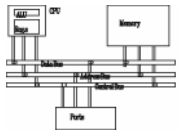
Intel SIMD architecture

Computer Organization and Assembly Languages

Yung-Yu Chuang

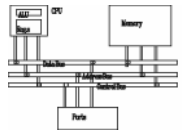
2005/12/29

Announcement



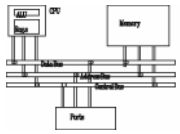
- TA evaluation on the next week

Reference



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

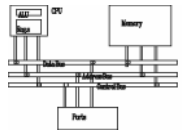
Overview



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

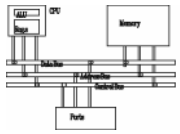
- SIMD instructions are probably the best place to use assembly since high level languages do not do a good job on using these instruction

Performance boost

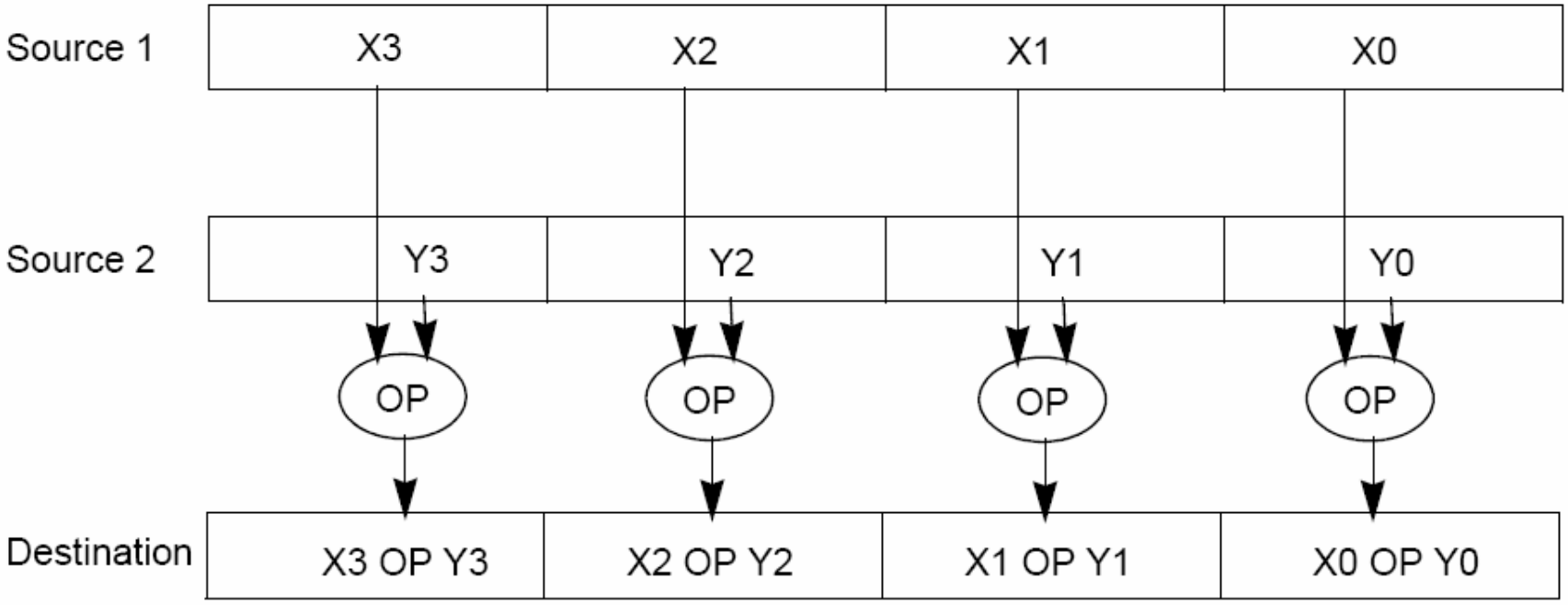


- Increasing clock rate is not fast enough for boosting performance
- Architecture improvement is more significant such as pipeline/cache/SIMD
- Intel analyzed multimedia applications and found they share the following characteristics:
  - Small native data types
  - Recurring operations
  - Inherent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

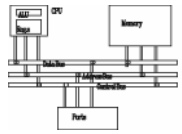
SIMD



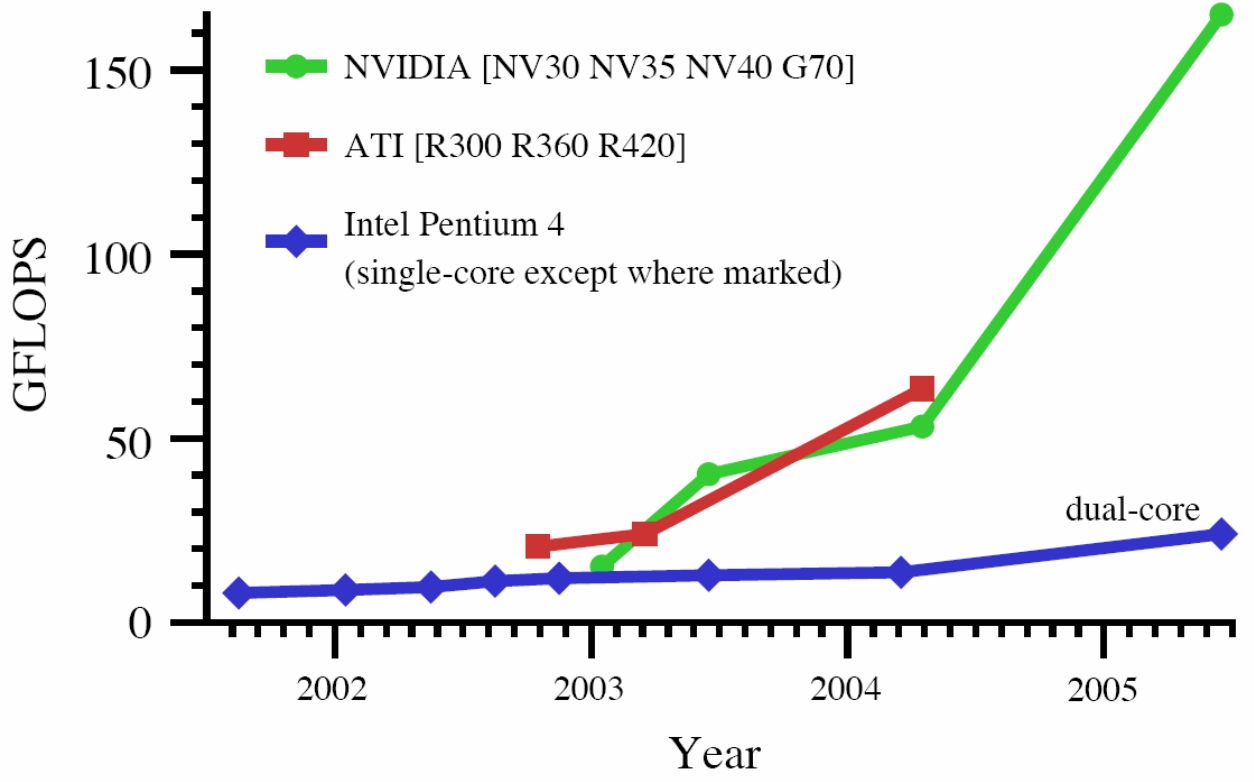
- SIMD (single instruction multiple data) architecture performs the same operation on multiple data elements in parallel
- **`PADDW MM0, MM1`**



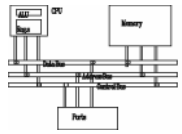
Other SIMD architectures



- Graphics Processing Unit (GPU): nVidia 7800, 24 fragment shader pipelines
- Cell Processor (IBM/Toshiba/Sony): POWERPC+8 SPEs, will be used in PS3.

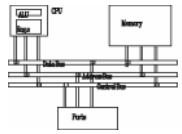


IA-32 SIMD development



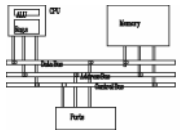
- MMX (Multimedia Extension) was introduced in 1996 (Pentium with MMX and Pentium II).
- SSE (Streaming SIMD Extension) was introduced with Pentium III.
- SSE2 was introduced with Pentium 4.
- SSE3 was introduced with Pentium 4 supporting hyper-threading technology. SSE3 adds 13 more instructions.

MMX

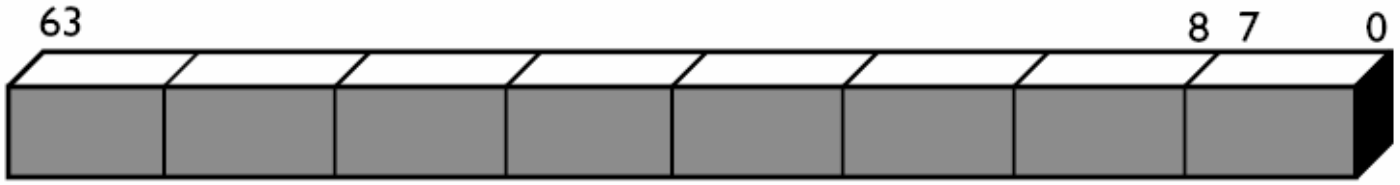


- After analyzing a lot of existing applications such as graphics, MPEG, music, speech recognition, game, image processing, they found that many multimedia algorithms execute the same instructions on many pieces of data in a large data set.
- Typical elements are small, 8 bits for pixels, 16 bits for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

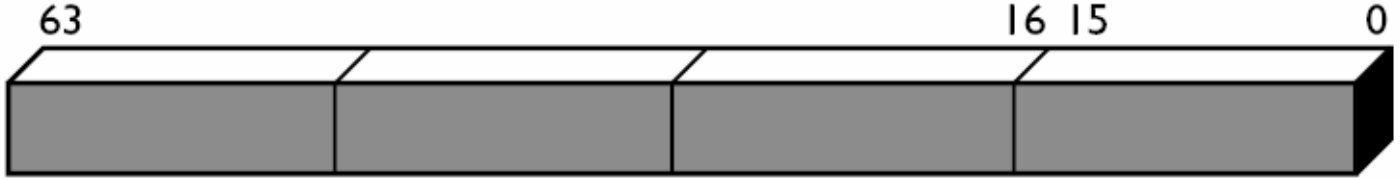
MMX data types



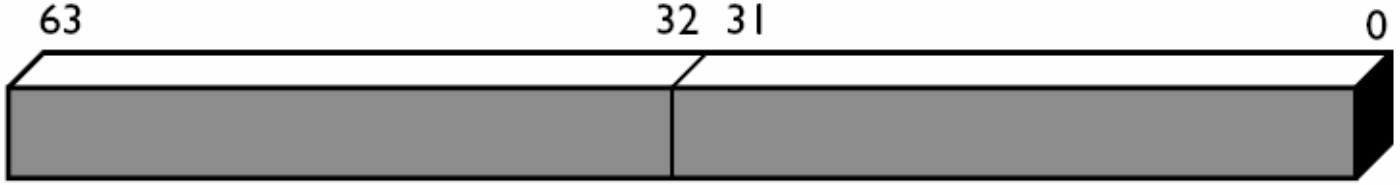
Packed Byte: 8 bytes packed into 64 bits



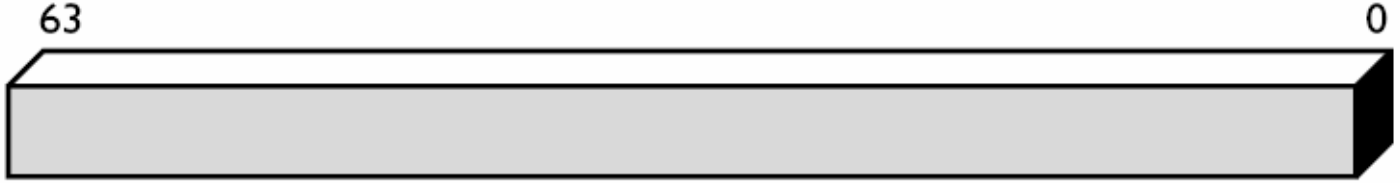
Packed Word: 4 words packed into 64 bits



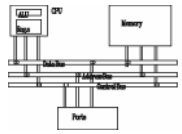
Packed Doubleword: 2 doublewords packed into 64 bits

