Computer Organization for Supercomputers

Elementary Machine Data Types
IEEE 754 Arithmetic and Exceptions

Texas A&M University Supercomputing Facility

Instructor Michael E. Thomadakis, Ph.D.
http://sc.tamu.edu/shortcourses
Outline

- **Binary System Review**
- **Elementary Data Types**
  - Binary Representation
  - Storage Layout
- **Numeric Types**
  - Integer-valued
  - Real-valued
- **IEEE 754 Binary Floating-Point System**
  - Floating point number representation
  - Floating point arithmetic
  - Floating point exceptions
  - Exception handling in IBM Power 5+ / AIX 5L
  - Exception handling in Intel Nehalem / Linux 2.6.x
Basic Data Types

- **Basic Data Types (BDTs) for digital H/W**
- **Numeric Types**
  - *Integers*:
    - signed or unsigned
    - 1, 2, 4, 8 bytes each;
  - *Floating Point*:
    - single, double, extended and long precision
    - 4, 8, 16, or more bytes each
- **Non-Numeric Types**
  - *Characters*:
    - ASCII 7-bit standard; 1 byte / char
      - 'a', 'b', ..., '1', '2', ..., '!', '@', ...
    - UNICODE 16-bit standard ("wide characters")
  - *Strings*:
    - 1 or 2 bytes per character;
    - '\0' at the end of string; or count and string
      “The smart fox jumped over the lazy dog.”
- **Composite**: arbitrary aggregates of BDTs
Intro to Binary Systems

- Use *binary system* to encode digitized “information”
  - base $B = 2$
  - let $x$ be a “binary” variable; $x \in \{0, 1\}$, → one “memory bit” needed for each $x$
  - let $X = (x_1, x_2, ..., x_M)$, an $M$ component vector, $x_i$ a binary → $M$ bits of storage or an “$M$-bit memory word”
  - 8 bits form a byte (usually ;)
- For Numeric types, define binary
  - representation
    - $M$-bit representation
  - operations
    - arithmetic $+,-,\times,\div,...$
    - logical $\neg,\land,\lor,\text{xor},...$
    - relational $>,<,=,\neq,...$
- Non-Numeric types are simple mappings
  - integer values to “character set” location
$M$-bit Integer Quantities and Representation

- Use $M$ position binary word
  - $M = 8, 16, 32$ (4 bytes), $64$ (8 bytes)
- Unsigned (non-negative) integer quantities
  - unsigned char UC=1; unsigned short uS=4;
    unsigned uI=3; unsigned long long uL;
- Signed integer quantities
  - char C=1; short S=2; int I=2;
    long long LL = -2LL;
$M$-bit Integer Quantities and Representation

- $M$ position binary object
- Big Endian vs. Little Endian Storage
  - big endian: address of object = address of most significant byte
    - usually RISC processors (Power5+, Alpha, Sparc, MIPS, etc.)
  - little endian: address of object = address of least significant byte
    - usually CISC processors (Intel processors)
- Significant when we transfer binary data across systems

![Binary Representation Diagram]

$M = 32$

$M = 64$

Texas A&M University Michael E. Thomadakis  IEEE Floating-Point Arithmetic July 2010
Unsigned $M$-bit Integers

- Let $M$ bit word storing an **unsigned** integer $X_{(2)}$
- **Value** $D_{(10)}$ ($B=10$) of **unsigned** binary integer $X_{(2)}$
  - unsigned binary integers use positional numbers
  - $D_{(10)} = \sum x_i \times 2^i$, $i = 0, 1, ..., M-1$, and in general
  - $D_{(10)} = \sum x_i \times B^i$, $i = 0, 1, ..., N-1$, from base $B$
  - Conversion of $X$ in $B=b_1$ to $Y$ in $B=b_2$, successively divide $X$ by $b_2$, and collect the remainders from left to right
- **Range of Values**
  - minimum $= 0$ (integer underflow $\equiv X<0$)
  - maximum $= 2^M - 1$ (integer overflow $\equiv X>2^M-1$)
Example: Unsigned 32-bit Integer

° Let $M = 32$
° 32-bit binary representation of (unsigned) number:

- $b_{31} \times 2^{31} + b_{30} \times 2^{30} + \cdots + b_{2} \times 2^{2} + b_{1} \times 2^{1} + b_{0} \times 2^{0}$

- One billion $1,000,000,000_{10}$ in binary is

$0011\ 1011\ 1001\ 1010\ 1100\ 1010\ 0000\ 0000_{2}$

$= 1\times2^{29} + 1\times2^{28} + 1\times2^{27} + 1\times2^{25} + 1\times2^{24} + 1\times2^{23} + 1\times2^{20} + 1\times2^{19} + 1\times2^{17} + 1\times2^{15} + 1\times2^{14} + 1\times2^{11} + 1\times2^{9}$

$= 536,870,912 + 268,435,456 + 134,217,728 + 33,554,432 + 16,777,216 + 8,388,608 + 1,048,576 + 524,288 + 131,072 + 32,768 + 16,384 + 2,048 + 512$
$M$-bit Signed Integers: Two's Complement

- Let $M$-bit word storing a signed integer $X_{(2)}$
- Representation in “two's complement”
  - $-X := (2^M - X)$, negative of $X$
  - $Y - X := Y + (2^M - X)$, subtraction
  - if $x_{M-1} = 0$, $X \geq 0$;
  - if $x_{M-1} = 1$, $X < 0$;
- **Value** $D_{(10)}$ (B=10) of signed binary integer $X_{(2)}$
  - $D_{(10)} = \sum x_i \times 2^i - (x_{M-1} \times 2^{(M-1)})$, $i = 0, 1, ..., M-2$,
- **Range of Values**
  - minimum $= -2^{(M-1)}$
    - integer underflow $\equiv X < -2^{(M-1)}$
  - maximum $= 2^{(M-1)} - 1$
    - integer overflow $\equiv X > 2^{(M-1)} - 1$
8-bit Two’s Complement Integers

