# ECE291

Lecture 13

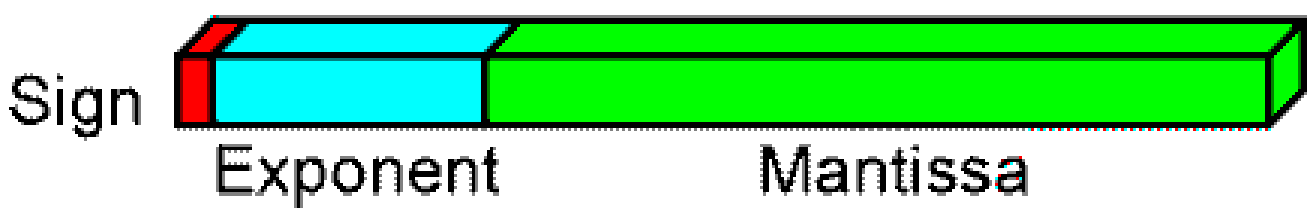
*Advanced Mathematics*

# Lecture outline

- Floating point arithmetic
- 80x87 Floating Point Unit
- MMX Instructions
- SSE Instructions

# Floating-point numbers

- Components of a floating point number
  - **Sign:** 0=pos(+); 1=neg(-)
  - **Exponent:** (with Constant bias added)
  - **Mantissa:** 1 <= M < 2
- Number = (Mantissa) * $2^{(\text{exponent-bias})}$
- Example: 10.25 (decimal)
  = 1010.01 (binary)
  = + 1.01001 * $2^3$ (normalized)



# Floating-point format

| Precision | Size (bits) | Exp. (bits) | Bias | Mant. (bits) | Mant. 1st Digit |
|---|---|---|---|---|---|
| Single | 32 | 8 | 127 (7Fh) | 23 | Implied |
| Double | 64 | 11 | 1023 (3FFh) | 52 | Implied |
| Extended | 80 | 15 | 16383 (3FFFh) | 64 | Explicit |

# Converting to 32-bit floating-point form

- Convert the decimal number into binary
- Normalize the binary number
- Calculate the biased exponent
- Store the number in floating point format
- Example:

1. 100.25 = 01100100.01
2. 1100100.01 = 1.10010001 x $2^6$
3. 110 + 01111111(7Fh) = 10000101(85h)
4. **Sign** = 0
   **Exponent** = 10000101
   **Mantissa** = 100 1000 1000 0000 0000 0000

**Note**

**mantissa of a number 1.XXXX is the XXXX**
**The 1. is an implied one-bit that is only stored in the extended precision form of the floating-point number as an explicit-one bit**

# Converting to floating-point form

| Format | Sign | Exponent | Mantissa |
|---|---|---|---|
| **Single** | (+) | 7F+6 = 85h (8 bits) | Implied 1 + (23 bits) |
| | 0 | 1000,0101 | 100 1000 1000 0000 0000 0000 |
| **Double** | (+) | 3FFh + 6 = 405h (11 bits) | Implied 1 + (52 bits) |
| | 0 | 100,0000,0101 | 100 1000 1000 0000 … 0000 |
| **Extended** | (+) | 3FFFh+6 = 4005h (15 bits) | Explicit 1 + (63 bits) |
| | 0 | 100,0000,0000,0101 | 1 100 1000 1000 0000 … 0000 |

# Special representations

- **Zero:**
  - Exp = All Zero, Mant = All Zero
- **NAN (not a number)** an invalid floating-point result
  - Exp = All Ones, Mant  non-zero
- **+ Infinity:**
  - Sign = 0, Exp = All Ones, Mant = 0
- **- Infinity:**
  - Sign = 1, Exp = All Ones, Mant = 0

# Converting from Floating-Point Form (example)

**Sign** = 1

**Exponent** = 1000,0011

**Mantissa** = 100,1001,0000,0000,0000

100 = 1000,0011 - 0111,1111 (convert biased exponent)

$1.1001001 \times 2^4$ (a normalized binary number)

11001.001 (a de-normalized binary number)

-25.125 ( a decimal number)

# Variable declaration examples

- Declaring and Initializing Floating Point Variables

```
Fpvar1      dd      -7.5      ; single precision (4 bytes)
Fpvar2      dq      0.5625    ; double precision (8 bytes)
Fpvar3      dt      -247.6    ; extended precision (10 bytes)
Fpvar4      dt      345.2     ; extended precision (10 bytes)
Fpvar5      dt      1.0       ; extended precision (10 bytes)
```

