizzassimo **FADD** senza operandi la somma verrebbe fatta tra **ST(0)** ed **ST(1)** (vedi il caso di **FMUL**).

Unità Floating Point (3)

NOTA BENE

I dati in virgola mobile quando vengono **caricati nei registri**, indipendentemente dal fatto che siano in singola precisione (**32 bit**), doppia precisione (**64 bit**) o doppia precisione estesa (**80 bit**) vengono convertiti in doppia precisione estesa e memorizzati nei registri a 80 bit.

Tutte le **operazioni aritmetiche floating point** sono eseguite in formato **doppia precisione estesa**. Qualora i risultati debbano essere nuovamente scritti in variabili o in memoria in formato più breve viene eseguita una nuova conversione.

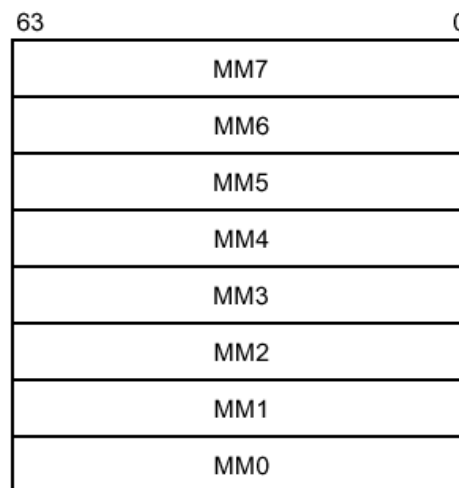
Come **regola generale**, quando **non vi sono particolari problemi di occupazione di memoria**, si consiglia di utilizzare normalmente variabili in **doppia precisione**: ciò non comporta **nessun aggravio di tempo** rispetto all'utilizzo di singola precisione e permette di **sfruttare la rappresentazione interna** a 80 bit per **minimizzare arrotondamenti** o perdite di cifre decimali.

Istruzioni MMX (cenni)

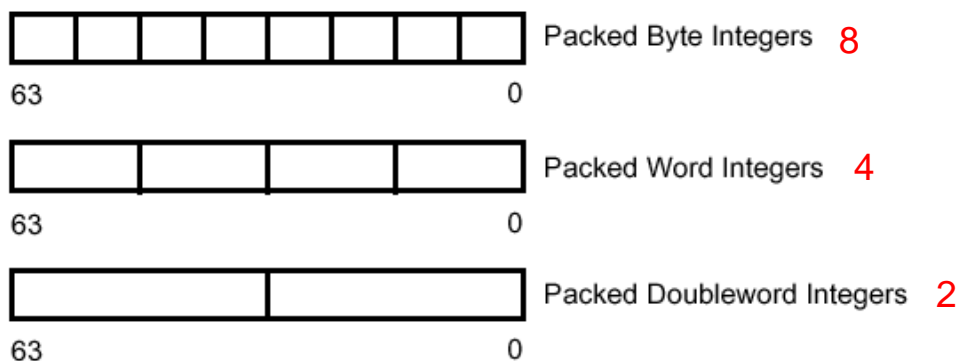
MMX nasce a partire dal **Pentium Pro** e viene incorporato su tutti i processori Intel successivi (Pentium II, Pentium III, Pentium 4, ...). Si tratta di un potente meccanismo che consente di utilizzare il processore come macchina **SIMD** (Single Instruction Multiple Data), ovvero di eseguire in parallelo la stessa operazione su più dati.

Ad esempio, attraverso un'istruzione MMX è possibile eseguire in parallelo 4 somme su parole di 2 byte (WORD).

MMX opera solo su **aritmetica intera**, utilizza **8 registri a 64 bit** (che condivide con la FPU, per questo non si possono utilizzare contemporaneamente istruzioni floating point e MMX):



e **tre nuovi tipi di dato**:



Istruzioni MMX (2)

Le istruzioni MMX (anch'esse **molto numerose**) possono essere classificate in base alla tabella seguente.

