

Intel SIMD architecture

Computer Organization and Assembly Languages
Yung-Yu Chuang
2007/1/7

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

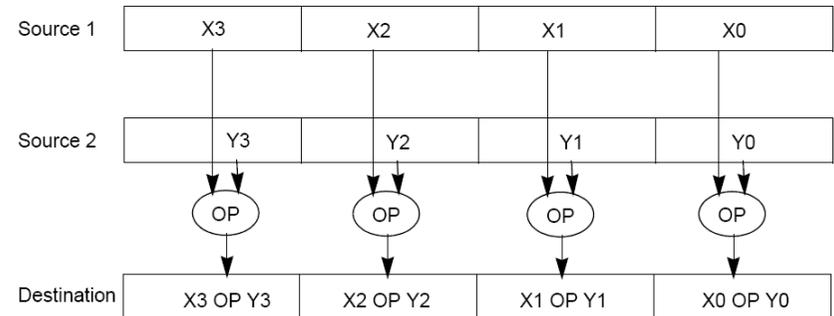


- Increasing clock rate is not fast enough for boosting performance
- Architecture improvements (such as pipeline/cache/SIMD) are more significant
- Intel analyzed multimedia applications and found they share the following characteristics:
 - Small native data types (8-bit pixel, 16-bit audio)
 - Recurring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

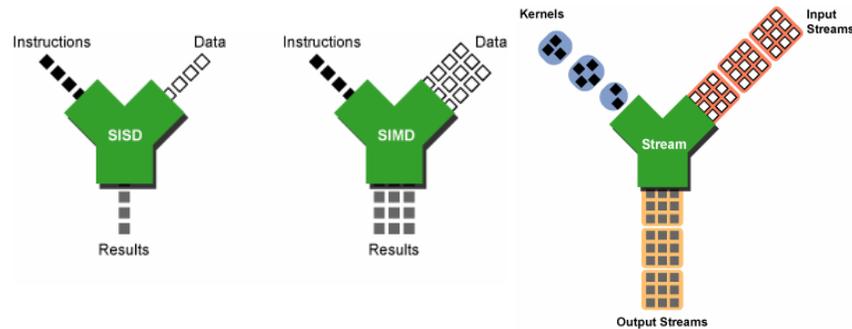
SIMD



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



SISD/SIMD/Streaming



5

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.

6

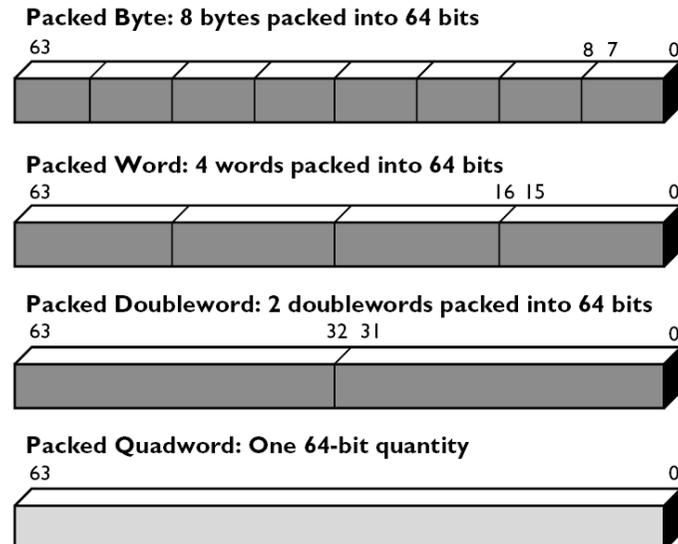
MMX



- 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

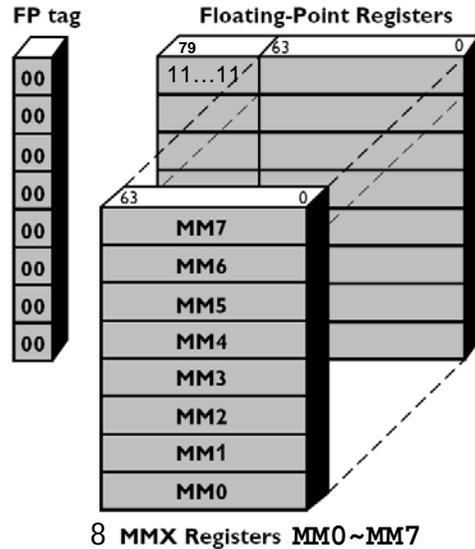
7

MMX data types



8

MMX integration into IA



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

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.

8 MMX Registers MM0~MM7

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. Big overhead to switch.

Compatibility



- Although Intel defends 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

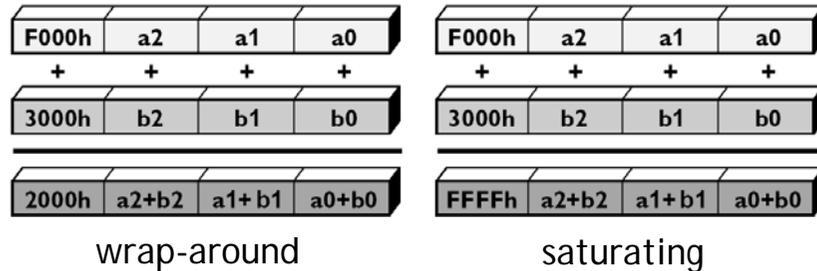


- 57 MMX instructions are defined to perform the parallel operations on multiple data elements packed into 64-bit data types.
- These include **add**, **subtract**, **multiply**, **compare**, and **shift**, **data conversion**, **64-bit data move**, **64-bit logical operation** and **multiply-add** for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

Saturation arithmetic

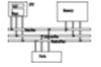


- Useful in graphics applications.
- When an operation overflows or underflows, the result becomes the largest or smallest possible representable number.
- Two types: signed and unsigned saturation



13

MMX instructions



Category		Wraparound	Signed Saturation	Unsigned Saturation
Arithmetic	Addition	PADDB, PADDW, PADDD	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	PCMPGTB, PCMPGTPW, PCMPGTPD		
Conversion	Pack		PACKSSWB, PACKSSDW	PACKUSWB
Unpack	Unpack High	PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ		
	Unpack Low	PUNPCKLBW, PUNPCKLWD, PUNPCKLDQ		

14

MMX instructions

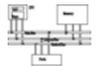


		Packed	Full Quadword
Logical	And And Not Or Exclusive OR		PAND PANDN POR PXOR
Shift	Shift Left Logical Shift Right Logical Shift Right Arithmetic	PSLLW, PSLLD PSRLW, PSRLD PSRAW, PSRAD	PSLLQ PSRLQ
Data Transfer	Register to Register Load from Memory Store to Memory	Doubleword Transfers Quadword Transfers	
		MOVD MOVD MOVD	MOVQ MOVQ MOVQ
Empty MMX State		EMMS	

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

15

Arithmetic



- **PADDB/PADDW/PADDD**: add two packed numbers, no EFLAGS 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.

