Intel SIMD architecture

Computer Organization and Assembly Languages

Yung-Yu Chuang

2006/12/25

Reference

- *Intel MMX for Multimedia PCs*, CACM, Jan. 1997
- Chapter 11 *The MMX Instruction Set*, The Art of Assembly
- Chap. 9, 10, 11 of IA-32 Intel Architecture Software Developer's Manual: Volume 1: Basic Architecture

Overview

- SIMD
- MMX architectures
- MMX instructions
- examples
- SSE/SSE2

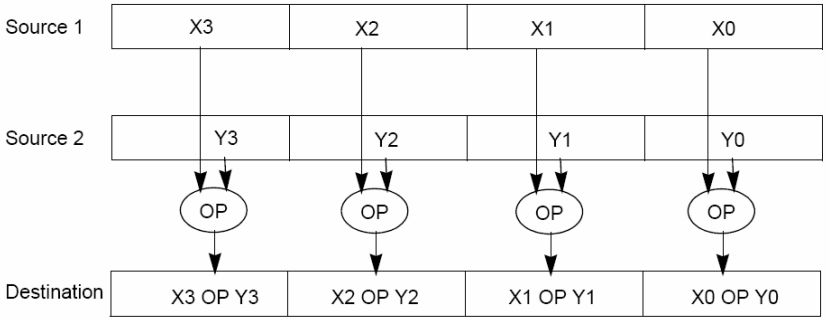
- SIMD instructions are probably the best place to use assembly since compilers usually do not do a good job on using these instructions

Performance boost

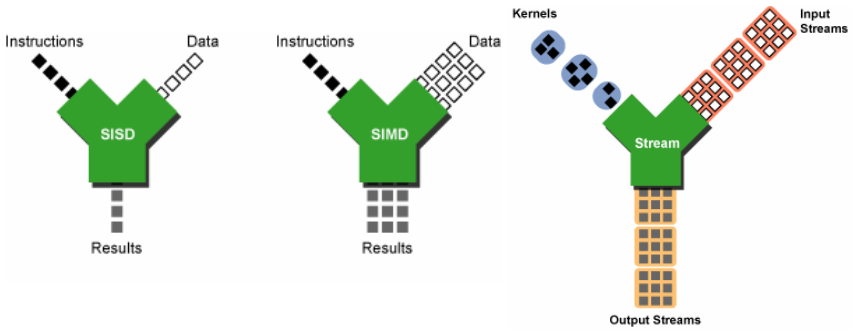
- Increasing clock rate is not fast enough for boosting performance
- Architecture improvement is more significant such as pipeline/cache/SIMD
- Intel analyzed multimedia applications and found they share the following characteristics:
  - Small native data types (8-bit pixel, 16-bit audio)
  - Recurring operations
  - Inherent parallelism

SIMD

- SIMD (single instruction multiple data) architecture performs the same operation on multiple data elements in parallel
- **`PADDW MM0, MM1`**



SISD/SIMD/Streaming



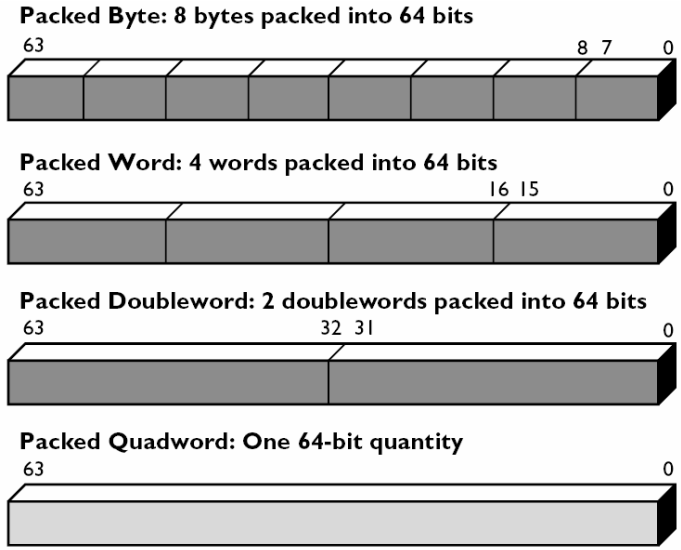
IA-32 SIMD development

- MMX (Multimedia Extension) was introduced in 1996 (Pentium with MMX and Pentium II).
- SSE (Streaming SIMD Extension) was introduced with Pentium III.
- SSE2 was introduced with Pentium 4.
- SSE3 was introduced with Pentium 4 supporting hyper-threading technology. SSE3 adds 13 more instructions.

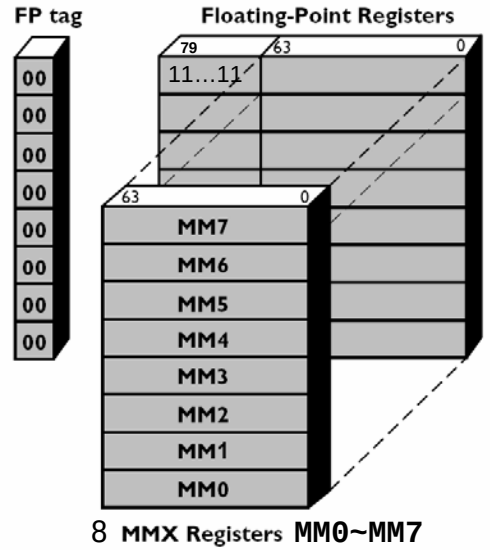
MMX

- After analyzing a lot of existing applications such as graphics, MPEG, music, speech recognition, game, image processing, they found that many multimedia algorithms execute the same instructions on many pieces of data in a large data set.
- Typical elements are small, 8 bits for pixels, 16 bits for audio, 32 bits for graphics and general computing.
- New data type: 64-bit packed data type. Why 64 bits?
  - Good enough
  - Practical

MMX data types



MMX integration into IA



NaN or infinity as real because bits 79-64 are zeros.

Even if MMX registers are 64-bit, they don't extend Pentium to a 64-bit CPU since only logic instructions are provided for 64-bit data.

Compatibility

- To be fully compatible with existing IA, no new mode or state was created. Hence, for context switching, no extra state needs to be saved.
- To reach the goal, MMX is hidden behind FPU. When floating-point state is saved or restored, MMX is saved or restored.
- It allows existing OS to perform context switching on the processes executing MMX instruction without be aware of MMX.
- However, it means MMX and FPU can not be used at the same time.

Compatibility

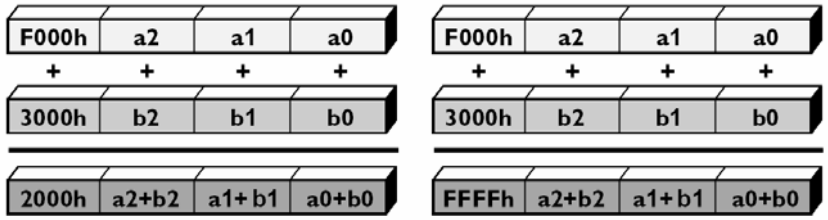
- Although Intel defenses their decision on aliasing MMX to FPU for compatibility. It is actually a bad decision. OS can just provide a service pack or get updated.
- It is why Intel introduced SSE later without any aliasing

MMX instructions

- 57 MMX instructions are defined to perform the parallel operations on multiple data elements packed into 64-bit data types.
- These include **`add`**, **`subtract`**, **`multiply`**, **`compare`**, and **`shift`**, **`data conversion`**, **`64-bit data move`**, **`64-bit logical operation`** and **`multiply-add`** for multiply-accumulate operations.
- All instructions except for data move use MMX registers as operands.
- Most complete support for 16-bit operations.

Saturation arithmetic

- Useful in graphics applications.
- When an operation overflows or underflows, the result becomes the largest or smallest possible representable number.
- Two types: signed and unsigned saturation



wrap-around saturating

MMX instructions

Category		Wraparound	Signed Saturation	Unsigned Saturation
Arithmetic	Addition	PADDB, PADDW, PADDD	PADDSB, PADDSW	PADDUSB, PADDUSW
	Subtraction	PSUBB, PSUBW, PSUBD	PSUBSB, PSUBSW	PSUBUSB, PSUBUSW
	Multiplication	PMULL, PMULH		
	Multiply and Add	PMADD		
Comparison	Compare for Equal	PCMPEQB, PCMPEQW, PCMPEQD		
	Compare for Greater Than	PCMPGTPB, PCMPGTPW, PCMPGTPD		
Conversion	Pack		PACKSSWB, PACKSSDW	PACKUSWB
Unpack	Unpack High	PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ		
	Unpack Low	PUNPCKLBW, PUNPCKLWD, PUNPCKLDQ		

MMX instructions

		Packed	Full Quadword
Logical	And And Not Or Exclusive OR		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	MOVD MOVD MOVD	MOVQ MOVQ MOVQ
Empty MMX State		EMMS	

Call it before you switch to FPU from MMX;
Expensive operation

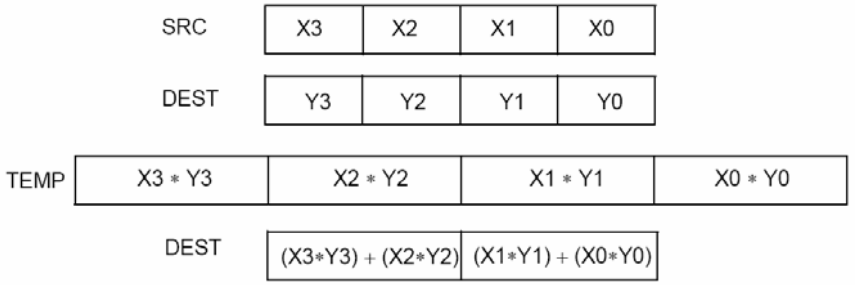
Arithmetic

- **PADDB/PADDW/PADDD**: add two packed numbers, no CFLAGS is set, ensure overflow never occurs by yourself
- Multiplication: two steps
- **PMULLW**: multiplies four words and stores the four lo words of the four double word results
- **PMULHW/PMULHUW**: multiplies four words and stores the four hi words of the four double word results. **PMULHUW** for unsigned.

Arithmetic

- **PMADDWD**

DEST[31:0] ← (DEST[15:0] * SRC[15:0]) + (DEST[31:16] * SRC[31:16]);
DEST[63:32] ← (DEST[47:32] * SRC[47:32]) + (DEST[63:48] * SRC[63:48]);



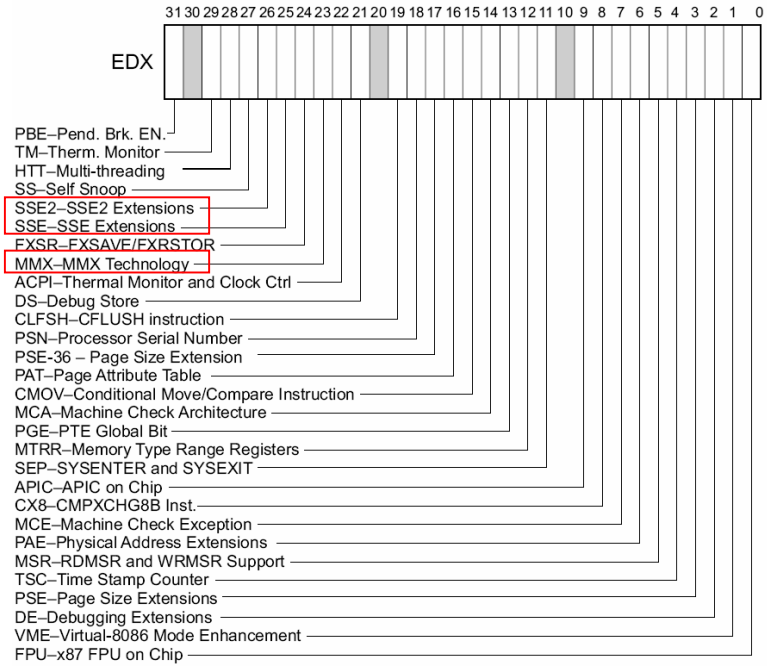
Detect MMX/SSE

```
mov   eax, 1 ; request version info
cpuid    ; supported since Pentium
test  edx, 00800000h ;bit 23
         ; 02000000h (bit 25) SSE
         ; 04000000h (bit 26) SSE2
jnz   HasMMX
```

cpuid

Initial EAX Value	Information Provided about the Processor	
	Basic CPUID Information	
0H	EAX	Maximum Input Value for Basic CPUID Information (see Table 3-13)
	EBX	"Genu"
	ECX	"ntel"
	EDX	"inel"
01H	EAX	Version Information: Type, Family, Model, and Stepping ID (see Figure 3-5)
	EBX	Bits 7-0: Brand Index Bits 15-8: CLFLUSH line size (Value * 8 = cache line size in bytes) Bits 23-16: Maximum number of logical processors in this physical package. Bits 31-24: Initial APIC ID
	ECX	Extended Feature Information (see Figure 3-6 and Table 3-15)
	EDX	Feature Information (see Figure 3-7 and Table 3-16)
02H	EAX	Cache and TLB Information (see Table 3-17)
	EBX	Cache and TLB Information
	ECX	Cache and TLB Information
	EDX	Cache and TLB Information

:
:

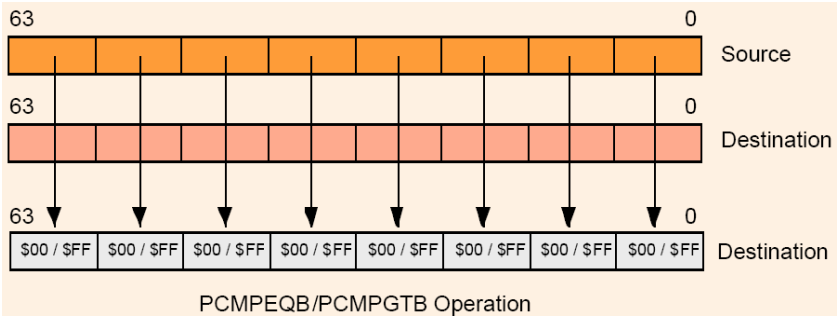


Example: add a constant to a vector

```
char d[]={5, 5, 5, 5, 5, 5, 5, 5};
char clr[]={65,66,68,...,87,88}; // 24 bytes
__asm{
    movq mm1, d
    mov cx, 3
    mov esi, 0
L1: movq mm0, clr[esi]
    paddb mm0, mm1
    movq clr[esi], mm0
    add esi, 8
    loop L1
    emms
}
```

