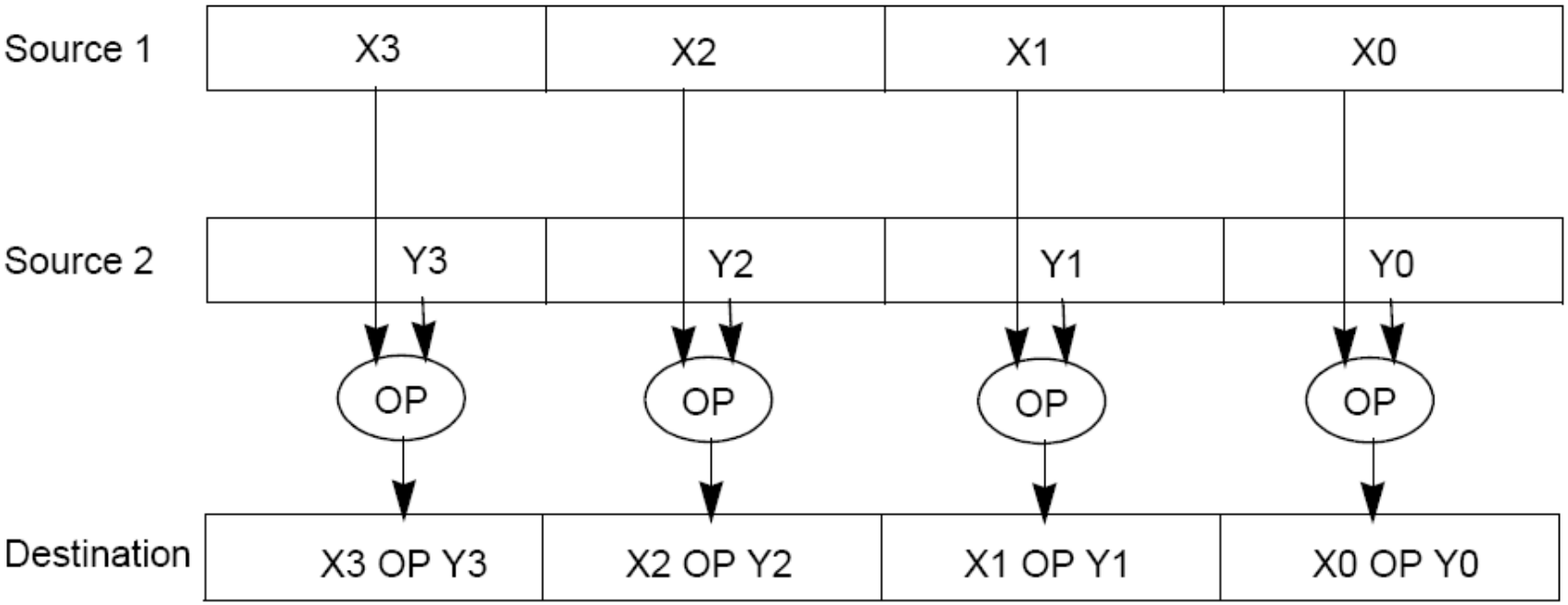
Intel SIMD extensions

Performance boost

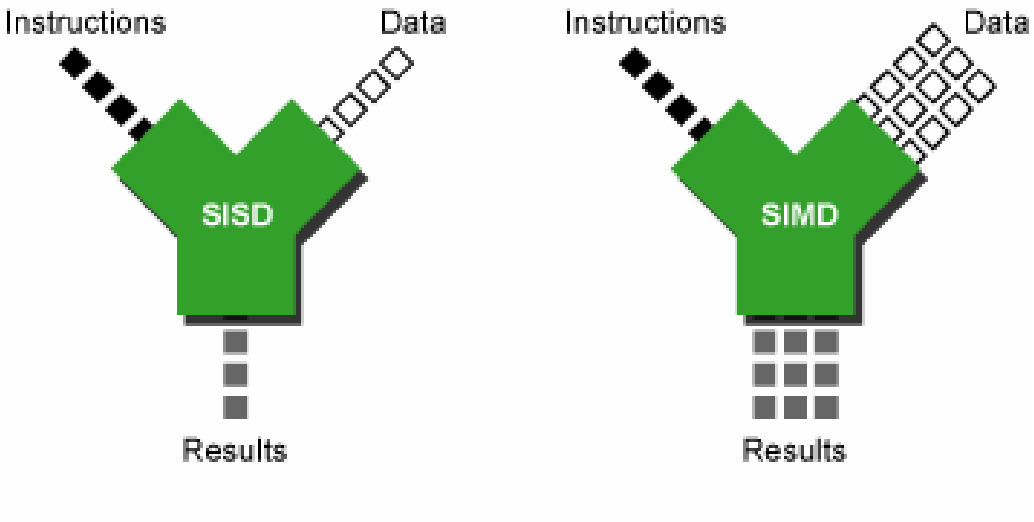
- Architecture improvements (such as pipeline/cache/SIMD) are more significant
- Intel analyzed multimedia applications and found they share the following characteristics:
  - Small native data types (8-bit pixel, 16-bit audio)
  - Recurring operations
  - Inherent parallelism

SIMD

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



SISD/SIMD


Instructions
Data
SISD
Results
Instructions
Data
SIMD
Results

Intel SIMD development

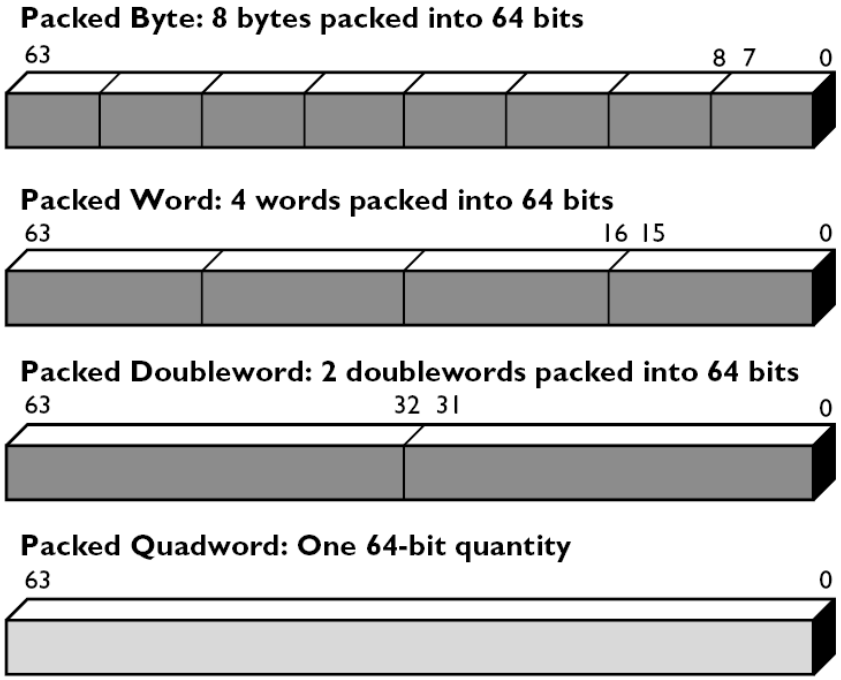
- MMX (Multimedia Extension) was introduced in 1996 (Pentium with MMX and Pentium II).
- SSE (Streaming SIMD Extension) was introduced with Pentium III.
- SSE2 was introduced with Pentium 4.
- SSE3 was introduced with Pentium 4 supporting hyper-threading technology. SSE3 adds 13 more instructions.
-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

MMX

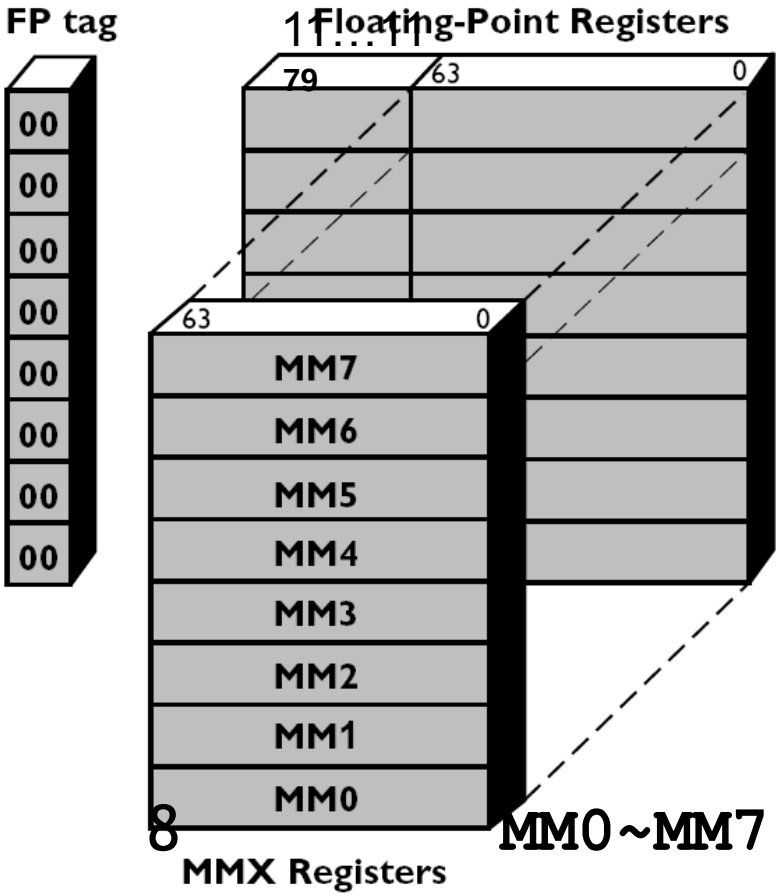
- After analyzing a lot of existing applications such as graphics, MPEG, music, speech recognition, game, image processing, they found that many multimedia algorithms execute the same instructions on many pieces of data in a large data set.
- Typical elements are small, 8 bits for pixels, 16 bits for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



MMX integration into IA



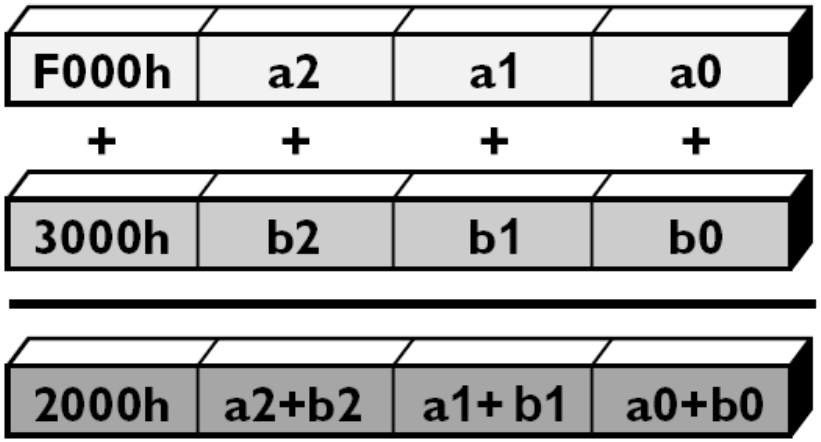
- To simplify the design and to avoid modifying the operating system to preserve additional state through context switches, MMX re-uses the existing eight IA-32 FPU registers.
- This made it difficult to work with floating point and SIMD data at the same time.
- To maximize performance, programmers must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

MMX instructions

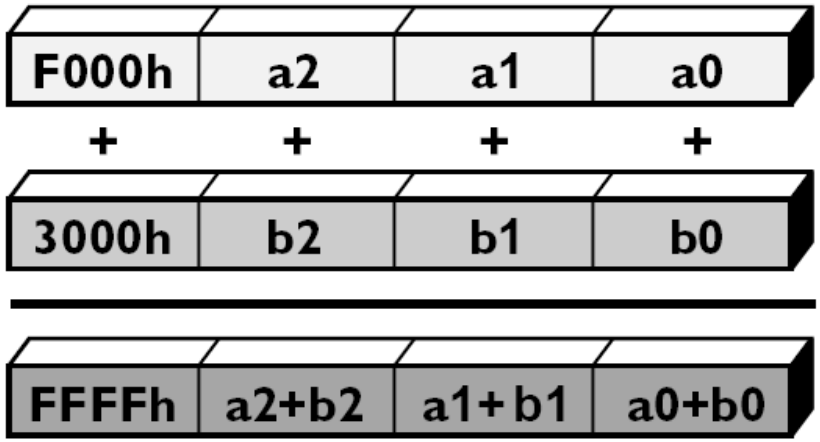
- 57 MMX instructions are defined to perform the parallel operations on multiple data elements packed into 64-bit data types.
- These include **`add`**, **`subtract`**, **`multiply`**, **`compare`**, and **`shift`**, **`data conversion`**, **`64-bit data move`**, **`64-bit logical operation`** and **`multiply-add`** 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



wrap-around



saturating

MMX instructions

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

MMX instructions

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

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

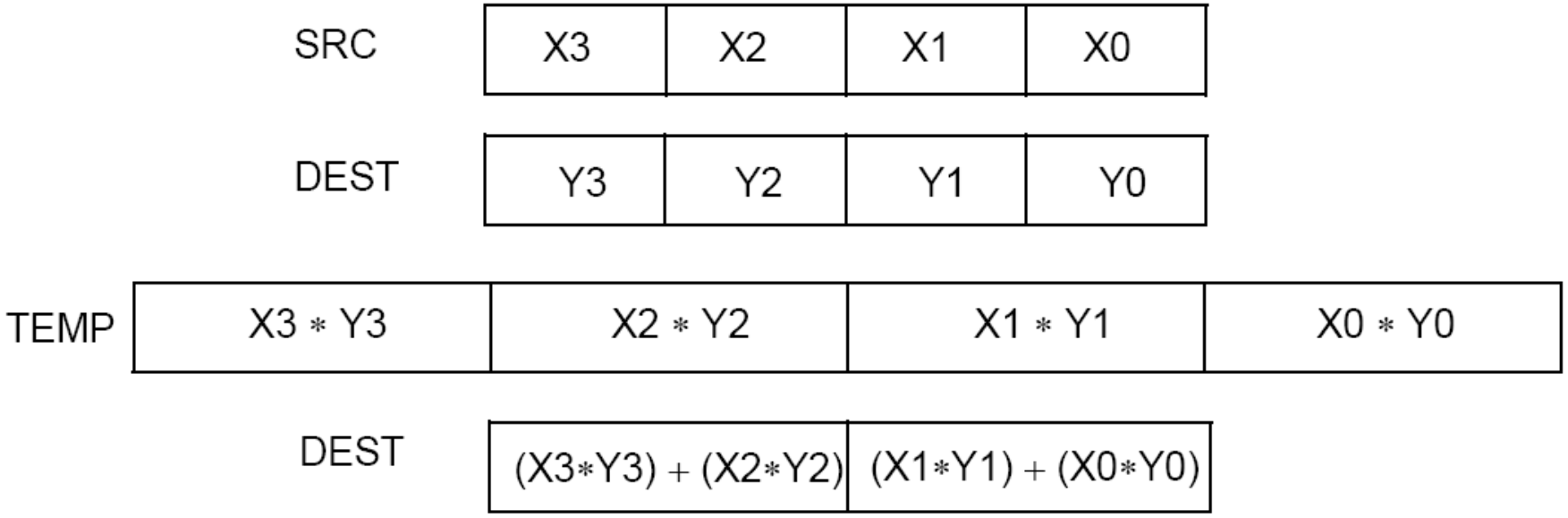
Arithmetic

- **`PADDB/PADDW/PADDD`**: add two packed numbers
- 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 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]);

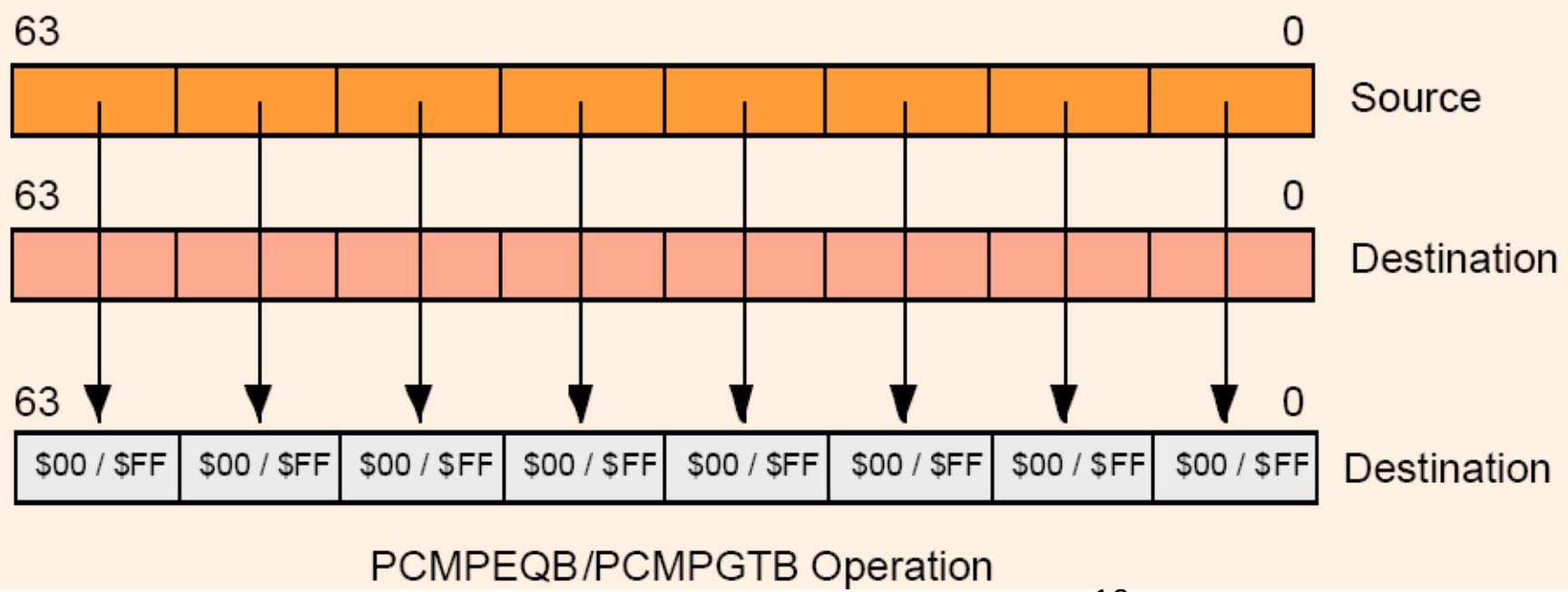


Example: add a constant to a vector

```
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
}
```

Comparison

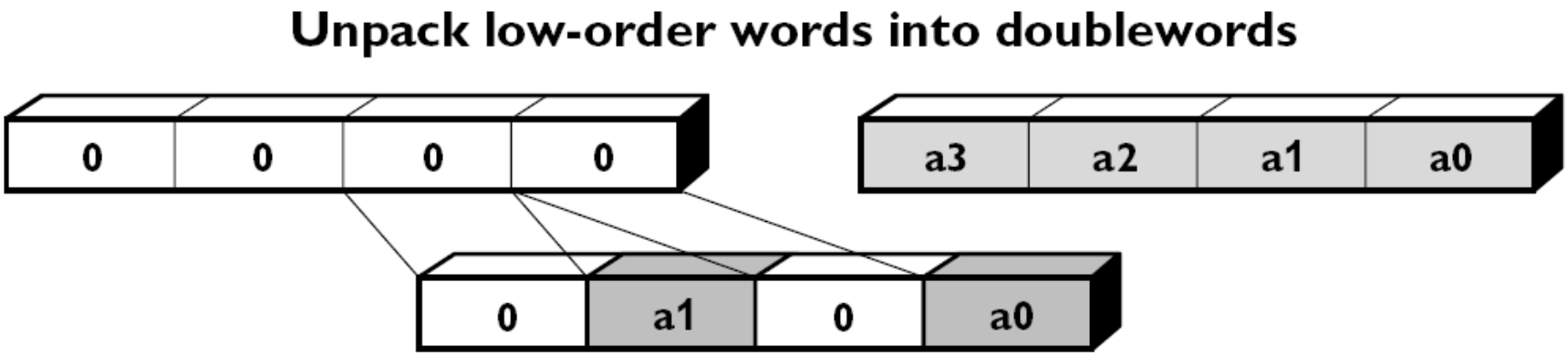
- No CFLAGS, how many flags will you need? Results are stored in destination.
- EQ/GT, no LT



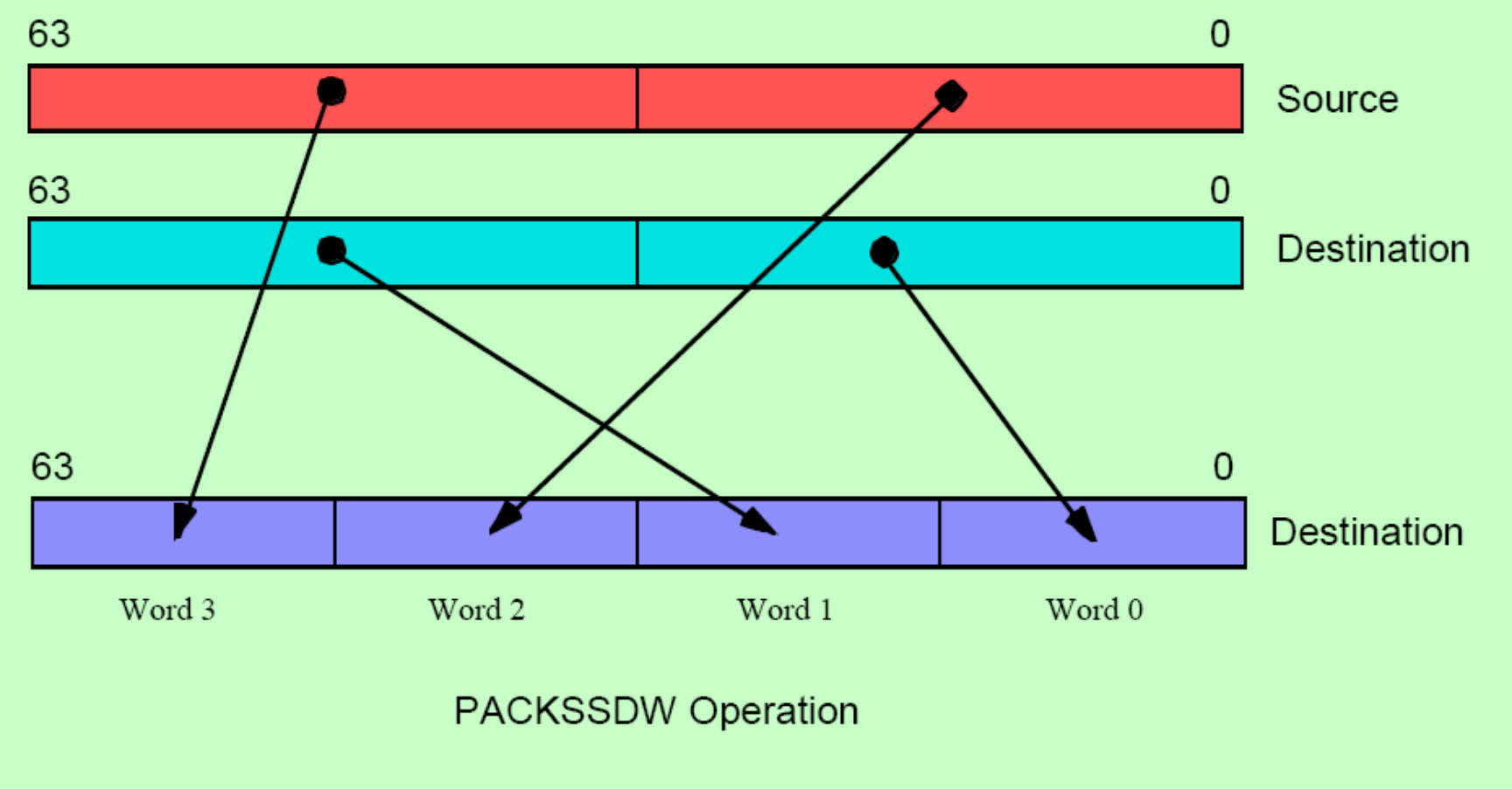
PCMPEQB/PCMPGTB Operation

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.

