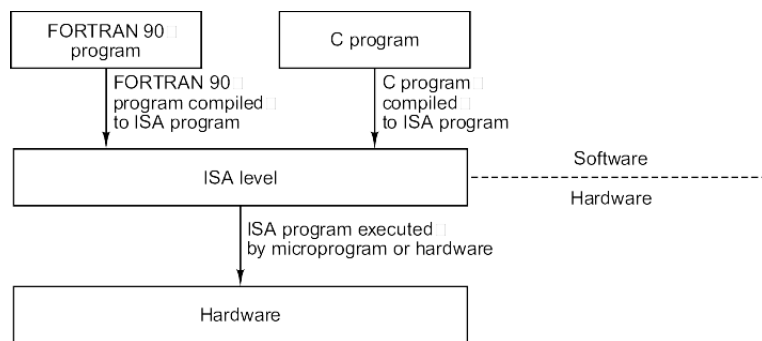


Il livello ISA

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**).

Java	Assembly	ISA (esadecimale)
<code>i = j + k;</code>	1 ILOAD j // i = j + k	0x15 0x02
<code>if (i == 3)</code>	2 ILOAD k	0x15 0x03
<code> k = 0;</code>	3 IADD	0x60
<code>else</code>	4 ISTORE i	0x36 0x01
<code> j = j - 1;</code>	5 ILOAD i // if (i < 3)	0x15 0x01
	6 BIPUSH 3	0x10 0x03
	7 IF_ICMPEQ L1	0x9F 0x00 0x0D
	8 ILOAD j // j = j - 1	0x15 0x02
	9 BIPUSH 1	0x10 0x01
	10 ISUB	0x64
	11 ISTORE j	0x36 0x02
	12 GOTO L2	0xA7 0x00 0x07
	13 L1: BIPUSH 0	// k = 0 0x10 0x00
	14 ISTORE k	0x36 0x03
	15 L2:	

- 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 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 indifferente. **Linguaggio macchina** viene talvolta usato per indicare Assembly language, altre volte per istruzioni ISA.

Perché studiare ISA e Assembler ?

- E' 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 di IA-32 ovvero dell'ISA dei processori x86 compatibili a 32 bit (es. Pentium, Athlon).

Tutti i processori Intel dall'80386 in poi (ma anche AMD) hanno adottato lo stesso ISA (IA-32) ad eccezione di differenze di secondaria importanza (ad esempio istruzioni MMX in Pentium Pro e successivi). Questo non significa affatto che tutti abbiano le stesse prestazioni !

Itanium è il primo processore Intel ad adottare la piattaforma IA-64. Brevi cenni alle caratteristiche di IA-64 vengono fornite nella parte finale delle dispense sulle "Architetture a Confronto".

Il Pentium ha tre modalità operative (due delle quali per compatibilità con vecchi modelli a 16 bit):

- **reale**: opera fisicamente come un 8088 (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.
- **virtuale**: opera in emulazione 8088, 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.
- **protetta**: opera veramente come Pentium e non come un costoso 8088. E' possibile impostare uno tra quattro possibili livelli di privilegio.
 - Il livello 0 corrisponde alla modalità kernel e ha completo accesso alla macchina (pericoloso !); viene utilizzato dal sistema operativo e dai driver di periferica.
 - Il livello 3 è riservato ai programmi utente. Impedisce l'accesso a certe istruzioni ISA critiche e a risorse vitali della macchina per evitare che errori accidentali nei programmi possano bloccare la macchina.
 - I livelli 1 e 2 sono usati raramente.

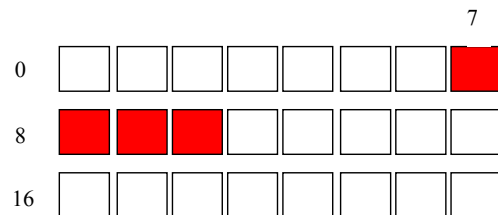
Modello di memoria del Pentium

Il Pentium è dotato di uno **spazio di indirizzamento** con memoria divisa in **16.384 segmenti**. Ogni segmento è in realtà talmente vasto (2^{32} byte - oltre 4 GB) che la maggior parte dei sistemi operativi (Windows, Unix, ...) utilizzano un solo segmento. Per questo motivo si è soliti parlare di **spazio di indirizzamento lineare** !

