

ECE390

Computer Engineering II

Lecture 16



Dr. Zbigniew Kalbarczyk
University of Illinois at Urbana- Champaign

Lecture outline



- MMX Instructions
- Example code using MMX

MMX MultiMedia eXtensions

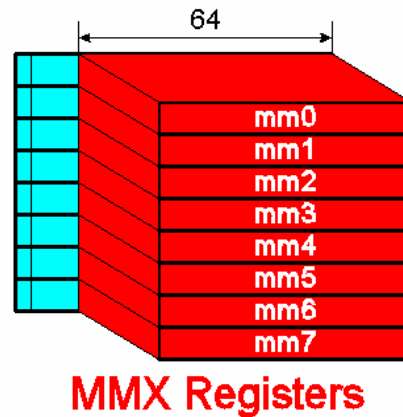
- Designed to accelerate multimedia and communication applications
 - motion video, image processing, audio synthesis, speech synthesis and compression, video conferencing, 2D and 3D graphics
- Includes new instructions and data types to significantly improve application performance
- Exploits the parallelism inherent in many multimedia and communications algorithms
- Maintains full compatibility with existing operating systems and applications

MMX MultiMedia eXtensions

- Provides a set of basic, general purpose integer instructions
- Single Instruction, Multiple Data (SIMD) technique
 - allows many pieces of information to be processed with a single instruction, providing parallelism that greatly increases performance
- 57 new instructions
- Four new data types
- Eight 64-bit wide MMX registers
- First available in 1997
- Supported on:
 - Intel Pentium-MMX, Pentium II, Pentium III (and later)
 - AMD K6, K6-2, K6-3, K7 (and later)
 - Cyrix M2, MMX-enhanced MediaGX, Jalapeno (and later)

Internal Register Set of MMX Technology

- Uses 64-bit mantissa portion of 80-bit FPU registers
- This technique is called aliasing because the floating-point registers are shared as the MMX registers
- Aliasing the MMX state upon the floating point state does not preclude applications from executing both MMX technology instructions and floating point instructions
- The same techniques used by FPU to interface with the operating system are used by MMX technology
 - preserves FSAVE/FRSTOR instructions



Data Types

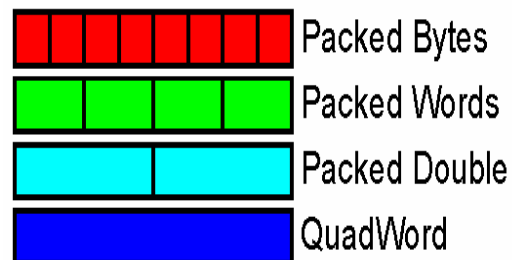
- MMX architecture introduces new packed data types
- Multiple integer words are grouped into a single 64-bit quantity

Eight 8-bit packed bytes (B)

Four 16-bit packed words (W)

Two 32-bit packed doublewords (D)

One 64-bit quadword (Q)



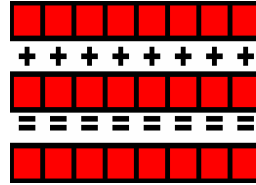
- Example: consider graphics pixel data represented as bytes.
 - with MMX, eight of these pixels can be packed together in a 64-bit quantity and moved into an MMX register
 - MMX instruction performs the arithmetic or logical operation on all eight elements in parallel

Arithmetic Instructions

- PADD(B/W/D): Addition

PADDB MM1, MM2

adds 64-bit contents of MM2 to MM1,
byte-by-byte any carries generated
are dropped, e.g., byte A0h + 70h = 10h

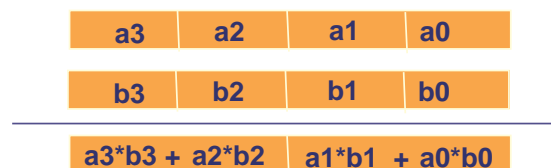


MMX Addition:
Each 8-byte
entity added
in parallel

- PSUB(B/W/D): Subtraction

Arithmetic Instructions

- PMUL(L/H)W: Multiplication (Low/High Result)
 - multiplies four pairs of 16-bit operands, producing 32 bit result
- PMADDWD: Multiply and Add



- Key instruction to many signal processing algorithms like matrix multiplies or FFTs

