Instruction Set extensions to X86

- Some extensions to x86 instruction set intended to accelerate 3D graphics
- AMD *3D-Now!* Instructions simply accelerate floating point arithmetic.
  - Accelerate object transformations
  - Allow multiple floating point operations to be done in one clock cycle.
- A similar extension is found on the Pentium III – just does not have the fancy name.

Floating Point SIMD instructions

- SIMD stands for Single Instruction, Multiple Data
- Same instruction applied to multiple operands
  - Do an add on four pairs of operands
    y0= a0 +b0, y1 = a1+b1, y2=a2+b2, y3 = a3+b3
- Pentium III added some 128 bit registers used to hold 'packed' single precision floating point numbers
  - A single precision floating point number is 32 bits

xmm Registers

New 128 bit registers are called XMM registers (XMM0 – XMM7)
Holds four 32-bit single precision floating point numbers

An instruction like ADDPS xmm0, xmm1 will add the two registers together, computing the sums of the four numbers.

Easy to see speed advantage over previous instructions

	4.0 (32 bits)	4.0 (32 bits)	3.5 (32 bits)	-2.0 (32 bits)
+	-1.5 (32 bits)	2.0 (32 bits)	1.7 (32 bits)	2.3 (32 bits)
	2.5 (32 bits)	6.0 (32 bits)	5.2 (32 bits)	0.3 (32 bits)

SIMD Extensions

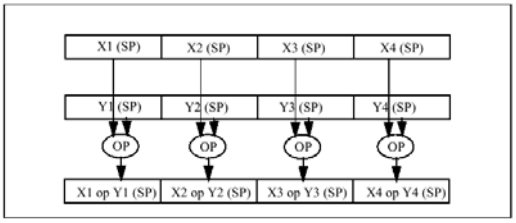


Figure 9-5. Packed Operations

More than 70 instructions. Arithmetic Operations supported: Addition, Subtraction, Mult, Division, Square Root, Maximum, Minimum. Can operate on Floating point or Integer data.

Flags

- Individual flags are not kept for each packed operation.
- Can only tell if an error (exception) occurred in one or more of the packed operations
- Some possible exceptions (not all listed)
  - Underflow (number too small)
  - Overflow (number too large)
  - Divide by Zero

Pentium 3 vs. Pentium 4

- The SIMD extensions on the Pentium 3 are called the SSE instructions and the 128 bit registers only support viewing the data as 4 single precision FP numbers.
- On the Pentium 4, the 128 bit registers can be viewed as these data types
  - 4 single precision FP values (SSE)
  - 2 double precision FP values (SSE2)
  - 16 byte values (SSE2)
  - 8 word values (SSE2)
  - 4 double word values (SSE2)
  - 1 128-bit integer value (SSE2)

MMX Instructions

Added eight 64 bit registers. The 64 bit register can be viewed as containing 8 packed bytes, 4 packed words, 2 dwords, or 1 quad.

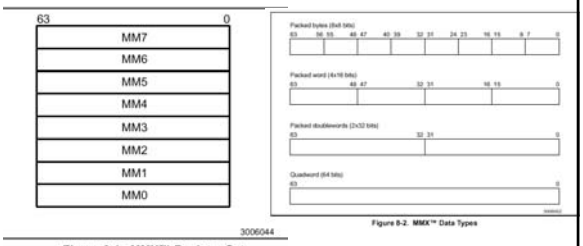


Figure 8-1. MMX™ Register Set

Figure 8-2. MMX™ Data Types

Saturating Arithmetic

The MMX instructions perform SIMD operations between MMX registers on packed bytes, words, or dwords.

The arithmetic operations can made to operate in Saturation mode.

What saturation mode does is clip numbers to Maximum positive or maximum negative values during arithmetic.

In normal mode: FFh + 01h = 00h (unsigned overflow)
In saturated, unsigned mode: FFh + 01 = FFh (saturated to maximum value, closer to actual arithmetic value)

In normal mode: 7fh + 01h = 80h (signed overflow)

In saturated, signed mode: 7fh + 01 = 7fh (saturated to max value)

Why Saturating Arithmetic?

- In case of integer overflow (either signed or unsigned), many applications are satisfied with just getting an answer that is close to the right answer or saturated to maxium result
- Many DSP (Digital Signal Processing) algorithms depend on this feature
  - Many DSP algorithms for audio data (8 to 16 bit data) and Video data (8-bit R,G,B values) are integer based, and need saturating arithmetic.
- This is easy to implement in hardware, but slow to emulate in software. A nice feature to have.