Il Pentium è in grado di **indirizzare fisicamente** la memoria con **allineamento a parole di 8 byte**; infatti delle 36 linee indirizzo (64GB) le 3 meno significative sono forzate a 0 (connesse a massa!). Il Pentium **legge/scrive** dalla/sulla memoria in blocchi di **8 byte per volta**.

D'altro canto, per motivi di compatibilità, è **possibile indirizzare** in memoria **ogni singolo byte** indipendentemente dall'allineamento.

ATTENZIONE però: il fatto che accessi non allineati **siano possibili** (e i relativi dettagli **siano nascosti al programmatore**) **non significa che siano anche efficienti**; infatti, supponiamo di voler accedere a una parola di 4 byte all'indirizzo 7:

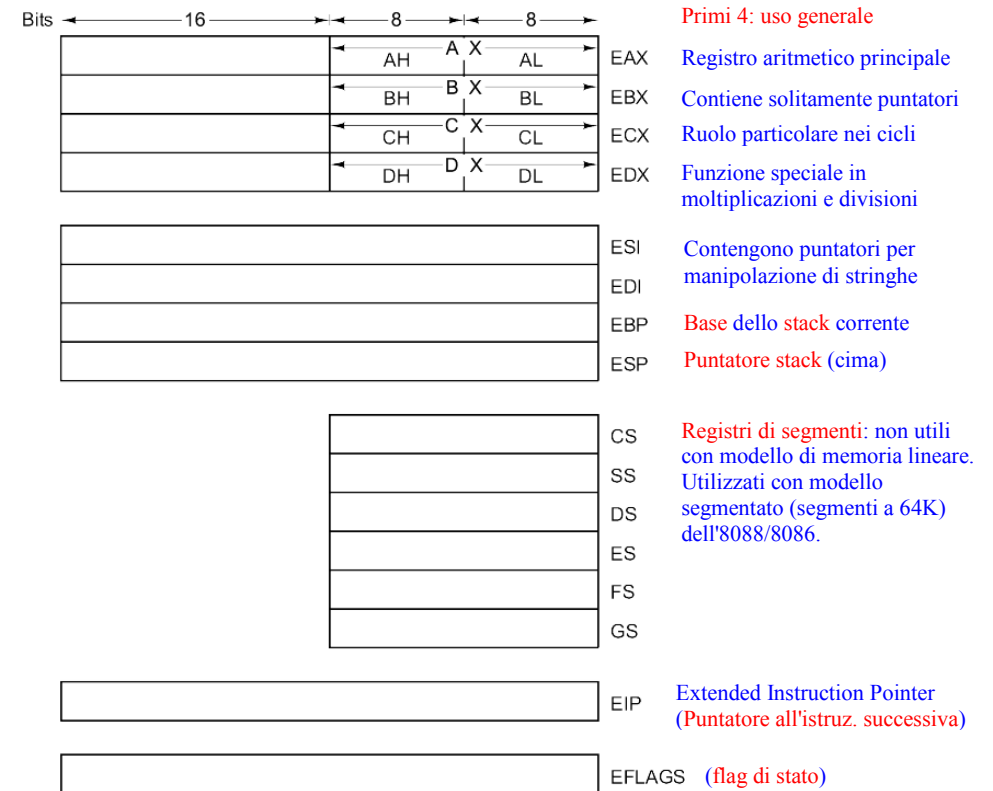


- l'hardware deve caricare un **primo blocco (byte 0..7)**
- un secondo riferimento alla memoria è necessario per il **blocco (8..15)**
- la CPU deve poi **estrarre i 4 byte** richiesti dai 16 letti e organizzarli nell'ordine giusto.

IMPORTANTE

Le informazioni sono memorizzate dal **Pentium** (e in generale dalle **CPU Intel**) in modalità **LITTLE ENDIAN** (prima byte meno significativo).

