Floating Point Instructions

Ray Seyfarth

June 29, 2012
Floating point instructions

- PC floating point operations were once done in a separate chip - 8087
- This chip managed a stack of eight 80 bit floating point values
- The stack and instructions still exist, but are largely ignored
- x86-64 CPUs have 16 floating point registers (128 or 256 bits)
- These registers can be used for single data instructions or single instruction multiple data instructions (SIMD)
- We will focus on these newer registers
- The older instructions tended to start with the letter “f” and referenced the stack using register names like ST0
- The newer instructions reference using registers with names like “XMM0”
Outline

1. Moving data in and out of floating point registers
2. Addition
3. Subtraction
4. Basic floating point instructions
5. Data conversion
6. Floating point comparisons
7. Mathematical functions
8. Sample floating point code
moving scalars to or from floating point registers

- **movss** moves a single 32 bit floating point value to or from an XMM register
- **movsd** moves a single 64 bit floating point value
- There is no implicit data conversion - unlike the old instructions which converted floating point data to an 80 bit internal format
- The instructions follow the standard pattern of having possibly one memory address

```assembly
movss xmm0, [x] ; move value at x into xmm0
movsd [y], xmm1 ; move value from xmm1 to y
movss xmm2, xmm0 ; move from xmm0 to xmm2
```
Moving packed data

- The XMM registers are 128 bits
- They can hold 4 floats or 2 doubles (or integers of various sizes)
- On newer CPUs they are extended to 256 bits and referred to as YMM registers when using all 256 bits
- movaps moves 4 floats to/from a memory address aligned at a 16 byte boundary
- movups does the same task with unaligned memory addresses
- The Core i series performs unaligned moves efficiently
- movapd moves 2 doubles to/from a memory address aligned at a 16 byte boundary
- movupd does the same task with unaligned memory addresses

movups xmm0, [x] ; move 4 floats to xmm0
movupd [a], xmm15 ; move 2 doubles to a
Floating point addition

- `addss` adds a scalar float (single precision) to another
- `addsd` adds a scalar double to another
- `addps` adds 4 floats to 4 floats - pairwise addition
- `addpd` adds 2 doubles to 2 doubles
- There are 2 operands: destination and source
- The source can be memory or an XMM register
- The destination must be an XMM register
- Flags are unaffected

```assembly
movss   xmm0, [a] ; load a
addss   xmm0, [b] ; add b to a
movss   [c], xmm0 ; store sum in c
movapd  xmm0, [a] ; load 2 doubles from a
addpd   xmm0, [b] ; add a[0]+b[0] and a[1]+b[1]
movapd  [c], xmm0 ; store 2 sums in c
```
Floating point subtraction

- `subss` subtracts the source float from the destination
- `subsd` subtracts the source double from the destination
- `subps` subtracts 4 floats from 4 floats
- `subpd` subtracts 2 doubles from 2 doubles

```
movss    xmm0, [a] ; load a
subss    xmm0, [b] ; add b from a
movss    [c], xmm0 ; store a-b in c
movapd   xmm0, [a] ; load 2 doubles from a
subpd    xmm0, [b] ; add a[0]-b[0] and a[1]-b[1]
movapd   [c], xmm0 ; store 2 differences in c
```
## Basic floating point instructions

<table>
<thead>
<tr>
<th>instruction</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>addsd</td>
<td>add scalar double</td>
</tr>
<tr>
<td>addss</td>
<td>add scalar float</td>
</tr>
<tr>
<td>addpd</td>
<td>add packed double</td>
</tr>
<tr>
<td>addps</td>
<td>add packed float</td>
</tr>
<tr>
<td>subsd</td>
<td>subtract scalar double</td>
</tr>
<tr>
<td>subss</td>
<td>subtract scalar float</td>
</tr>
<tr>
<td>subpd</td>
<td>subtract packed double</td>
</tr>
<tr>
<td>subps</td>
<td>subtract packed float</td>
</tr>
<tr>
<td>mulsd</td>
<td>multiply scalar double</td>
</tr>
<tr>
<td>mulss</td>
<td>multiply scalar float</td>
</tr>
<tr>
<td>mulpd</td>
<td>multiply packed double</td>
</tr>
<tr>
<td>mulps</td>
<td>multiply packed float</td>
</tr>
<tr>
<td>divsd</td>
<td>divide scalar double</td>
</tr>
<tr>
<td>divss</td>
<td>divide scalar float</td>
</tr>
<tr>
<td>divpd</td>
<td>divide packed double</td>
</tr>
<tr>
<td>divps</td>
<td>divide packed float</td>
</tr>
</tbody>
</table>
Conversion to a different length floating point

- `cvtss2sd` converts a scalar single (float) to a scalar double
- `cvtps2pd` converts 2 packed floats to 2 packed doubles
- `cvtsd2ss` converts a scalar double to a scalar float
- `cvtpd2ps` converts 2 packed doubles to 2 packed floats

```
cvtss2sd    xmm0, [a] ; get a into xmm0 as a double
addsd      xmm0, [b] ; add a double to a
 cvtsd2ss   xmm0, xmm0 ; convert to float
movss       [c], xmm0
```
Converting floating point to/from integer