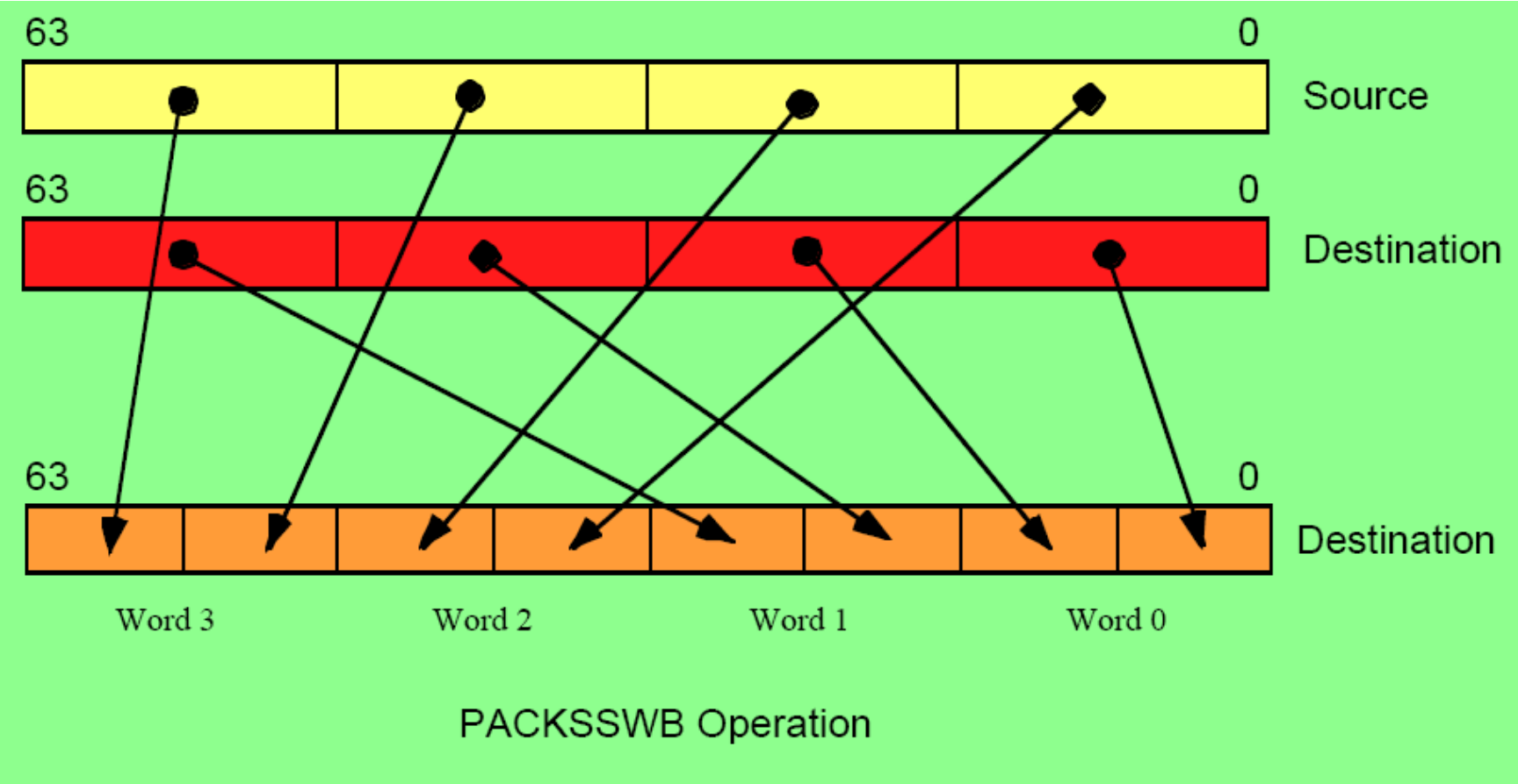


Pack with signed saturation



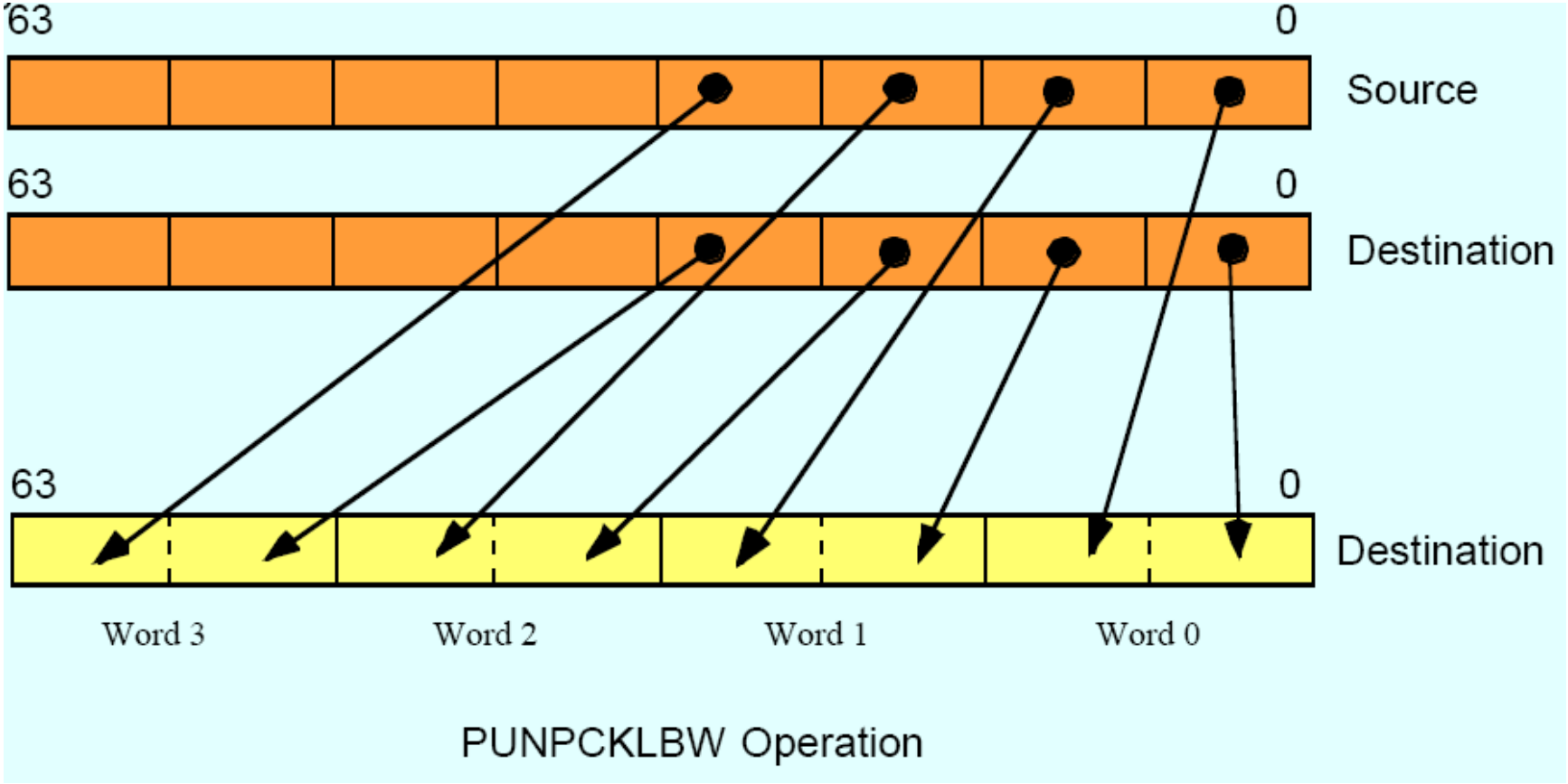
PACKSSDW mm_d, mm_s

Pack with signed saturation

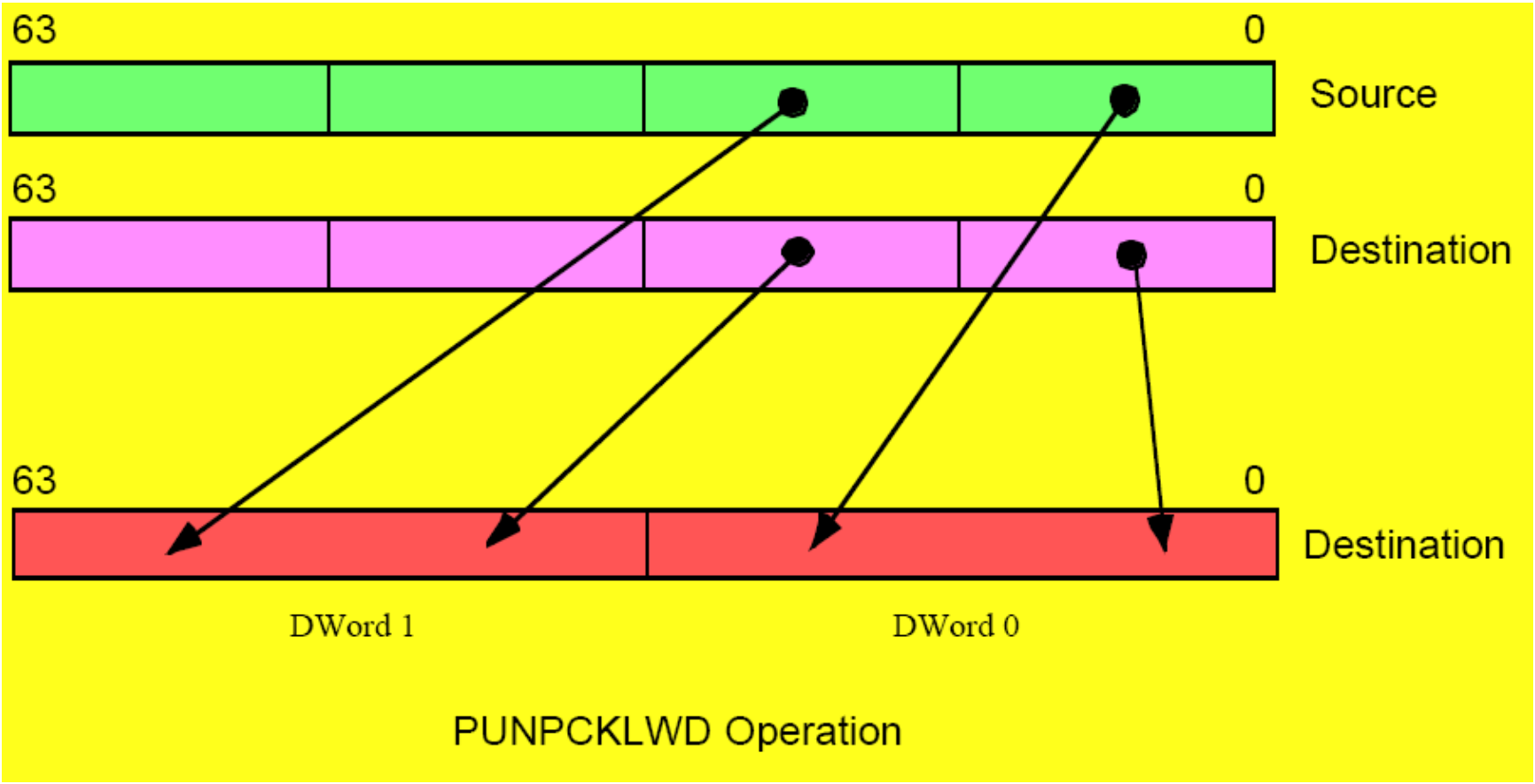


PACKSSWB mm_d, mm_s

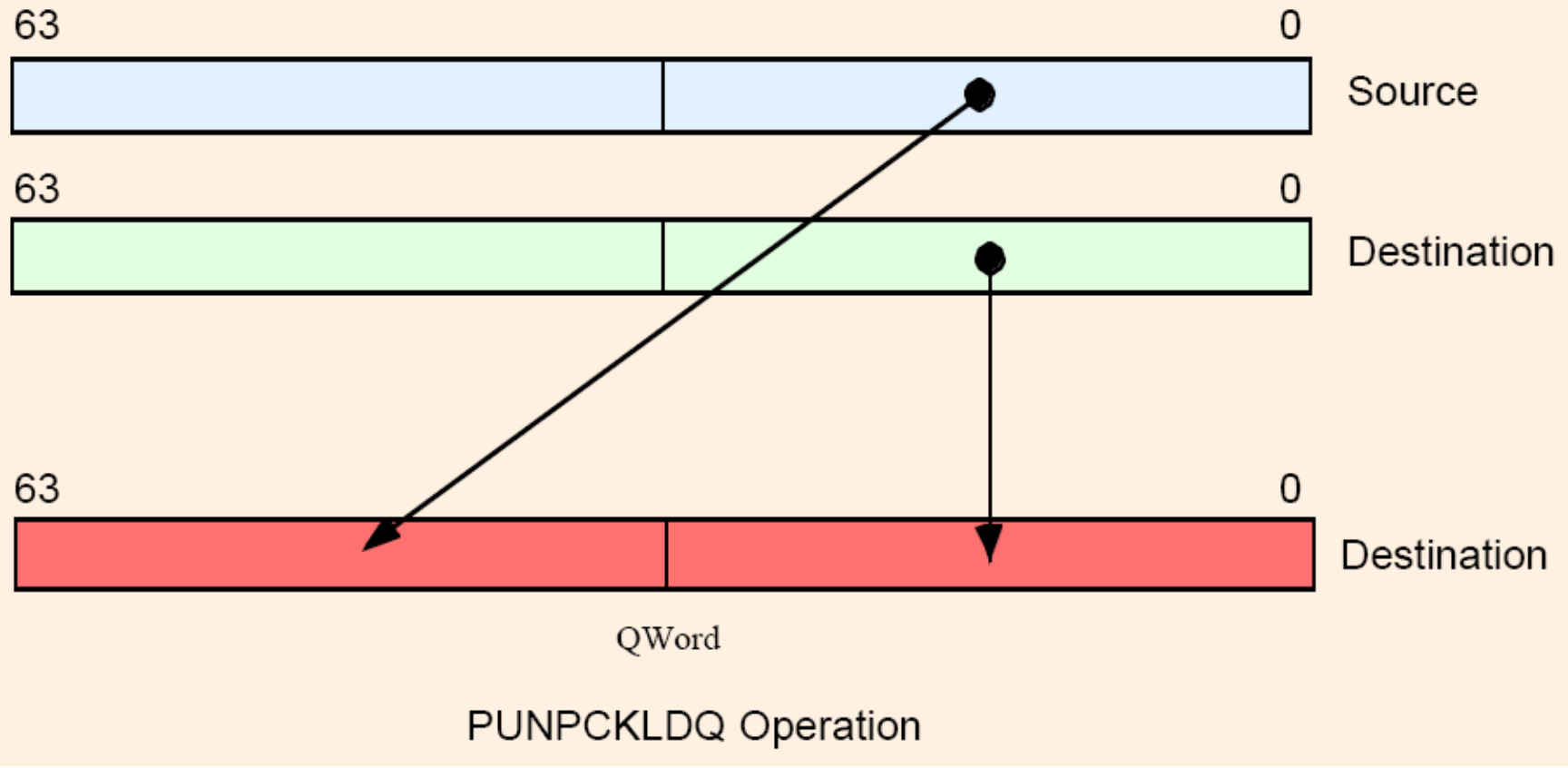
Unpack low portion



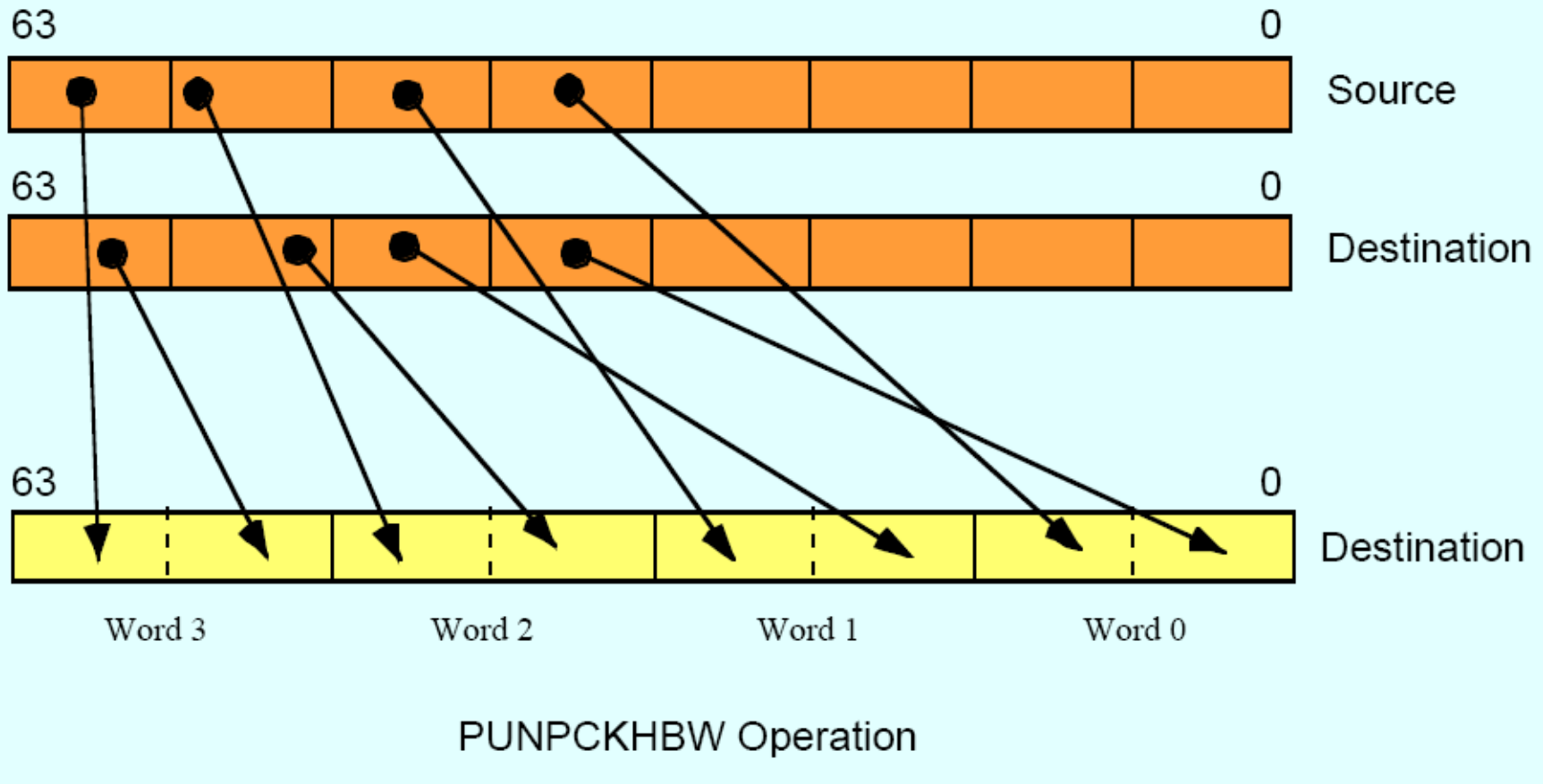
Unpack low portion



Unpack low portion



Unpack high portion



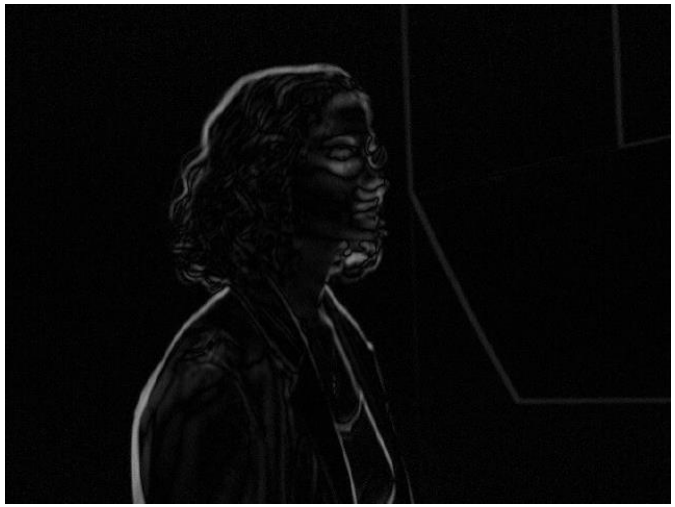
Keys to SIMD programming

- Efficient data layout
- Elimination of branches

Application: frame difference

Application: frame difference

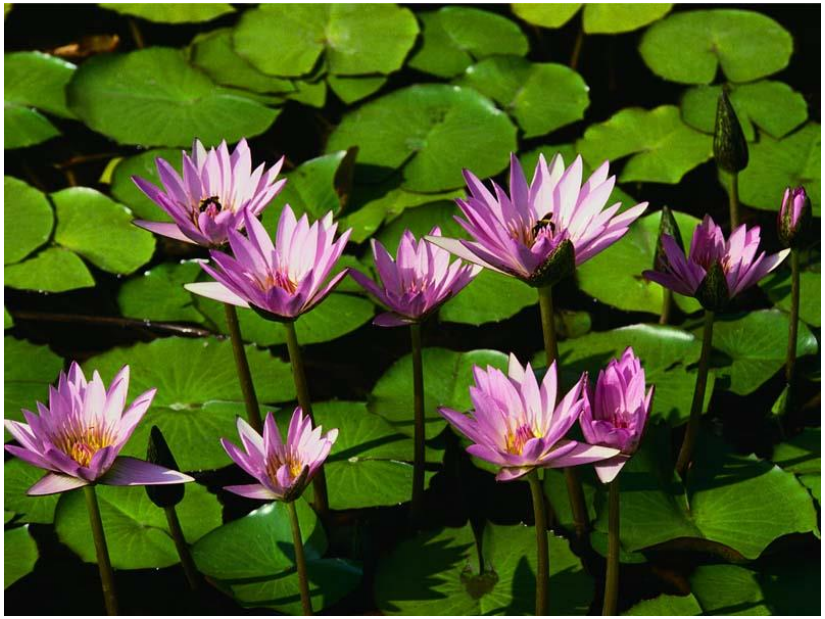
(A-B) or (B-A)