Logical, Shifting, and Compare Instructions

Logical

PAND: Logical AND (64-bit)

POR: Logical OR (64-bit)

PXOR: Logical Exclusive OR (64-bit)

PANDN: Destination = (NOT Destination) AND Source

Shifting

PSLL(W/D/Q): Packed Shift Left (Logical)

PSRL(W/D/Q): Packed Shift Right (Logical)

PSRA(W/D/Q): Packed Shift Right (Arithmetic)

Compare

PCMPEQ: Compare for Equality

PCMPGT: Compare for Greater Than

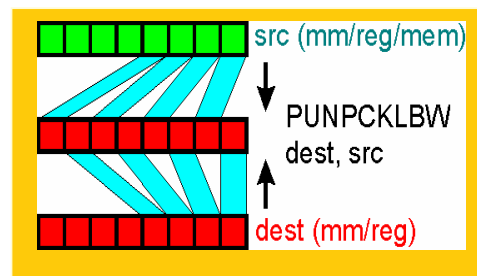
Sets Result Register to ALL 0's or ALL 1's

Conversion Instructions Unpacking

- Unpacking (Increasing data size by 2^n bits)

PUNPCKLBW: Reg1, Reg2:

Unpacks lower four bytes to create four words.



PUNPCKLWD: Reg1, Reg2:

Interleaves lower two words to create two doubles

PUNPCKLDQ: Reg1, Reg2:

Interleaves lower double to create Quadword

Conversion Instructions Packing

- Packing (Reducing data size by 2^n bits)
PACKSSDW Reg1, Reg2: Pack Double to Word
Four doubles in Reg2:Reg1 compressed to Four words in Reg1

PACKSSWB Reg1, Reg2: Pack Word to Byte
Eight words in Reg2:Reg1 compressed to Eight bytes in Reg1
- The pack and unpack instructions are especially important when an algorithm needs higher precision in its intermediate calculations, e.g., an image filtering

Data Transfer Instructions

- MOVQ Dest, Source : 64-bit move
 - One or both arguments must be a MMX register
- MOVD Dest, Source : 32-bit move
 - Zeros loaded to upper MMX bits for 32-bit move

Saturation/Wraparound Arithmetic

- Wraparound: carry bit lost (significant portion lost) (PADD)

$$\begin{array}{r} \begin{array}{|c|c|c|c|} \hline a3 & a2 & a1 & \text{FFFFh} \\ \hline \end{array} \\ + \\ \begin{array}{|c|c|c|c|} \hline b3 & b2 & b1 & 8000h \\ \hline \end{array} \\ \hline \begin{array}{|c|c|c|c|} \hline a3+b3 & a2+b2 & a1+b1 & 7FFFh \\ \hline \end{array} \end{array}$$

Saturation/Wraparound Arithmetic (cont.)

- Unsigned Saturation: add with unsigned saturation (PADDUS)

$$\begin{array}{r} \begin{array}{|c|c|c|c|} \hline a3 & a2 & a1 & \text{FFF4h} \\ \hline \end{array} \\ + \\ \begin{array}{|c|c|c|c|} \hline b3 & b2 & b1 & 1123h \\ \hline \end{array} \\ \hline \begin{array}{|c|c|c|c|} \hline a3+b3 & a2+b2 & a1+b1 & \text{FFFFh} \\ \hline \end{array} \end{array}$$

Saturation

if addition results in overflow or subtraction results in underflow, the result is clamped to the largest or the smallest value representable

- for an unsigned, 16-bit word the values are FFFFh and 0000h
- for a signed 16-bit word the values are 7FFFh and 8000h

Saturation/Wraparound Arithmetic (cont.)

