Floating Point Operations and Streaming SIMD Extensions

Advanced Topics Spring 2009

Prof. Robert van Engelen

Overview

- IEEE 754 Floating point
- IEEE 754 Exceptions
- FPU control and status registers
- Language and compiler issues with IEEE floating point
- FP tricks
- FP error analysis
- SIMD short vector extensions
- Programming with SSE
- GNU multi-precision library (GMP)
- GPU programming (**next topic**)

Floating Point

- Definitions
  - Notation: $s\ d.dd\ldots d \times r^e$
  - **Sign**: s (+ or -)
  - **Significand**: $d.d\ldots d$ with p digits (precision p)
  - **Radix**: r (typically 2 or 10)
  - Signed **exponent**: e where $e_{\text{min}} \leq e \leq e_{\text{max}}$

- Represents a floating point value (really a *rational* value!)

$$\pm\,(d_0 + d_1 r^{-1} + d_2 r^{-2} + \ldots + d_{p-1} r^{-(p-1)})\ r^{e}$$

where $0 \leq d_i < r$

IEEE 754 Floating Point

- The IEEE 754 standard specifies
  - Binary floating point format ($r = 2$)
  - Single, double, extended, and double extended precision
  - Representations for indefinite values (NaN) and infinity (INF)
  - Signed zero and denormalized numbers
  - Masked exceptions
  - Roundoff control
  - Standardized algorithms for arithmetic to ensure accuracy and bit-precise portability

IEEE 754 Floating Point

- *Standardized algorithms for arithmetic to ensure accuracy and bit-precise portability*

- But programs that rely on IEEE 754 **may still not** be bit-precise portable, because many math function libraries are not identical across systems

- Unless you write your own libraries

IEEE 754 Floating Point versus Binary Coded Decimal (BCD)

- Binary floating point (radix *r* = 2) with limited precision *p* cannot represent decimal values accurately
  - 0.10000 ≈ $2^{-4} + 2^{-5} + 2^{-8} + \ldots$
  - `for (float x = 0.0; x < 1.0; x += 0.01) { … }`
    will not work correctly! (`x = 0.999999 < 1.0`)
  - `DO X = 0.0, 1.0, 0.01`
    will work: Fortran determines number of iter's from loop bounds
  - Use `if (fabs(x-y) < 0.0001)` instead of `if (x == y)`
- Packed binary coded decimal (BCD) encodes decimal digits in groups of 4 bits (nibbles): 0000 (0) … 1001 (9)
  - 351.20 = 0011 0101 0001 . 0010 0000
  - Used by calculators, some spreadsheet programs (not Excel!), and many business/financial data processing systems, COBOL

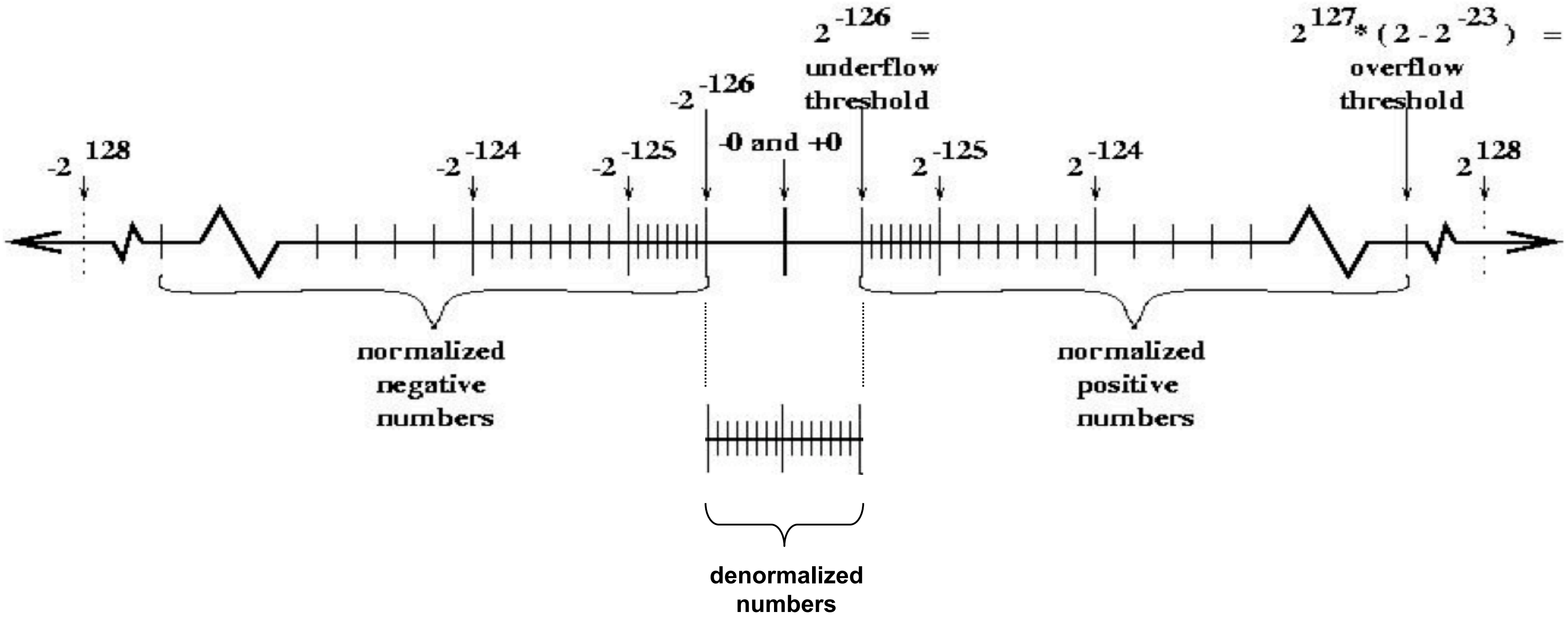
IEEE 754 Floating Point Formats

- Four formats:

Parameter	Format			
	Single	Single-Extended	Double	Double-Extended
p	24	≥32	53	≥64
e_{max}	+127	≥+1023	+1023	≥16383
e_{min}	-126	≤-1022	-1022	≤16382
Exponent width	8 bits	≥11 bits	11 bits	≥15 bits
Format width	32 bits	≥43 bits	64 bits	≥79 bits

IEEE 754 Floating Point

- Most significant bit of the significand d_0 not stored
- **Normalized numbers**: ±1.$dd\ldots d$ 2^e
- **Denormalized numbers**: ±0.$dd\ldots d$ $2^{emin\text{-}1}$



2^{-126} = underflow threshold
$2^{127} * (2 - 2^{-23})$ = overflow threshold
-2^{-126}
-2^{128}
-2^{-124}
-2^{-125}
-0 and +0
2^{-125}
2^{-124}
2^{128}
normalized negative numbers
normalized positive numbers
denormalized numbers

IEEE 754 Floating Point Overflow and Underflow

- Arithmetic operations can **overflow** or **underflow**
- Overflow: result value requires $e > e_{\text{max}}$
  - □ Raise exception or return ±infinity
  - □ Infinity (INF) represented by zero significand and $e = e_{\text{max}}+1$
    - 1/0.0 gives INF, -1/0.0 gives –INF, 3/INF gives 0
- Underflow: result value requires $e < e_{\text{min}}$
  - □ Raise exception or return denorm or return **signed** zero
  - □ Denorm represented by with $e = e_{\text{min}}-1$
- Why bother returning a denorm? Consider:

```
        if (a != b) then x = a/(a-b);
```

- Why bother distinguishing +0 from -0? Consider:

```
        if (a > b) then x = log(a-b);
```

IEEE 754 Floating Point NaN

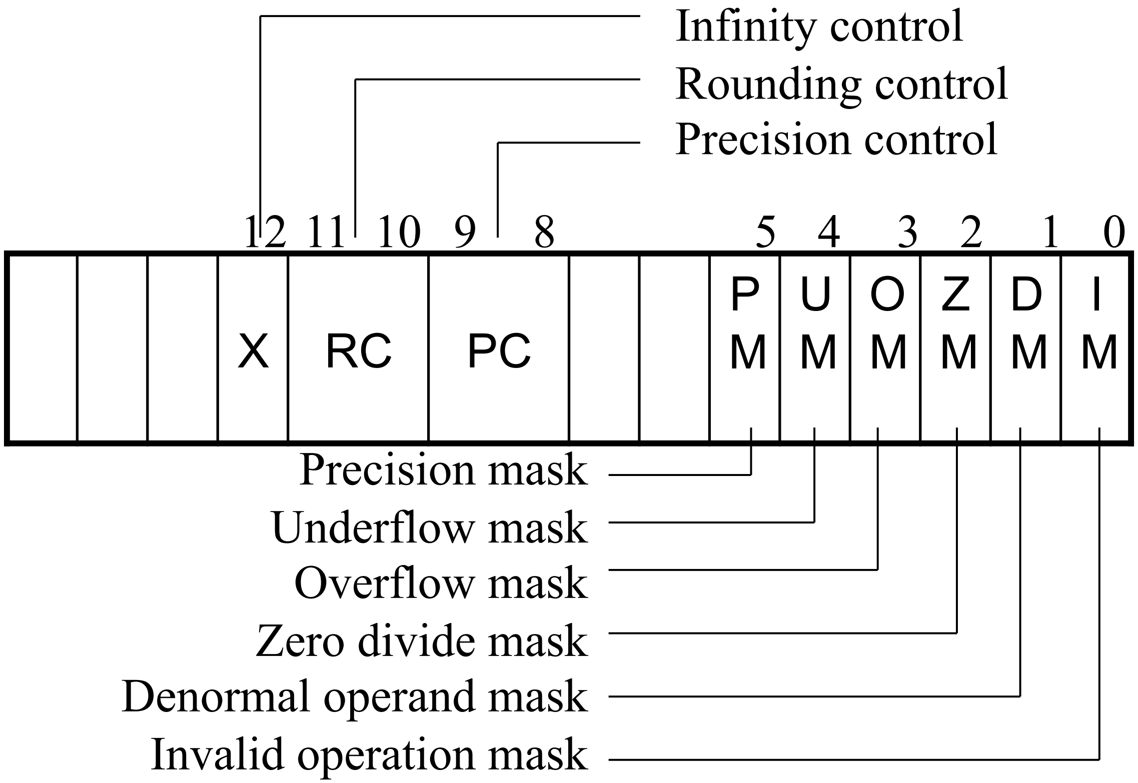
- **Not-a-number** (NaN) represented by all 1 bits in exponent $e = e_{max}+1$ (e is **biased** by $+2^{exp\_width-1}-1$)
- Sign and significand>0 are irrelevant (but may carry info)
- Generated by indeterminate and other operations
  - 0/0
  - sqrt(-1)
  - INF-INF, INF/INF, 0*INF
- Two kinds of NaN
  - **Quiet**: propagates NaN through operations without raising exception
  - **Signaling**: raise an exception when touched
- Fortran initializes reals to NaN by default
  - Signaling NaN automatically detects uninitialized data

IEEE 754 Floating Point Exceptions

- Exceptions
  - **Invalid operation**: raised by a signaling NaN or illegal operation on infinity
  - **Divide by zero**
  - **Denormal operand**: indicates loss of precision
  - **Numeric overflow** or **underflow**
  - **Inexact result or precision**: result of operation cannot be accurately represented, e.g. 3.5 x 4.3 = 15.0 for *r*=10 and *p*=3
- Exceptions can be **masked** using hardware control registers of an FPU
  - Masking means that quiet NaN and INF are returned and propagated