# The 80x87 floating point unit

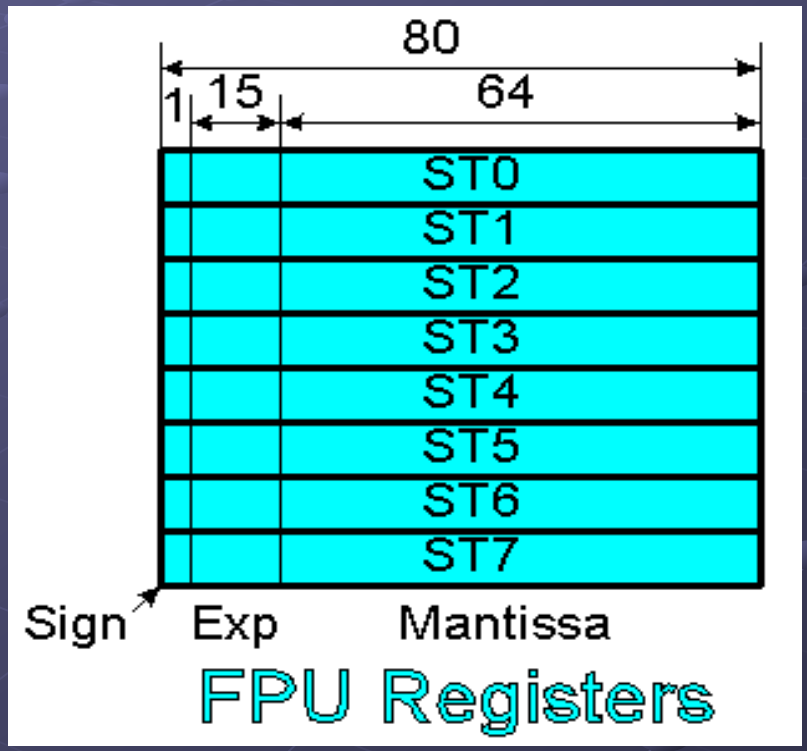
- On-chip hardware for fast computation on floating point numbers
- FPU operations run in parallel with integer operations
- Historically implemented as separate chip
  - 8087-80387 Coprocessor
- 80486DX-Pentium III contain their own internal and fully compatible versions of the 80387
- CPU/FPU Exchange data through memory
  - X87 always converts Single and Double to Extended-precision

# Arithmetic Coprocessor Architecture

- Control unit
  - interfaces the coprocessor to the microprocessor-system data bus
- Numeric execution unit (NEU)
  - executes all coprocessor instructions
- NEU has eight-register stack
  - each register is 80-bit wide (extended-precision numbers)
  - data appear as any other form when they reside in the memory system
  - holds operands for arithmetic instructions and the results of arithmetic instructions
- Other registers - status, control, tag, exception

# Arithmetic Coprocessor FPU Registers

- The eight 80-bit data registers are organized as a stack
- As data items are pushed into the top register, previous data items move into higher-numbered registers, which are lower on the stack
- All coprocessor data are stored in registers in the 10-byte real format
- Internally, all calculations are done on the greatest precision numbers
- Instructions that transfer numbers between memory and coprocessor automatically convert numbers to and from the 10-byte real format



# Sample 80x87 FPU Opcodes

FLD: Load (lets you load a memory address into an fp reg)
FST: Store (moves fp reg to memory address)
FADD: Addition
FMUL: Multiplication
FDIV: Division (divides the destination by the source)
FDIVR: Division (divides the source by the destination)
FSIN: Sine (uses radians)
FCOS: Cosine (uses radians)
FSQRT: Square Root
FSUB: Subtraction
FABS: Absolute Value
FYL2X: Calculates $Y * \log_2 (X)$ (Really)
FYL2XP1: Calculates $Y * \log_2(X+1)$ (Really)

# Programming the FPU

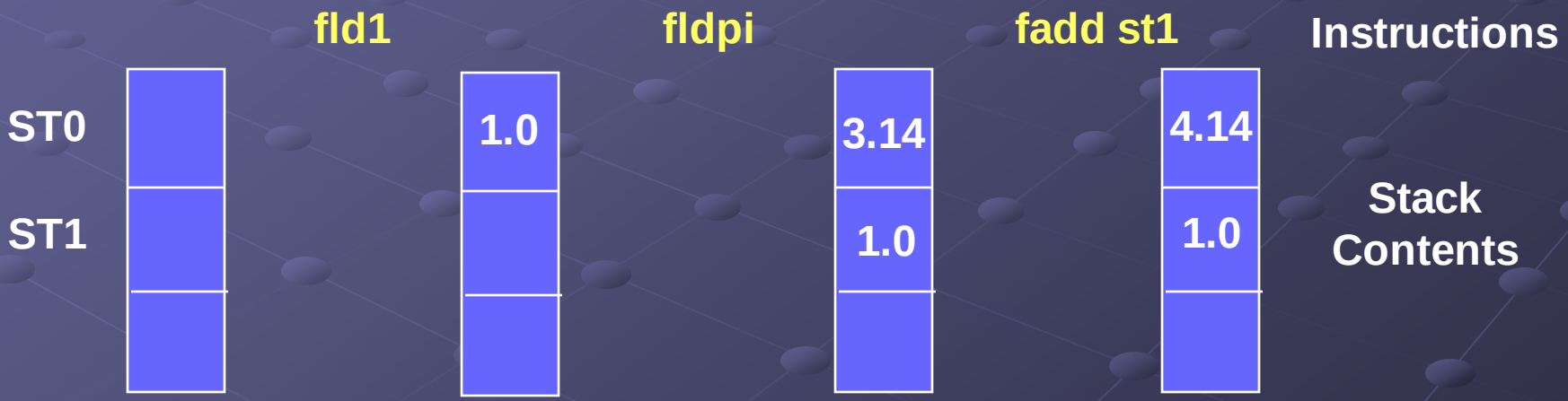
- FPU Registers = ST0 … ST7
- Format: OPCODE source
- Restrictions: Source can usually be ST# or memory location
- Use of ST0 as destination is implicit when not specified otherwise
- FINIT - Execute before starting to initialize FPU registers!

# Programming the FPU (cont.)

- FLD size[MemVar]
  - Load ST0 with MemVar &
  - Push all other STi to ST(i+1) for i=0..6
- FSTP size[MemVar]
  - Move top of stack (STO) to memory &
  - Pop all other STi to ST(i-1) for I = 1..7
- FST size[MemVar]
  - Move top of stack to memory
  - Leave result in ST0

# FPU Programming
# Classical-Stack Format

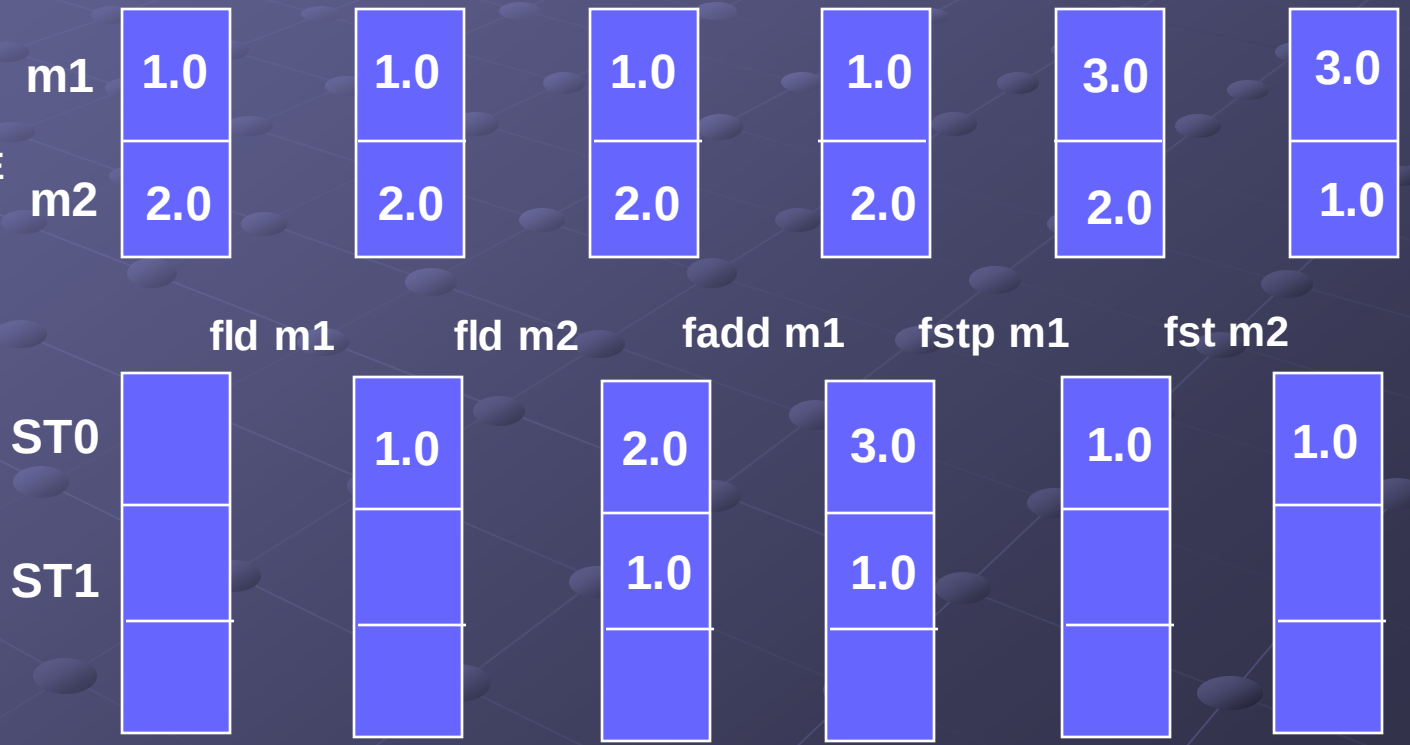
- Instructions treat the coprocessor registers like items on the stack
- ST0 (the top of the stack) is the source operand
- ST1, the second register, is the destination



# FPU Programming Memory Format

- The stack top (ST0) is always the implied destination

```
SEGMENT DATA
m1   DD  1.0
m2   DD  2.0
SEGMENT CODE
....
fld dword [m1]
fld dword [m2]
fadd dword [m1]
fstp dword [m1]
fst dword [m2]
```

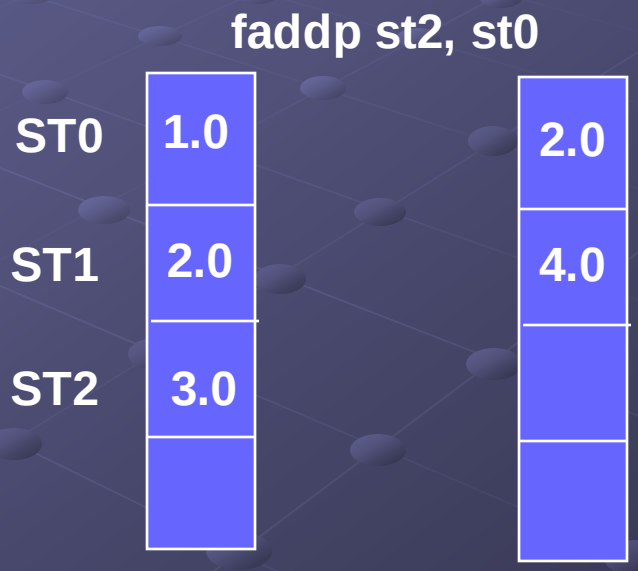


# FPU Programming Register Format

- Instructions treat coprocessor registers as registers rather than stack elements

# FPU Programming Register-Pop Format

- Coprocessor registers are treated as a modified stack
  - the source register must always be a stack top



# FPU Programming Timing Issues

- A processor instruction following a coprocessor instr
  - coordinated by assembler for 8086 and 8088
  - coordinated by processor on 80186 - Pentium
- A processor instruction that accesses memory following a coprocessor instruction that accesses the same memory
  - for 8087 need to include ***WAIT or FWAIT*** to ensure that coprocessor finishes before the processor begins
  - assembler did this automatically

# Calculating the area of a circle

```
RAD     DD   2.34, 5.66, 9.33, 234.5, 23.4
AREA    RESD  5

..START
    mov  si, 0      ;source element 0
    mov  di, 0      ;destination element 0
    mov  cx, 5      ;count of 5
    finit
Main1
    fld  dword [RAD + si]  ;radius to ST0
    fmul st0        ; square radius
    fldpi       ; pi to ST0
    fmul st1    ; multiply ST0 = ST0 x ST1
    fstp  dword [AREA + di]
    add   si, 4
    add   di, 4
    loop  Main1

    mov   ax, 4C00h
    int 21h
```