- Saturation: add with signed saturation (PADDS)

$$\begin{array}{r} \begin{array}{|c|c|c|c|} \hline a3 & a2 & a1 & 7FF4h \\ \hline \end{array} \\ + \\ \begin{array}{|c|c|c|c|} \hline b3 & b2 & b1 & 0050h \\ \hline \end{array} \\ \hline \begin{array}{|c|c|c|c|} \hline a3+b3 & a2+b2 & a1+b1 & 7FFFh \\ \hline \end{array} \end{array}$$

Adding 8 8-bit Integers with Saturation

X0 dq **8080555580805555**

X1 dq **009033FF009033FF**

MOVQ mm0, [X0]

PADDSB mm0, [X1]

Result mm0 = **80807F5480807F54**

80h+00h=80h (addition with zero)
80h+90h=80h (saturation to maximum negative value)
55h+33h=7fh (saturation to maximum positive value)
55h+FFh=54h (subtraction by one)

Use of FPU Registers for Storing MMX Data

- The EMMS (empty MMX-state) instruction sets (11) all the tags in the FPU, so the floating-point registers are listed as empty
- EMMS must be executed before the return instruction at the end of any MMX procedure
 - otherwise any subsequent floating point operation will cause a floating point interrupt error, potentially crashing your application
 - if you use floating point within MMX procedure, you must use EMMS instruction before executing the floating point instruction
- Any MMX instruction resets (00) all FPU tag bits, so the floating-point registers are listed as valid

Example MMX Program

```
GLOBAL _main
SECTION .bss ;=====
                ; Uninitialized data

SECTION .data ;=====
    Array_1 db 01h, 02h, 03h, 04h, 05h, 06h, 07h, 08h
             db 01h, 01h, 01h, 01h, 01h, 01h, 01h, 01h
             db 08h, 09h, 0Ah, 0Bh, 0Ch, 0Dh, 0Eh, 0Fh

    Array_2 db 09h, 09h, 09h, 09h, 09h, 09h, 09h, 09h
             db 00h, 01h, 02h, 03h, 04h, 05h, 06h, 07h
             db 80h, 90h, 0A0h, 0B0h, 0C0h, 0D0h, 0E0h, 0F0h

    ClearMMX dd 00h, 00h
```

Example MMX Program

```

SECTION .text ;=====
_main
    mov ebx, Array_1
    mov edx, Array_2
    mov ecx, 3

;call _MMX_Reset ;Reset MMX registers to zero

.Add_Array
    movq mm0, [ebx+8*ecx-8]
    paddb mm0, [edx+8*ecx-8]
    movq qword [edx+8*ecx-8], mm0 ;Store the sum
    dec ecx
    cmp ecx, 0
    ja .Add_Array
;loop .Add_Array

;Move sums to MMX
    movq mm5, [Array_2]
    movq mm6, [Array_2+8]
    movq mm7, [Array_2+16]
    emms
    ret

_MMX_Reset
    movq mm0, [ClearMMX]
    movq mm1, mm0
    movq mm2, mm0
    movq mm3, mm0
    movq mm4, mm0
    movq mm5, mm0
    movq mm6, mm0
    movq mm7, mm0
    ret