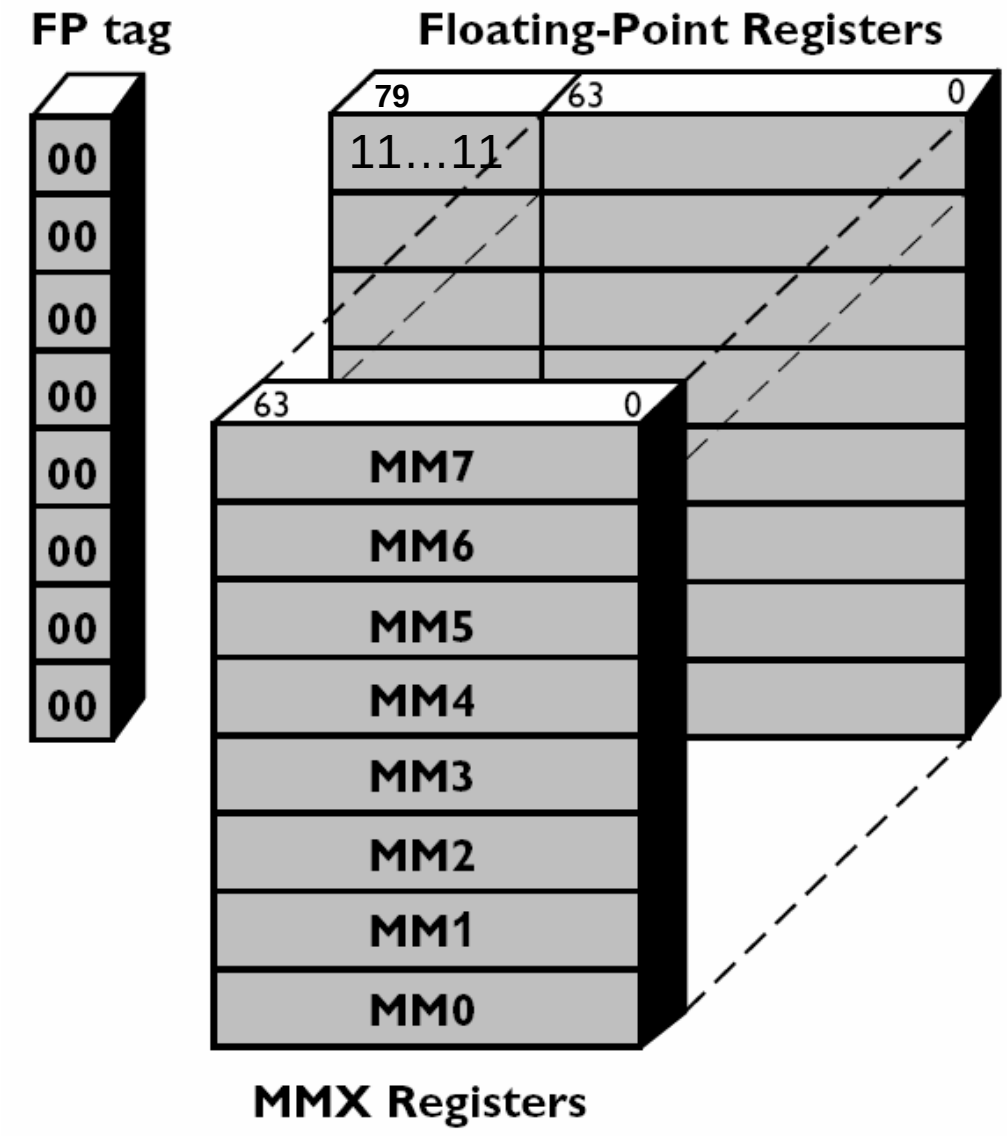


Packed Quadword: One 64-bit quantity



MMX integration into IA

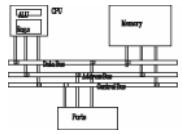




NaN or infinity as real

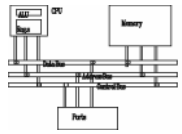
Even if MMX registers are 64-bit, they don't extend Pentium to a 64-bit CPU since only logic instructions are provided for 64-bit data.

Compatibility



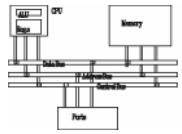
- To be fully compatible with existing IA, no new mode or state was created. Hence, for context switching, no extra state needs to be saved.
- To reach the goal, MMX is hidden behind FPU. When floating-point state is saved or restored, MMX is saved or restored.
- It allows existing OS to perform context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

Compatibility



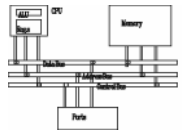
- Although Intel defenses their decision on aliasing MMX to FPU for compatibility. It is actually a bad decision. OS can just provide a service pack or get updated.
- It is why Intel introduced SSE later without any aliasing

MMX instructions

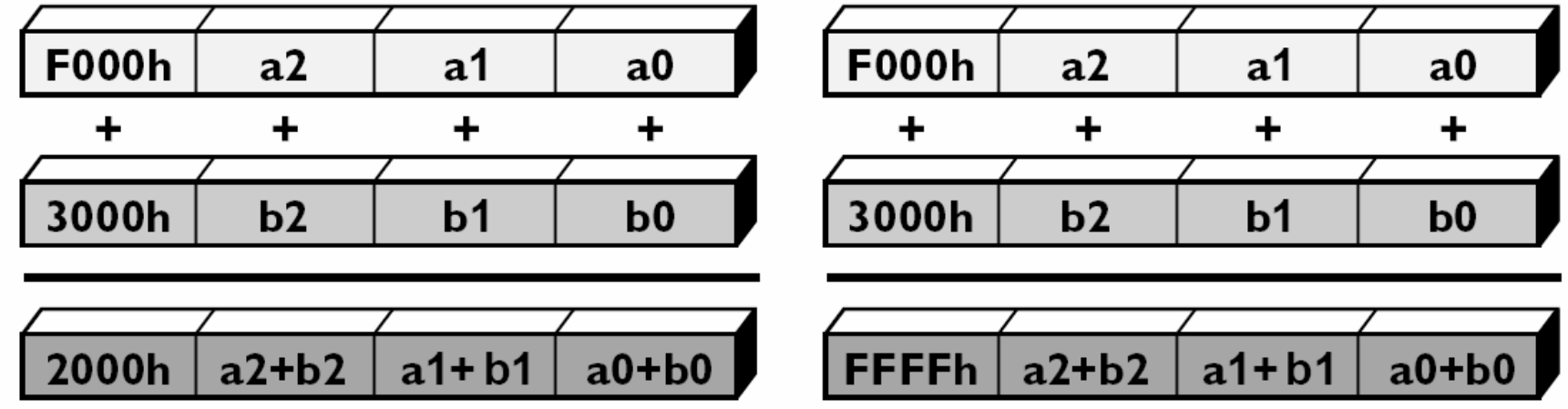


- 57 MMX instructions are defined to perform the parallel operations on multiple data elements packed into 64-bit data types.
- These include **`add`**, **`subtract`**, **`multiply`**, **`compare`**, and **`shift`**, **`data conversion`**, **`64-bit data move`**, **`64-bit logical operation`** and **`multiply-add`**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

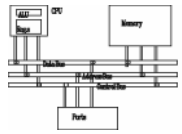
Saturation arithmetic



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

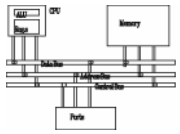


MMX instructions



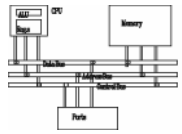
Category		Wraparound	Signed Saturation	Unsigned Saturation
Arithmetic	Addition	PADDB, PADDW, PADDD	PADDSB, PADDSW	PADDUSB, PADDUSW
	Subtraction	PSUBB, PSUBW, PSUBD	PSUBSB, PSUBSW	PSUBUSB, PSUBUSW
	Multiplication	PMULL, PMULH		
	Multiply and Add	PMADD		
Comparison	Compare for Equal	PCMPEQB, PCMPEQW, PCMPEQD		
	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



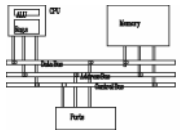
		Packed	Full Quadword
Logical	And And Not Or Exclusive OR		PAND PANDN POR PXOR
Shift	Shift Left Logical Shift Right Logical Shift Right Arithmetic	PSLLW, PSLLD PSRLW, PSRLD PSRAW, PSRAD	PSLLQ PSRLQ
		Doubleword Transfers	**Quadword Transfers**
Data Transfer	Register to Register Load from Memory Store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

Arithmetic



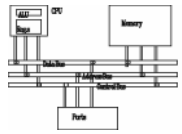
- **PADDB/PADDW/PADDD**: add two packed numbers, no CFLAGS is set, ensure overflow never occurs by yourself
- Multiplication: two steps
- **PMULLW**: multiplies four words and stores the four lo words of the four double word results
- **PMULHW/PMULHUW**: multiplies four words and stores the four hi words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

Detect MMX/SSE



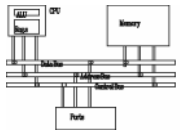
```
mov   eax, 1
cpuid     ; supported since Pentium
test  edx, 00800000h ;bit 23
          ; 02000000h (bit 25) SSE
          ; 04000000h (bit 26) SSE2
jnz   HasMMX
```

Example: add a constant to a vector

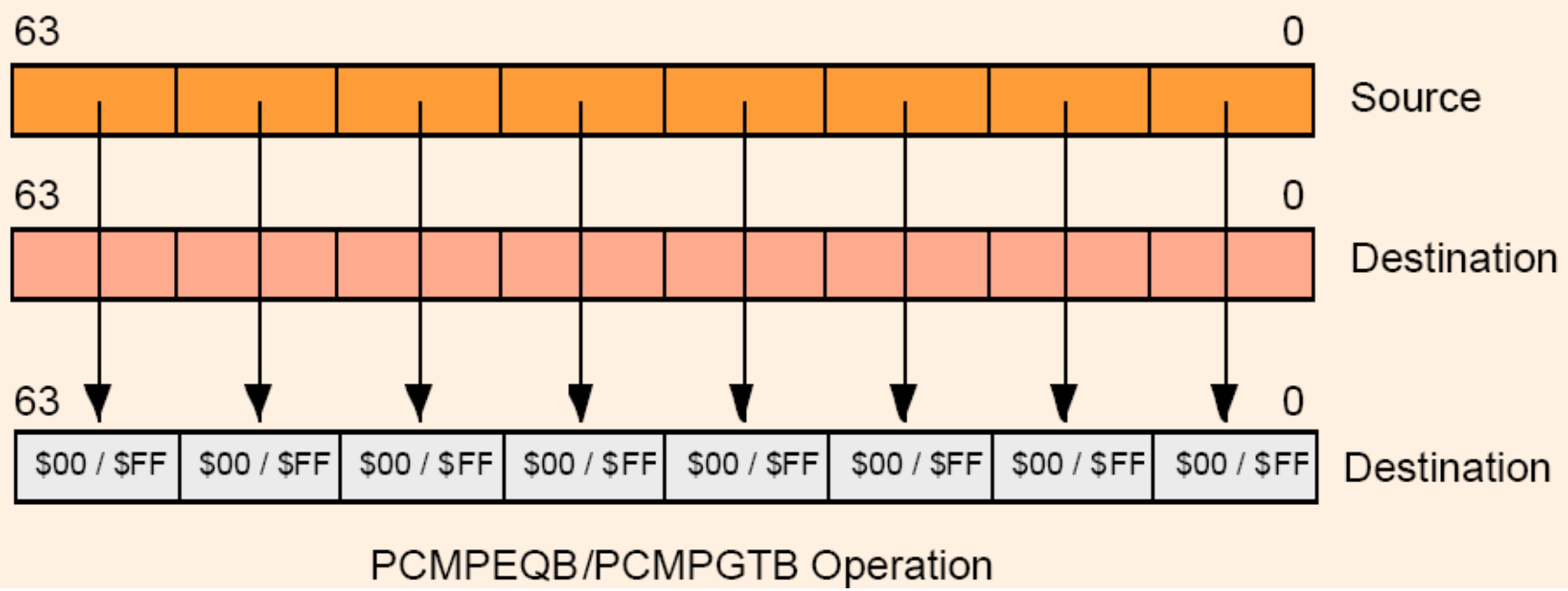


```
char d[]={5, 5, 5, 5, 5, 5, 5, 5};
char clr[]={65,66,68,...,87,88}; // 24 bytes
__asm{
    movq mm1, d
    mov cx, 3
    mov esi, 0
L1: movq mm0, clr[esi]
    paddb mm0, mm1
    movq clr[esi], mm0
    add esi, 8
    loop L1
    emms
}
```

Comparison

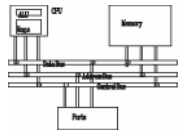


- No CFLAGS, how many flags will you need? Results are stored in destination.
- EQ/GT, no LT

