

**ESA/390 Enhanced Floating Point Support: An
Overview**

**SHARE 93, Summer Meeting 1999, Session
8163**

John R. Ehrman

IBM Santa Teresa Laboratory
IBM Software Systems Division
555 Bailey Avenue/F4
San Jose, California 95141
Internet: ehrman@vnet.ibm.com

August, 1999

Table of Contents

Overview	2
Floating Point Data: Representations and Behavior	3
The Reality of Realistic Data	4
Computing With Numeric Data	5
Basic Floating Point Representations	7
Terminology	10
Machine Representations of Floating Point Numbers	11
A Sampling of Floating Point Representations	12
Real Numbers vs. Floating-Point Numbers	13
Consequences of Finite Precision	14
Precision Properties of Floating-Point Numbers	15
Rounding	16
Conversions To and From Decimal	17
How Many “Significant Digits”?	19
Consequences of Finite Range	21
Value Ranges	22
Exponent Range: Representable and Computable Values	23
Overflow and Underflow	24
Handling Overflow and Underflow	25
Meaningless Computations	26
Real-Mathematics Fictions, Floating-Point Facts	27
The Laws of “Real” Arithmetic	28

Table of Contents

Real vs. Realistic (Floating Point) Arithmetic	30
Floating Point Arithmetic Considerations	31
The Facts of Floating Point Arithmetic	32
The Facts of Floating Point Arithmetic: Consequences	37
The Facts of Floating Point Arithmetic: Inequalities	39
Floating Point Peculiarities and Pathologies	40
Why Does Floating Point Work At All?	42
Some General Conclusions	43
A Review of IBM Hex Floating Point	44
IBM Hex Floating Point	45
IEEE Floating Point Standard	48
IEEE Floating Point Standard: Representations	49
IEEE Floating Point Standard: Normal Values	50
IEEE Floating Point Standard: Special Values	52
IEEE Floating Point: Full Range of the Representation	55
IEEE Floating Point Standard: Required Capabilities	56
IEEE Floating Point Standard: Further Details	61
IEEE Floating Point Standard: Recommended Functions	62
IEEE Floating Point Standard: Justifications	63
ESA/390 Floating Point Enhancements	64
ESA/390 Floating Point Enhancements: Overview	65

Table of Contents

ESA/390 Hardware Enhancements	66
ESA/390 Floating Point Enhancements: Processor Features	67
ESA/390 Floating Point Enhancements: BFP Data Types	68
ESA/390 Floating Point Control Register	70
ESA/390 Floating Point Enhancements: Rounding Modes	73
ESA/390 HFP Enhancements	74
ESA/390 Floating Point Support Instructions	75
ESA/390 Floating Point Enhancements: BFP Instructions	77
ESA/390 Floating Point Enhancements: Control Instructions	86
ESA/390 Software Enhancements	87
OS/390 Operating System Support	88
ESA/390 Floating Point: HLASM Support	90
ESA/390 Floating Point: HLASM Toolkit Feature Support	94
C/C++ Compiler Support	95
C/C++ and LE Runtime Library Support	97
Java Language Definition	99
Java Support: Implementations	100
Other Products' Support	101
ESA/390 Floating Point Software Conventions	102
Contrasting Binary and Hexadecimal Floating Point on ESA/390	105
Binary and Hex Floating Point: General Differences	106
Binary and Hex Floating Point: Representation Differences	108
Binary and Hex Floating Point: Computational Differences	109
Binary vs. Hex Floating Point: 4-Byte (Short) Precision	110

Table of Contents

Binary vs. Hex Floating Point: 8-Byte (Long) Precision	111
Binary vs. Hex Floating Point: 16-Byte (Extended) Precision	112
IEEE vs. IBM Hex Floating Point: Exception Differences	113
Other IEEE Implementations	114
General Observations	115
Intel	116
RISC System/6000	118
Binary and Hexadecimal: Differences to Watch Out For	120
Standard Conformance and Language Issues	121
Coding Implications	122
Compiling and Optimization Implications	123
Library/Run-Time Issues	125
Summary	126
References	128

Trademarks

The following are trademarks or registered trademarks of the International Business Machines Corporation or other companies:

- IBM
- System/360
- System/370
- System/390
- S/360
- S/370
- S/390
- ESA/390
- OS/390
- DEC
- VAX
- Alpha
- HP
- PA-RISC
- SUN
- SPARC

Overview

- Why floating point?
- About floating point: precision, range, and the like
 - How floating point arithmetic differs from “real” arithmetic
- IBM hexadecimal floating point
- The IEEE Floating Point Standard
- ESA/390 and OS/390 floating point support
 - ESA/390 hardware enhancements
 - OS/390 software enhancements
- Contrasting binary and hexadecimal floating point on ESA/390
- Usage and migration considerations
 - Other IEEE floating point implementations
- Summary

**Floating Point Data:
Representations and Behavior**

The Reality of Realistic Data

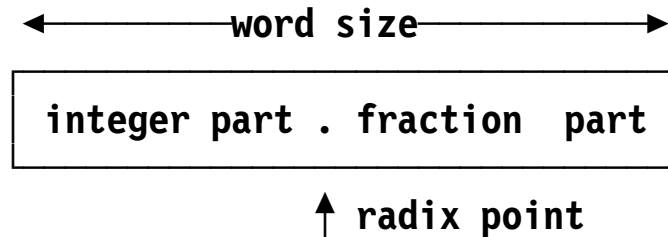
- Numeric data comes in all sizes:

	6023 00000 00000 00000 00000.	Avogadro's Number
	9 46000 00000 00000.	meters/light-year
	2997 76000.	meters/sec (light)
	33136.	cm/sec (sound)
	745.7	Watts/horsepower
	440.	hertz (Concert A)
	16.3872	cc/cubic inch
	3.14159 26535 89793	π
kilometers/mile	1.60935	
$\ln 2$.69314 71805 59945	
Coulombs/electron	.00000 00000 00000 00016	
Planck's const (erg-sec)	.00000 00000 00000 00000 00000 06624	
grams/electron	.00000 00000 00000 00000 00000 00910 7	

- Programs must manage these widely varying magnitudes

Computing With Numeric Data

- Early computers provided only integer arithmetic
 - “Reals” were approximated by scaled fixed-point numbers



- Programmers (or compilers) keep track of the radix point
 - Fixed point requires “shifts” for re-scaling after some operations
 - Still used for decimal arithmetic (e.g., COBOL, PL/I)
- Problems: handling full range of values; maximizing precision
 - Magnitude overflow loses the most significant digits
 - Truncation of low-order (least significant) digits loses precision
 - Full field width (to handle full range of values) cannot retain full precision for normal (non-extreme) values

Computing With Numeric Data ...

- Solution: floating point!

“For each arithmetic operation we ask the computer to present us with the first few significant digits and the count telling us where the decimal point lies. These operations are referred to as floating point arithmetic.” (Sterbenz, p.4)

- Increased, more uniform precision
 - Precision losses involve the least significant digits
 - That's the way most of us do arithmetic...
- Manages scaling automatically
 - Hardware keeps track of the radix point for you
- Much less likelihood of “magnitude overflow”

Basic Floating Point Representations

- Floating Point numbers are a subset of the **rational** numbers
- A value X is represented by $\pm M \times r^k$
 - M** an integer satisfying $0 \leq M < r^p$
 - r** **radix** (or **base**) of the representation
 - p** **precision** (the number of base- r digits)
 - k** **exponent**
- The set of all such values is a **floating point system** $FP(r,p)$
- Examples using $FP(10,4)$:

0 1 2 3.

$k = 0$
 $X = 123$

2 3 4 0.

$k = -1$
 $X = 234$

0 0 7 3.

$k = 0$
 $X = 73$

0 0 7 3.

$k = 2$
 $X = 7300$

0 0 7 3.

$k = -2$
 $X = 0.73$

Basic Floating Point Representations ...

- If a number is not exactly representable, it must be approximated by one of the nearest representable values
- Approximations using FP(10,4):

Rounded Down

3 3 3 3.

$$k = -4$$

$$X = 1/3$$

Rounded Down

6 6 6 6.

$$k = -4$$

$$X = 2/3$$

Rounded Up

6 6 6 7.

$$k = -4$$

$$X = 2/3$$

- Examples of base-16 and base-2 representations of 0.1 (base 10):
 - in FP(16,6): $1677722/16^6 = 1677722/16777216 = 0.10000002384$
 - in FP(2,24): $13421773/2^{27} = 13421773/134217728 = 0.10000000149$
- Decimal fractions aren't generally precisely representable in base 2

Basic Floating Point Representations ...

- Common practice: use a fraction $f = M \div r^p$ satisfying $0 \leq f < 1.0$
 - Puts the radix point at the left end of the digits
- X is then represented by $\pm f \times r^e$
 - $e = k + p$
 - If f is normalized, $r^{-1} \leq f < 1.0$ (most significant digit $\neq 0$)
- Examples of normalized and unnormalized representations:

normalized

.1 2 3 0

$k = 3$

$X = 123$

unnormalized

.0 1 2 3

$k = 4$

$X = 123$

normalized

.7 3 0 0

$k = 2$

$X = 73$

unnormalized

.0 7 3 0

$k = 3$

$X = 73$

unnormalized

.0 0 7 3

$k = 4$

$X = 73$

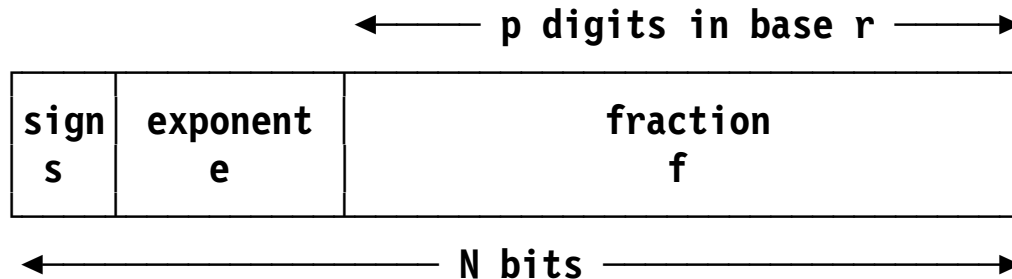
- Unnormalization allows redundant representations!
 - May also have many redundant representations of zero

Terminology

- A floating-point number (say, $123.45E-67$) is composed of two parts:
 1. the “number stuff” (the 123.45)
 2. the “exponent stuff” (the $E-67$)
- **Fraction:** some or all of the digits of the “number stuff”
 - DEC™ VAX™ and IEEE store “some” digits (all but one implicit bit)
 - IBM hex float stores “all” digits
 - Sometimes called “mantissa” (but that word comes from logarithms...)
- **Significand:** the “number stuff,” with all numerically significant digits exposed; what the machine actually calculates with
- **Exponent:** the “exponent stuff,” correctly signed
- **Characteristic, Biased Exponent:** exponent+bias
 - The bias is chosen so that valid values of the exponent cause the biased exponent to be always non-negative

Machine Representations of Floating Point Numbers

- Many ways to “package” the pieces into a “representation”
- Example of a typical format:



- Typically, fraction sign $s = 0$ means +, $s = 1$ means –
- e is almost always represented in binary
 - May be signed or unsigned
- f may be sign-magnitude, radix-complement, etc.
- Finiteness imposes limits on both p and the range of e
 - This finiteness has many interesting consequences...
- Designers must choose among base, precision, exponent width/range
 - