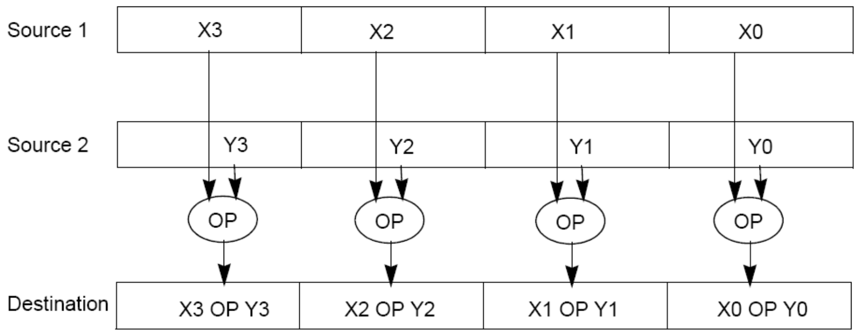
Intel SIMD extensions

Performance boost

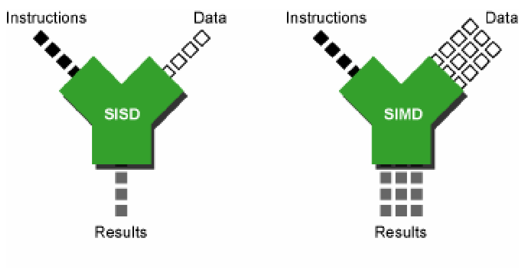
- Architecture improvements (such as pipeline/cache/SIMD) are more significant
- Intel analyzed multimedia applications and found they share the following characteristics:
  - Small native data types (8-bit pixel, 16-bit audio)
  - Recurring operations
  - Inherent parallelism

SIMD

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



SISD/SIMD



Intel SIMD development

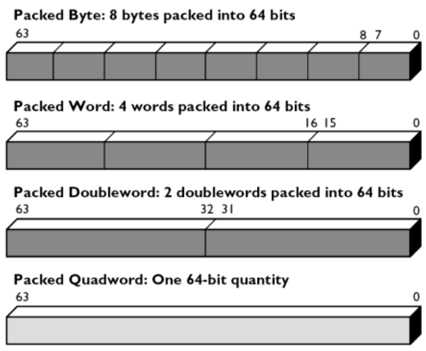
- MMX (Multimedia Extension) was introduced in 1996 (Pentium with MMX and Pentium II).
- SSE (Streaming SIMD Extension) was introduced with Pentium III.
- SSE2 was introduced with Pentium 4.
- SSE3 was introduced with Pentium 4 supporting hyper-threading technology. SSE3 adds 13 more instructions.
- Advanced Vector Extensions (2010)

MMX

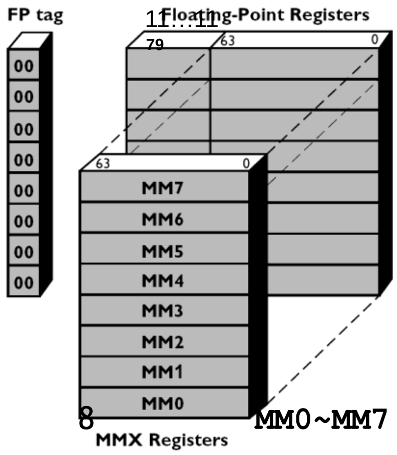
- After analyzing a lot of existing applications such as graphics, MPEG, music, speech recognition, game, image processing, they found that many multimedia algorithms execute the same instructions on many pieces of data in a large data set.
- Typical elements are small, 8 bits for pixels, 16 bits for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

MMX data types

Each of the MMn registers is a 64-bit integer. However, one of the main concepts of the MMX instruction set is the concept of packed data types, which means instead of using the whole register for a single 64-bit integer (quadword), two 32-bit integers (doubleword), four 16-bit integers (word) or eight 8-bit integers (byte) may be used.


