

## Contents

[hide]

### 1 Assembly MMX

- 1.1 Link e Riferimenti
- 1.2 Essay
- 1.3 SIMD Execution Model
- 1.4 Packed Data Types
  - 1.4.1 64bit Packed Data Types
  - 1.4.2 128bit Packed Data Types
- 1.5 MMX Technology
  - 1.5.1 MMX registers
  - 1.5.2 Saturazione e wraparound

modes

- 1.5.3 MMX instructions
- 1.5.4 Example
- 1.6 SSE Technology
- 1.7 Note Finali
- 1.8 Disclaimer



## Link e Riferimenti

Manuali Intel  
 MASM Programmer's Guide  
**Masm 9**

Col Pentium II sono state introdotte delle estensioni all'architettura del processore: **MMX** e **SSE**, che permettono di effettuare le operazioni **SIMD** (single-instruction multiple-data).

Ognuna di queste estensioni fornisce un gruppo di istruzioni, che operano tramite i **packed integer** ed i **packed floating point**, usando i registri MMX (a 64bit) o quelli XMM (a 128bit).

## SIMD Execution Model

Il modello SIMD velocizza di molto le performance del software, permettendo di operare su diversi dati in parallelo con una sola istruzione.

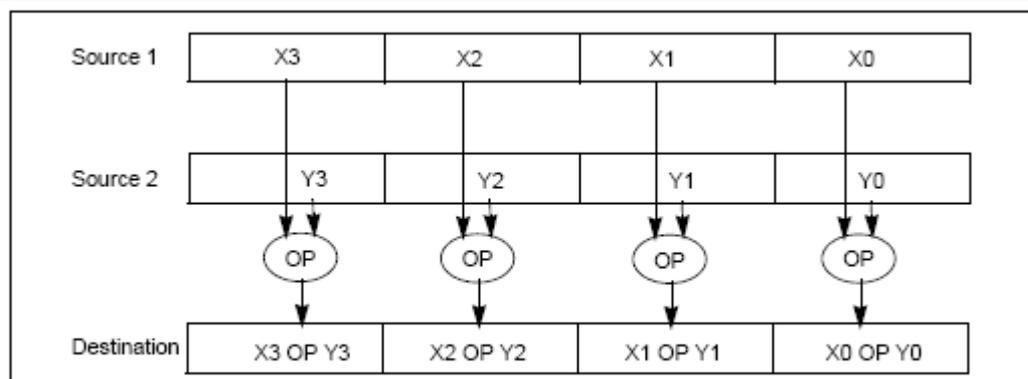


Figure 9-4. SIMD Execution Model

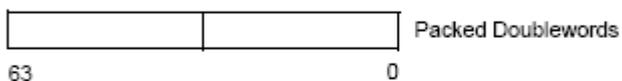
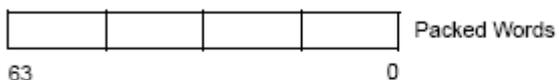
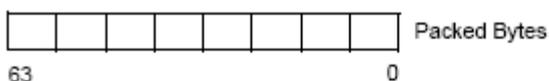
# Packed Data Types

I Packed Data Type sono di due tip: 64 e 128 bit, e sono la versione digitale dei vettori.

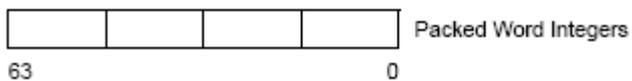
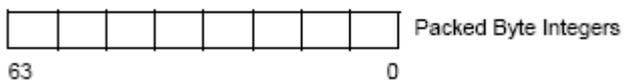
## 64bit Packed Data Types

I 64-bit packed SIMD data types sono stati introdotti con la tecnologia MMX. I tipi fondamentali sono: packed byte, packed word, packed dword e possono contenere soltanto valori integer.

### Fundamental 64-Bit Packed SIMD Data Types



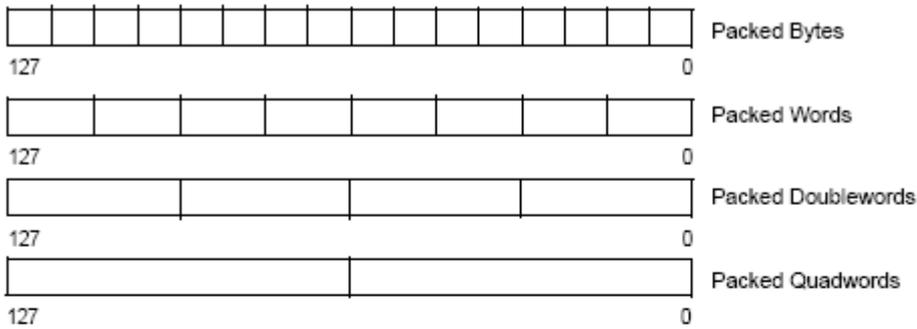
### 64-Bit Packed Integer Data Types



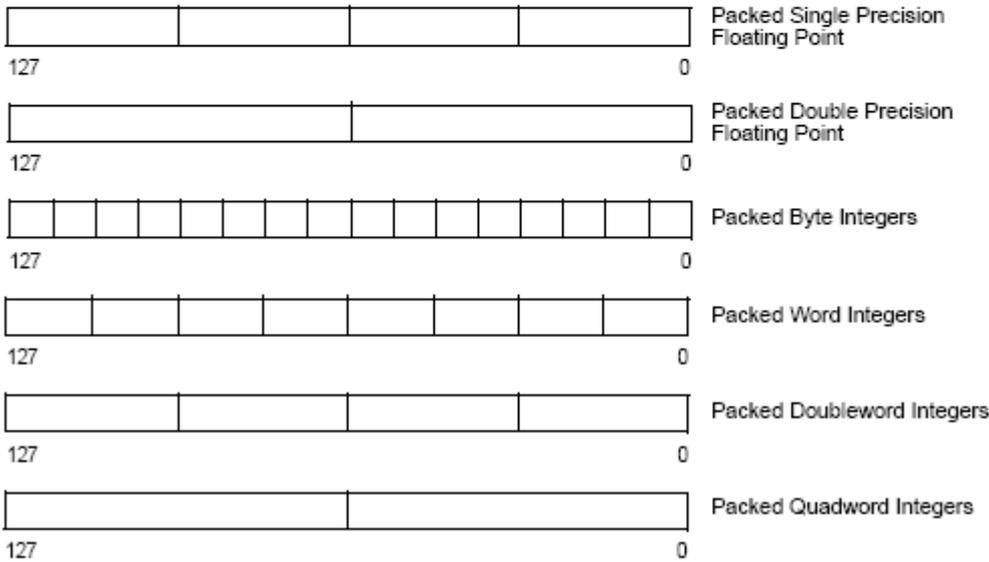
## 128bit Packed Data Types

I 128-bit packed SIMD data types sono stati introdotti con la tecnologia XMM. I tipi fondamentali sono: packed byte, packed word, packed dword e packed qword. Questo tipo di packed possono contenere valori integer o a virgola mobile(floating point)

### Fundamental 128-Bit Packed SIMD Data Types



### 128-Bit Packed Floating-Point and Integer Data Types

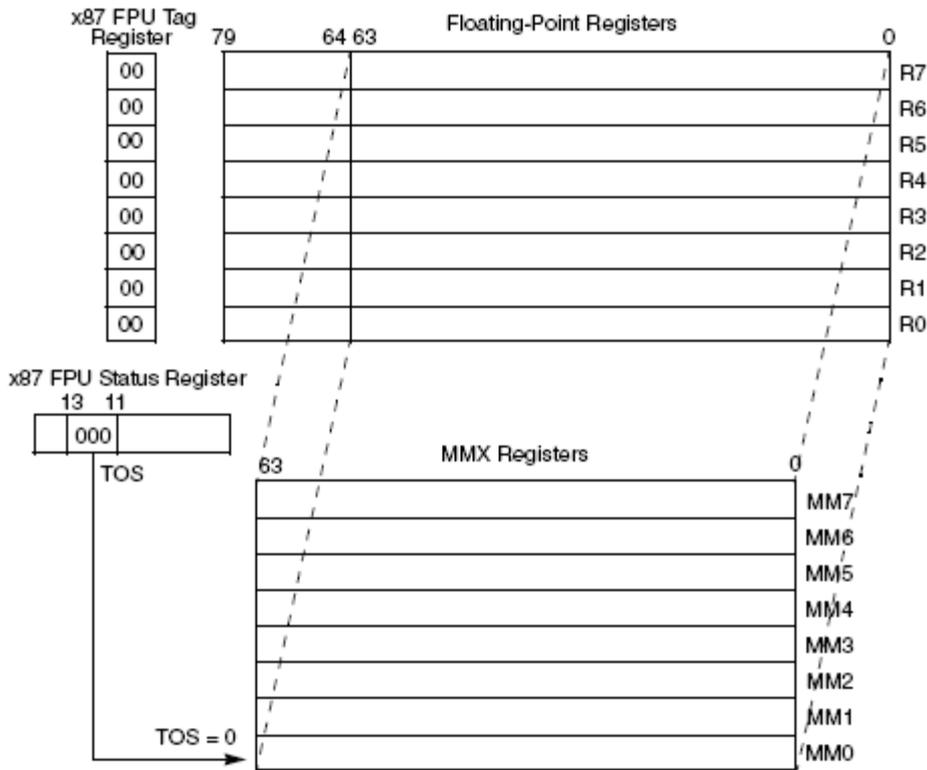


## MMX Technology

La tecnologia MMX ha introdotto il concetto di operazioni SIMD , i packed data types, e poi l' **ambiente MMX** .

### MMX registers

L'ambiente MMX implementa otto registri da 64bit, utilizzati per effettuare operazioni sui packed data. I registri MMX sono "contenuti" nei registri della FPU, in effetti, si agisce fisicamente sui i registri della FPU, l'immagine chiarisce meglio questo concetto.



E' possibile accedere ai registri con modalità a 32bit ed a 64bit.

In entrambe le modalità d'accesso alla memoria, i registri svolgono le seguenti operazioni:

- Accesso alla memoria secondo la codifica **little indian**
- Trasferimento dati da un registro MMX all'altro
- Istruzioni per unpack/pack

## Saturazione e wraparound modes

Quando si esegue un' operazione aritmetica, essa può risultare fuori range, es: sommando due word, si può ottenere un overflow se il risultato è maggiore di 16bit.

Per ovviare a ciò MMX implementa due tecniche:

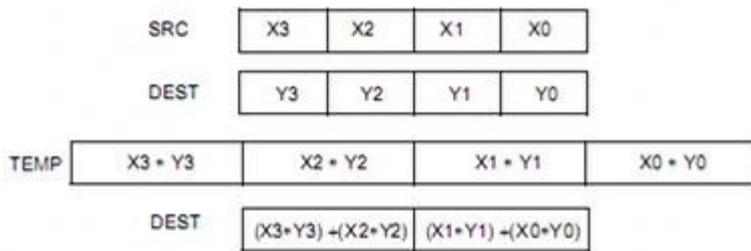
- **Wraparound:** con questa tecnica il risultato è truccato: il carry o l'overflow bit è ignorato e solo l'ultimo bit significativo del risultato è ritornato alla destinazione.
- **Signed and Unseigned Saturation:** il risultato è limitato alla grandezza del packed data operante: se l'addizione tra due word packed è maggiore di 16bit, il risultato è saturato(per i signed) a 7FFF, se avviene un positive overflow, o 8000, se avviene un negative overflow o è un'operazione unsigned.

## MMX instructions

Ecco a voi una tabella contenente le istruzioni supportate dalla tecnologia MMX

Category		Wraparound	Signed Saturation	Unsigned Saturation
Arithmetic	Addition	PADDB, PADDW, PADD	PADD SB, PADD SW	PADD USB, PADD USW
	Subtraction	PSUBB, PSUBW, PSUBD	PSUB SB, PSUB SW	PSUB USB, PSUB USW
	Multiplication Multiply and Add	PMULL, PMULH 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	
Shift	Shift Left Logical Shift Right Logical Shift Right Arithmetic	PSLLW, PSLLD PSRLW, PSRLD PSRAW, PSRAD	PSLLQ PSRLQ	
		Doubleword Transfers	Quadword Transfers	
Data Transfer	Register to Register Load from Memory Store to Memory	MOVB MOVD MOVQ	MOVQ MOVQ MOVQ	
		Empty MMX State	EMMS	

- **EMMS:** quest'istruzione svuota lo stato dell'ambiente MMX settando a 11B il FPU tag word. Quest'istruzione deve essere inserita alla fine di ogni blocco di codice che utilizzi le istruzioni MMX, prima di eseguire codice per la FPU.
- **Arithmetic Instructions:**
  - **PADDB/PADDW/PADD:** sono le istruzioni per l'addizione dei packed data considerandoli come vettori di byte/word/dword
  - **PSUBB/PSUBW/PSUBD:** solo le istruzioni per la sottrazione dei packed data.
  - **PADDUSB/PADDSW:** aggiunge i packed data, considerati come numeri con segno(byte e word), usando la signed saturation.
  - **PSUBSB/PSUBSW:** come sopra, solo che queste istruzioni vengono usate per effettuare la sottrazione
  - **PADDUSB/PADDUSW:** aggiunge i packed data, considerandoli come numeri senza segno(byte e word), usando l'unsigned saturation.
  - **PSUBSUB/PSUBUSW:** come sopra ma utilizzati per la sottrazione.
  - **PMULHW/PMULHUW:** moltiplica due packed signed/unsigned word e salva i 16bit più alti, dal bit 31 al 16, dei 32bit risultanti della moltiplicazione.
  - **PMADDWD:** Moltiplica ed aggiunge due packed word integer: per farla breve:



- **Comparison Instructions:**

- **PCMPEQB:** Compare packed bytes for equal
- **PCMPEQW:** Compare packed words for equal
- **PCMPEQD:** Compare packed doublewords for equal
- **PCMPGTB:** Compare packed signed byte integers for greater than
- **PCMPGTW:** Compare packed signed word integers for greater than
- **PCMPGTD:** Compare packed signed doubleword integers for greater than

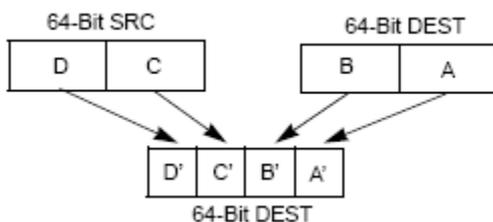
- **Logical Instructions:**

- **PAND:** bitwise logical AND
- **PANDS:** bitwise logical AND NOT
- **POR:** bitwise logical OR
- **PXOR:** bitwise logical XOR

- **Shift and Rotate Instructions:**

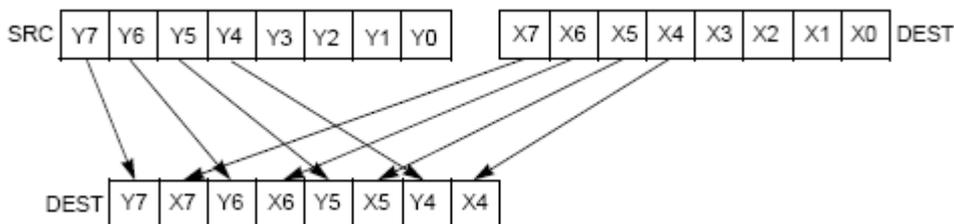
- **PSLLW:** shift packed words left logical
- **PSLLD:** shift packed dwords left logical
- **PSLLQ:** shift packed qwords left logical
- **PSRLW:** shift packed words right logical
- **PSRLD:** shift packed dwords right logical
- **PSRLQ:** shift packed qwords right logical
- **PSRAW:** shift packed words right arithmetic
- **PSRAD:** shift packed dwords right arithmetic

- **Conversion instructions:** Le istruzioni **PACKSSWB** (pack word in byte con signed saturation), **PACKSSDW** (pack dword in word con signed saturation) e **PACKUSWB** (pack word in byte con l'unsigned saturation): sono delle istruzioni di conversione. Per comprendere meglio questo tipo d' istruzioni guardate quest'immagine:

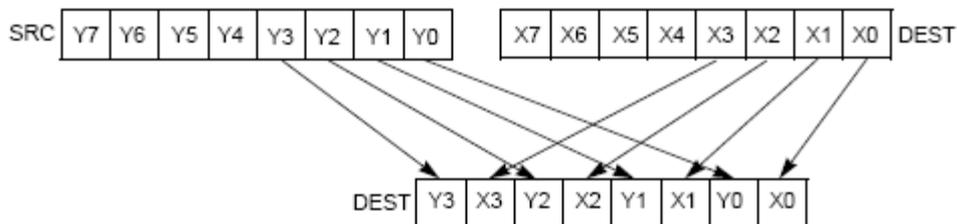


- **Unpack instructions:** Le istruzioni di unpack sono di due tipi: quelle che operano sui High Data e quelli sui Low Data:

- *High Data:* estraggono ed intervallano gli high-order data element (prende dalla parte superiore del registro i byte/word/dword/qword) dalla sorgente e dal destinatario nel registro del destinatario.



- *Low data:* come sopra, solo che operano sui low data (la parte inferiore del registro)



Per approfondire le altre istruzioni vi rimando ai manuali intel volume 1 e 2.

## Example

Ora mostro un semplice programma che effettua delle operazioni sui vettori (secondo voi i packed data cosa sono ;).

### Codice per MASM/x86

```
.686
.MMX
.model flat, stdcall

include windows.inc
include kernel32.inc
include user32.inc
includelib kernel32.lib
includelib user32.lib

.data
r1 dw 0,0,-2,712 ;array di word
r2 dw 0,2,4,-10
r3 dw 0,2,4,5

Titolo db "ciao",0
Testo1 db "minore",0
Testo2 db "maggiore",0

.code
Main:

;verifico che il processore supporti MMX
mov eax, 1
cpuid
test edx, 0800000h
jz exit

movq MM0, qword ptr [r1] ;copio tre qword nei registri MMX
movq mm1, qword ptr[r2]
movq mm2, qword ptr[r3]; rappresentato come 0005 0004 0002 0000, secondo codifica little
indian
paddsb mm0, mm1 ;addizione due vettori
psubsb mm2,mm0 ;sottraggo due vettori
movq qword ptr[r1],mm0
pcmpeqb mm0, mm1 ;faccio una comparazione
jb minor
invoke MessageBox, NULL, OFFSET Testo2,OFFSET Titolo, 0
jmp exit
minor:
invoke MessageBox, NULL, OFFSET Testo1, OFFSET Titolo, 0
exit:
emms ;è d'obbligo
invoke ExitProcess, NULL

end Main
```

### Codice per ml64/x64

```
extrn MessageBoxA : proc
extrn ExitProcess : proc

.data
r1 dw 0,0,-2,712 ;array di word
```

```

r2 dw 0,2,4,-10
r3 dw 0,2,4,-5

Titolo db "ciao",0
Testo1 db "minore",0
Testo2 db "maggiore",0

.code
Main proc
sub rsp, 28h

;verifico che il processore supporti MMX
mov eax, 1
cpuid
test edx, 0800000h
jz exit

movq MM0, qword ptr [r1] ;copio tre qword nei registri MMX
movq mm1, qword ptr[r2]
movq mm2, qword ptr[r3]
paddsb mm0, mm1 ;addizione due vettori
psubsb mm2,mm0 ;sottraggo due vettori
movq qword ptr[r1],mm0
pcmpeqb mm0, mm1 ;faccio una comparazione
jb minor
xor r9, r9
lea r8, Titolo
lea rdx, Testo2
xor rcx, rcx
call MessageBoxA
jmp exit
minor:
xor r9, r9
lea r8, Titolo
lea rdx, Testo1
xor rcx, rcx
call MessageBoxA
exit:
emms ; e' d'obbligo
xor rcx, rcx
call ExitProcess

Main endp
end

```

## Le istruzioni SSE in ambito IA32

# Contents

[hide]

## 1 Le istruzioni SSE in ambito IA32

### 1.1 Introduzione

### 1.2 Tools & Files

### 1.3 Essay

### 1.4 SIMD

### 1.5 Istruzioni SIMD per calcoli floating

#### 1.5.1 Istruzioni Matematiche di Base

#### 1.5.2 Istruzione radice quadrata

#### 1.5.3 Istruzioni di approssimazione rapida

#### 1.5.4 Istruzioni di comparazione Min/Max

#### 1.5.5 Istruzione di Shuffle

#### 1.5.6 Istruzione di Unpack

#### 1.5.7 Istruzioni di Movimento Dati (Data Movement)

#### 1.5.8 Istruzione di Movimento BitMask

#### 1.5.9 Istruzioni di Confronto e Set BitMask

#### 1.5.10 Istruzioni Logiche

#### 1.5.11 Istruzioni di Confronto e Set EFLAGS

#### 1.5.12 Istruzioni di Conversione

troncamento

- 1.6 Istruzioni SIMD per calcoli integer
  - 1.6.1 Istruzione di Estrazione
  - 1.6.2 Istruzione di Inserimento
  - 1.6.3 Istruzioni di Confronto MIN/MAX su interi
  - 1.6.4 Istruzione di Movimento BitMask intero
  - 1.6.5 Istruzione di Moltiplicazione Unsigned con
  - 1.6.6 Istruzione di Shuffle su registri MMX
  - 1.6.7 Istruzioni di Media
  - 1.6.8 Istruzioni di Somma delle differenze
- 1.7 Istruzioni SIMD per Controllo della Cache
  - 1.7.1 Istruzioni di Streaming Store
  - 1.7.2 Istruzione di Movimento MaskedBytes
  - 1.7.3 Istruzione Store Fence
  - 1.7.4 Istruzione di Prefetch
- 1.8 Istruzioni SIMD per la Gestione dello Stato
  - 1.8.1 Istruzioni Load/Store
- 1.9 Scrivere codice Intrinsic col VisualC
- 1.10 Organizzare i dati, AoS e SoA
- 1.11 Piccolo programma di test per le SSE
  - 1.11.1 Considerazioni
- 1.12 Note Finali
- 1.13 Disclaimer



## Introduzione

A questo giro scomoderemo le istruzioni SSE. La stesura dell'articolo risale all'agosto del 2004, a quei tempi c'erano soltanto le istruzioni SSE e SSE2, e io avevo un P3, adesso le cose sono cambiate un attimino... però le cose che scrissi allora sono tutt'oggi valide, anche se inevitabilmente inutili. Mi scuso con Pnluck per avergli duplicato l'articolo, in realtà era pronto nel mio hd già nel 2005, però non l'ho mai tirato fuori quando per un motivo, quando per l'altro. Cercherò di far seguire a questo articolo un altro che parli delle SSE2, anch'esse ormai abusate, e SSE3 e 4, inoltre penso che valga la pena indagare un po' meglio le istruzioni 'intrinsic' messe a disposizione includendo nei propri progetti l'include <xmmintrin.h> (MMX e SSE) e <emmintrin.h> (SSE2).

## Tools & Files

Solo il VisualC++ o un compilatore di vostra scelta, basta che supporti l'inline delle istruzioni assembler (direttiva `_asm`). Se proprio volete divertirvi anche l'Intel Compiler (adesso alla versione 10, molto bello) o il VectorC della Codemaster (ormai sorpassato), ma non sono necessari.

*Agosto 2004* Il programma dell'esempio fa del calcolo vettoriale senza scomodare il coprocessore matematico, ma usando le più veloci istruzioni SSE, sì lo so, le SSE2 sono ancora meglio, gestiscono numeri in virgola a doppia precisione (64bit), a breve (??) cercherò di fare un articoletto anche su quelle. Avrete il supporto SSE se installate l'estensione "ProcessorPack" (solo per visual 6.0, il Visual lo ha già) e non importa fare inlining di istruzioni assembler, basta appoggiarsi all'include `xmmintrin.h` (vi consiglio di spulciarlo), noi vedremo ambedue i metodi.