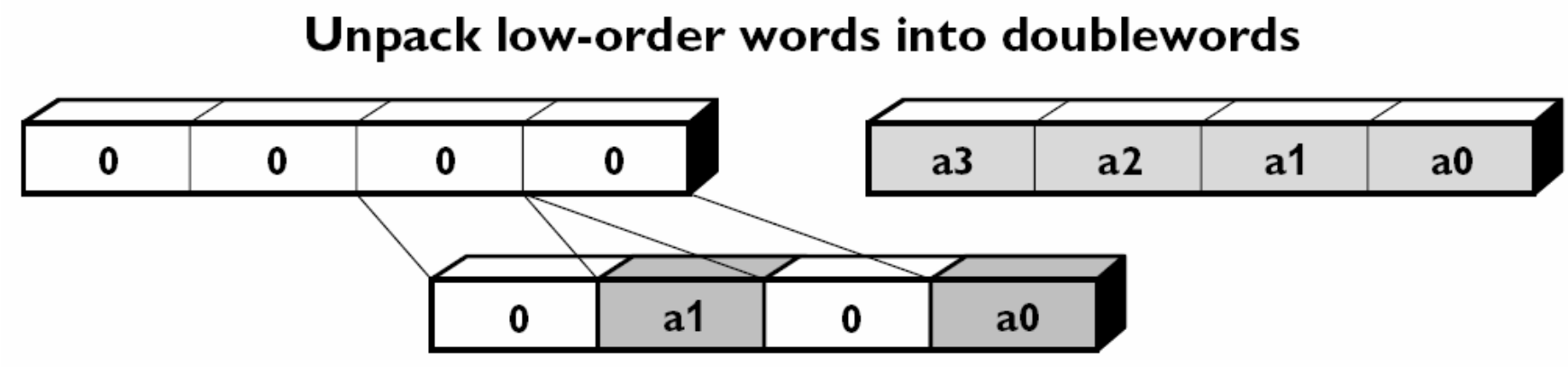


PCMPEQB/PCMPGTB Operation

Change data types

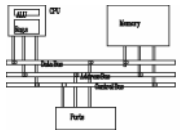


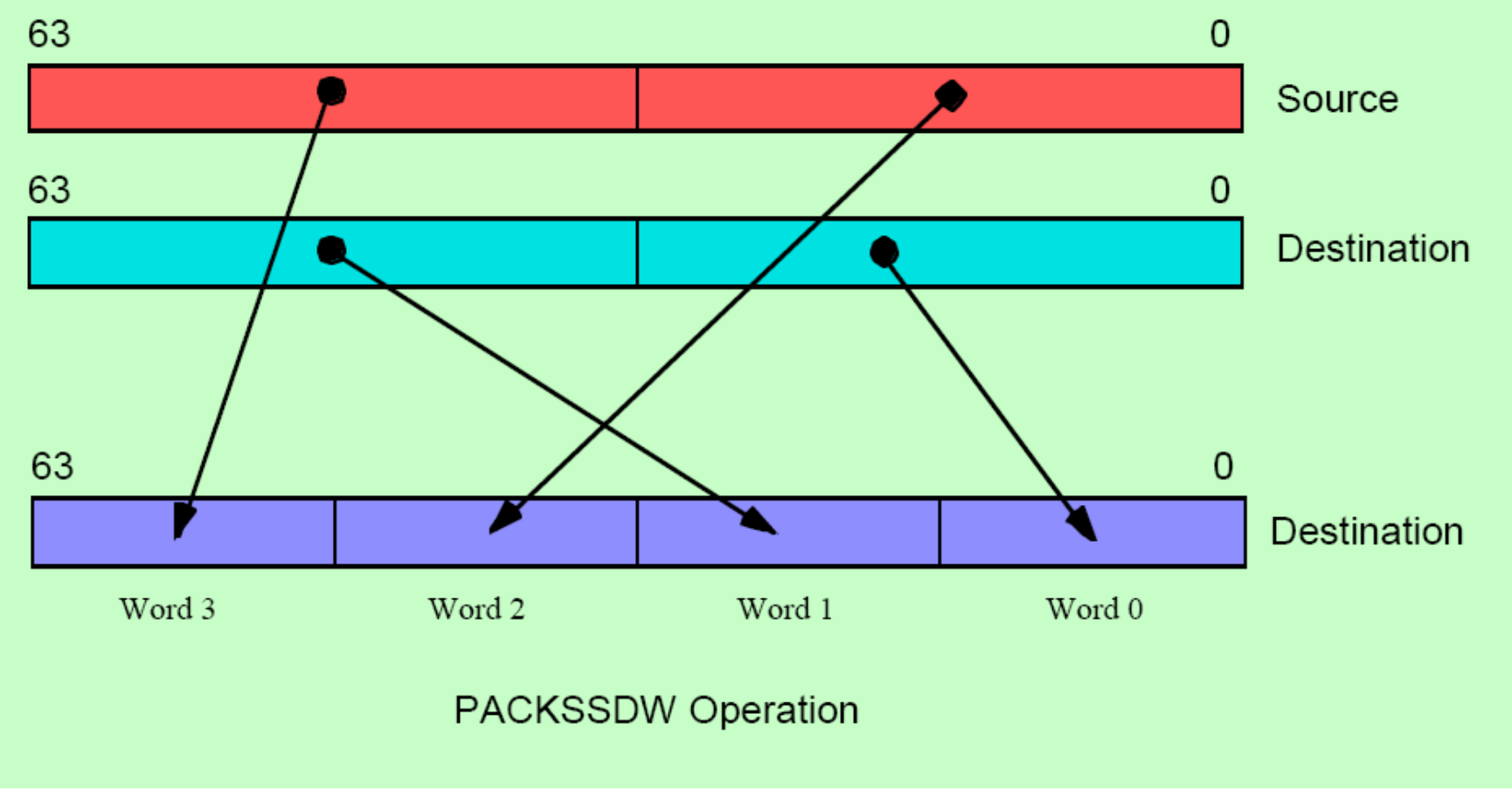
- Unpack: takes two operands and interleave them. It can be used for expand data type for immediate calculation.



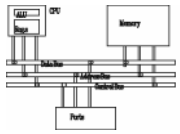
- Pack: converts a larger data type to the next smaller data type.

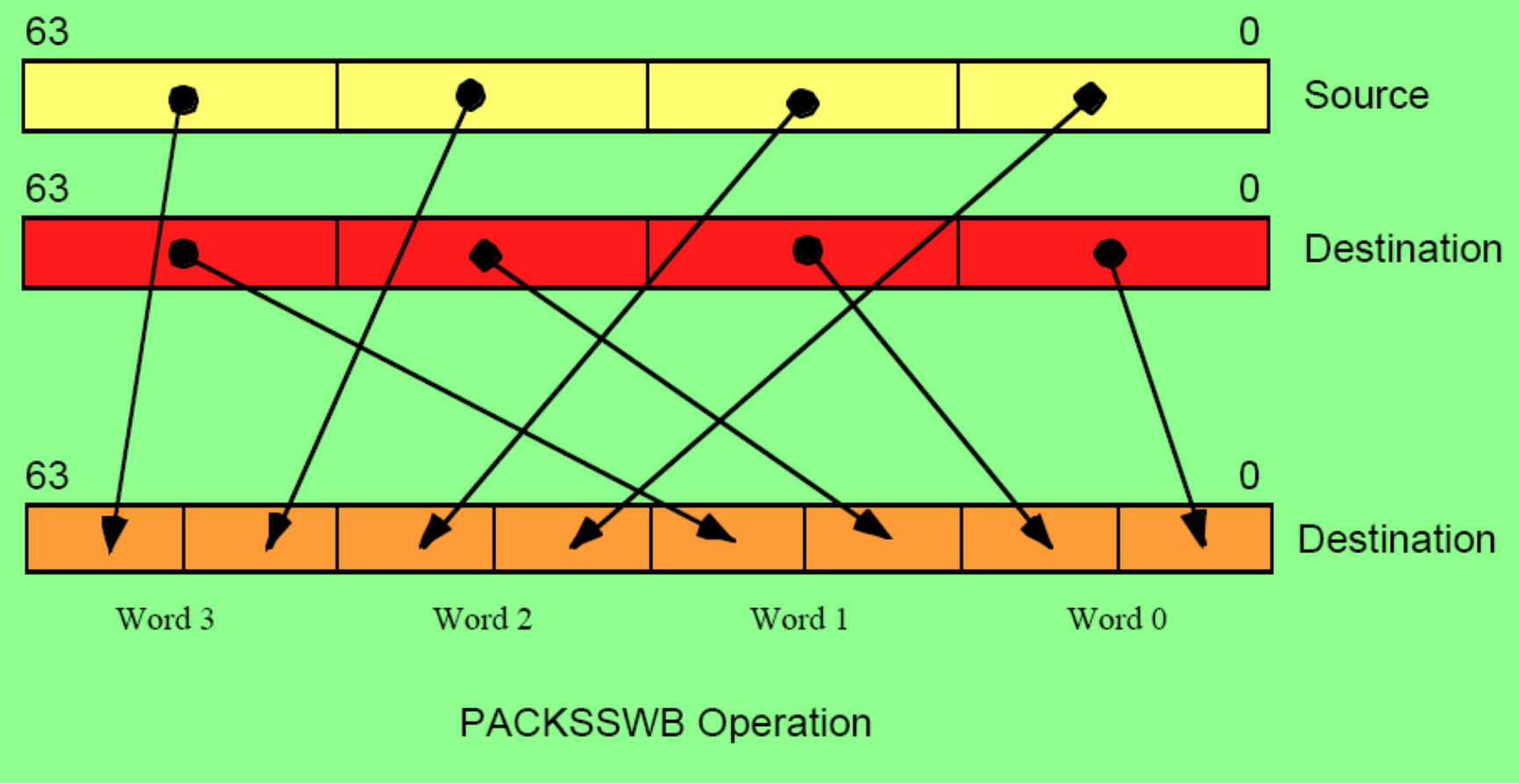
Pack and saturate signed values



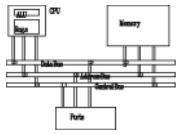


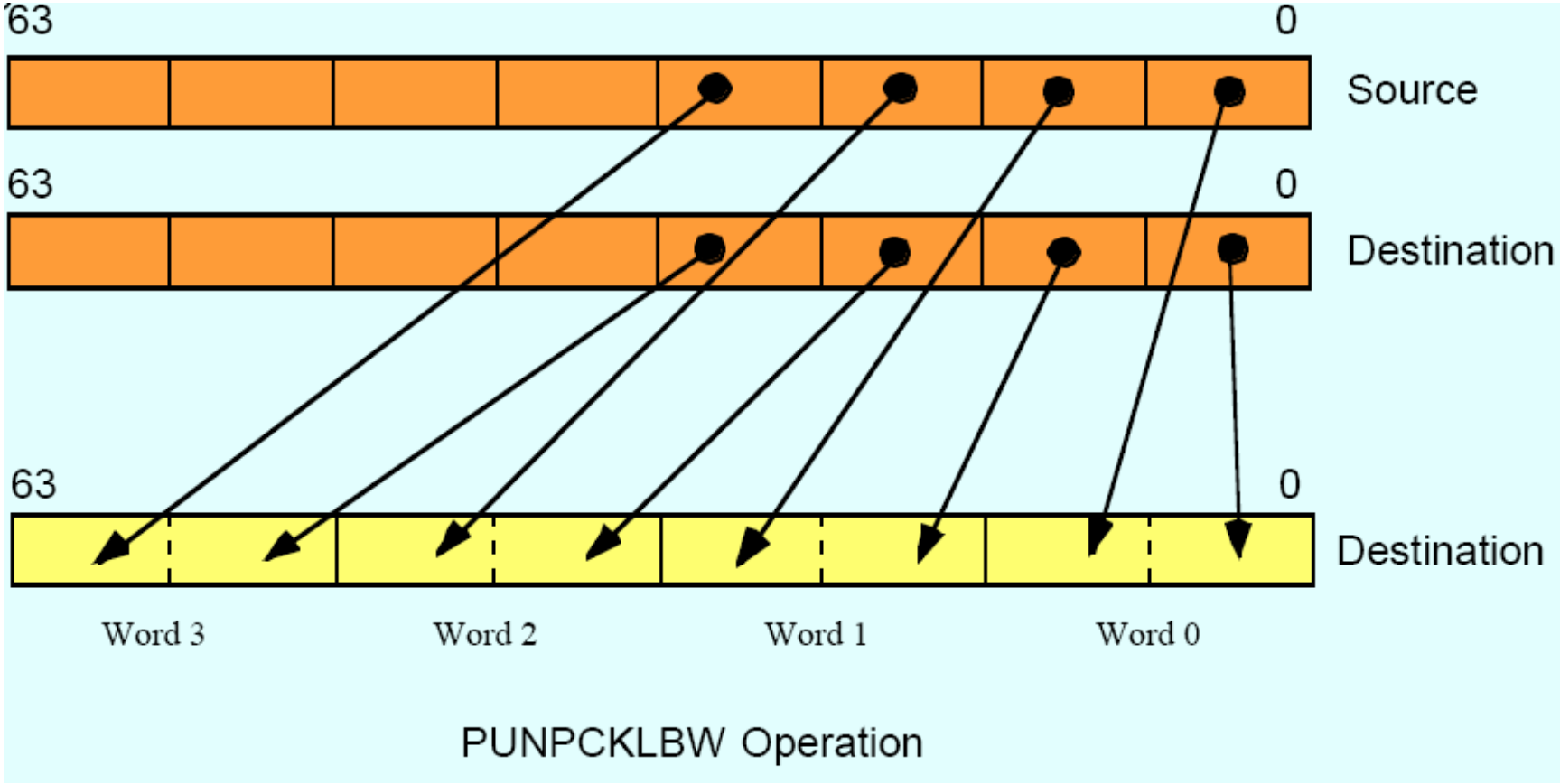
Pack and saturate signed values



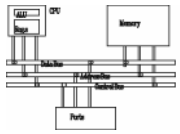


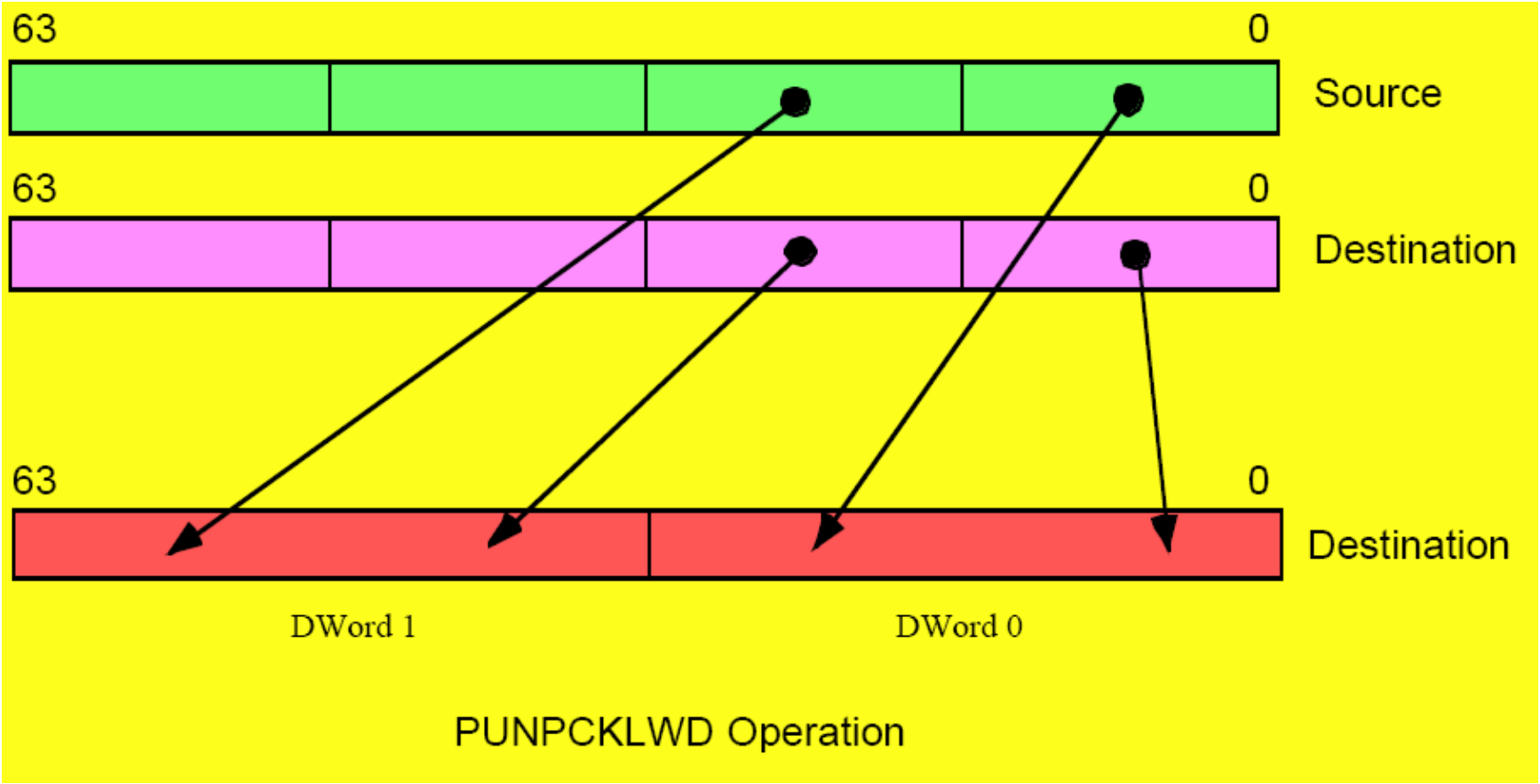
Unpack low portion



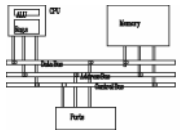


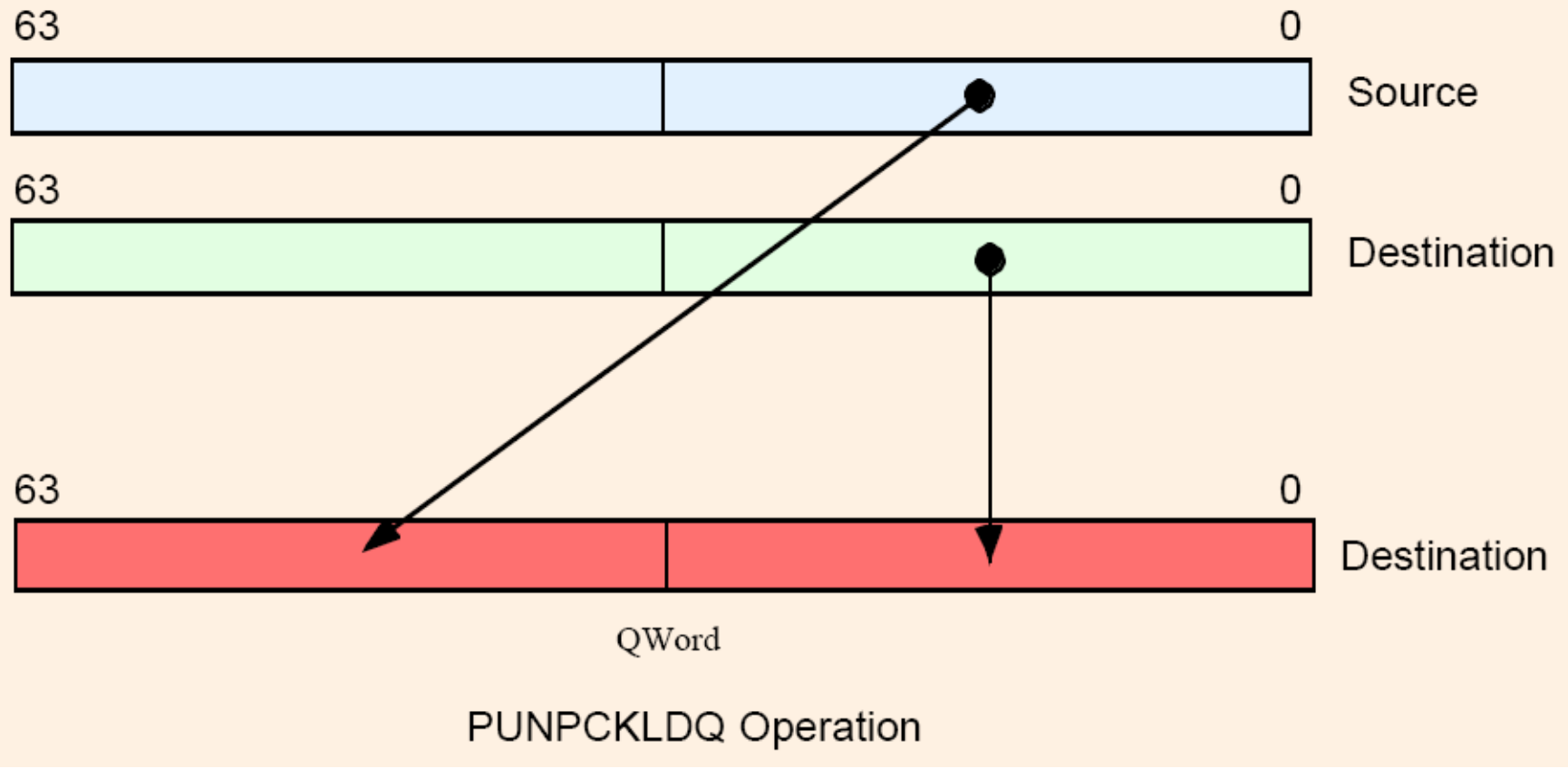
Unpack low portion



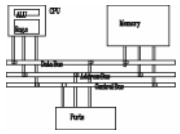


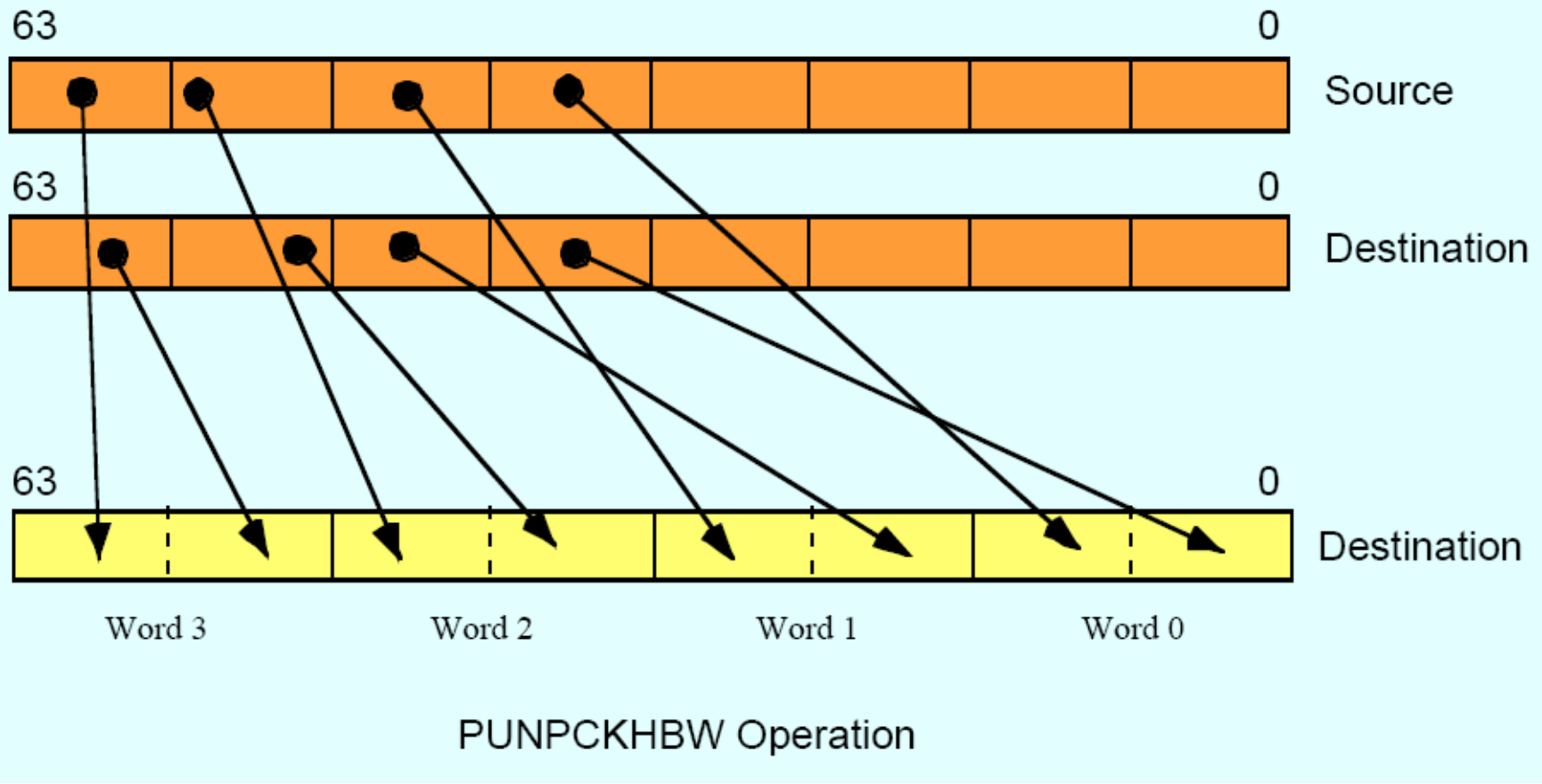
Unpack low portion



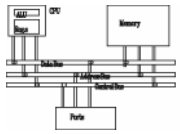


Unpack high portion