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

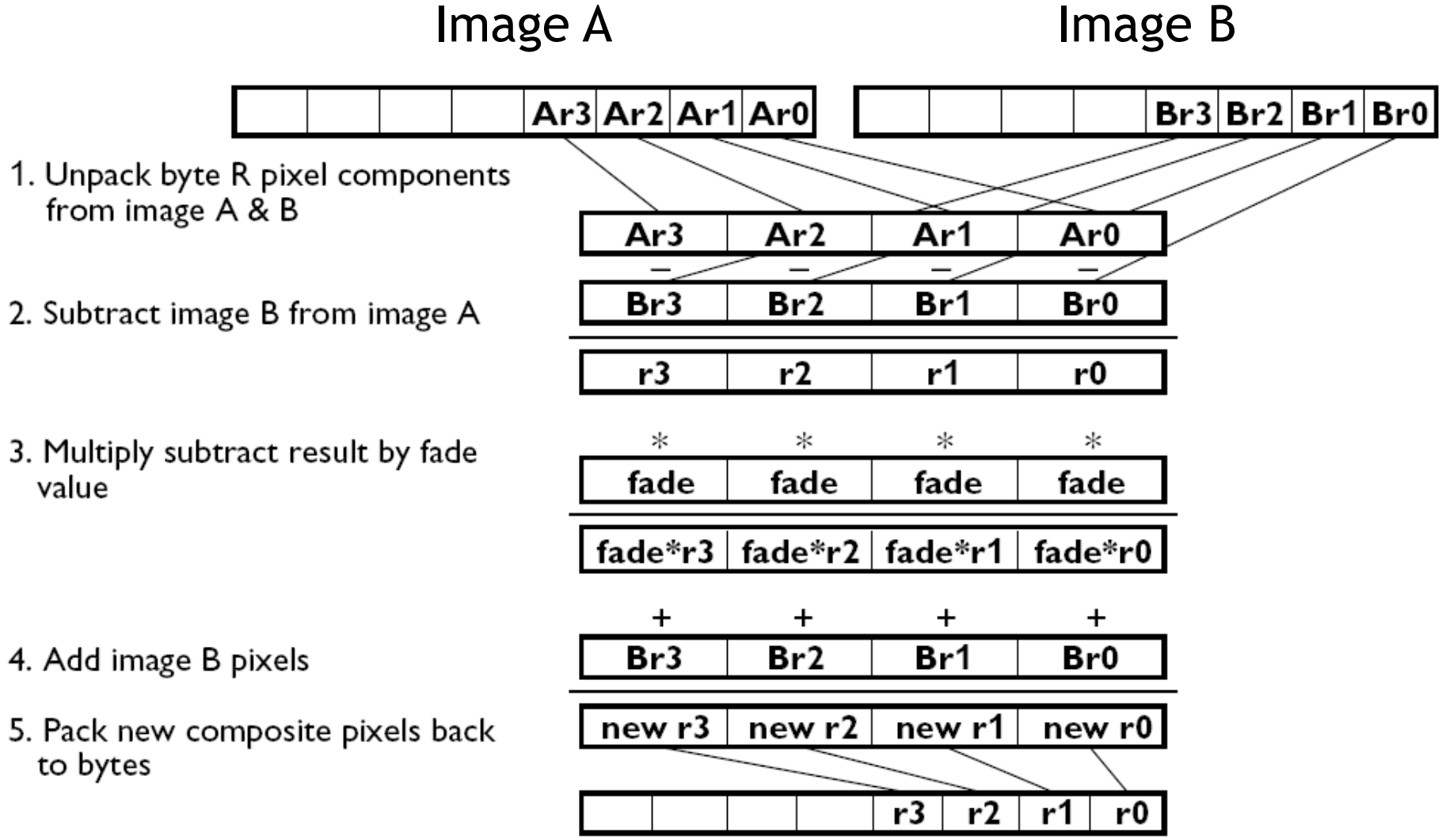
$$A*\alpha+B*(1-\alpha) = B+\alpha(A-B)$$

$\alpha=0.75$

$\alpha=0.5$

$\alpha=0.25$

Example: image fade-in-fade-out



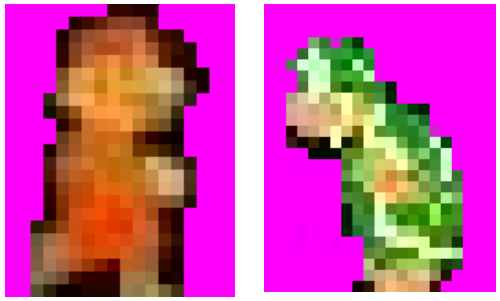
Example: image fade-in-fade-out

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

Data-independent computation

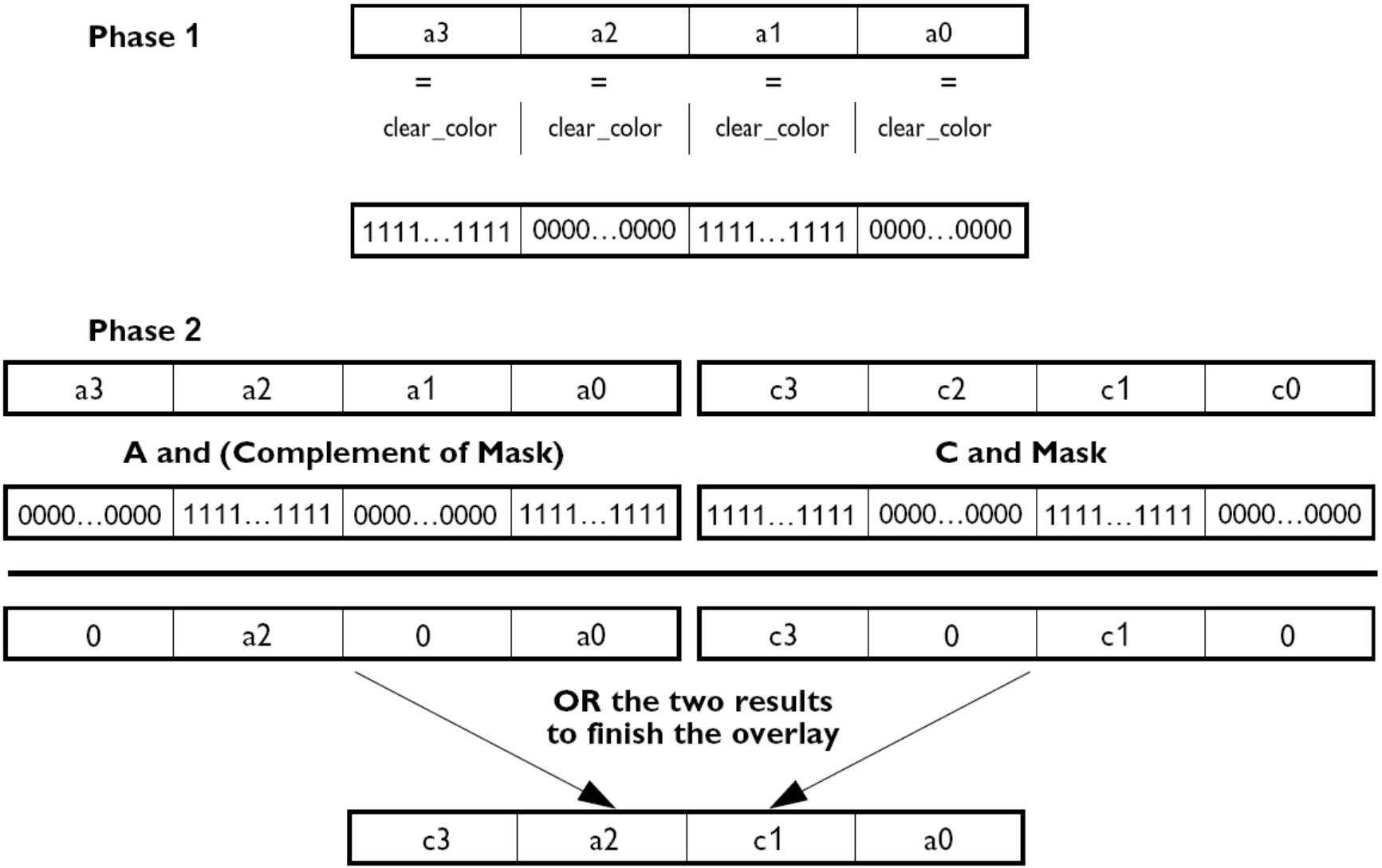
- Each operation can execute without needing to know the results of a previous operation.
- Example, sprite overlay

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



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

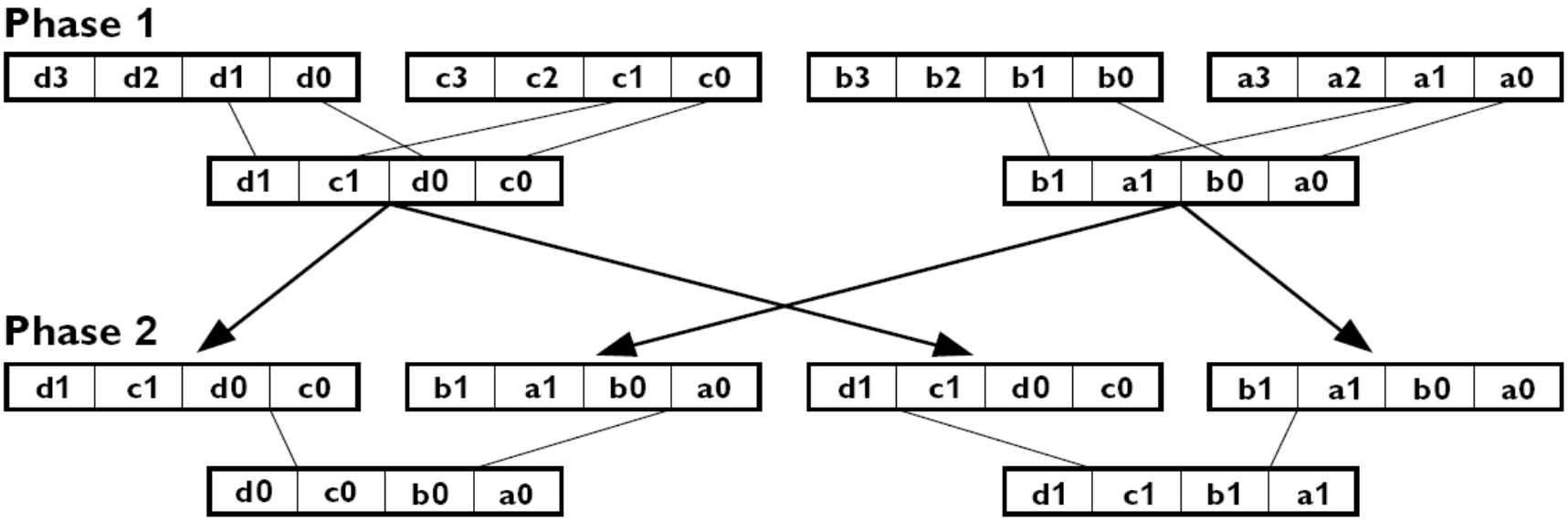
Application: sprite overlay



Application: sprite overlay

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

Application: matrix transport



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

MMX code sequence operation:

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

Application: matrix transport

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

Application: matrix transport

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

Application: matrix transport

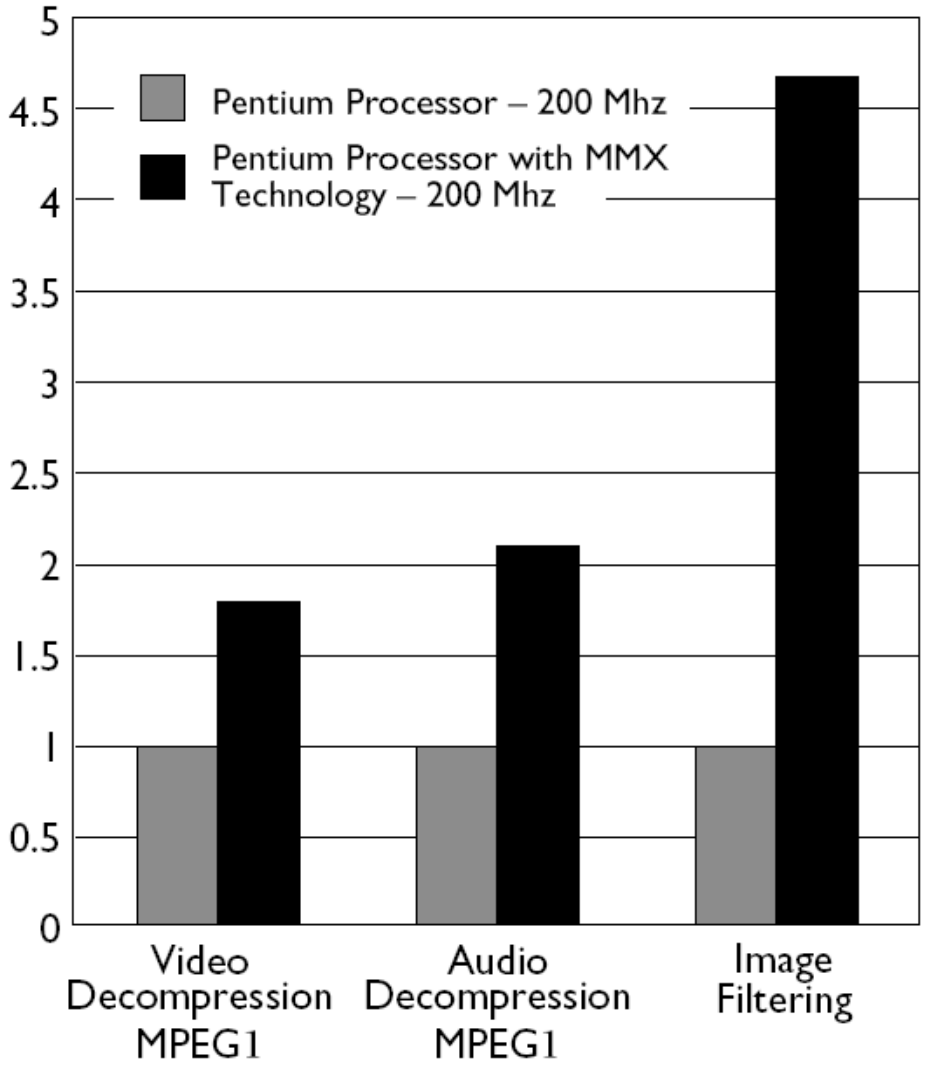
```
//generate rows 5 to 8 of M2
movq mm1, M1 //get row 1 of M1
movq mm3, M1+16 //get row 3 of M1
punpckhbw mm1, mm2
punpckhbw mm3, mm4
movq mm0, mm1
punpcklwd mm1, mm3 //mm1 has row 6 & row 5
punpckhwd mm0, mm3 //mm0 has row 8 & row 7
//save results to M2
movq M2+16, mm1
movq M2+24, mm0
emms
} //end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

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



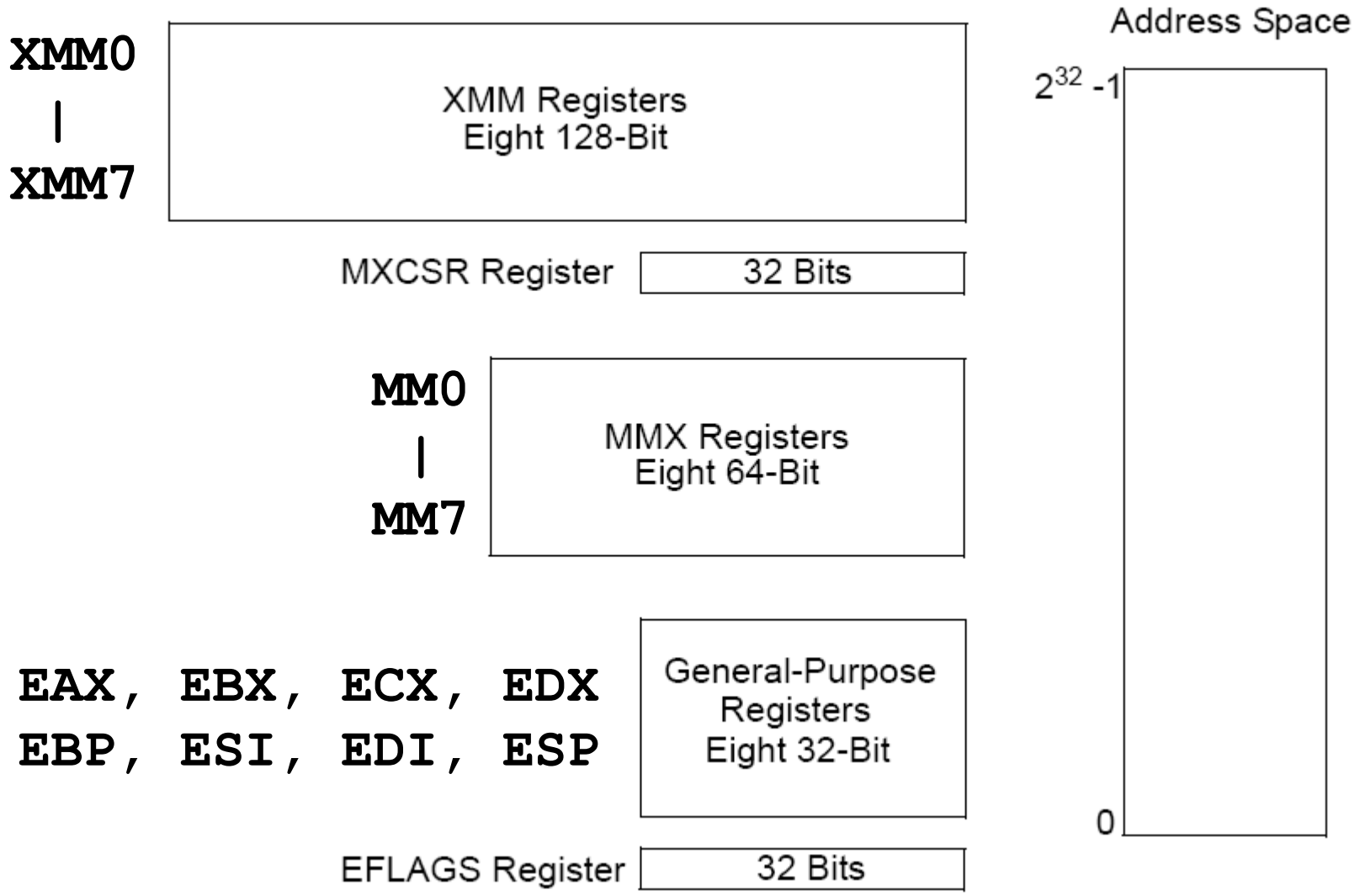
SSE

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

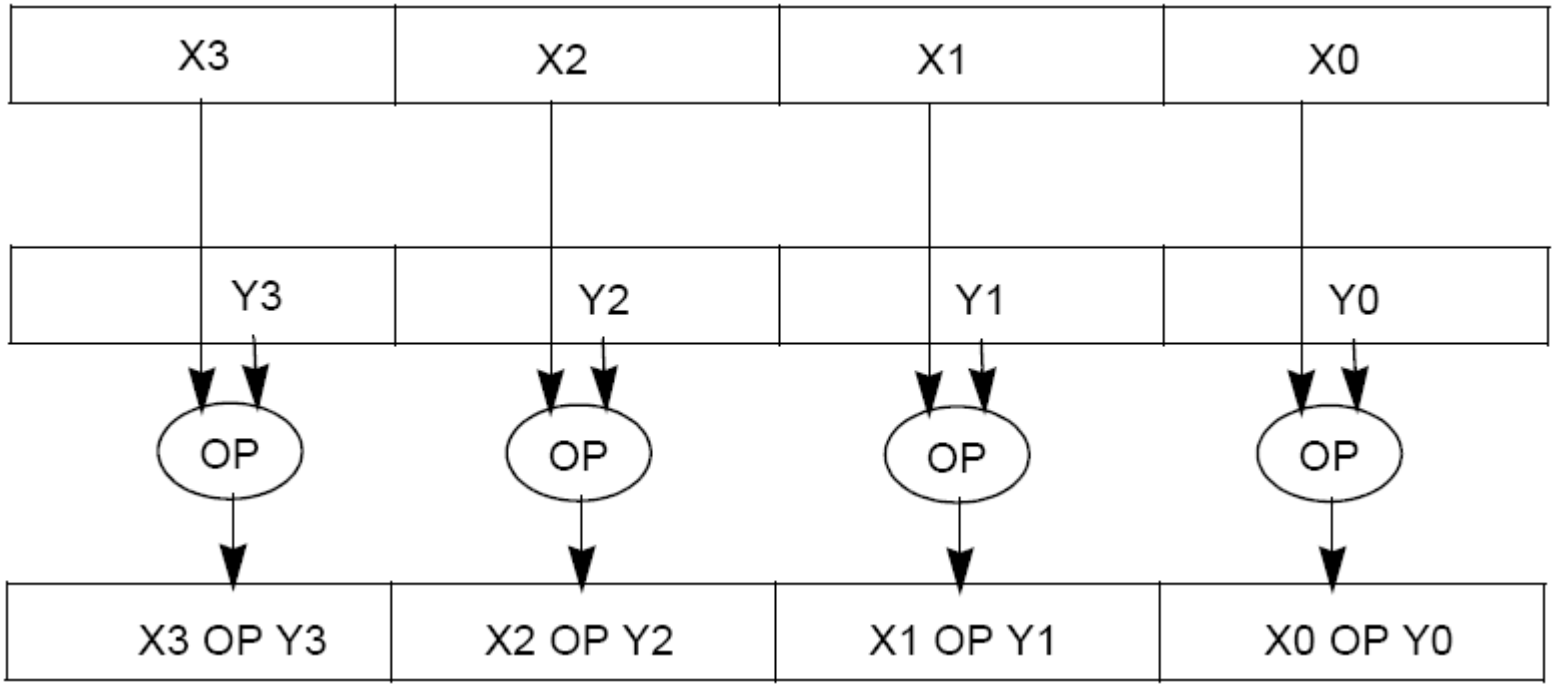