*Ottobre 2005* NB: dati i tempi biblici con cui scrivo i tutorial, adesso, a quasi un anno di distanza da quando sono partito a scrivere, ho finalmente cambiato il P3 in favore di un P4, quindi ora ho anche le SSE2. Per fortuna quello che ho scritto fino ad ora è sempre valido...:-P cercherò di finire di scrivere il tutorial prima che i processori neurali arrivino nelle vostre case....

Gennaio 2009' NBB: come supponevo i tempi biblici sono diventati ere geologiche, altro che un anno di distanza!! Siamo ormai a ben cinque (5!!) anni, che schifo..comunque ora esistono le SSE3 e le SSE4, i P4 adesso ci sono anche quad core e oltre, e il parallelismo di calcolo si è spostato dalla CPU alla GPU, e la sta facendo da padrone nVidia con la sua tecnologia CUDA. Forse (e dico : forse) è la volta buona che finisco questo articolo, oppure per davvero è la volta buona che i processori neurali arrivano nelle vostre case...comunque ancora una volta: che schifo.

## SIMD

Prima di tutto decriptiamo questo cavolo di acronimo, SSE sta per "Streaming SIMD Extension" che al suo interno contiene un altro acronimo che spiega al volo la natura di queste istruzioni; **SIMD** ovvero "**S**ingle **I**nstruction **M**ultiple **D**ata" cioè la capacità di manipolare più dati allo stesso tempo con una sola istruzione (velocizzando ovviamente il tutto). Non capisco come mai gli americani abbiano questa mania degli acronimi, se continuano di questo passo parleranno come i quindicenni con gli sms.....vabbè chi se ne frega. Le istruzioni SSE possono essere suddivise in 4 categorie fondamentali:

- Istruzioni SIMD per calcoli floating (in single precision).
- Istruzioni SIMD per calcoli integer.
- Istruzioni per il controllo della cache L1 e L2.
- Istruzioni per la gestione dello stato della "macchina" SSE.

Le SSE nascono sulle ceneri delle MMX, e non condividono i registri del coprocessore matematico come loro, ma bensì introducono un set di 8 nuovi registri a 128bit, che vanno da XMM0 a XMM7 e un registro di stato MXCSR. Questi registri si occupano solo di fare calcoli e non hanno a che vedere niente con l'indirizzamento della memoria, praticamente SSE è come se fosse un nuovo coprocessore che offre ulteriori funzionalità prettamente dedicate al calcolo matematico necessario alla tristemente famosa "multimedialità" un termine che qualche anno fa (circa 8 o 9) riempiva la bocca di tutti gli esperti (o meno) di informatica, se non avevi il computer "multimediale" non eri nessuno...ma cosa mai volevano dire? Boh. Viene anche introdotto un nuovo tipo di dati, che si rifà allo standard IEEE-754, che definisce un numero a 32bit a singola precisione che ha 1bit per il segno, 8bit per l'esponente e 23bit per la mantissa. Le SSE quindi possono gestire 4 di questi numeri contemporaneamente (i registri sono da 128bit), una scheggia! Le istruzioni si possono a loro volta suddividere in due grandi macrofamiglie:

- Istruzioni SIMD per calcoli **PACKED**.
- Istruzioni SIMD per calcoli **SCALAR**.

La differenza? Le istruzioni "Packed" operano contemporaneamente su quattro paia di float alla volta, si possono riconoscere dal suffisso **ps** (ad esempio addps, mulps, subps e divps) mentre le istruzioni "Scalar" operano soltanto sulla parte meno significativa dei registri implicati (i primi 32bit), lasciando inalterato il valore della restante parte alta del registro destinazione ovvero gli ultimi 96bit. Si possono riconoscere dal suffisso **ss** (quindi addss, mulss, subss e divss). Sono state aggiunte ben 70 nuove istruzioni, 50 dedicate ai calcoli SIMD-FloatingPoint, 12 "NewMedia Instructions" che utilizzano i registri MMX e lavorano sugli interi, un nuovo registro di stato, il primo da quando il processore i386 aggiunse l'x87-FP, istruzioni per la gestione della cache e del nuovo registro di stato. Le istruzioni SSE da 128bit sono scomposte in due micro-operazioni interne a 64bit, il che permette di riutilizzare e minimizzare le modifiche alle precedenti unità di decodifica. La microarchitettura è essenzialmente quella del PentiumII, cui sono state apportate delle modifiche alle unità dispatch/execute. Sulla Port0 è stato aggiunto il moltiplicatore per packed FP, raddoppiando quello già esistente, sulla Port1 sono state aggiunte le nuove unità che sono:

1. un sommatore per 2 packed FP alla volta con throughput di 1ciclo e latenza di 3cicli, questa unità esegue anche sottrazioni, min/max, confronto e conversione.
2. un'unità shuffle che opera unpack, move e alcune micro-operazioni logiche. E' di supporto anche per le operazioni su interi come PINSRW. L'operazione di shuffle a 128bit è eseguita mediante tre micro istruzioni.
3. un'unità ISSE Reciprocal che esegue le istruzioni RCP e RSQRT mediante lettura di tabelle hardware.

I dati a 128bit sono scomposti in due micro-operazioni a 64bit, e se una delle due provoca un errore il contenuto del registro a 128bit potrebbe contenere dati errati. Per ovviare è stato introdotto un meccanismo hardware chiamato CNU (Check Next Microinstruction) che non permette il ritiro della prima microistruzione se la seconda microistruzione causa un'eccezione, ma non scendiamo oltre nei particolari implementativi dell'hardware, anche perché adesso sono drasticamente cambiati rispetto ai tempi del P3, vediamo piuttosto le istruzioni, suddividendole per tipologia in base alle operazioni che svolgono:

# Istruzioni SIMD per calcoli floating

## Istruzioni Matematiche di Base

Svolgono operazioni di base sui dati, possono essere sia scalari che packed e sono:

### Operazioni Packed

Assembler	Intrinsic
ADDPS	<code>__m128 _mm_add_ps( __m128 _A, __m128 _B)</code>
SUBPS	<code>__m128 _mm_sub_ps( __m128 _A, __m128 _B)</code>
MULPS	<code>__m128 _mm_mul_ps( __m128 _A, __m128 _B)</code>
DIVPS	<code>__m128 _mm_div_ps( __m128 _A, __m128 _B)</code>

### Operazioni Scalar

Assembler	Intrinsic
ADDSS	<code>__m128 _mm_add_ss( __m128 _A, __m128 _B)</code>
SUBSS	<code>__m128 _mm_sub_ss( __m128 _A, __m128 _B)</code>
MULSS	<code>__m128 _mm_mul_ss( __m128 _A, __m128 _B)</code>
DIVSS	<code>__m128 _mm_div_ss( __m128 _A, __m128 _B)</code>

penso che qui ci sia poco da spiegare, sono le classiche + - \* / in formato SSE, si caricano i due registri con i dati voluti e si chiama l'operazione da svolgere. Il primo registro è sovrascritto dal risultato dell'operazione. Stessa cosa per le istruzioni intrinsic, solo che il risultato dell'operazione è ritornato come valore `__m128` dalla funzione stessa.

## Istruzione radice quadrata

Anche questa presente sia packed che scalare, la forma è:

### Operazione Packed

Assembler	Intrinsic
SQRTPS	<code>__m128 _mm_sqrt_ps( __m128 _A)</code>

### Operazione Scalar

Assembler	Intrinsic
SQRTSS	<code>__m128 _mm_sqrt_ss( __m128 _A)</code>

Il registro in cui sono caricati i dati è sovrascritto dal risultato dell'operazione. Per quanto riguarda l'operazione...è una radice quadrata. Stessa cosa dicasi per la versione intrinsic.

## Istruzioni di approssimazione rapida

Anche questa presente sia packed che scalare, possono operare sia sui registri che sulla memoria, il risultato è scritto nel registro XMM di destinazione, la forma è:

### Operazione Packed

Assembler	Intrinsic
-----------	-----------

RCPSS	<code>__m128 __mm_rcp_ps(__m128 _A)</code>	Reciproco Packed 1/x
RSQRTPS	<code>__m128 __mm_rsqrt_ps(__m128 _A)</code>	Reciproco Packed della radice (1/sqrt(x))

## Operazione Scalar

Assembler	Intrinsic
-----------	-----------

RCPSS	<code>__m128 __mm_rcp_ss(__m128 _A)</code>	Reciproco Scalar 1/x
RSQRTSPS	<code>__m128 __mm_rsqrt_ps(__m128 _A)</code>	Reciproco Scalar della radice (1/sqrt(x))

Queste operazioni usano una look-up table presente sul chip stesso per calcolare approssimativamente, ma molto velocemente, il reciproco di un numero float e il reciproco della radice quadrata di un numero float. La precisione scende a 11bit, ma le performance in termini di velocità sono incrementate di molto e nel caso si voglia ottenere una maggiore precisione si può applicare l'algoritmo Newton-Raphson (N-R) che permette di arrivare a circa 22bit con poco overhead di calcolo. Queste istruzioni sono molto usate nella parte della pipeline 3D che si occupa di luci e superfici riflettenti, in quanto il reciproco (1/x) è molto usato proprio in queste routine. Il reciproco della radice è invece usato nelle operazioni di normalizzazione dei vettori. Per spiegare il Newton-Raphson ci vorrebbe un miniarticoletto, è comunque matematica molto accessibile, puntate il browser su wikipedia e levatevi il dubbio.

## Istruzioni di comparazione Min/Max

Presenti sia packed che scalar, possono operare sia sui registri che sulla memoria, l'operando di destinazione deve essere per forza uno dei registri XMM

### Operazione Packed

Assembler	Intrinsic
-----------	-----------

MAXPS	<code>__m128 __mm_max_ps(__m128 _A, __m128 _B)</code>
MINPS	<code>__m128 __mm_min_ps(__m128 _A, __m128 _B)</code>

### Operazione Scalar

Assembler	Intrinsic
-----------	-----------

MAXSS	<code>__m128 __mm_max_ss(__m128 _A, __m128 _B)</code>
MINSS	<code>__m128 __mm_min_ps(__m128 _A, __m128 _B)</code>

L'operando destinazione contiene, a seconda dell'istruzione, il minimo oppure il massimo dei due valori da controllare.

## Istruzione di Shuffle

Presente solo nella forma Packed, agisce sull'intera lunghezza del registro XMM e permette di scambiare di posto i quattro float contenuti nel registro appoggiandosi ad una maschera ad 8bit che specifica dove prendere e dove mettere i numeri, vediamo la cosa più in dettaglio. Chiamiamo "elemento" uno dei quattro valori float contenuti nel registro XMM, la maschera ad 8 bit specifica nel nibble alto quali elementi (due) del primo registro spostare nelle prime due posizioni (bit 0-64) del secondo registro e nel nibble basso quali elementi (altri due) del secondo registro spostare nella parte alta (bit 64-128). A questo punto "shuffliamo" i registri con un esempio, che forse vale più di tante parole:

```
SHUFPS XMM1, XMM2, 0x9C
```

la maschera di bit dell'immediato 0x9C è in binario

Second Src		First Src
1 0 0 1 1 1 0 0		

e la tabella per la selezione dell'elemento è

- 00 primo elemento
- 01 secondo elemento
- 10 terzo elemento
- 11 quarto elemento

facciamo conto che i registri siano caricati così:

```
128.....1      (XMM1)
F3   F2   F1   F0   (4 numeri float)
```

```
128.....1      (XMM2)
G3   G2   G1   G0   (4 numeri float)
```

il nibble basso della maschera è 1100 che vuol dire 00->seleziona F0 e 11->seleziona F3 quindi sposta nelle prime due posizioni del registro XMM2 l'elemento F0 e l'elemento F3, il nibble alto della maschera è 1001 che vuol dire 01->seleziona G1 e 10->seleziona G2 Il risultato sarà caricato nel registro destinazione (XMM2) e sarà quindi:

```
128.....1      (XMM2)
G2   G1   F3   F0   (4 numeri float)
```

va un po' meglio adesso? Si può anche "shufflare" un registro su se stesso, ed effettuare operazioni di "broadcast" (cioè riempire un registro con uno dei quattro possibili float), ruotare un registro oppure swappare parte alta e parte bassa del registro XMM, tutto in base al tipo di immediato che scegliamo. I Valori per il broadcast sono 0x00 per il primo elemento, 0x55 per il secondo 0xAA per il terzo e 0xFF per il quarto (a voi giocare con la rappresentazione binaria e capire cosa succede). La rotazione del registro ha come immediato 0x39 e infine lo swap 0x1B. Anche la controparte intrinsic funziona ovviamente alla stessa maniera, eccola qui:

```
_mm128_mm_shuffle_ps(_mm128_A, _mm128_B, unsigned int _Imm8);
```

il terzo parametro, la maschera di 32bit di cui abbiamo parlato prima, può essere gestito attraverso una macro messa a disposizione dall'include, che si chiama `_MM_SHUFFLE`, eccola qui di seguito:

```
#define _MM_SHUFFLE(fp3,fp2,fp1,fp0) (((fp3) << 6) | ((fp2) << 4) | ((fp1) << 2) | ((fp0)))
```

Valgono ovviamente le stesse regole dell'istruzione assembler.