Registri del Pentium



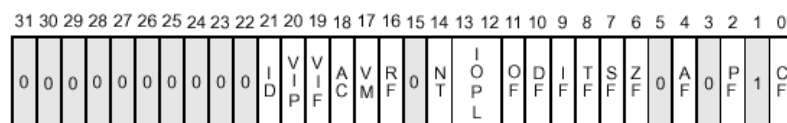
Il **Pentium** dispone di **16** registri di base (un numero **piuttosto limitato**), molti dei quali sono tra l'altro **specializzati** o **obsoleti**. In realtà le cose migliorando se consideriamo anche i registri **MMX** (8 registri a 64 bit).

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**, che sono di uso **più o meno generale**, possono essere utilizzati a **8** bit (es: AL, AH), a **16** bit (es: AX) o a **32** bit (es: EAX).

Registri del Pentium (2)

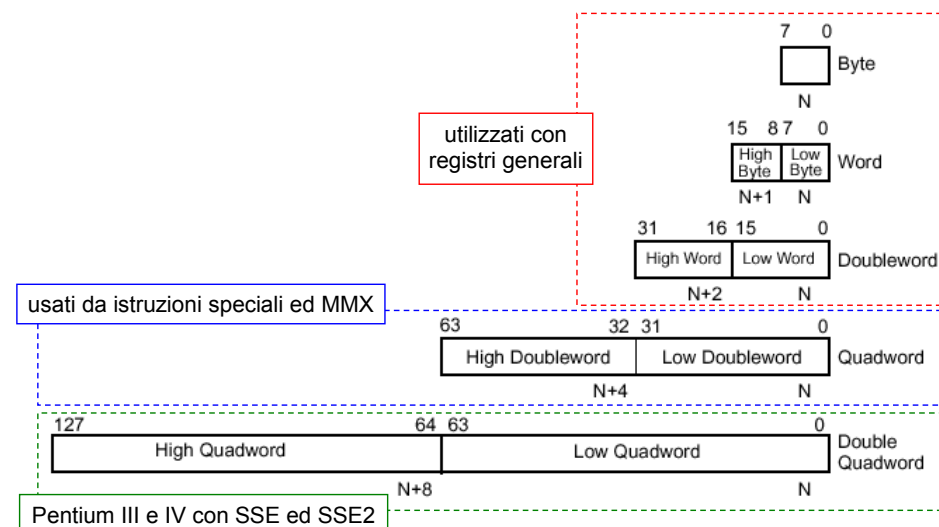
- **EIP (Instruction pointer)**: contiene l'indirizzo della prossima istruzione da eseguire. Viene incrementato **automaticamente** durante il fetch delle istruzioni e **modificato** dalle **istruzioni di salto**. I programmi utente non lo modificano esplicitamente.
- **EFLAGS (bit di stato)**: questo registro contiene diversi bit utili sia alla CPU sia al programmatore. I bit principali determinano i cosiddetti **condition code**: questi bit 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:
 - **CF (bit 0)**: attivo quando il risultato ha determinato riporto (**carry**).
 - **PF (bit 2)**: attivo quando il risultato ha "**parità pari**", ovvero numero di "uni" o "zeri" pari.
 - **AF (bit 4)**: attivo quando il risultato ha determinato riporto intermedio sul bit 3 (**auxiliary carry**); utile in codifica BCD.
 - **ZF (bit 6)**: attivo quando il risultato è **zero**
 - **SF (bit 7)**: bit **segno**; attivo quando il risultato è negativo.
 - **OF (bit 11)**: attivo quando il risultato ha causato **overflow** (traboccamento) con operazioni in aritmetica intera **con segno**.



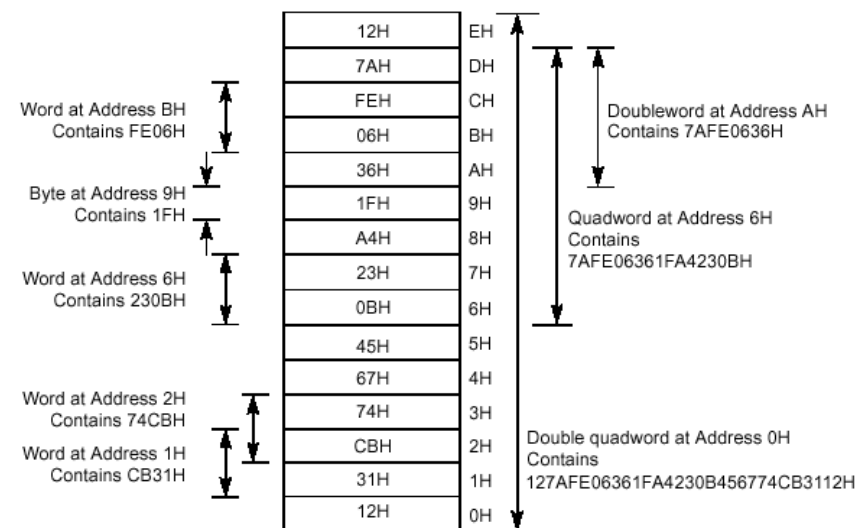
Altri bit di EFLAGS (vedi figura) sono dedicati al modo di funzionamento (**reale, virtuale, protetto**) e a particolari modalità di funzionamento per operazioni di debugging di programmi (esecuzione **step by step, interrupt**, ...).