Intel x87 FPU FPCW

- Masking exceptions on the Intel x87 FPU using the FPCW control word



```
uint16_t setmask = …;
uint16_t oldctrl, newctrl;
_asm {
  FSTCW oldctrl
  mov ax, oldctrl
  and ax, 0ffc0h
  or ax, setmask
  mov newctrl,ax
  FLDCW newctrl
}
```

Intel x87 FPU FPCW

- The Intel x87 FPU uses a pre-specified precision for all *internal* floating point operations
  - □ Extended double (80 bits) for Linux
  - □ Double (64 bits) for Windows
- Using **`float`** and **`double`** in C only affects storage, not the internal arithmetic precision
  - □ Changing the FPU precision can speed up div, rem, and sqrt

```
uint16_t prec = 0x0000; // 0x0000=sgl, 0x0200=dbl, 0x0300=ext
uint16_t oldctrl, newctrl;
_asm {
  FSTCW oldctrl
  mov ax, oldctrl
  and ax, 0fcffh
  or ax, prec
  mov newctrl,ax
  FLDCW newctrl
}
```

Language and Compiler Issues with IEEE Floating Point

- Associative rule does not hold: $(x + y) + z \neq x + (y + z)$
  - Take $x = 10^{30}$, $y = -10^{30}$, and $z = 1$ then result is 1 or 0, respectively
- Cannot replace division by multiplication: $x/10.0 \neq 0.1*x$
  - 0.1 is not accurately represented
  - But $x/2.0 == 0.5*x$ is okay
- Distributive rule does no hold: $x*y + x*z \neq x*(y + z)$
  - Take for example $y \approx -z$
- Negation is not subtraction, since zero is signed: $-x \neq 0-x$
  - Take $x = 0$, then $-x == -0$ and $0-x == +0$
  - Note: FP hardware returns true when comparing -0 == +0
- IEEE rounding modes may differ from language's rounding

Language and Compiler Issues with IEEE Floating Point

- NaN is unordered, which affects comparisons
  - Any comparison to NaN returns false, thus when x < NaN fails this does not imply x >= NaN
  - Cannot sort array of floats that includes NaNs
  - !(x < y) is not identical to x >= y
  - x == x is not true when x = NaN
- Preserving the evaluation of comparisons matters, similar to preserving parenthesis

```
eps = 1;
do eps = 0.5*eps;
while (eps + 1 > 1);
```

Correct

`(eps + 1)` = 1
when eps is small

```
eps = 1;
do eps = 0.5*eps;
while (eps > 0);
```

Incorrect

Language and Compiler Issues with IEEE Floating Point (cont)

- Exceptions (e.g. signaling NaN) disallow expression optimization
  - These two instructions have no dependence and can potentially be reordered:

    ```
    x = y*z;
    a = b+c;
    ```

    but each may trigger an exception and the reorder destroys relationship (what if b+c triggers exception and exception handler wants to read x?)
- A change in rounding mode affects common sub-expressions
  - The expression a*b is not common in this code:

    ```
    x = a*b;
    set_round_mode = UP;
    y = a*b;
    ```

Language and Compiler Issues with IEEE Floating Point (cont)

- Programming languages differ in narrowing and widening type conversions
  - □ Use the type of the destination of the assignment to evaluate operands
    `float x = n/m;` // causes n and m to be widened to float first
  - □ Obey type of operands, widen intermediate values when necessary, and then narrow final value to destination type
    - More common, e.g. C, Java
- IEEE ensures the following are valid for all values of x and y:
  - □ x+y = y+x
  - □ x+x = 2*x
  - □ 1.0*x = x
  - □ 0.5*x = x/2.0

IEEE 754 Floating Point Manipulation Tricks

- Fast FP-to-integer conversion (rounds towards -∞)

```
#define FLOAT_FTOI_MAGIC_NUM (float)(3<<21)
#define IT_FTOI_MAGIC_NUM (0x4ac00000)
inline int FastFloatToInt(float f)
{
   f += FLOAT_FTOI_MAGIC_NUM;
   return (*((int*)&f) - IT_FTOI_MAGIC_NUM)>>1;
}
```

IEEE 754 Floating Point Manipulation Tricks

- Fast square root approximation with only <5% error

```
inline float FastSqrt(float x)
{
   int t = *(int*)&x;
   t -= 0x3f800000;
   t >>= 1;
   t += 0x3f800000;
   return *(float*)&t;
}
```

IEEE 754 Floating Point Manipulation Tricks

- Fast reciprocal square root approximation for x > 0.25 with only <0.6% error

```
inline float FastInvSqrt(float x)
{
   int tmp = ((0x3f800000 << 1) +
               0x3f800000 - *(long*)&x) >> 1;
   float y = *(float*)&tmp;
   return y * (1.47f - 0.47f * x * y * y);
}
```

Floating Point Error Analysis

- Error analysis formula
  - *fl*(a op b) = (a op b)*(1 + ε)
  - op is +, -, *, /
  - | ε | $\leq$ machine eps = $2^{\#\text{significant bits}}$ = relative error in each op
  - Assumes no overflow, underflow, or divide by zero occurs
  - Really a worst-case upper bound, no error cancellation
- Example
  - *fl*(x + y + z)
    = *fl*(*fl*(x + y) + z)
    = $((x + y)*(1+\varepsilon) + z)*(1+\varepsilon)$
    = $x + 2\varepsilon x + \varepsilon^2 x + y + 2\varepsilon y + \varepsilon^2 y + z + \varepsilon z$
    $\approx x*(1+2\varepsilon) + y*(1+2\varepsilon) + z*(1+\varepsilon)$
- Series of *n* operations: *result**(1+$n\varepsilon$)

Numerical Stability

- **Numerical stability** is an algorithm design goal
- **Backward error analysis** is applied to determine if algorithm gives the exact result for slightly changed input values