```

Z. Kalbarczyk

ECE390

19

Parallel Compare

- PCMPGT[B/W/D] mmxreg, r/m64

23	45	16	34
Gt?	Gt?	gt/?	Gt?
31	7	16	67
0000h	FFFFh	0000h	0000h

- PCMPEQ[B/W/D] mmxreg, r/m64

23	45	16	34
Eq?	Eq?	Eq?	Eq?
31	7	16	67
0000h	0000h	FFFFh	0000h

Z. Kalbarczyk

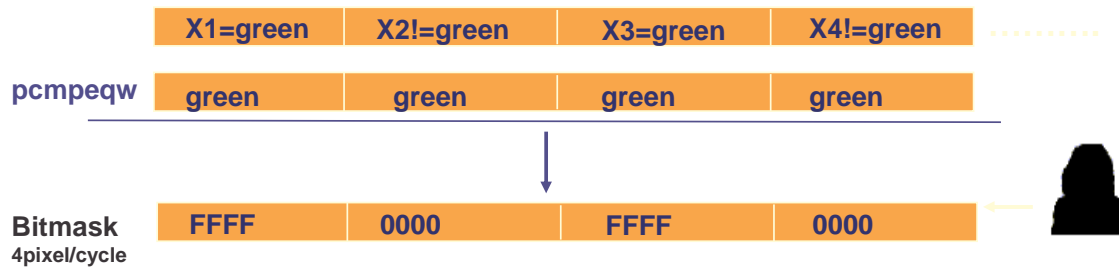
ECE390

20

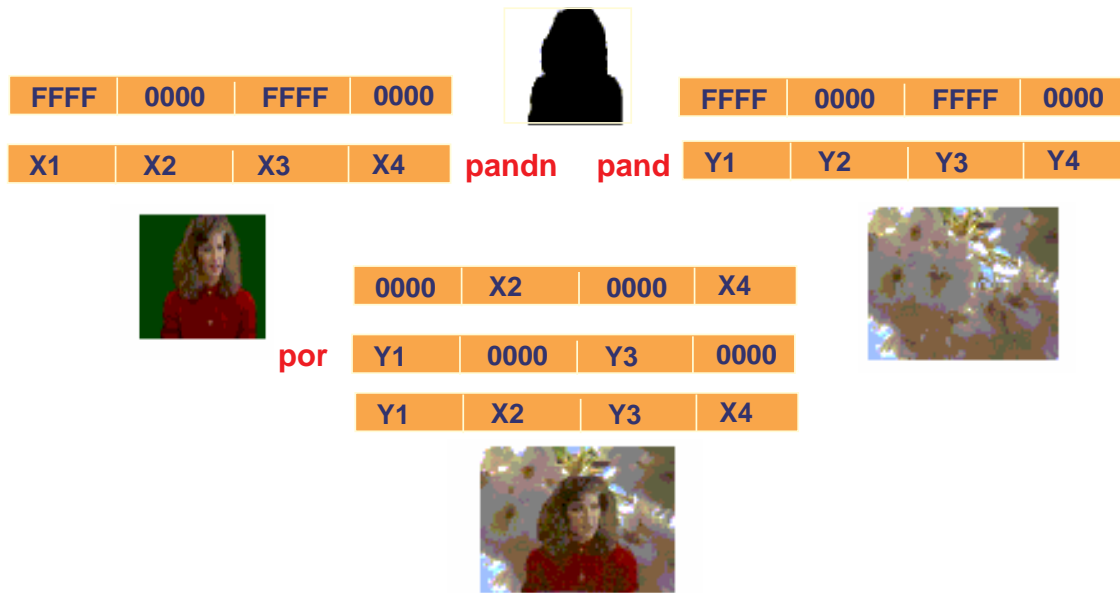
Overlaid of the Image



Overlaid of the Image



Overlaid of the Image

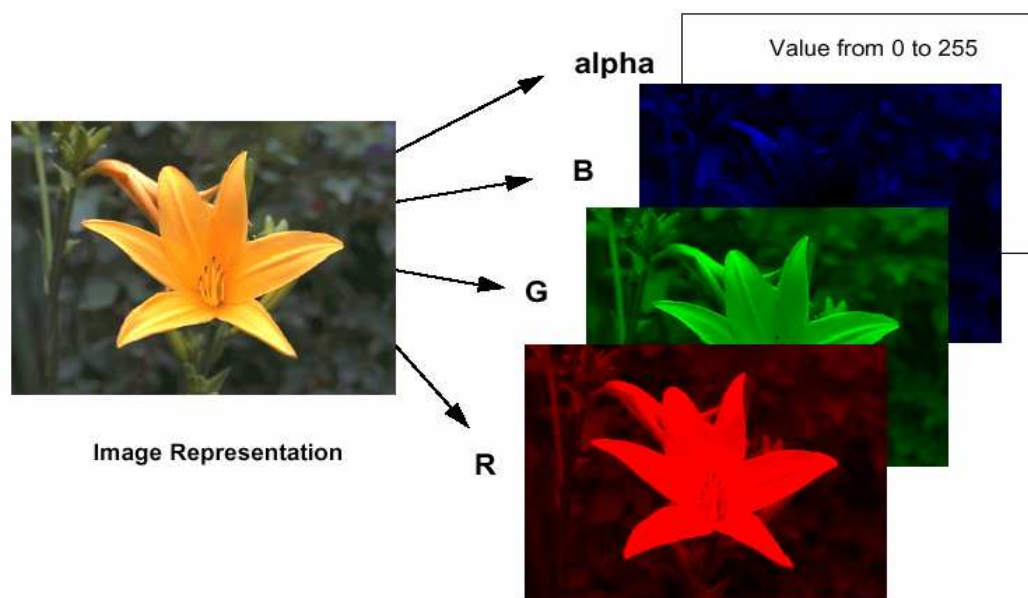


Z. Kalbarczyk

ECE390

23

Image Dissolve Using Alpha Blending



Z. Kalbarczyk

ECE390

24

Image Dissolve Using Alpha Blending

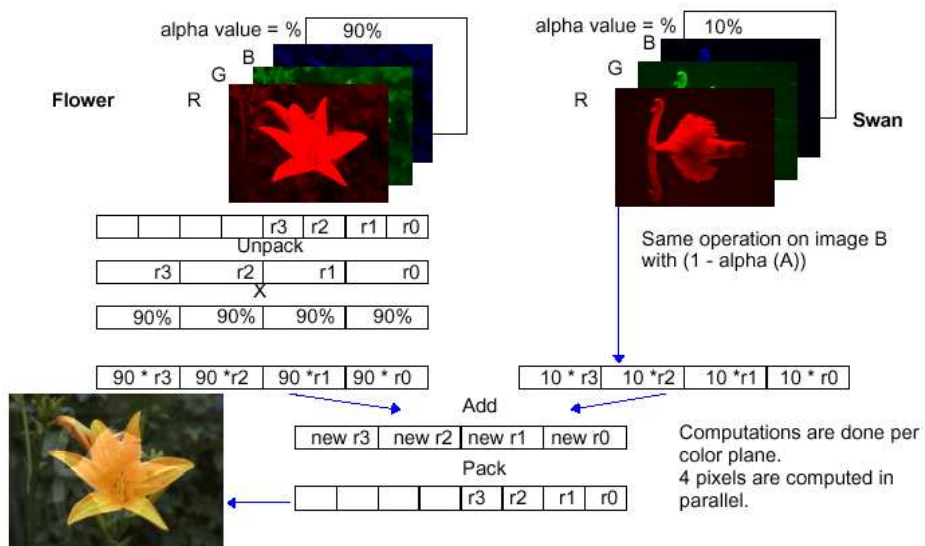
- MMX instructions speed up image composition
- A flower will dissolve into a swan
- Alpha determines the intensity of the flower
- The full intensity, the flower's 8-bit alpha value is FFh, or 255
- The equation below calculates each pixel:

$$\text{Result_pixel} = \text{Flower_pixel} * (\text{alpha}/255) + \text{Swan_pixel} * [1 - (\text{alpha}/255)]$$

For alpha 230, the resulting pixel is 90% flower and 10% swan



Image Dissolve Using Alpha Blending



Instruction Count with/without MMX Technology

Operation	Calculation without MMX™ Technology	Number of Instructions without MMX Technology	Number of MMX Instructions
Load	$(640*480)*255*3*2$	470 million	117 million
Unpack	-	-	117 million
Multiply	$(640*480)*255*3*2$	470 million	117 million