- Esistono inoltre altri **registri di sistema** GDTR, IDTR, LDTR, TR che contengono i puntatori a tabelle di sistema importanti (es: **IDTR** = **I**nterrupt **D**escription **T**able **R**egister), e altri registri utilizzati per il **debugging** di programmi e per il **supporto** della **cache**.

Tipi di dati del Pentium (1)



I dati in memoria: **attenzione il Pentium è Little Endian !**

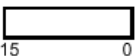


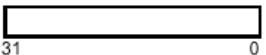
Tipi di dati del Pentium (2)

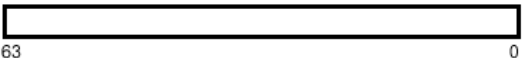
Attenzione alla differenza tra **aritmetica**:


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

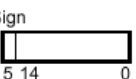
da 0 a 255  Byte Unsigned Integer

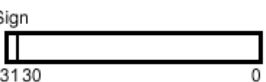
da 0 a 65.535  Word Unsigned Integer

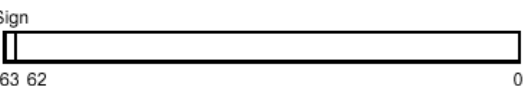
da 0 a $2^{32}-1 = 4.294.967.295$  Doubleword Unsigned Integer

da 0 a $2^{64}-1$  Quadword Unsigned Integer

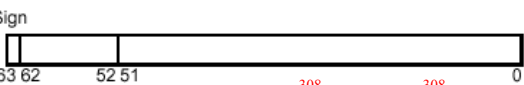
da -128 a +127  Byte Signed Integer

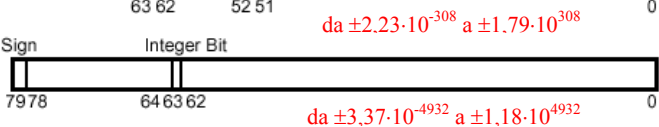
da -32.768 a 32.767  Word Signed Integer

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

 Double Precision Floating Point

da $\pm 2,23 \cdot 10^{-308}$ a $\pm 1,79 \cdot 10^{308}$  Double Extended Precision Floating Point

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 caricare 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 1 solo operando e istruzioni a 2 o più operandi.

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

Esempi:

```
MOV EAX,10           // indirizzamento immediato
MOV EAX,[10345467h]  // indirizzamento diretto
MOV EAX,EBX           // indirizzamento registro
MOV EAX,[ECX]         // indirizzamento indiretto
MOV EAX,[ECX+2]       // indirizzamento ind. con offset
...
```

Ogni ISA è caratterizzato da una serie di **modi di indirizzamento** ciascuno dei quali specifica le modalità di reperimento di operandi da parte della CPU.

Grazie al programma assemblatore possiamo 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 diretto
```


Modi di Indirizzamento (2)

- **Indirizzamento immediato:** l'operando contiene **direttamente un valore** costante (e non un indirizzo di memoria); 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,d3c5h       // carica il numero esad. d3c5 in AX
MOV EAX,104ed3c5h  // carica il numero esad. 104ed3c5 in EAX

MOV AX,d3c5001ah   // 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 nessun operando dalla memoria.

- **Indirizzamento diretto (assoluto):** l'operando **specifica un indirizzo** di memoria; grazie all'assemblatore è possibile utilizzare nomi simbolici.