- Extensive literature, not further discussed here…

Conditioning

- An algorithm is **well conditioned** (or **insensitive**) if relative change in input causes commensurate relative change in result

  $Cond$ = | relative change in solution | / | relative change in input |

  $= | (f(x+h) - f(x)) / f(x) | / | h/x |$

  if the derivative f' of f is known:

  $Cond = | x | \, | f'(x) | / | f(x) |$

- Problem is **sensitive** or **ill-conditioned** if $Cond \gg 1$

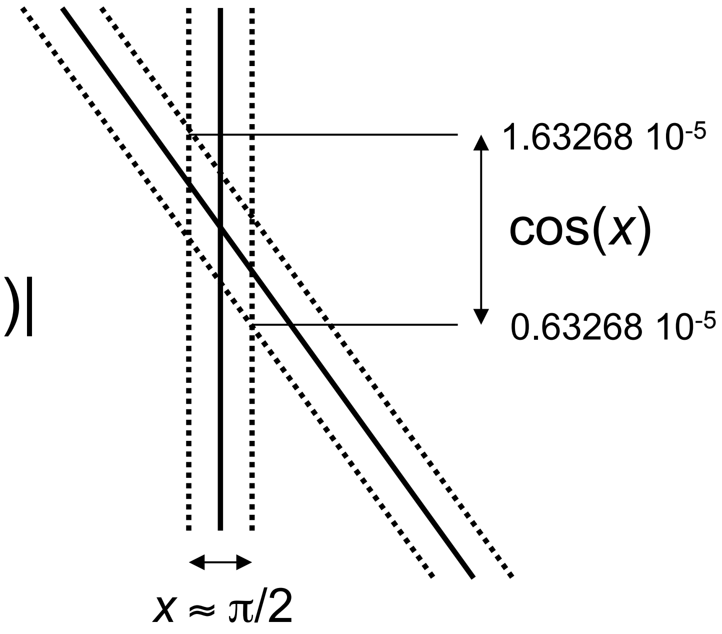
- Other definitions
  - □ **Absolute error** $= f(x+h) - f(x) \approx h\, f'(x)$
  - □ **Relative error** $= (f(x+h) - f(x)) / f(x) \approx h\, f'(x) / f(x)$

Conditioning Examples

f	x	$f(x)$	$f'(x)$	cond	$\log_{10}$(cond)
exp	1	e	e	1	0
exp	0	1	1	0	-∞
exp	-1	1/e	1/e	1	0
log	e	1	1/e	1	0
log	1	0	1	∞	∞
log	1/e	-1	e	1	0
sin	π	0	-1	∞	∞
sin	π/2	1	0	0	-∞
sin	0	0	1	NaN	NaN

Example

- Let x = π/2 and let h be a small perturbation to x
  - □ Absolute error = $\cos(x+h) - \cos(x) \approx -h \sin(x) \approx -h$
  - □ Relative error = $(\cos(x+h) - \cos(x)) / \cos(x) \approx -h \tan(x) \approx -\infty$
- Small change in x near π/2 causes relative large change in cos(x)
  - □ $\cos(1.57078) = 1.63268\ 10^{-5}$
  - □ $\cos(1.57079) = 0.63268\ 10^{-5}$
- Cond = $|\pi/2|*|\sin(\pi/2)|/|\cos(\pi/2)|$
  $= \pi/2 * 1/0 = \infty$



SIMD Short Vector Extensions

- Using **SIMD short vector extensions** can result in large performance gains
  - Instruction set extensions execute fast
  - New wide registers to hold short vectors of ints, floats, doubles
  - Parallel operations on short vectors
  - Typical vector length is 128 bit
    - Vector of 4 floats, 2 doubles, or 1 to 16 ints (128 bit to 8 bit ints)

- Technologies:
  - MMX and SSE (Intel)
  - 3DNow! (AMD)
  - AltiVec (PowerPC)
  - PA-RISC MAX (HP)

SSE SIMD Technology History

Technology	First appeared	Description
MMX	Pentium with MMX	Introduced 8-byte packed integers
SSE	Pentium III	Added 16-byte packed single precision floating point numbers
SSE2	Pentium 4	Added 16-byte packed double precision floating point numbers and integers
SSE3	Pentium 4 with HT	Added horizontal operations on packed single and double precision floating point
SSE4	P4 & Core i7	Added various instructions not specifically intended for multimedia
SSE5	AMD	Added fused/accumulate and permutation instructions, and precision control

SSE Instruction Set

- Eight 128 bit registers xmm0 … xmm7
- Each register packs
  - 16 bytes (8 bit int)
  - 8 words (16 bit int)
  - 4 doublewords (32 bit int)
  - 2 quadwords (64 bit int)
  - 4 floats (IEEE 754 single precision)
  - 2 doubles (IEEE 754 double precision)
- Note: integer operations are signed or unsigned