Packed Byte: 8 bytes packed into 64 bits
63 8 7 0
Packed Word: 4 words packed into 64 bits
63 16 15 0
Packed Doubleword: 2 doublewords packed into 64 bits
63 32 31 0
Packed Quadword: One 64-bit quantity
63 0

MMX integration into IA



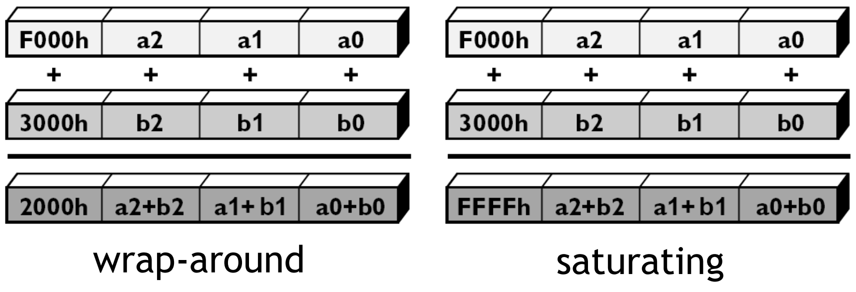
- To simplify the design and to avoid modifying the operating system to preserve additional state through context switches, MMX re-uses the existing eight IA-32 FPU registers.
- This made it difficult to work with floating point and SIMD data at the same time.
- To maximize performance, programmers must use the processor exclusively in one mode or the other

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



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

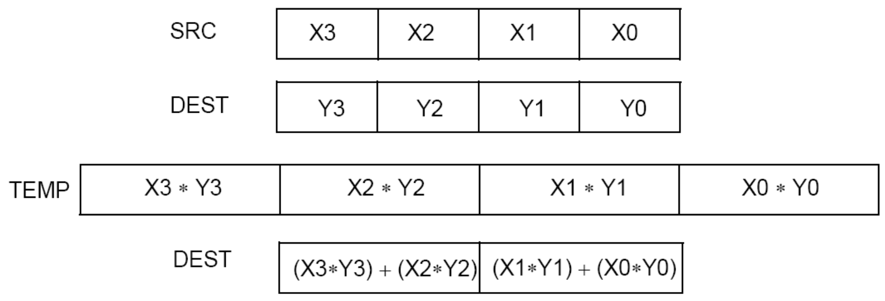
- **PADDB/PADDW/PADDD**: add two packed numbers
- Multiplication: two steps
- **PMULLW**: multiplies four words and stores the four lo words of the four double word results
- **PMULHW/PMULHUW**: multiplies four words and stores the four hi words of the four double word results. **PMULHUW** for unsigned.

13

Arithmetic

- **PMADDWD mmi, mmj**

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]);



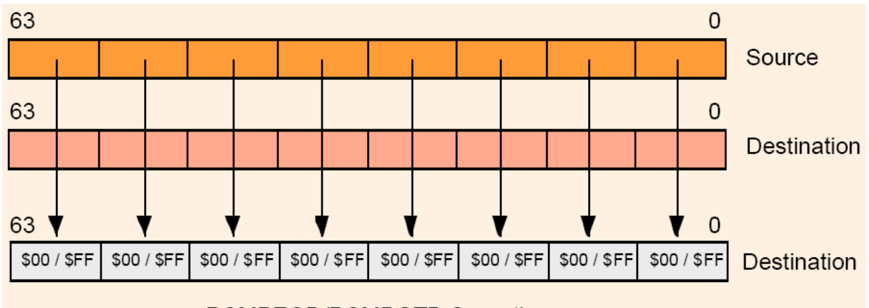
Example: add a constant to a vector

```
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
}
```

15

Comparison

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



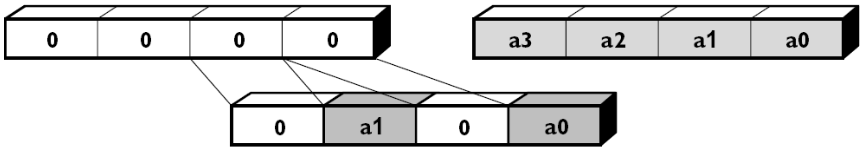
PCMPEQB/PCMPGTB Operation

16

Change data types

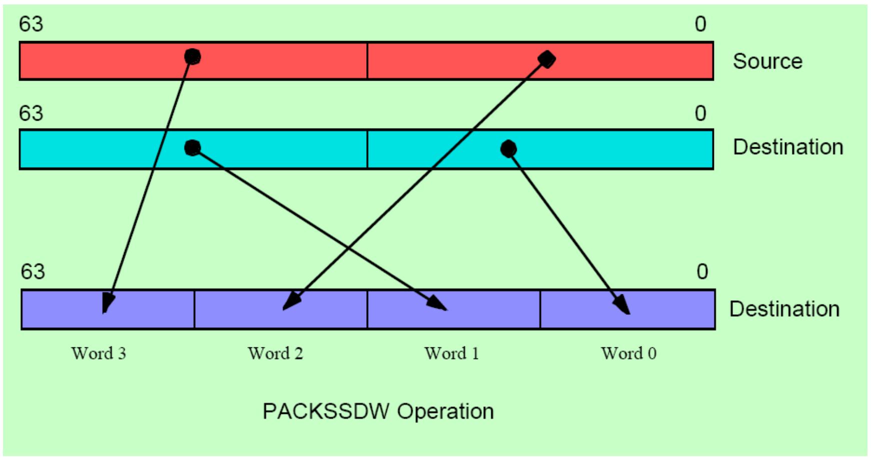
- Pack: converts a larger data type to the next smaller data type.
- Unpack: takes two operands and interleave them. It can be used for expand data type for immediate calculation.