# Floating Point Operations Calculating Sqrt($x^2 + y^2$)

```
StoreFloat   resb    94
X            dd      4.0
Y            dd      3.0
Z            resd

%macro% ShowFP
      fsave  StoreFloat
      frstor StoreFloat
%endmacro%

..start
   mov     ax, cs    ; Initialize DS=CS
   mov     ds, ax

   finit
   ShowFP
   fld     dword [X]
   ShowFP
   fld     st0
   ShowFP
   fmulp   st1, st0
   ShowFP
   fld     dword [Y]
   ShowFP
   fld     st0
   ShowFP
   fmulp   st1, st0
   ShowFP

   fadd    st0, st1
   ShowFP
   fsqrt
   ShowFP

   fst     dword [Z]
   ShowFP

   mov     ax, 4C00h   ;Exit to DOS
   int     21h
```

# MMX **M**ulti**M**edia e**X**tentions

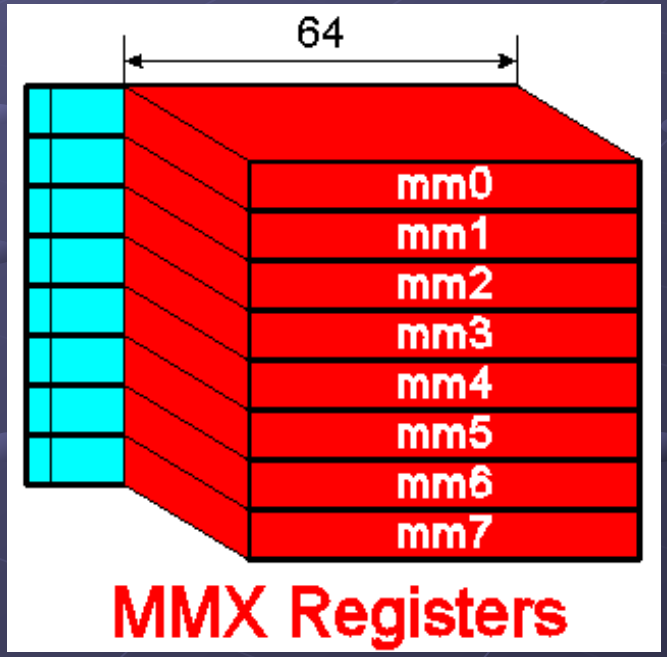
- Designed to accelerate multimedia and communication applications
  - motion video, image processing, audio synthesis, speech synthesis and compression, video conferencing, 2D and 3D graphics
- Includes new instructions and data types to significantly improve application performance
- Exploits the parallelism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

# MMX **Multi**Media e**X**tentions

- 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)

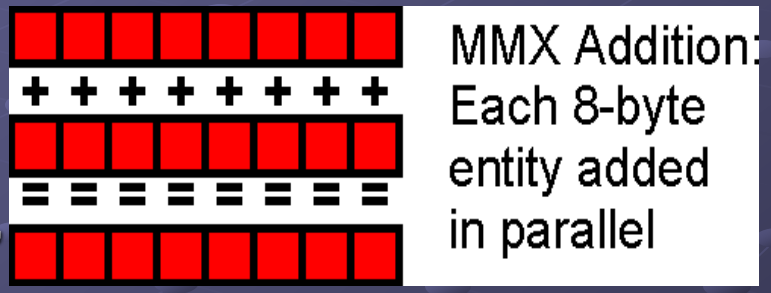
Two 32-bit packed doublewords (D)