- Signed Char
- \( M = 8 \)
  
  \[
  \begin{align*}
  +3 & = 0000\ 0011 \\
  +2 & = 0000\ 0010 \\
  +1 & = 0000\ 0001 \\
  +0 & = 0000\ 0000 \\
  -1 & = 1111\ 1111 \\
  -2 & = 1111\ 1110 \\
  -3 & = 1111\ 1101 \\
  \end{align*}
  \]
Example Two’s Complement 32-bit Integer

- Recognizing role of sign bit, can represent positive **and negative** numbers in terms of the bit value times a power of 2:
  \[(d_{31} \times -2^{31}) + d_{30} \times 2^{30} + \cdots + d_2 \times 2^2 + d_1 \times 2^1 + d_0 \times 2^0\]

- Example (given 32-bit two’s comp. number)
  1111 1111 1111 1111 1111 1111 1111 1100\_two

  \[= 1 \times -2^{31} + 1 \times 2^{30} + 1 \times 2^{29} + \cdots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0\]

  \[= -2^{31} + 2^{30} + 2^{29} + \cdots + 2^2 + 0 + 0\]

  \[= -2,147,483,648_{10} + 2,147,483,644_{10}\]

  \[= -4_{10}\]
2’s Complement Shortcut

- To convert integer $X$ into its 2's complement $-X$
  - invert every bit,
  - $x_i' = \sim x_i$ or equivalently
  - $x_i' = x_i \text{ XOR } 1$
  - then add 1 to the result
- Let $x'$ mean the inverted representation of $x$
- Then $x + x' = -1 \iff x + x' + 1 = 0 \iff x' + 1 = -x$

**Example: -4 to +4 to -4**
- $x : 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1100_2$
  - $x' : 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0011_2$
  - $+1 : 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0100_2$
  - $(\cdot)' : 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1011_2$
  - $+1 : 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1100_2$
Negation Special Case 1

- 0 = 00000000
- Bitwise not = 11111111
- Add 1 to LSB = +1
- Result = 1 00000000
- Overflow is ignored, so:
  - $-0 = 0 \sqrt{\phantom{.}}$
Negation Special Case 2

- $-128 = 10000000$
- bitwise not $01111111$
- Add 1 to LSB $+1$
- Result $10000000$
- So:
  - $-(-128) = -128 \times X$
  - Monitor MSB (sign bit)
  - It should change during negation
Benefits of 2's Complement

- 2's Complement is the representation of choice
  - can use the same HW for both addition and subtraction
  - one representation of zero
  - arithmetic works easily
  - negating is fairly easy
Addition and Subtraction

- Normal binary addition
- Monitor sign bit for overflow
- Take twos complement of subtrahend and add to minuend, *i.e.*, 
  \[ a - b = a + (-b) \]
- So we only need addition and complement circuits
Two’s complement: Sign extension

- Convert 2's complement number represented in $k$ bits to more than $k$ bits
  - e.g., 16-bit integer converted to 32-bits integer
- Simply **replicate** the most significant bit (sign bit) of smaller quantity to fill new bits
  - 2's comp. positive number has **infinite 0s** to left
  - 2's comp. negative number has **infinite 1s** to left
  - Finite representation hides most leading bits; sign extension restores those that fit in the integer variable
- **16-bit $-4_{10}$ to 32-bit:**

```
1111 1111 1111 1111 1100
```

```
1111 1111 1111 1111 1100
```
Signed $M$-bit Integers: Sign-Magnitude

- Alternative signed integer representation
  - Left most bit is sign bit
  - 0 means positive
  - 1 means negative

- Example
  - $+18 = 00010010$
  - $-18 = 10010010$

- Problems
  - Need to consider both sign and magnitude in arithmetic
  - Two representations of zero ($+0$ and $-0$)
  - Adder hardware may need extra step since can’t always know sign of result in advance

- Sing-Magnitude was abandoned
## Bit-Pattern, Unsigned, 2’s Comp, 1’s Comp, Biased

$$b_3b_2b_1b_0$$

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>15</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>-2</td>
<td>-1</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>-3</td>
<td>-2</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>-4</td>
<td>-3</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>-5</td>
<td>-4</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>-6</td>
<td>-5</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>-7</td>
<td>-6</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>-8</td>
<td>-7</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Bias = 8
Good Coding Habits

- Understand ILP32 and LP64 universal data models
  - **ILP32**: Int Long Pointer need 32 bits
  - **LP64**: Long Pointer need 64 bits
  - **ILP64**: Int Long Pointer need 64 bits
- Do NOT assume that
  - `sizeof(int) == sizeof(void *)`
  - `sizeof(int) == sizeof(long)`
  - `sizeof(long) == 4`
  - `sizeof(*void) == 4`
- Avoid
  - implicitly declared functions
  - constants with the high-order bit set
  - arithmetic with long types (including shifts involving mixed types and code that may overflow 32 bits)
Floating Point Numbers and Arithmetic

- **Floating Point Representation**
  - Scientific Notation with base $B=10$

- **Issues:**
  - Arithmetic operations? ($+, -, \times, \div$, remainder, sqrt, exp)
  - Representation, normal forms
  - Range and Precision
  - Rounding
  - Exceptions (e.g., divide by zero, overflow, underflow)
  - Errors
  - Properties (negation, inversion, if $A \neq B$ then $A - B \neq 0$)
Fixed-Point Numerical Representation

- Fixed Point numbers use a radix point at fixed location
- A fixed point number $X$ stored in $M$-bit vector
  - $M = M_1 + M_2$ with $M_1$ integer and $M_2$ fraction bits (to the right of binary point)
  - $X = 0110\ 1111\ .\ 1010...0000$
  - if $M_1 = 1$ and $x_0 = 1$, then normalized representation
    - $1.101\ 1111\ 010...0000$ (problems?)
- Value $V_{(10)}$ of fixed point binary integer $X_{(2)}$
  - $V_{(10)}X_{(2)} = \sum x_i \times 2^i, \ i = -M_2, ..., 0, 1, ..., M_1 - 1$
- 2's Complement can be used
- Fixed point representation is restrictive and cannot use the storage efficiently.
- Cannot represent too big or too small numbers
Scientific Notation in Binary $B = 2$