Add	$(640*480)*255*3$	235 million	58 million
Pack	-	-	58 million
Store	$(640*480)*255*3$	235 million	58 million
Total		1.4 billion	525 million

Alpha Blending Techniques

Dissolve (combine)	Fade in, fade out effect $A * \alpha(A) + B * (1-\alpha)$
A over B	Transparent image place on the background $A + (B * (a - \alpha(A)))$
A in B	Image A only where B has Opacity $A * \alpha(B)$
A out B	Image A only where B has transparency $A * (1 - \alpha(B))$
A top B	(A in B) over B $(A * \alpha(B)) + (B * (1-\alpha(A)))$
A XOR B	$(B * (1 - \alpha(A))) + (A * (1 - \alpha(B)))$

Streaming SIMD Extensions Intel Pentium III

- Streaming SIMD defines a new architecture for floating point operations
- Operates on IEEE-754 Single-precision 32-bit Real Numbers
- Uses eight new 128-bit wide general-purpose registers (XMM0 - XMM7)
- Introduced in Pentium III in March 1999
 - Pentium III includes floating point, MMX technology, and XMM registers

Streaming SIMD Extensions Intel Pentium III (cont.)

- Supports packed and scalar operations on the new packed single precision floating point data types
- **Packed** instructions operate vertically on four pairs of floating point data elements in parallel
 - instructions have suffix **ps**, e.g., addps
- **Scalar** instructions operate on the least-significant data elements of the two operands
 - instructions have suffix **ss**, e.g., addss