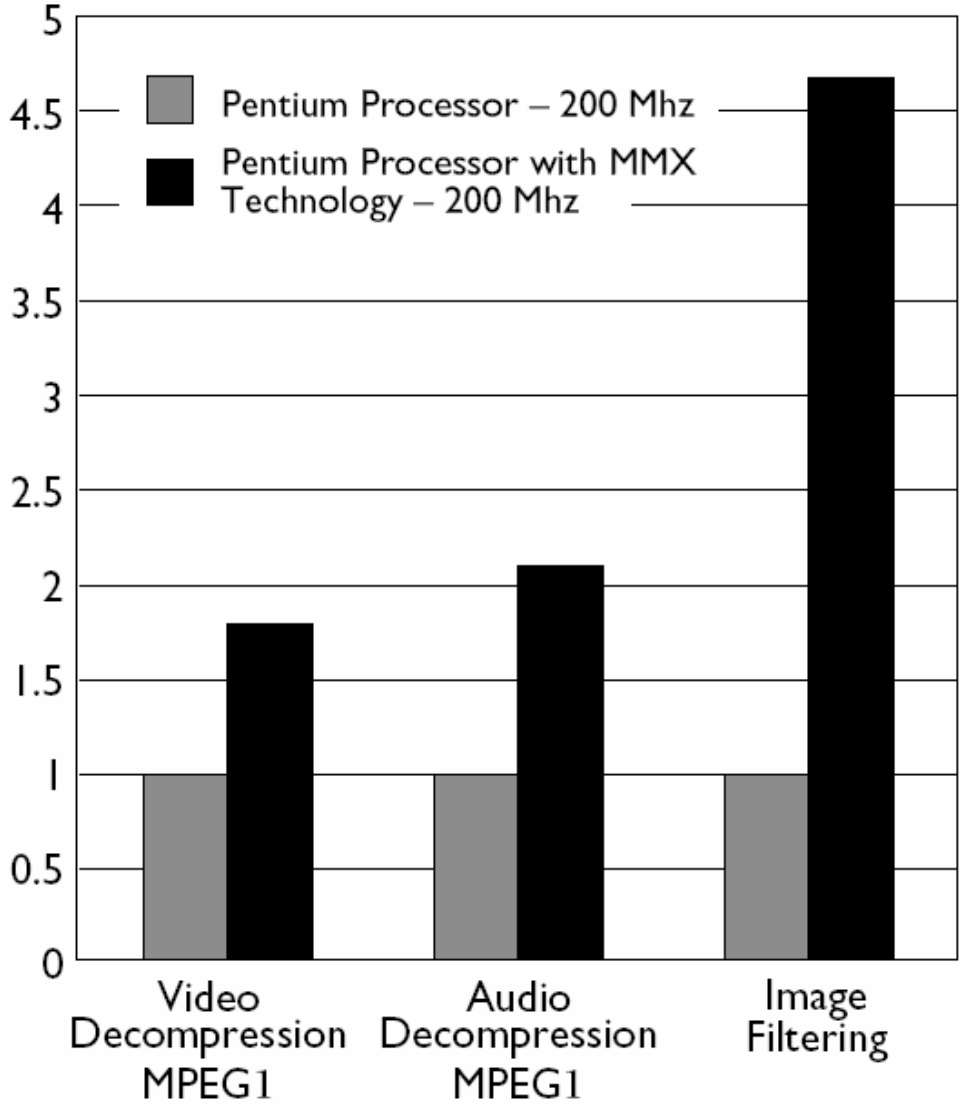
Performance boost (data from 1996)



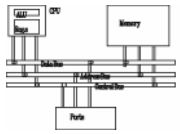
Benchmark kernels: FFT, FIR, vector dot-product, IDCT, motion compensation

65% performance gain

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

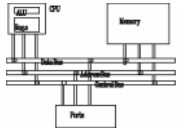


Keys to SIMD programming

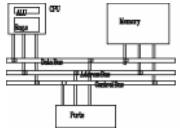


- Efficient memory layout
- Elimination of branches

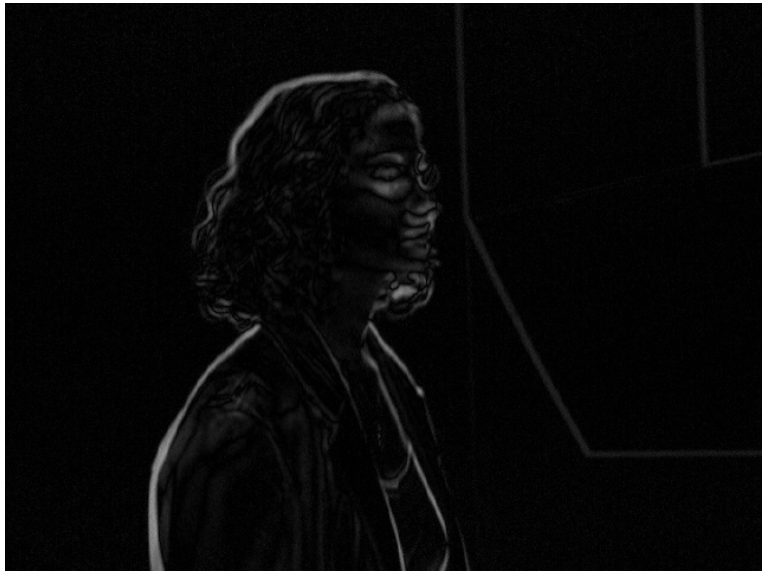
Application: frame difference



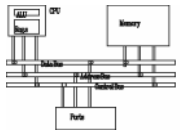
Application: frame difference



(A-B) or (B-A)