## Istruzione di Unpack

Questa istruzione può essere solo Packed e può agire su un registro oppure sulla memoria. L'operando di destinazione deve essere per forza un registro XMM. L'istruzione esiste in due versioni:

```
Assembler          Intrinsic
UNPCKLPS           _mm128_mm_unpacklo_ps(_mm128_A, _mm128_B);
UNPCKHPS           _mm128_mm_unpackhi_ps(_mm128_A, _mm128_B);
```

La prima agisce sulla parte bassa del registro (i primi 64bit) la seconda su quella alta (i secondi 64), anche qui un esempietto pratico chiarirà un po' cosa combina questa istruzione, prendiamo i due soliti registri:

```
128.....1      (XMM1)
F3   F2   F1   F0   (4 numeri float)
```

```
128.....1      (XMM2)
G3   G2   G1   G0   (4 numeri float)
```

e facciamogli un bel

### UNPCKLPS XMM1, XMM2

il risultato finale, sovrascritto nel registro destinazione sarà:

```
128.....1      (XMM2)
G1   F1   G0   F0   (4 numeri float)
```

se invece avessimo fatto

### UNPCKHPS XMM1, XMM2

il risultato finale sarebbe stato:

```
128.....1 (XMM2)
G3 F3 G2 F2 (4 numeri float)
```

Come potete vedere oltre a spostare i float li interpone tra loro. Versione Intrinsic, niente da dire.

## Istruzioni di Movimento Dati (Data Movement)

Queste simpaticissime istruzioni ci permetteranno di spostare quattro float alla volta da un registro alla memoria, dalla memoria ad un registro oppure tra un registro ed un'altra. La cosa è notevolmente "scheggiante" specialmente se si utilizzano indirizzi di memoria "aligned" a 16bit (direttiva VisualC++ `__declspec(align(16))`), si possono usare anche indirizzi di memoria non aligned, ma l'esecuzione risulta rallentata. Per scopiazzare aree di memoria in giro un bel memmove con queste funzioni è molto veloce, anche se c'è la rottura di scatole di dover tenere allineati ad indirizzi a 16bit i dati da spostare. Le istruzioni sono tutte packed tranne una e sono in questa forma:

```
Assembler      Intrinsic
MOVAPS         __m128 __mm_load_ps(float * p )
               void __mm_store_ps(float *p, __m128 a)
MOVUPS        __m128 __mm_loadu_ps(float * p )
               void __mm_storeu_ps(float *p, __m128 a)
MOVHPS        __m128 __mm_loadh_pi(__m128, __m64 const*)
               __m128 __mm_storeh_pi(__m128, __m64 const*)
MOVLPS        __m128 __mm_loadl_pi(__m128, __m64 const*)
               __m128 __mm_storel_pi(__m128, __m64 const*)
MOVSS         __m128 __mm_move_ss(__m128 _A, __m128 _B)
```

In dettaglio avremo:

**MOVAPS XMM1, [EAX]** sposta 128bit (quattro float) dalla locaz. (aligned) puntata da EAX nel registro XMM1  
**MOVAPS [EDI], XMM1** sposta 128bit (quattro float) dal registro XMM1 alla locaz. (aligned) puntata dal registro EDI  
**MOVUPS XMM1, [EAX]** sposta 128bit (quattro float) dalla locaz. (non aligned) puntata da EAX nel registro XMM1  
**MOVUPS [EDI], XMM1** sposta 128bit (quattro float) dal registro XMM1 alla locaz. (non aligned) puntata dal reg. EDI  
**MOVHPS XMM1, [EAX]** sposta 64bit (due float) dalla locaz. puntata da EAX nei 64bit alti del registro XMM1  
**MOVHPS [EDI], XMM1** sposta 64bit (due float) dai 64bit alti del registro XMM1 alla locaz. puntata dal registro EDI  
**MOVLPS XMM1, [EAX]** sposta 64bit (due float) dalla locazione puntata da EAX nei 64bit bassi del registro XMM1  
**MOVLPS [EDI], XMM1** sposta 64bit (due float) dai 64bit bassi del registro XMM1 alla locaz. puntata dal registro EDI  
**MOVSS XMM1, [EAX]** sposta 32bit (un float) dalla locaz. puntata da EAX nei 32bit bassi del registro XMM1  
**MOVSS [EDI], XMM1** sposta 32bit (un float) dai 32bit bassi del registro XMM1 alla locaz. puntata dal registro EDI

Poi c'è un'altra istruzione usata per copiare la parte bassa (primi 64bit) o la parte alta (secondi 64bit) di un registro sorgente in un registro destinazione, la forma è questa:

```
Assembler      Intrinsic
MOVHLPS        __m128 __mm_movehl_ps(__m128, __m128)      copia dalla parte alta alla parte bassa
MOVLHPS        __m128 __mm_movelh_ps(__m128, __m128)      copia dalla parte bassa alla parte alta
```

Giusto per chiarire:

```
128.....1 (XMM1)
F3 F2 F1 F0 (4 numeri float)
```

```
128.....1 (XMM2)
G3 G2 G1 G0 (4 numeri float)
```

facciamo **MOVHLPS XMM1,XMM2** e avremo:

```
128.....1 (XMM1)
F3 F2 G3 G2 (4 numeri float)
```

facciamo **MOVLHPS XMM1,XMM2** e avremo:

```
128.....1 (XMM1)
G3 G2 F1 F0 (4 numeri float)
```

## Istruzione di Movimento BitMask

L'istruzione permette di avere nei primi 4bit di un registro integer i primi bit di ciascuno dei float contenuti nel registro XMM scelto come operando, l'istruzione ha questa forma:

```
Assembler Intrinsic
MOVMSKPS EAX,XMM1 int _mm_movemask_ps( __m128 _A);
```

se il registro XMM1 contiene ad esempio

```
127.....95.....63.....31.....0 <- posizione
100.....011.....110.....011..... <- valore in bit del contenuto
```

allora dopo l'esecuzione dell'istruzione avremo che EAX sarà

```
31.....3 2 1 0 <-posizione
0000.....1 0 1 0 <-valore della maschera
```

i primi 28 bit del registro EAX saranno azzerati dall'istruzione. Questa istruzione è stata pensata per l'utilizzo in abbinamento con istruzione logiche (AND, ANDN, OR, XOR) per permettere spostamenti condizionali di dati, come ad esempio può succedere nelle routines di clipping e di back culling delle facce, ovvero l'eliminazione delle facce nascoste per non farle entrare nella pipeline di rendering del motore 3D. La versione Intrinsic accetta come parametro un registro xmm e restituisce la maschera sotto forma di intero.

## Istruzioni di Confronto e Set BitMask

Queste istruzioni, sia in formato packed che scalar, fanno un confronto del tipo "minore di" sugli operandi e settano ad 1 oppure a 0 tutto il contenuto del registro sorgente per indicare vero (stato logico 1) oppure falso (stato logico 0). La forma dell'istruzione è:

```
Assembler Intrinsic
CMPLTSS XMM1,XMM2 __m128 _mm_cmplt_ps( __m128 _A, __m128 _B);
CMPLTSS XMM1,XMM2 __m128 _mm_cmplt_ss( __m128 _A, __m128 _B);
```

solito esempio chiarificatore:

```
127.....0 (XMM1)
10.0f 15.0f 12.0f 7.0f (4 numeri float)

127.....0 (XMM2)
9.0f 12.0f 17.0f 8.0f (4 numeri float)
```

facciamo **CMPLTSS XMM1,XMM2** e avremo:

```
127.....0 (XMM1)
FFFF...0000...0000...1111 (i 4 numeri float sono tutti a 0 oppure tutti a 1)
```

facciamo **CMPLTSS XMM1,XMM2** e avremo:

```
127.....0      (XMM1)
10.0f 15.0f 12.0f 1111 (i primi 3 numeri float sono invariati)
```

La sorgente può essere un operando in memoria oppure un registro XMM, la destinazione può essere solo un registro XMM. Per la versione intrinsic, idem come sopra.

## Istruzioni Logiche

Potevano mancare i classici And, Or et similia? Certo che no, SSE mette a disposizione le istruzioni SIMD anche per i confronti logici bit a bit. Le istruzioni sono tutte packed ed hanno questa forma:

Assembler	Intrinsic
ANDPS	<code>__m128 _mm_and_ps(__m128 _A, __m128 _B);</code>
ANDNPS	<code>__m128 _mm_andnot_ps(__m128 _A, __m128 _B);</code>
ORPS	<code>__m128 _mm_or_ps(__m128 _A, __m128 _B);</code>
XORPS	<code>__m128 _mm_xor_ps(__m128 _A, __m128 _B);</code>

Possono operare tra memoria e registri XMM oppure tra registri stessi, cosa fanno?.....la spiegazione non ve la faccio perdavvero! ;-)) sono i classici AND OR e XOR che lavorano sull'intera estensione dei registri XMM, unica istruzione "diversa" è ANDNPS che prima effettua una negazione logica dell'operando sorgente e poi esegue l'AND con l'operando destinazione, per il resto andatevi a rivedere l'algebra booleana e le tabelle di verità. Per la versione intrinsic invece andatevi a rivedere le tabelle di verità e l'algebra booleana :)

## Istruzioni di Confronto e Set EFLAGS

Queste istruzioni, solo in formato scalar, fanno un confronto tra i due numeri float (32bit) del registro sorgente e destinazione, settando di conseguenza il flag di stato del processore in concomitanza col risultato del confronto. I bit di EFLAGS che sono coinvolti sono ZF, PF e CF (Zero, Parità e Carry) mentre i bit OF, SF e AF (Overflow, Segno e Ausiliario) sono settati sempre a zero. L'istruzione è nella forma

Assembler	Intrinsic
COMISS reg1,reg2	<code>int _mm_comxxx_ss(__m128 _A, __m128 _B)</code>
UCOMISS reg1,reg2	<code>int _mm_ucomxxx_ss(__m128 _A, __m128 _B)</code>

L'unica differenza tra le due istruzioni è che COMISS genera un'eccezione di tipo numerica nel caso in cui l'operando sorgente sia un numero del tipo QNAN o SNAN, mentre UCOMISS genera l'eccezione solo nel caso in cui l'operando sorgente sia un numero SNAN. QNAN e SNAN sono numeri definiti nello standard IEEE e indicano rispettivamente 'Quite Not A Number' (bit più significativo del numero frazionario settato a 1) e 'Signalling NAN' (bit più significativo del numero frazionario settato a 0 e almeno un altro bit della frazione settato a 1 se tutti i bit sono 0 allora il numero indica infinito). C'è un interessante articolo in giro che riporta le differenze tra fpu amd e intel proprio riguardo questi casi particolari, sembra che la fpu intel perda parecchio tempo a definirli. I bit del registro di stato EFLAGS vanno interpretati come segue:

REGISTRO EFLAGS	ZF	PF	CF
Casi			
Unordered	1	1	1
<	0	0	1
>	0	0	0
=	1	0	1

Vedi le istruzioni di confronto riportate più avanti per conoscere tutte le varianti possibili.