- Similar to standard scientific notation just in binary form

- **Normalized form**: $1.xxxxxxxxxxxxx_2 \times 2^{yyyy2}$
  - Simplifies data exchange, increases accuracy
IEEE 754 Standard Binary Floating Point Arithmetic

• ANSI/IEEE 754-2008 Standard
  • international standard for representing and calculating with floating-point numerical quantities in binary
  • initially formalized in 1985 (ieee754-1985)
  • designed for portability of representation
  • produce repeatable results across h/w platforms
  • provides well defined concepts of overflow, underflow, accuracy and round-off errors
  • some parts of IEEE 754 are expensive to implement and most vendors allow programs to bypass strict adherence to IEEE arithmetic (“fast arithmetic”)
    • faster arithmetic → lower precision
IEEE 754 Requirements

• IEEE 754 defines and requires support for
  – FP arithmetic \(+, -, \times, /\), remainder, square root
  – conversions between binary and decimal representations
  – 32, 64-bit and extended precision intermediate results and binary radix
• Specifies what should be done when results fall \textit{outside} the range of valid FP quantities
• Issues
  – Arithmetic operations? (\(+, -, \times, /\))
  – Representation, normal forms
  – Range and Precision
  – Rounding
  – Exceptions (\textit{e.g.}, divide by zero, overflow, underflow)
  – Errors
  – Properties (negation, inversion, if \(A \ne B\) then \(A - B \ne 0\))
Floating-Point Numerical Representation

- **Floating-Point, Normalized**, representation stores numbers in a fixed number \( N \) of bits and uses part of this to store the value of an exponent \( E \)

  - Let \( X = -0110 \ 1111 \ . \ 1010...0000 \times 2^0 \); normalize it:
    - \( X = -1. \ 101 \ 111... \times 2^6 \)
    - exponent \( E = 6 \) \( N_3 \) bits (radix 2)
    - **Mantissa** = 1.101 111... \( N_2 + 1 \) bits
    - sign \( S = 1 \) 1 bit;
      - \( S=0, X > 0; S=1, X < 0 \)
    - \( N = 2 + N_2 + N_3 \) bits

- **Value** \( V \) of floating point \( X \) is given by
  - \( V(X) = (-1)^S \times \text{Mantissa} \times 2^E \)
    \[ = - \left( \sum x_i \times 2^i \right) \times 2^6, \ i = -N_2, ..., \ -1, \ 0 \]
IEEE 754 Single-Precision Floating Point

- **Single Precision** IEEE 754 Floating-Point Layout
  - `float a = 0.123E-10; /* C */`
  - $M=32$ memory bits; more bits in the FPALU
  - stores fractional part in *normalized* form
    - MSb (always '1') is not stored
- Form of Single precision FP $f$
  - $S$: 1 bit for sign ;
  - **Exponent**: 8 bits; **Exponent** = $E + SBIAS$
    - $SBIAS = 127$ (“exponent bias”)
    - $-126 \leq E \leq 127$
  - **Significand**: 23 bits; **Significand** = Mantissa − 1.0 and
    - $0 < \text{Significand} < 1$
Decimal Value of Single-Precision Floating Point

- Given SP FP \( X \), to convert to IEEE representation
  - normalize \( X \) and adjust \( E \);
  - set Significand = mantissa \(- 1.0\);
  - set Exponent = \( E + 127 \); (**)
  - if \( X \geq 0 \) then \( S = 0 \)
    else \( S = 1 \);

- Value \((X) = (-1)^{S} \times (1 + \text{Significand}) \times 2^{(\text{Exponent} - 127)}\)
  = \((-1)^{S} \times \text{Mantissa} \times 2^{E}\)
Single-Precision FP Overflow and Underflow

- **Single Precision Absolute Value Ranges**
  - maximum: \((2^{24} - 1) \times 2^{104} \cong 2.0 \times 10^{38}\)
  - minimum: \(2^{-126} \cong 2.0 \times 10^{-38}\)

- **Overflow** \(\iff\) exponent \(E > 127\)
  - larger than can be represented in 8-bit Exponent field
  - result > \(2.0 \times 10^{38}\)

- **Underflow** \(\iff\) Negative exponent \(E < -126\)
  - result > 0 and result < \(2.0 \times 10^{-38}\)

- Overflows and underflows generate “exceptions” also known as “traps” that are under user control

- Exceptions take a lot of time to handle relative to the time it takes to handle even the heaviest FP operation
IEEE 754 Double-Precision Floating Point

- **Double Precision**
  - `double A = 0.123E-100; /* C */`
  - $M=64$ memory bits; more bits in the FPALU
  - fractional part is in *normalized* form

- **Double precision FP $D$ memory layout**
  - $S$: sign bit; 1 bit;
  - **Exponent**: 11 bits; Exponent = $E + D_{BIAS}$
    - $D_{BIAS} = 1023$
    - $-1022 \leq E \leq 1023$
  - **Significand**: 52 bits; Significand = Mantissa − 1.0 and
    - $0 < \text{Significand} < 1$

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>20</th>
<th>19</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Exponent</td>
<td>Significand</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 bit</td>
<td>11 bits</td>
<td>20 bits</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Significand (cont’d here)

32 bits
Double-Precision FP Overflow and Underflow

- **Single Precision Absolute Value Ranges**
  - maximum: \((2^{53} - 1) \times 2^{971} \approx 10^{308}\)
  - minimum: \(2^{-1022} \approx 10^{-308}\)
  - advantage over single-precision is greater accuracy

- **Overflow** ⇔ exponent \(E > 1023\)
  - larger than can be represented in 11-bit Exponent field
  - result > \(2.0 \times 10^{308}\)

- **Underflow** ⇔ Negative exponent \(E < -1022\)
  - result > 0 and result < \(2.0 \times 10^{-308}\)

- Overflows and underflows generate “exceptions” also known as “traps” that are under user control
Special Values for SP $\infty$ and NaN

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Significand</th>
<th>Numeric Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 0</td>
<td>nonzero</td>
<td>???????????</td>
</tr>
<tr>
<td>1-254</td>
<td>anything</td>
<td>+/- FP number</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>+/- infinity</td>
</tr>
<tr>
<td>255</td>
<td>nonzero</td>
<td>NaN</td>
</tr>
</tbody>
</table>