SSE Instruction Set

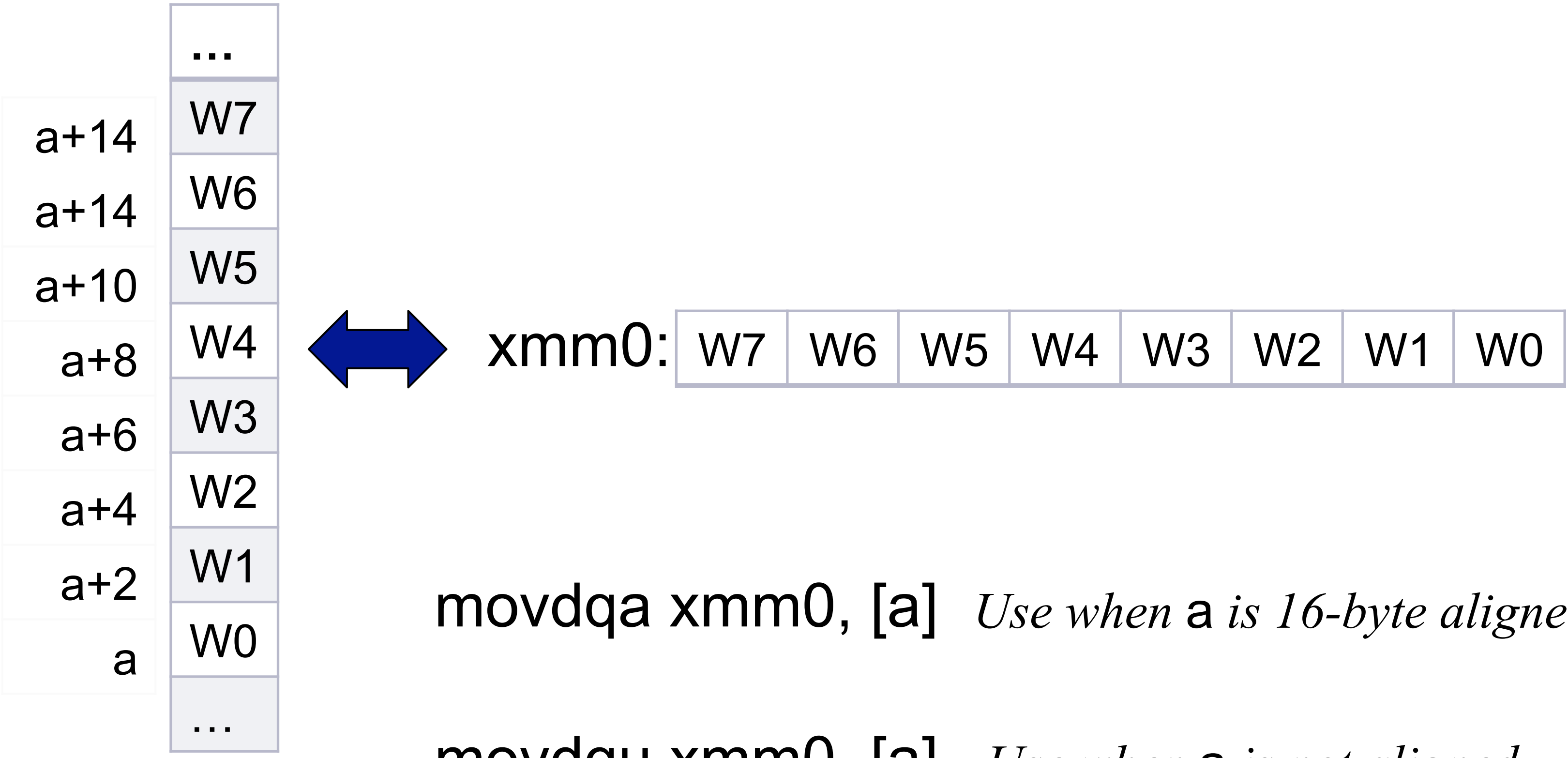
- Instruction format:

  instruction<suffix> xmm, xmm/m128, [imm8/r32]

  m128 is a 128-bit memory location (16-byte aligned address), imm8 is an 8-bit immediate operand, r32 a 32-bit register operand
- Instruction suffix for floating-point operations:
  - ps: packed single precision float
  - pd: packed double precision float
  - ss: scalar (applies to lower data element) single precision float
  - sd: scalar (applies to lower data element) double precision float
- Instruction suffix for integer operations:
  - b: byte
  - w: word
  - d: doubleword
  - q: quadword
  - dq: double quadword

SSE Data Movement

- Little endian order



	…
a+14	W7
a+14	W6
a+10	W5
a+8	W4
a+6	W3
a+4	W2
a+2	W1
a	W0
	…

xmm0:

W7	W6	W5	W4	W3	W2	W1	W0

movdqa xmm0, [a] *Use when* a *is 16-byte aligned*

movdqu xmm0, [a] *Use when* a *is not aligned (expensive!)*

SSE Data Movement

Instruction	Suffix	Description
`movdqa` `movdqu`		Move double quadword aligned Move double quadword unaligned
`mova` `movu`	`ps,pd`	Move single/double precision float aligned Move single/double precision float unaligned
`movhl` `movlh`	`ps` `ps`	Move packed float high to low Move packed float low to high
`moveh` `movel`	`ps,pd` `ps,pd`	Move high packed float (single/double) Move low packed float (single/double)
`mov`	`d,q,ss,sd`	Move scalar data

`lddqu`
`movddup`
`movshdup`
`movsldup`
		Load double quadword unaligned
		Move quadword and duplicate
		Move doubleword and duplicate into high position
		Move doubleword and duplicate into low position

SSE Data Movement

Instruction	Suffix	Description
`pextr` `pinsr`	`w` `w`	Extract word to r32 Insert word from r32
`pmovmsk`	`b`	Move mask
`movmsk`	`ps,pd`	Move mask

Note:
Instructions that start with 'p' historically operate on 64-bit MM registers
Some of these are upgraded by SSE to operate on 128-bit XMM registers

SSE Integer Arithmetic

Instruction	Suffix	Description
`padd`	`b,w,d,q`	Packed addition (signed/unsigned)
`psub`	`b,w,d,q`	Packed subtraction (signed/unsigned)
`padds`	`b,w`	Packed addition with saturation (signed)
`paddus`	`b,w`	Packed addition with saturation (unsigned)
`psubs`	`b,w`	Packed subtraction with saturation (signed)
`psubus`	`b,w`	Packed subtraction with saturation (unsigned)
`pmins`	`w`	Packed minimum (signed)
`pminu`	`b`	Packed minimum (unsigned)
`pmaxs`	`w`	Packed maximum (signed)
`pmaxu`	`b`	Packed maximum (unsigned)