Category		Wraparound	Signed Saturation	Unsigned Saturation
Arithmetic	Addition	PADDB, PADDW	PADDSB, PADDW	PADDUSB, PADDUSW
	Subtraction	PSUBB, PSUBW, PSUBD	PSUBSB, PSUBSW	PSUBUSB, PSUBUSW
	Multiplication	PMULL, PMULH		
	Multiply and Add	PMADD		
Comparison	Compare for Equal	PCMPEQB, PCMPEQW, PCMPEQD		
	Compare for Greater Than	PCMPGTPB, PCMPGTPW, PCMPGTPD		
Conversion	Pack		PACKSSWB, PACKSSDW	PACKUSWB
Unpack	Unpack High	PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ		
	Unpack Low	PUNPCKLBW, PUNPCKLWD, PUNPCKLDQ		
Logical	And And Not Or Exclusive OR	Packed		Full Quadword
				PAND PANDN POR PXOR
				PSLLQ PSRLQ
Shift	Shift Left Logical Shift Right Logical Shift Right Arithmetic	PSLLW, PSLLD PSRLW, PSRLD PSRAW, PSRAD		
Data Transfer	Register to Register Load from Memory Store to Memory	Doubleword Transfers	Quadword Transfers	
		MOVD MOVD MOVD	MOVQ MOVQ MOVQ	
Empty MMX State		EMMS		

Opera su packed WORD

Opera su packed BYTE

Opera su packed DWORD

Operazioni a 64 bit

Istruzioni MMX (3)

Supponiamo di voler sommare in parallelo 4 WORD signed con un'unica istruzione. **Come gestire il problema dell'overflow?** Essendo praticamente impossibile (e inefficiente) gestire singoli bit di carry e overflow, vengono introdotte **tre modalità operative esplicite** che la maggior parte delle operazioni MMX supporta:

- **Wraparound**: in caso di traboccamento **si perdono i bit più significativi** e il registro conserva i bit meno significativi. *La somma di 2 ad un byte senza segno con valore 255 dà come risultato 1.*
- **Signed saturation**: in caso di traboccamento verso l'alto il valore viene **rimpiazzato con l'intero positivo maggiore (signed)** esprimibile con la lunghezza di parola considerata. In caso di traboccamento verso il basso con il **negativo maggiore (signed)**. *Ad esempio nel caso di byte sommare 10 a +120 produce il valore 127 mentre sottrarre 50 a -120 produce -128.*
- **Unsigned saturation**: in caso di traboccamento verso l'alto il valore viene **rimpiazzato con l'intero positivo maggiore (unsigned)** esprimibile con la lunghezza di parola considerata. In caso di traboccamento verso il basso **con 0**. *Ad esempio nel caso di byte sommare 40 a +220 produce il valore 255 mentre sottrarre 50 a 40 produce 0.*

Esempio: dati due vettori di byte (Vettore1 e Vettore2) di lunghezza 8, eseguire la **somma byte a byte** con una sola operazione:

```
// dichiarazione in C
unsigned char Vettore1[]={10,20,30,40,50,60,70,80};
unsigned char Vettore2[]={80,70,60,50,40,30,20,10};
...

MOVQ  MM0,Vettore1    // Carica 8 byte in MM0
MOVQ  MM1,Vettore2    // Carica 8 byte in MM1
PADDB MM0,MM1         // Esegue la somma (wraparound)
EMMS                  // Pulizia finale registri MMX
```

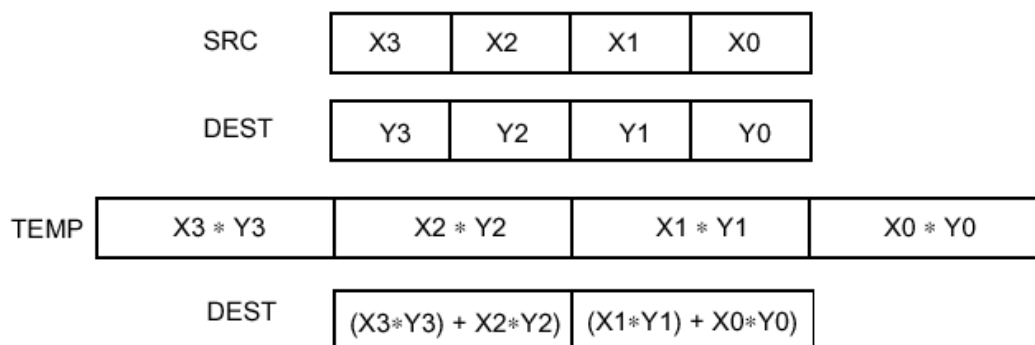
Istruzioni MMX (4)

Esempio: dati due vettori di unsigned word (V1 e V2) di lunghezza 4, eseguirne il **prodotto scalare**; il prodotto scalare di due vettori V1 e V2 di lunghezza 4 è definito come:

$$\text{DotProd} = V1[1] \times V2[1] + V1[2] \times V2[2] + V1[3] \times V2[3] + V1[4] \times V2[4]$$

Per questa **particolare operazione**, molto frequente in applicazioni di calcolo numerico, matematico, grafica, analisi del suono e di immagini, è prevista un'istruzione speciale **PMADDWD** che esegue:

- **moltiplicazione** WORD a WORD delle due packed word in input
- memorizza i risultati intermedi in un **registro interno a 128 bit** (evitando traboccamenti tranne in casi particolari)
- **somma a due a due** le coppie contigue per ottenere come risultato un packet dword



```
// dichiarazione in C
unsigned short V1[]={2,4,8,16};
unsigned short V2[]={2,4,8,16};

...
MOVQ    MM0,V1        // carica packet signed word V1 in MM0
MOVQ    MM1,V2        // carica packet signed word V2 in MM1
PMADDWD MM0,MM1        // moltiplica e accumula
MOVD    EAX,MM0        // carica in EAX la dword bassa (0..31)
PSRLQ   MM0,32         // sposta la dword alta in (0..31)
MOVD    EBX,MM0        // carica in EBX la dword bassa (0..31)
ADD     EAX,EBX        // EAX = EAX + EBX -> Risultato!
EMMS                    // Pulizia finale registri MMX
```

Ulteriori evoluzioni SIMD (1)

- A partire dal **Pentium III** è stato introdotto **SSE**:
 - aggiunge **8 nuovi registri a 128 bit** chiamati **XMM** (una notevole limitazione di MMX era infatti quella di dover usare in condivisione i registri con l'unità floating point).

127	0
XMM7	
XMM6	
XMM5	
XMM4	
XMM3	
XMM2	
XMM1	
XMM0	

- Sui nuovi registri è possibile eseguire operazioni **SIMD su floating point in singola precisione** (in modo analogo a quanto MMX fa su interi).
- SSE **estende anche le funzionalità MMX** (es: nuove operazioni quali **media**, **max**, **min** su packed data).
- SSE include nuove operazioni per il controllo e l'ottimizzazione di **pre-fetching** e **caching**.
- A partire dal **Pentium 4** (con core Willamette nel 2001) è stato introdotto **SSE2**:
 - Consente di eseguire operazioni **SIMD su floating point in doppia precisione**.
 - Estende MMX rendendo possibile operare su **packet di interi a 128 bit!** (16 byte per volta o 8 word per volta).
 - Aggiunge **nuove operazioni SIMD**.
 - Maggiore controllo della **cache** e dell'**ordine di esecuzione** delle istruzioni.

Ulteriori evoluzioni SIMD (2)

- Le istruzioni **SSE3** sono state introdotte agli inizi del 2004 con il **Pentium 4** (con core Prescott).
 - SSE3 aggiunge **13 nuove istruzioni** rispetto a SSE2
 - la più rivoluzionaria di queste istruzioni consente di lavorare **orizzontalmente** in un registro (precedentemente era possibile solo verticalmente). Più precisamente, sono state aggiunte le istruzioni per sommare e sottrarre i molteplici valori memorizzati in un singolo registro.
- SSE4** (nel 2007) disponibili sui processori Intel multi Core (a partire da Core 2 Duo).
 - SSE4 aggiunge **54 nuove istruzioni** (partizionate in due gruppi: SSE4.1 e SSE 4.2) orientate principalmente ad accelerazione video e grafica.
 - Prodotto scalare floating point
 - Conteggio numero di bit a 1 in una parola (POPCNT)
- AVX - Advanced Vector Extensions** (nel 2011) disponibile su processori Intel (con Core Sandy Bridge).
 - I registri passano da 128 a **256 bit** (YMMn)
 - Istruzioni a tre operandi: $c = a + b$
 - Per applicazioni floating point-intensive
- FMA** esplicitamente dedicato a operazioni Moltiplica-Accumula.
- AVX2** dal 2013: operazioni su interi con i registri a 256 bit e altre nuove istruzioni
- AVX-512** dal 2017: **32** registri a **512 bit** (ZMMn) e numerose nuove istruzioni

511	256	255	128	127	0
ZMM0	YMM0	XMM0			
ZMM1	YMM1	XMM1			
ZMM2	YMM2	XMM2			
ZMM3	YMM3	XMM3			
ZMM4	YMM4	XMM4			
ZMM5	YMM5	XMM5			
ZMM6	YMM6	XMM6			
ZMM7	YMM7	XMM7			
ZMM8	YMM8	XMM8			
ZMM9	YMM9	XMM9			
ZMM10	YMM10	XMM10			
ZMM11	YMM11	XMM11			
ZMM12	YMM12	XMM12			
ZMM13	YMM13	XMM13			
ZMM14	YMM14	XMM14			
ZMM15	YMM15	XMM15			
ZMM16	YMM16	XMM16			
ZMM17	YMM17	XMM17			
ZMM18	YMM18	XMM18			
ZMM19	YMM19	XMM19			
ZMM20	YMM20	XMM20			
ZMM21	YMM21	XMM21			
ZMM22	YMM22	XMM22			
ZMM23	YMM23	XMM23			
ZMM24	YMM24	XMM24			
ZMM25	YMM25	XMM25			
ZMM26	YMM26	XMM26			
ZMM27	YMM27	XMM27			
ZMM28	YMM28	XMM28			
ZMM29	YMM29	XMM29			
ZMM30	YMM30	XMM30			
ZMM31	YMM31	XMM31			