## Istruzioni di Conversione

Le istruzioni in questione permettono di convertire tra numeri packed o scalar a 32bit con segno a numeri packed o scalar floating e viceversa. La sintassi come sempre si appoggia sui suffissi per chiarire meglio l'operazione che quell'istruzione svolge.

- **Conversione FP -> Integer con arrotondamento (l'arrotondamento è settato tramite registro MXCSR, spiegato più avanti nelle istruzioni di controllo)**

Assembler

```
CVTTPS2PI
converte due float (XMM) in due packed integer (MMX), il risultato è messo nel
registro MMX
CVTSS2SI
converte un float (XMM) in uno scalar integer (MMX), il risultato è messo nel
registro MMX
```

Intrinsic

```
_m64 _mm_cvt_ps2pi(__m128 _A)
int _mm_cvtt_ss2si(__m128 _A)
```

- **Conversione FP -> Integer con troncamento (il registro MXCSR non influisce sull'esito dei risultati, il valore è semplicemente troncato)**

Assembler

```
CVTTTPS2PI
converte due float (XMM) in due packed integer (MMX), il risultato è messo nel
registro MMX
CVTTSS2SI
converte un float (XMM) in uno scalar integer (MMX), il risultato è messo nel
registro MMX o in un registro generico a 32bit (esempio EAX)
```

Intrinsic

```
_m64 _mm_cvtt_ps2pi(__m128 _A)
int _mm_cvtt_ss2si(__m128 _A)
```

- **Conversione Integer -> FP con arrotondamento (l'arrotondamento è settato tramite registro MXCSR, spiegato più avanti nelle istruzioni di controllo)**

Assembler

```
CVTPI2PS
converte due packed integer (MMX) in due floating point (XMM), il risultato è
messo nel registro XMM
CVTSI2SS
converte uno scalar integer (MMX) in uno scalar floating point (XMM), il
risultato è messo nel registro XMM
```

Intrinsic

```
_m128 _mm_cvt_pi2ps(__m128, __m64)
_m128 _mm_cvt_si2ss(__m128, int)
```

Questo tipo di operazioni è usato ad esempio nella parte del motore 3D che si occupa dell'illuminazione, in quanto i coprocessori delle schede grafiche possono(potevano) leggere solo valori interi e non floating (adesso non più, viva le immagini HDR!). Tramite queste istruzioni di conversione i numeri float sono convertiti in packed MMX. Le istruzioni possono operare tra registri XMM, MMX (registri a 64bit) e "normali" (registri a 32bit). Non possono operare sulla memoria.

## Istruzioni SIMD per calcoli integer

Con l'introduzione delle istruzioni SIMD è stato ampliato anche il set di istruzioni del tipo MMX, per permettere la manipolazione contemporanea di più interi in una singola istruzione. Come nel suo predecessore, la tecnologia SIMD per gli interi utilizza i registri MMX 'ghostati' sui registri del coprocessore matematico e con l'utilizzo di prefissi e suffissi indica il tipo di dato che è manipolato. Ricordo che i registri MMX hanno estensione 64bit, quindi i tipi di dato possono essere:

Packed byte (8x8bit per registro)

Packed word (4x16bit per registro)

Packed doubleword (2x32bit per registro)

Quadword (1x64bit per registro)

e i prefissi e suffissi indicano:

**p** (prefisso) operazioni su dati di tipo packed

**b** (suffisso) byte

**w** (suffisso) word

**d** (suffisso) doubleword

**q** (suffisso) quadword

**u** (suffisso) unsigned

**s** (suffisso) signed

Ad esempio l'istruzione **pmulwu** indica una **m**ultiplication su dati **p**acked di tipo **w**ord **u**nsigned. E adesso beccatevi la carrellata delle nuove istruzioni introdotte:

### Istruzione di Estrazione

L'istruzione copia un valore packed 16bit (pword) da un registro mmx alla parte bassa di un registro "normale" (eax ad esempio) azzerandone i 16bit alti. Un immediato indica all'istruzione quale dei 4 pword prendere dal registro mmx. L'istruzione opera solo tra registri, non in memoria. Forma dell'istruzione:

Assembler	Intrinsic
PEXTRW	<code>int _m_pextrw(__m64, int)</code>

I due bit più bassi dell'operando indicano quale pword prendere, come nell'istruzione SHUFFLE indicheremo:

00 primo elemento

01 secondo elemento

10 terzo elemento

11 quarto elemento

Esempio:

```

63.....0      (MM1)
I3   I2   I1   I0   (4 numeri integer pword)

immediato: 00000011 (quarto elemento)

PEXTRW MM1,EAX,0x04

31....15....0  (EAX)
000..0   I3     (Contenuto di EAX dopo l'operazione)

```

## Istruzione di Inserimento

L'istruzione inserisce un valore packed 16bit (pword) in un registro mmx alla posizione indicata da un immediato, prelevando il dato da un registro oppure da una locazione di memoria. Praticamente è l'istruzione inversa di quella precedente, quindi funziona allo stesso modo ma al contrario. Esempio:

Assembler	Intrinsic
PINSRW	<code>__m64 _m_pinsrw( __m64, int, int);</code>

ecco un esempio dell'istruzione,

### **PINSRW MM1,EAX,0x01**

```

31.....0      (EAX)
K1       K0

63.....0      (MM1)
I3   I2   I1   I0   (4 numeri integer pword)

```

dopo l'istruzione il registro MM1 conterrà:

```

63.....0      (MM1)
I3   I2   K0   I0   (4 numeri integer pword)

```

stesso discorso se il dato lo si prende in memoria, la sintassi sarà qualcosa del tipo:

### **PINSRW MM1,[EAX],0x01**

gli altri valori packed contenuti in MM1 restano invariati.

## Istruzioni di Confronto MIN/MAX su interi

Tramite queste istruzioni si possono operare dei confronti sia tra valori packed word con segno che tra valori packed byte senza segno. Si possono operare quindi due oppure otto confronti contemporaneamente. Il risultato del confronto sarà scritto nell'operando di destinazione che conterrà il minimo o il massimo dei valori da confrontare in base all'istruzione impartita. Possono operare sia tra registri MMX che tra registri e memoria. Le istruzioni sono:

Assembler		Intrinsic
PMINSW	- Packed Signed Word	<code>__m64 _m_pminsw( __m64, __m64)</code>
PMAWSW	- Packed Signed Word	<code>__m64 _m_pmaxsw( __m64, __m64)</code>
PMINUB	- Packed Unsigned Byte	<code>__m64 _m_pminub( __m64, __m64)</code>
PMAXUB	- Packed Unsigned Byte	<code>__m64 _m_pmaxub( __m64, __m64)</code>

Esempio:

## PMINSW MM1,MM2

```
63.....0      (MM1)
0   4   5   3   (4 numeri integer pword)

63.....0      (MM2)
3   3   1   2   (4 numeri integer pword)
```

dopo l'esecuzione dell'istruzione il contenuto di MM1 sarà

```
63.....0      (MM1)
0   3   1   2   (4 numeri integer pword)
```

Queste istruzioni sono molto usate negli algoritmi di riconoscimento vocale, all'interno di un algoritmo di riconoscimento del modello "Hidden-Markov" (cosa sia non lo so...google aiutaci te!!) oppure nella parte dei motori 3D che si occupano della rasterizzazione e della quantizzazione dei colori.

## Istruzione di Movimento BitMask intero

Come la sua sorellona MOVMSKPS, questa istruzione preleva i bit più significativi dei packed byte del registro indicato e crea una maschera nei primi 8bit del registro destinazione. Il resto dei bit del registro destinazione sono posti a zero. Istruzione:

```
Assembler          Intrinsic
PMOVMSKB EAX,MM1   int _m_pmovmskb( __m64)
```

se il registro MM1 contiene ad esempio

```
63...56...47...39...31...14...7...0   <- posizione
11101010110101010100010110101001010  <- valore in bit del contenuto
```

allora dopo l'esecuzione dell'istruzione avremo che EAX sarà

```
31..... 7 6 5 4 3 2 1 0   <-posizione
0000.....1 0 0 1 0 1 0 0   <-valore della maschera
```

i primi 24 bit del registro EAX saranno azzerati dall'istruzione.L'utilizzo tipico è stato descritto nella funzione SIMD-FP MOVMSKPS.

## Istruzione di Moltiplicazione Unsigned con troncamento

Questa istruzione effettua la moltiplicazione tra operando sorgente e operando destinazione, il sorgente può essere un registro MMX oppure la memoria, la destinazione è sempre un registro MMX. Sintassi:

```
Assembler          Intrinsic
PMULUW MM1,MM2     __m64 _m_pmulhw( __m64, __m64)
```

sono presi i 16bit alti del risultato intermedio (a 32bit) di ogni moltiplicazione svolta tra i 4 pword dei registri MMX.

## Istruzione di Shuffle su registri MMX

Come la sorellona SHUFPS, questa istruzione "mischia le carte" all'interno dei registri MMX, in base alle direttive imposte da un immediato. Non sto spiegarvela perché è identica all'altra, soltanto che opera su 4 packed word intere contenute all'interno del registro MMX a 64bit. La sintassi ovviamente cambia, ma quello che ci potete fare è in sostanza lo stesso (rotazioni, broadcasting del dato, swap del registro). Sintassi:

Assembler	Intrinsic
PSHUFW MM1,MM2,0x9C	<code>__m64 mm_shuffle_pi16(__m64 a, int n)</code>

Il sorgente può essere un registro MMX oppure la memoria, la destinazione è sempre un registro MMX.

## Istruzioni di Media

Operano sui registri MMX ed eseguono la media di due valori packed, possono essere byte (accuratezza della media 8bit) o word (accuratezza 16bit). E' preferibile usare la versione a 8bit per incrementare il parallelismo dell'esecuzione dell'istruzione.

Assembler	Intrinsic
PAVGB MM0,MM1	<code>__m64 mm_avg_pu8(__m64 a, __m64 b)</code>
PAVGW MM0,MM1	<code>__m64 mm_avg_pi16(__m64 a, __m64 b)</code>

Questa istruzione è molto usata nella parte di compensazione del moto dell'algoritmo MPEG-2, il processo che ricrea i frame intermedi interpolando i frame "chiave" della scena. Può operare tra due registri MMX oppure tra un registro e la memoria, l'operando destinazione è sempre un registro.

## Istruzioni di Somma delle differenze

Operano sui registri MMX ed eseguono la somma del quadrato delle differenze oppure la somma delle differenze assolute

Assembler	Intrinsic	
PSSDBW MM0,MM1		Somma del quadrato delle diff.
PSADBW MM0,MM1	<code>__m64 mm_sad_pu8(__m64 a, __m64 b)</code>	Somma delle diff. assolute

queste istruzioni sono dedicate all' algoritmo di motion-estimation tipico delle pipeline grafiche MPEG-2, ad esempio nel caso di PSADBW si possono risparmiare ben 7 istruzioni MMX nel ciclo interno dell'algoritmo di motion-estimation, riuscendo a migliorarne l'esecuzione di un fattore di circa 2volte.

## Istruzioni SIMD per Controllo della Cache

Si, per velocizzare ulteriormente l'esecuzione dei calcoli è possibile intervenire direttamente sul meccanismo di caching e prefetching del processore, forzando alcuni stati o precaricando i dati che sappiamo che utilizzeremo a breve. Pastrocchiare la cache è delicato come tutti sappiamo, e richiede molta attenzione perchè rischiamo di rallentare le cose invece di velocizzarle. E' bene introdurre due o tre termini che ci serviranno per identificare i tipi di dati e la loro relazione con la cache:

- **Non-Temporali**

sono dati a cui si accede irregolarmente e dopo lunghi intervalli di tempo, ad esempio come una serie di coordinate 3D di un oggetto, che vengono lette e usate solo una volta per ogni frame che viene generato.