SSE Floating-Point Arithmetic

Instruction	Suffix	Description
`add`	`ss,ps,sd,pd`	Addition (scalar/packed, single/double)
`sub`	`ss,ps,sd,pd`	Subtraction (scalar/packed, single/double)
`mul`	`ss,ps,sd,pd`	Multiplication (scalar/packed, single/double)
`div`	`ss,ps,sd,pd`	Division (scalar/packed, single/double)
`min`	`ss,ps,sd,pd`	Minimum (scalar/packed, single/double)
`max`	`ss,ps,sd,pd`	Maximum (scalar/packed, single/double)
`sqrt`	`ss,ps,sd,pd`	Square root (scalar/packed, single/double)
`rcp`	`ss,ps`	Approximate reciprocal
`rsqrt`	`ss,ps`	Approximate reciprocal square root

SSE Idiomatic Arithmetic

Instruction	Suffix	Description
`pavg`	`b,w`	Packed average with rounding (unsigned)

`pmulh`
`pmulhu`
`pmull`
	`w`
	`w`
	`w`
		Packed multiplication high (signed)
		Packed multiplication high (unsigned)
		Packed multiplication low (signed/unsigned)

`psad` `pmadd`	`bw` `wd`	Packed sum of absolute differences (unsigned) Packed multiplication and addition (signed)

`addsub`
`hadd`
`hsub`
	`ps,pd`
	`ps,pd`
	`ps,pd`
		Floating point addition and subtraction
		Floating point horizontal addition
		Floating point horizontal subtraction

SSE Logical Instructions

Instruction	Suffix	Description
`pand`		Bitwise logical AND
`pandn`		Bitwise logical AND-NOT
`por`		Bitwise logical OR
`pxor`		Bitwise logical XOR
`and`	`ps,pd`	Bitwise logical AND
`andn`	`ps,pd`	Bitwise logical AND-NOT
`or`	`ps,pd`	Bitwise logical OR
`xor`	`ps,pd`	Bitwise logical XOR

SSE Comparison Instructions

Instruction	Suffix	Description
`pcmpeq` `pcmpgt`	`b,w,d` `b,w,d`	Packed compare equal Packed compare greater than
`cmp`	`ss,ps,sd,pd`	Floating-point compare imm8 field is eq, lt, le, unord, neq, nlt, nle, ord Use intrinsic _mm_cmp<cc>_x

SSE Conversion Instructions

Instruction	Suffix	Description

`packss`
`packus`
	`wb,dw`
	`wb`
		Pack with saturation (signed)
		Pack with saturation (unsigned)

`cvt<c>`
`cvtt<c>`
		Conversion
		Conversion with truncation
		c = dq2pd two signed doublewords to two double FP
		c = pd2dq (vice versa)
		c = dq2ps four signed doublewords to four single FP
		c = ps2dq (vice versa)
		c = pd2ps two double FP to two single FP
		c = ps2pd (vice versa)
		c = sd2ss one double FP to one single FP
		c = ss2sd (vice versa)

SSE Shift and Shuffle Instructions

Instruction	Suffix	Description

`psll`
`psra`
`psrl`
	`w,d,q,dq`
	`w,d`
	`w,d,q,dq`
		Shift left logical (zero in)
		Shift right arithmetic (sign in)
		Shift right logical (zero in)

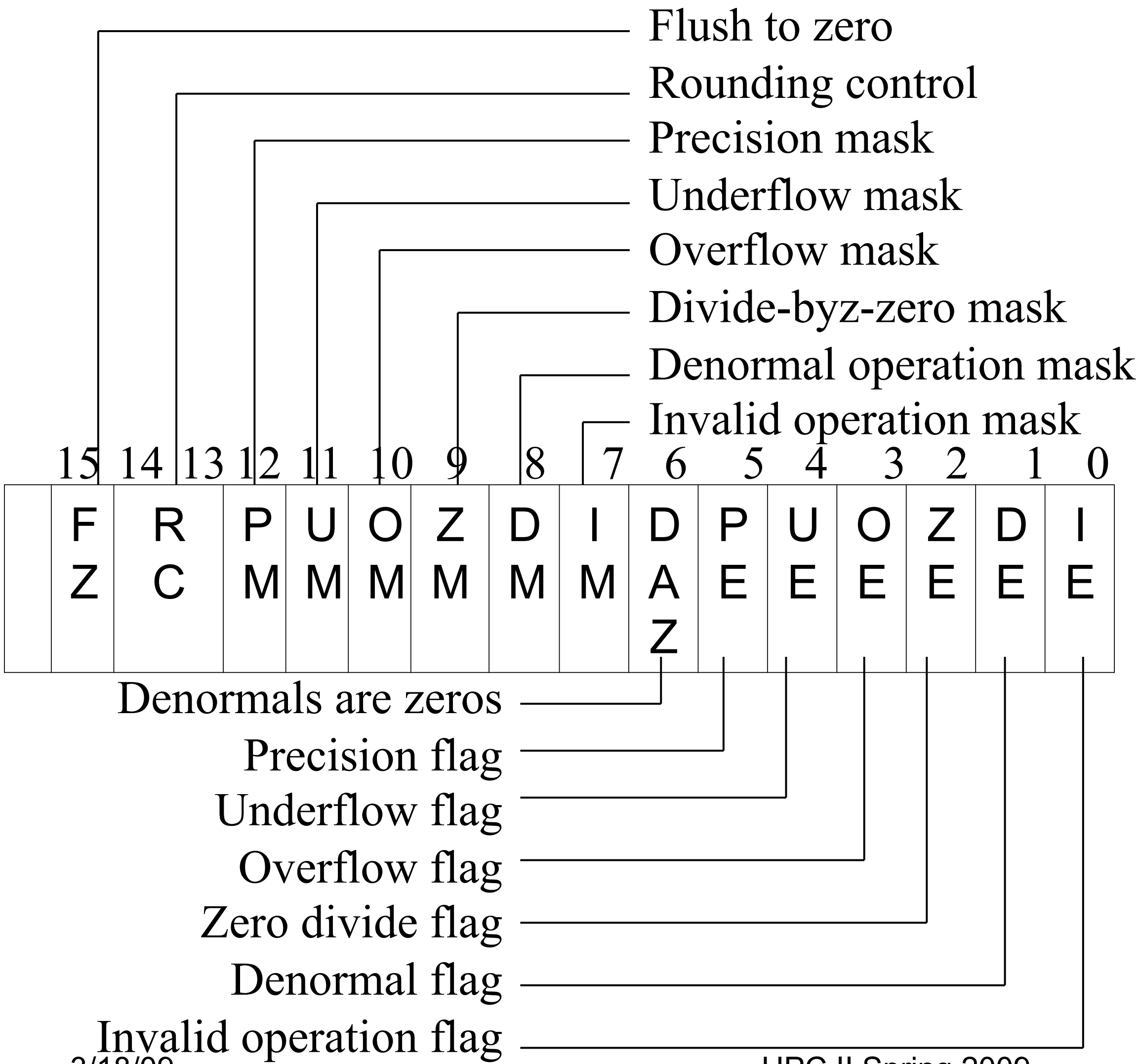
`pshuf`
`pshufh`
`pshufl`
	`w,d`
	`w`
	`w`
		Packed shuffle
		Packed shuffle high
		Packed shuffle low