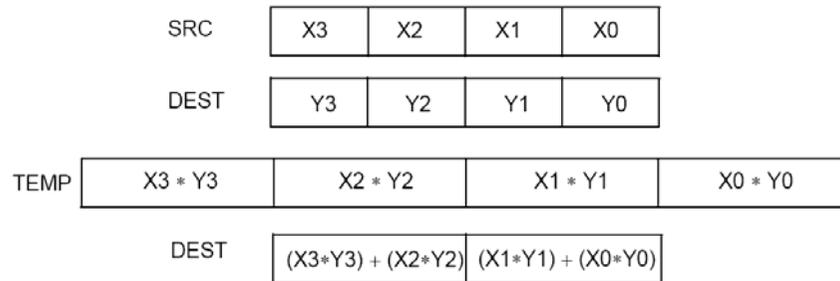
16

Arithmetic



• PMADDWD

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



Detect MMX/SSE



```

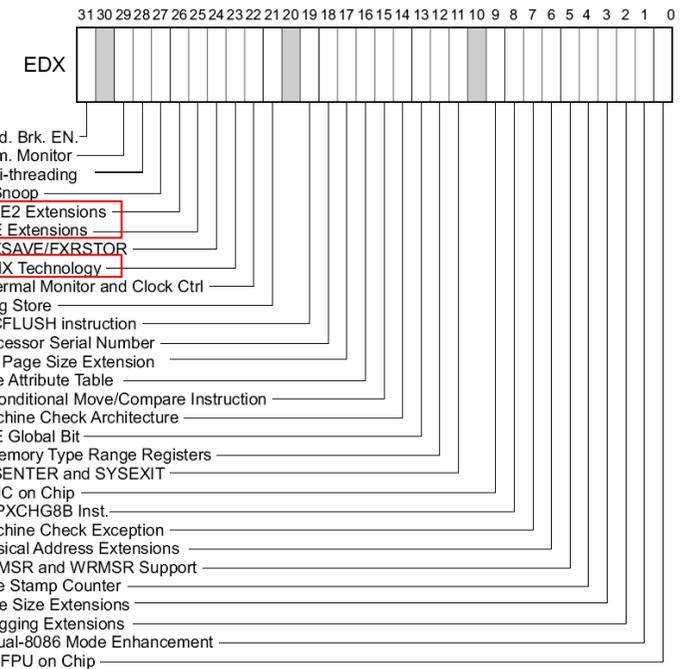
mov    eax, 1 ; request version info
cpuid  ; supported since Pentium
test   edx, 00800000h ;bit 23
                ; 02000000h (bit 25) SSE
                ; 04000000h (bit 26) SSE2
jnz    HasMMX
    
```

cpuid



Initial EAX Value	Information Provided about the Processor	
0H	<i>Basic CPUID Information</i>	
	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

⋮



- PBE—Pend. Brk. EN.
- TM—Therm. Monitor
- HTT—Multi-threading
- SS—Self Snoop
- SSE2—SSE2 Extensions
- SSE—SSE Extensions
- FXSR—FXSAVE/FXRSTOR
- MMX—MMX Technology
- ACPI—Thermal Monitor and Clock Ctrl
- DS—Debug Store
- CLFSH—CFLUSH instruction
- PSN—Processor Serial Number
- PSE-36—Page Size Extension
- PAT—Page Attribute Table
- CMOV—Conditional Move/Compare Instruction
- MCA—Machine Check Architecture
- PGE—PTE Global Bit
- MTRR—Memory Type Range Registers
- SEP—SYSENTER and SYSEXIT
- APIC—APIC on Chip
- CX8—CMPXCHG8B Inst.
- MCE—Machine Check Exception
- PAE—Physical Address Extensions
- MSR—RDMSR and WRMSR Support
- TSC—Time Stamp Counter
- PSE—Page Size Extensions
- DE—Debugging Extensions
- VME—Virtual-8086 Mode Enhancement
- FPU—x87 FPU on Chip

Example: add a constant to a vector



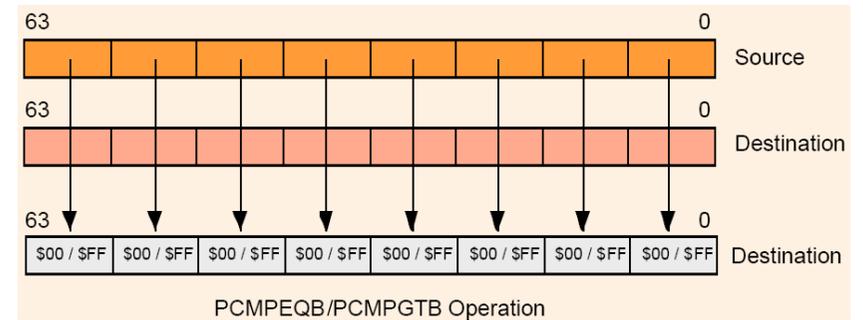
```
char d[]={5, 5, 5, 5, 5, 5, 5, 5};
char clr[]={65,66,68,...,87,88}; // 24 bytes
__asm{
    movq mm1, d
    mov cx, 3
    mov esi, 0
L1: movq mm0, clr[esi]
    paddb mm0, mm1
    movq clr[esi], mm0
    add esi, 8
    loop L1
    emms
}
```

21

Comparison



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



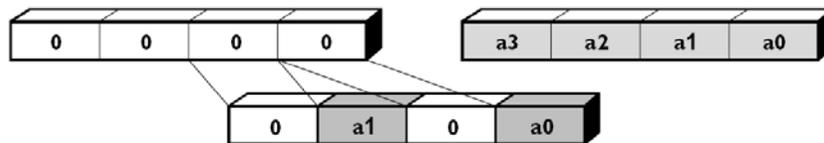
22

Change data types



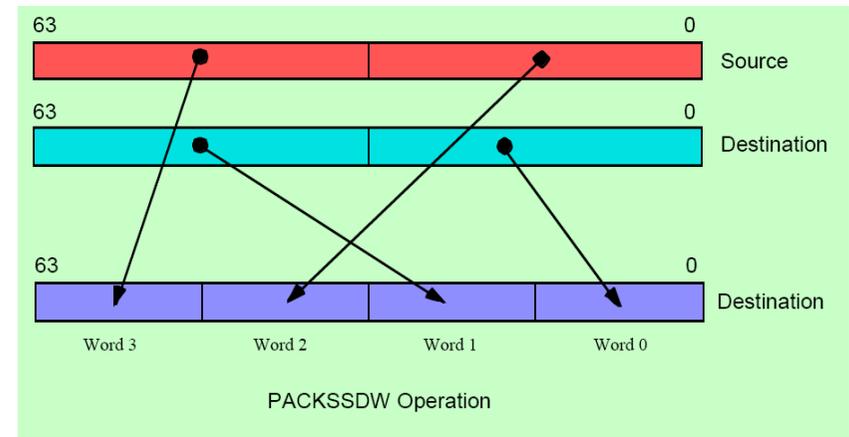
- Pack: converts a larger data type to the next smaller data type.
- Unpack: takes two operands and interleaves them. It can be used for expand data type for immediate calculation.