Unpack low-order words into doublewords



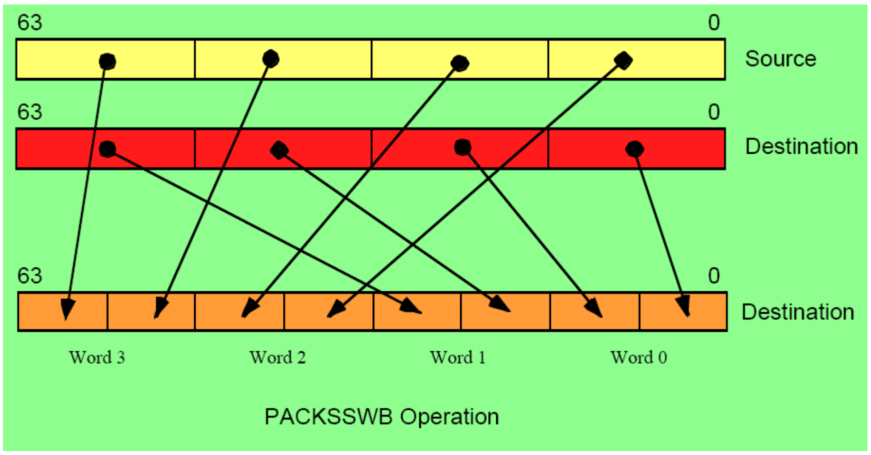
17

Pack with signed saturation



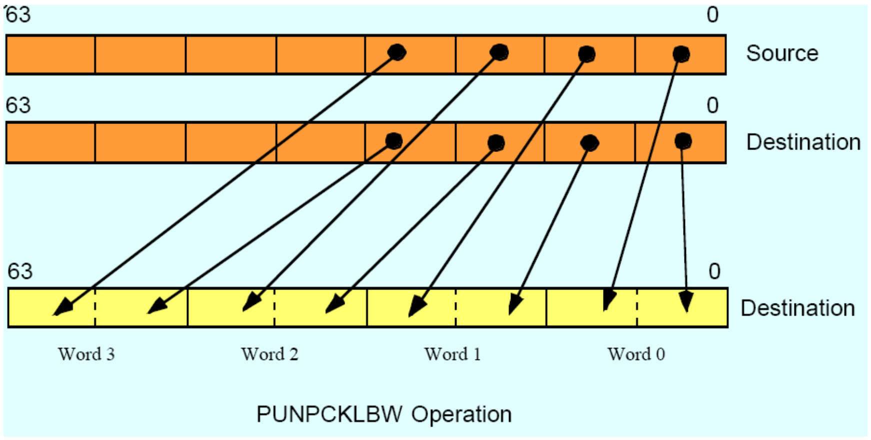
PACKSSDW mm_d, mm_s

Pack with signed saturation



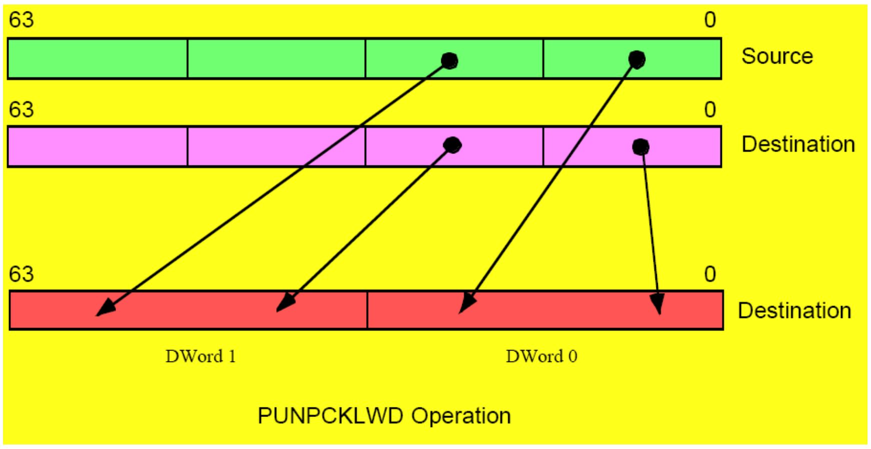
PACKSSWB mm$_d$, mm$_s$

Unpack low portion



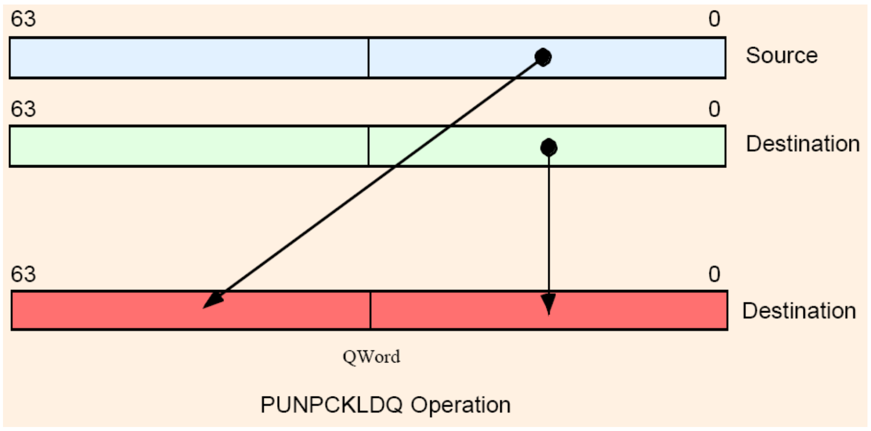
20

Unpack low portion



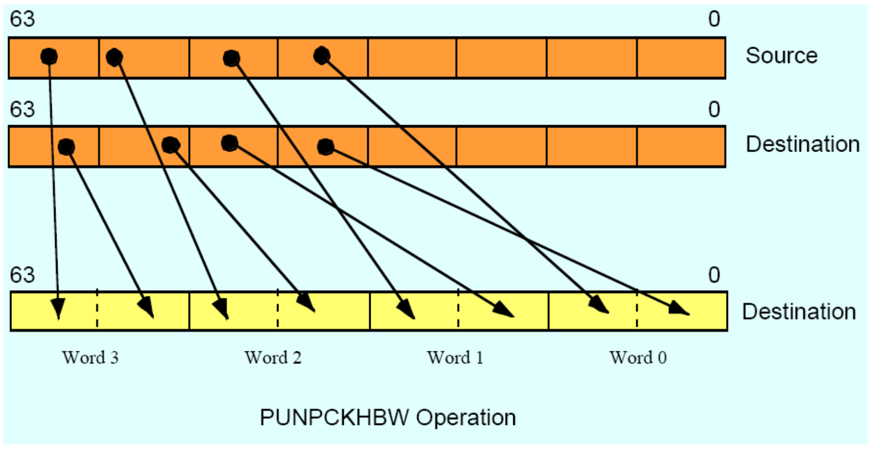
21

Unpack low portion



22