- **Single Precision Infinity**
  - exponent = 255, significand = 0
- **Single Precision NaN**
  - e.g., sqrt(-4)
  - exponent = 255, significand ≠ 0
  - NaNs propagate (“contaminate”) FP operations
    - $(\text{NaN} \theta X) \equiv \text{NaN}$
  - Valid NaN operations are '==' and '!='
Special Values for DP $\infty$ and NaN

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Significand</th>
<th>Numeric Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>nonzero</td>
<td>???????????</td>
</tr>
<tr>
<td>1-2046</td>
<td>anything</td>
<td>+/- FP number</td>
</tr>
<tr>
<td>2047</td>
<td>0</td>
<td>+/- infinity</td>
</tr>
<tr>
<td>2047</td>
<td>nonzero</td>
<td>NaN</td>
</tr>
</tbody>
</table>

- **Double Precision Infinity**
  - exponent = 2047, significand = 0
- **Double Precision NaN**
  - exponent = 2047, significand ≠ 0
  - e.g., sqrt(-4)
  - NaNs propagate (“contaminate”) FP operations
    - (NaN $\theta X$) $\equiv$ NaN
    - Valid NaN operations are '==' and '!='

- Valid NaN operations are '==', and '!='
Elaboration: Infinity and NaNs

**Infinity** Result of operation *overflows*, *i.e.*, is larger than the largest number that can be represented

Overflow is not the same as divide by zero (raises a different exception)

\[ \pm \infty \quad S \ 1 \ldots 1\ 0\ldots 0 \]

It may make sense to do further computations with infinity *e.g.*, \(X/0 > Y\) may be a valid comparison

**Not a Number**, but not infinity (*e.g.*, \(\sqrt{-4}\)) invalid operation exception (unless operation is = or ≠)

\[ NaN \quad S \ 1 \ldots 1 \text{ non-zero} \quad \text{HW decides what goes here} \]

NaNs propagate: \(f(NaN) = NaN\)

---

Texas A&M University Michael E.Thomadakis  
IEEE Floating-Point Arithmetic July 2010
Floating-Point Addition (Basic Algorithm)

For addition (or subtraction) of $X$ to $Y$ ($X < Y$):

1. Compute $D = \text{Exp}_Y - \text{Exp}_X$

2. Right shift $(1+\text{Sig}_X)D$ bits $\Rightarrow (1+\text{Sig}_X)2^{-D}$ (align binary points)

3. Compute $(1+\text{Sig}_X)2^{-D} + (1+\text{Sig}_Y)$; Normalize if necessary; continue until MS bit is 1

4. Too small (e.g., 0.001xx...) left shift result, decrement result exponent; check for underflow

4'. Too big (e.g., 10.1xx...) right shift result, increment result exponent; check for overflow

5. If result significand is 0, set exponent to 0
Floating-Point Subtraction

- **Similar to addition**
  - De-normalize to match exponents
  - Subtract significands
  - Keep the same exponent
  - Normalize (possibly changing exponent)
- **Problems in implementing FP add/sub:**
  - Managing the signs,
  - determining to add or sub,
  - swapping the operands.
FP Binary Multiplication (Basic Algorithm)

For multiplication of $P = X \times Y$:

1. **Compute Exponent:** $\text{Exp}_P = (\text{Exp}_Y + \text{Exp}_X) - \text{Bias}$

2. **Compute Product:** $(1 + \text{Sig}_X) \times (1 + \text{Sig}_Y)$
   Normalize if necessary; continue until most significant bit is 1

4. **Too small** (e.g., 0.001xx...) →
   left shift result, decrement result exponent

4'. **Too big** (e.g., 10.1xx...) →
   right shift result, increment result exponent

5. If (result significand is 0) then set exponent to 0

6. if $(\text{Sgn}_X = \text{Sgn}_Y)$ then
   $\text{Sgn}_P = $ positive (0)
   else
   $\text{Sgn}_P = $ negative (1)
A Basic Floating Point ALU

FP ADD:
Exponents are subtracted by small ALU; the difference controls the 3 MUXes;

- Separate datapath from integer datapaths with FP-ALU
Elaboration: Add / Multiply Issues

Let two FP numbers $X$ and $Y$; assume $Y_e > X_e$

**Addition (or subtraction) steps:**
1. compute $Y_e - X_e$ (prepare to align binary point)
2. right shift $X_m$ that many positions to form $X_m \times 2^{X_e - Y_e}$
3. compute $(X_m \times 2^{X_e - Y_e}) + Y_m$

If representation demands normalization, then normalization step follows:

- left shift result, decrement result exponent (e.g., 0.001xx...)
- right shift result, increment result exponent (e.g., 101.1xx...)
- continue until MSb of data is 1 (NOTE: Hidden bit in IEEE Standard)

For **Multiply**, doubly biased exponent must be corrected:

- $X_e = 7$
- $Y_e = -3$
- Excess 8 extra subtraction step of the bias amount

If result is 0 mantissa, may need to zero exponent by special step

$$
\begin{align*}
X_e &= 1111 & = 15 & = 7 + 8 \\
Y_e &= 0101 & = 5 & = -3 + 8 \\
10100 &= 20 & = 4 + 8 + 8
\end{align*}
$$
Rounding Issues and IEEE Rounding

- When we perform math on “real” numbers, we have to worry about rounding to fit the result in the significant field.

- The FP hardware carries at least two extra bits of precision, and then round to get the proper value.

