Floating Point Instructions

Floating point instructions

- PC floating point operations were once done in a separate chip - 8087
  - This chip managed a stack of 80 bit floating point values.
  - The stack and instructions still exist, but are largely ignored.
  - In the absence of an 8087 chip, floating point operation where emulated in software.

Floating point instructions

- PC floating point operations were once done in a separate chip - 8087
  - This chip managed a stack of 80 bit floating point values.
  - The stack and instructions still exist, but are largely ignored.
  - In the absence of an 8087 chip, floating point operation where emulated in software.
- 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)

Floating point instructions

- PC floating point operations were once done in a separate chip - 8087
  - This chip managed a stack of 80 bit floating point values.
  - The stack and instructions still exist, but are largely ignored.
  - In the absence of an 8087 chip, floating point operation where emulated in software.
- 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`", and "`ymm0`". (`zmm0 in new CPUs as well`)

Floating point instructions

- There are 16 floating point registers.
- `ymm0` to `ymm15` (AVX registers)
- Each one is 256 bits.
- The lower half (128 bits) of `ymm-` is refereed to at `xmm-`
- `xmm0` to `xmm15` (SSE registers)
- The full 256 bit register are available from the Core i series.

Floating point instructions

- There are 16 floating point registers.
- `ymm0` to `ymm15` (AVX registers)
- Each one is 256 bits.
- The lower half (128 bits) of `ymm-` is refereed to at `xmm-`
- `xmm0` to `xmm15` (SSE registers)
- The full 256 bit register are available from the Core i series.
- We will mainly discuss `xmm` registers, all operations are the same for `ymm` registers, we just append a `v` in front of the instruction.

Moving scalars to or from floating point registers

Moving floating point numbers

- The two instructions available are `movss` and `movsd`
- `movss` moves a single 32 bit floating point value to or from an `xmm` register (float/single)
- `movsd` moves a single 64 bit floating point value (double)

Moving scalars to or from floating point registers

Moving floating point numbers

- The two instructions available are `movss` and `movsd`
- `movss` moves a single 32 bit floating point value to or from an `xmm` register (float/single)
- `movsd` moves a single 64 bit floating point value (double)
- It should be noted that there is no implicit data conversion - unlike the old instructions which converted floating point data to an 80 bit internal format

Moving scalars to or from floating point registers

- The instructions follow the standard pattern of having at most one memory address

```
    segment .data
x:    dd 12.35 ; float/single
y:    dq 14.36 ; double

    movss   xmm0, [x]    ; move the float value at x into xmm0
    movsd   [y], xmm1    ; move double 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)
- The `YMM` registers are 256 bits
  - They can hold 8 floats or 4 doubles (or integers of various sizes)

Moving packed data

- The `XMM` registers are 128 bits
  - They can hold 4 floats or 2 doubles (or integers of various sizes)
- The `YMM` registers are 256 bits
  - They can hold 8 floats or 4 doubles (or integers of various sizes)
- But how do we load them?
  - There are two types of packed move instructions available.
  - An **aligned** version and an **unaligned** version.
  - **aligned** move requires the data to be of a 16 byte boundary, but is faster in general.
  - **unaligned** move is slower, though more so on older CPUs.

Moving packed data

- The `XMM` registers are 128 bits
  - They can hold 4 floats or 2 doubles (or integers of various sizes)
- The `YMM` registers are 256 bits
  - They can hold 8 floats or 4 doubles (or integers of various sizes)
- But how do we load them?
  - There are two types of packed move instructions available.
  - An **aligned** version and an **unaligned** version.
  - **aligned** move requires the data to be of a 16 byte boundary, but is faster in general.
  - **unaligned** move is slower, though more so on older CPUs.
- If you try an use a aligned move on unaligned data you will get a segmentation fault.

Moving packed data

Actual packed move instructions:

- `movaps` moves 4 floats to/from a memory address aligned at a 16 byte boundary
- `movups` does the same task with unaligned memory addresses
- `movapd` moves 2 doubles to/from a memory address aligned at a 16 byte boundary
- `movupd` does the same task with unaligned memory addresses

```
   segment .data
x: dd 12.3, 9.3, 123.2, 0.1
a: dq 0, 0
....
   movups  xmm0, [x]    ; move 4 floats to xmm0
   movupd  [a], xmm15   ; move 2 doubles to a
```

Moving packed data

If you wish to use the aligned move:

```
   segment .data
align 16
x: dd 12.3, 9.3, 123.2, 0.1
....
   movaps  xmm0, [x]    ; move 4 floats to xmm0
```

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

```
movss   xmm0, [a]   ; load a
addss   xmm0, [b]   ; add b to a
movss   [c], xmm0   ; store sum in c
```

And

```
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
```

And

```
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

instruction	effect
addsd	add scalar double
addss	add scalar float
addpd	add packed double
addps	add packed float
subsd	subtract scalar double
subss	subtract scalar float
subpd	subtract packed double
subps	subtract packed float
mulsd	multiply scalar double
mulss	multiply scalar float
mulpd	multiply packed double
mulps	multiply packed float
divsd	divide scalar double
divss	divide scalar float
divpd	divide packed double
divps	divide packed float

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 by rounding
- cvtsd2si converts a float to a double word or quad word integer by rounding
- 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 integers from memory a size qualifier is needed

```
cvtss2si    eax, xmm0    ; convert to dword integer
cvtss2si    rax, xmm0    ; convert to qword integer
cvtsi2sd    xmm0, rax    ; convert qword to double
cvtsi2sd    xmm0, dword [x] ; convert dword integer
```

Unordered versus ordered comparisons

- In the IEEE-754 floating point standard there are two types of NaNs (not a number)
- 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

Unordered versus ordered comparisons

- Floating point comparisons can cause exceptions
- Ordered comparisons cause exceptions on QNaN or SNaN
- An unordered comparison causes exceptions only for SNaN
- `gcc` uses unordered comparisons
- If it's good enough for `gcc`, it's good enough for us
- `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]
jbe  less_eq   ; jmp if a*b <= c
```

Conditional floating point jumps

instruction	meaning	aliases	flags
jb	jump if below	jc jnae	CF=1
jbe	jump if below or equal	jna	ZF=1 or CF=1
ja	jump if above	jnbe	ZF=0 or CF=0
jae	jump if above or equal	jnc jnb	CF=0
je	jump if equal	jz	ZF=1
jne	jump if not equal	jnz	ZF=0

c= carry flag set

z= zero flag set

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`

```
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

mode	meaning
0	round, giving ties to even numbers
1	round down
2	round up
3	round toward 0 (truncate)

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
    sqrtss  xmm0, xmm0
    ret
```

Dot product in 3D

$$d = x_1 x_2 + y_1 y_2 + z_1 z_2$$

```
dot_product:
        movss   xmm0, [rdi]
        mulss   xmm0, [rsi]
        movss   xmm1, [rdi+4]
        mulss   xmm1, [rsi+4]
        addss   xmm0, xmm1
        movss   xmm2, [rdi+8]
        mulss   xmm2, [rsi+8]
        addss   xmm0, xmm2
        ret
```