Unpack low-order words into doublewords



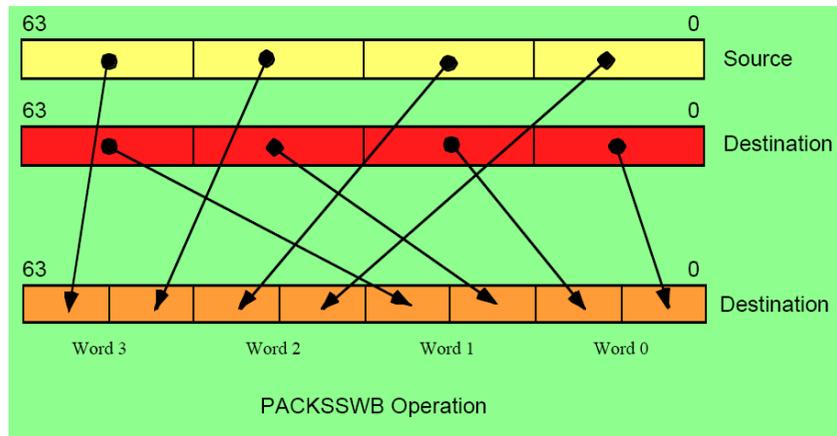
23

Pack with signed saturation



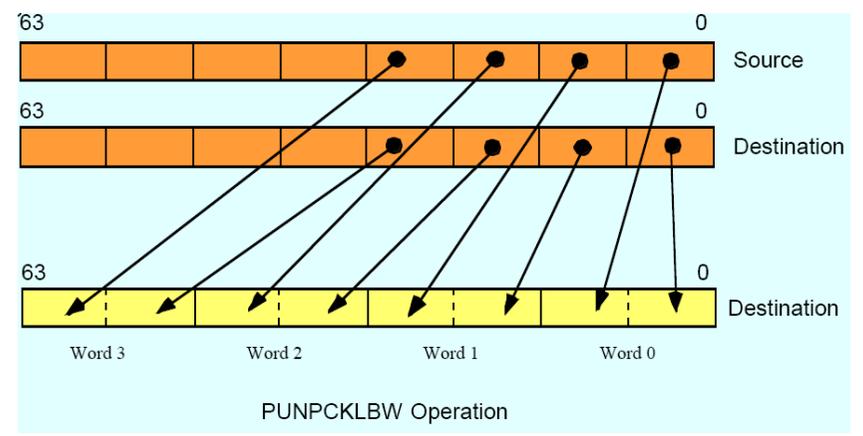
24

Pack with signed saturation



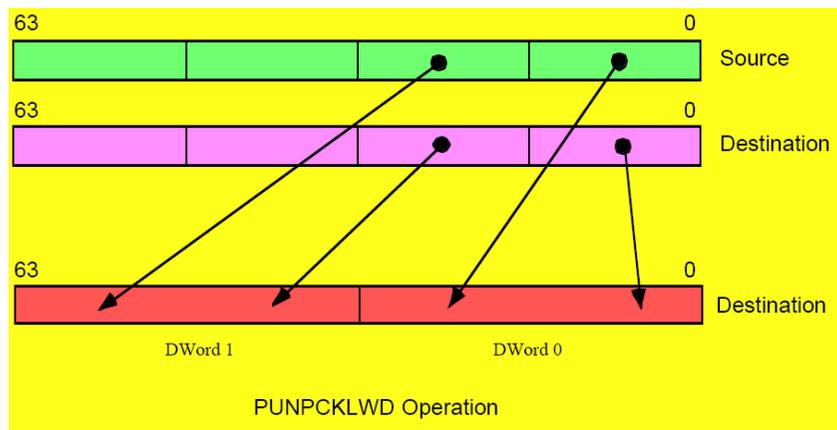
25

Unpack low portion



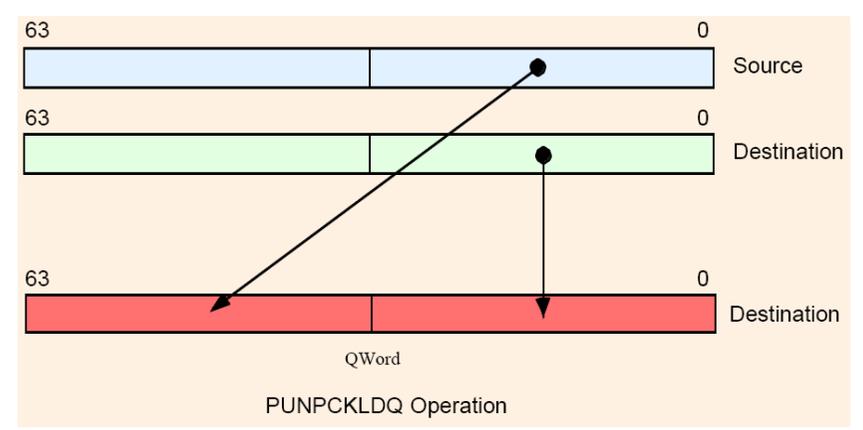
26

Unpack low portion



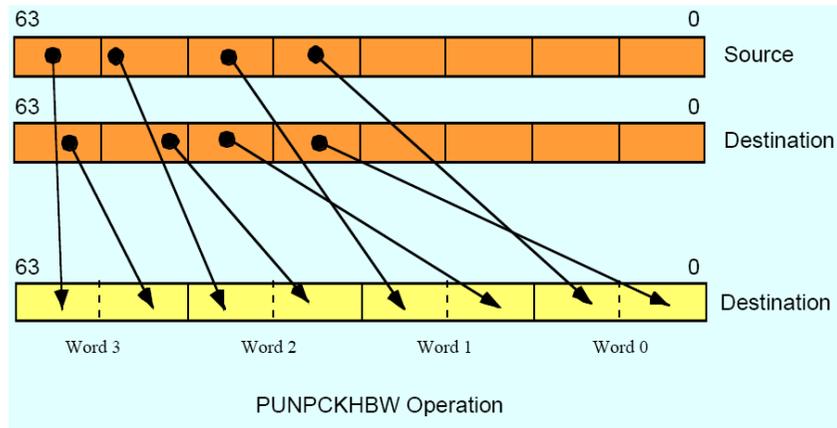
27

Unpack low portion



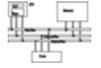
28

Unpack high portion



29

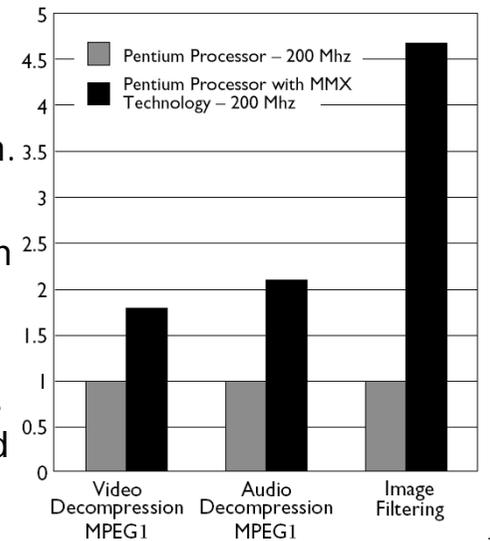
Performance boost (data from 1996)