```
MOV AL,[104532a0h]  // carica in AL il byte alla locazione
                    104532a0
MOV EAX,[104ed3c5h] // carica in EAX la double word alla
                    locazione 104ed3c5
MOV AX,pippo        // carica in AX la word specificata dalla
                    variabile pippo
MOV [104ed3c5h],EAX // salva alla locazione 104ed3c5 il
                    contenuto del registro EAX (4 byte)
```

Attenzione, il Pentium lavora in modo **Little Endian**, e quindi il byte basso è il primo memorizzato a partire dall'indirizzo, ...

Modi di Indirizzamento (3)

- **Indirizzamento dei registri:** come l'indirizzamento diretto ma invece di specificare una locazione di memoria si specifica un **registro**.

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

Attenzione alle dimensioni (devono essere compatibili) !

- **Indirizzamento indiretto dei registri:** l'operando che viene specificato viene caricato o salvato in memoria, ma l'indirizzo di memoria non è cablato nell'istruzione ma **specificato da un registro**.

```
MOV EAX,[ECX]       // carica in EAX la double word alla
                    locazione di memoria indicata da ECX
```

Attenzione, in caso di omissione delle [] l'indirizzamento **non è indiretto**: l'istruzione è comunque valida anche la semantica completamente differente.

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

```
MOV EAX,Vettore[ECX] // carica in EAX la double word alla
                    locazione di memoria il cui indirizzo
                    è indicato da Vettore+ECX
```

In questo modo, è possibile accedere ad esempio 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 double word ?

Modi di Indirizzamento (4)

- **Indirizzamento con offset**: l'operando che viene specificato viene caricato o salvato in memoria, l'indirizzo di memoria non è cablato nell'istruzione ma **determinato durante il funzionamento sulla base di un offset calcolato nel modo seguente**:

$$\text{Offset} = \text{Base} + (\text{Indice} \times \text{Scala}) + \text{Spiazzamento}$$

dove:

- **Base**: se presente è specificata da un registro (es. [EAX]). *Utilizzato normalmente per indicare un indirizzo di partenza variabile.*
- **Indice**: può essere solo un registro. *Utilizzato normalmente per scorrere gli elementi di un vettore durante un ciclo.*
- **Scala**: assume valore costante pari a 2, 4 o 8; può essere omessa (scala =1). *Utilizzato normalmente come "passo" di avanzamento nel vettore; ovvero se ad ogni lettura devo leggere un valore di 4 byte devo avanzare in memoria con passi di 4 e non di un byte.*
- **Spiazzamento**: assume valore costante (8-bit, 16-bit o 32 bit); può essere omesso. *Utilizzato normalmente per accedere a un vettore o una struttura a partire da una certa posizione;*

esempi:

```
MOV AL,Vettore[ECX*4+40] // carica in AL il byte all'indirizzo
                          Vettore+ECX*4+40. In questo caso:
                          Vettore+40 costituisce l'offset
                          "pre-calcolato" dal compilatore

MOV EAX,[EBX][EDX]      // carica in EAX la double word
                          all'indirizzo EBX+EDX
```

con questo tipo di indirizzamento è possibile accedere a tutti gli elementi di un **vettore di double word**:

```

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

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

Istruzioni del Pentium (1)

Il Pentium, come in genere tutte le CPU di categoria CISC, è dotato 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 del Pentium. 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 del Pentium (2)

- **MOV**: ne abbiamo già discusso a lungo. Si riporta l'elenco dei modi di utilizzo estratto da manuale Intel:

Instruction	Description	
MOV <i>r/m8,r8</i>	Move <i>r8</i> to <i>r/m8</i>	r8, r16 ed r32 specificano un registro a 8, 16 o 32 bit
MOV <i>r/m16,r16</i>	Move <i>r16</i> to <i>r/m16</i>	
MOV <i>r/m32,r32</i>	Move <i>r32</i> to <i>r/m32</i>	r/m8, r/m16 ed r/m32 specificano un registro oppure indirizzo di memoria per una parola a 8, 16 o 32 bit
MOV <i>r8,r/m8</i>	Move <i>r/m8</i> to <i>r8</i>	
MOV <i>r16,r/m16</i>	Move <i>r/m16</i> to <i>r16</i>	
MOV <i>r32,r/m32</i>	Move <i>r/m32</i> to <i>r32</i>	
MOV <i>r/m16,Sreg**</i>	Move segment register to <i>r/m16</i>	fanno uso di segmenti
MOV <i>Sreg,r/m16**</i>	Move <i>r/m16</i> to segment register	
MOV AL, <i>moffs8*</i>	Move byte at (<i>seg:offset</i>) to AL	
MOV AX, <i>moffs16*</i>	Move word at (<i>seg:offset</i>) to AX	
MOV EAX, <i>moffs32*</i>	Move doubleword at (<i>seg:offset</i>) to EAX	
MOV <i>moffs8*,AL</i>	Move AL to (<i>seg:offset</i>)	
MOV <i>moffs16*,AX</i>	Move AX to (<i>seg:offset</i>)	
MOV <i>moffs32*,EAX</i>	Move EAX to (<i>seg:offset</i>)	
MOV <i>r8,imm8</i>	Move <i>imm8</i> to <i>r8</i>	imm8, imm16 ed imm32 specificano un dato immediato (costante) a 8, 16 o 32 bit.
MOV <i>r16,imm16</i>	Move <i>imm16</i> to <i>r16</i>	
MOV <i>r32,imm32</i>	Move <i>imm32</i> to <i>r32</i>	
MOV <i>r/m8,imm8</i>	Move <i>imm8</i> to <i>r/m8</i>	
MOV <i>r/m16,imm16</i>	Move <i>imm16</i> to <i>r/m16</i>	
MOV <i>r/m32,imm32</i>	Move <i>imm32</i> to <i>r/m32</i>	

NOTA: non è possibile un'istruzione del tipo **MOV pippo,pluto** che copia valori da memoria a memoria.

- **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**

Vedremo nel seguito esempio di utilizzo dello stack (con ISA JVM)

Istruzioni del Pentium (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 (normalmente **un registro a 32 bit**) l'indirizzo di SRC (un riferimento a memoria, normalmente **il nome di una variabile**).

LEA EAX,[10456de4h] // EAX = 10456de4

LEA EAX,pippo // EAX = 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 del Pentium (4)

- **CMOVcc 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 flags**: **OF, SF, ZF, AF, CF, e PF**

Istruzioni del Pentium (5)

- **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 flags**: **OF, SF, ZF, AF, CF, e PF**

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

Instruction	Description
MUL r/m8	Unsigned multiply (AX ← AL * r/m8)
MUL r/m16	Unsigned multiply (DX:AX ← AX * r/m16)
MUL r/m32	Unsigned multiply (EDX:EAX ← 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 del Pentium (6)

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 del Pentium (7)

- **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 differentemente 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: se **SRC = 0** viene generato errore run-time (eccezione) !

Istruzioni del Pentium (8)

- **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 quanto OF è influenzato solo dalle operazioni in aritmetica intera con segno (es. IMUL, IDIV) e se considerassimo ffffffffh come signed (-1 in complemento a 2) il suo incremento darebbe 0 e quindi nessun traboccamento. 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.

Istruzioni del Pentium (9)

Operazioni logiche e spostamento di 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 -> AZZERAZIONE EAX
XOR EBX,0a0b0c0dh // EBX = EBX XOR 0a0b0c0dh
```

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
```

Istruzioni del Pentium (10)

- **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
```

Istruzioni del Pentium (11)

- **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 del Pentium (12)

- **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 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 flags. 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 del Pentium (13)

- **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 (unsigned)
...
```


Istruzioni del Pentium (14)

L'elenco dei **condition code**, inclusivo dei rispettivi codici mnemonici e corrispondenza in termini di flags è riportato nel lucido successivo. Nella pratica, l'utilizzo dei **codici mnemonici** consente spesso di "ignorare" il funzionamento in termini di flags. 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 del Pentium (15)

EFLAGS Condition Codes

Mnemonic (cc)	Condition Tested For	Status Flags Setting
O	Overflow	OF = 1
NO	No overflow	OF = 0
B NAE	Below Neither above nor equal	CF = 1
NB AE	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 del Pentium (16)

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 del Pentium (17)

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 posso 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).