- Rounding also occurs when converting a double to a single precision value, or converting a floating point number to an integer:
  - **Round towards** $\infty$
    - ALWAYS round “up”: $2.001 \rightarrow 3$
    - $-2.001 \rightarrow -2$
  - **Round towards** $-\infty$
    - ALWAYS round “down”: $1.999 \rightarrow 1,$
    - $-1.999 \rightarrow -2$
  - **round towards 0** (Truncate)
    - Just drop the last bits
  - **Round to (nearest) even**
    - Normal IEEE rounding, almost
Round to Nearest Even

- If the value is right on the borderline, we round to the nearest EVEN number
  - 2.5 → 2
  - 3.5 → 4

- Insures *fairness* on calculation in the long term
  - \( P\{\text{we round up on tie}\} = P\{\text{we round down}\} \rightarrow 0.5 \)
  - if bit patterns are actually uniformly distributed

- This is the *default* IEEE rounding mode
Extra Bits for Rounding

- Extra “Guard” digits needed for rounding
  - IEEE: “As if computed the result exactly and rounded”
- Extra Digits: Guard, Round and Sticky
  - let there be $p$ digits to the right of radix point
  - 3 extra ("guard") digits to the right of the $p$ digits of significand to guard against loss of digits – can later be shifted left into first $p$ places during normalization.
  - Addition: carry-out shifted in
  - Subtraction: borrow digit and guard
  - Multiplication: carry and guard; Division: guard

Addition:

<table>
<thead>
<tr>
<th></th>
<th>1.xxxxx</th>
<th>1.xxxxx</th>
<th>1.xxxxx</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ 1.xxxxx</td>
<td>0.001xxxxx</td>
<td>0.01xxxxx</td>
<td></td>
</tr>
<tr>
<td>1x.xxxxxy</td>
<td>1.xxxxxyy</td>
<td>1x.xxxxxyy</td>
<td></td>
</tr>
</tbody>
</table>

post-normalization  pre-normalization  pre and post
Rounding Elaboration: Sticky Bit

Additional bit to the right of the round digit to better fine tune rounding

\[
+ \begin{array}{c}
d_0 \cdot d_1 d_2 d_3 \ldots d_{p-1} 0 0 0 \\
0 . 0 0 X \ldots X X X S \\
\hline
XXS
\end{array}
\]

Sticky bit: set to 1 if any 1 bits fall off the end of the round digit

\[
- \begin{array}{c}
d_0 \cdot d_1 d_2 d_3 \ldots d_{p-1} 0 0 0 \\
0 . 0 0 X \ldots X X X 1 \\
\hline
XX0
\end{array}
\]

generates a borrow

Rounding Summary

Radix 2 minimizes wobble in precision

Normal operations in +, −, *, / require one carry/borrow bit + one guard digit

One round digit needed for correct rounding

Sticky bit needed when round digit is B/2 for max accuracy

Rounding to nearest has mean error = 0, if uniform distribution of digits are assumed
Representation for “Denormals” (1/2)

- **Problem**: There’s a gap among representable FP numbers around 0
  - Smallest representable positive SP FP quantity:
    - $a = 1.0\ldots \times 2^{-126} = 2^{-126}$
  - Second smallest representable positive num:
    - $b = 1.000\ldots1 \times 2^{-126} = 2^{-126} + 2^{-149}$
    - $a - 0 = 2^{-126}$
    - $b - a = 2^{-149}$
Representation for Denormals (2/2)

- **Solution:**
  - We still haven’t used Exponent = 0, Significand ≠ 0
  - Denormalized number: no leading 1, exponent = -126.
  - “Gradual underflow” condition
  - Smallest representable positive number: 
    \[ a = 2^{-149} \]
  - Second smallest representable pos num: 
    \[ b = 2^{-148} \]
  - **Problem:** denormals are handled usually by SW and are thus very expensive.
Denormalized Numbers

The gap between 0 and the next representable number (one with magnitude > 0) is much larger than the gaps between nearby representable numbers. IEEE standard uses denormalized numbers to fill in the gap, making the distances between numbers near 0 more alike.

same spacing, half as many values!
### Special Values for SP and DP Denormals

- **Special FP values**

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Significand</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>nonzero</td>
<td>Denormal</td>
</tr>
<tr>
<td>1-254</td>
<td>anything</td>
<td>+/- FP number</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>+/- infinity</td>
</tr>
<tr>
<td>255</td>
<td>nonzero</td>
<td>NaN</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Significand</th>
<th>Numeric Object</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>nonzero</td>
<td>Denormal</td>
<td></td>
</tr>
<tr>
<td>1-2046</td>
<td>anything</td>
<td>+/- FP number</td>
<td></td>
</tr>
<tr>
<td>2047</td>
<td>0</td>
<td>+/- infinity</td>
<td></td>
</tr>
<tr>
<td>2047</td>
<td>nonzero</td>
<td>NaN</td>
<td></td>
</tr>
</tbody>
</table>
IEEE 754 Exceptions - 1

- An FP Exception (FPE) is an invalid or undesirable state of the FP computation, which requires special handling
  - An exception is not an error unless handled badly
  - Need to understand the nature and impact of FPEs
- Five types of exceptions rigorously defined by the 754 standard
  - overflow
  - underflow
  - division by zero
  - inexact results
  - invalid operation
IEEE 754 Exceptions - 2

- IEEE 754 Floating-Point Exceptions
  1. **overflow**: exponent of a value is too large to be represented
  2. **underflow**: exponent of a value is too small
  3. **division by zero**: finite quantity divided by 0.0
  4. **inexact results**: computed value cannot be represented exactly, so a rounding error is introduced.

  ▪ This exception is *very common* and results in *rounding*;
IEEE 754 Exceptions - 3

- IEEE 754 Floating-Point Exceptions

5) invalid operation: operations performed on values for which the results are not defined, including

- infinity − infinity
- − infinity + infinity
- ±infinity × 0
- ±0.0 / ±0.0;
- f(SNaN)
- sqrt(X), X < 0
- log(X), X < 0
- asin(X), |X| > 1
- mod(x, y), x = INF or y = 0.0
- FP → integer when result cannot be represented faithfully
IEEE 754 Exceptions - 4

• Users can select one of three modes of FP Exception Handling
  – **Default**: IEEE 754 compliant h/w returns default value
  – **Compiler-Aided**: instruct compiler (Fortran, C/C++) to execute (“trap to”) predetermined FP routines upon FPE
  – **User Directed**: user instructs system to generate SIGFPE and programs routines to handle this signal in his code