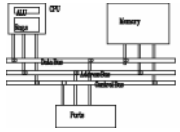


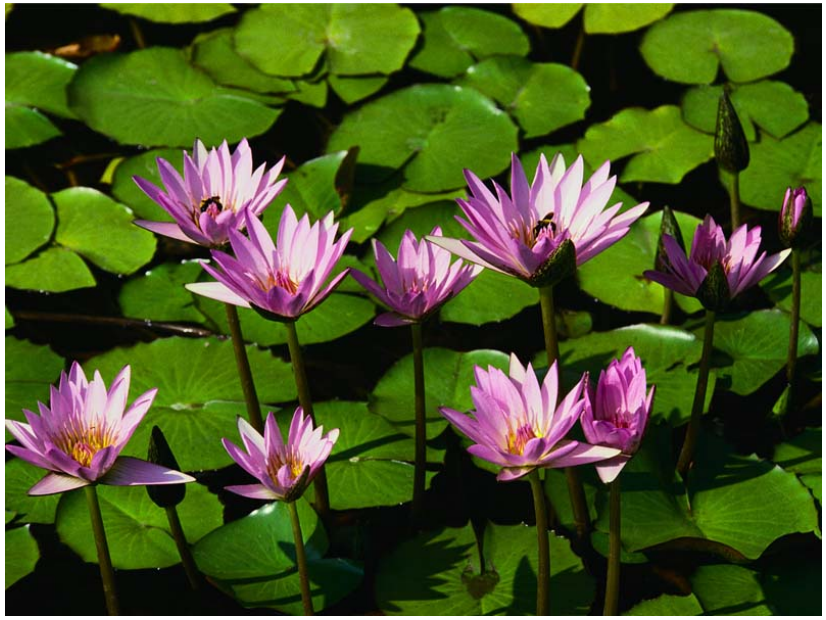
Application: frame difference



```
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



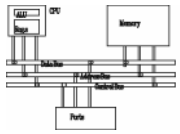


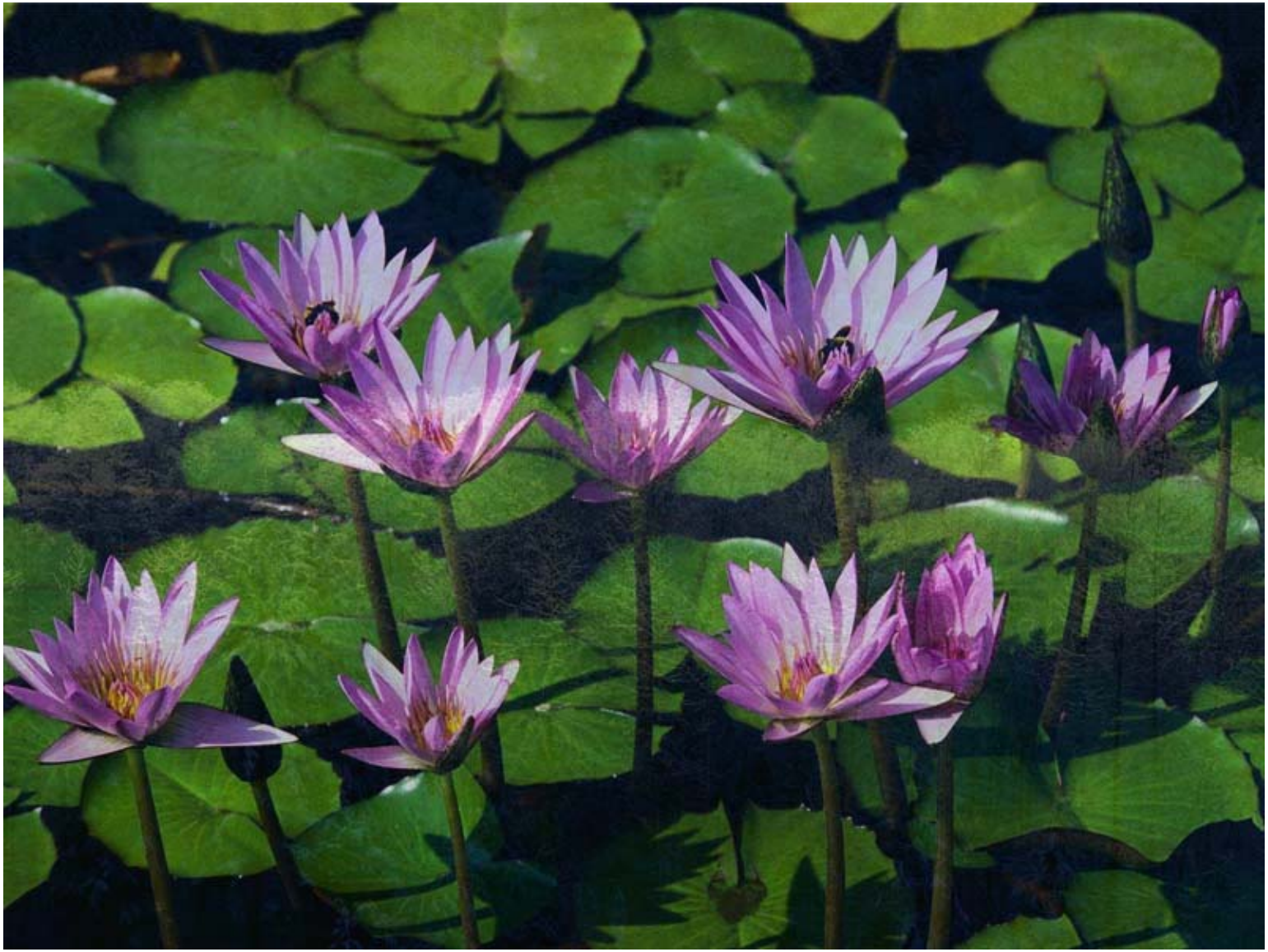
A

B

$A^{*}\alpha + B^{*}(1-\alpha)$

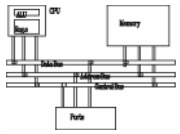
α=0.75



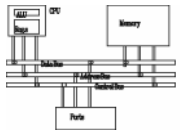


α=0.5

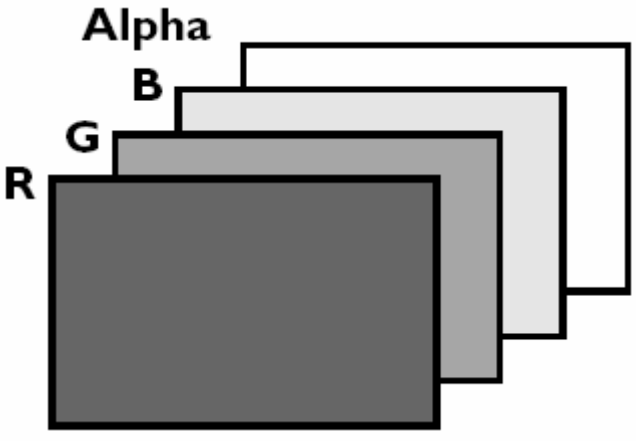
α=0.25

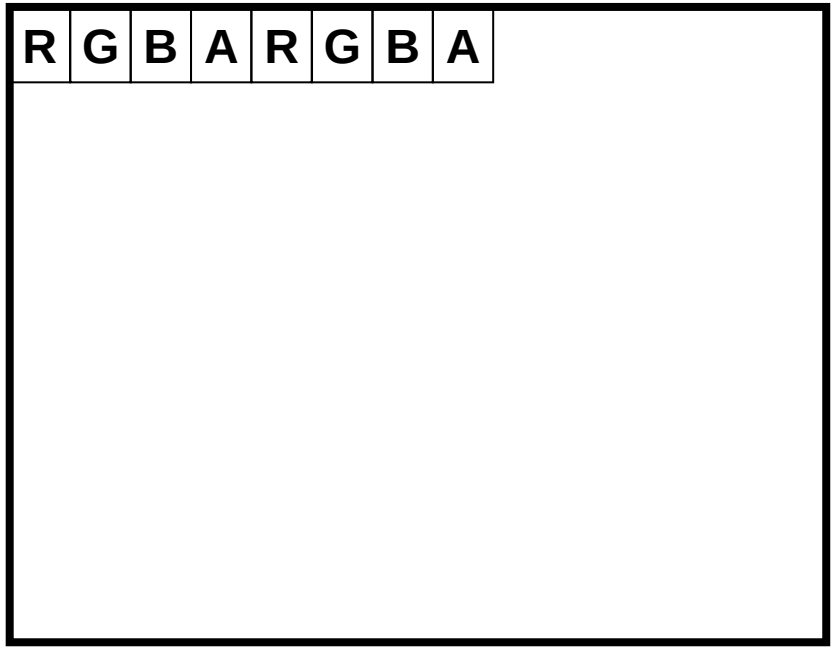


Example: image fade-in-fade-out

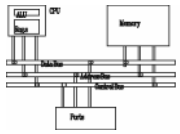


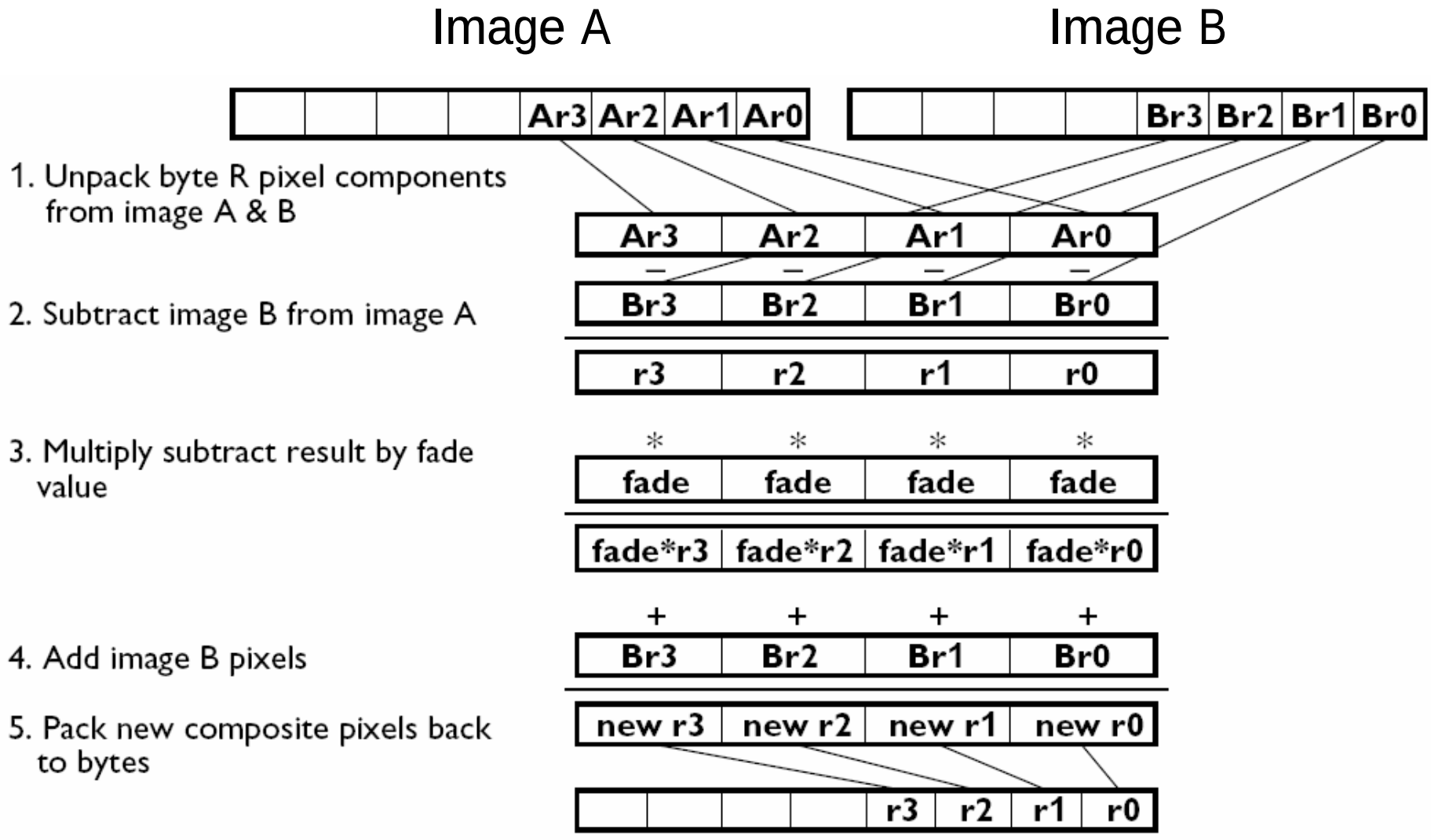
- Two formats: planar and chunky
- In Chunky format, 16 bits of 64 bits are wasted



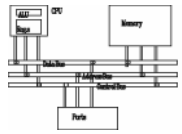


Example: image fade-in-fade-out



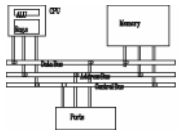


Example: image fade-in-fade-out



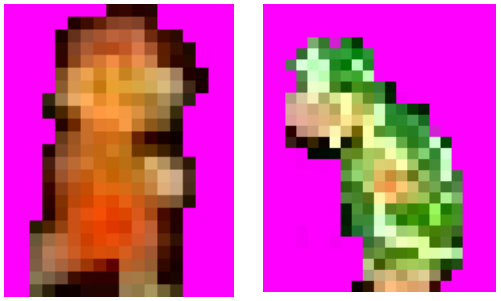
```
MOVQ       mm0, alpha//mm0 has 4 copies alpha
MOVD       mm1, A //move 4 pixels of image A
MOVD       mm2, B //move 4 pixels of image B
PXOR       mm3, mm3 //clear mm3 to all zeroes
//unpack 4 pixels to 4 words
PUNPCKLBW mm1, mm3
PUNPCKLBW mm2, mm3
PSUBW      mm1, mm2 //(B-A)
PMULLW     mm1, mm0 //(B-A)*fade
PADDW      mm1, mm2 //(B-A)*fade + B
//pack four words back to four bytes
PACKUSWB   mm1, mm3
```

Data-independent computation



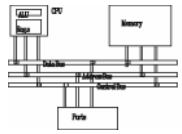
- Each operation can execute without needing to know the results of a previous operation.
- Example, sprite overlay

```
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



Phase 1

a3	a2	a1	a0
=	=	=	=
clear_color	clear_color	clear_color	clear_color

1111…1111	0000…0000	1111…1111	0000…0000

Phase 2

a3	a2	a1	a0

c3	c2	c1	c0

A and (Complement of Mask)

0000…0000	1111…1111	0000…0000	1111…1111

C and Mask

1111…1111	0000…0000	1111…1111	0000…0000

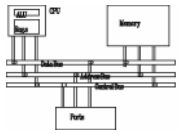
0	a2	0	a0

c3	0	c1	0

OR the two results to finish the overlay

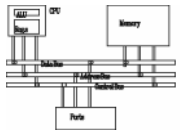
c3	a2	c1	a0

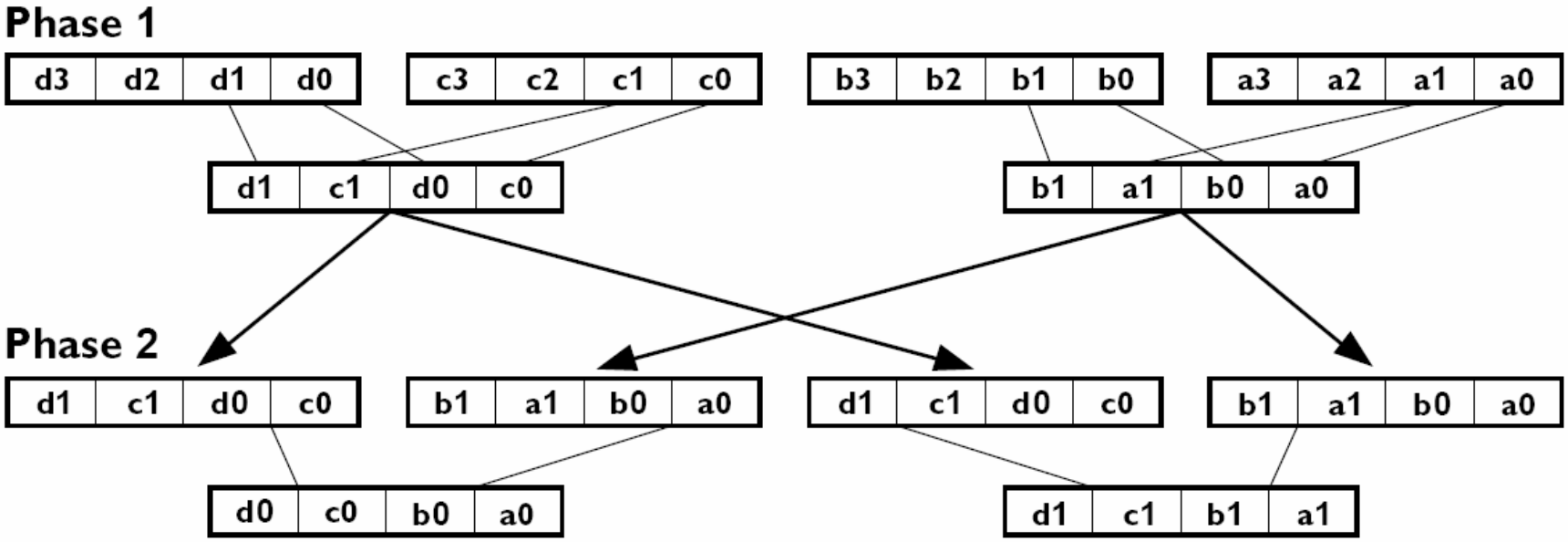
Application: sprite overlay



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

Application: matrix transport



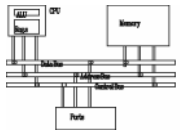


Note: Repeat for the other rows to generate ([d_3, c_3, b_3, a_3] and [d_2, c_2, b_2, a_2]).

MMX code sequence operation:

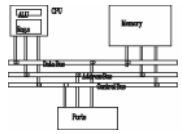
```
movq        mm1, row1    ; load pixels from first row of matrix
movq        mm2, row2    ; load pixels from second row of matrix
movq        mm3, row3    ; load pixels from third row of matrix
movq        mm4, row4    ; load pixels from fourth row of matrix
punpcklwd   mm1, mm2     ; unpack low order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

Application: matrix transport



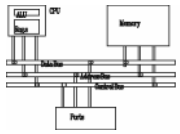
```
char M1[4][8];// matrix to be transposed
char M2[8][4];// transposed matrix
int n=0;
for (int i=0;i<4;i++)
  for (int j=0;j<8;j++)
    { M1[i][j]=n; n++; }
__asm{
//move the 4 rows of M1 into MMX registers
movq mm1,M1
movq mm2,M1+8
movq mm3,M1+16
movq mm4,M1+24
```

Application: matrix transport



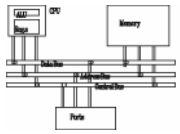
```
//generate rows 1 to 4 of M2
punpcklbw mm1, mm2
punpcklbw mm3, mm4
movq mm0, mm1
punpcklwd mm1, mm3 //mm1 has row 2 & row 1
punpckhwd mm0, mm3 //mm0 has row 4 & row 3
movq M2, mm1
movq M2+8, mm0
```

Application: matrix transport



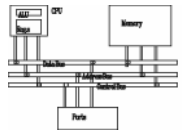
```
//generate rows 5 to 8 of M2
movq mm1, M1 //get row 1 of M1
movq mm3, M1+16 //get row 3 of M1
punpckhbw mm1, mm2
punpckhbw mm3, mm4
movq mm0, mm1
punpcklwd mm1, mm3 //mm1 has row 6 & row 5
punpckhwd mm0, mm3 //mm0 has row 8 & row 7
//save results to M2
movq M2+16, mm1
movq M2+24, mm0
emms
} //end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

SSE



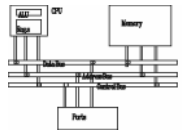
- Adds eight 128-bit registers
- Allows SIMD operations on packed single-precision floating-point numbers.

