Intel SIMD architecture

Z. Jerry Shi
Associate Professor of Computer Science and Engineering
University of Connecticut

Revised from Yung-Yu Chuang's slides , 2007

Overview

- SIMD
- MMX architectures
- MMX instructions
- Examples
- SSE/SSE2/SSE3/SSE4

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

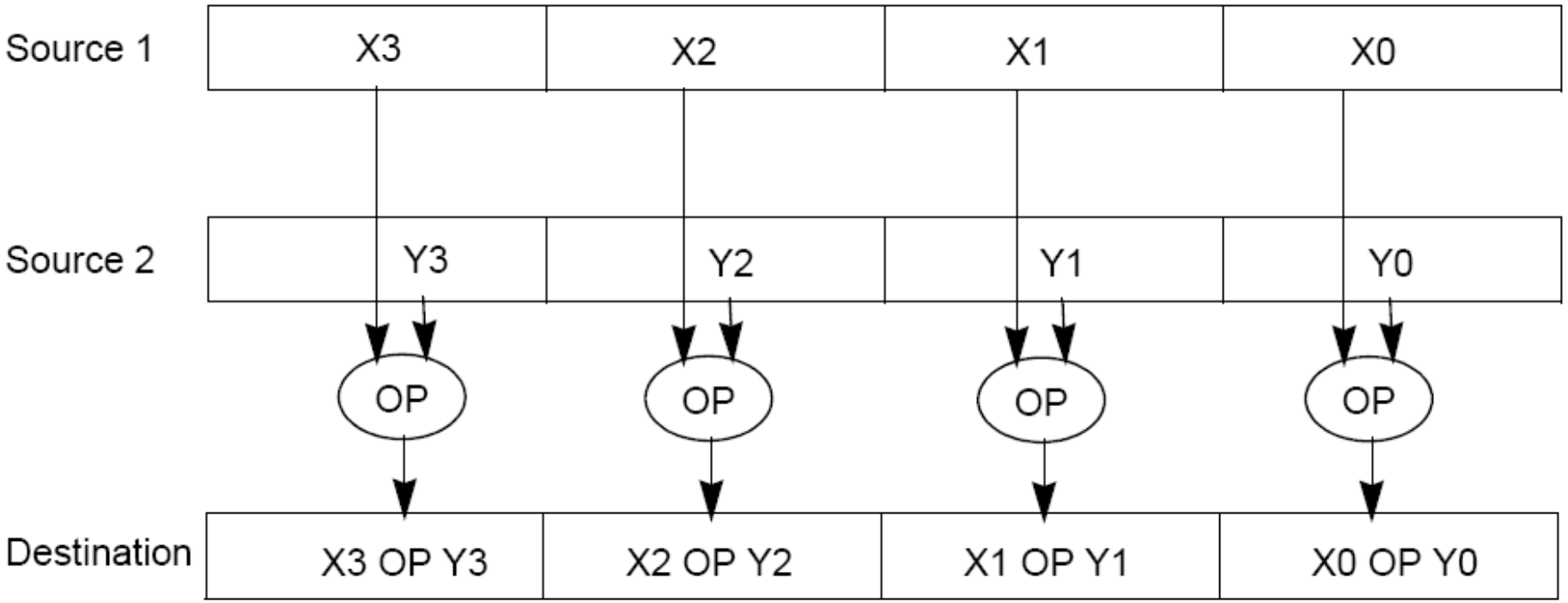
Performance boost

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

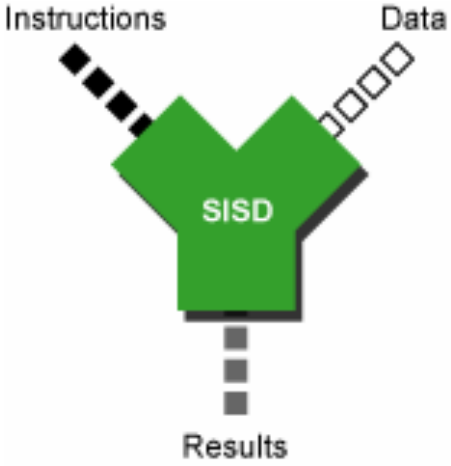
SIMD

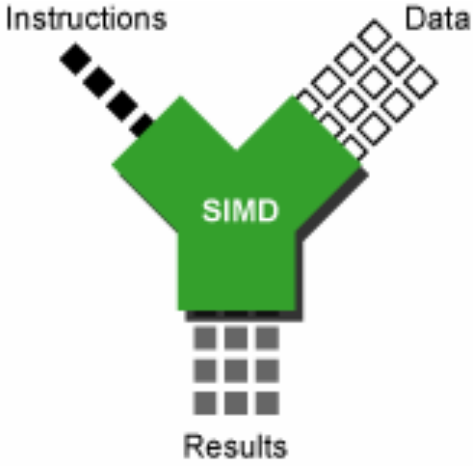
- SIMD (single instruction multiple data) architecture performs the same operation on multiple data elements in parallel

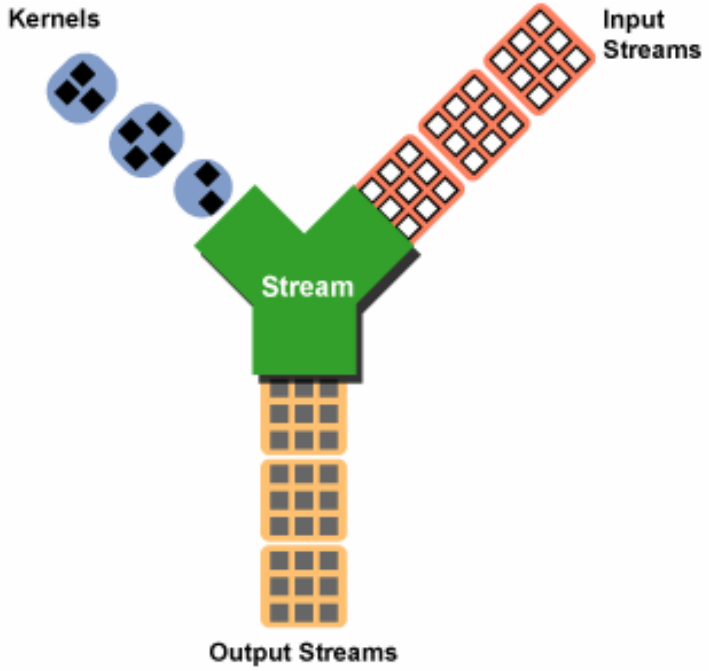
```
PADDW MM0, MM1
```



SISD/SIMD/Streaming







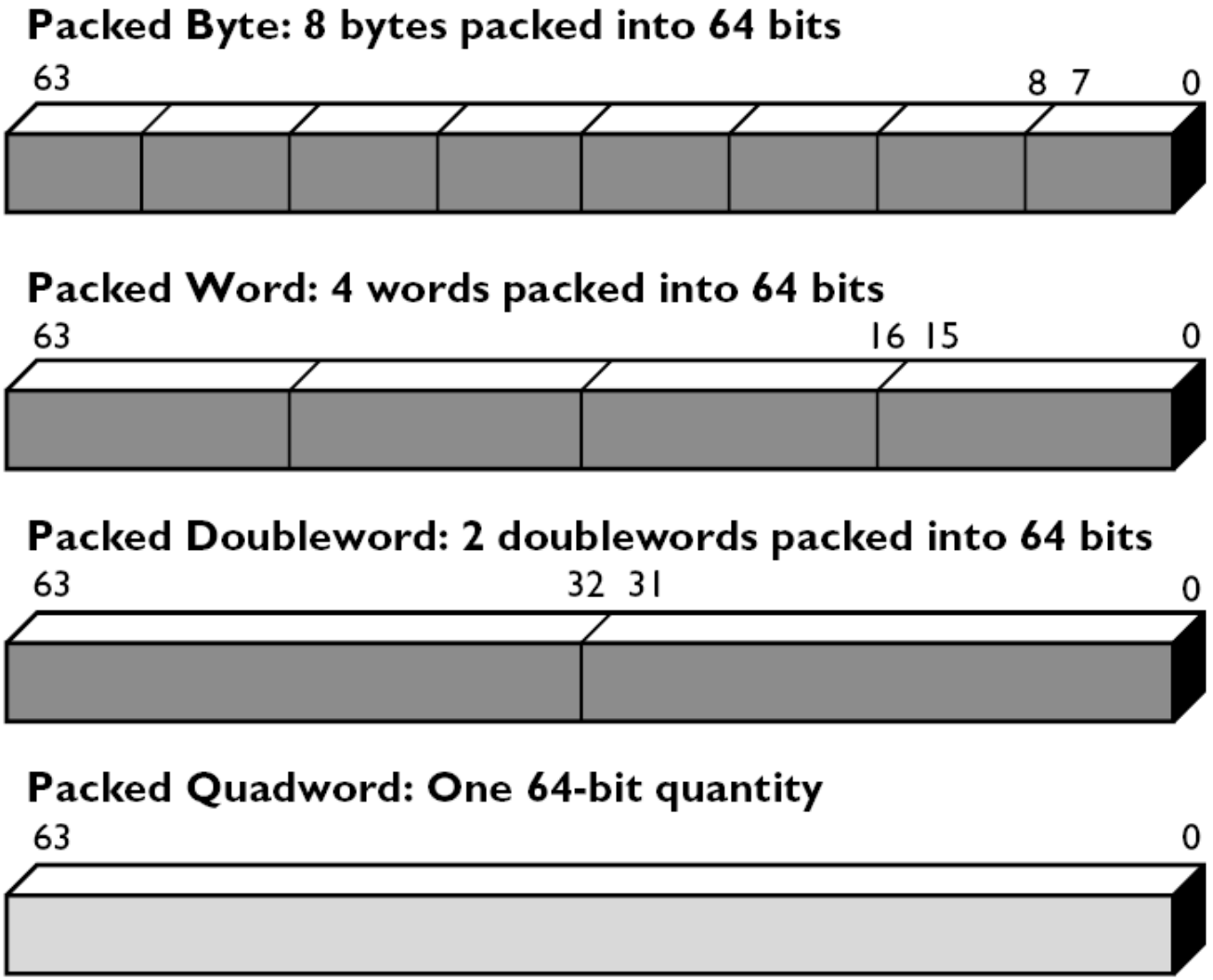
IA-32 SIMD development

- 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
  - For supporting hyper-threading technology
  - 13 more instructions
- SSSE3 (Supplemental) in June 2006 in the “Woodcrest” Xeons
  - 16 new discrete instructions
- SSE4 in spring 2007 has 54 new instructions
  - 47 in SSE4.1
  - 7 in SSE 4.2
- SSE4a from AMD are different from SSE4.1

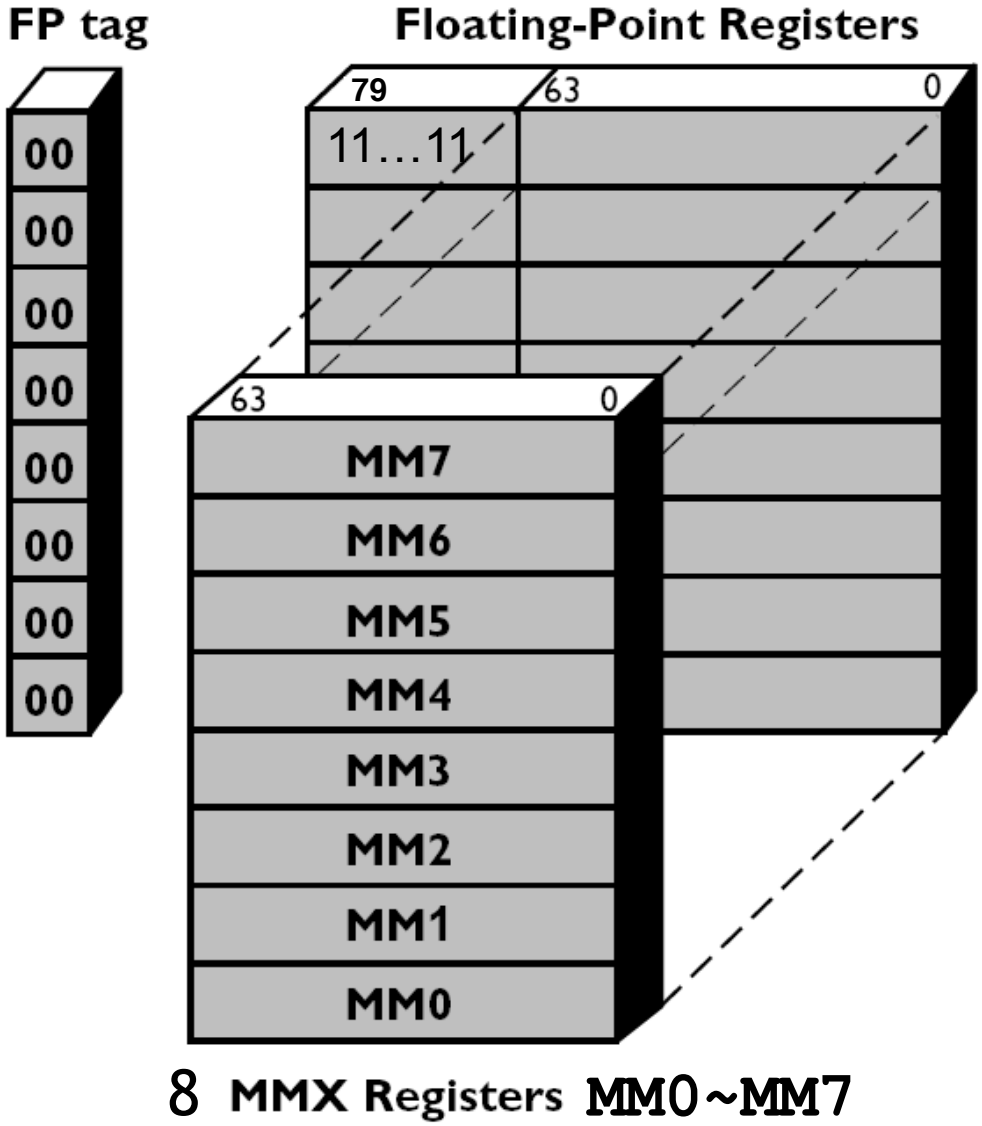
MMX

- Typical elements in many applications are small
  - 8 bits for pixels
  - 16 bits for audio
  - 32 bits for general computing
- New data type: 64-bit packed data type. Why 64 bits?
  - Good enough
  - Practical, see in a moment

MMX data types



MMX integration into IA-32



8 MMX Registers MM0~MM7

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.

Compatibility

- Fully compatible with existing IA
  - No new mode or state was created
  - No extra state needs to be saved for context switching
- MMX is hidden behind FPU
  - When floating-point state is saved or restored, MMX is saved or restored.
- Existing OS to perform context switching on the processes executing MMX instruction without be aware of MMX
  - MMX and FPU cannot be used at the same time
    - It may be a bad decision
    - OS can just provide a service pack or get updated
    - Intel introduced SSE later without any aliasing

MMX instructions

- 57 MMX instructions
  - **`add`**, **`subtract`**, **`multiply`**, **`multiply-add`**
  - **`compare`**
  - **`shift`**, **`logical operation`**
  - **`data conversion`**
  - **`64-bit data move`**
- All instructions except for data move use MMX registers as operands
  - All starts with p except for movd, movq, and emms
- Most complete support for 16-bit operations

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		

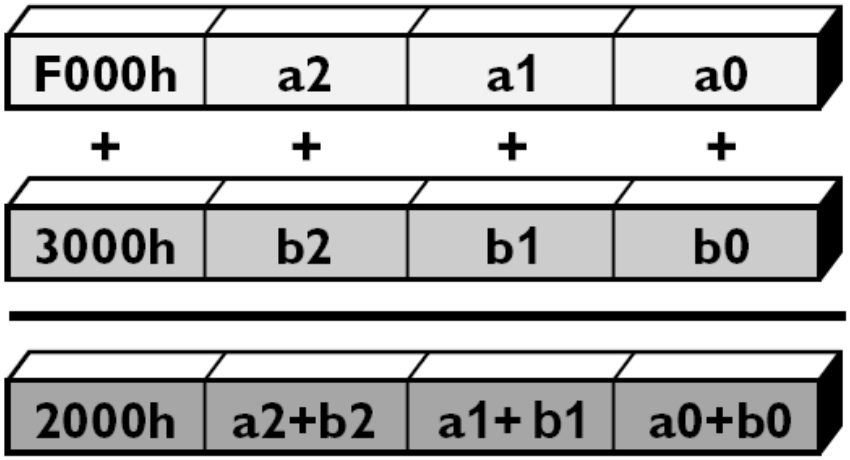
MMX instructions

		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	

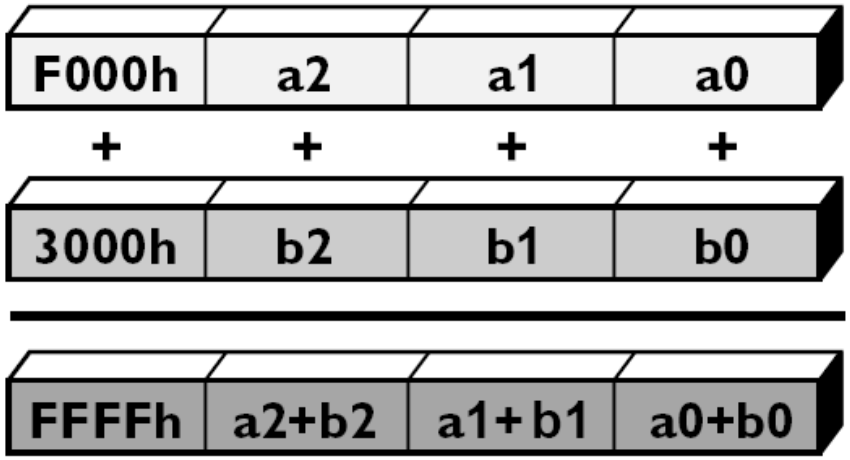
Call it before you switch to FPU from MMX

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

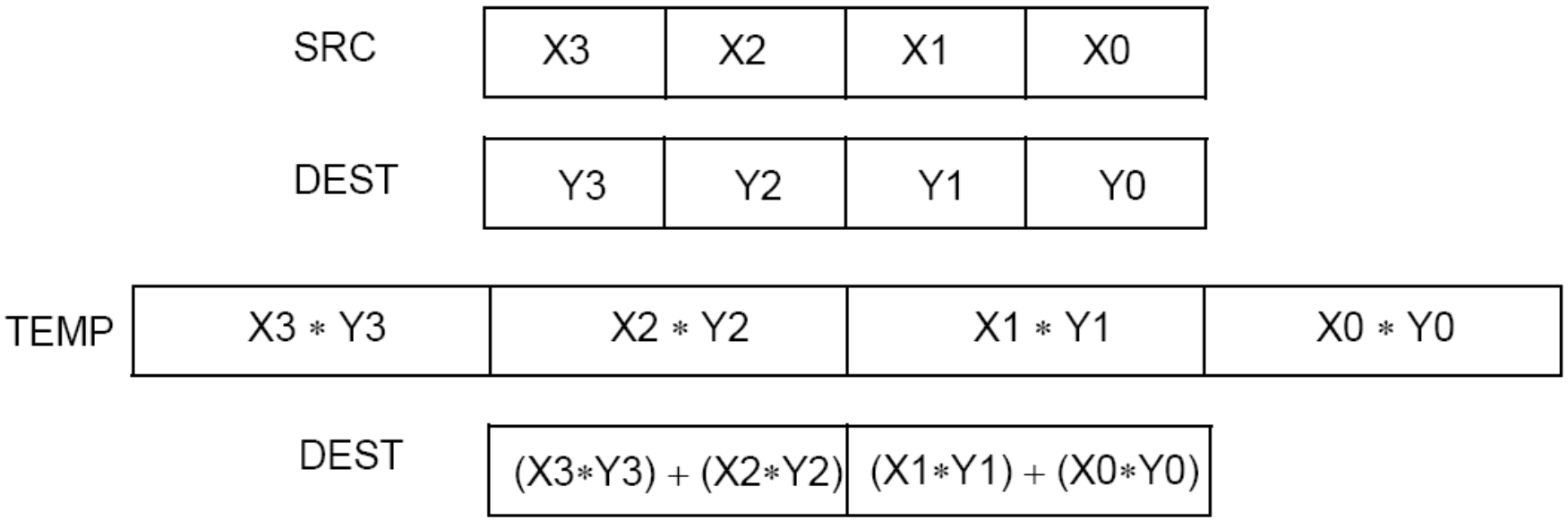
Arithmetic

- **`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.

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]);

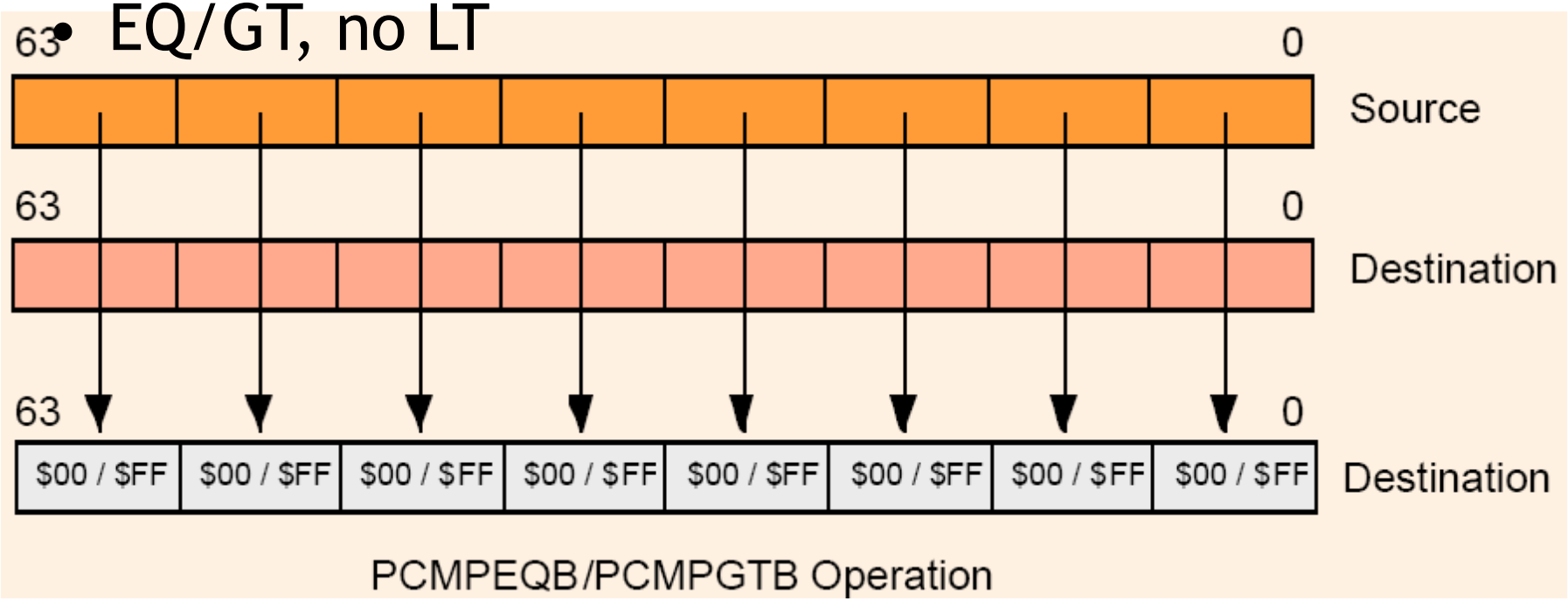


Example: add a constant to a vector

```
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

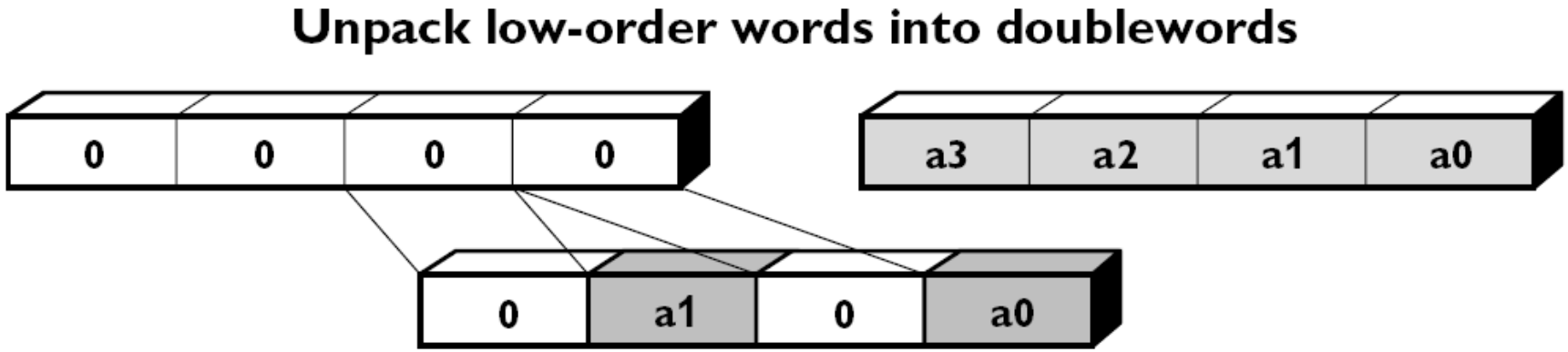
- No CFLAGS, how many flags will you need? Results are stored in destination.
- EQ/GT, no LT



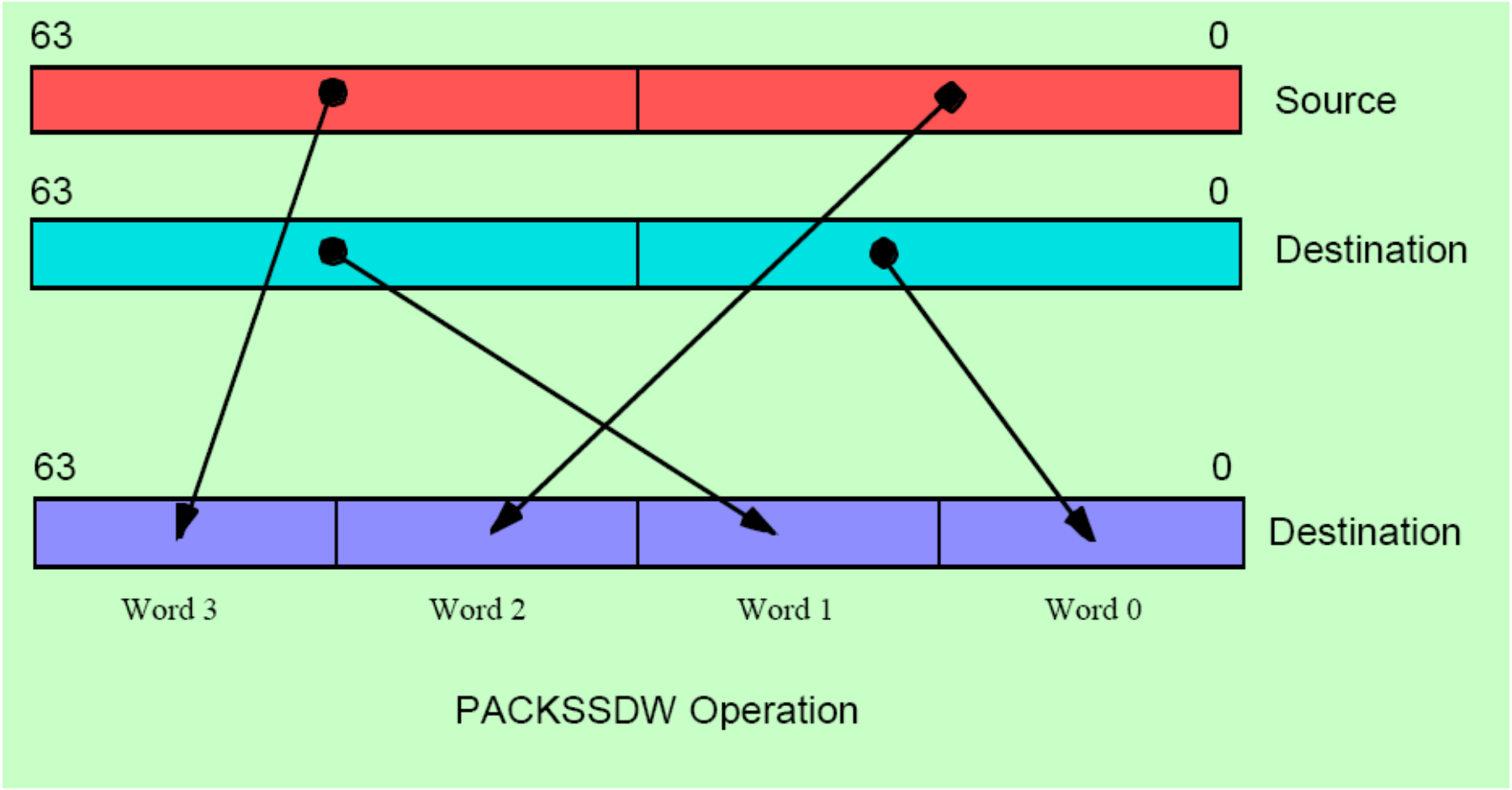
PCMPEQB/PCMPGTB Operation

Change data types

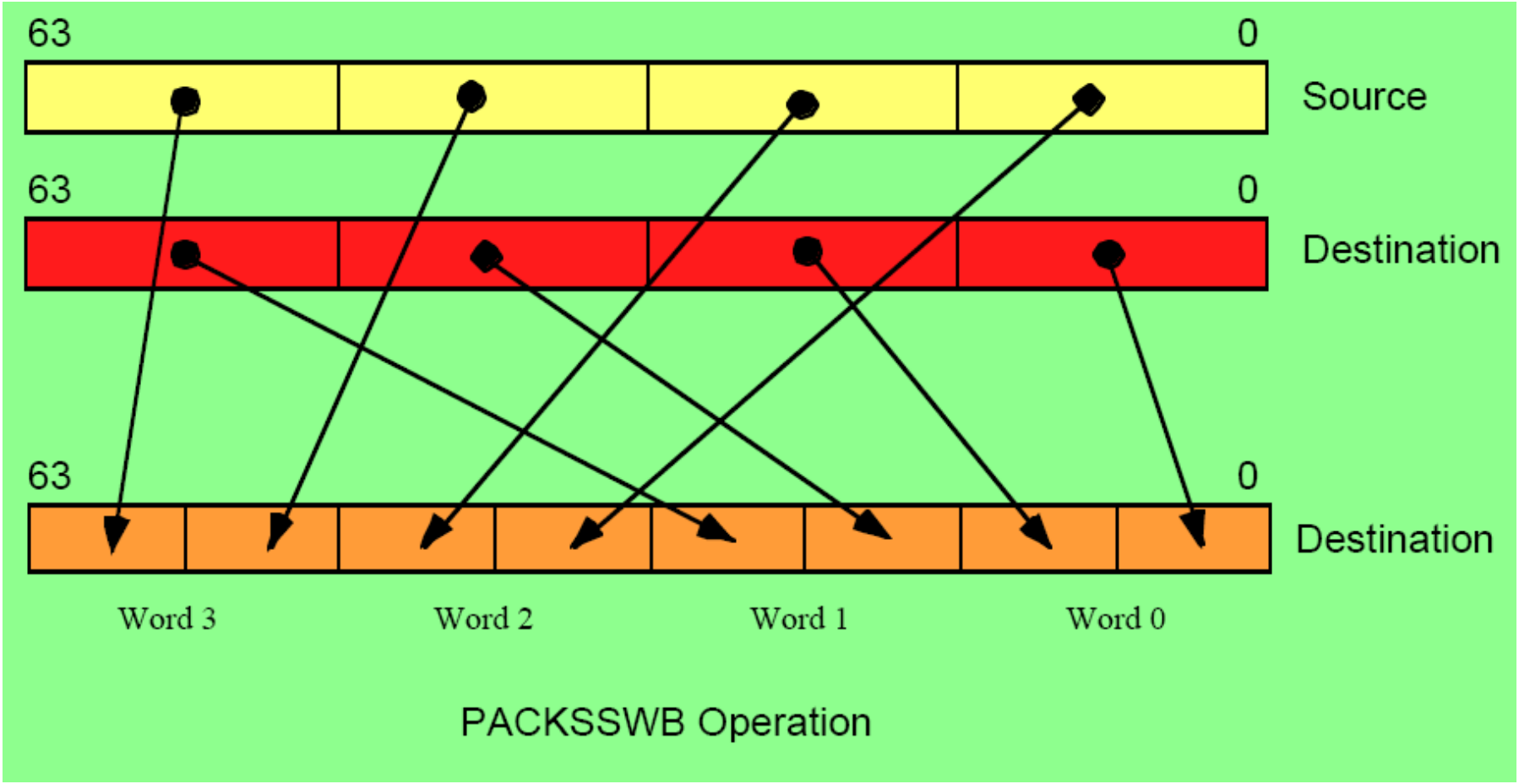
- Pack: converts a larger data type to the next smaller data type.
- Unpack: takes two operands and interleave them. It can be used for expand data type for immediate calculation.



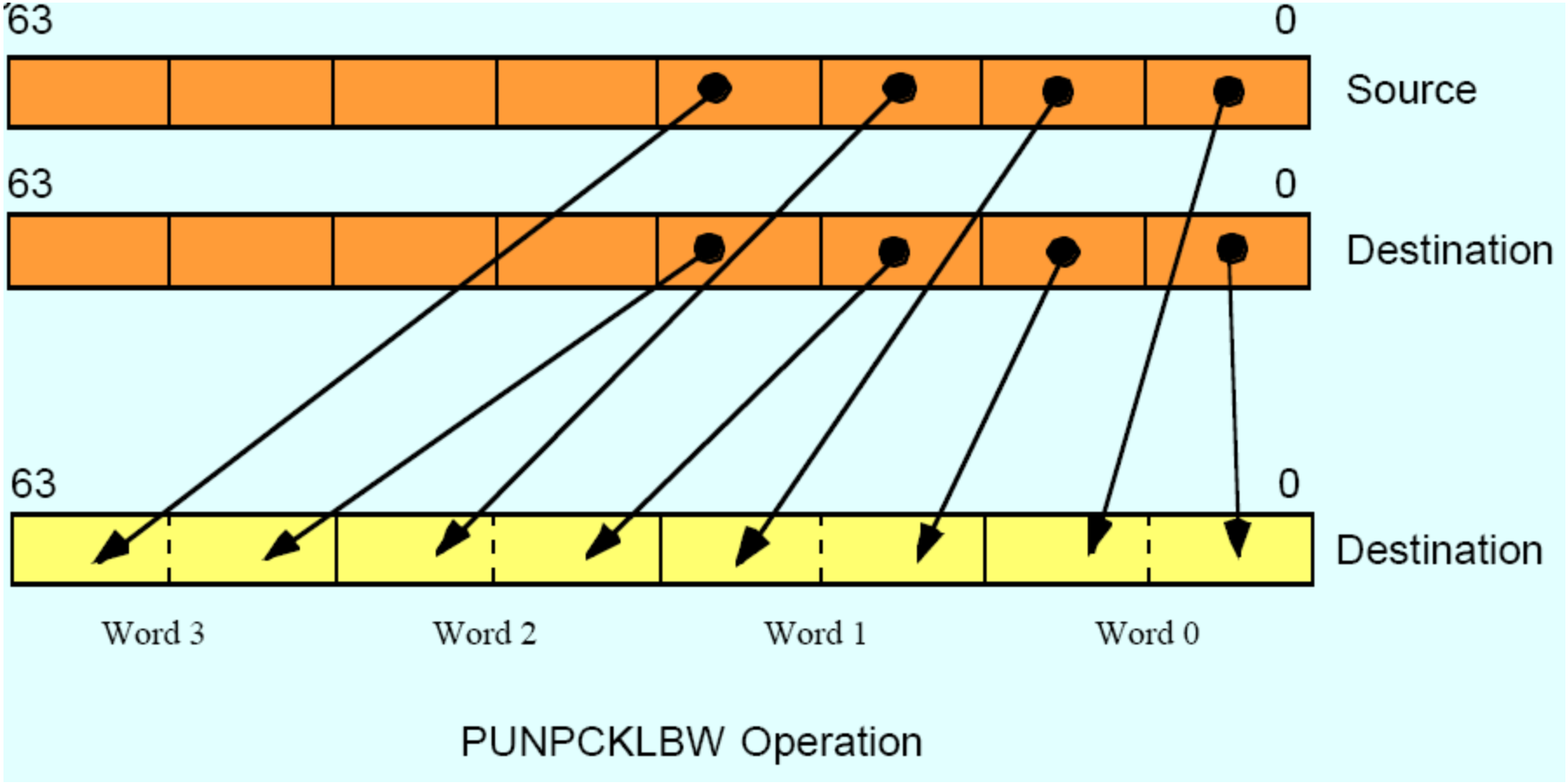
Pack with signed saturation



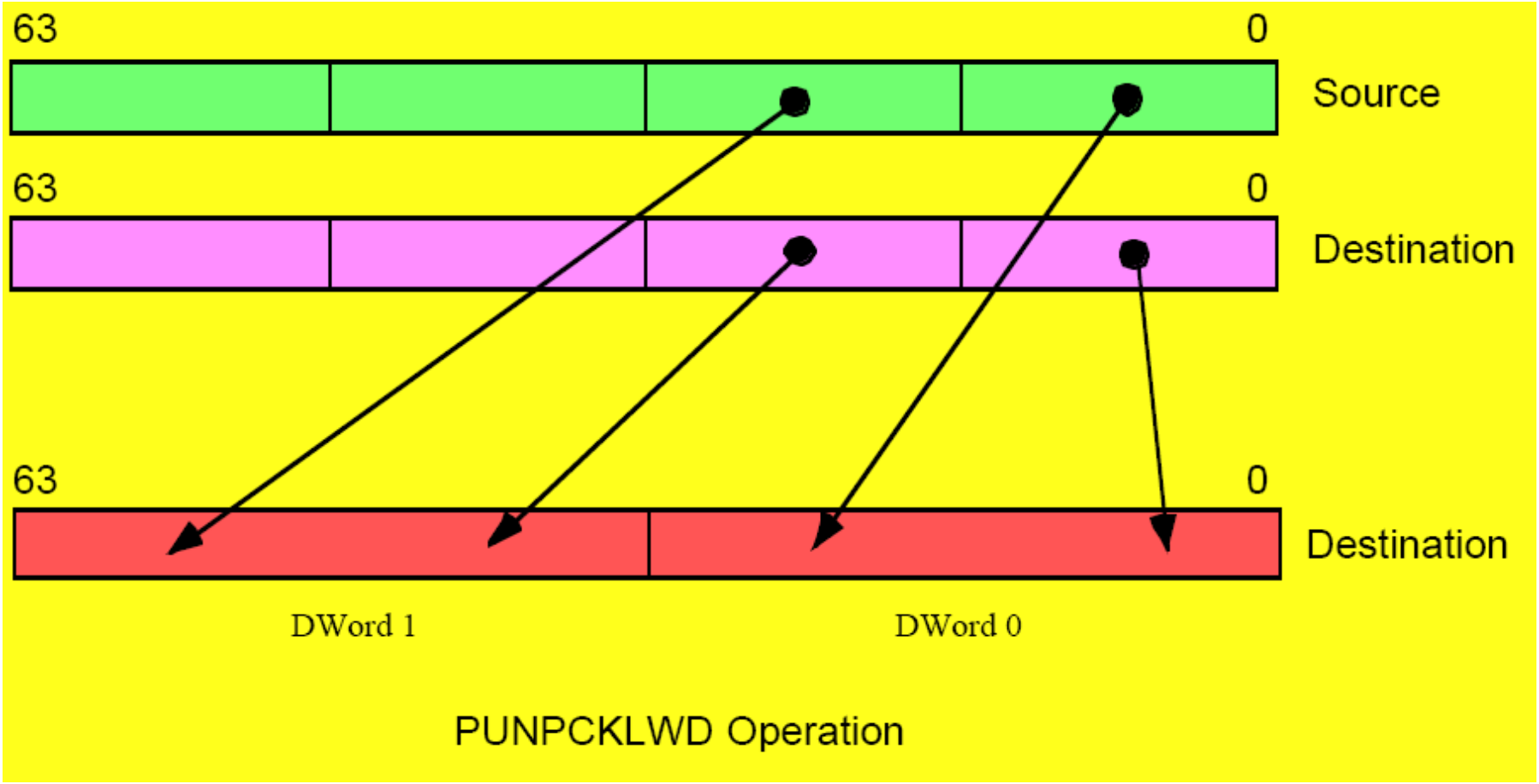
Pack with signed saturation



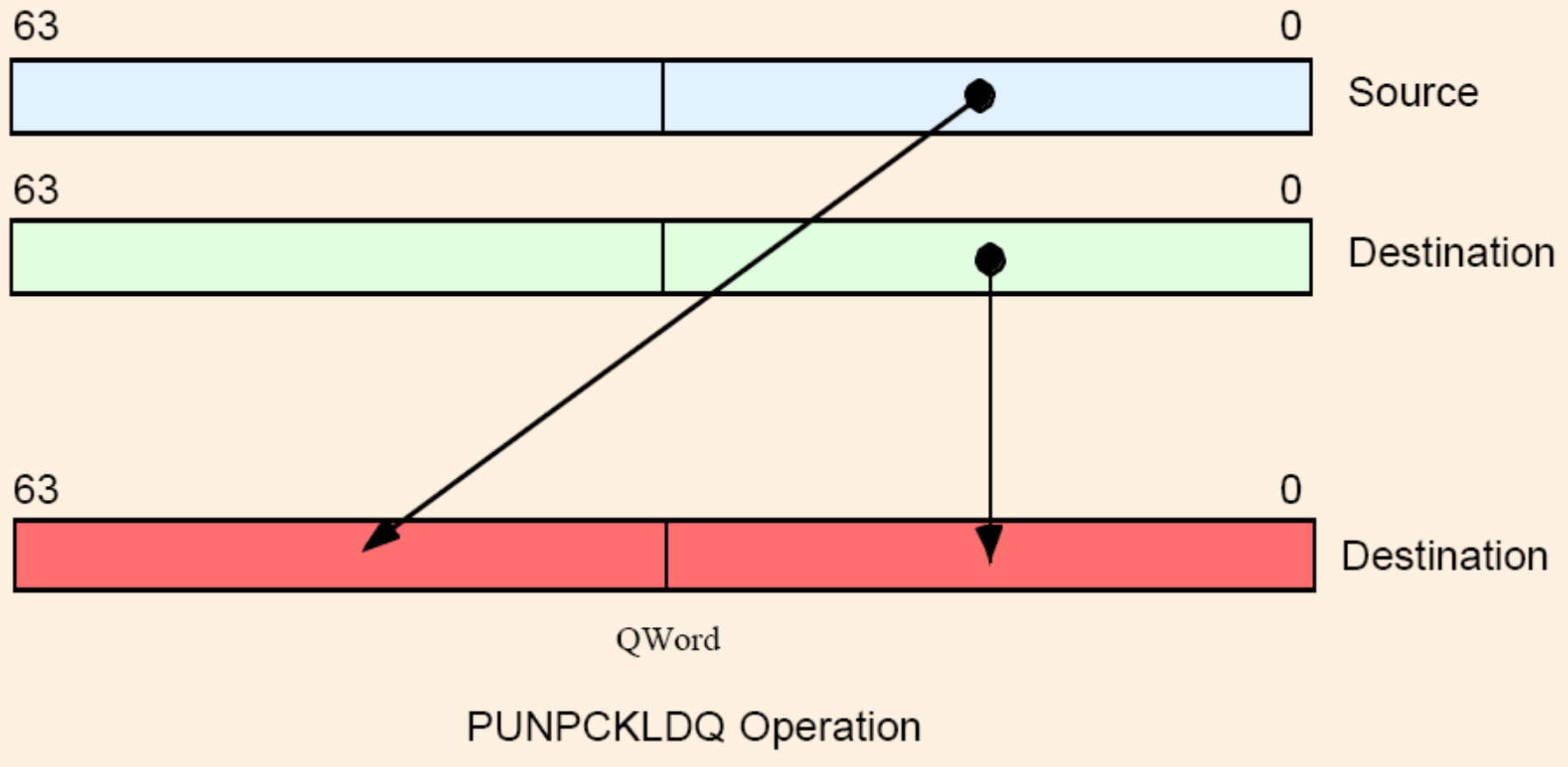
Unpack low portion



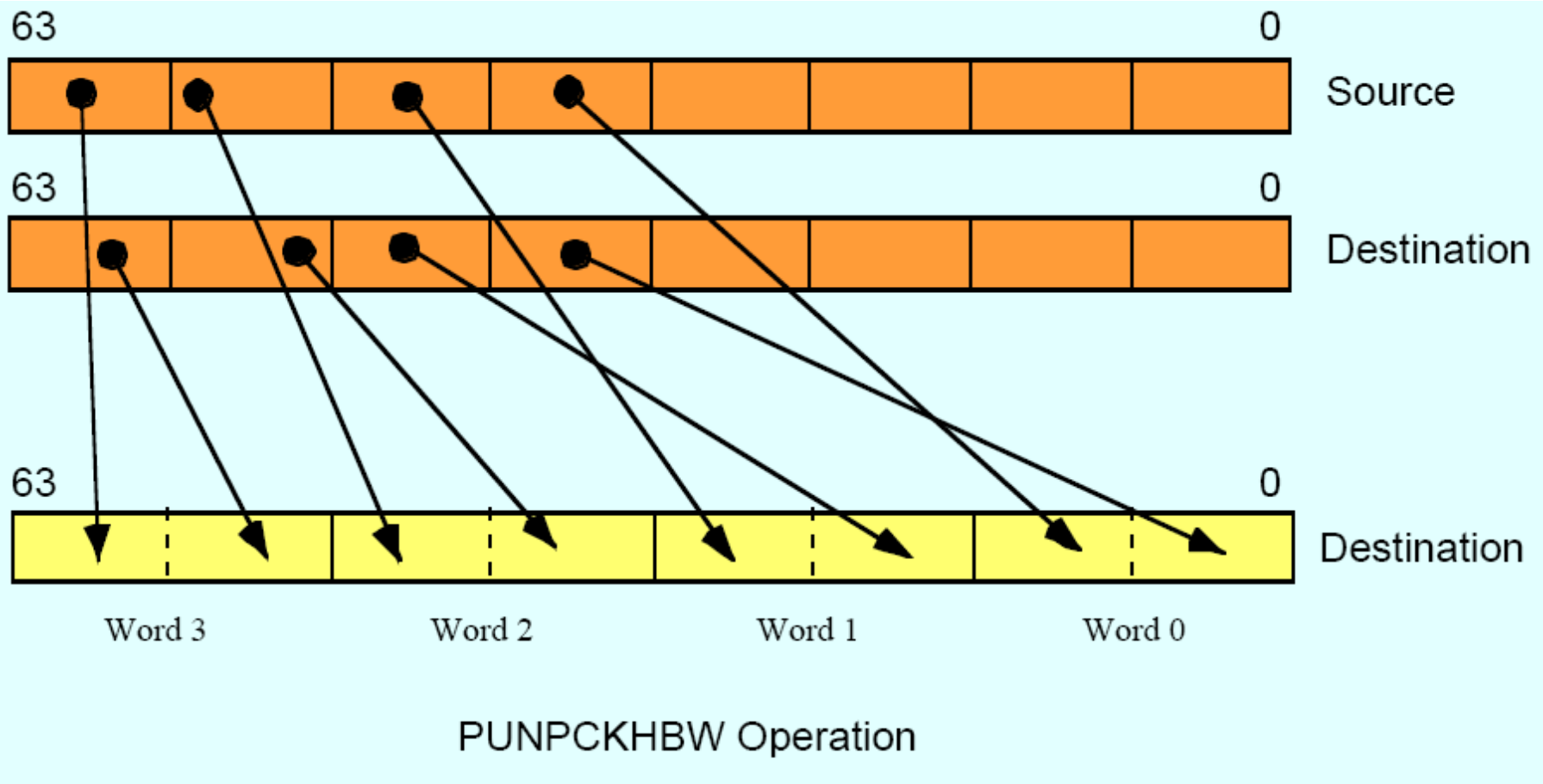
Unpack low portion



Unpack low portion



Unpack high portion

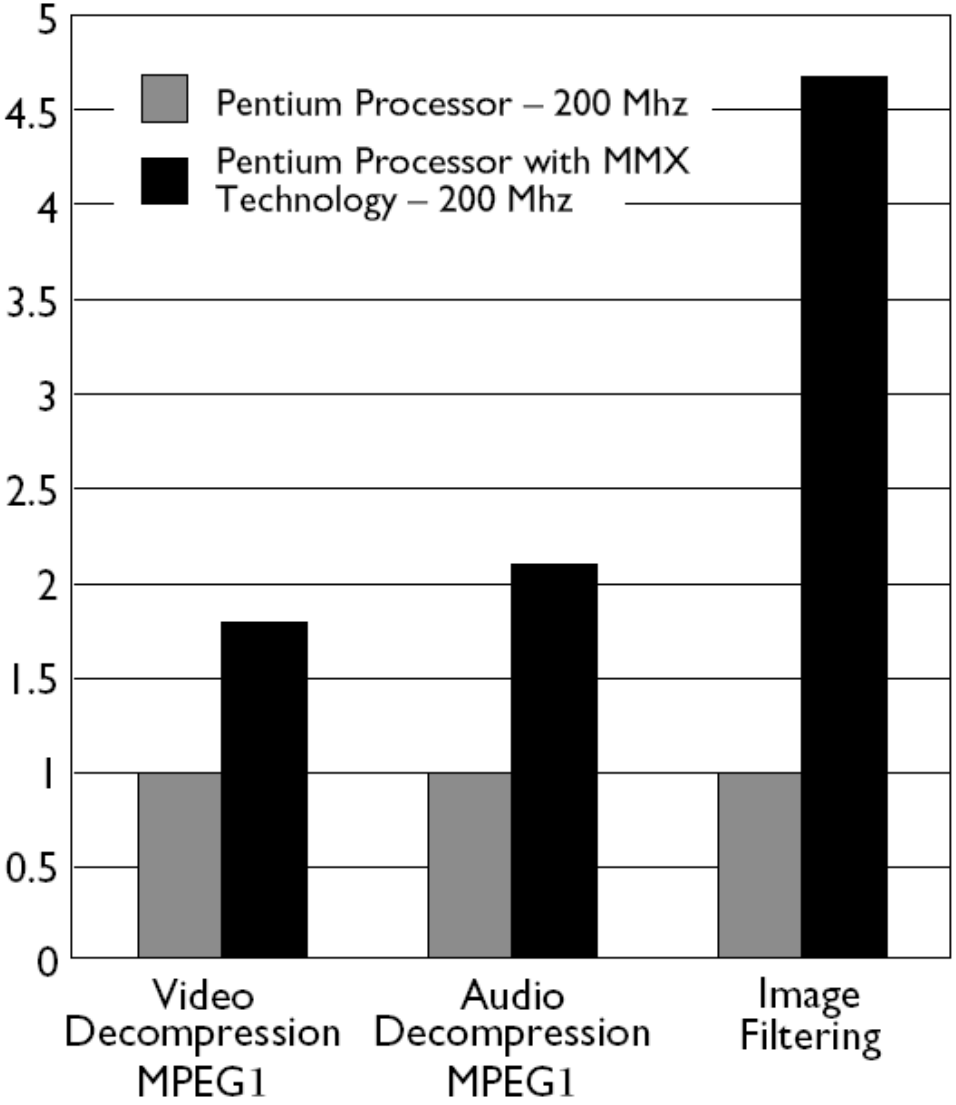


Performance boost (data from 1996)

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

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

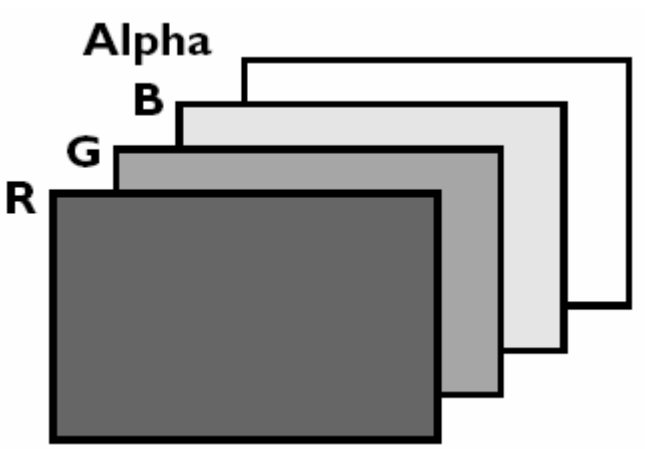
α=0.75

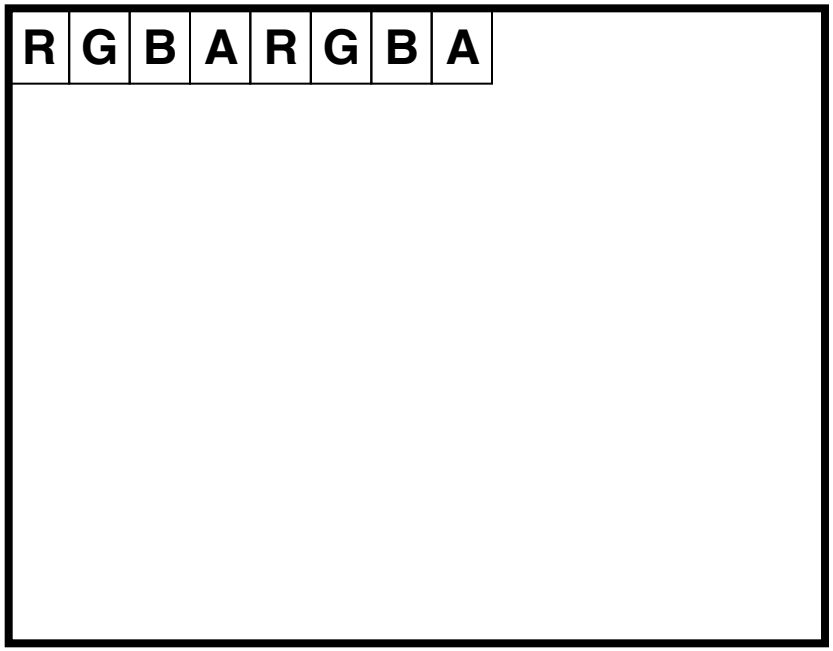
α=0.5

α=0.25

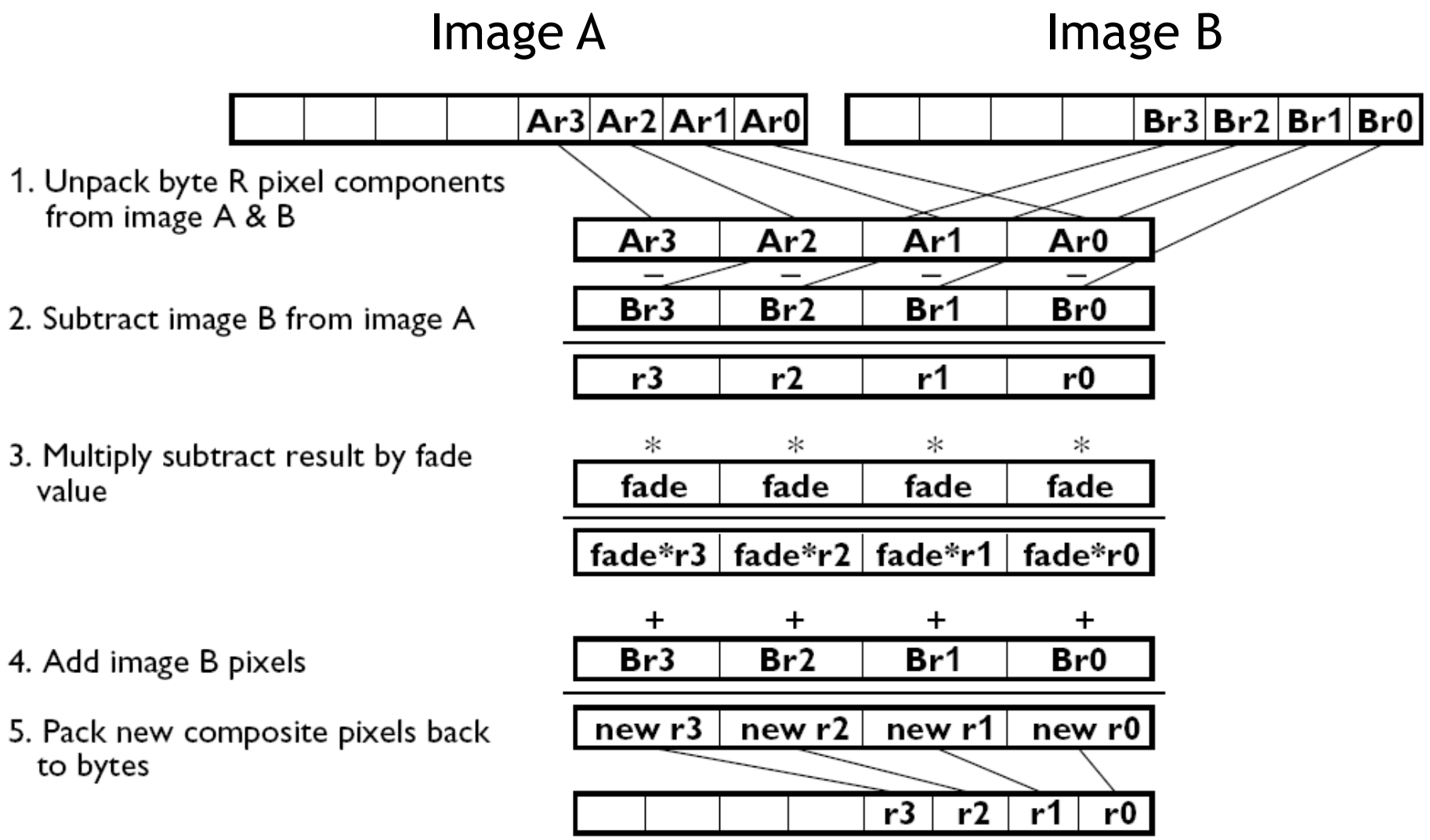
Example: image fade-in-fade-out

- 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



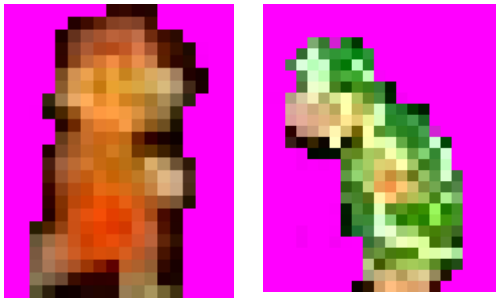
Example: image fade-in-fade-out

```
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
```

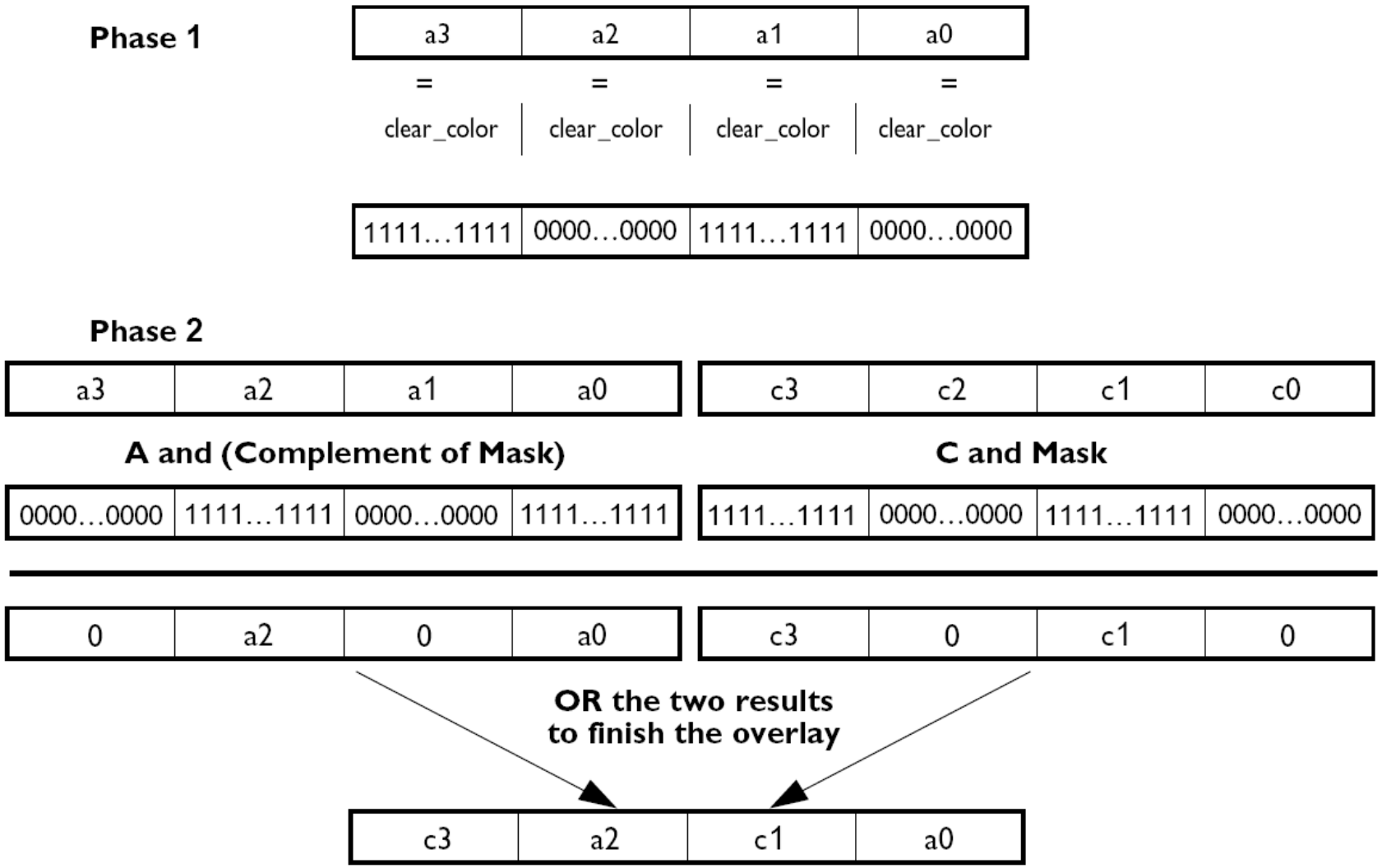
Data-independent computation

- 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]
```



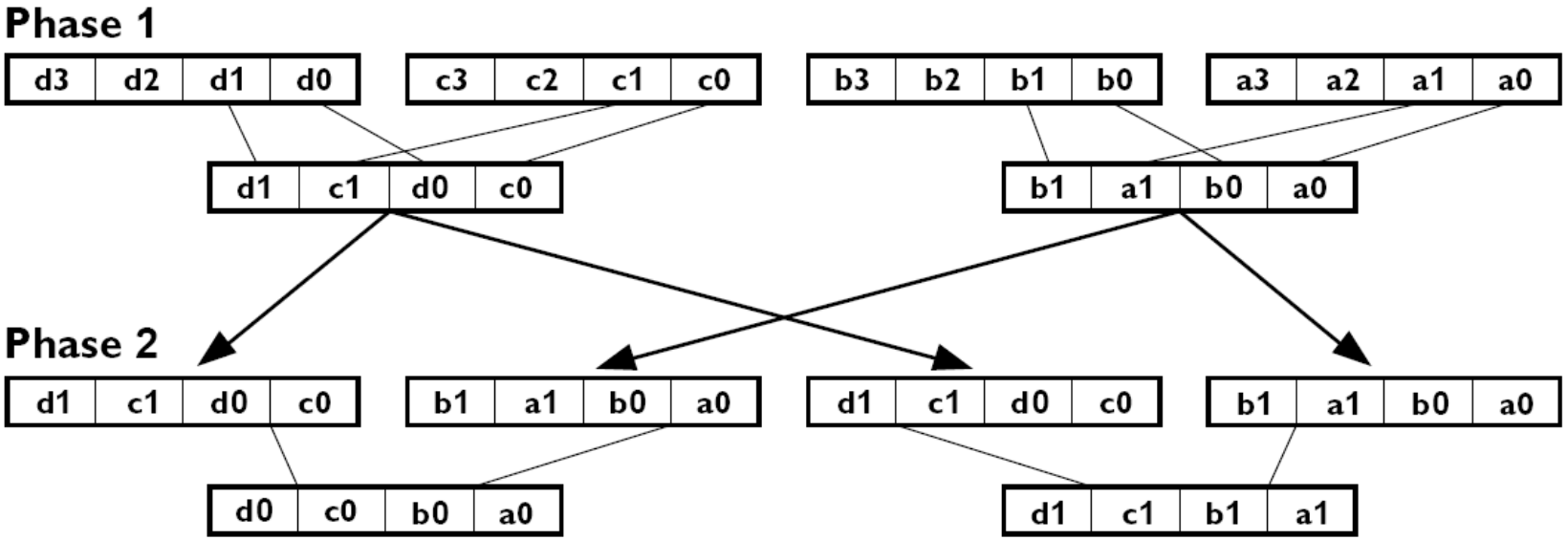
Application: sprite overlay



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 transpose



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 transpose (C code)

```
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++; }
```

Application: matrix transpose (MMX) - 1

```
__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

//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 transpose (MMX) - 2

```
//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
```

How to use assembly in projects

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

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


31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
EDX
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

Link ASM and High-level Language 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 high-level language's capabilities

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

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.

__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
}
```

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

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 ) );
```

SSE features

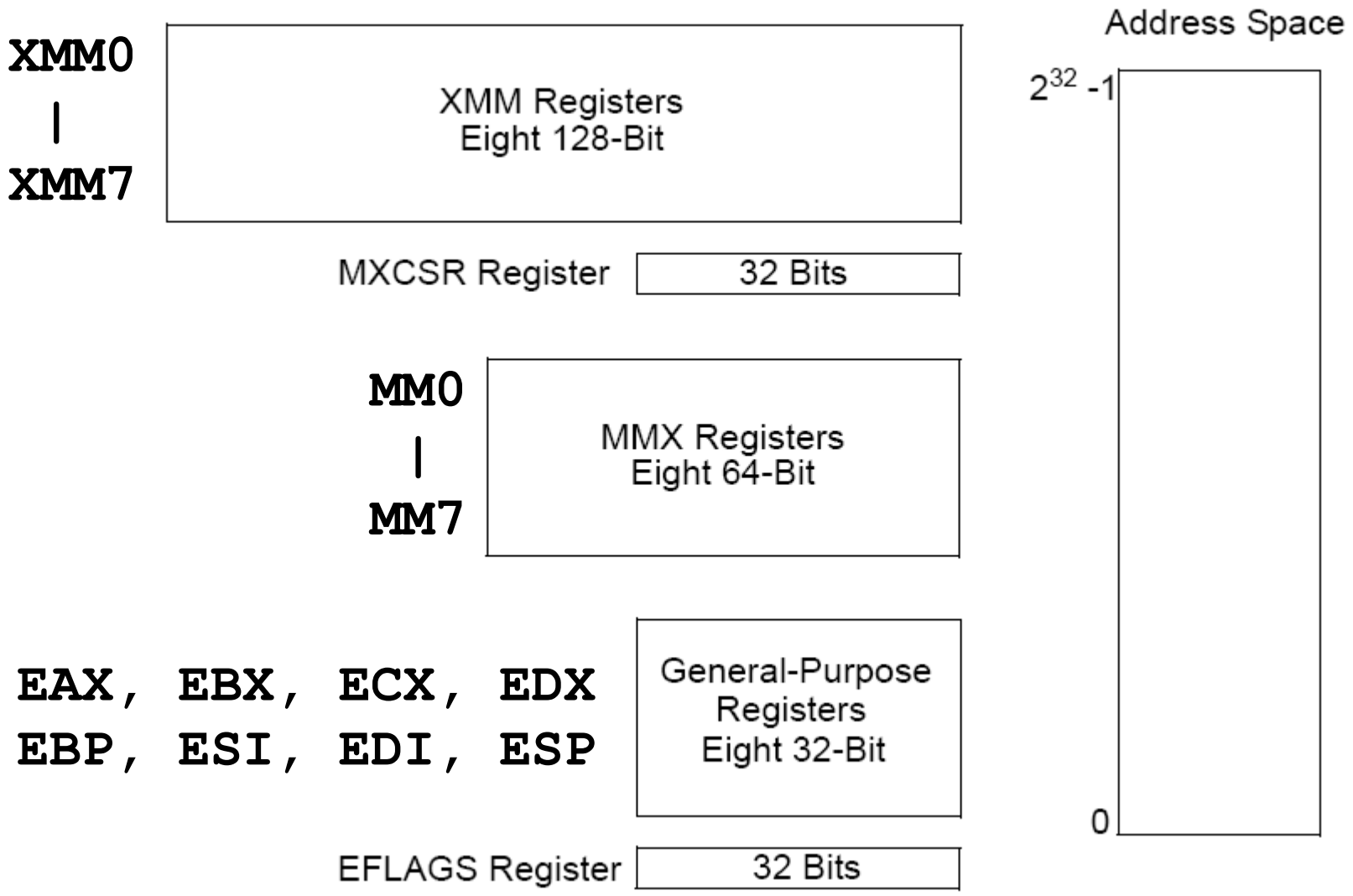
- Add eight 128-bit data registers (XMM registers)
- Sixteen XMM registers are available in 64-bit mode
- 32-bit MXCSR register (control and status)
- Add a new data type
  - 4 single-precision floating-point numbers in a 128-bit register
- New instructions:
  - Instruction to perform SIMD operations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

Advantages of SSE

In MMX

- An application cannot execute MMX instructions and perform floating-point operations simultaneously.
- A large number of processor clock cycles are needed to change the state of executing MMX instructions to the state of executing FP operations and vice versa.

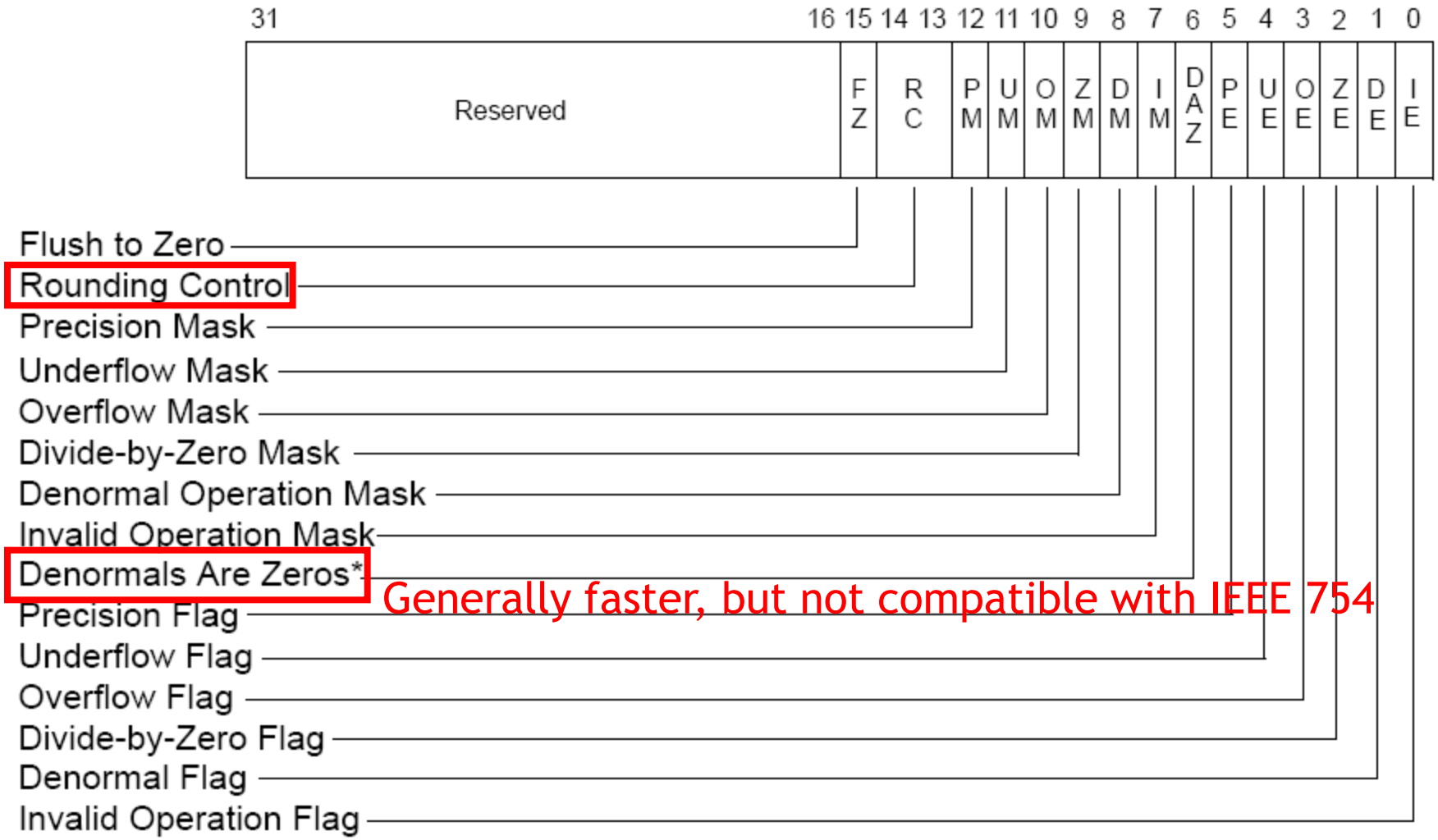
SSE programming environment



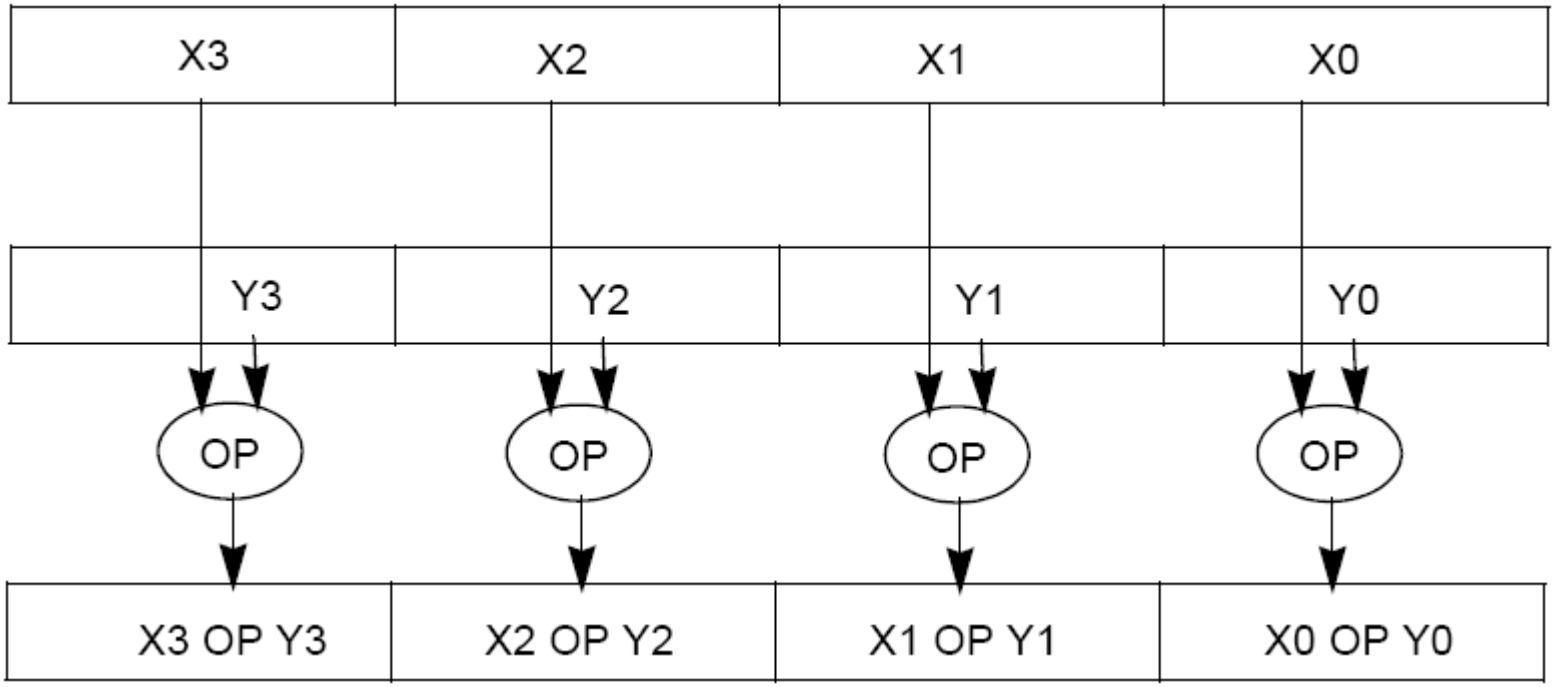
Exception

```
_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;
}
```

MXCSR control and status register



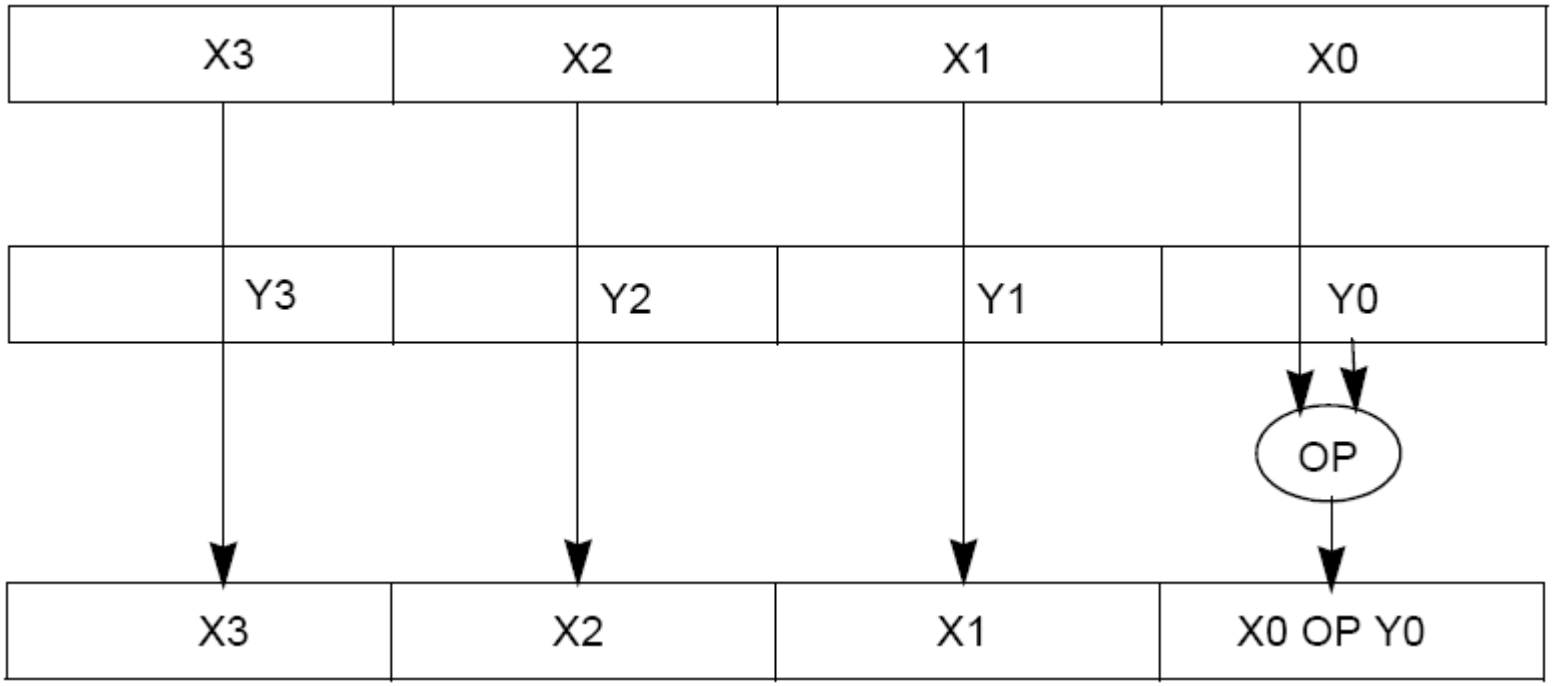
SSE Packed FP Operation



Packed single-precision FP

ADDPS, SUBPS, MULPS, DIVPS, RCPPS, SQRTPS, RSQRTPS, MAXPS, MINPS

SSE Scalar FP Operation



Scalar single-precision FP used as FPU

ADDSS, SUBSS, MULSS, DIVSS, RCPSS, SQRTSS, RSQRTSS, MAXSS, MINSS

SSE Shuffle Packed Single-Precision (SHUFPS)

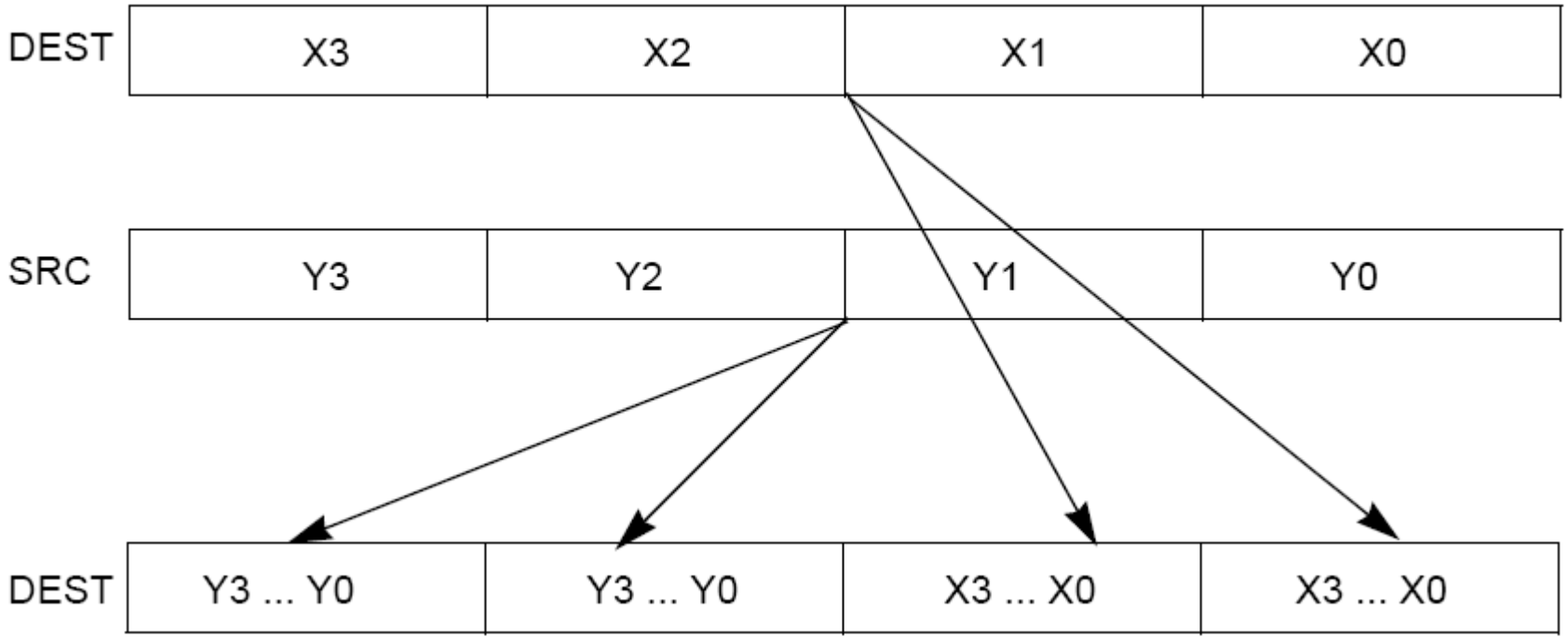
SHUFPS xmm1, xmm2, imm8

Select [1..0] select DEST[0] from DEST // xmm1

Select [3..2] select DEST[1] from DEST

Select [5..4] select DEST[2] from SRC // xmm2

Select [7..6] select DEST[3] from SRC

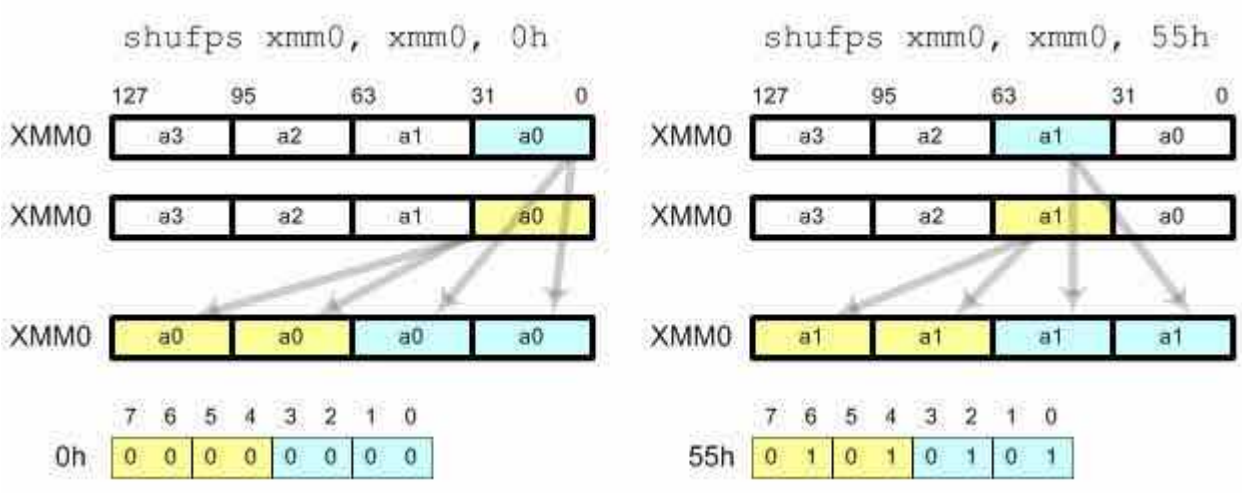


SSE Shuffle Packed Single-Precision (SHUFPS) - 2

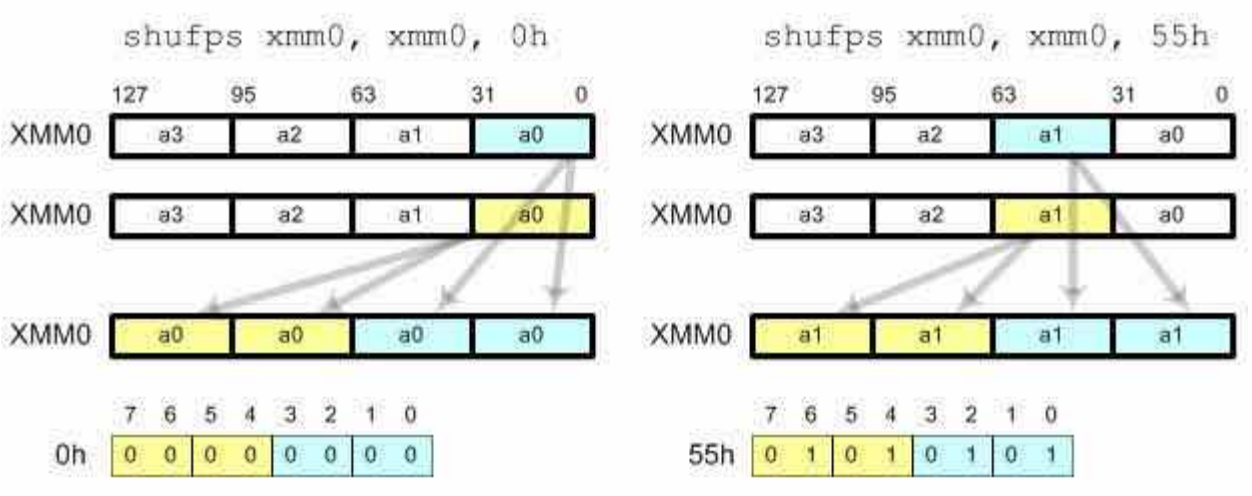
```
CASE (SELECT[1:0]) OF
    0:  DEST[31:0] ← DEST[31:0];
    1:  DEST[31:0] ← DEST[63:32];
    2:  DEST[31:0] ← DEST[95:64];
    3:  DEST[31:0] ← DEST[127:96];
ESAC;