Comparison

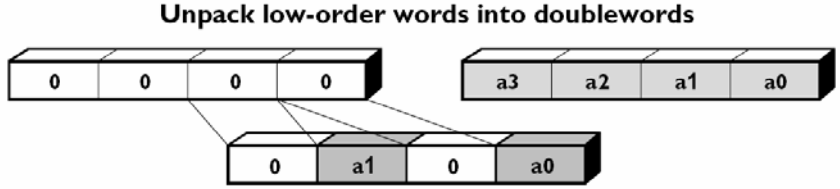
- No CFLAGS, how many flags will you need? Results are stored in destination.
- EQ/GT, no LT



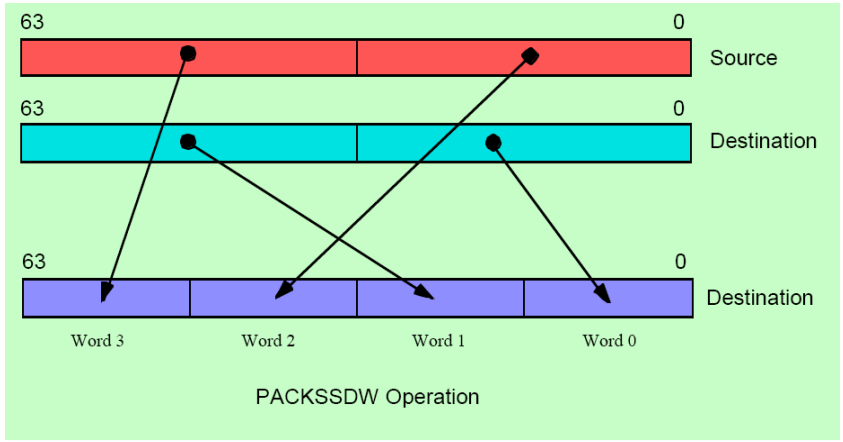
PCMPEQB/PCMPGTB Operation

Change data types

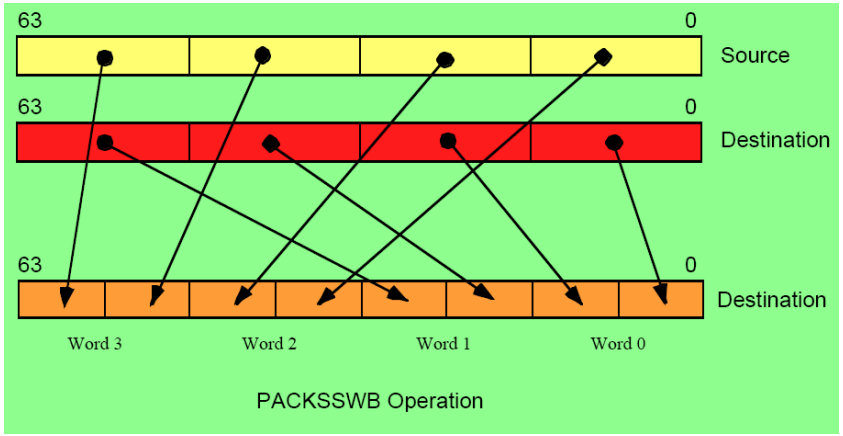
- Pack: converts a larger data type to the next smaller data type.
- Unpack: takes two operands and interleave them. It can be used for expand data type for immediate calculation.



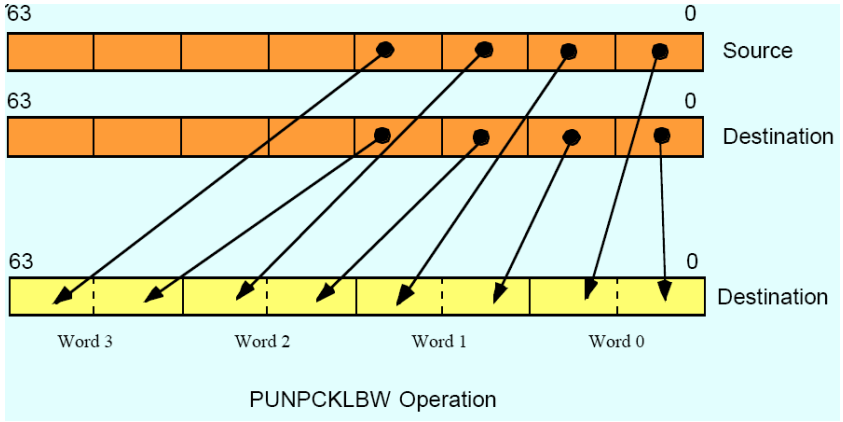
Pack with signed saturation



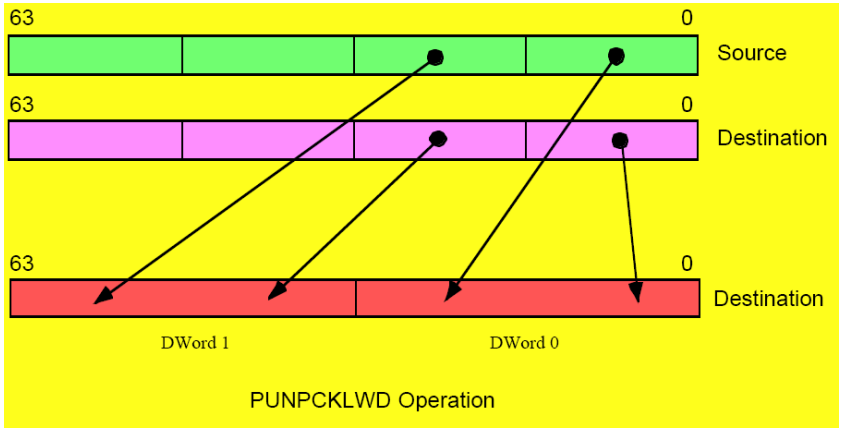
Pack with signed saturation



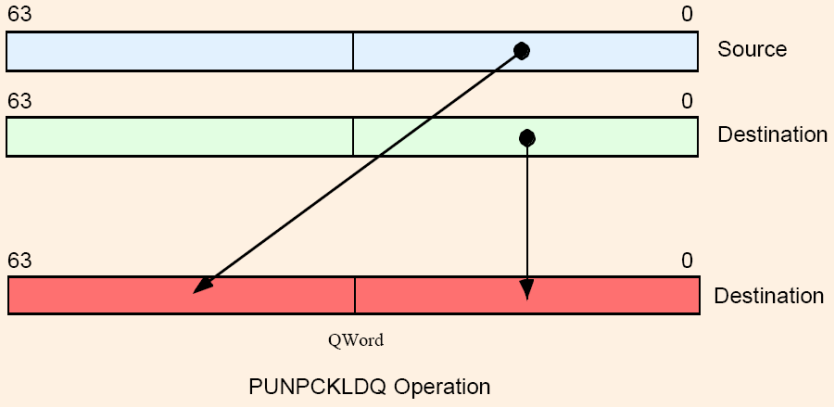
Unpack low portion



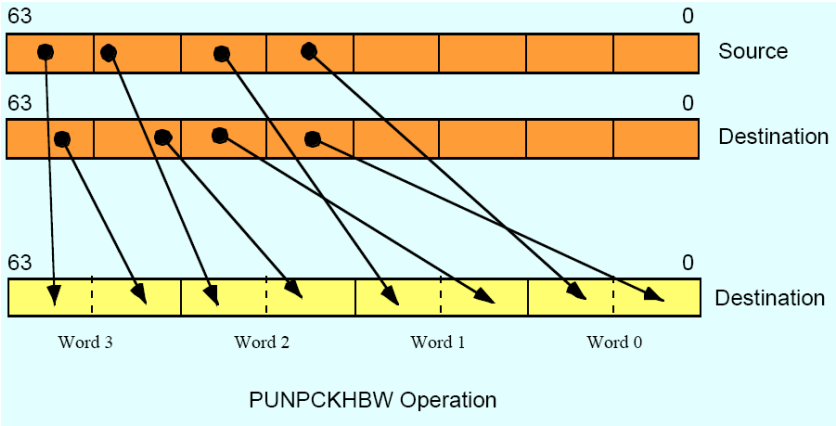
Unpack low portion



Unpack low portion



Unpack high portion

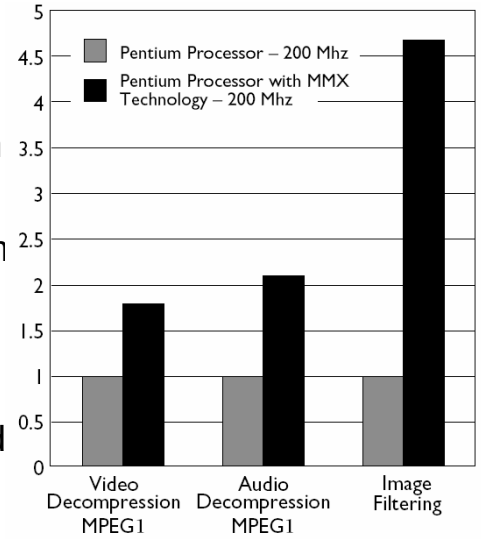


Performance boost (data from 1996)

Benchmark kernels: FFT, FIR, vector dot-product, IDCT, motion compensation

65% performance gain

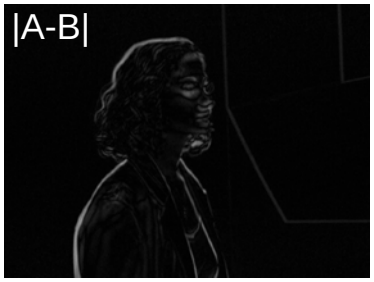
Lower the cost of multimedia programs by removing the need of specialized DSP chips



Keys to SIMD programming

- Efficient data layout
- Elimination of branches

Application: frame difference



Application: frame difference

(A-B) or (B-A)

Application: frame difference

```
MOVQ      mm1, A //move 8 pixels of image A
MOVQ      mm2, B //move 8 pixels of image B
MOVQ      mm3, mm1 // mm3=A
PSUBSB    mm1, mm2 // mm1=A-B
PSUBSB    mm2, mm3 // mm2=B-A
POR       mm1, mm2 // mm1=|A-B|
```

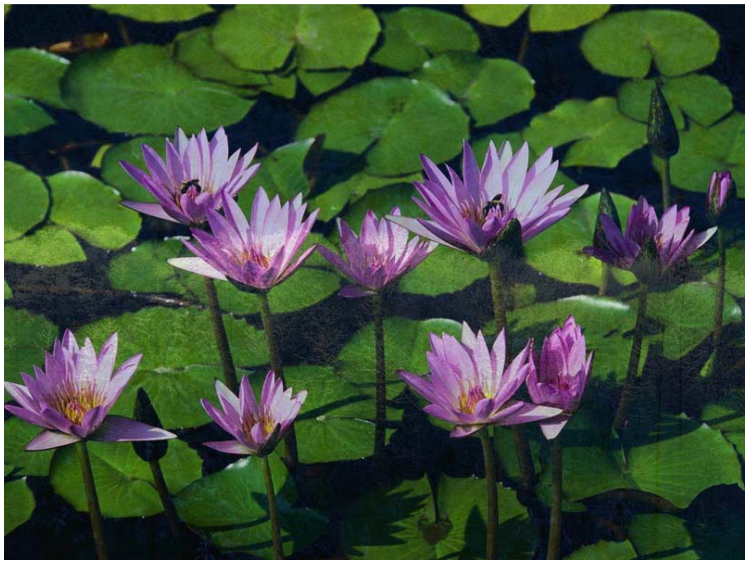
Example: image fade-in-fade-out

A

B

$A*\alpha + B*(1-\alpha) = B+\alpha(A-B)$

α=0.75

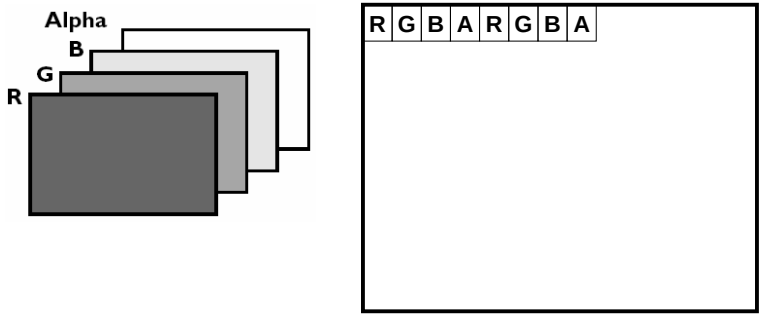


α=0.5

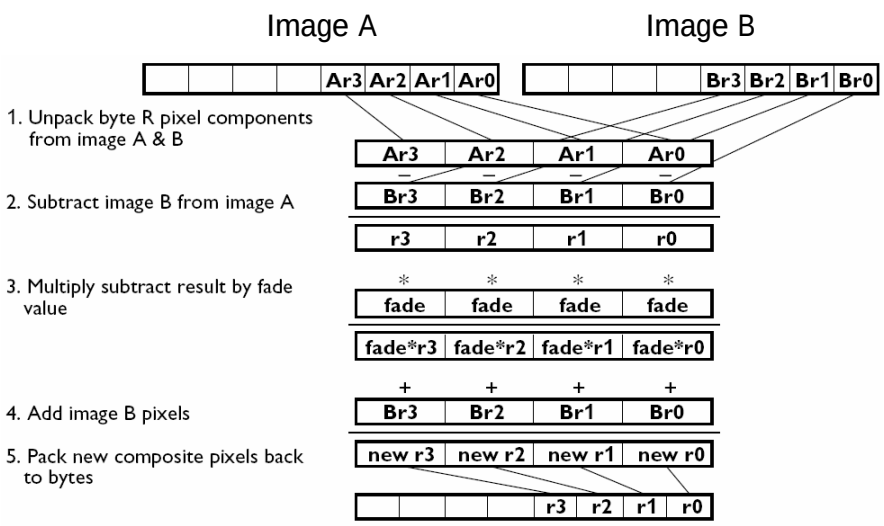
α=0.25

Example: image fade-in-fade-out

- Two formats: planar and chunky
- In Chunky format, 16 bits of 64 bits are wasted
- So, we use planar in the following example



Example: image fade-in-fade-out



Example: image fade-in-fade-out

```
MOVQ      mm0, alpha//4 16-b zero-padding α
MOVD      mm1, A //move 4 pixels of image A
MOVD      mm2, B //move 4 pixels of image B
PXOR      mm3, mm3 //clear mm3 to all zeroes
//unpack 4 pixels to 4 words
PUNPCKLBW mm1, mm3 // Because B-A could be
PUNPCKLBW mm2, mm3 // negative, need 16 bits
PSUBW     mm1, mm2 //(B-A)
PMULHW    mm1, mm0 //(B-A)*fade/256
PADDW     mm1, mm2 //(B-A)*fade + B
//pack four words back to four bytes
PACKUSWB  mm1, mm3
```