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



30

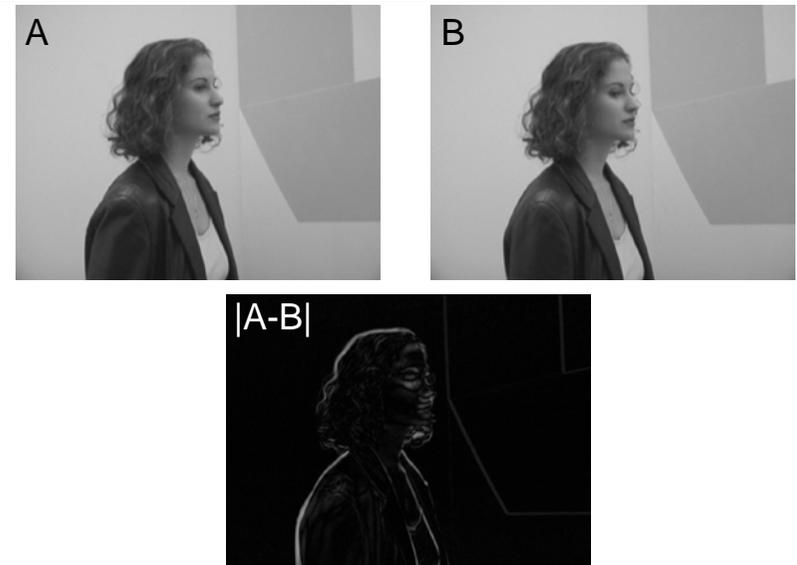
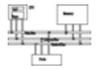
Keys to SIMD programming



- Efficient data layout
- Elimination of branches

31

Application: frame difference



32

Application: frame difference



(A-B) or (B-A)



33

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|
```

34

Example: image fade-in-fade-out



A

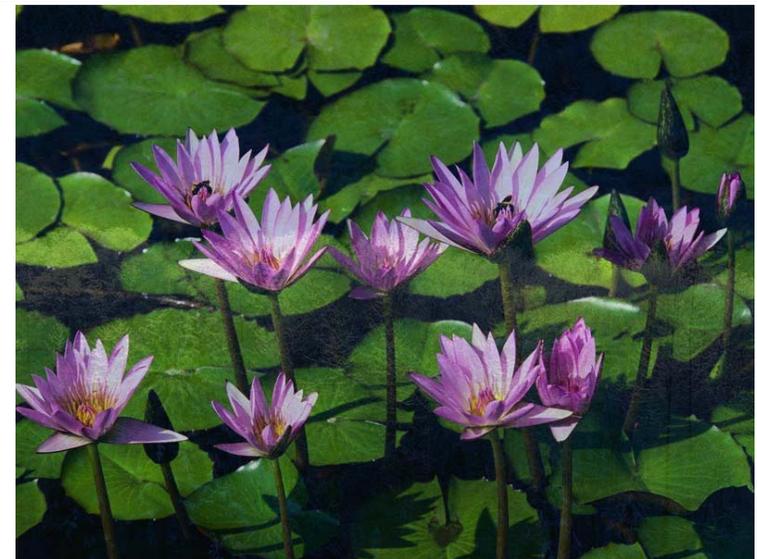


B

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

35

$\alpha = 0.75$



36

$\alpha = 0.5$



37

$\alpha = 0.25$

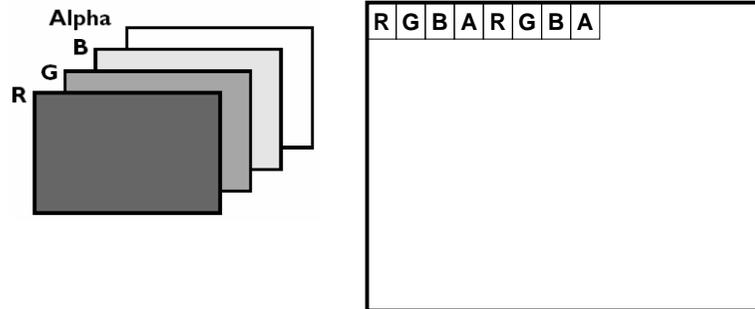


38

Example: image fade-in-fade-out

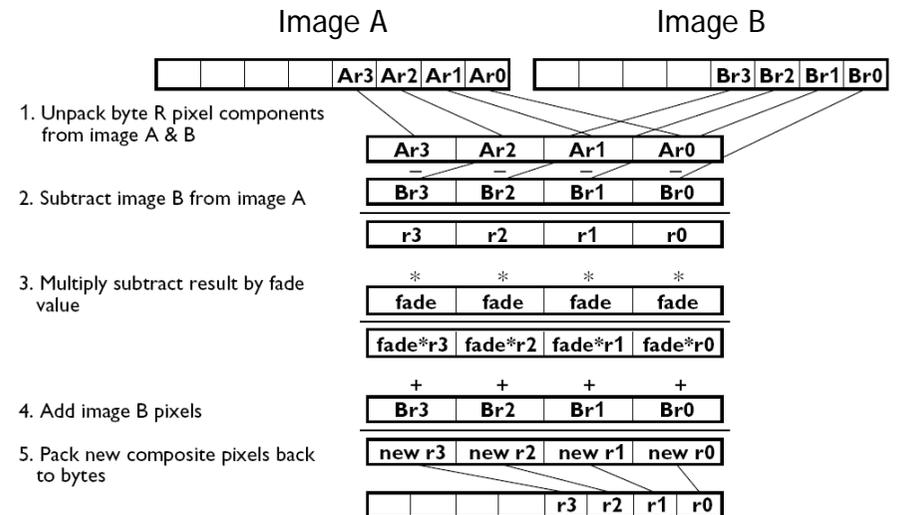


- Two formats: planar and chunky
- In Chunky format, 16 bits of 64 bits are wasted
- So, we use planar in the following example



39

Example: image fade-in-fade-out



40

Example: image fade-in-fade-out



```

MOVQ      mm0, alpha//4 16-b zero-padding  $\alpha$ 
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
    
```

41

Data-independent computation

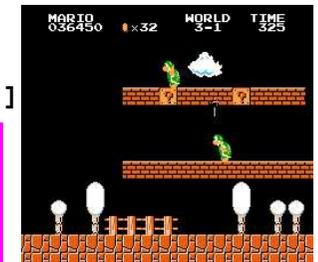
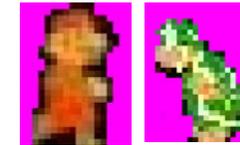


- Each operation can execute without needing to know the results of a previous operation.

- Example, sprite overlay