Floating Point Representations

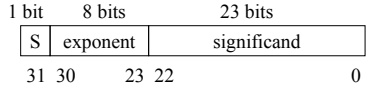
- The goal of floating point representation is represent a large range of numbers
- Floating point in decimal representation looks like:
  $+3.0 \times 10^{3}$, 4.5647×10^{-20} , etc
- In binary, sample numbers look like:
  -1.0011×2^{4} , 1.10110×2^{-3} , etc
- Our binary floating point numbers will always be of the general form:
  (*sign*) 1.*mmmmmm* x $2^{exponent}$
- The sign is positive or negative, the bits to the right of decimal point is the mantissa or *significand*, exponent can be either positive or negative. The numeral to the left of the decimal point is ALWAYS 1 (normalized notation).

Floating Point Encoding

- The number of bits allocated for exponent will determine the maximum, minimum floating point numbers (range)
  1.0×2^{-max} (small number) to
  $1.0 \times 2^{+max}$ (large number)
- The number of bits allocated for the significand will determine the precision of the floating point number
- The sign bit only needs one bit (negative:1, positive: 0)

Single Precision, IEEE 754

Single precision floating point numbers using the IEEE 754 standard require 32 bits:



Exponent encoding is *bias 127*. To get the encoding, take the exponent and add 127 to it.

If exponent is –1, then exponent field = -1 + 127 = 126 = 7Eh
If exponent is 10, then exponent field = 10 + 127 = 137 = 89h
Smallest allowed exponent is –126, largest allowed exponent is +127. This leaves the encodings 00H, FFH unused for normal numbers.

Convert Floating Point Binary Format to Decimal

1 10000001 010000.....0

S	exponent	significand

What is this number?

Sign bit = 1, so negative.

Exponent field = 81h = 129.
Actual exponent = Exponent field – 127 = 129 – 127 = 2.

Number is:

-1 . (01000...000) x 2^2
-1 . (0 x 2^{-1} + 1 x 2^{-2} + 0 x 2-3 .. +0) x 4
-1 . (0 + 0.25 + 0 +..0) x 4
-1.25 x 4 = -5.0.

Convert Decimal FP to binary encoding

What is the number -28.75 in Single Precision Floating Point?

1. Ignore the sign, convert integer and fractional part to binary representation first:
   a. 28 = 1Ch = 0001 1100
   b. .75 = .5 + .25 = $2^{-1} + 2^{-2}$ = .11

-28.75 in binary is - **00011100.11** (ignore leading zeros)

2. Now NORMALIZE the number to the format 1.mmmm x 2^{exp}
   Normalize by shifting. Each shift right add one to exponent, each shift left subtract one from exponent:

- 11100.11 x 2^0 = - 1110.011 x 2^1
= - 111.0011 x 2^2
= - 1.110011 x 2^4

Convert Decimal FP to binary encoding (cont)

Normalized number is: **- 1.110011 x 2^4**

Sign bit = 1

Significand field = 110011000...000

Exponent field = 4 + 127 = 131 = 83h = 1000 0011

Complete 32-bit number is:

1 10000011 110011000....000

S	exponent	significand

Algorithm for converting fractional decimal to Binary

An algorithm for converting any fractional decimal number to its binary representation is successive multiplication by two (results in shifting left). Determines bits from MSB to LSB.

a. Multiply fraction by 2.

b. If number >= 1.0, then current bit = 1, else current bit = 0.

c. Take fractional part of number and go to 'a'. Continue until fractional number is 0 or desired precision is reached.

Example: Convert .5625 to binary
.5625 x 2 = 1.125 (>= 1.0, so MSB bit = '1').
.125 x 2 = .25 (< 1.0 so bit = '0')
.25 x 2 = .5 (< 1.0 so bit = '0')
.5 x 2 = 1.0 (>= 1.0 bit = 1), finished.
.5625 = .1001b

Overflow/Underflow, Double Precision

- Overflow in floating point means producing a number that is too big or too small (underflow)
  - Depends on Exponent size
  - Min/Max exponents are 2^{-126} to 2^{+127} is 10^{-38} to 10^{+38}.
- To increase the range, need to increase number of bits in exponent field.
- Double precision numbers are 64 bits - 1 bit sign bit, 11 bits exponent, 52 bits for significand
- Extra bits in significand gives more precision, not extended range.

Special Numbers

Min/Max exponents are 2^{-126} to 2^{+127}.
This corresponds to exponent field values of of 1 to 254.

The exponent field values 0 and 255 are reserved for *special numbers*. *Special Numbers* are zero, +/- infinity, and NaN (not a number)

Zero is represented by ALL FIELDS = 0.

+/- Infinity is Exponent field = 255 = FFh, significand = 0. +/- Infinity is produced by anything divided by 0.

NaN (Not A Number) is Exponent field = 255 = FFh, significand = nonzero. NaN is produced by invalid operations like zero divided by zero, or infinity – infinity.