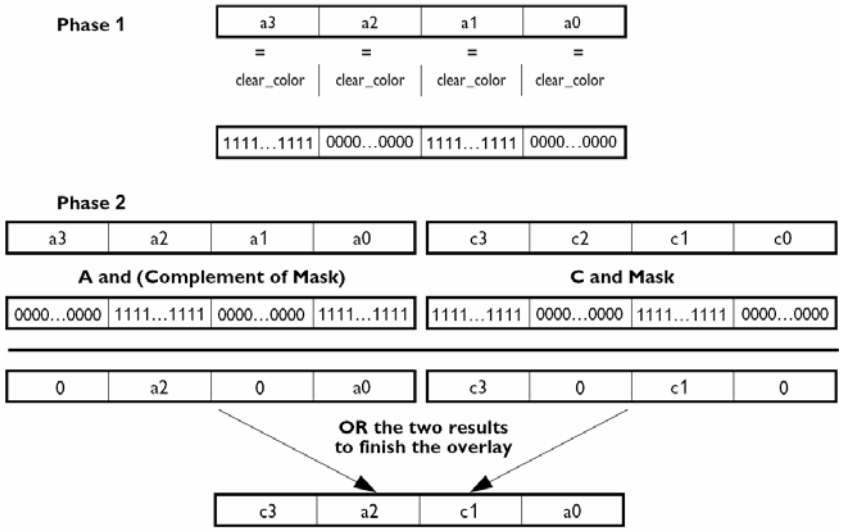
Data-independent computation

- Each operation can execute without needing to know the results of a previous operation.
- Example, sprite overlay

```
for i=1 to sprite_Size
  if  sprite[i]=clr
  then out_color[i]=bg[i]
  else out_color[i]=sprite[i]
```

- How to execute data-dependent calculations on several pixels in parallel.

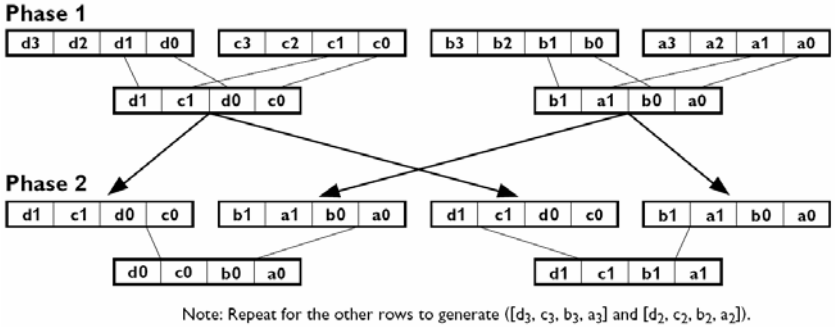
Application: sprite overlay



Application: sprite overlay

```
MOVQ     mm0, sprite
MOVQ     mm2, mm0
MOVQ     mm4, bg
MOVQ     mm1, clr
PCMPEQW  mm0, mm1
PAND     mm4, mm0
PANDN    mm0, mm2
POR      mm0, mm4
```

Application: matrix transport



Note: Repeat for the other rows to generate ([d_3, c_3, b_3, a_3] and [d_2, c_2, b_2, a_2]).

MMX code sequence operation:

```
movq      mm1, row1   ; load pixels from first row of matrix
movq      mm2, row2   ; load pixels from second row of matrix
movq      mm3, row3   ; load pixels from third row of matrix
movq      mm4, row4   ; load pixels from fourth row of matrix
punpcklwd mm1, mm2    ; unpack low order words of rows 1 & 2, mm 1 = [b1, a1, b0, a0]
punpcklwd mm3, mm4    ; unpack low order words of rows 3 & 4, mm3 = [d1, c1, d0, c0]
movq      mm5, mm1    ; copy mm1 to mm5
punpckldq mm1, mm3    ; unpack low order doublewords -> mm2 = [d0, c0, b0, a0]
punpckhdq mm5, mm3    ; unpack high order doublewords -> mm5 = [d1, c1, b1, a1]
```

Application: matrix transport

```
char M1[4][8];// matrix to be transposed
char M2[8][4];// transposed matrix
int n=0;
for (int i=0;i<4;i++)
  for (int j=0;j<8;j++)
    { M1[i][j]=n; n++; }
__asm{
//move the 4 rows of M1 into MMX registers
movq mm1,M1
movq mm2,M1+8
movq mm3,M1+16
movq mm4,M1+24
```

Application: matrix transport

```
//generate rows 1 to 4 of M2
punpcklbw mm1, mm2
punpcklbw mm3, mm4
movq mm0, mm1
punpcklwd mm1, mm3 //mm1 has row 2 & row 1
punpckhwd mm0, mm3 //mm0 has row 4 & row 3
movq M2, mm1
movq M2+8, mm0
```

Application: matrix transport

```
//generate rows 5 to 8 of M2
movq mm1, M1 //get row 1 of M1
movq mm3, M1+16 //get row 3 of M1
punpckhbw mm1, mm2
punpckhbw mm3, mm4
movq mm0, mm1
punpcklwd mm1, mm3 //mm1 has row 6 & row 5
punpckhwd mm0, mm3 //mm0 has row 8 & row 7
//save results to M2
movq M2+16, mm1
movq M2+24, mm0
emms
} //end
```

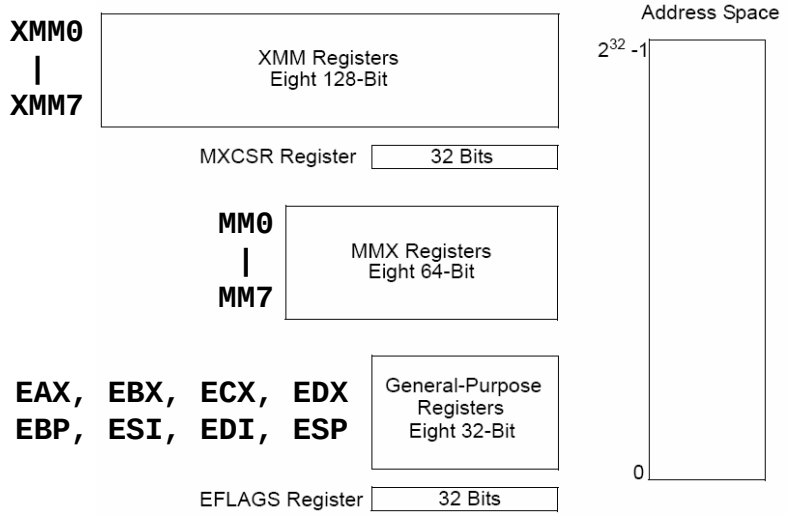
SSE

- Adds eight 128-bit registers
- Allows SIMD operations on packed single-precision floating-point numbers.