```

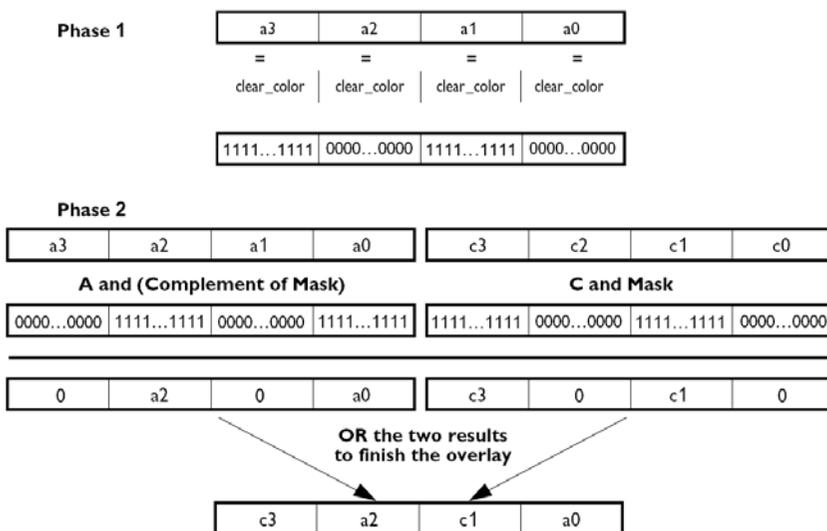
for i=1 to sprite_Size
  if sprite[i]=clr
    then out_color[i]=bg[i]
  else out_color[i]=sprite[i]
    
```



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

42

Application: sprite overlay



43

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
    
```

44

Application: matrix transport



Phase 1



Phase 2



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, mm1 = [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]
    
```

45

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
    
```

46

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
    
```

47

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
    
```

48

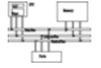
How to use assembly in projects



- Write the whole project in assembly
- Link with high-level languages
- Inline assembly
- Intrinsics

49

Link ASM and HLL programs



- Assembly is rarely used to develop the entire program.
- Use high-level language for overall project development
 - Relieves programmer from low-level details
- Use assembly language code
 - Speed up critical sections of code
 - Access nonstandard hardware devices
 - Write platform-specific code
 - Extend the HLL's capabilities

50

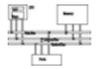
General conventions



- Considerations when calling assembly language procedures from high-level languages:
 - Both must use the same naming convention (rules regarding the naming of variables and procedures)
 - Both must use the same memory model, with compatible segment names
 - Both must use the same calling convention

51

Inline assembly code



- Assembly language source code that is inserted directly into a HLL program.
- Compilers such as Microsoft Visual C++ and Borland C++ have compiler-specific directives that identify inline ASM code.
- Efficient inline code executes quickly because CALL and RET instructions are not required.
- Simple to code because there are no external names, memory models, or naming conventions involved.
- Decidedly not portable because it is written for a single platform.

52

__asm directive in Microsoft Visual C++



- Can be placed at the beginning of a single statement
- Or, It can mark the beginning of a block of assembly language statements
- Syntax:

```
__asm statement

__asm {
    statement-1
    statement-2
    ...
    statement-n
}
```

53

Intrinsics



- An *intrinsic* is a function known by the compiler that directly maps to a sequence of one or more assembly language instructions.
- 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.
- Intrinsic functions are inherently more efficient than called functions because no calling linkage is required. But, not necessarily as efficient as assembly.
- `_mm_<opcode>_<suffix>` [ps: packed single-precision](#)
[ss: scalar single-precision](#)

54

Intrinsics



```
#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 ) );
```

55

SSE



- Adds eight 128-bit registers
- Allows SIMD operations on packed single-precision floating-point numbers
- Most SSE instructions require 16-aligned addresses

56

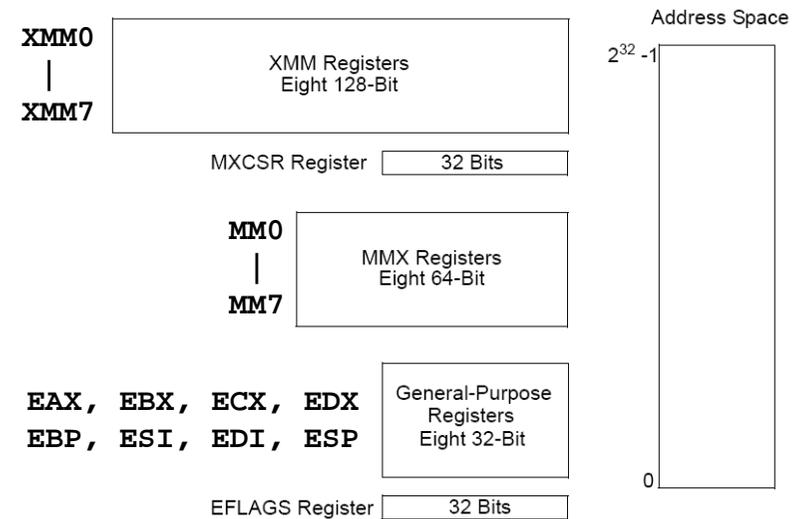
SSE features



- 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

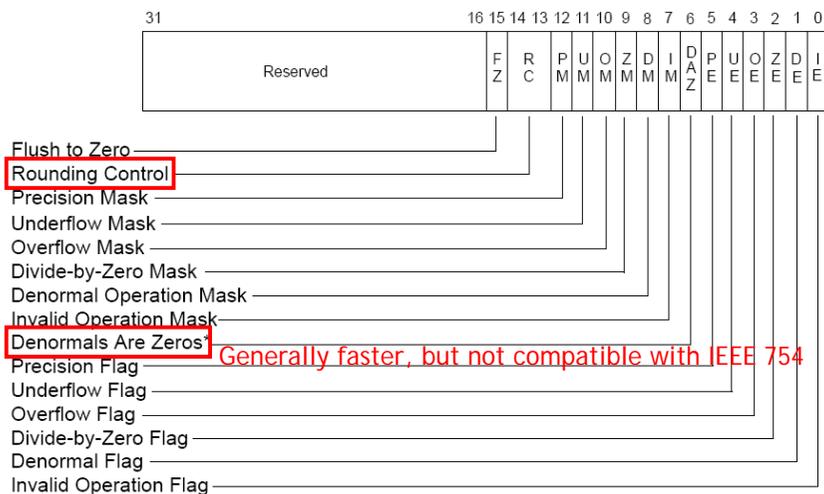
57

SSE programming environment



58

MXCSR control and status register



59

Exception

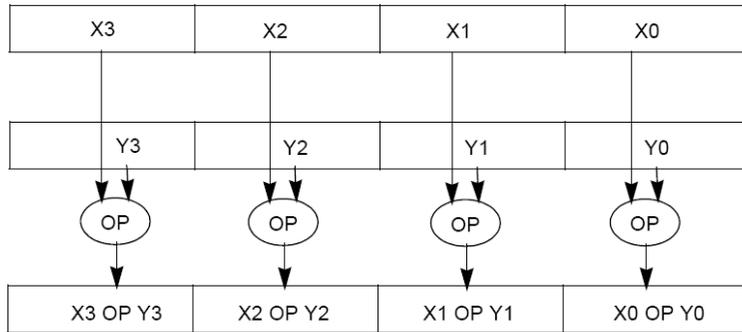


```

_MM_ALIGN16 float test1[4] = { 0, 0, 0, 1 };
_MM_ALIGN16 float test2[4] = { 1, 2, 3, 0 };
_MM_ALIGN16 float out[4];
_MM_SET_EXCEPTION_MASK(0); //enable exception
try {
    Without this, result is 1.#INF
    __m128 a = _mm_load_ps(test1);
    __m128 b = _mm_load_ps(test2);
    a = _mm_div_ps(a, b);
    _mm_store_ps(out, a);
}
except(EXCEPTION_EXECUTE_HANDLER) {
    if(_mm_getcsr() & _MM_EXCEPT_DIV_ZERO)
        cout << "Divide by zero" << endl;
    return;
}
    
```

60

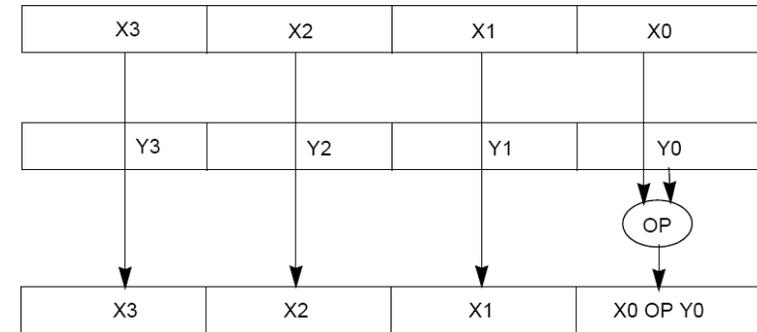
SSE packed FP operation



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

61

SSE scalar FP operation



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

62

SSE2



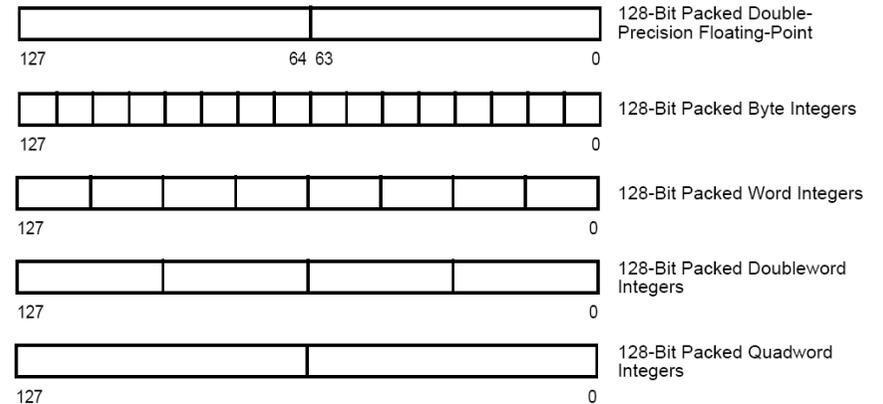
- Provides ability to perform SIMD operations on double-precision FP, allowing advanced graphics such as ray tracing
- Provides greater throughput by operating on 128-bit packed integers, useful for RSA and RC5

63

SSE2 features



- Add data types and instructions for them



- Programming environment unchanged

64

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

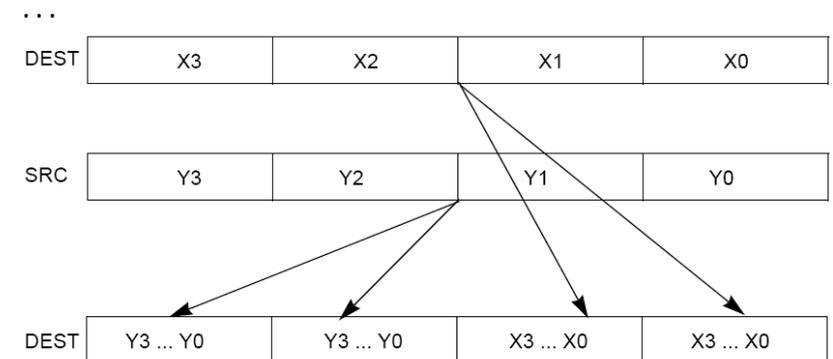
65

SSE Shuffle (SHUFPS)



SHUFPS xmm1, xmm2, imm8

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



66

SSE Shuffle (SHUFPS)



<p>CASE (SELECT[1:0]) OF</p> <p>0: DEST[31:0] ← DEST[31:0];</p> <p>1: DEST[31:0] ← DEST[63:32];</p> <p>2: DEST[31:0] ← DEST[95:64];</p> <p>3: DEST[31:0] ← DEST[127:96];</p> <p>ESAC;</p> <p>CASE (SELECT[3:2]) OF</p> <p>0: DEST[63:32] ← DEST[31:0];</p> <p>1: DEST[63:32] ← DEST[63:32];</p> <p>2: DEST[63:32] ← DEST[95:64];</p> <p>3: DEST[63:32] ← DEST[127:96];</p> <p>ESAC;</p>	<p>CASE (SELECT[5:4]) OF</p> <p>0: DEST[95:64] ← SRC[31:0];</p> <p>1: DEST[95:64] ← SRC[63:32];</p> <p>2: DEST[95:64] ← SRC[95:64];</p> <p>3: DEST[95:64] ← SRC[127:96];</p> <p>ESAC;</p> <p>CASE (SELECT[7:6]) OF</p> <p>0: DEST[127:96] ← SRC[31:0];</p> <p>1: DEST[127:96] ← SRC[63:32];</p> <p>2: DEST[127:96] ← SRC[95:64];</p> <p>3: DEST[127:96] ← SRC[127:96];</p> <p>ESAC;</p>
---	---

67

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] ) );
}

```

68

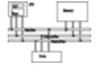
Example (cross product)



```
/* 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 );
}
```

69

Example: dot product



- Given a set of vectors $\{v_1, v_2, \dots, v_n\} = \{(x_1, y_1, z_1), (x_2, y_2, z_2), \dots, (x_n, y_n, z_n)\}$ 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;`

70

Example: dot product (AoS)

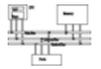


```
movaps xmm0, v ; xmm0 = DC, x0, y0, z0
movaps xmm1, vc ; xmm1 = DC, xc, yc, zc
mulps xmm0, xmm1 ; xmm0 = DC, x0*xc, y0*yc, z0*zc
movhyps xmm1, xmm0 ; xmm1 = DC, DC, DC, x0*xc
addps xmm1, xmm0 ; xmm1 = DC, DC, DC,
                    ; x0*xc+z0*zc
movaps xmm2, xmm0
shufps xmm2, xmm2, 55h ; xmm2 =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

```
movhyps:DEST[63..0] := SRC[127..64]
```

71

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*yc, 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)...
```

72

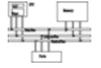
SSE examples