Comments on IEEE Format

- Sign bit is placed is in MSB for a reason – a quick test can be used to sort floating point numbers by sign, just test MSB
- If sign bits are the same, then extracting and comparing the exponent fields can be used to sort Floating point numbers. A larger exponent field means a larger number since the 'bias' encoding is used.
- All microprocessors that support Floating point use the IEEE 754 standard. Only a few supercomputers still use different formats.

x86 Floating Point Instructions

x86 Floating Point instructions handled by a separate execution engine – used to be a separate chip (80387) but moved onto the same die as the integer unit starting with the 80486.

st0(st)
st1(st-1)
st2
st3
st4
st5
st6
st7

Floating Point Unit (FPU) has eight 80-bit registers (ST0-ST7) arranged as a stack.

80-bits used for extra precision over IEEE format.

Top of the stack normally referred to as ST, next register as ST-1

Classical Stack Instructions

A classical stack instruction has 0 operands specified in the instruction.

The two operands are assumed to be ST and ST(1). The result is temporarily stored in ST(1), then ST is popped off the stack leaving the result on top of the stack.

FADD ; ST(1) = ST(1) + ST; pop ST(1) into ST

Before FADD

ST	**100.0**
ST(1)	**20.0**

After FADD

ST	**120.0**
ST(1)	

Note: FSUB does ST(1) = ST(1)-ST; pop ST(1) into ST

Real Memory and Integer Memory

Real Memory and Integer Memory instructions have an implied first operand that is ST and a memory operand for the second operand. The result is placed into ST.

```
            .data
floatval    dd    10.0  ;floating point value
int16val    dw    10    ; 16 bit integer
int32val    dd    10    ; 32 bit integer
                        ; no decimal point!
            .code
….
            fadd  floatval    ;st=st+10.0
            fiadd int16val    ;st = st+10
            fiadd int32val    ;st = st+10
```

Integer values converted to floating point format before operation is performed.

Register Operands

Register instructions use the FPU registers as ordinary operands.

```
fadd st,st(1)  ; ST = ST + ST(1)
fsub st,st(1)  ; ST = ST – ST(1)
```

Register pop is same as register, except that ST is popped off stack afterwards.

```
faddp  st(1),st; ST(1) = ST(1)+ST, pop ST
```

Before FADDP		Intermediate		After FADDP	
ST	100.0	ST	100.0	ST	120.0
ST(1)	20.0	ST(1)	120.0	ST(1)	

fld, fild

The *fld* instruction is used to load a memory floating point operand into ST0. The *fild* instruction loads a memory integer operand into ST0.

```
            .data
  op1       dd    6.0  ;floating point value
  op2       dw    2    ; integer value
            .code
  ….
             fld   op1
             fild  op2
```

Before 1st fld		after ‘fld op1’		after ‘fld op2’	
ST	??	ST	6.0	ST	2.0
ST(1)	??	ST(1)	??	ST(1)	6.0

fst, fstp

The *fst* (Float store) is used to save ST0 to memory.
The *fstp* (Float store and pop) is used to save ST0 to memory, then pop the stack.

```
            .data
  result    dd    ??  ;floating point value
            .code
  ….
             fst  result   ;; save ST0 to result
             fstp result   ;; save ST0 to result
                           ;; and pop
```

Before *fstp*

ST	100.0
ST(1)	20.0
result	??

After *fstp*

ST	20.0
ST(1)	??
result	100.0

A Sample Program

```
      .model small
      .586
      .stack 100h
      .data
op1   dd    6.0
op2   dd    2.0
op3   dd    5.0
result  dd  ?
      .code
main  proc
      mov ax,@data
      mov   ds,ax
      finit
      fld  op1
      fld  op2
      fmul
      fld   op3
```

Init FPU

ST	6.0

ST0	2.0
ST(1)	6.0

ST0	12.0
ST(1)	??

ST0	5.0
ST(1)	12.0

A Sample Program (cont)

```
main  proc
      mov ax,@data
      mov   ds,ax
      finit
      fld  op1
      fld  op2
      fmul
      fld   op3
      fsub
      fwait
      fstp result
      mov   ax,4c00h
      int   21h
main  endp
end   main
```

ST0	5.0
ST(1)	12.0

ST(1) = ST(1) – ST; pop

ST0	7.0
ST(1)	??

Wait for exceptions to finish

result	7.0

ST0	??
ST(1)	??

Other Instructions

```
fmul     ;    st(1) = st(1)* st(0), pop
fdiv     ;    st(1) = st(1)/ st(0), pop
fdivr    ;    st(1) = st(0)/ st(1), pop
fsqrt    ;    st(0) = square root(st(0))
fsin     ;    st(0) = sine(st(0));
fcos     ;    st(0) = fcos(st(0));
```