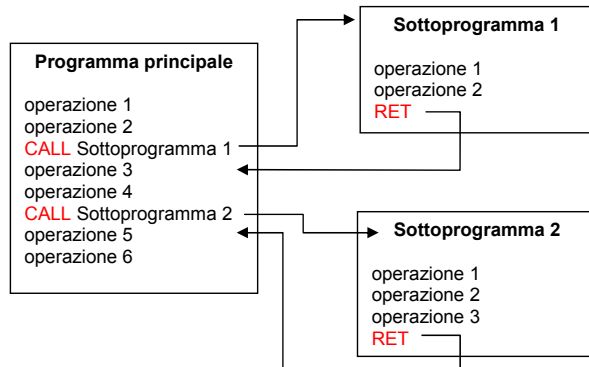
```
MOV ECX,10      // Valore iniziale di ECX
XOR EAX,EAX
Ciclo: ADD EAX, Vettore[ECX*4-4]
      CMP EAX,100
      LOOPNE Ciclo
```

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 del Pentium (18)

- **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 del Pentium (19)

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 del Pentium (20)

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.*

Il Pentium mette a disposizione una serie di **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).

Istruzioni del Pentium (21)

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 MOVS           // 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.

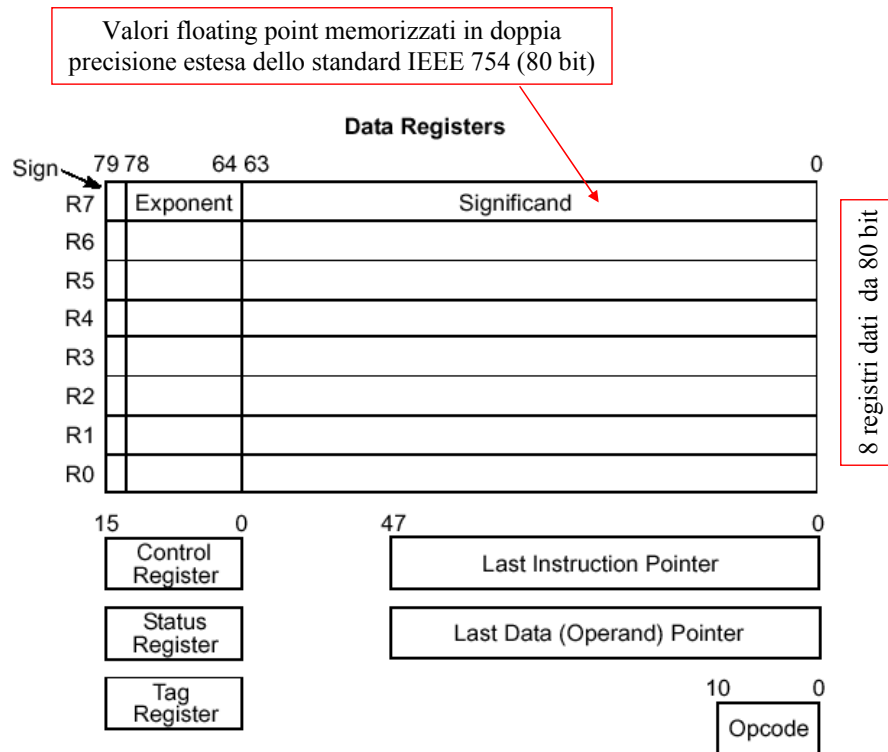
```
CLD                // Scorrimento in avanti
MOV ECX,256        // Numero di elementi
LEA ESI,IndBlocco  // Indirizzo di partenza in ESI
LEA EDI,IndBlocco2 // Indirizzo di partenza in EDI
REPE CMPS          // Confronta [ESI] con [EDI] e setta il flag
                    // di conseguenza, decrementa 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).

Istruzioni del Pentium (22)

Unità Floating Point (cenni)

L'unità floating point (incorporata all'interno del chip del Pentium) 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.

Istruzioni del Pentium (23)

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**).

Istruzioni del Pentium (24)

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.

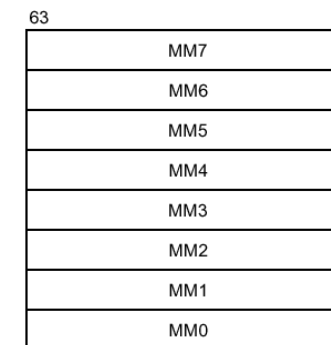
Istruzioni del Pentium (25)

Istruzioni MMX (cenni)

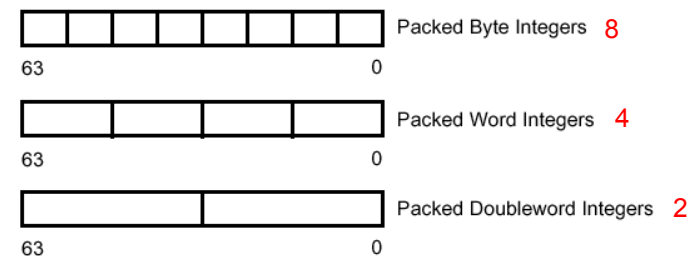
MMX nasce a partire dal **Pentium Pro** e viene incorporato **su tutti i processori Intel successivi** (Pentium II, Pentium III, Pentium 4, ...). Si tratta di un potente meccanismo che consente di utilizzare il processore come macchina **SIMD** (**S**ingle **I**nstruction **M**ultiple **D**ata), ovvero 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**, introduce **8 nuovi registri a 64 bit**:



e **tre nuovi tipi di dato**:



Istruzioni del Pentium (26)

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, PADDQ	PADDSB, PADDQW, PADDQ	PADDUSB, PADDUSW, PADDUSQ
	Subtraction	PSUBB, PSUBW, PSUBD	PSUBSB, PSUBQW, PSUBQ	PSUBUSB, PSUBUSW, PSUBUSQ
	Multiplication	PMULL, PMULH		
	Multiply and Add	PMADD		
Comparison	Compare for Equal	PCMPEQB, PCMPEQW, PCMPEQD		
	Compare for Greater Than	PCMPGTB, PCMPGTPW, PCMPGTPD		
Conversion	Pack		PACKSSWB, PACKSSDW	PACKUSWB
Unpack	Unpack High	PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ		
	Unpack Low	PUNPCKLBW, PUNPCKLWD, PUNPCKLDQ		
			Operazioni a 64 bit	
Logical	And	Packed	Full Quadword	
	And Not		PAND, PANDN	
	Or		POR	
	Exclusive OR		PXOR	
Shift	Shift Left Logical	PSLLW, PSLLD, PSRLW, PSRLD, PSRAW, PSRAD	PSLLQ, PSRLQ	
	Shift Right Logical			
	Shift Right Arithmetic			
Data Transfer	Register to Register	Doubleword Transfers	Quadword Transfers	
	Load from Memory		MOVQ, MOVQ, MOVQ	
	Store to Memory			
Empty MMX State		EMMS		

Istruzioni del Pentium (27)

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 da 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};
...

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

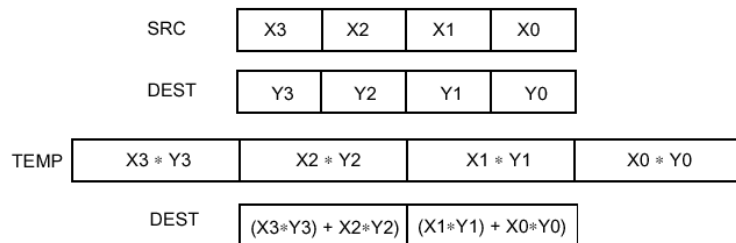
Istruzioni del Pentium (28)

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



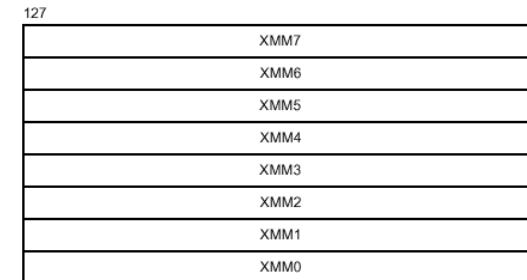
```
// dichiarazione in C
short V1[]={2,4,8,16};
short V2[]={2,4,8,16};
```

```
...
EMMS                // inizializza MMX
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 !
```

Pentium II, III e 4 (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).



- 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.

Pentium II, III e 4 (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 2006) disponibili sui processori Intel multi Core (a partire da Core 2 Duo).