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



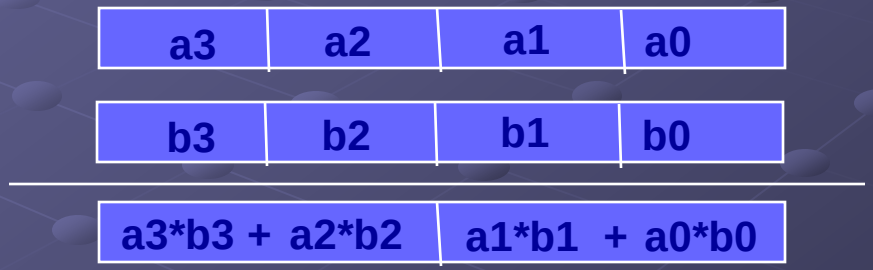

PADDB MM1, MM2

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

- 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

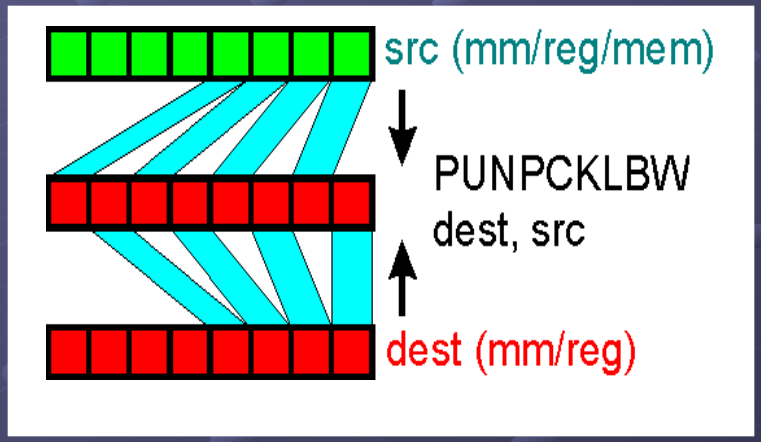
# Logical, Shifting, and Compare Instructions

- 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 Rigth (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

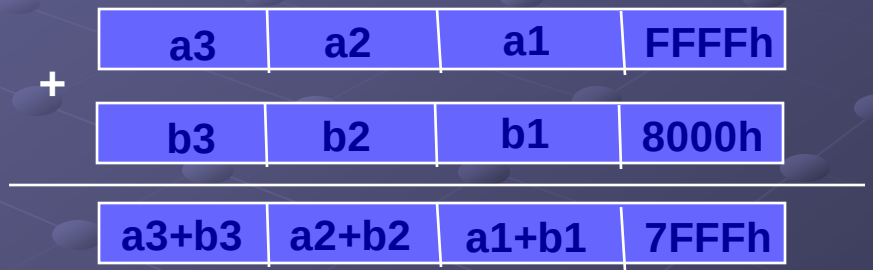
- The pack and unpack instructions are especially important when an algorithm needs higher precision in its intermediate calculations, e.g., an image filtering

# 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

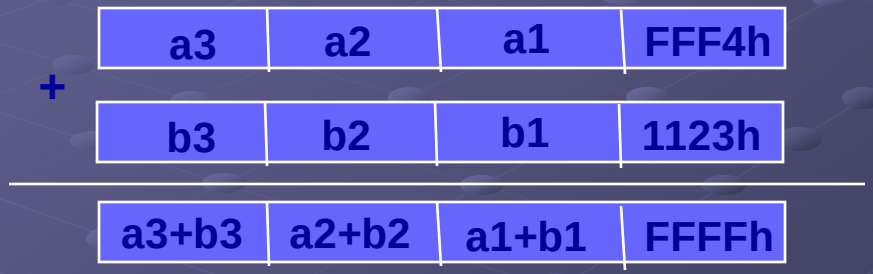
# Saturation/Wraparound Arithmetic

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



# Saturation/Wraparound Arithmetic (cont.)

- Unsigned Saturation: add with unsigned saturation (PADDUS)



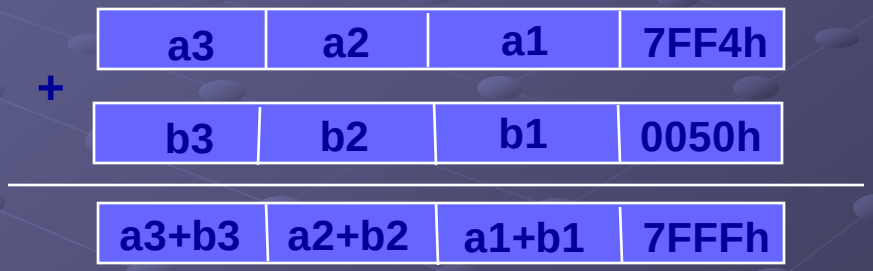

Saturation

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)



# Adding 8 8-bit Integers with Saturation

X0 dq **8080555580805555**

X1 dq **0090 33FF 0090 33FF**

```
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)

# 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 |

# 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

# 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