- **Temporali**

sono dati a cui si accede ripetutamente in un breve periodo di tempo, tipicamente si incontrano nei loop dei programmi.

- **Spaziali**

sono dati che hanno un a posizione in memoria molto vicina a quella dell'ultimo dato letto, ad esempio i membri di una struttura.

Le istruzioni qui di seguito permettono di dare dei suggerimenti (Hints) per la gestione temporale dei dati in cache di primo e secondo livello, e in quanto tali non generano eccezioni. Se cerchiamo di prefetchare un indirizzo di memoria non valido non sarà segnalato quindi nessun errore. Queste istruzioni permettono ad esempio di immagazzinare dati solo nella cache di primo livello (L1) evitando di rimuoverlo anche dalla cache di secondo livello (L2) se si presume che quel dato servirà a breve (PREFETCHNTA), oppure di immagazzinare una linea di cache contemporaneamente nella L1 e nella L2 (PREFETCH0). Con le istruzioni di prefetch si può quindi caricare un

dato prima del suo utilizzo, eliminando del tutto o in parte lo stallo del processore dovuto al tempo da attendere perché il dato sia caricato in cache o nei registri.

## Istruzioni di Streaming Store

Depositano in memoria un registro XMM (128bit) oppure MMX (64bit) bypassando la cache. Se il dato da depositare è già presente nella cache di scrittura è assicurata la coerenza del programma. Se l'elemento non è allineato a 16bit è sollevata un'eccezione.

Assembler	Intrinsic
MOVNTPS [EAX],XMM1	void _mm_stream_ps(float *p, __m128 a)

sposta 128bit nella locazione indirizzata da EAX senza usare la cache.

Assembler	Intrinsic
MOVNTQ [EAX],MM1	void _mm_stream_pi(__m64 *p, __m64 a)

sposta 128bit nella locazione indirizzata da EAX senza usare la cache.

## Istruzione di Movimento MaskedBytes

Questa istruzione permette di scrivere dati direttamente in memoria prelevandoli da un registro MMX. Un registro funziona da maschera (0x01 "fa passare" in memoria 0x00 no) mentre il registro EDI indica dove immagazzinare il dato. Esempio:

### **MASKMOVQ MM1,MM2**

63.....0	Registro MM1
A7 A6 A5 A4 A3 A2 A1 A0	
63.....0	Registro MM2 (Maschera)
0 0 0 1 0 1 0 0	

Dopo l'esecuzione dell'istruzione alla locazione [EDI] avremo:

0 0 0 A4 0 A2 0 0	(Contenuto della memoria indirizzata da EDI)
-------------------	--

Questa istruzione adotta una politica di write-combining, cioè sono accumulate più istruzioni di write in un apposito buffer del processore e il loro seguente accesso alla memoria in un'unica soluzione, riducendo così il traffico sul bus. Nel PentiumIII questo buffer è di 32bytes e ce ne sono ben 4 a disposizione. Questa tecnica però, unita all'esecuzione "out-of-order" delle istruzioni produce un accesso alla memoria debolmente ordinato, cioè non eseguiti per ordine di programma. Tale istruzione è molto usata nei processi di rasterizzazione delle immagini e nei programmi di image processing.

## Istruzione Store Fence

Con questa istruzione si ovvia al problema accennato prima, assicurando che ogni istruzione di store presente nel write buffer e nella cache L1 siano eseguiti prima di effettuare altre istruzioni, in modo da renderli globalmente visibili. Sintassi dell'istruzione:

Assembler	Intrinsic
SFENCE	void _mm_sfence(void)

Non necessita di parametri.

## Istruzione di Prefetch

Le varianti di questa istruzione permettono di caricare i dati nella cache di primo o di secondo livello, o in ambedue.

```
Assembler                Intrinsic
PREFETCH0 [ESI]         void _mm_prefetch(char const*a, int sel)
PREFETCH1 [ESI]
PREFETCH2 [ESI]
PREFETCHNTA [ESI]
```

PREFETCH0,1,2 si occupano di caricare i dati dall'indirizzo di memoria puntato dal registro trattandoli come Temporali (vedi sopra), mentre la PREFETCHNTA fa la stessa cosa ma trattando i dati come Non-Temporali. PREFETCH0 carica i dati nella cache L1 e nella cache L2, PREFETCH1 e 2 solo nella cache di secondo livello L2 e PREFETCHNTA è caricato direttamente nella cache di primo livello L1, bypassando il transito nella L2. Nella versione intrinsic viene passato come parametro un intero che seleziona queste varie modalità con delle costanti predefinite che si chiamano `_MM_HINT_T0`, `_MM_HINT_T1`, `_MM_HINT_T2` e `_MM_HINT_NTA`.

## Istruzioni SIMD per la Gestione dello Stato

Per gestire i nuovi registri XMM, il PentiumIII aggiunge all'esistente set anche il registro di controllo e di stato MXCSR e un vettore di eccezione, che ci permette di gestire il tipo di arrotondamento, il flush-to-zero mode, le eccezioni numeriche e il loro masking/unmasking. MXCSR è un registro a 32bit così suddivisi:

```
31.....16..15..10.....5.....0
Riservati  FZ RC mask flags  eccezioni
Bit 16-31 reserved
```

Bit 15 Flush-to-zero mode

Bit 14-13 Controllo Tipo Arrotondamento

Bit 12-7 Maschera delle eccezioni numeriche

Bit 6 reserved

Bit 5-0 Indica quale eccezioni si è verificata

Quando un'operazione genera uno stato di underflow, se il bit flush-to-zero mode è settato allora il processore ritorna zero, con il segno del vero risultato, setta il precision flag del registro di stato (risultato inaccurato) e i flags di exception underflow nei bit della maschera di MXCSR. Meno lo facciamo cadere nello stato di underflow e meglio è, perchè più eccezioni generiamo e più rallentiamo tutto quanto, però è anche vero che se lavoriamo con numeri in doppia precisione (double), valutando ad esempio differenze infinitesimali (ad esempio due punti adiacenti dell'insieme di Mandelbrot) ci dobbiamo cadere per forza. Ehhabbè, pazienza :-). I bit 13 e 14 (RC) determinano il tipo di arrotondamento in base alla seguente tabella:

R	C	
Arrotonda al più vicino	0 0	(arrotondamento di default)
Arrotonda in basso	0 1	
Arrotonda in alto	1 0	
Arrotonda con troncamento	1 1	

Le istruzioni SIMD possono generare le seguenti eccezioni:

- Operazione invalida (#I)
- Operando denormalizzato (#D)
- Divisione per zero (#Z)
- Overflow numerico (#O)
- Underflow numerico (#U)
- Risultato inesatto (#P)

ed hanno tutte il loro flag di stato (bit0-5) e la loro maschera (bit7-12) nel registro MXCSR

Discorso a parte merita l'underflow e il conseguente flush denormals to zero, ovvero "quando" ci si accorge che il dato che stiamo trattando è talmente piccolo da non poter più essere rappresentato dai bit a nostra disposizione, e quindi dobbiamo assimilarlo come zero (anche se in effetti non lo è). Come già accennato in precedenza ci sono differenze sia in termini di velocità che in termini di precisione tra cpu amd e intel. Questa cosa sarà oggetto di un altro articolo (spero).

## Istruzioni Load/Store

Le istruzioni Load e Store rispettivamente caricano dalla memoria e immagazzinano in memoria il registro MXCSR, sono nella forma:

Assembler	Intrinsic
LDMXCSR	<code>void _mm_setcsr(unsigned int i)</code>
STMXCSR	<code>unsigned int _mm_getcsr(void)</code>

sono normalmente usate per caricare il registro MXCSR ed azzerare lo status flag. Raggruppo sotto questa tipologia di istruzioni anche queste due:

```
FXSAVE
FXRSTORE
```

che rispettivamente salvano in memoria e ricaricano dalla memoria lo stato dei registri XMM e MMX. Lo stato occupa 512byte in memoria e l'indirizzo in cui salvare lo stato deve essere allineato ai 16bit, pena eccezione.

## Scrivere codice Intrinsic col VisualC

Come già preannunciato, il VisualC (ma anche lo gnu e altri) mette a disposizione la gestione 'intrinsic' delle istruzioni SSE, questo vuole dire in parole povere che viene fornito un wrapup delle istruzioni SSE all'interno del codice C, senza bisogno di scrivere assembler inline. Questa caratteristica è disponibile se avete upgradato il vostro VisualC++6 con il Processor Pack fornito gratuitamente dalla M\$. Se avete il VisualStudio (2003,2005,2008,Express) tale opzione è già presente. Includendo nei vostri progetti l'header "xmmintrinsic.h" avrete a vostra disposizione tutto il set di istruzioni SSE più un nuovo tipo di dato a 128bit che viene già gestito con allineamento a 16byte:

```
/* using real intrinsics */
typedef long long __m128;
```

più tante macro per la gestione di ogni aspetto della macchina SSE, ad esempio con:

```
#define _MM_ALIGN16 __declspec(align(16))
```

è possibile allineare i dati ad indirizzi multipli di 16bit, oppure con:

```
#define _MM_SET_ROUNDING_MODE(mode) \
_mm_setcsr((_mm_getcsr() & ~_MM_ROUND_MASK) | (mode))

#define _MM_GET_ROUNDING_MODE() \
(_mm_getcsr() & _MM_ROUND_MASK)
```

avrete due comode macro per settare o leggere la modalità di arrotondamento. E ancora, sono state "snocciate" tutte le possibili comparazioni senza doversi andare a guardare i flag di stato per il risultato, quindi avrete le seguenti funzioni a disposizione:

```
/*
 * FP, comparison
 */
extern __m128 _mm_cmpeq_ss(__m128 a, __m128 b);
extern __m128 _mm_cmpeq_ps(__m128 a, __m128 b);
extern __m128 _mm_cmplt_ss(__m128 a, __m128 b);
extern __m128 _mm_cmplt_ps(__m128 a, __m128 b);
extern __m128 _mm_cmpge_ss(__m128 a, __m128 b);
```

```

extern __m128 __mm_cmple_ps(__m128 a, __m128 b);
extern __m128 __mm_cmpgt_ss(__m128 a, __m128 b);
extern __m128 __mm_cmpgt_ps(__m128 a, __m128 b);
extern __m128 __mm_cmpge_ss(__m128 a, __m128 b);
extern __m128 __mm_cmpge_ps(__m128 a, __m128 b);
extern __m128 __mm_cmpneq_ss(__m128 a, __m128 b);
extern __m128 __mm_cmpneq_ps(__m128 a, __m128 b);
extern __m128 __mm_cmpnlt_ss(__m128 a, __m128 b);
extern __m128 __mm_cmpnlt_ps(__m128 a, __m128 b);
extern __m128 __mm_cmpnle_ss(__m128 a, __m128 b);
extern __m128 __mm_cmpnle_ps(__m128 a, __m128 b);
extern __m128 __mm_cmpngt_ss(__m128 a, __m128 b);
extern __m128 __mm_cmpngt_ps(__m128 a, __m128 b);
extern __m128 __mm_cmpnge_ss(__m128 a, __m128 b);
extern __m128 __mm_cmpnge_ps(__m128 a, __m128 b);
extern __m128 __mm_cmpord_ss(__m128 a, __m128 b);
extern __m128 __mm_cmpord_ps(__m128 a, __m128 b);
extern __m128 __mm_cmpunord_ss(__m128 a, __m128 b);
extern __m128 __mm_cmpunord_ps(__m128 a, __m128 b);
extern int __mm_comieq_ss(__m128 a, __m128 b);
extern int __mm_comilt_ss(__m128 a, __m128 b);
extern int __mm_comile_ss(__m128 a, __m128 b);
extern int __mm_comigt_ss(__m128 a, __m128 b);
extern int __mm_comige_ss(__m128 a, __m128 b);
extern int __mm_comineq_ss(__m128 a, __m128 b);
extern int __mm_ucomieq_ss(__m128 a, __m128 b);
extern int __mm_ucomilt_ss(__m128 a, __m128 b);
extern int __mm_ucomile_ss(__m128 a, __m128 b);
extern int __mm_ucomigt_ss(__m128 a, __m128 b);
extern int __mm_ucomige_ss(__m128 a, __m128 b);
extern int __mm_ucomineq_ss(__m128 a, __m128 b);