• In modern super-pipelined, super-scalar, out-of-order execution processors (Power4 / 5, Itanium-2, MIPS R10K, etc.)
  – FP exceptions are *inexact*
  – the system may not be able to determine the instruction which caused it
IEEE 754 Default Exception Handling

- **Default**: IEEE 754 compliant h/w
  - set status flags in FP hardware (type and cause of FPE)
  - substitute default value for the result of the operation,
  - return to user program which *continues*;
- This method has the **least overhead** and allows programs to finish as early as possible.
- Results may vary unexpectedly and there is no user control

<table>
<thead>
<tr>
<th>FP Exception</th>
<th>Default IEEE 754 Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>overflow</td>
<td>± Infinity</td>
</tr>
<tr>
<td>underflow</td>
<td>± 0</td>
</tr>
<tr>
<td>invalid operation</td>
<td>NaN or FALSE</td>
</tr>
<tr>
<td>division by zero</td>
<td>± Infinity</td>
</tr>
<tr>
<td>inexact results</td>
<td>rounded value per rounding mode</td>
</tr>
</tbody>
</table>
IEEE 754 Compiler-Aided FPE Handling

- **Compiler-Aided**: instruct compiler (Fortran, C/C++) to generate code that
  - monitors and
  - executes ("traps to") predetermined routines upon FPE

- This is the *recommended* exception handling mode
  - introduces minimal overhead in the computation
  - allow user controlled action in response to FP exceptions
  - valuable during code development / testing and when user believes that FPEs / round-off errors impact the accuracy of results

- Each platform has its own way to instruct the compiler to provide this
Power5+ Pipeline and Datapath

The Power5+ core has the following execution units:

- 2 integer fixed point units (FXU)
- 2 load/store units (LSU)
- 2 floating point units (FPU)
  - each capable of DP fused multiply-add

Power5+ cores on hydra operate at 1.9GHz maximum (ideal) giga-flops/sec / core

\[ 7.6 = 2 \times 2 \times 1.9 \text{ giga-flops/sec / core} \]

See [http://sc.tamu.edu/systems/hydra](http://sc.tamu.edu/systems/hydra) for complete technical coverage of hardware and software architecture of Power5+ and Hydra 1600 Cluster.
By default, Power4 and 5 execute in *pipelined-mode* and do not handle FPEs in user code (default 754 handling).

Allow automatic FPE tracking and handling with compiler options `-qflttrap` and `-qsigtrap`.

`-flttrap`: compiler produces code that generates a TRAP signal (*SIGTRAP*) to flag the occurrence of any *enabled* floating-point exception “after” each FP operation.

<table>
<thead>
<tr>
<th>Compiler Option</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-qflttrap</code></td>
<td>specifies which FPE(s) to enable</td>
</tr>
<tr>
<td><code>-qsigtrap</code></td>
<td>specifies which user or system routine to call to handle enabled FPE(s)</td>
</tr>
</tbody>
</table>
Each of the five exception types is controlled by a separate sub-option
If 'enable' is not used NO trapping is done

\begin{verbatim}
xlf95 -qflttrap=overflow:underflow:enable mycode.f
xlf95 -qflttrap mycode.f
\end{verbatim}

\begin{tabular}{ll}
-qflttrap & value \hline
ENable & Detect and Trap enable trapping of listed exception(s) 
OVerflow & overflow 
UNderflow & underflow 
ZEROdivide & division by zero 
INValid & floating-point invalid operations 
INEXact & inexact results; (do NOT enable this, as it is a common condition) 
IMPrecise & check for exception(s) on entry/exit to subprograms only NOT after each FP op 
\textit{no value} & exact location in the code 
\end{tabular}
Option '-qsigtrap' specifies *how to handle* the SIGTRAP

```bash
xlf95 -qflttrap -qsigtrap mycode.f
xlf95 -qflttrap -qsigtrap=sigtrap_handler mycode.f
```

<table>
<thead>
<tr>
<th><code>-qsigtrap_value</code></th>
<th><strong>Action</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><em>no -qsigtrap</em></td>
<td>dump core on an exception</td>
</tr>
<tr>
<td><em>no value</em></td>
<td>like <code>xl__trce</code></td>
</tr>
<tr>
<td><em>xl__ieee</em></td>
<td>produces traceback and explanation, sets IEEE default value and continues</td>
</tr>
<tr>
<td><em>xl__trce</em></td>
<td>produces traceback and stops program</td>
</tr>
<tr>
<td><em>xl__trcedump</em></td>
<td>produces traceback, core and stops program</td>
</tr>
<tr>
<td><em>my_handler</em></td>
<td>supply own SIGTRAP handler</td>
</tr>
</tbody>
</table>

**Other handlers**

- `xl__sigdump` produces traceback, explanation for signal; called from within user handler and does not stop program
- `xl__trbk` produces traceback; called as a regular routine from program
On Power5+, when exception trapping is enabled, exceptions produce the following results:

- Note: different results are generated when exception trapping is enabled or not

### FP Exception

<table>
<thead>
<tr>
<th>Exception</th>
<th>Default IEEE 754 Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>overflow</td>
<td>± Infinity</td>
</tr>
<tr>
<td>underflow</td>
<td>FP denormal with biased exponent</td>
</tr>
<tr>
<td>invalid operation</td>
<td>No result</td>
</tr>
<tr>
<td>division by zero</td>
<td>No result</td>
</tr>
<tr>
<td>inexact results</td>
<td>rounded value per rounding mode</td>
</tr>
</tbody>
</table>
One can also install alternative exception handler than one supplied by XL Fortran
– one you have written yourself, by calling the SIGNAL subroutine (defined in /usr/include/fexcp.h):

```
INCLUDE 'fexcp.h'
CALL SIGNAL(SIGTRAP, handler_name)
CALL SIGNAL(SIGFPE, handler_name)
```

A C/C++ user can install its own SIGTRAP handler using the `signal()` system call
- **-qfloat=\texttt{sub0ption\_list}**
  - Determines how the compiler generates or optimizes code to handle particular types of floating-point calculations so that accuracy or speed is improved.
  - Defaults: nofltint, fold, nohsflt, nohssngl, nonans, norndsnegl, maf, norrm, norsqrt, and nostricntnmaf; some of these settings are different with \texttt{-O3} optimization turned on or with \texttt{-qarch=ppc}.