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;
```

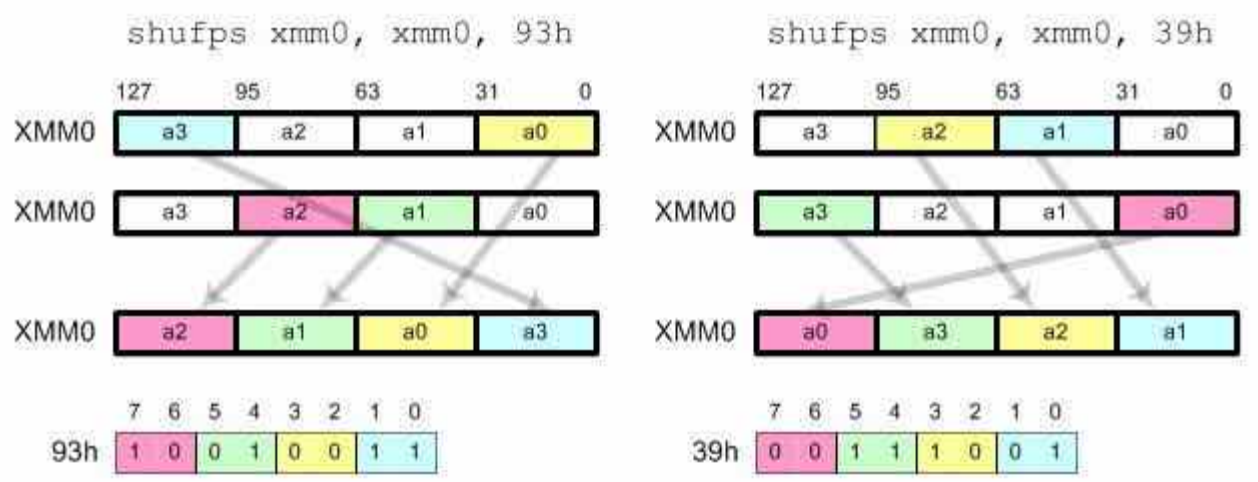
Swap bytes with SHUFPS



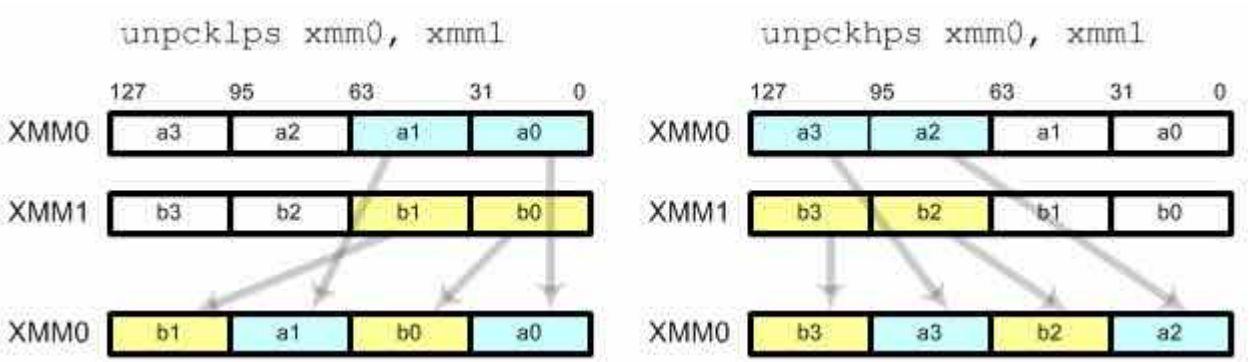
Broadcast with SHUFPS



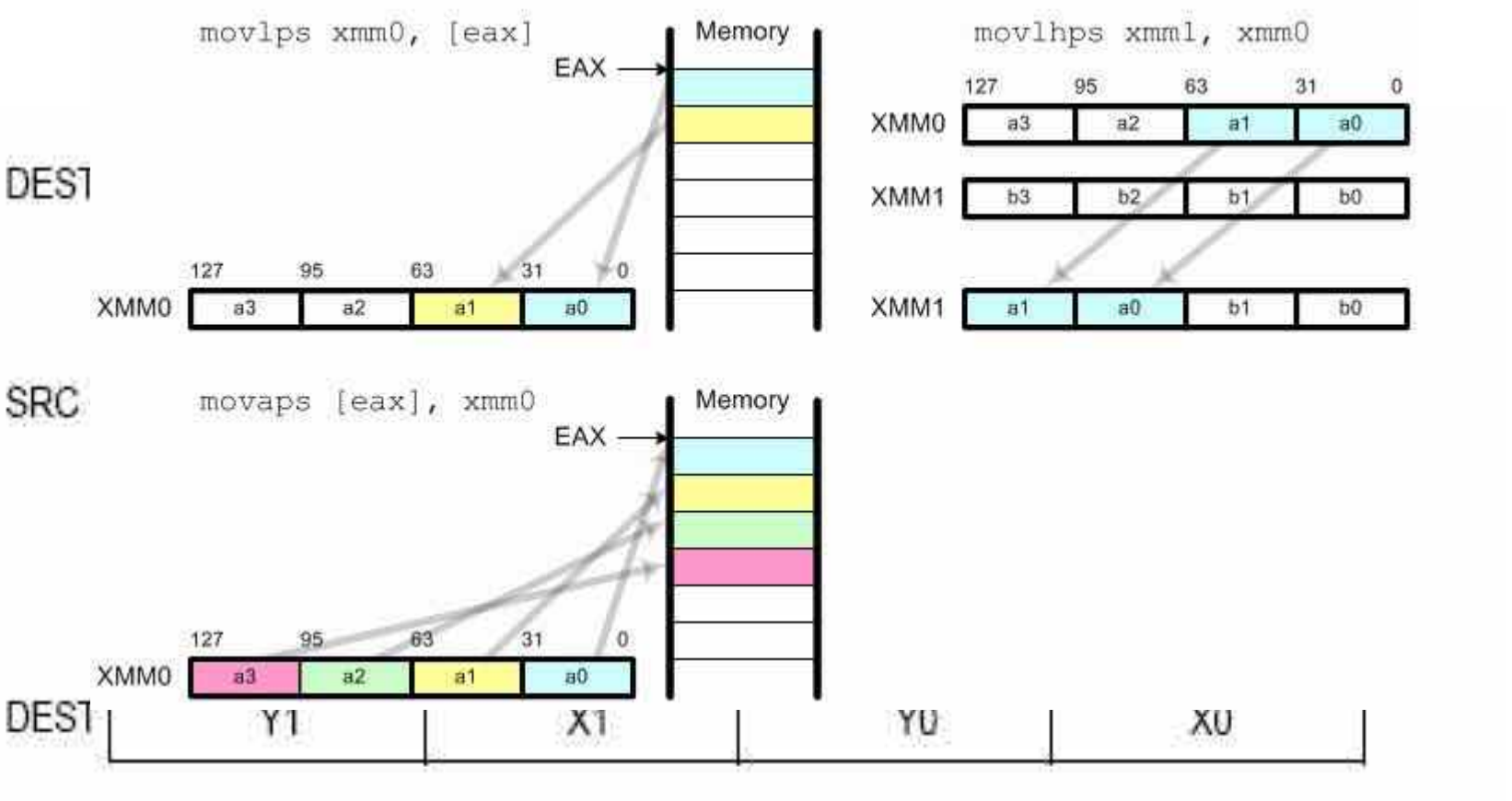
Rotate with SHUFPS



SSE Unpack Shuffle (`UNPCKLPS and UNPCKHPS`)

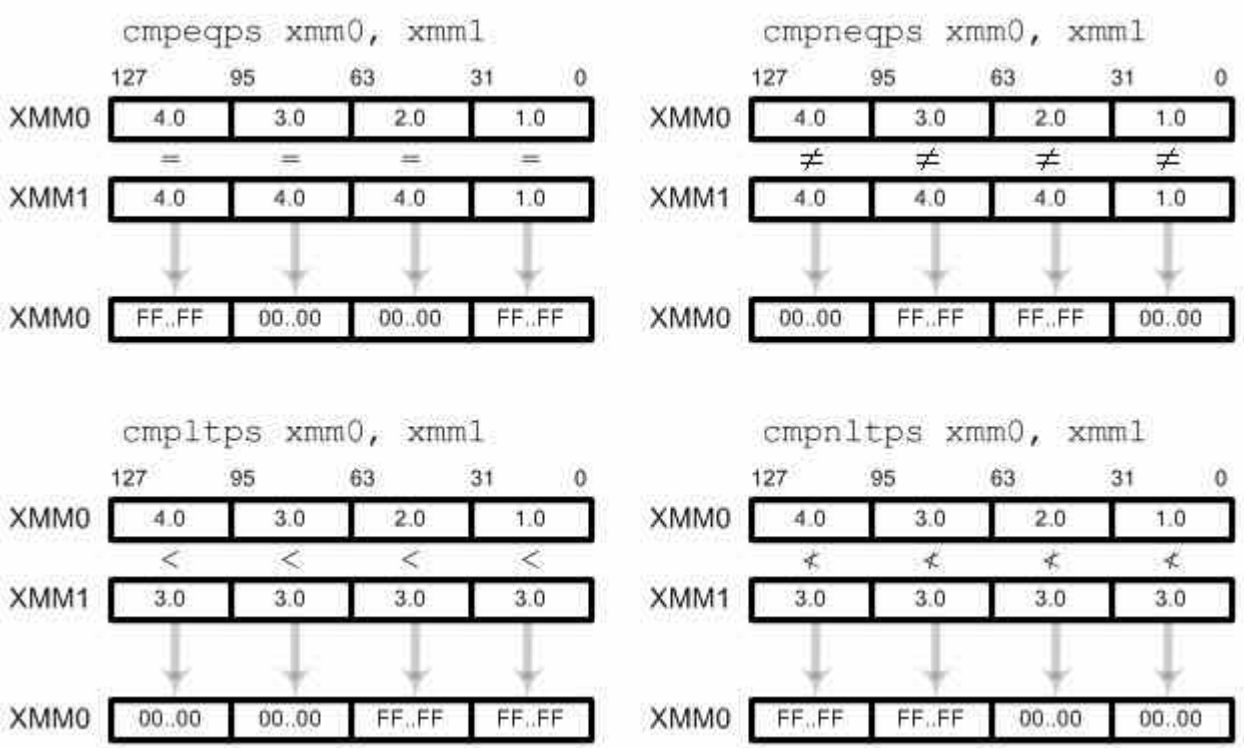


SSE MOVLPS, MOVLHPS, MOVAPS



Many types: SS, LPS, HPS, APS, UPS, HLPS, LHPS

SSE Comparison for Single-precision FP



Two versions: packed (ps) and scalar (ss)

`CMPEQPS, CMPNEQPS, CMPLTPS, CMPNLTPS, CMPLEPS, CMPNLEPS`

`CMPEQSS, CMPNEQSS, CMPLTSS, CMPNLTSS, CMPLESS, CMPNLESS`

SSE Instruction Set (Floating-point Instructions)

- Memory-to-Register / Register-to-Memory / Register-to-Register data movement
  - Scalar – MOVSS
  - Packed – MOVAPS, MOVUPS, MOVLPS, MOVHPS, MOVLHPS, MOVHLPS
- Arithmetic
  - Scalar – ADDSS, SUBSS, MULSS, DIVSS, RCPSS, SQRTSS, MAXSS, MINSS, RSQRTSS
  - Packed – ADDPS, SUBPS, MULPS, DIVPS, RCPPS, SQRTPS, MAXPS, MINPS, RSQRTPS
- Compare
  - Scalar – CMPSS, COMISS, UCOMISS
  - Packed – CMPPS
- Data shuffle and unpacking
  - Packed – SHUFPS, UNPCKHPS, UNPCKLPS
- Data-type conversion
  - Scalar – CVTSI2SS, CVTSS2SI, CVTTSS2SI
  - Packed – CVTPI2PS, CVTPS2PI, CVTTPS2PI
- Bitwise logical operations
  - Packed – ANDPS, ORPS, XORPS, ANDNPS

SSE Instruction Set (Integer and Other)

Integer instructions

- Arithmetic
  - PMULHUW, PSADBW, PAVGB, PAVGW, PMAXUB, PMINUB, PMAXSW, PMINSW
- Data movement
  - PEXTRW, PINSRW
- Other
  - PMOVMSKB, PSHUFW

Other instructions

- MXCSR management
  - LDMXCSR, STMXCSR
- Cache and Memory management
  - PREFETCH0, PREFETCH1, PREFETCHNTA
  - MOVNTQ, MOVNTPS, MASKMOVQ, SFENCE

SSE Example: Packed Floating-point Addition

```
void add(float *a, float *b, float *c) {
  for (int i = 0; i < 4; i++)
    c[i] = a[i] + b[i];
}
```

movaps: move aligned packed SP FP
addps: add packed SP FP

```
__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
}
```

SSE Example: PS Addition with Intrinsics

```
_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);
```

SSE Example: Cross Product (C code)

```
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] ) );
}
```

SSE Example: Cross Product

```
__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) );

__m128 xb = _mm_mul_ps( a , b ); // multiply

return _mm_sub_ps( xa , xb ); // subtract
}
```

SSE Example: Dot Product

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

SSE 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,  x+z
                   ;                   x0*xc+z0*zc

movaps xmm2, xmm0
shufps xmm2, xmm2, 55h ; xmm2=DC, DC,  DC,y0*yc
addps  xmm1, xmm2  ; xmm1 = DC,   DC,   DC,x+y+z
                   ;            x0*xc+y0*yc+z0*zc
```

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

SSE 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)…
```

SSE Example: Dot Product (SoA), Intrinsics

```
__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); // x1 * x2
__m128 t2 = _mm_mul_ps(y1, y2); // y1 * y2
       t1 = _mm_add_ps(t1, t2); // x + y
       t2 = _mm_mul_ps(z1, z2); // z1 * z2
       t1 = _mm_add_ps(t1, t2); // x + y + z

_mm_store_ps(output, t1);
```

SSE 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.

SSE Example: Dot Product with Prefetch

```
__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);

_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);
_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);

// 1.5x speedup
```

…

SSE Example: Exponential Function

```
float result = coeff[8] * x;
int i;

for(i = 7; i >= 2; i--) {
   result += coeff[i];
   result *= x;
}

return (result + 1) * x + 1;
```

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots$$

Example: Exponential with Intrinsics

```
int i;
__m128 X = _mm_load_ps(data);
__m128 result = _mm_mul_ps(X,coeff_sse[8]);