Unpack high portion

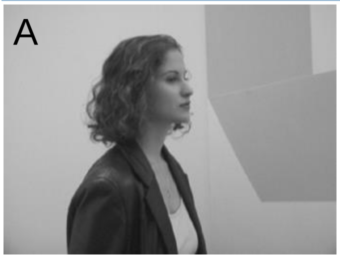


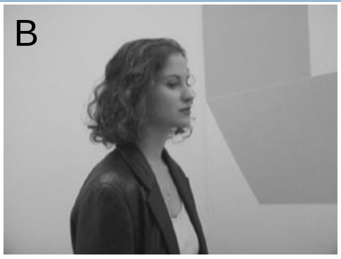
23

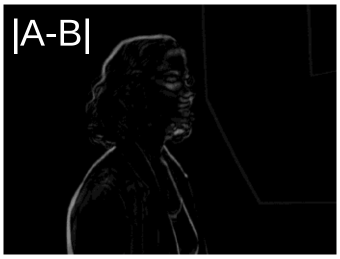
Keys to SIMD programming

- Efficient data layout
- Elimination of branches

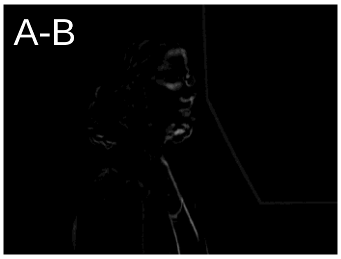
Application: frame difference

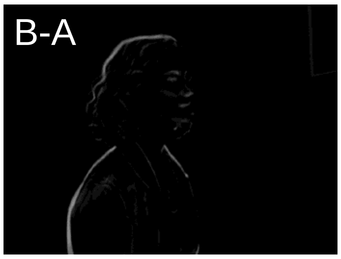




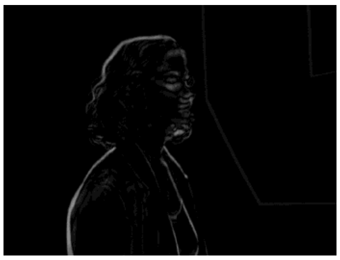


Application: frame difference





(A-B) or (B-A)