SSE features

- 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.

SSE programming environment

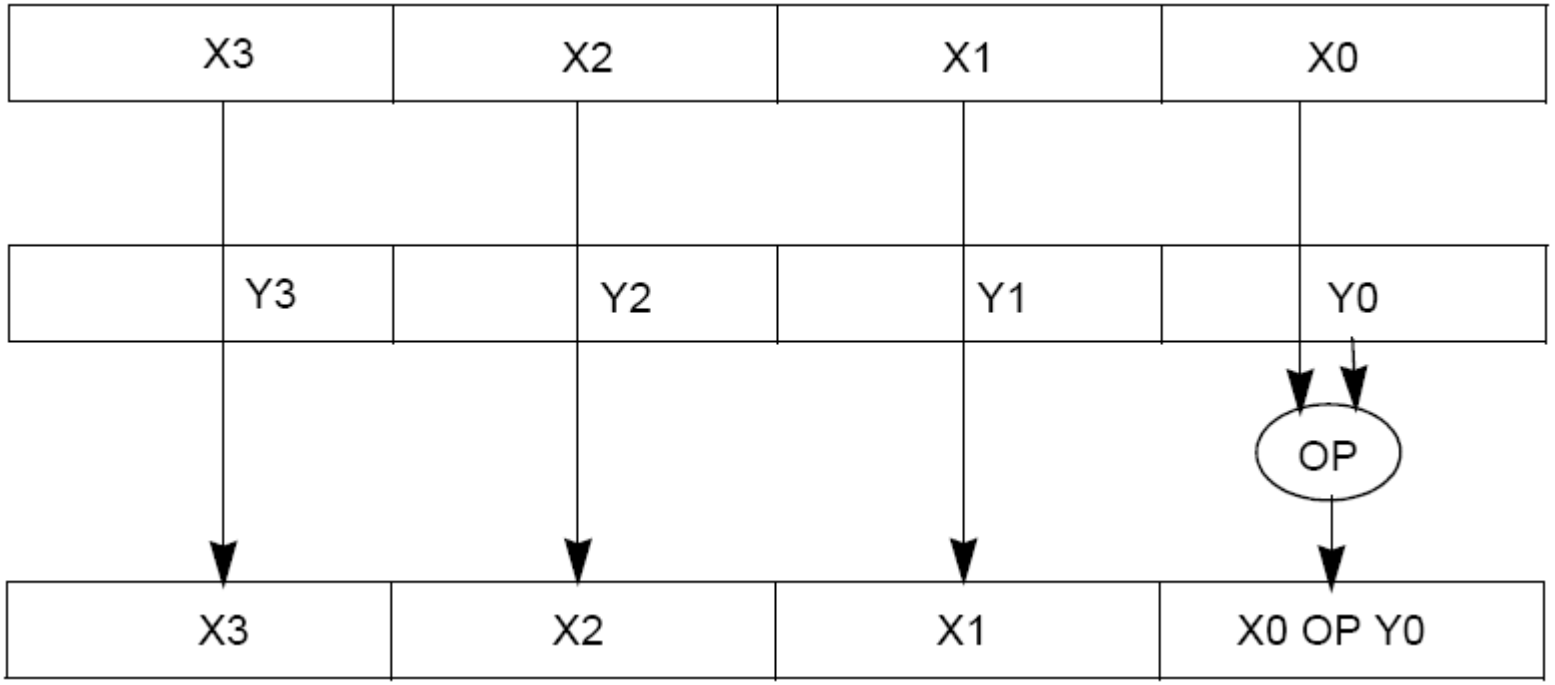


SSE packed FP operation



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

SSE scalar FP operation

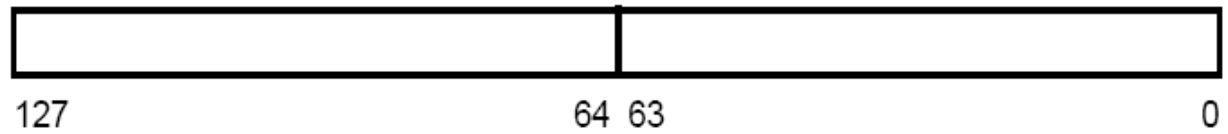


- **`ADDSS/SUBSS`**: scalar single-precision FP used as FPU?

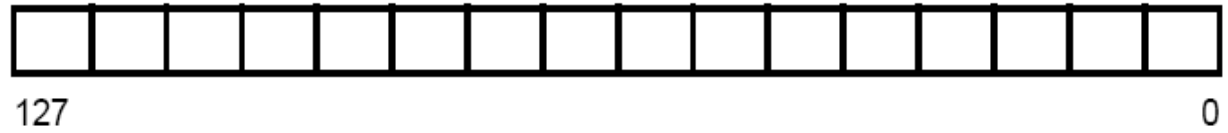
SSE2

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

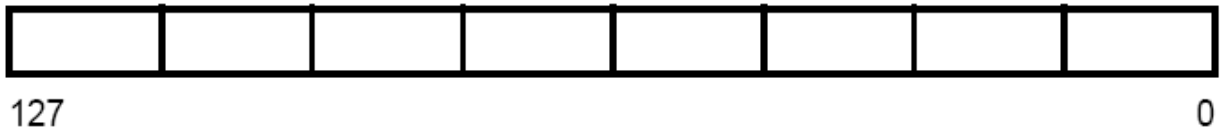
SSE2 features



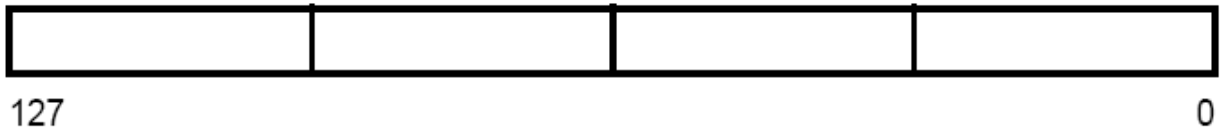
128-Bit Packed Double-Precision Floating-Point



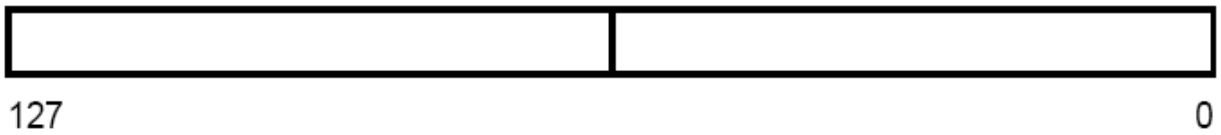
128-Bit Packed Byte Integers



128-Bit Packed Word Integers



128-Bit Packed Doubleword Integers



128-Bit Packed Quadword 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