`shuf`	`ps,pd`	Shuffle, imm8 contains sequence of two (pd) or four (ps) 2-bit encodings of which source operand is stored in the destination operand

SSE Unpack Instructions

Instruction	Suffix	Description
`punpckh` `punpckl`	`bw,wd,dq,qdq` `bw,wd,dq,qdq`	Unpack high Unpack low
`unpckh` `unpckl`	`ps,pd` `ps,pd`	Unpack high Unpack low

MXCSR Control/Status Register



Flush to zero
Rounding control
Precision mask
Underflow mask
Overflow mask
Divide-byz-zero mask
Denormal operation mask
Invalid operation mask

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

	F Z	R C	P M	U M	O M	Z M	D M	I M	D A Z	P E	U E	O E	Z E	D E	I E

Denormals are zeros
Precision flag
Underflow flag
Overflow flag
Zero divide flag
Denormal flag
Invalid operation flag

```
uint32_t setmask = …;
uint32_t oldctrl, newctrl;
_asm {
  STMXCSR oldctrl
  mov eax, oldctrl
  and eax, 0ffffe07fh
  or eax, setmask
  mov newctrl,eax
  LDMXCSR newctrl
}
```

Note: FZ and DAZ improve performance but are not IEEE 754 compatible

Intel SSE Programming

- Programming languages such as C, C++, and Fortran do not natively support SIMD instructions
- The Intel compiler supports four methods to use SSE, from hard (top) to easy (bottom) they are:
  - **Assembly**: direct control, but hard to use and processor-specific
  - **Intrinsics**: similar to assembly instructions with operands that are C expressions, but may be processor-specific
  - **C++ class libraries**: easier to use and portable, but limited support for instructions and gives lower performance
  - **Automatic vectorization**: no source code changes needed, new instruction sets automatically used, but compiler may fail to automatically vectorize code when dependences cannot be disproved

SSE Instruction Intrinsics

- Use **`#include <emmintrin.h>`** (SSE2) or **`<pmmintrin.h>`** (SSE3)
- Data types:

`__m64`	MM register
`__m128`	packed single precision (XMM register)
`__m128d`	packed double precision (XMM register)
`__m128i`	packed integer (XMM register)

- Intrinsics operate on these types and have the format:

  `_mm_`*`instruction`*`_`*`suffix`*`(…)`

  where op is an operation and suffix

`ss,ps`	scalar/packed single precision
`sd,pd`	scalar/packed double precision
`si#`	scalar integer (8, 16, 32, 64, 128 bits)
`su#`	scalar unsigned integer (8, 16, 32, 64, 128 bits)
`[e]pi#`	packed integer (8, 16, 32, 64, 128 bits)
`[e]pu#`	packed unsigned integer (8, 16, 32, 64, 128 bits)

SSE Instruction Intrinsics

- Intrinsics add a number of shorthands for common composite instructions

Instruction	Suffix	Description
_mm_setzero_	si64,si128,ps,pd	Set to zero
_mm_set1_	pi8,pi16,pi32,ps,pd epi8,epi16,epi32,epi64	Set all elements to a value
_mm_set_	(as above)	Set elements from scalars
_mm_setr_	(as above)	Set in reverse order
_mm_load_		MOVA (aligned)
_mm_loadu_		MOVU (unaligned)
_mm_loadr_		MOVA and shuffles to rev
_mm_loadh_		MOVH
_mm_loadl_		MOVL
_mm_load1_		MOV and shuffles

SIMD Instruction Intrinsics Examples

- Load (**`movapd`**) two 16-byte aligned doubles in a vector:

```
double a[2] = {1.0, 2.0}; // a must be 16-byte aligned
__m128d x = _mm_load_pd(a);
```

- Add two vectors containing two doubles:

```
__m128d a, b;
__m128d x = _mm_add_pd(a, b);
```

- Multiply two vectors containing four floats:

```
__m128 a, b;
__m128 x = _mm_mul_ps(a, b);
```

- Add two vectors of 8 16-bit signed ints using saturating arithmetic

```
__m128i a, b;
__m128i x = _mm_adds_epi16(a, b);
```

- Compare two vectors of 16 8-bit signed integers

```
__m128i a, b;
__m128i x = _mm_cmpgt_epi8(a, b);
```

- Note: rounding modes and exception handling are set by masking the MXCSR register