- `cvtss2si` converts a float to a double word or quad word integer
- `cvtsd2si` converts a float to a double word or quad word integer
- These 2 round the value
- `cvttss2si` and `cvttsd2si` convert by truncation
- `cvtsi2ss` converts an integer to a float in an XMM register
- `cvtsi2sd` converts an integer to a double in an XMM register
- When converting from memory a size qualifier is needed

```assembly
  cvtss2si    eax, xmm0 ; convert to dword integer
  cvtsi2sd    xmm0, rax ; convert qword to double
  cvtsi2sd    xmm0, dword [x] ; convert dword integer
```
Unordered versus ordered comparisons

- Floating point comparisons can cause exceptions
- Ordered comparisons cause exceptions one QNaN or SNaN
  - QNaN means “quiet not a number”
  - SNaN means “signalling not a number”
  - Both have all exponent field bits set to 1
  - QNaN has its top fraction bit equal to 1
- An unordered comparison causes exceptions only for SNaN
- gcc uses unordered comparisons
- If it’s good enough for gcc, it’s good enough for me
- ucomiss compares floats
- ucomisd compares doubles
- The first operand must be an XMM register
- They set the zero flag, parity flag and carry flags

```
movss  xmm0, [a]
mulss  xmm0, [b]
ucomiss xmm0, [c]
jmple  less_eq  ; jmp if a*b <= c
```
Mathematical functions

- 8087 had sine, cosine, arctangent and more
- The newer instructions omit these operations on XMM registers
- Instead you are supposed to use efficient library functions
- There are instructions for
  - Minimum
  - Maximum
  - Rounding
  - Square root
  - Reciprocal of square root
Minimum and maximum

- **minss** and **maxss** compute minimum or maximum of scalar floats
- **minsd** and **maxsd** compute minimum or maximum of scalar doubles
- The destination operand must be an XMM register
- The source can be an XMM register or memory
- **minps** and **maxps** compute minimum or maximum of packed floats
- **minpd** and **maxpd** compute minimum or maximum of packed doubles
- **minps xmm0, xmm1** computes 4 minimums and places them in xmm0

```assembly
movss xmm0, [x] ; move x into xmm0
maxss xmm0, [y] ; xmm0 has max(x,y)
movapd xmm0, [a] ; move a[0] and a[1] into xmm0
minpd xmm0, [b] ; xmm0[0] has min(a[0],b[0])
               ; xmm0[1] has min(a[1],b[1])
```
Rounding

- `roundss` rounds 1 float
- `roundps` rounds 4 floats
- `roundsd` rounds 1 double
- `roundpd` rounds 2 doubles

The first operand is an XMM destination register.
The second is the source in an XMM register or memory.
The third operand is a rounding mode.

<table>
<thead>
<tr>
<th>mode</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>round, giving ties to even numbers</td>
</tr>
<tr>
<td>1</td>
<td>round down</td>
</tr>
<tr>
<td>2</td>
<td>round up</td>
</tr>
<tr>
<td>3</td>
<td>round toward 0 (truncate)</td>
</tr>
</tbody>
</table>
Square roots

- `sqrtss` computes 1 float square root
- `sqrtps` computes 4 float square roots
- `sqrtsd` computes 1 double square root
- `sqrtpd` computes 2 double square roots
- The first operand is an XMM destination register
- The second is the source in an XMM register or memory
Distance in 3D

\[ d = \sqrt{((x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2)} \]

distance3d:

```
movss  xmm0, [rdi]    ; x from first point
subss  xmm0, [rsi]    ; subtract x from second point
mulss  xmm0, xmm0     ; (x1-x2)^2
movss  xmm1, [rdi+4]  ; y from first point
subss  xmm1, [rsi+4]  ; subtract y from second point
mulss  xmm1, xmm1     ; (y1-y2)^2
movss  xmm2, [rdi+8]  ; z from first point
subss  xmm2, [rsi+8]  ; subtract z from second point
mulss  xmm2, xmm2     ; (z1-z2)^2
addss  xmm0, xmm1     ; add x and y parts
addss  xmm0, xmm2     ; add z part
sqrt   xmm0, xmm0     ;
ret                 
```
Dot product in 3D

\[ d = x_1 x_2 + y_1 y_2 + z_1 z_2 \]

dot_product:

\[
\begin{align*}
\text{movss} & \quad \text{xmm0}, \ [\text{rdi}] \\
\text{mulss} & \quad \text{xmm0}, \ [\text{rsi}] \\
\text{movss} & \quad \text{xmm1}, \ [\text{rdi+4}] \\
\text{mulss} & \quad \text{xmm1}, \ [\text{rsi+4}] \\
\text{addss} & \quad \text{xmm0}, \ \text{xmm1} \\
\text{movss} & \quad \text{xmm2}, \ [\text{rdi+8}] \\
\text{mulss} & \quad \text{xmm2}, \ [\text{rsi+8}] \\
\text{addss} & \quad \text{xmm0}, \ \text{xmm2} \\
\text{ret} & 
\end{align*}
\]
Polynomial evaluation by Horner’s Rule

\[ P(x) = p_0 + p_1x + p_2x^2 \cdots p_nx^n \]

\[ b_n = p_n \]
\[ b_{n-1} = p_{n-1} + b_nx \]
\[ b_{n-2} = p_{n-2} + b_{n-1}x \]
\[ b_0 = p_0 + b_1x \]

```assembly
horner: movsd xmm1, xmm0 ; use xmm1 as x
       movsd xmm0, [rdi+rsi*8] ; accumulator for b_k
       test esi, 0 ; is the degree 0?
       jz done
more:  sub esi, 1
       mulsd xmm0, xmm1 ; b_k * x
       addsd xmm0, [rdi+rsi*8] ; add p_k
       jnz more
done:  ret
```