Intel Intrinsics

Non tutte le famiglie di istruzioni SIMD sono supportate dai compilatori o sono accessibili tramite assembly inline.

Intel rende disponibili funzioni **Intrinsics** (C style), una sorta di wrapper in C per le diverse famiglie di istruzioni SIMD. Un intrinsic può essere tradotto dal compilatore in una o più istruzioni.

<https://software.intel.com/sites/landingpage/IntrinsicsGuide/>



Technologies

- ☐ MMX
- ☐ SSE
- ☐ SSE2
- ☐ SSE3
- ☐ SSSE3
- ☐ SSE4.1
- ☐ SSE4.2
- ☐ AVX
- ☐ AVX2
- ☐ FMA
- ☐ AVX-512
- ☐ KNC
- ☐ SVMML
- ☐ Other

The Intel Intrinsics Guide is an interactive reference tool for Intel intrinsic instructions, which are C style functions that provide access to many Intel instructions - including Intel® SSE, AVX, AVX-512, and more - without the need to write assembly code.

Esempio con x86-64

Esempio di codice C compilato per x64:

```
float somma(float *a, __int64 n) {  
    float somma = 0;  
    for (__int64 j = 0; j < n; j++) {  
        somma += a[j];  
    }  
    return somma;  
}
```

	xor	eax, eax	
	xorps	xmm0, xmm0	
	test	rdx, rdx	
	jle	somma+17h (07FF788FD1017h)	→L2:
L1:	addss	xmm0, dword ptr [rcx+rax*4]	
	inc	rax	
	cmp	rax, rdx	
	j1	somma+0Ah (07FF788FD100Ah)	→L1:
L2:	ret		

Si può notare che il compilatore:

- ha utilizzato registri a 64 bit (**RAX**, **RDX**), disponibili in x64, per gli interi **__int64** e registri a 128 bit (**XMM0**) per i float a 32 bit (utilizzandone solo i 32 bit più bassi);
- per la somma fra due float a 32 bit ha utilizzato **ADDSS** (Add Scalar Single-Precision Floating-Point Values).

Esempio con AVX-512

Esempio di codice C con intrinsics AVX-512 compilato per x64:

```
float somma_avx512(float *a, __int64 n) {  
    __m512 tmp_x16 = _mm512_setzero_ps();  
    for (__int64 j = 0; j < n; j += 16) { // Assume n multiplo di 16  
        tmp_x16 = _mm512_add_ps(tmp_x16, _mm512_loadu_ps(a + j));  
    }  
    return _mm512_reduce_add_ps(tmp_x16);  
}
```

```
vxorps    xmm1,xmm1,xmm1  
test      rdx,rdx  
jle       somma_avx512+24h (07FF788FD1044h)→L2  
lea       rax,[rdx-1]  
shr       rax,4  
inc       rax  
L1: vaddps  zmm1,zmm1,zmmword ptr [rcx]  
lea       rcx,[rcx+40h]  
sub       rax,1  
jne       somma_avx512+14h (07FF788FD1034h)→L1  
L2: vextractf64x4 ymm0,zmm1,1  
vaddps    ymm3,ymm0,ymm1  
vpermpd   ymm2,ymm3,0AAh  
vpermpd   ymm1,ymm3,0FFh  
vaddps    ymm1,ymm1,ymm3  
vpermpd   ymm0,ymm3,55h  
vaddps    ymm0,ymm2,ymm0  
vaddps    ymm2,ymm0,ymm1  
vpsrlq    xmm1,xmm2,20h  
vaddss    xmm0,xmm1,xmm2  
vzeroupper  
ret
```

Si può notare che il compilatore:

- per tradurre l'intrinsic `_mm512_setzero_ps`, ha utilizzato un xor fra registri a 512 bit (**VXORPS**);
- per `_mm512_add_ps` ha utilizzato **VADDPS** che somma i 16 float a 32 bit di due registri a 512 bit;
- `_mm512_reduce_add_ps` (somma orizzontale dei 16 float in un registro) viene tradotto con ben **11 istruzioni**.