Intrinsics Example 1

```
int array[len];
…
for (int i = 0; i < len; i++)
  array[i] = array[i] + 1;
```

```
#include <emmintrin.h> // SSE2
…
// array of ints, 16-byte aligned
__declspec(align(16)) int array[len];
…
__m128i ones4 = _mm_set1_epi32(1);
__m128i *array4 = (__m128i*)array;
for (int i = 0; i < len/4; i++)
  array4[i] = _mm_add_epi32(array4[i], ones4);
```

Memory Alignment

- Memory operands must be **aligned** for maximum performance
  - 8-byte aligned for MMX
  - 16-byte aligned for SSE
  - Use `_declspec(align(8))` and `_declspec(align(16))`
- Aligned memory load/store operations segfault on unaligned memory operands
  - `__m128d x = _mm_load_pd(aligned_address);`
- Unaligned memory load/store operations are safe to use but incur high cost
  - `__m128d x = _mm_loadu_pd(unaligned_address);`
- Use `_mm_malloc(len, 16)` for dynamic allocation

Data Layout

- Application's data layout may need to be reconsidered to use SIMD instructions effectively
- Vector operations require consecutively stored operands in memory
  - □ Cannot vectorize row-wise with row-major matrix layout
  - □ Cannot vectorize column-wise with column-major matrix layout
- Aligned structs may have members that are unaligned
  - □

```
struct node {
  int x[7];
  int dummy; // padding to make a[] aligned
  float a[4];
}
```

C++ Class Libraries for SSE

- Integer class types of the form I*b*vec*n*

`I8vec8`	(8 8bit)	`I8vec16`	(16 8bit)
`I16vec4`	(4 16bit)	`I16vec8`	(8 16bit)
`I32vec2`	(2 32bit)	`I32vec4`	(4 32bit)
`I64vec1`	(1 64bit)	`I64vec2`	(2 64bit)
`I128vec1`	(1 128bit)		

  Note: place an ‘`s`’ or ‘`u`’ after ‘`I`’ for packed signed or packed unsigned integers, e.g. `Is32vec4`

- Floating point class types of the form F*b*vec*n*

`F32vec4`	(4 32bit)	`F64vec2`	(2 64bit)

C++ Class Library Example

```
#include <dvec.h> // SSE2
…
// array of ints, 16-byte aligned
__declspec(align(16)) int array[len];
…
Is32vec4 *array4 = (Is32vec4*)array;
for (int i = 0; i < len/4; i++)
  array4[i] = array4[i] + 1; // increment 4 ints
```

GMP:
GNU Multi-Precision Library

- GMP is a portable library written in C for arbitrary precision arithmetic on integers, rational numbers, and floating-point numbers
- GMP aims to provide the fastest possible arithmetic for all applications that need higher precision than is directly supported by the basic C types
- Used by many projects, including computer algebra systems
- Programming language bindings: C, C++, Fortran, Java, Prolog, Lisp, ML, Perl, …
- License: LGPL

GMP Usage

- Introduces three types (C language binding):

`mpz_t`	bigint
`mpq_t`	big rational
`mpf_t`	bignum

- Use (similar for mpq and mpf):

```
#include <gmp.h>
mpz_t n;
mpz_init(n);
mpz_init2(n, 123);
mpz_init_set_str(n, “6”, 10);
…
mpz_clear(n);
```

Use one of these to initialize.
Note: **`mpf_init2`** sets precision

base

- Link with **`-lgmp`**

GMP

- Dynamic memory allocation
  - □ Efficient implementation limits the need for frequent resizing
  - □ Configurable
- 150 integer operations on unlimited length bigint
  - □ Arithmetic
  - □ Comparison
  - □ Logic and bit-wise operations
  - □ Number theoretic functions
  - □ Random numbers
- 60 floating point operations on high-precision bignum
  - □ Arithmetic
  - □ Comparison

GMP C Example

```
void myfunction(mpz_t result, mpz_t param, unsigned long n)
{
  unsigned long  i;

  mpz_mul_ui(result, param, n);
  for (i = 1; i < n; i++)
    mpz_add_ui(result, result, i*7);
}

int main(void)
{
  mpz_t  r, n;
  mpz_init(r);
  mpz_init_set_str(n, "123456", 0);

  myfunction(r, n, 20L);
  mpz_out_str(stdout, 10, r); printf("\n");

  return 0;
}
```

GMP C++ Bindings

- Defines three classes:

`mpz_class`	for bigint
`mpq_class`	for big rationals
`mpf_class`	for bignum

- Most GMP functions have C++ wrappers, but not all
  - □ Root of 0.2 in 1000 bit precision:

```
mpf_class x(0.2, 1000), y(sqrt(x));
```

  - □ GCD of two bigints:

```
mpz_class a, b, c;
…
mpz_gcd(a.get_mpz_t(), b.get_mpz_t(), c.get_mpz_t());
```

- Use **`#include <gmpxx.h>`** and link **`-lgmpxx -lgmp`**

GMP C++ Example

```
#include <gmpxx.h>

mpz_class a, b, c; // integers

a = 1234;
b = "-5678";
c = a+b;
cout << "sum is " << c << "\n";
cout << "absolute value is " << abs(c) << "\n";
```

Expression like a=b+c results in a single call to the corresponding **`mpz_add`**, without using a temporary for the b+c part.

The classes can be freely intermixed in float, double, int/long, expressions.

Further Reading

- "*What Every Computer Scientist Should Know About Floating Point Arithmetic*" by D. Goldberg, Computing Surveys, 1991 http://docs.sun.com/source/806-3568/ncg_goldberg.html
- Chapters 11 and 12 of "*The Software Optimization Cookbook*" 2nd ed by R. Gerber, A. Bik, K, Smith, and X. Tian, Intel Press.
- "The Software Vectorization Handbook", A. Bik, Intel Press.
- Intel Compiler intrinsics reference: http://download.intel.com/support/performancetools/c/linux/v9/intref_cls.pdf
- GNU GMP: http://gmplib.org