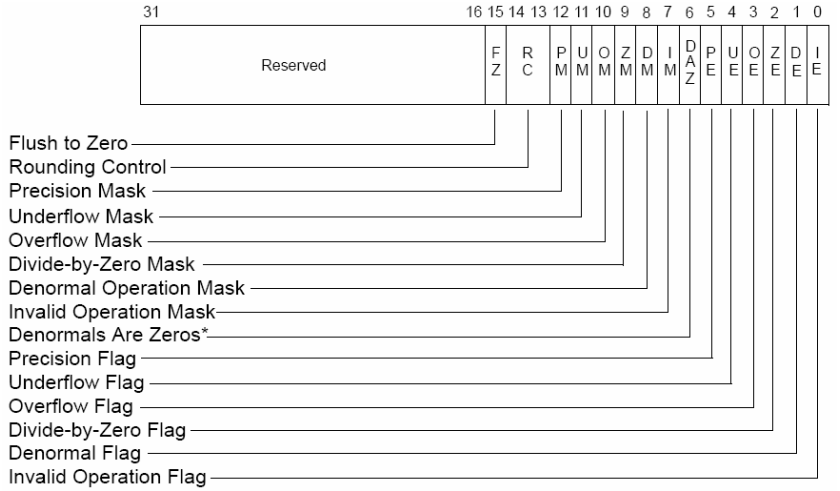
SSE features

- Add eight 128-bit data registers (XMM registers) in non-64-bit modes; sixteen XMM registers are available in 64-bit mode.
- 32-bit MXCSR register (control and status)
- Add a new data type: 128-bit packed single-precision floating-point (4 FP numbers.)
- Instruction to perform SIMD operations on 128-bit packed single-precision FP and additional 64-bit SIMD integer operations.
- Instructions that explicitly prefetch data, control data cacheability and ordering of store

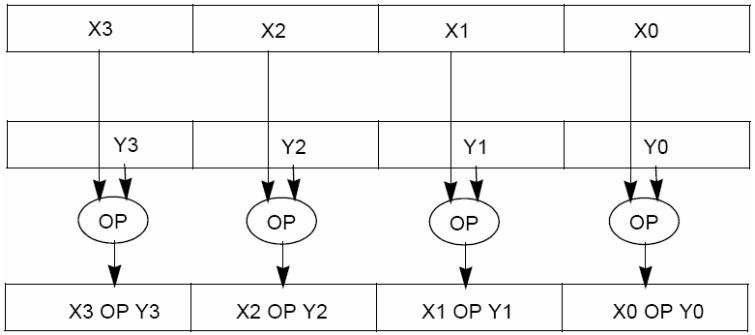
SSE programming environment



MXCSR control and status register

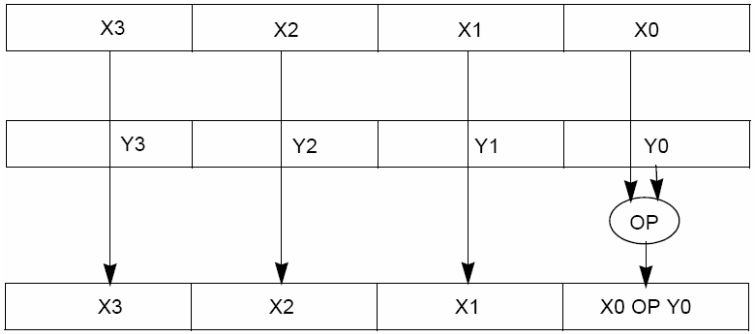


SSE packed FP operation



- **ADDPS/SUBPS**: packed single-precision FP

SSE scalar FP operation



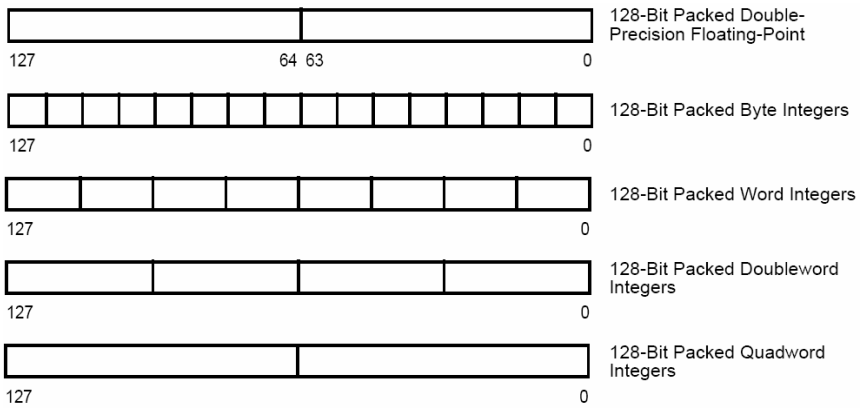
- **ADDSS/SUBSS**: scalar single-precision FP used as FPU?

SSE2

- Provides ability to perform SIMD operations on double-precision FP, allowing advanced graphics such as ray tracing
- Provides greater throughput by operating on 128-bit packed integers, useful for RSA and RC5

SSE2 features

- Add data types and instructions for them



- Programming environment unchanged

Example

```
void add(float *a, float *b, float *c) {
  for (int i = 0; i < 4; i++)
    c[i] = a[i] + b[i];
}
__asm {
mov    eax, a
mov    edx, b
mov    ecx, c
movaps xmm0, XMMWORD PTR [eax]
addps  xmm0, XMMWORD PTR [edx]
movaps XMMWORD PTR [ecx], xmm0
}
```

movaps: move aligned packed single-precision FP
addps: add packed single-precision FP

Intrinsics

- An *intrinsic* is a function known by the compiler that directly maps to a sequence of one or more assembly language instructions. Intrinsic functions are inherently more efficient than called functions because no calling linkage is required.
- Intrinsics make the use of processor-specific enhancements easier because they provide a C/C++ language interface to assembly instructions. In doing so, the compiler manages things that the user would normally have to be concerned with, such as register names, register allocations, and memory locations of data.

Vector algebra

- Used extensively in graphics
- In C era, `typedef float vector[3];`
- In C++ era,

```
class Vector {
  private:
    float x , y , z;
};

Vector operator + ( const Vector& a ,
                    const float & b ) {
  return Vector( a.x+b, a.y+b, a.z+b );
}
```

SSE intrinsic

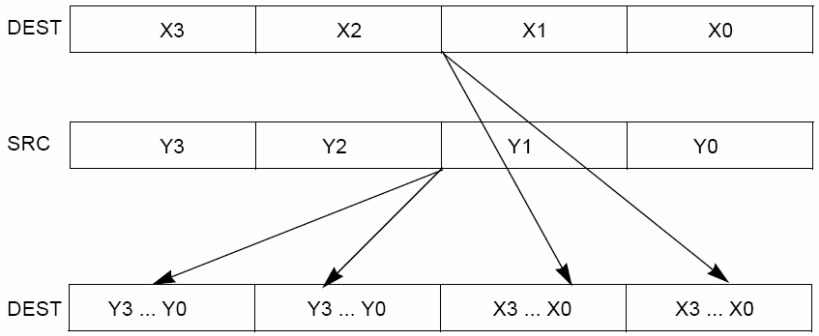
```
#include <xmmintrin.h>

__m128 a , b , c;
c = _mm_add_ps( a , b );

float a[4] , b[4] , c[4];
for( int i = 0 ; i < 4 ; ++ i )
    c[i] = a[i] + b[i];

// a = b * c + d / e;
__m128 a = _mm_add_ps( _mm_mul_ps( b , c ) ,
                       _mm_div_ps( d , e ) );
```

SSE Shuffle (SHUFPS)

SHUFPS xmm1, xmm2, imm8

Select[1..0] decides which DW of DEST to be copied to the 1st DW of DEST

...