- **-qieeef= \{ near | minus | plus | zero \}**
  - Specifies the rounding mode for the compiler to use when evaluating constant floating-point expressions at compile time.
  - Default -qieeef=near
Power/AIX 5L FP-related Compiler Flags-2

- `-qfold`
  - constant floating-point expressions are to be evaluated at compile time
- `-qlodb1128 -qlongdouble` 128 bit long doubles
- `-qlargepage` use 16MB VM pages
- `-qlonglong` allow 64-bit integers
- `-qstrict` switch off optimizations that may alter the FP semantics of user code
Users can *manually* track and handle FPEs from within their code, using their own trap handler to **SIGFPE**

Use **fp_trap** subroutine (in **libc.a**) which
- Queries / changes the mode of the user process to *allow floating-point exceptions* to generate traps

```c
#include <fptrap.h>
extern int fp_trap(int flag);
```

<table>
<thead>
<tr>
<th><strong>flag</strong> value</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP_TRAP_SYNC</td>
<td>put process in <em>precise</em> trapping mode</td>
</tr>
<tr>
<td>FP_TRAP_OFF</td>
<td>set trap off mode</td>
</tr>
<tr>
<td>FP_TRAP_QUERY</td>
<td>return current process mode</td>
</tr>
<tr>
<td>FP_TRAP_IMP</td>
<td>put process in <em>non-recoverable imprecise</em> trapping mode</td>
</tr>
<tr>
<td>FP_TRAP_IMP_REC</td>
<td>put process in <em>recoverable imprecise</em> trapping mode</td>
</tr>
<tr>
<td>FP_TRAP_FASTMODE</td>
<td>put process in fastest trapping mode</td>
</tr>
</tbody>
</table>
```

available on the platform
Use the `fp_trap` subroutine to enable FPE tracking

Use the `fp_xx_enable()` and `fp_xx_disable()` (in libc.a) to specify which FPE(s) to enable/disable