```
float input1[4]={ 1.2f, 3.5f, 1.7f, 2.8f };
float input2[4]={ -0.7f, 2.6f, 3.3f, -0.8f };
float output[4];
For (int i = 0; i < 4; i++) {
    output[i] = input1[i] + input2[i];
}
```

73

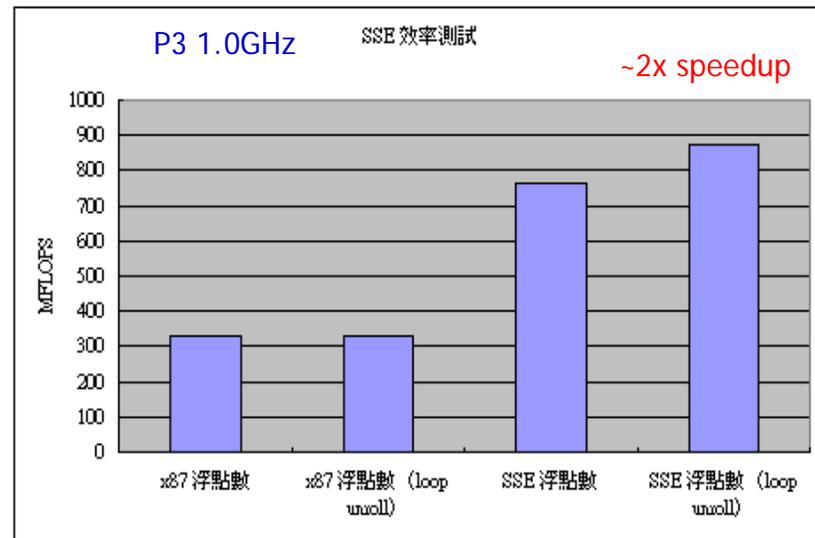
SSE examples



```
_MM_ALIGN16 float input1[4]
    = { 1.2f, 3.5f, 1.7f, 2.8f };
_MM_ALIGN16 float input2[4]
    = { -0.7f, 2.6f, 3.3f, -0.8f };
_MM_ALIGN16 float output[4];
_m128 a = _mm_load_ps(input1);
_m128 b = _mm_load_ps(input2);
_m128 t = _mm_add_ps(a, b);
_mm_store_ps(output, t);
```

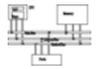
74

SSE examples (1,024 FP additions)



.5

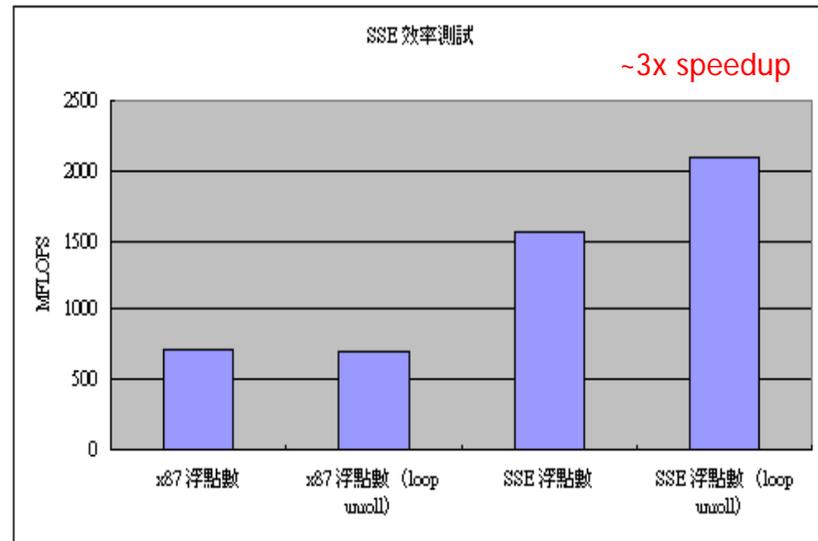
Inner product



```
_m128 x1 = _mm_load_ps(vec1_x);
_m128 y1 = _mm_load_ps(vec1_y);
_m128 z1 = _mm_load_ps(vec1_z);
_m128 x2 = _mm_load_ps(vec2_x);
_m128 y2 = _mm_load_ps(vec2_y);
_m128 z2 = _mm_load_ps(vec2_z);
_m128 t1 = _mm_mul_ps(x1, x2);
_m128 t2 = _mm_mul_ps(y1, y2);
t1 = _mm_add_ps(t1, t2);
t2 = _mm_mul_ps(z1, z2);
t1 = _mm_add_ps(t1, t2);
_mm_store_ps(output, t1);
```

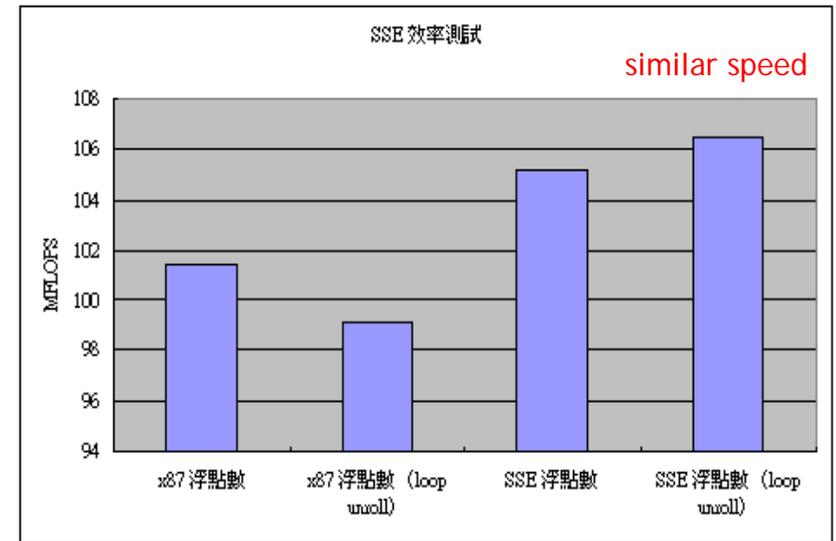
76

Inner product (1,024 3D vectors)



.7

Inner product (102,400 3D vectors)



.8

Cache control



- **prefetch** (`_mm_prefetch`): a hint for CPU to load operands for the next instructions so that data loading can be executed in parallel with computation.
- **Movntps** (`_mm_stream_ps`): ask CPU not to write data into cache, but to the memory directly.

79

Cache control



```
_mm128 x1 = _mm_load_ps(vec1_x);  
_mm128 y1 = _mm_load_ps(vec1_y);  
_mm128 z1 = _mm_load_ps(vec1_z);  
_mm128 x2 = _mm_load_ps(vec2_x);  
_mm128 y2 = _mm_load_ps(vec2_y);  
_mm128 z2 = _mm_load_ps(vec2_z);  
_mm_prefetch((const char*)(vec1_x + next),  
_MM_HINT_NTA);  
_mm_prefetch((const char*)(vec1_y + next),  
_MM_HINT_NTA);  
_mm_prefetch((const char*)(vec1_z + next),  
_MM_HINT_NTA);
```

80

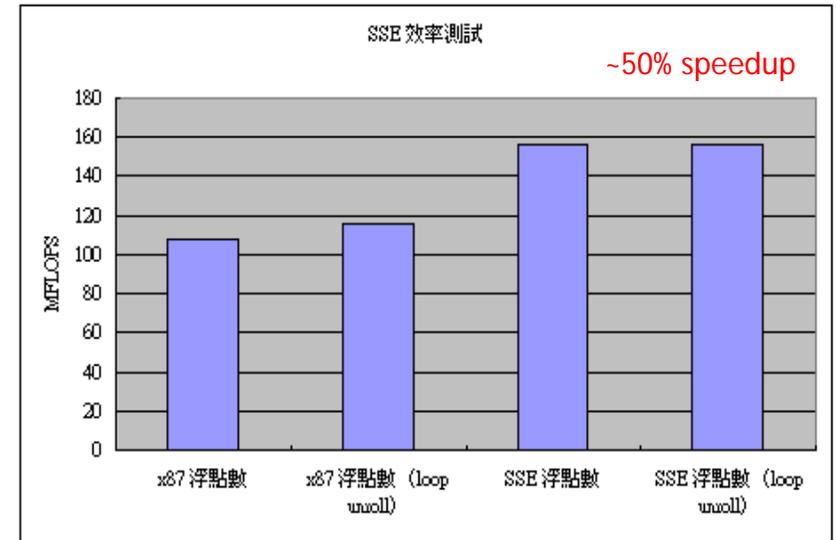
Cache control