26

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|
```

27

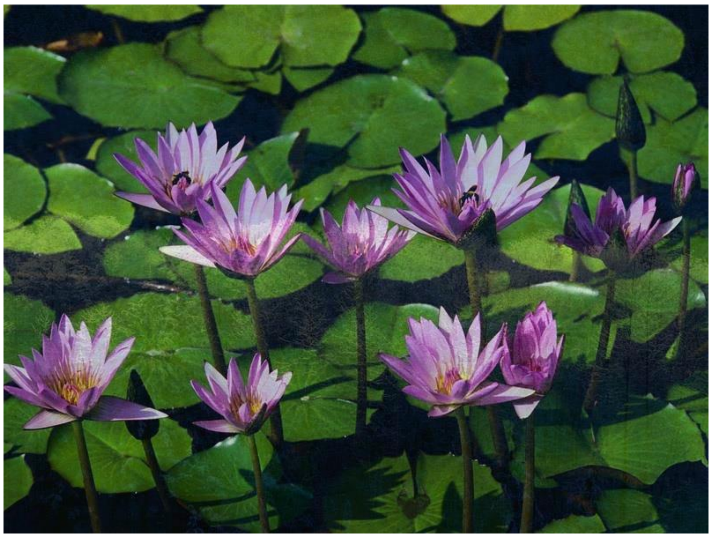
Example: image fade-in-fade-out

A

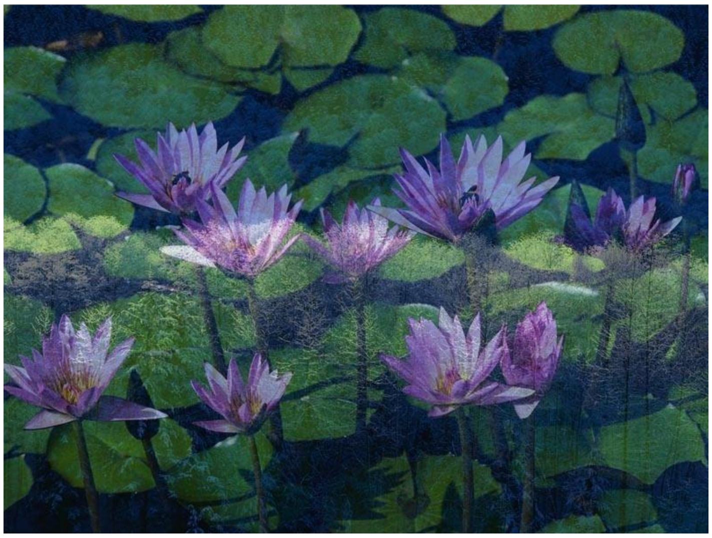
B

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

α=0.75



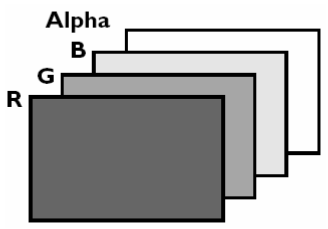
α=0.5

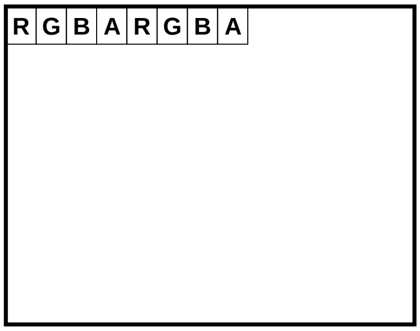


α=0.25

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

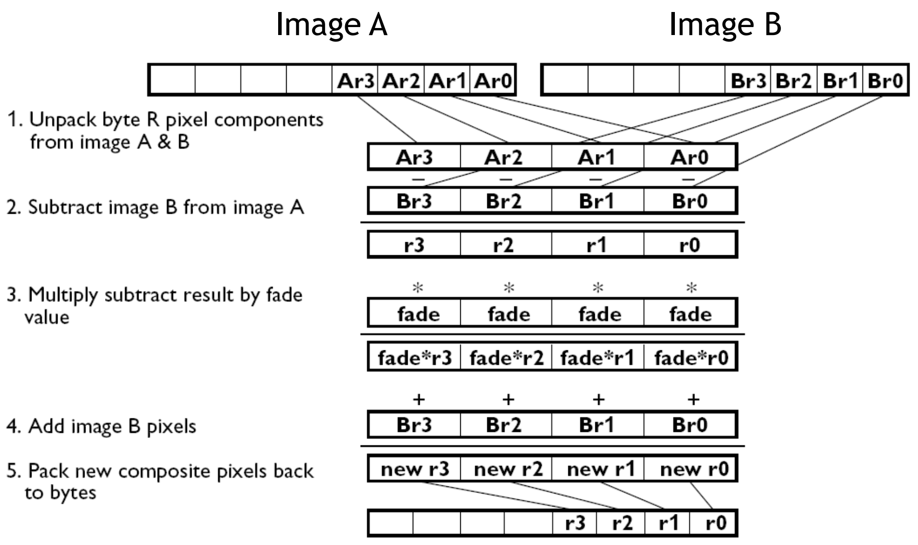


Image A
Image B
Ar3 Ar2 Ar1 Ar0
Br3 Br2 Br1 Br0
1. Unpack byte R pixel components from image A & B
2. Subtract image B from image A
r3 r2 r1 r0
3. Multiply subtract result by fade value
fade
fade*r3 fade*r2 fade*r1 fade*r0
4. Add image B pixels
5. Pack new composite pixels back to bytes
new r3 new r2 new r1 new r0

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

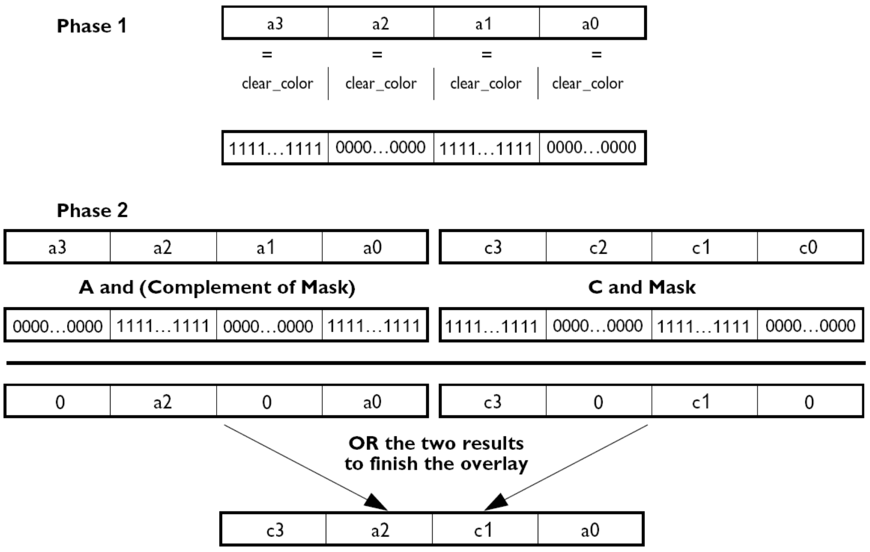
- Each operation can execute without needing to know the results of a previous operation.
- Example, sprite overlay

```
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.

35

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
```

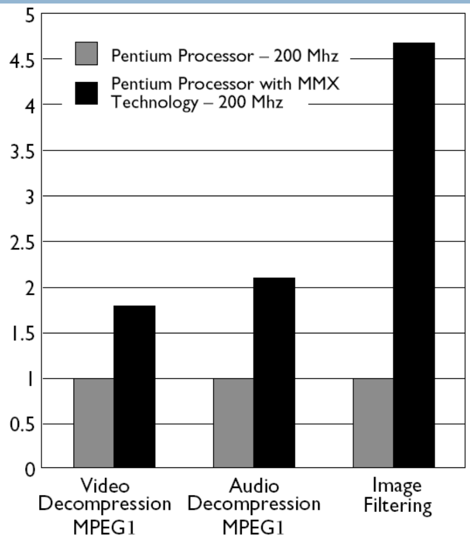
37

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



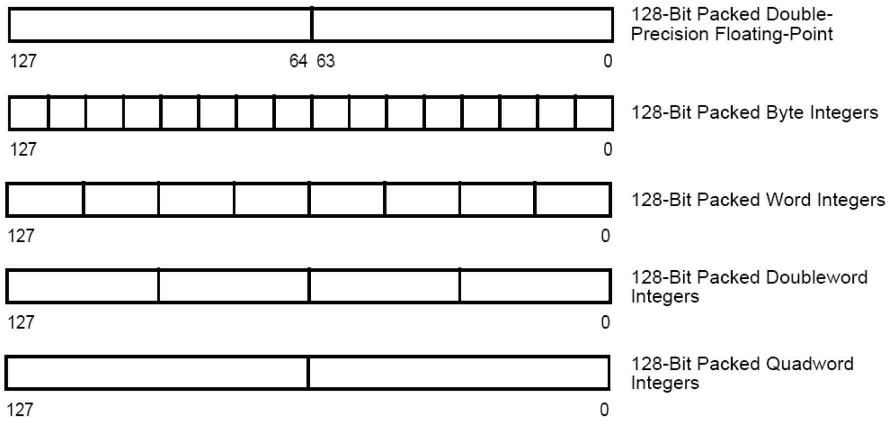
SSE

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

SSE features

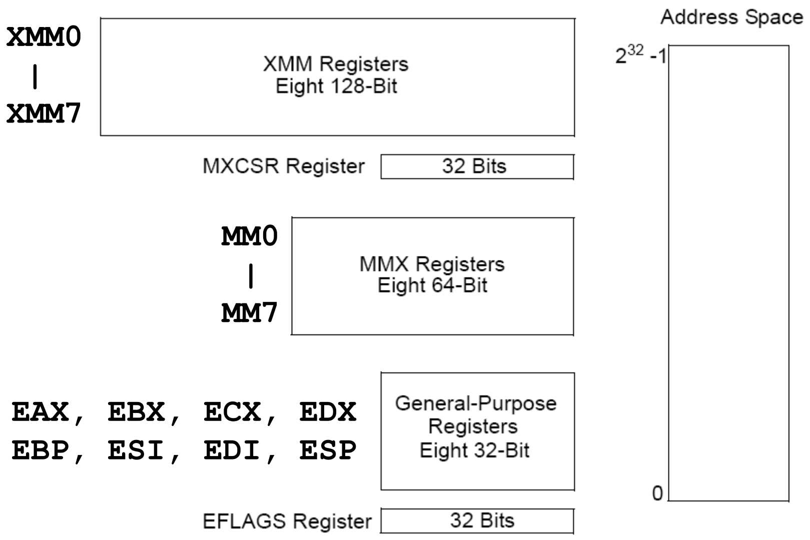
- Add eight 128-bit data registers (XMM registers) in non-64-bit modes; sixteen XMM registers are available in 64-bit mode.
- 32-bit MXCSR register (control and status)
- Add a new data type: 128-bit packed single-precision floating-point (4 FP numbers.)
- Instruction to perform SIMD operations on 128-bit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

SSE2 features

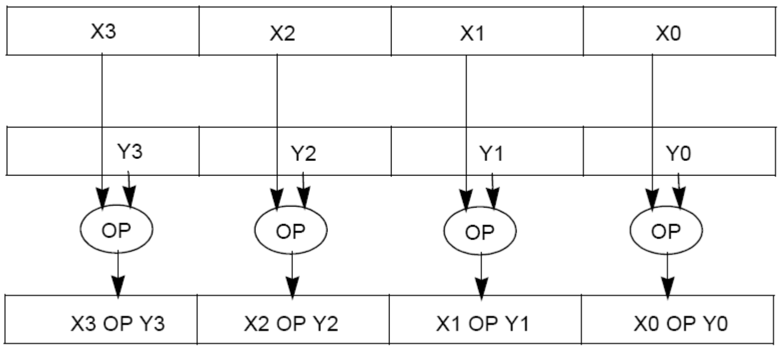