```c
#include <fptrap.h>
extern int fp_enable_any();
extern int fp_enable( fptrap_t mask );
extern int fp_enable_all();
extern int fp_is_enabled( fptrap_t mask );
extern int fp_disable_all();
extern int fp_disable( fptrap_t mask );
```

<table>
<thead>
<tr>
<th>value in <code>mask</code></th>
<th>FPE Requested</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRP_INVALID</td>
<td>Invalid operation</td>
</tr>
<tr>
<td>TRP_DIV_BY_ZERO</td>
<td>Division by Zero</td>
</tr>
<tr>
<td>TRP_UNDERFLOW</td>
<td>Underflow</td>
</tr>
<tr>
<td>TRP_OVERFLOW</td>
<td>Overflow</td>
</tr>
<tr>
<td>TRP_INEXACT</td>
<td>Inexact Result</td>
</tr>
</tbody>
</table>
**fp_trap** uses system hardware to detect FPE and generates SIGFPE signals when exceptions occur

- Fortran definitions for the values needed to call it are in `/usr/include/fp_fort_c.f, fp_fort_t.f`, or the `xlf_fp_util` module.

**Advantages**

- works for any code, regardless of the language and without the need to compile with any special options.
- generates SIGFPE signals, the same as other popular Unix systems.

**Disadvantages**

- Program may run **much slower** while exception checking is turned on
- call to FP_TRAP requires source-code change and recompilation
• Use `xlf_fp_util` module routines to control FP h/w
  – allow to query and set floating-point status and control register `pfscr`
  – add `USE XLF_FP_UTIL` in your fortran code;
• See also `fpgets` and `fpsets` routines
Intel Nehalem – 64-bit CISC

- 64-bit CISC processor implementation of Intel64 ISA
  - Out-of-order execution engine with multiple execution units accessible via 6 ports (see next slide)
    - Port 0
      - Integer ALU and Shift Units
      - Integer SIMD ALU and SIMD shuffle
      - Single precision FP MUL, double precision FP MUL, FP MUL (x87), FP/SIMD/SSE2 Move and Logic and FP Shuffle, DIV/SQRT
    - Port 1
      - Integer ALU, integer LEA and integer MUL
      - Integer SIMD MUL, integer SIMD shift, PSAD and string compare
      - FP ADD
    - Port 2
      - Integer loads
    - Port 3
      - Store address
    - Port 4
      - Store data
    - Port 5
      - Integer ALU and Shift Units, jump
      - Integer SIMD ALU and SIMD shuffle
      - FP/SIMD/SSE2 Move and Logic
  - See http://sc.tamu.edu/systems/eos for complete technical coverage of hardware and software architecture of Nehalem and EOS iDataPlex Cluster
Nehalem Execution Engine
Out-of-order Pipelines

Retirement Register File
(Architected State)

Reorder-Buffer
(ROB) 128 entries

Unified Reservation Stations (URS) 36 entries

IDQ

Register Alias Table and Allocator

4 µ ops

4 µ ops

1 Reg File WB /cycle /port

Nehalem RISC micro-operations

Nehalem RISC micro-operations

micro-op issue

out-of-order dispatch

out-of-order dispatch

and execution

 Integer ALU & Shift

 Integer ALU & LEA

 FP Multiply

 FP Add

 Divide

 Complex Integer

 SSE Integer ALU

 Integer Shuffles

 SSE Integer Multiply

 Load

 Store Address

 Store Data

 Integer ALU & Shift

 Branch

 FP Shuffle

 SSE Integer ALU

 Integer Shuffles

 Memory Order-Buffer
 (MOB)

 16B load /cycle

 128 bits

 128 bits

 16B store /cycle

32 kIB L1 Data Cache

Intel Nehalem – 64-bit CISC

Texas A&M University Michael E. Thomadakis

IEEE Floating-Point Arithmetic July 2010
Nehalem – SIMD Support

- Nehalem supports in h/w the SIMD instructions up to SSE 4.2 using specialized h/w in the back-end execution engine
  - two “4-wide” ALUs for SIMD computation
  - use 16 128-bit XMM and 8 64-bit MMX registers for SIMD operands
Nehalem – SIMD Support

- **SIMD Operations**: two ports can dispatch SIMD FP
  - 4 SP FPs per port, or 2 DP FPs per port
  - Nehalem cores on ΕΟΣ operate at 1.9GHz, ideal DP giga-flops/sec / core
    \[11.2 = 2 \times 2 \times 2.8 \text{ giga-flops/sec / core}\]

1 128 bit XMM data register ≡ 2 packed DP FP quantities

1 128 bit XMM data register ≡ 4 packed SP FP quantities

2 64-bit **Double**-Precision SIMD FP operations with XMM data registers

\[\leftrightarrow\]

2 DP FP ops / cycle / port

4 32-bit **Single**-Precision SIMD FP operations with XMM data registers

\[\leftrightarrow\]

4 SP FP ops / cycle / port
Intel Nehalem FP Exceptions (intel64)

- Nehalem Floating-Point Exceptions
  1. IEEE standard exception: invalid operation exception for invalid arithmetic operands and unsupported formats (#IA)
     - Signaling NaN
     - Infinity – Infinity
     - Infinity ÷ Infinity
     - Zero ÷ Zero
     - Infinity × Zero
     - Invalid Compare
     - Invalid Square Root
     - Invalid Integer Conversion
  2. Zero Divide Exception (#Z)
  3. Numeric Overflow Exception (#O)
  4. Underflow Exception (#U)
  5. Inexact Exception (#P)
Exceptions are masked by default
- unmask first 3 by compiling main with \texttt{-fpe0} (Fortran), which also sets \texttt{-ftz}
- unmask for individual source files with \texttt{-fpe-all=0}
- C/C++ switches in next compiler version
  - For now, use intrinsics, \textit{e.g.} (see \texttt{xmmmintrin.h})
    \texttt{_MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON);}
Intel Compiler-Aided FP Handling - 1

- See `icc(1)` (C/C++) and `ifort(1)` (Fortran)
  - `-fpe n`: specifies floating-point exception handling for the main program at run-time. This includes whether exceptional floating-point values are allowed and how precisely run-time exceptions are reported
  - `n=3`: default FP invalid, divide-by-zero, and overflow exceptions produce exceptional values; FP underflow is gradual and execution continues; standard IEEE support
  - `n=0`: Enables the overflow, the divide-by-zero, and the invalid floating-point exceptions. The program will print an error message and abort if any of these exceptions occurs. FP underflow goes to zero and execution continues (`flush-to-zero`).
  - `n=1`: on FP underflow flush it to zero; otherwise same as in `n=0`
Intel Compiler-Aided FP Handling - 2

- **-fp-model model**: controls the semantics of floating-point calculations
  - **model**: precise, fast, strict, source, double, extended, except
    - **precise**: strictly adhere to value-safe optimizations when implementing floating-point calculations. It disables optimizations that can change the result of floating-point calculations, which is required for strict ANSI conformance.
    - **fast[=1 | 2]**: compilers use more (2) or less (1) aggressive FP optimizations which increase speed but may impact accuracy. FPE handling is disabled.
    - **strict**: strictly adhere to value safe optimizations; this is the strictest FP model
    - **source double extended**: intermediate results are use the specified precision and rounding (of the source, or double or extended 80-bit x87 precisions)
    - **except**: use FP exception semantics
Gradual underflow to 0 can degrade performance. You can improve performance by using higher optimization levels to get the default abrupt underflow or explicitly setting `-ftz`

- `-ftz` : results in abrupt underflow to 0 and execution continues; this avoids using denormal values in computations and no floating exception occurs; default unless `-O0` is used
  - `-no-ftz` : results in gradual underflow to 0; the result of a floating underflow is a denormalized number or a zero
  - at optimization level `-O2` or `-O3`, underflow flushes to zero by default
Other less influential options

- `-fp-port` : it rounds results after each operation which takes place at conversions and assignments; default is to keep intermediate result at full precision
- `-fp-relaxed` : it enables use of faster but slightly less accurate math functions (/ or sqrt())
- `-speculation=model` : enables FP speculation; `model`
  - `fast` : the same FP speculation as when optimizations are enabled
  - `safe` : the default with `-O0`
Other less influential options

- `-mp1` : improves floating-point precision and consistency and has less impact on performance by disabling fewer optimizations than the `-fp-model precise`
- `-[no-]prec-div` : improves precision of floating-point division when with certain optimizations the compiler transforms FP div into multiplication ($X/f = X \times 1/f$); this may not be as accurate as IEEE division;
- `-[no-]prec-sqrt` : improves precision of floating-point square-root prohibiting certain optimizations which produce sqrt results with a less precise algorithm
- `-[no-]fast-transcendentals` : allows compiler to replace calls to transcendental functions with functions with less precise but faster algorithms
SVID3 User Coded IEEE 754 FPE Handling

- SVID3 UNIX FPE handling in user code
- Platform Independent
  - user supplies `matherr()` routine to handle FPEs, otherwise default action takes place
  - `#include <math.h>
    int matherr (struct exception *x);
    struct exception {
      int type;
      char name;
      double arg1, arg2, retval; /*... */
    } *x;

- `type`
  - DOMAIN : Argument domain error
  - SING : Argument singularity
  - OVER/UNDERFLOW : Overflow/underflow range error
  - TLOSS : Total loss of significance
  - PLOSS : Partial loss of significance
SVID3 User Coded IEEE 754 FPE Handling

- **name**: string with the routine in error
- **arg1**: pointer to 1\textsuperscript{st} argument
- **arg2**: pointer to 2\textsuperscript{nd} argument
- **retval**: specifies the default value that is returned by the routine unless the user's matherr function sets it to a different value.

**Return Values**

- If the user's matherr function returns a non-zero value, no error message is printed, and the \texttt{errno} global variable will not be set.
- If \texttt{matherr()} is not supplied, \texttt{errno} is set to EDOM or ERANGE and the program continues.
References


Explanation on the SC Logo

Cluster 1600
Power5+ AIX
5.3
pSeries 690
AIX 5.2

Altix 3700
Linux PP4.0

Origin
IRIX 6.5