SSE Shuffle (SHUFPS)

```
CASE (SELECT[1:0]) OF
    0:  DEST[31:0] ← DEST[31:0];
    1:  DEST[31:0] ← DEST[63:32];
    2:  DEST[31:0] ← DEST[95:64];
    3:  DEST[31:0] ← DEST[127:96];
ESAC;

CASE (SELECT[3:2]) OF
    0:  DEST[63:32] ← DEST[31:0];
    1:  DEST[63:32] ← DEST[63:32];
    2:  DEST[63:32] ← DEST[95:64];
    3:  DEST[63:32] ← DEST[127:96];
ESAC;

CASE (SELECT[5:4]) OF
    0:  DEST[95:64] ← SRC[31:0];
    1:  DEST[95:64] ← SRC[63:32];
    2:  DEST[95:64] ← SRC[95:64];
    3:  DEST[95:64] ← SRC[127:96];
ESAC;

CASE (SELECT[7:6]) OF
    0:  DEST[127:96] ← SRC[31:0];
    1:  DEST[127:96] ← SRC[63:32];
    2:  DEST[127:96] ← SRC[95:64];
    3:  DEST[127:96] ← SRC[127:96];
ESAC;
```

Example (cross product)

```
Vector cross(const Vector& a , const Vector& b ) {
    return Vector(
        ( a[1] * b[2] - a[2] * b[1] ) ,
        ( a[2] * b[0] - a[0] * b[2] ) ,
        ( a[0] * b[1] - a[1] * b[0] ) );
}
```

Example (cross product)

```
/* cross */
__m128 _mm_cross_ps( __m128 a , __m128 b ) {
  __m128 ea , eb;
  // set to a[1][2][0][3] , b[2][0][1][3]
  ea = _mm_shuffle_ps( a, a, _MM_SHUFFLE(3,0,2,1) );
  eb = _mm_shuffle_ps( b, b, _MM_SHUFFLE(3,1,0,2) );
  // multiply
  __m128 xa = _mm_mul_ps( ea , eb );
  // set to a[2][0][1][3] , b[1][2][0][3]
  a = _mm_shuffle_ps( a, a, _MM_SHUFFLE(3,1,0,2) );
  b = _mm_shuffle_ps( b, b, _MM_SHUFFLE(3,0,2,1) );
  // multiply
  __m128 xb = _mm_mul_ps( a , b );
  // subtract
  return _mm_sub_ps( xa , xb );
}
```

Example: dot product

- Given a set of vectors $\{v_1,v_2,...v_n\}=\{(x_1,y_1,z_1), (x_2,y_2,z_2),..., (x_n,y_n,z_n)\}$ and a vector $v_c=(x_c,y_c,z_c)$, calculate $\{v_c \cdot v_i\}$
- Two options for memory layout
- Array of structure (AoS)

```
typedef struct { float dc, x, y, z; } Vertex;
Vertex v[n];
```

- Structure of array (SoA)

```
typedef struct { float x[n], y[n], z[n]; }
        VerticesList;
VerticesList v;
```

Example: dot product (AoS)

```
movaps xmm0, v  ; xmm0 = DC, x0, y0, z0
movaps xmm1, vc ; xmm1 = DC, xc, yc, zc
mulps  xmm0, xmm1 ;xmm0=DC,x0*xc,y0*yc,z0*zc
movhlps xmm1, xmm0 ; xmm1= DC, DC, DC, x0*xc
addps  xmm1, xmm0 ; xmm1 = DC, DC, DC,
                  ;              x0*xc+z0*zc
movaps xmm2, xmm0
shufps xmm2, xmm2, 55h ; xmm2=DC,DC,DC,y0*yc
addps  xmm1, xmm2 ; xmm1 = DC, DC, DC,
                  ;        x0*xc+y0*yc+z0*zc
```

movhlps:DEST[63..0] := SRC[127..64]

Example: dot product (SoA)

```
; X = x1,x2,...,x3
; Y = y1,y2,...,y3
; Z = z1,z2,...,z3
; A = xc,xc,xc,xc
; B = yc,yc,yc,yc
; C = zc,zc,zc,zc
movaps xmm0, X ; xmm0 = x1,x2,x3,x4
movaps xmm1, Y ; xmm1 = y1,y2,y3,y4
movaps xmm2, Z ; xmm2 = z1,z2,z3,z4
mulps  xmm0, A ;xmm0=x1*xc,x2*xc,x3*xc,x4*xc
mulps  xmm1, B ;xmm1=y1*yc,y2*yc,y3*xc,y4*yc
mulps  xmm2, C ;xmm2=z1*zc,z2*zc,z3*zc,z4*zc
addps  xmm0, xmm1
addps  xmm0, xmm2 ;xmm0=(x0*xc+y0*yc+z0*zc)…
```

Reciprocal

```
#define FP_ONE_BITS 0x3F800000
// r = 1/p  from NVidia's fastmath.cpp
#define FP_INV(r,p)                                \
{                                                  \
    int _i = 2 * FP_ONE_BITS - *(int *)&(p);       \
    r = *(float *)&_i;                             \
    r = r * (2.0f - (p) * r);                      \
}
```

$$f(x_0+\varepsilon)=f(x_0)+f'(x_0)\varepsilon+f''(x_0)\varepsilon^2+\ldots\sim f(x_0)+f'(x_0)\varepsilon$$

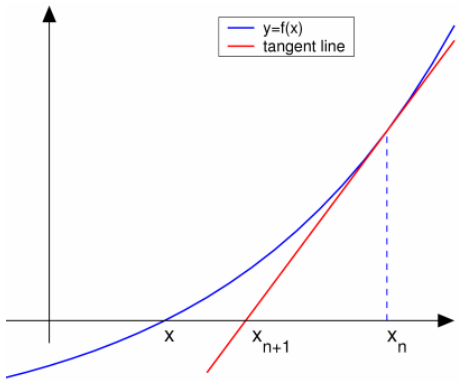
That is, if we want to find the root for $f(x)=0$ with an initial guess x_0. Then the correction term should be

$$f(x_0+\varepsilon)\sim f(x_0)+f'(x_0)\varepsilon=0 \longrightarrow \varepsilon=-\frac{f(x_0)}{f'(x_0)}$$

So we can solve it by this iteration $x_{n+1}=x_n-\frac{f(x_n)}{f'(x_n)}$

Reciprocal

$$x_{n+1}=x_n-\frac{f(x_n)}{f'(x_n)}$$



If r is the reciprocal of p, It means that r is the root for

$$f(x)=\frac{1}{x}-p$$

$$f'(x)=-\frac{1}{x^2}$$

Thus, if r_0 is the initial guess, the next one is

$$r=r_0-\frac{\frac{1}{r_0}-p}{-\frac{1}{r_0^2}}=2r_0-pr_0^2$$

Reciprocal

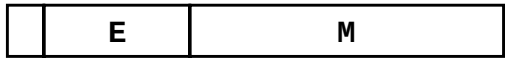
```
#define FP_ONE_BITS 0x3F800000
// r = 1/p  from NVidia's fastmath.cpp
#define FP_INV(r,p)                                \
{                                                  \
    int _i = 2 * FP_ONE_BITS - *(int *)&(p);       \
    r = *(float *)&_i;                             \
    r = r * (2.0f - (p) * r);                      \
}
```

The remaining question is how to pick up the initial guess.