128-Bit Packed Double-Precision Floating-Point
127
64 63
0
128-Bit Packed Byte Integers
127
0
128-Bit Packed Word Integers
127
0
128-Bit Packed Doubleword Integers
127
0
128-Bit Packed Quadword Integers
127
0

41

SSE programming environment



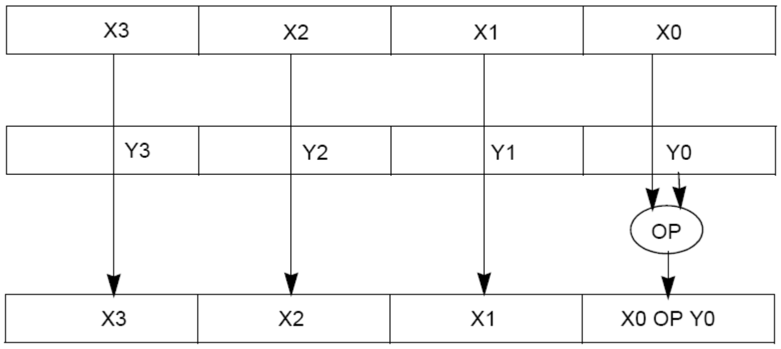
SSE packed FP operation



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

43

SSE scalar FP operation



- **ADDSS/SUBSS**: scalar single-precision FP

used as FPU?

44

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

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
}
```

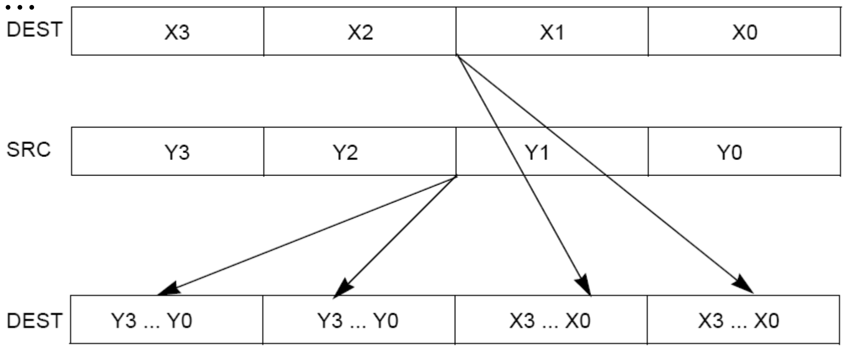
movaps: move aligned packed single-precision FP
addps: add packed single-precision FP

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 );
}
```