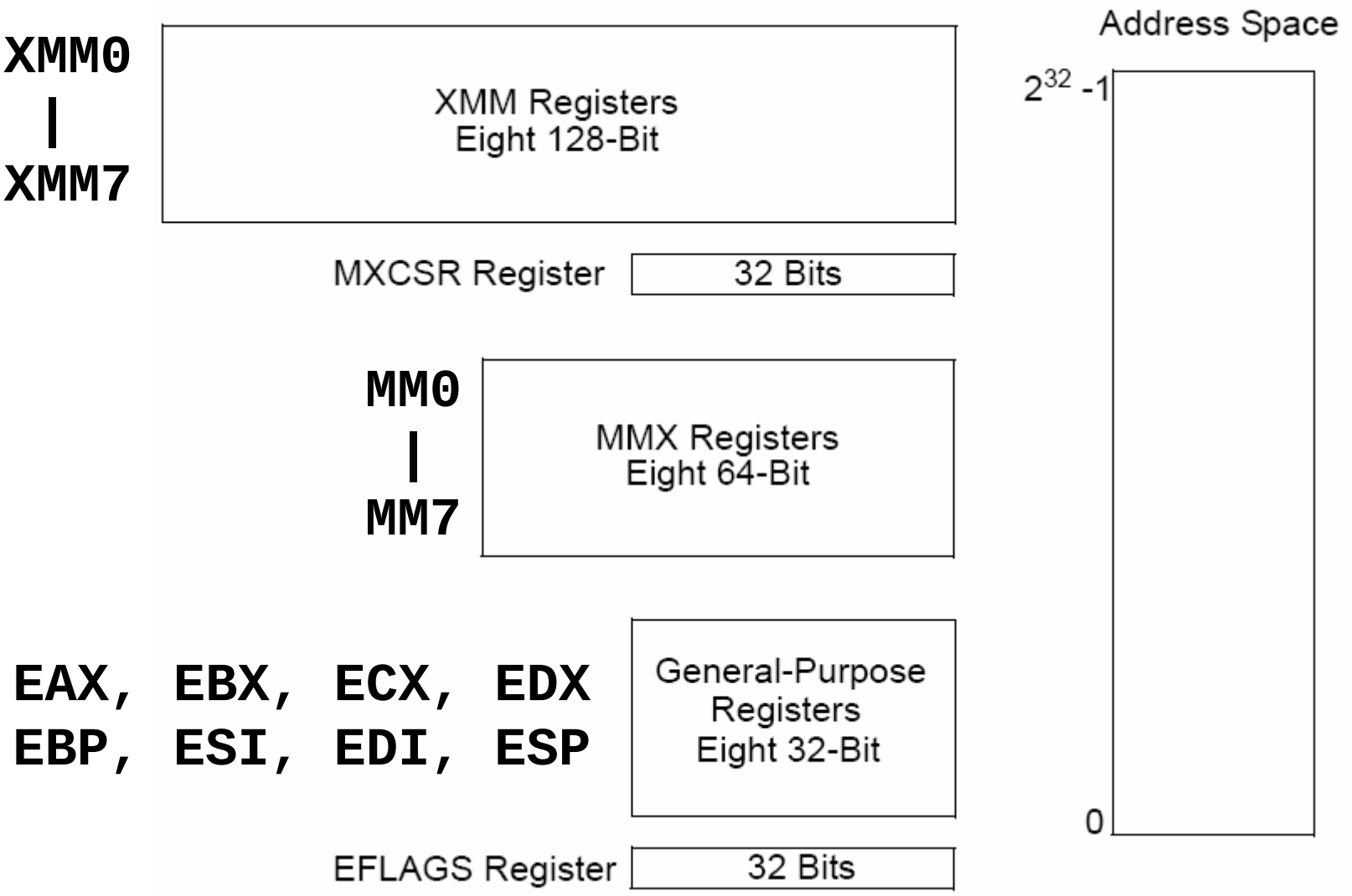
SSE features

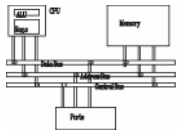


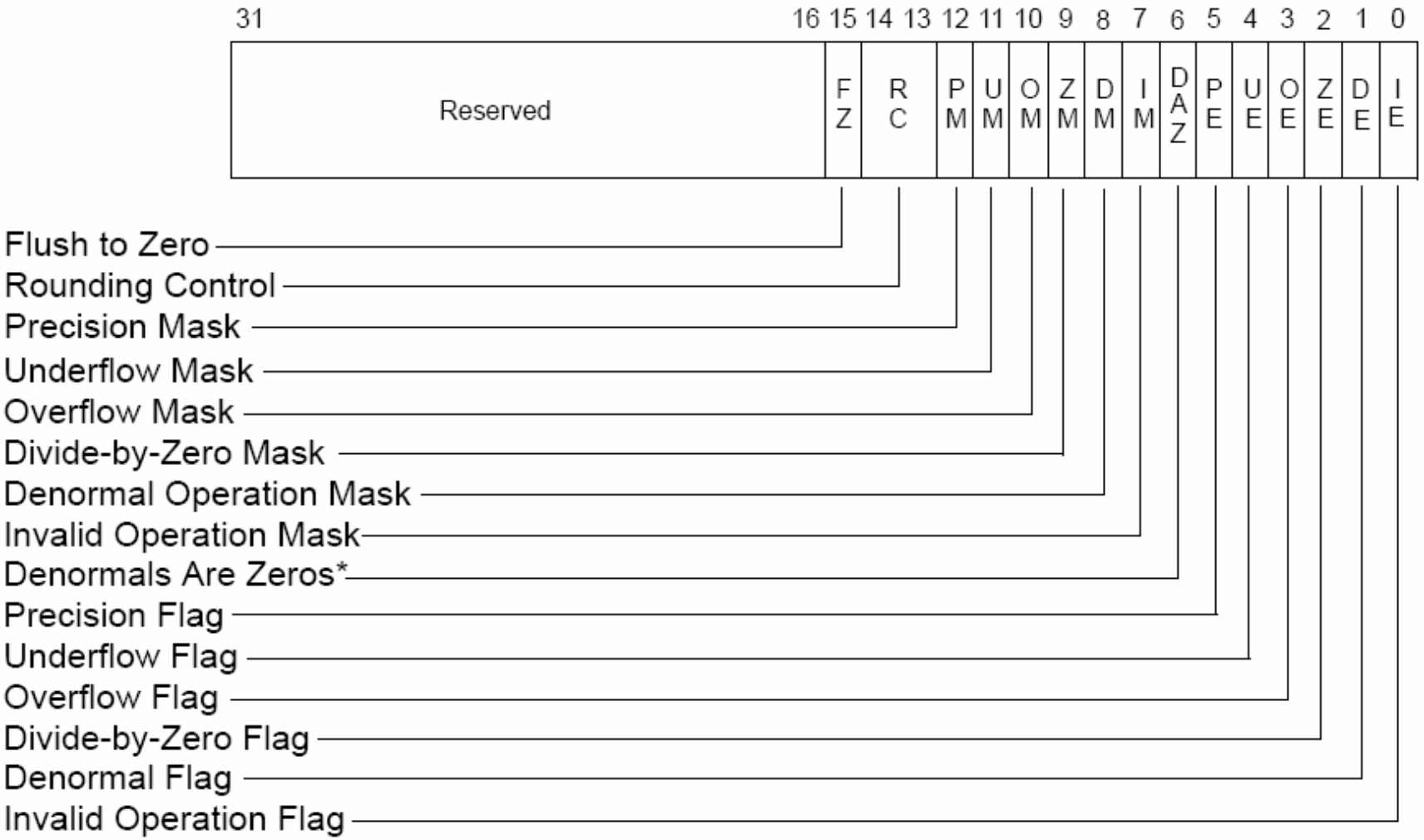
- Add eight 128-bit data registers (XMM registers) in non-64-bit modes; sixteen XMM registers are available in 64-bit mode.
- 32-bit MXCSR register (control and status)
- Add a new data type: 128-bit packed single-precision floating-point (4 FP numbers.)
- Instruction to perform SIMD operations on 128-bit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

SSE programming environment

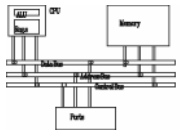


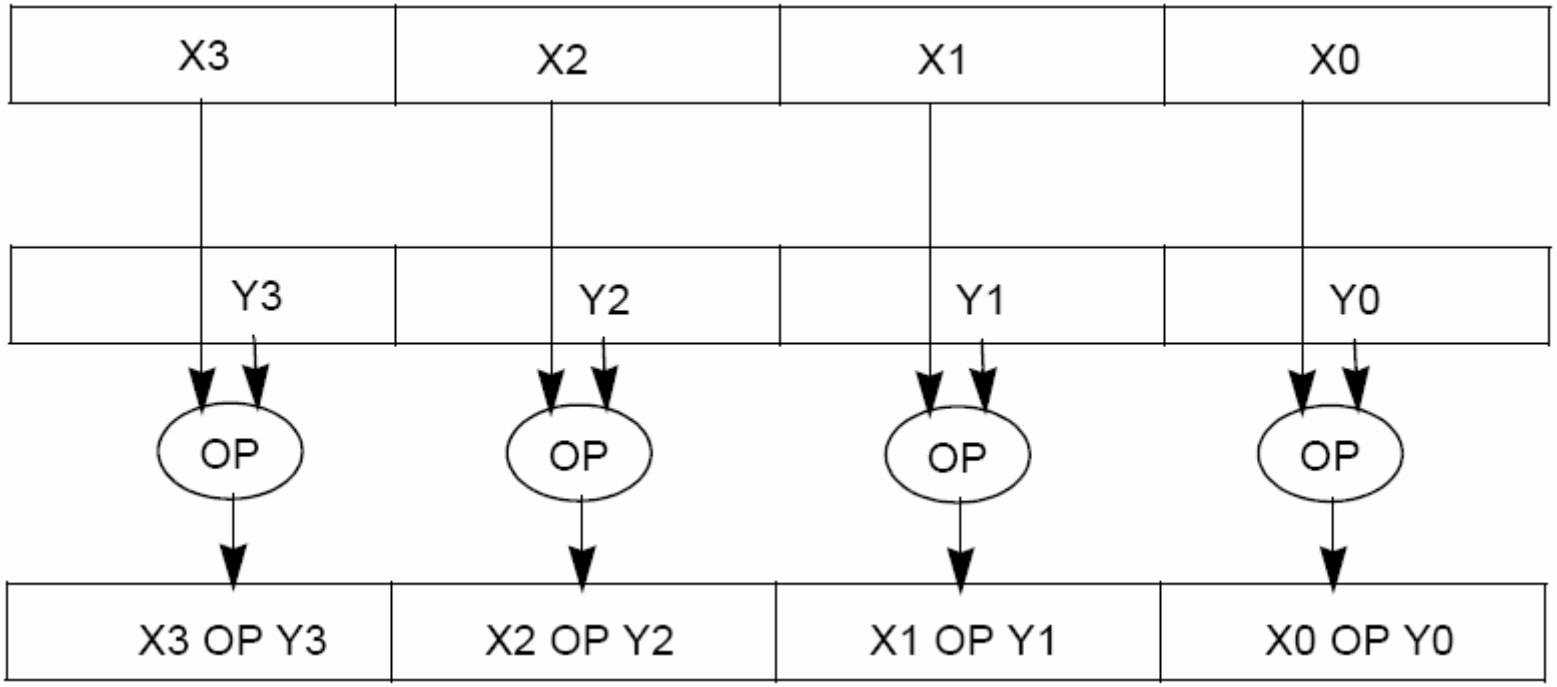





31
16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Reserved
FZ
RC
PM
UM
OM
ZM
DM
IM
DAZ
PE
UE
OE
ZE
DE
IE
Flush to Zero
Rounding Control
Precision Mask
Underflow Mask
Overflow Mask
Divide-by-Zero Mask
Denormal Operation Mask
Invalid Operation Mask
Denormals Are Zeros*
Precision Flag
Underflow Flag
Overflow Flag
Divide-by-Zero Flag
Denormal Flag
Invalid Operation Flag