$$r=r_0-\frac{\frac{1}{r_0}-p}{-\frac{1}{r_0^2}}=2r_0-pr_0^2$$

Reciprocal

```
#define FP_ONE_BITS 0x3F800000
// r = 1/p  from NVidia's fastmath.cpp
#define FP_INV(r,p)                                \
{                                                  \
    int _i = 2 * FP_ONE_BITS - *(int *)&(p);       \
```



$$\left(1+\frac{M}{2^{23}}\right)\times 2^{E-127}$$

$$\frac{1}{\left(1+\frac{M}{2^{23}}\right)\times 2^{E-127}}=\frac{1}{1+\frac{M}{2^{23}}}\times 2^{127-E}\approx\left(1-\frac{M}{2^{23}}\right)\times 2^{127-E}$$

$$=\left(2-\frac{2M}{2^{23}}\right)\times 2^{126-E}=\left(1+\left(1-\frac{2M}{2^{23}}\right)\right)\times 2^{126-E}\approx\left(1+\left(1-\frac{M}{2^{23}}\right)\right)\times 2^{126-E}$$

Inverse square root

- In graphics, we often have to normalize a vector

```
Vector &normalize()
  float invlen=1.0/sqrt(x*x + y*y + z*z);
  x *= invlen;
  y *= invlen;
  z *= invlen;
  return *this;
}
```

invSqrt

```
// used in QUAKE3
float InvSqrt (float x)
{
  float xhalf = 0.5f*x;
  int i = *(int*)&x;
  i = 0x5f3759df - (i >> 1);
  x = *(float*)&i;
  x = x*(1.5f - xhalf*x*x);
  return x;
}
```

invSqrt (experiments by littleshan)

```
void inv_sqrt_v1(float* begin, float* end, float* out)
{ /* naive method */
  for(; begin < end; ++begin, ++output)
  *out = 1.0f / sqrtf(*begin);
}
void inv_sqrt_v2(float* begin, float* end, float* out)
{
  float xhalf, x;
  int i;
  for(; begin < end; ++begin, ++out){
    xhalf = 0.5f * (*begin);
    i = *(int*)begin;
    i = 0x5f3759df - (i>>1);
    x = *(float*)&i;
    *out = x*(1.5f - xhalf*x*x);
  }
}
```

invSqrt

```
void inv_sqrt_v3(float* begin, float* end, float* out)
{ /* vectorized SSE */
  long size = end - begin;
  long padding = size % 16;
  size -= padding;

  // each time, we use simd to do 16 invsqrt
  // do the rest (padding) first
  for(; padding > 0; --padding, ++begin, ++output)
    *output = 1.0f / sqrt(*begin);
```

invSqrt

```
__asm {
        mov esi, [begin]
        mov edi, [output]
    loop_begin:
        cmp   esi, [end]
        ja    loop_end

        movups      xmm0, [esi   ]
        movups      xmm1, [esi+16]
        movups      xmm2, [esi+32]
        movups      xmm3, [esi+48]

        rsqrtps     xmm4, xmm0
        rsqrtps     xmm5, xmm1
        rsqrtps     xmm6, xmm2
        rsqrtps     xmm7, xmm3
```

invSqrt

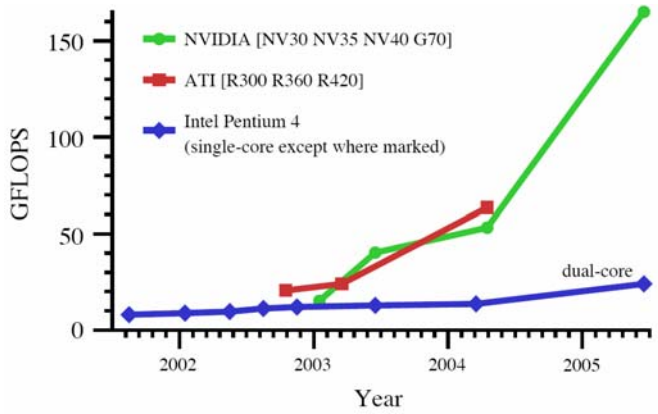
```
        movups      [edi   ], xmm4
        movups      [edi+16], xmm5
        movups      [edi+32], xmm6
        movups      [edi+48], xmm7

        add   esi, 64
        add   edi, 64

        jmp   loop_begin
    loop_end:
}
```

Experiments

```
method 1: naive sqrt()
CPU cycle used: 13444770
method 2: marvelous solution
CPU cycle used:  2806215
method 3: vectorized SSE
CPU cycle used:  1349355
```

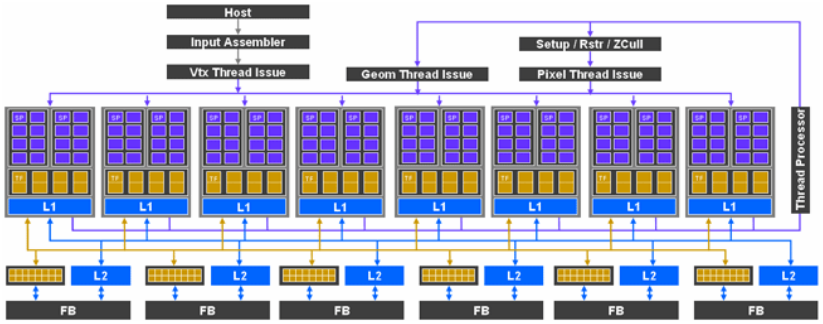
Other SIMD architectures

- Graphics Processing Unit (GPU): nVidia 7800, 24 pipelines (8 vector/16 fragment)



NVidia GeForce 8800, 2006

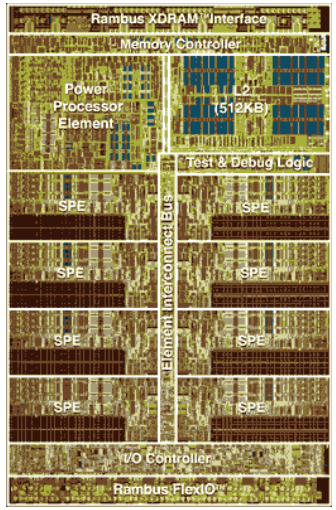
- Each GeForce 8800 GPU stream processor is a fully generalized, fully decoupled, scalar, processor that supports IEEE 754 floating point precision.
- Up to 128 stream processors

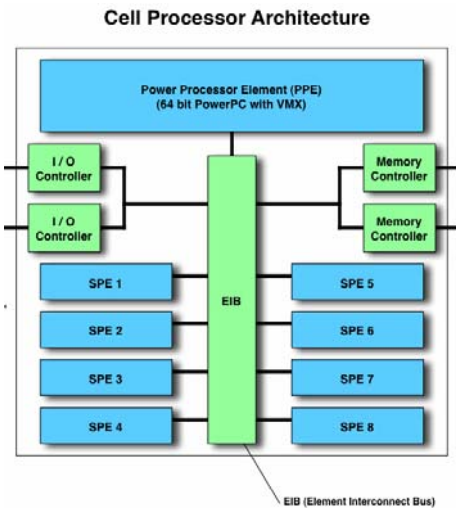


Cell processor

- Cell Processor (IBM/Toshiba/Sony): 1 PPE (Power Processing Unit) +8 SPEs (Synergistic Processing Unit)
- An SPE is a RISC processor with 128-bit SIMD for single/double precision instructions, 128 128-bit registers, 256K local cache
- used in PS3.

Cell processor





Announcements

- Voting
- TA evaluation on 1/8
- Final project due date? 1/24 or 1/31?