```

Se avete bisogno di allocare un array dinamico potete (dovete!) usare le funzioni **`__mm_malloc()`** e la rispettiva **`__mm_free()`**, definite nell'include `<malloc.h>` (lo include automaticamente `xmmintrin.h`) come alias delle funzioni `_aligned_malloc()` e `_aligned_free()`, messe a disposizione dalla common run time (CRT) del Visual. Si usano pari pari come il `malloc()` e il `free()` classici, solo che `__mm_malloc()` ritorna un indirizzo allineato ad un valore che gli passiamo noi come argomento e che deve essere multiplo di due (es: 16byte), per poter usare il chunk di memoria con le nostre funzioni SSE. Esempio:

```

//richiedo 1024 float, il chunk di ritorno deve avere un indirizzo allineato a 16byte
float* alignedmem = (float*) __mm_malloc(1024, 16);
..
..
..
// libero la memoria precedentemente allocata
__mm_free(alignedmem);

```

Un'ultima cosa riguardo alle funzioni intrinsic: non tutte hanno una corrispondente istruzione assembler, mi spiego meglio, non sono sempre tradotte dal compilatore con un'unica istruzione, in alcuni casi ne servono di più, in altri sono addirittura macro, quindi se volete un controllo totale del codice prodotto, ad esempio nel caso di scrittura di demo 4k in cui lo spazio è vitale, l'unica alternativa è scrivere codice direttamente in assembler. La intel chiama queste istruzioni 'composite', vi faccio un esempio:

### **`__m128 __mm_set_ps1(float _A)`**

Questa istruzione non esiste nel set degli mnemonici SSE, ma risulta comunque comoda, non fa altro che settare i 4 float del registro con il valore passato come argomento. Il compilatore traduce questa funzione così:

```
movss xmm0, DWORD PTR [A]
```

```
shufps xmm0, xmm0, 0
```

e cioè carica nei primi 32bit del registro `xmm0` il valore del float, e poi effettua un'operazione di broadcasting del dato sull'intero registro. Dopo `shufps` il registro `xmm0` conterrà quindi 'AAAA'. Altro esempio la macro `__MM_TRANSPOSE4_PS()` che opera la trasposta (inversione di posto tra colonne e righe di una matrice) di una matrice 4x4 usando l'istruzione `shufps` in questo modo:

```

#define __MM_TRANSPOSE4_PS(row0, row1, row2, row3) {
    __m128 tmp3, tmp2, tmp1, tmp0;
    tmp0 = __mm_shuffle_ps((row0), (row1), 0x44);
    tmp2 = __mm_shuffle_ps((row0), (row1), 0xEE);
}

```

```

    tmp1  = _mm_shuffle_ps((row2), (row3), 0x44);      \
    tmp3  = _mm_shuffle_ps((row2), (row3), 0xEE);      \
                                                    \
    (row0) = _mm_shuffle_ps(tmp0, tmp1, 0x88);         \
    (row1) = _mm_shuffle_ps(tmp0, tmp1, 0xDD);         \
    (row2) = _mm_shuffle_ps(tmp2, tmp3, 0x88);         \
    (row3) = _mm_shuffle_ps(tmp2, tmp3, 0xDD);         \
}

```

Fanno parte delle istruzioni 'composite' e quindi usabili solo scrivendo codice intrinsic, le funzioni dette di 'set':

```

__m128 _mm_set_ss(float w )
// setta il valore float basso con w e clear degli altri tre

__m128 _mm_set1_ps(float w )
// setta i 4 valori float a w

__m128 _mm_set_ps(float z, float y, float x, float w )
// setta i 4 valori float con i 4 valori in input

__m128 _mm_setr_ps (float z, float y, float x, float w )
// come sopra ma in ordine inverso

__m128 _mm_setzero_ps (void)
// setta i 4 valor float a zero

```

Se la pazienza vi ha retto fino ad ora, vi consiglio di andare a leggere `xmmintrinsic.h`, con le conoscenze che avete incamerato vi risulterà banale il suo utilizzo all'interno dei vostri programmi. Oltretutto il codice intrinsic ha praticamente la stessa velocità di esecuzione dell'assembler inline (direttiva `_asm`). Ricordatevi inoltre che è finito (purtroppo? :-)) il tempo in cui i compilatori creavano del codice che andava per forza rivisitato a mano per ottenere il massimo delle prestazioni, i moderni compilatori producono del codice veramente ottimo (ma che ve lo dico a fare...siete reverser no! ;) e se la progettazione dell'algoritmo è buona raramente si deve intervenire in assembler per guadagnare ancora in velocità. Tenete poi conto che attualmente le librerie DirectX hanno già al loro interno le routines ottimizzate in SSE (o SSE2 e 3 a seconda del processore che trovano) quindi per esempio se usate DirectX per le operazioni di calcolo matriciale avrete già il top delle performance, anche se a dire il vero disassemblandole si intravede(va) spazio per ulteriori ottimizzazioni, (ai tempi delle DirectX 7 e 8, già la 9 è tutt'altra storia) una su tutte l'adozione di strutture SOA in modo da usare il più veloce `movaps` al posto di `movups` (non ricordate la differenza? rileggete il tut!).

## Organizzare i dati, AoS e SoA

Descriverò qui un altro aspetto importante per la gestione dei dati sotto SSE, definito con due acronimi, SOA e AOS.

### AOS

Sta per "Array Of Structure" ovvero i dati in memoria sono organizzati consecutivamente in questo modo:

```

Struct vertice{
    float x;
    float y;
    float z;
    float w;
}

```

in memoria avremo quindi "xyzw" per ogni vertice (la rappresentazione "classica"), molti usano definire un vertice con una cosa del tipo:

```
typedef vertice float[4];
```

tanto poi sappiamo che `vertice[0]` corrisponde alla coordinata x, `vertice[1]` alla coordinata y etc.etc. una maniera più pratica? Secondo me si, oltretutto definendo la nostra struttura in questo modo:

```

struct Vettore{
    union{
        __m128  vVect;
        float   fVect[4];
    }
}

```

```
};
}
```

ci ritroviamo a gratis due modi per accedere sia al dato xmm nella sua totalità che in ognuna delle sue componenti float, aggiungete il fatto che appena il compilatore trova `__m128` nella union alloca la memoria necessaria alla struttura stessa già 16byte aligned e quindi non dovete rompervi la testa con la macro `_MM_ALIGN16` et similia, niente crash dovuti ad accessi a indirizzi di memoria unaligned da parte delle istruzioni!

## SOA

Sta per "Structure Of Array" ovvero i singoli dati di ogni vertice sono organizzati consecutivamente in memoria:

```
Struct Oggetto{
    float x[n];
    float y[n];
    float z[n];
    float w[n];
}
```

stavolta in memoria avremo "xxxx.....yyyy.....zzzz.....www....." cioè prima tutte le x poi tutte le y etc.etc.

Per come funziona SSE è più conveniente usare una rappresentazione del tipo SOA, perché ad esempio è possibile caricare in una sola volta ben 4 coordinate X e processarle contemporaneamente. Visto l'overhead che introdurrebbe il dover riorganizzare completamente i dati di un oggetto (potrebbe trattarsi di migliaia di poligoni e vertici) è più conveniente un approccio di tipo misto, cioè riorganizzare una piccola quantità di vertici da AOS a SOA, applicare le nostre operazioni SSE e iterare fino alla fine dei dati. Oppure se siete dei perfezionisti potete sempre crearvi i vostri tool che convertiranno la disposizione di tutti i dati necessari al vostro motore in formato SOA a priori, in modo che siano già pronti all'uso. Di seguito vi pasto un esempio di codice preso dall'ottimo "Intel Pentium 4 processor optimization reference manual" che vi potete scaricare liberamente dal sito della Intel e che tratta molto bene l'argomento SIMD. Il programma esegue il prodotto scalare tra un array di vettori ed un vettore fisso, operazione comunissima nei motori3D, e mostra le due tecniche AOS e SOA:

```
; The dot product of an array of vectors (Array) and a
; fixed vector (Fixed) is a common operation in 3D
; lighting operations,
; where Array = (x0,y0,z0),(x1,y1,z1),...
; and Fixed = (xF,yF,zF)
; A dot product is defined as the scalar quantity
; d0 = x0*xF + y0*yF + z0*zF.
; AoS code (1 vertice alla volta)
; All values marked DC are "don't-care."
; In the AOS model, the vertices are stored in the
; xyz format
    movaps xmm0, Array      ; xmm0 = DC, x0, y0, z0
    movaps xmm1, Fixed     ; xmm1 = DC, xF, yF, zF
    mulps xmm0, xmm1      ; xmm0 = DC, x0*xF, y0*yF, z0*zF
    movhps xmm1, xmm0     ; xmm1 = DC, DC, DC, x0*xF
    addps xmm1, xmm0      ; xmm0 = DC, DC, DC,
    ; x0*xF+z0*zF
    movaps xmm2, xmm1
    shufps xmm2, xmm2,55h ; xmm2 = DC, DC, DC, y0*yF
    addps mm2, xmm1      ; xmm1 = DC, DC, DC,
    ; x0*xF+y0*yF+z0*zF
; SoA code (4 vertici alla volta)
;
; X = x0,x1,x2,x3
; Y = y0,y1,y2,y3
; Z = z0,z1,z2,z3
; A = xF,xF,xF,xF
; B = yF,yF,yF,yF
; C = zF,zF,zF,zF
    movaps xmm0, X        ; xmm0 = x0,x1,x2,x3
    movaps xmm1, Y        ; xmm0 = y0,y1,y2,y3
    movaps xmm2, Z        ; xmm0 = z0,z1,z2,z3
    mulps xmm0, A         ; xmm0 = x0*xF, x1*xF, x2*xF, x3*xF
    mulps xmm1, B         ; xmm1 = y0*yF, y1*yF, y2*yF, y3*xF
    mulps xmm2, C         ; xmm2 = z0*zF, z1*zF, z2*zF, z3*zF
    addps xmm0, xmm1
    addps xmm0, xmm2     ; xmm0 = (x0*xF+y0*yF+z0*zF), ...
```

La tecnica SoA permette di gestire 4 vettori alla volta, aggiungete il fatto che giocando con i comandi di prefetch è possibile diminuire al massimo la latenza dovuta all'accesso in memoria per prelevare i dati, (li facciamo trovare al processore già nella cache), e vedrete che i benefici ci sono eccome, specialmente nel trattare grosse quantità di

dati. Immaginate ogni singolo componente della vostra struttura come un singolo flusso (stream) di dati, con l'organizzazione SoA i dati meno frequentemente usati non saranno caricati, e quindi sarà risparmiata banda nell'accesso in memoria, evitando il prefetch di dati che poi non verranno usati, esempio:

```
NumOfGroups = NumOfVertices/SIMDwidth

typedef struct{
    float x[SIMDwidth];
    float y[SIMDwidth];
    float z[SIMDwidth];
} VerticesCoordList;
typedef struct{
    int a[SIMDwidth];
    int b[SIMDwidth];
    int c[SIMDwidth];
    . . .
} VerticesColorList;
VerticesCoordList VerticesCoord[NumOfGroups];
VerticesColorList VerticesColor[NumOfGroups];
```