```
__mm_prefetch((const char*)(vec2_x + next),
  __MM_HINT_NTA);
__mm_prefetch((const char*)(vec2_y + next),
  __MM_HINT_NTA);
__mm_prefetch((const char*)(vec2_z + next),
  __MM_HINT_NTA);
__m128 t1 = __mm_mul_ps(x1, x2);
__m128 t2 = __mm_mul_ps(y1, y2);
t1 = __mm_add_ps(t1, t2);
t2 = __mm_mul_ps(z1, z2);
t1 = __mm_add_ps(t1, t2);
__mm_stream_ps(output, t1);
```

81

Cache control



82

Exponential



```
int i;
float result = coeff[8] * x;
for(i = 7; i >= 2; i--) {
    result += coeff[i];
    result *= x;
}
return (result + 1) * x + 1;
```

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

83

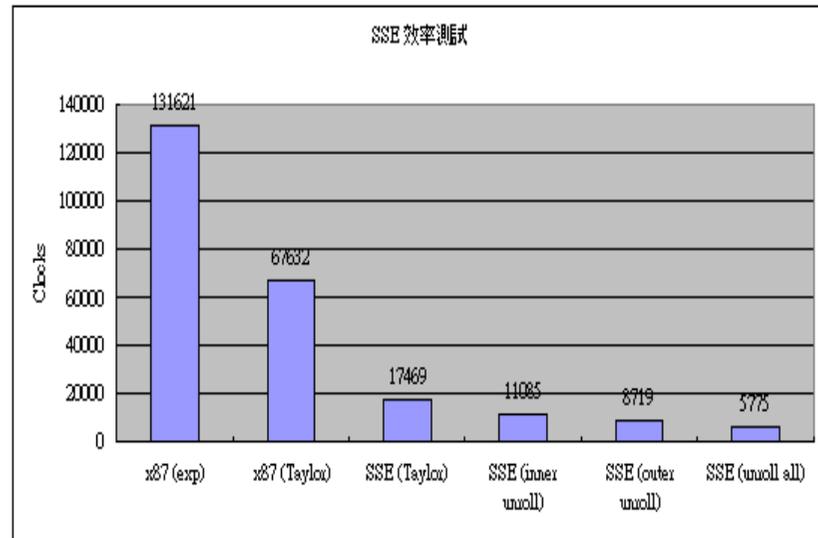
Exponential



```
int i;
__m128 X = __mm_load_ps(data);
__m128 result = __mm_mul_ps(coeff_sse[8], X);
for(i = 7; i >= 2; i--) {
    result = __mm_add_ps(result, coeff_sse[i]);
    result = __mm_mul_ps(result, X);
}
result = __mm_add_ps(result, sse_one);
result = __mm_mul_ps(result, X);
result = __mm_add_ps(result, sse_one);
__mm_store_ps(out, result);
```

84

Exponential (1,024 times)

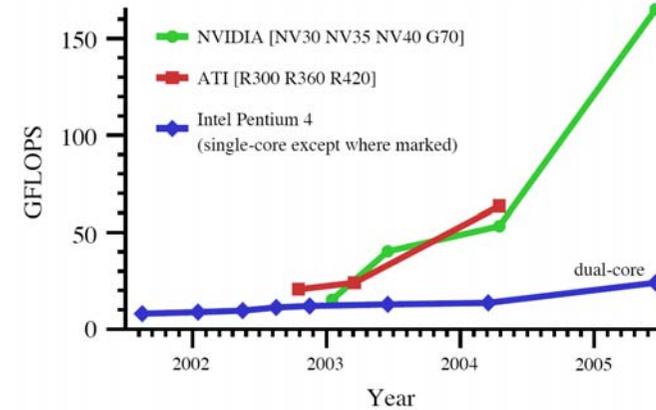


J5

Other SIMD architectures



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

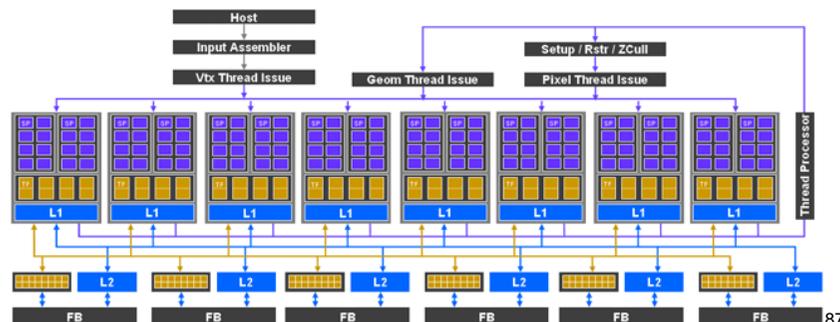


86

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



87

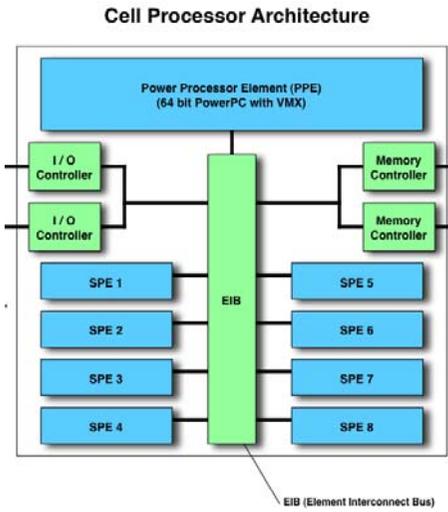
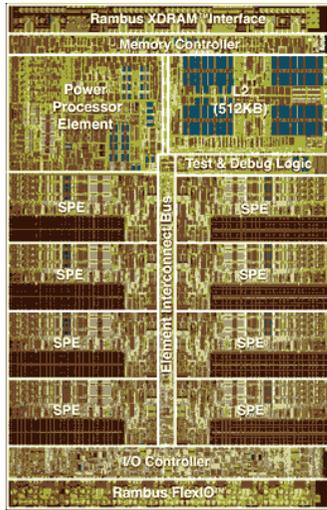
Cell processor



- 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.

88

Cell processor



89

References



- *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*
- http://www.csie.ntu.edu.tw/~r89004/hive/sse/page_1.html

90