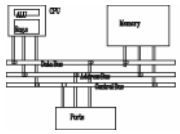
SSE packed FP operation

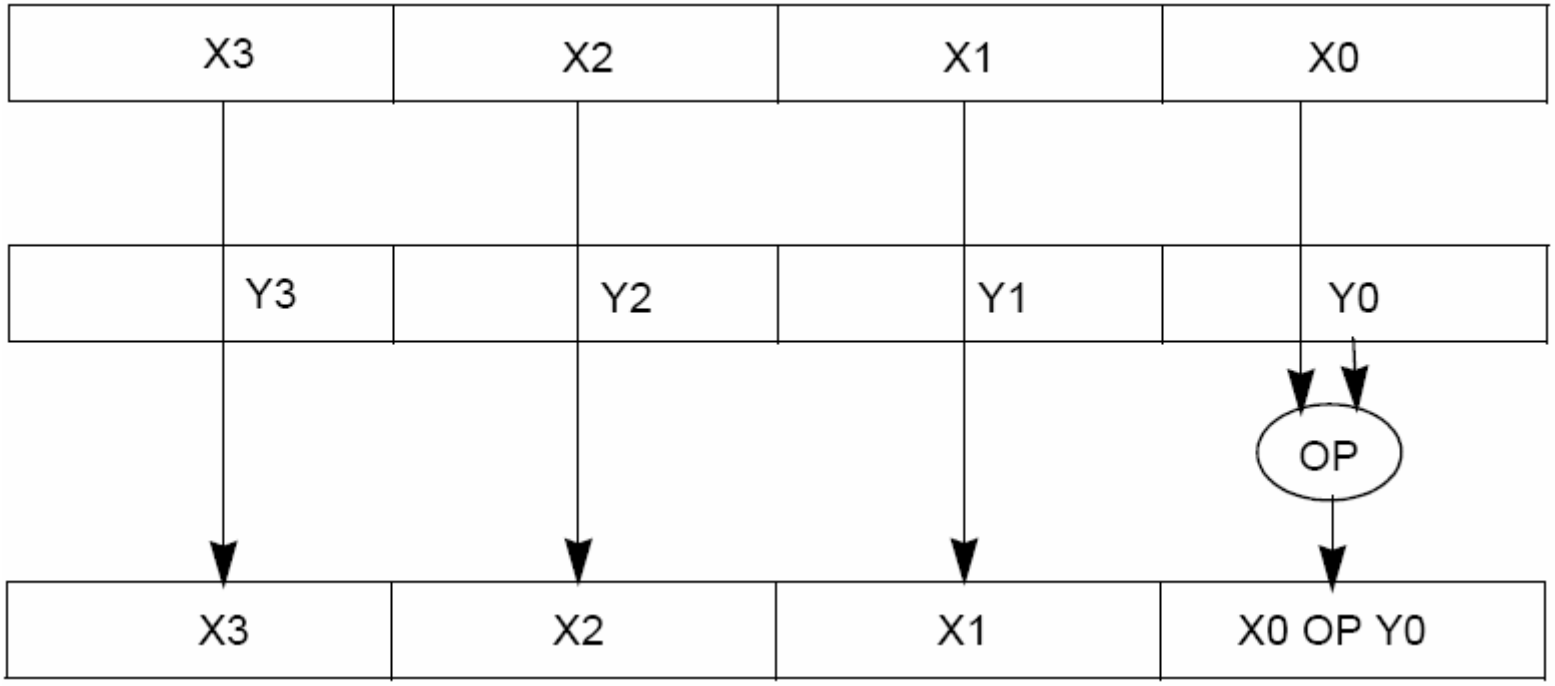




- ADDPS/ADDSS: add packed single-precision FP

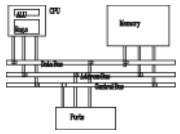
SSE scalar FP operation





- ADDSS/SUBSS: add scalar single-precision FP

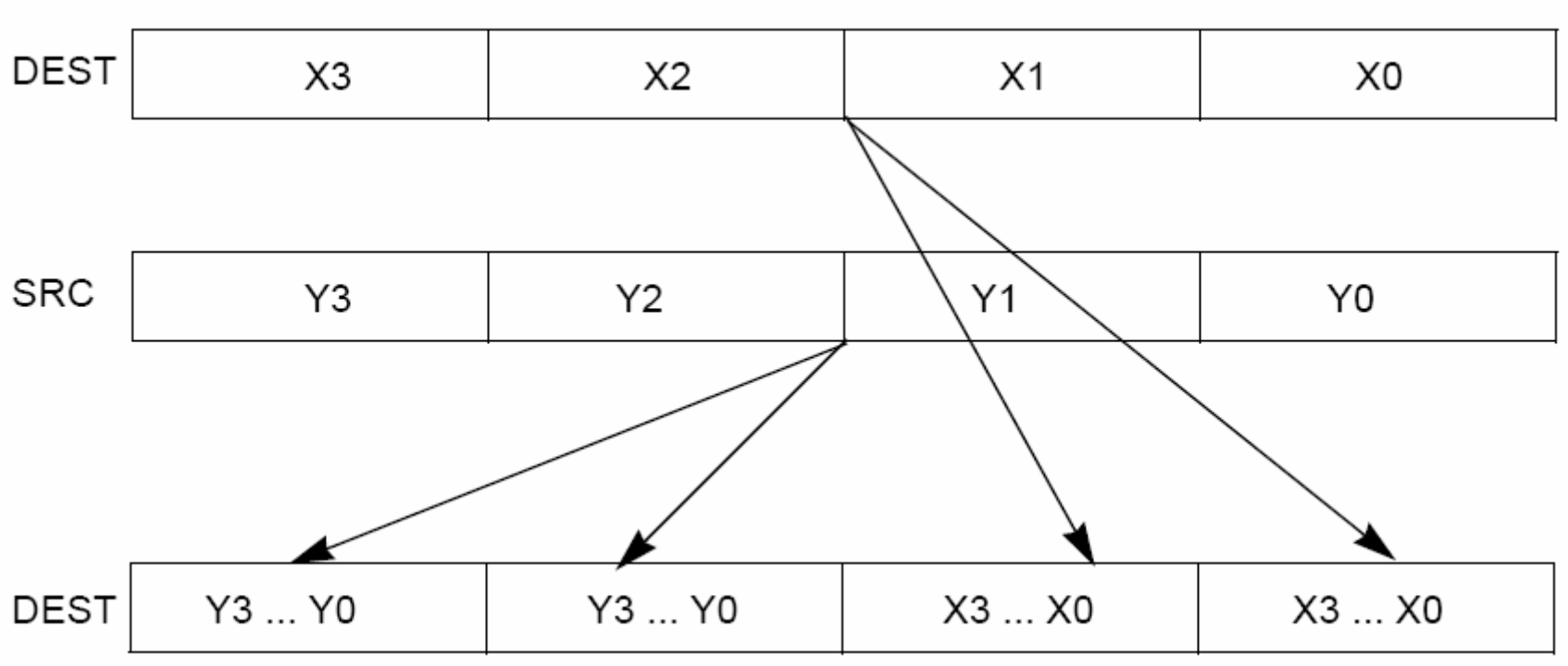
SSE Shuffle (SHUFPS)



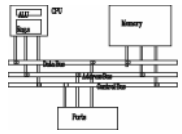
SHUFPS xmm1, xmm2, imm8

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

...

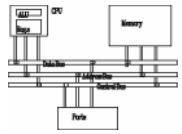


SSE2

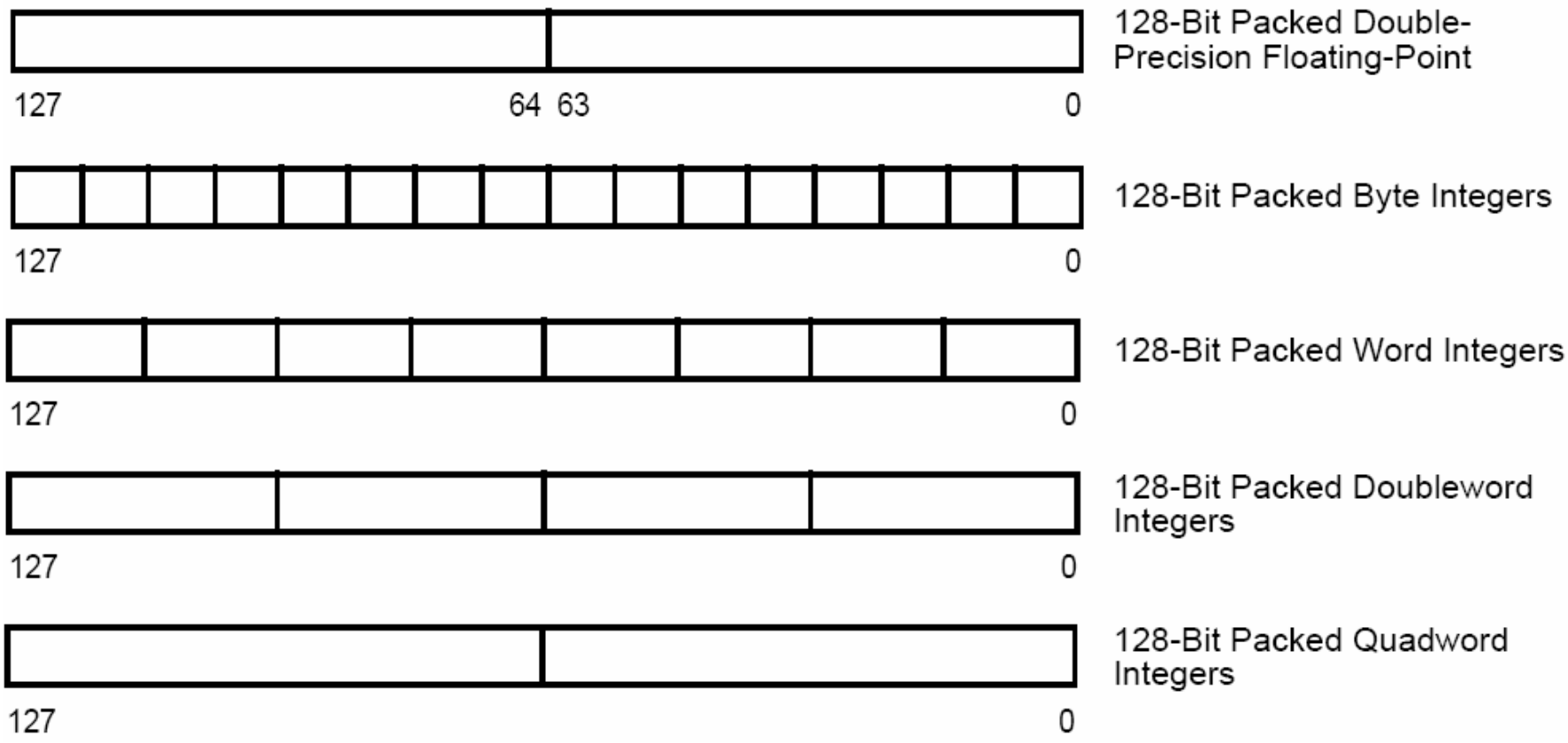


- Provides ability to perform SIMD operations on double-precision FP, allowing advanced graphics such as ray tracing
- Provides greater throughput by operating on 128-bit packed integers, useful for RSA and RC5

SSE2 features

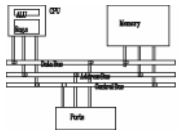


- Add data types and instructions for them



- Programming environment unchanged

Example

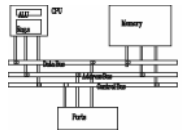


```
void add(float *a, float *b, float *c) {
  for (int i = 0; i < 4; i++)
    c[i] = a[i] + b[i];
}
__asm {
mov    eax, a
mov    edx, b
mov    ecx, c
movaps xmm0, XMMWORD PTR [eax]
addps  xmm0, XMMWORD PTR [edx]
movaps XMMWORD PTR [ecx], xmm0
}
```

movaps: move aligned packed single-precision FP

addps: add packed single-precision FP

Example: dot product



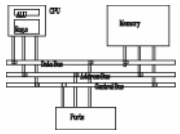
- Given a set of vectors $\{v_1,v_2,...v_n\}=\{(x_1,y_1,z_1), (x_2,y_2,z_2),..., (x_n,y_n,z_n)\}$ and a vector $v_c=(x_c,y_c,z_c)$, calculate $\{v_c \cdot v_i\}$
- Two options for memory layout
- Array of structure (AoS)

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

- Structure of array (SoA)

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

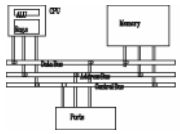
Example: dot product (AoS)



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

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

Example: dot product (AoS)



```
; 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)…
```