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);
```

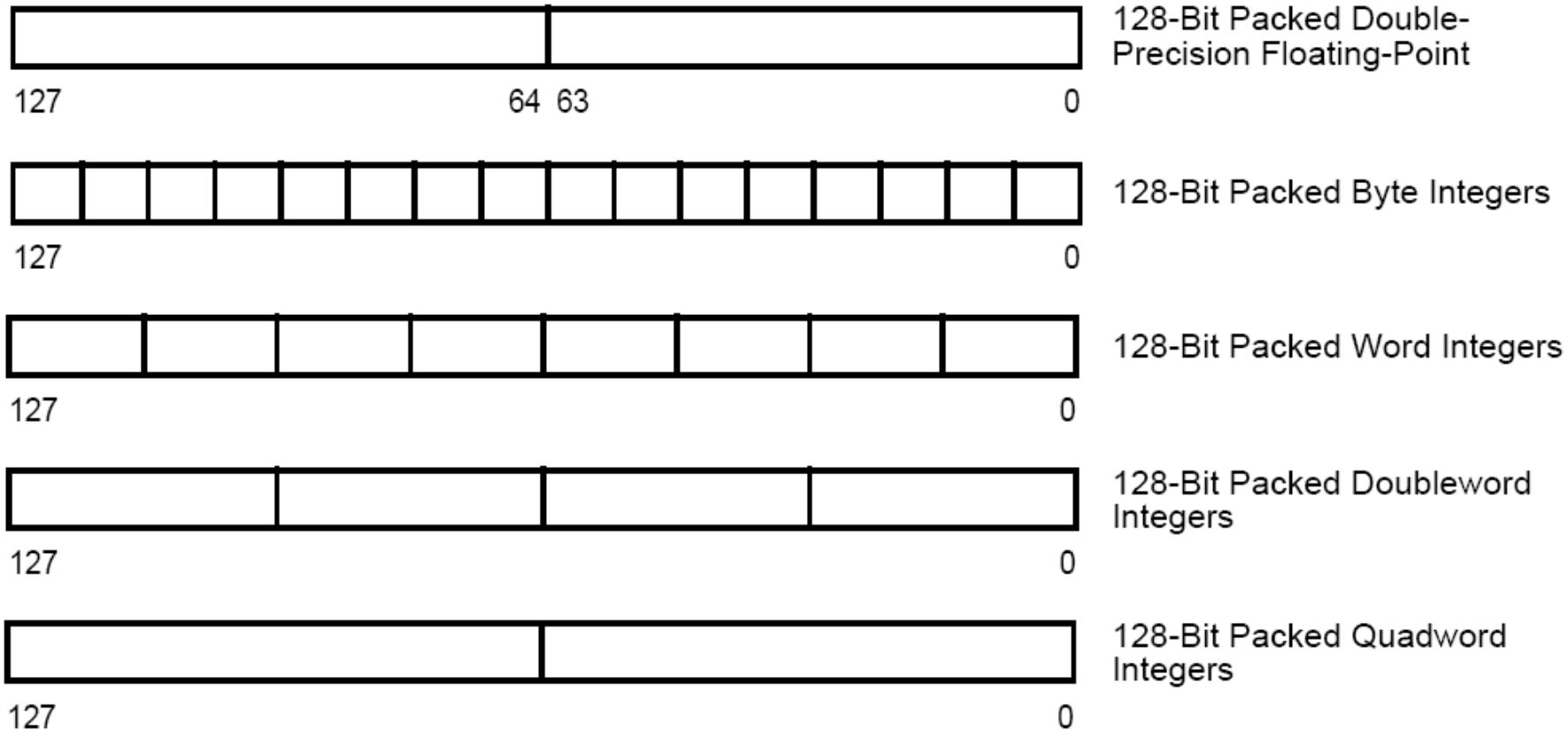
SSE2

- Introduced into the IA-32 architecture in the Pentium 4 and Intel Xeon processors in 2001
- Allowing advanced graphics such as 3-D graphics, video decoding/encoding, speech recognition
- What is new?
  - SIMD operations on 128-bit packed double-precision FP
  - SIMD operations on 128-bit packed 64-bit integers
    - Offers more flexibility in big numbers
- 144 new instructions

- AMD didn't support SSE2 until 2003, with their Opteron and Athlon64 processors

SSE2 Features

- Add data types and instructions for them



- Programming environment unchanged (also packed and scalar)

SSE2 Instructions (ARITHMETIC)

addpd - Adds 2 64bit doubles.
addsd - Adds bottom 64bit doubles.
subpd - Subtracts 2 64bit doubles.
subsd - Subtracts bottom 64bit doubles.
mulpd - Multiplies 2 64bit doubles.
mulsd - Multiplies bottom 64bit doubles.
divpd - Divides 2 64bit doubles.
divsd - Divides bottom 64bit doubles.
maxpd - Gets largest of 2 64bit doubles for 2 sets.
maxsd - Gets largest of 2 64bit doubles to bottom set.
minpd - Gets smallest of 2 64bit doubles for 2 sets.
minsd - Gets smallest of 2 64bit values for bottom set.
paddb - Adds 16 8bit integers.
paddw - Adds 8 16bit integers.
paddd - Adds 4 32bit integers.
paddq - Adds 2 64bit integers.
paddsb - Adds 16 8bit integers with saturation.
paddsw - Adds 8 16bit integers using saturation.
paddusb - Adds 16 8bit unsigned integers using saturation.
paddusw - Adds 8 16bit unsigned integers using saturation.
psubb - Subtracts 16 8bit integers.
psubw - Subtracts 8 16bit integers.
psubd - Subtracts 4 32bit integers.
psubq - Subtracts 2 64bit integers.
psubsb - Subtracts 16 8bit integers using saturation.
psubsw - Subtracts 8 16bit integers using saturation.
psubusb - Subtracts 16 8bit unsigned integers using saturation.
psubusw - Subtracts 8 16bit unsigned integers using saturation.
pmaddwd - Multiplies 16bit integers into 32bit results and adds results.
pmulhw - Multiplies 16bit integers and returns the high 16bits of the result.
pmullw - Multiplies 16bit integers and returns the low 16bits of the result.
pmuludq - Multiplies 2 32bit pairs and stores 2 64bit results.
rcpps - Approximates the reciprocal of 4 32bit singles.
rcpss - Approximates the reciprocal of bottom 32bit single.
sqrtpd - Returns square root of 2 64bit doubles.
sqrtsd - Returns square root of bottom 64bit double.

SSE2 Instructions (Logic)

andnpd - Logically NOT ANDs 2 64bit doubles.
andnps - Logically NOT ANDs 4 32bit singles.
andpd - Logically ANDs 2 64bit doubles.
pand - Logically ANDs 2 128bit registers.
pandn - Logically Inverts the first 128bit operand and ANDs with the second.
por - Logically ORs 2 128bit registers.
pslldq - Logically left shifts 1 128bit value.
psllq - Logically left shifts 2 64bit values.
pslld - Logically left shifts 4 32bit values.
psllw - Logically left shifts 8 16bit values.
psrad - Arithmetically right shifts 4 32bit values.
psraw - Arithmetically right shifts 8 16bit values.
psrldq - Logically right shifts 1 128bit values.
psrlq - Logically right shifts 2 64bit values.
psrld - Logically right shifts 4 32bit values.
psrlw - Logically right shifts 8 16bit values.
pxor - Logically XORs 2 128bit registers.
orpd - Logically ORs 2 64bit doubles.
xorpd - Logically XORs 2 64bit doubles.

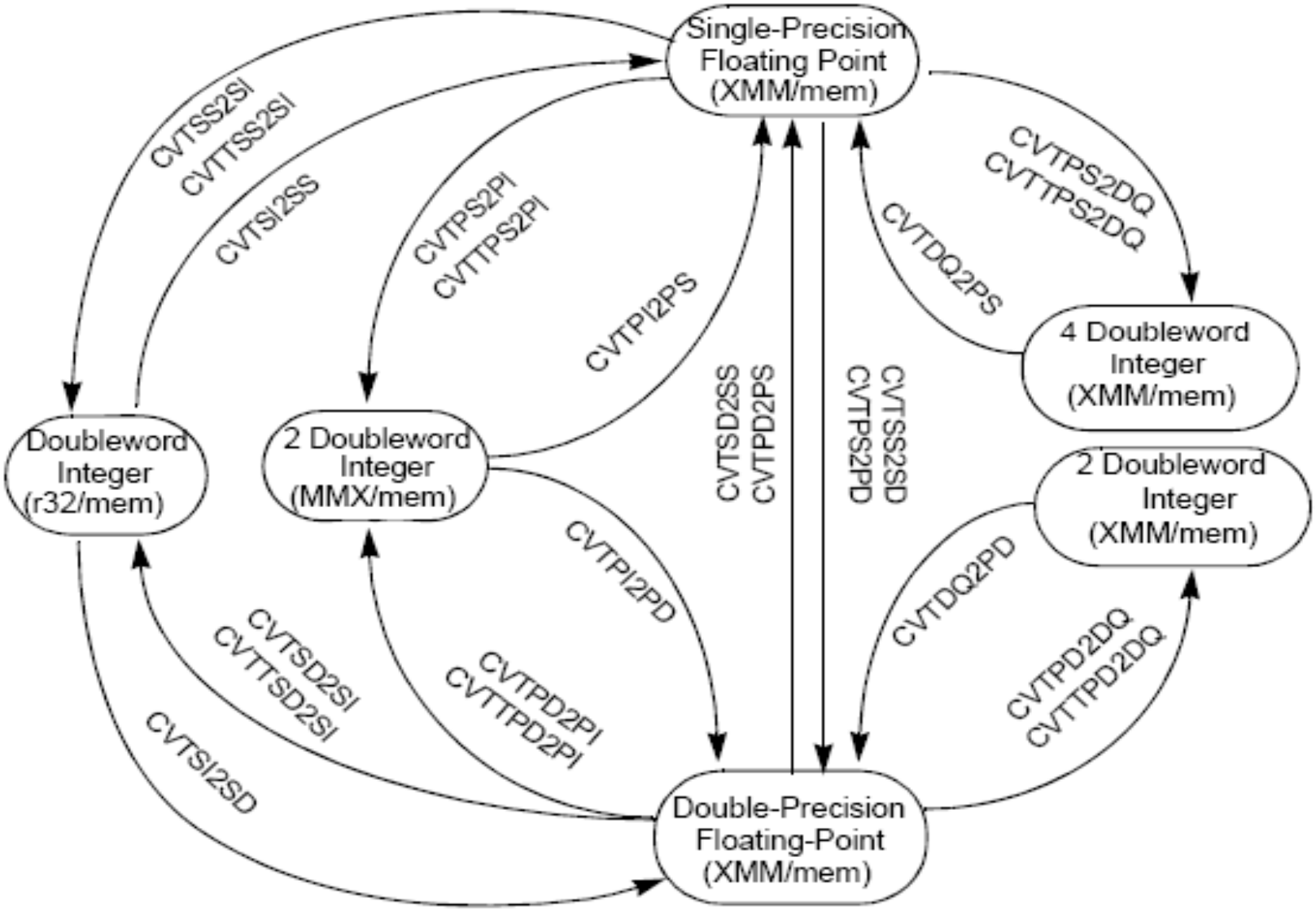
SSE2 Instructions (Compare)

cmppd - Compares 2 pairs of 64bit doubles.
cmpsd - Compares bottom 64bit doubles.
comisd - Compares bottom 64bit doubles and stores result in EFLAGS.
ucomisd - Compares bottom 64bit doubles and stores result in EFLAGS. (QNaNs don't throw exceptions with ucomisd, unlike comisd.
pcmpxxb - Compares 16 8bit integers.
pcmpxxw - Compares 8 16bit integers.
pcmpxxd - Compares 4 32bit integers.
Compare Codes (the xx parts above):
eq - Equal to.
lt - Less than.
le - Less than or equal to.
ne - Not equal.
nlt - Not less than.
nle - Not less than or equal to.
ord - Ordered.
unord - Unordered.

SSE2 Instructions (Conversion)

cvtdq2pd - Converts 2 32bit integers into 2 64bit doubles.
cvtdq2ps - Converts 4 32bit integers into 4 32bit singles.
cvtpd2pi - Converts 2 64bit doubles into 2 32bit integers in an MMX register.
cvtpd2dq - Converts 2 64bit doubles into 2 32bit integers in the bottom of an XMM register.
cvtpd2ps - Converts 2 64bit doubles into 2 32bit singles in the bottom of an XMM register.
cvtpi2pd - Converts 2 32bit integers into 2 32bit singles in the bottom of an XMM register.
cvtps2dq - Converts 4 32bit singles into 4 32bit integers.
cvtps2pd - Converts 2 32bit singles into 2 64bit doubles.
cvtsd2si - Converts 1 64bit double to a 32bit integer in a GPR.
cvtsd2ss - Converts bottom 64bit double to a bottom 32bit single. Tops are unchanged.
cvtsi2sd - Converts a 32bit integer to the bottom 64bit double.
cvtsi2ss - Converts a 32bit integer to the bottom 32bit single.
cvtss2sd - Converts bottom 32bit single to bottom 64bit double.
cvtss2si - Converts bottom 32bit single to a 32bit integer in a GPR.
cvttpd2pi - Converts 2 64bit doubles to 2 32bit integers using truncation into an MMX register.
cvttpd2dq - Converts 2 64bit doubles to 2 32bit integers using truncation.
cvttps2dq - Converts 4 32bit singles to 4 32bit integers using truncation.
cvttps2pi - Converts 2 32bit singles to 2 32bit integers using truncation into an MMX register.
cvttsd2si - Converts a 64bit double to a 32bit integer using truncation into a GPR.
cvttss2si - Converts a 32bit single to a 32bit integer using truncation into a GPR.

SSE2 Instructions



SSE2 Instructions

Load/Store:

(is "minimize cache pollution" the same as "without using cache"??)
movq - Moves a 64bit value, clearing the top 64bits of an XMM register.
movsd - Moves a 64bit double, leaving tops unchanged if move is between two XMMregisters.
movapd - Moves 2 aligned 64bit doubles.
movupd - Moves 2 unaligned 64bit doubles.
movhpd - Moves top 64bit value to or from an XMM register.
movlpd - Moves bottom 64bit value to or from an XMM register.
movdq2q - Moves bottom 64bit value into an MMX register.
movq2dq - Moves an MMX register value to the bottom of an XMM register. Top is cleared to zero.
movntpd - Moves a 128bit value to memory without using the cache. NT is "Non Temporal."
movntdq - Moves a 128bit value to memory without using the cache.
movnti - Moves a 32bit value without using the cache.
maskmovdqu - Moves 16 bytes based on sign bits of another XMM register.
pmovmskb - Generates a 16bit Mask from the sign bits of each byte in an XMM register.

SSE2 Instructions

Shuffling:

pshufd - Shuffles 32bit values in a complex way.
pshufhw - Shuffles high 16bit values in a complex way.
pshuflw - Shuffles low 16bit values in a complex way.
unpckhpd - Unpacks and interleaves top 64bit doubles from 2 128bit sources into 1.
unpcklpd - Unpacks and interleaves bottom 64bit doubles from 2 128 bit sources into 1.
punpckhbw - Unpacks and interleaves top 8 8bit integers from 2 128bit sources into 1.
punpckhwd - Unpacks and interleaves top 4 16bit integers from 2 128bit sources into 1.
punpckhdq - Unpacks and interleaves top 2 32bit integers from 2 128bit sources into 1.
punpckhqdq - Unpacks and interleaces top 64bit integers from 2 128bit sources into 1.
punpcklbw - Unpacks and interleaves bottom 8 8bit integers from 2 128bit sources into 1.
punpcklwd - Unpacks and interleaves bottom 4 16bit integers from 2 128bit sources into 1.
punpckldq - Unpacks and interleaves bottom 2 32bit integers from 2 128bit sources into 1.
punpcklqdq - Unpacks and interleaces bottom 64bit integers from 2 128bit sources into 1.
packssdw - Packs 32bit integers to 16bit integers using saturation.
packsswb - Packs 16bit integers to 8bit integers using saturation.
packuswb - Packs 16bit integers to 8bit unsigned integers unsing saturation.

Cache Control:

clflush - Flushes a Cache Line from all levels of cache.
lfence - Guarantees that all memory loads issued before the lfence instruction are completed before anyloads after the lfence instruction.
mfence - Guarantees that all memory reads and writes issued before the mfence instruction are completed before any reads or writes after the mfence instruction.
pause - Pauses execution for a set amount of time.

SSE3/SSSE3/SSE4

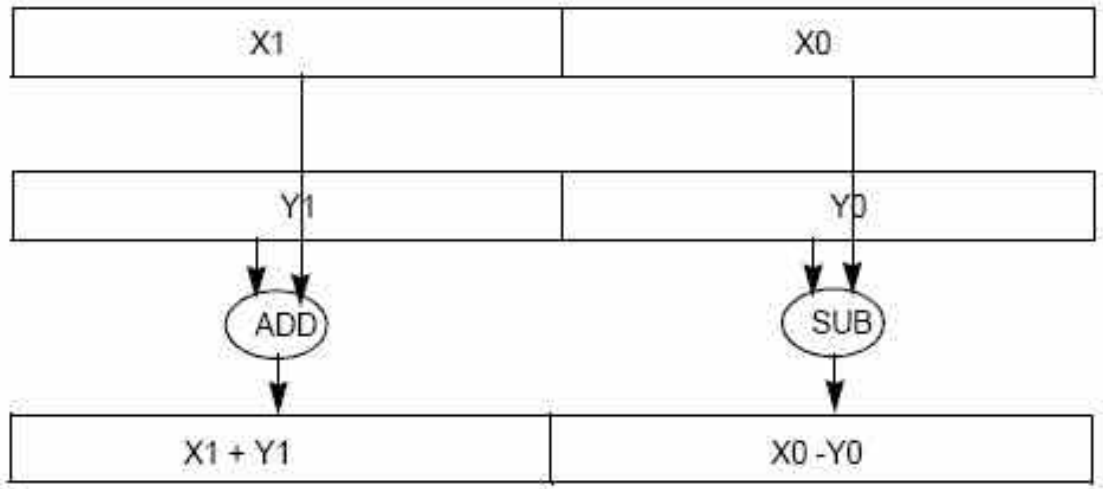
- Introduced for Pentium 4 processor supporting Hyper-Threading Technology in 2004.
- The Intel Xeon processor 5100 series, Intel Core 2 processor families introduced Supplemental Streaming SIMD Extensions 3 (SSSE3)
- SSE4 are introduced in Intel processor generations built from 45nm process technology in 2006
- SSE3/SSSE3/SSE4 do not introduce new data types
  - XMM registers are used to operate on packed data types
    - integer, single-precision FP, or double-precision FP

SSE3

- 13 new instructions
  - Support horizontal operations across a single register
    - Instead of down through multiple registers
  - Asymmetric processing
- Unaligned access instructions are new type of instructions
- Process control instructions to boost performance with Intel's hyper-threading feature

- AMD started to support SSE3 in 2005

SSE3 Instructions ADDSUBPD

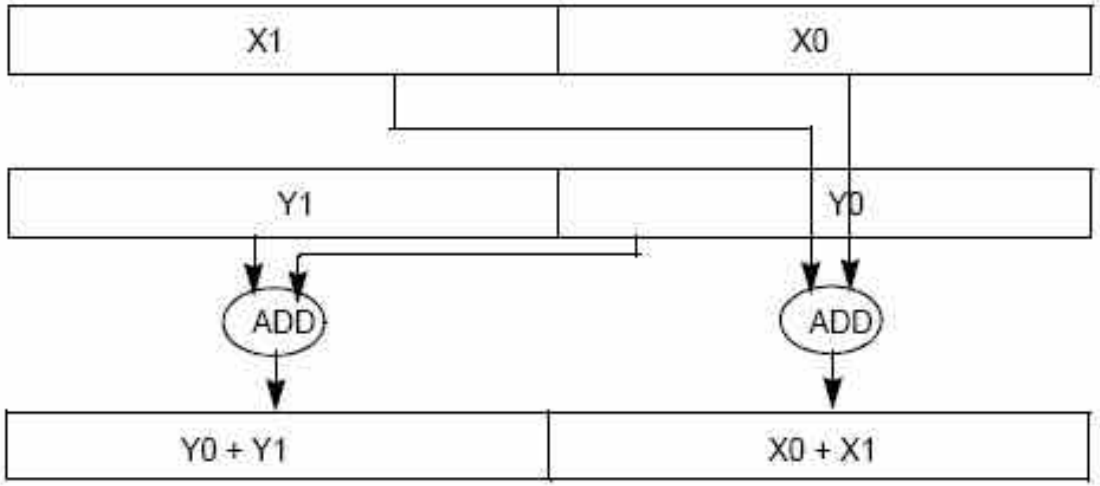


Asymmetric processing: ADDSUBPD

Add and Sub of packed double-precision FP

The second operand may be from memory

SSE3 Instructions: HADDPD



Horizontal data movement: HADDPD

Horizontal add of packed double-precision FP

The second operand may be from memory

SSE3 Instructions Summary

Arithmetic:

addsubpd – DP. Additon on higher pair, subtraction on lower pair (+, -).
addsubps – SP. Two adds and two subs interleaved (+, -, +, -).
haddpd – DP. Horizontal addition. (src1+src0, dst1+dst0)
haddps – SP. Horizontal addition. (src3+src2, src1+src0, dst3+dst2, dst1+dst0)
hsubpd – DP. Horizontal addition. (src1+src0, dst1+dst0)
hsubps – SP. Horizontal addition. (src3+src2, src1+src0, dst3+dst2, dst1+dst0)

Load/Store:

lddqu – Loads an unaligned 128bit value
movddup – Loads or move a DP into lower half and duplicate to the higher
movshdup – Duplicates the higher singles. (src3, src3, src1, src1)
movsldup – Duplicates the lower singles. (src2, src2, src0, src0)
fisttp – Converts a floating-point value to an integer using truncation

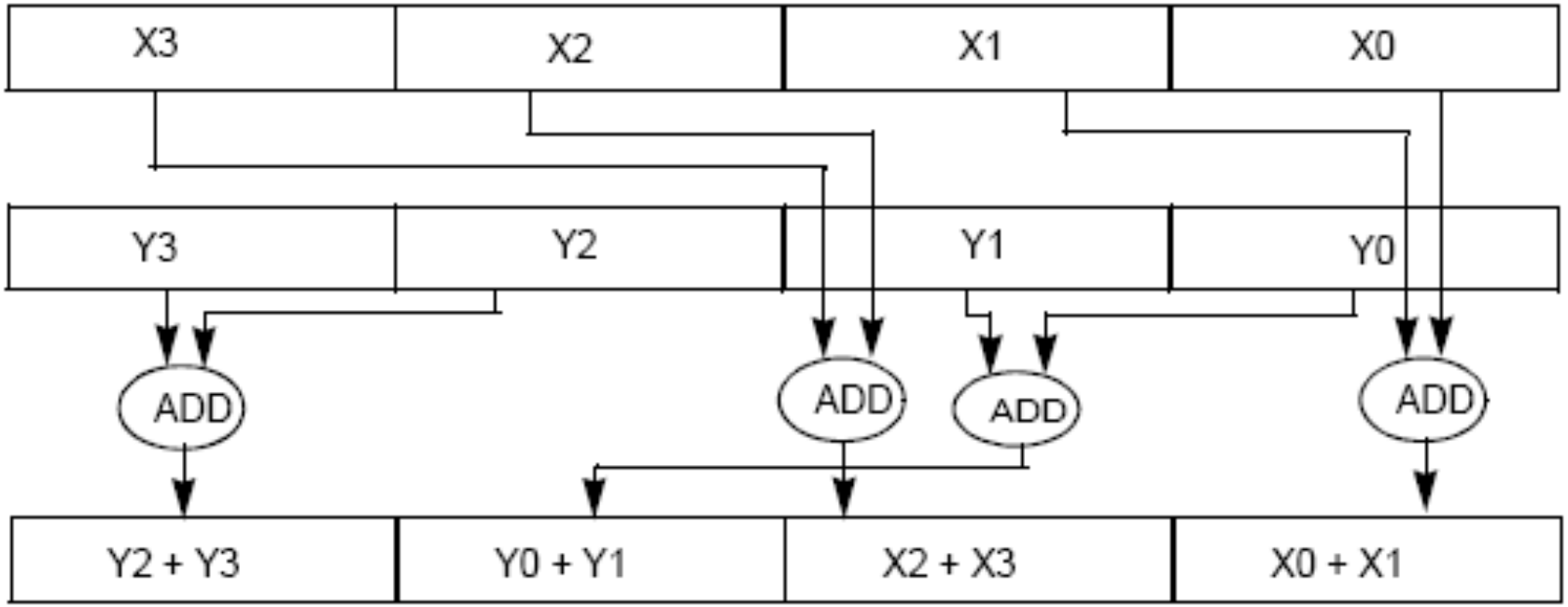
Process Control:

monitor – Sets up a region to monitor for activity
mwait – Waits until activity happens in a region specified by monitor

SSSE3

- New 32 instructions designed for to accelerate a variety of multimedia and signal processing applications
  - Only 16, for both MMX and XMM
- Integer data types include packed byte, word, or double word
- Operands can be 64 or 128 bit in MMX registers, XMM registers, or memory

SSSE3 Instructions: PHADDD



Horizontal data movement: PHADDD

Horizontal add of packed DW

The second operand may be from memory

SSSE3 Instructions Summary

- 12 for horizontal addition or subtraction operations
  - ADDW, ADDD, ADDSW, and three for SUB
- 6 for evaluating absolute values
  - PABSB, PABSW, PABSD
- 2 for multiply and add operations
  - PMADDUBSW (byte mul, add pairs, and to saturated words)
  - Speed up dot products
- 2 for packed-integer multiply operations
  - PMULHRSW (Q.15 multiplications, with rounding and scaling)
- 2 for a byte-wise, in-place shuffle
  - PSHUFB (similar to PERMUTE)
- 6 instructions negating packed integers in the destination
  - B, W, and D version
- 2 for alignment data from the composite of two operands
  - PALIGNR (similar to SHIFT PAIR)

SSE4

- SSE4 comprises of two sets of extensions
  - SSE4.1 includes 47 new instructions
    - Targets media, imaging and 3D graphics
    - Adds instructions for improving compiler vectorization
    - Significantly increases support for packed dword computation
  - SSE4.2 has 7 new instructions
    - Improves performance in string and text processing
- Registers
  - Two SSE4.2 instructions operate on general-purpose registers
  - All other instructions operate on XMM registers
    - No MMX registers

SSE4.1 Instructions Summary (1)

- Six instructions for conditional copying
  - BLENDPS, BLENDPD, BLENDVPS, BLENDVPD
  - PBLENDVB, PBLENDW
- Eight instructions expand support for packed integer MIN/MAX
  - PMINSB, PMINUW, PMINUD, PMINSD and PMAX versions
- Four instructions for floating-point rounding
  - ROUNDPS, ROUNDSS, ROUNDPD, ROUNDSD
  - One of the four rounding modes specified by an immediate
- Seven instructions for data insertion and extractions
  - INSERTPS, PINSRB, PINSRD/Q
  - EXTRACTPS, PEXTRB, PEXTRW, PEXTRD/PEXTRQ
- Twelve instructions converting packed integer format to D
  - PMOVSXBW, PMOVZXBW, also BQ, WD, WQ, DQ
  - Sign and zero extensions

SSE 4.1 Insructions Summary (2)

- Two instructions perform packed DW multiplications
  - PMULDQ, 32-bit to 64-bit, two DW from the source are used
  - PMULLD, 32-bit to 32-bit
- Two instructions floating-point dot products
  - DPPS, dot product of PS, result in any (one or more) locations
  - DPPD, dot product of PD
- MPSADBW – Computes 8 offset sums of absolute differences
- PTEST – Compare two 128 bits values and set ZF and CF flags
- PCMPEQQ – QD equality comparison
- PACKUSDW – Signed DW to unsigned W with saturation
- MOVNTDQA – Move DQ
- PHMINPOSUW – Packed horizontal word minimum
  - Value in lower 16 bits and 3-bit index are bits 16 – 18

SSE 4.1 BLEND Instructions

- Copy PS of PD from SRC if the control bit is 1
  - PS needs four bits while PD needs only two bits
- V versions use XMM0 as control
  - Use the MSB

Instructions	Packed Double FP	Packed Single FP	Packed QWord	Packed DWord	Packed Word	Packed Byte	Blend Control
BLENDPS		X					Imm8
BLENDPD	X						Imm8
BLENDVPS		X		X(1)			XMM0
BLENDVPD	X		X(1)				XMM0
PBLENDVB			(2)	(2)	(2)	X	XMM0
PBLENDW			X	X	X		Imm8

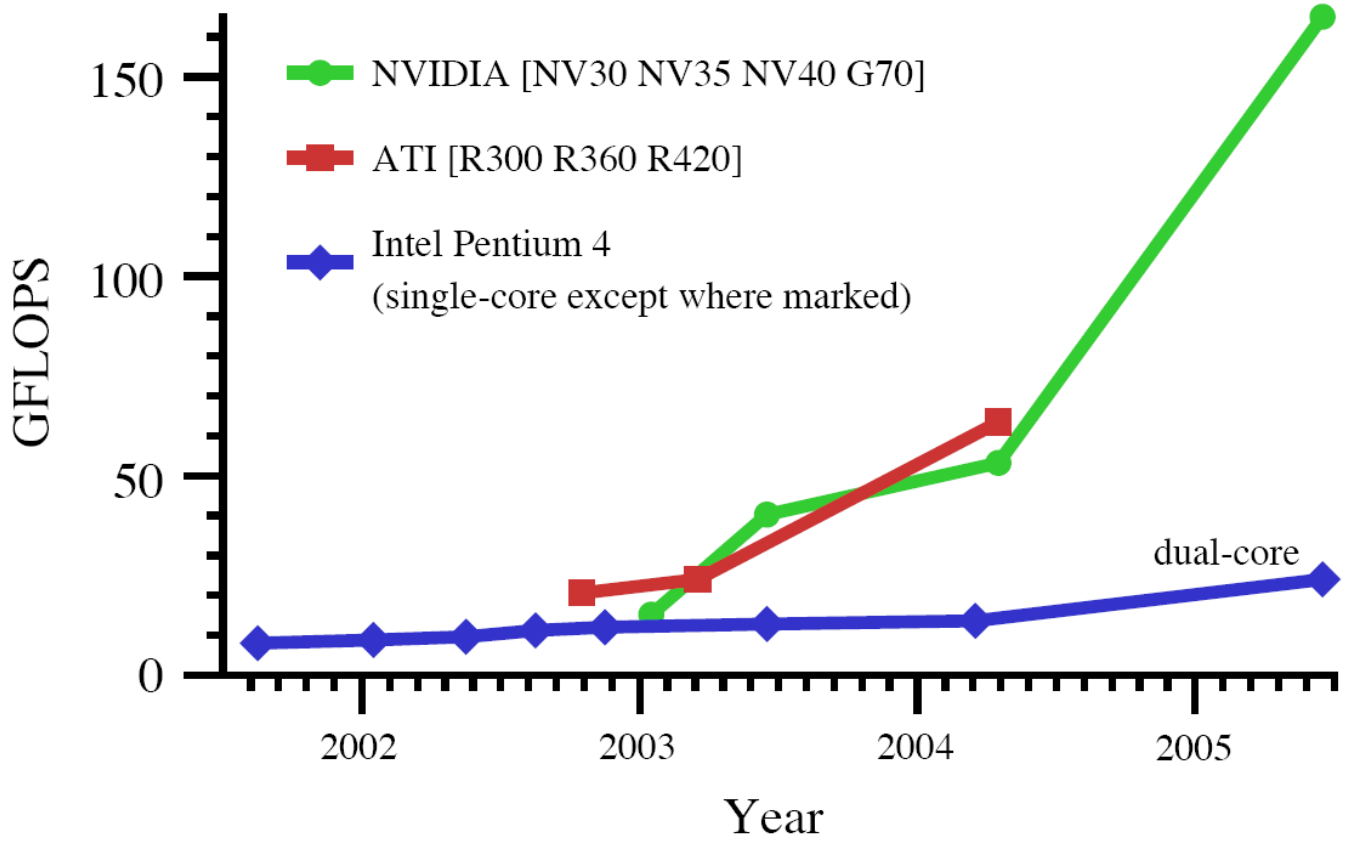
SSE4.2 Instructions

- CRC32 – Use the polynomial 0x11EDC6F41
  - R32 or R64 mode
- Four string comparison instructions
  - PCMPESTRI
  - PCMPESTRM
  - PCMPISTRI
  - PCMPISTRM
- PCMPGTQ – Compare packed QW for greater than

- POPCNT – Population count
- LZCNT – Leading zero count

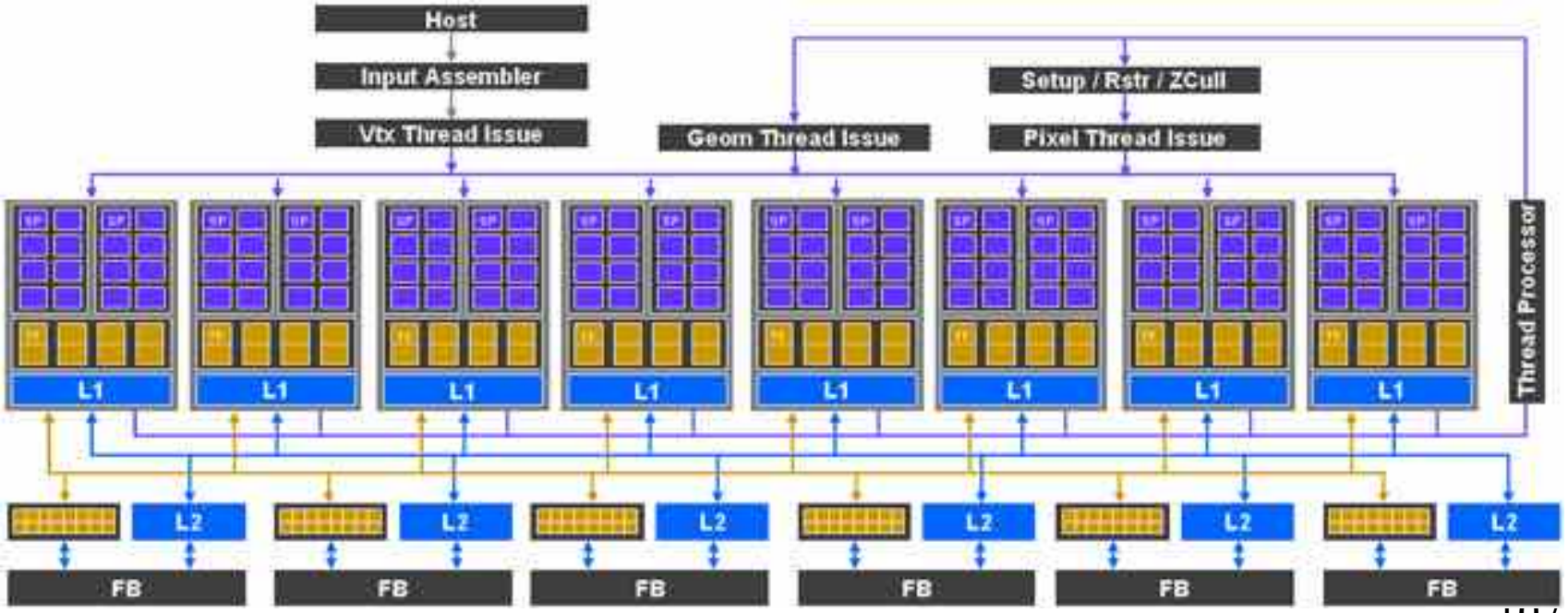
Other SIMD architectures

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



NVidia GeForce 8800, 2006

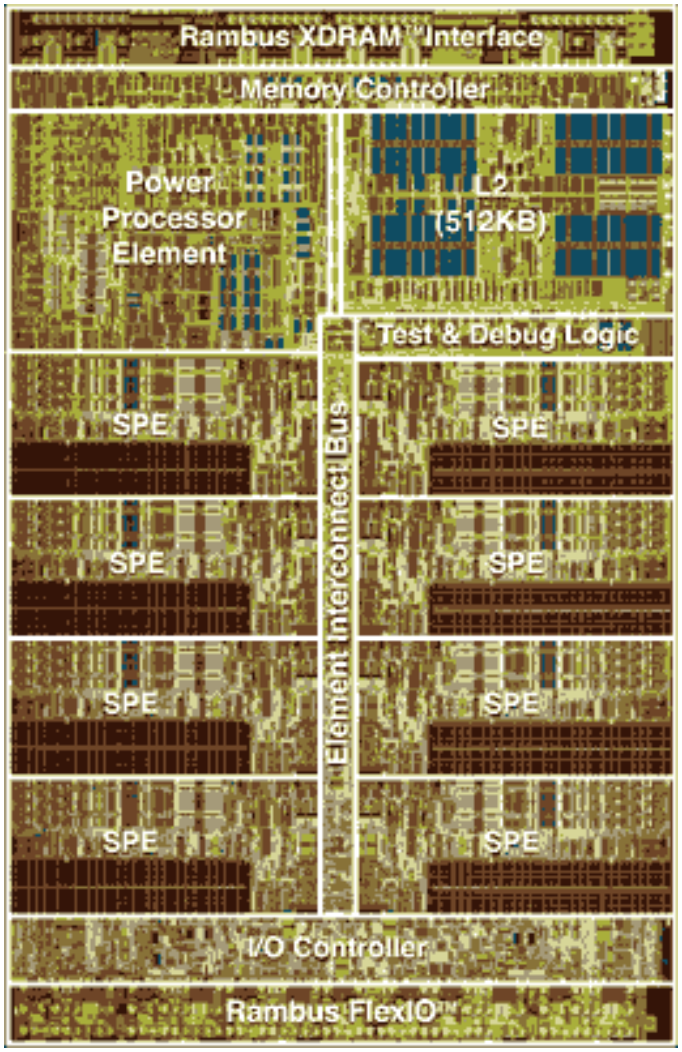
- Each GeForce 8800 GPU stream processor is a fully generalized, fully decoupled, scalar, processor that supports IEEE 754 floating point precision.
- Up to 128 stream processors

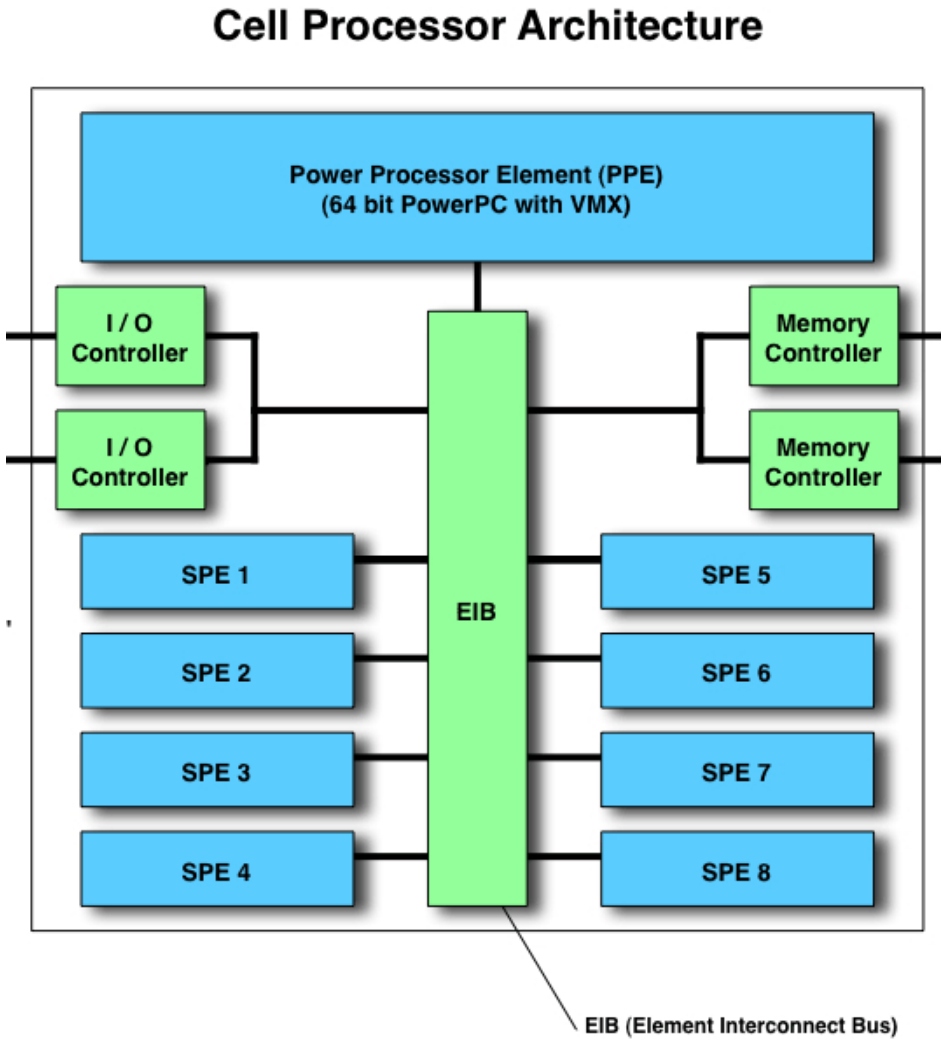


Cell processor

- 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





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