Quindi se usiamo lo stream VerticesCoord[] 10 volte di più che quello VerticesColor[], risparmieremo tempo perché tratteremo solo lui, senza caricare inutilmente ogni volta anche i dati dell'altro stream. Tutto molto bello per quel che riguarda la velocità che acquistiamo nei calcoli, ma DirectX, giusto per citare un API grafica, non accetta una lista di vertici in formato SOA, dobbiamo trasformare la nostra vertex list in formato AOS, in modo da poterla passare con tranquillità ad un vertex buffer DirectX per il successivo render a video. (Vi rimando alla documentazione ufficiale M\$ per la definizione e l'uso del vertex buffer e su come funziona DirectX stesso). Ancora una volta possiamo appoggiarci alle funzioni messe a disposizione da SSE, e trasformare 'al volo' 4 vertici SOA in AOS per poi darli in pasto nel modo corretto a DirectX, ecco qui lo snip di codice che fa proprio questa cosa, scritto usando le istruzioni intrinsic:

```
;g_sseparticles è una struttura globale organizzata secondo il metodo SOA
;quindi m_vCurPosX sono quattro 'coordinate x' x0,x1,x2,x3 stesso dicasi
;per m_vCurPosY e m_vCurPosZ
;v1..v4 sono semplicemente 4 vettori AOS però allocati 'aligned 16byte'
;per permettere l'uso di istruzioni SSE più veloci rispetto a quelle che devono
;gestire dati 'non aligned'.

xmm0 = _mm_load_ps(&g_sseparticles[i].m_vCurPosX.m128_f32[0]);
xmm1 = _mm_load_ps(&g_sseparticles[i].m_vCurPosY.m128_f32[0]);
xmm2 = _mm_load_ps(&g_sseparticles[i].m_vCurPosZ.m128_f32[0]);
xmm3 = _mm_set_ps1(1.0f);

xmm4 = _mm_unpacklo_ps(xmm0, xmm1);
xmm6 = _mm_unpackhi_ps(xmm0, xmm1);
xmm5 = _mm_unpacklo_ps(xmm2, xmm3);
xmm7 = _mm_unpackhi_ps(xmm2, xmm3);

xmm0 = _mm_shuffle_ps(xmm4, xmm5, _MM_SHUFFLE(1, 0, 1, 0));
xmm1 = _mm_shuffle_ps(xmm4, xmm5, _MM_SHUFFLE(3, 2, 3, 2));
xmm2 = _mm_shuffle_ps(xmm6, xmm7, _MM_SHUFFLE(1, 0, 1, 0));
xmm3 = _mm_shuffle_ps(xmm6, xmm7, _MM_SHUFFLE(3, 2, 3, 2));

_mm_store_ps(v1, xmm0);
_mm_store_ps(v2, xmm1);
_mm_store_ps(v3, xmm2);
_mm_store_ps(v4, xmm3);

;alla fine della fiera v1 conterrà x0,y0,z0,w0 e ancora v2 conterrà x1,y1,z1,w1 etc.etc.
```

## Piccolo programma di test per le SSE

Di questi micro-benchmark ne sono pieni gli hard disk, poteva mancare nel mio? Certo che no, eccolo qui, l'inutilità fatta programma, però permette comunque di muovere i primi passi e fare qualche considerazione sull'uso di queste istruzioni. Il programma in oggetto lo potete scrivere con qualsiasi compilatore che è in grado di gestire le SSE tramite la direttiva `_asm`. Non vi riporto tutto, ma solo qualche snip del codice che vi servirà per mettere in piedi un'applicazione console e fare due prove:

Questa è la parte relativa all'allocazione dinamica degli array necessari alla prova

```
.
.
.
```

```

#define TCOUNT 80000000 //ottanta milioni di vettori (float*3)
#define QCOUNT 20000000 //venti milioni di quad-float (__m128)
.
.
.
.
.
// initialize test data

vector3* v = (vector3*) malloc(TCOUNT*sizeof(vector3));
if(v == NULL){
    printf("(malloc)Memoria Insufficiente\n\r");
    return 1;
}

__m128* vx4 = (__m128*)_mm_malloc(QCOUNT*sizeof(__m128),16);
float* vx = (float*)vx4;

__m128* vy4 = (__m128*)_mm_malloc(QCOUNT*sizeof(__m128),16);
float* vy = (float*)vy4;

__m128* vz4 = (__m128*)_mm_malloc(QCOUNT*sizeof(__m128),16);
float* vz = (float*)vz4;

if(vx4 == NULL || vy4 == NULL || vz4 == NULL){
    printf("(MM_MALLOC) Memoria Insufficiente\n\r");
    return 1;
}

```

c'è poco da dire, vector3 è una classe d'appoggio che definisce un vettore composto tra tre float (x,y,z), vx4,vy4,vz4 sono array organizzati secondo il metodo SOA, e conterranno rispettivamente le coordinate x,y e z dei vettori allocati precedentemente. Nota di colore: nella mia applicazione sono riuscito ad allocare un massimo di 1,5Gb totali di ram usando il Intel Compiler v10 e 1,8 usando invece il Visual Studio 2008 Express. Questo nonostante avessi 2,5Gb liberi di memoria. Sembra che ci siano differenze nella common runtime delle due versioni, ma non ho avuto voglia di indagare oltre. Potrebbe essere una cosa interessante capire cosa diamine cambia tra le due. Questa di seguito è la routine che esegue l'inverso della radice di un vettore utilizzando codice SSE e una sola iterazione del metodo Newton-Rhapson:

```

const __m128 _half4={0.5f,0.5f,0.5f,0.5f};
const __m128 _three={3.0f,3.0f,3.0f,3.0f};

static __forceinline __m128 fastrsqrt( const __m128 v ){
    const __m128 approx = _mm_rsqrt_ps( v );
    const __m128 muls = _mm_mul_ps( _mm_mul_ps(v, approx), approx);
    const __m128 uno = _mm_mul_ps( _half4, approx);
    const __m128 due = _mm_sub_ps( _three, muls);
    return _mm_mul_ps( uno, due );
}

```

ed infine i due cicli da testare, uno scritto per le SSE e l'altro invece no

```

.
.
.
//SSE cycle
for ( i = 0; i < QCOUNT; i++ ) {
    const __m128 sqx4 = _mm_mul_ps( vx4[i], vx4[i] );
    const __m128 sqy4 = _mm_mul_ps( vy4[i], vy4[i] );
    const __m128 sqz4 = _mm_mul_ps( vz4[i], vz4[i] );
    const __m128 rlen4 = fastrsqrt( _mm_add_ps( _mm_add_ps( sqx4, sqy4 ), sqz4 ) );

    vx4[i] = _mm_mul_ps( vx4[i], rlen4 );
    vy4[i] = _mm_mul_ps( vy4[i], rlen4 );
    vz4[i] = _mm_mul_ps( vz4[i], rlen4 );
}

.
.
.
//plain FPU cycle
for ( i = 0; i < TCOUNT; i++ ) {
    const float rlen = 1.0f / sqrtf( v[i].x * v[i].x + v[i].y * v[i].y + v[i].z * v[i].z );
    v[i].x *= rlen, v[i].y *= rlen; v[i].z *= rlen;
}
.

```

mettete un paio di GetTickCount() per cronometrare i due cicli e fatevi quattro risate (oppure usate il QueryPerformanceCounter() di cui ho spiegato l'uso nel mio articolo sugli exe ai minimi termini).

## Considerazioni

Compilete il programma utilizzando vari switch e comparate i tempi ottenuti, vi riporto di seguito i tempi per l'esecuzione del test su un P4 HT 3.0GHz, sfruttando metà processore. Si poteva fare un'applicazione multithread per saturare le capacità dell'hyperthreading, ma ai fini del programma non era necessario, anzi potrebbe essere uno spunto per parlare in futuro di OpenMP e del multiprocessing in generale. Comunque ecco qui i tempi per snocciolare 80 milioni di vettori (!!):

Project Properties	SSE cycle (ms)	FPU cycle (ms)	Gain
/fp:strict	1016	4328	4.25
/fp:precise	1032	3562	3.45
/fp:fast	984	2328	2.36
/fp:fast /arch:SSE	985	2218	2.25
/fp:fast /arch:SSE2	968	1953	2.01

Potete cambiare le proprietà del codice generato dal Visual nelle proprietà del progetto, pagina "C/C++" item "Code Generation" tag "Floating point model" e "Enable Enhanced Instruction Set". Come potete vedere c'è qualcosa che non torna, come mai la routine SSE non è sempre 4 volte più veloce della sua corrispondente FPU? E' presto detto: nel primo caso utilizziamo il modello strict, che setta la fpu all'adesione perfetta allo standard IEEE, e che la rallenta, anche se in questo modo otteniamo i risultati più precisi. Nel secondo caso la fpu ricorre a qualche scappatoia per svincolarsi un po' dallo standard, sempre ottenendo risultati di una certa precisione (ricordatevi che la rappresentazione dei float interna alla FPU usa ben 80bit). Negli ultimi due casi il compilatore ricorre alla vettorizzazione del codice, ovvero sostituisce le istruzioni della FPU con più veloce codice SIMD, ecco come è tradotto:

```
; 147 : for ( i = 0; i < TCOUNT; i++ ) {
; 148 :   const float rlen = 1.0f / sqrtf( v[i].x * v[i].x + v[i].y * v[i].y + v[i].z * v[i].z );

      movss   xmm1, DWORD PTR [eax-4]
      movss   xmm2, DWORD PTR [eax-8]
      movss   xmm0, DWORD PTR [eax]
      movaps  xmm3, xmm2
      mulss   xmm3, xmm2
      movaps  xmm5, xmm1
      mulss   xmm5, xmm1
      addss   xmm3, xmm5
      movaps  xmm5, xmm0
      mulss   xmm5, xmm0
      addss   xmm3, xmm5
      sqrtss  xmm5, xmm3
      movaps  xmm3, xmm4
      divss   xmm3, xmm5

; 149 :           v[i].x *= rlen, v[i].y *= rlen; v[i].z *= rlen;

      mulss   xmm2, xmm3
      mulss   xmm1, xmm3
      mulss   xmm0, xmm3
      movss   DWORD PTR [eax-8], xmm2
      movss   DWORD PTR [eax-4], xmm1
      movss   DWORD PTR [eax], xmm0
      add     eax, 12 ; 0000000cH
      sub     ecx, 1
      jne     SHORT $LL3@main

; 150 :   }
; 151 : }
```

Bada ganzo, non ci avevo fatto nemmeno caso, il compilatore ritiene più veloce fare la divisione 1/sqrtss piuttosto che usare rsqrtss, eppure come unico valore al numeratore c'è 1.0f, vabbè problemi suoi. Comunque come potete vedere usa le forme "scalar" delle istruzioni, quindi in realtà utilizza un quarto dell'ampiezza che il registro XMM mette a disposizione. Nonostante questo il 'coprocessore' SIMD esegue il codice al doppio della velocità della FPU, anche se ci tengo ancora una volta a ricordare che la precisione non è assolutamente uguale. Ci tengo a dirlo

perchè se è vero che per un Engine3D non ci interessa la precisione assoluta, per una qualsiasi altra applicazione nell'ambito scientifico invece ci interessa eccome. Ad esempio, il codice sopra riportato produce, caricando 1, 2 e 3 rispettivamente in x,y e z:

```
v[0].x=0.267261 v[0].y=0.534522 v[0].z=0.801784 // Risultato FPU  
v[0].x=0.267261 v[0].y=0.534522 v[0].z=0.801784 // Risultato SSE con 1 iterazione Newton-Raphson  
v[0].x=0.267212 v[0].y=0.534424 v[0].z=0.801636 // Risultato SSE senza Newton-Raphson
```

Come vedete già nel float, col solo ausilio delle istruzioni SSE, non possiamo spingerci in precisione oltre la terza cifra decimale, fa un po schifo effettivamente. E' anche vero che ho volutamente forzato la mano andando a cercare le istruzioni del reciproco, che leggono da una tabella, con le istruzioni "normali" la precisione è ovviamente maggiore. Il bello comunque viene quando si lavora con i double! Per fortuna la matematica viene incontro con ausili del tipo Newton-Raphson, questo ci permette di dare una bella mano ad SSE per rimmetterlo in competizione con la FPU. Se nelle opzioni del visual gli dite "per favore mi fai vedere cosa combini?" che è settabile in proprietà del progetto, pagina "C/C++" item "Output Files" tag "Assembler Output" otterrete il listato del codice C interposto alla corrispondente compilazione in assembler, e vi potrete rendere conto di persona di come il codice viene man mano interpretato.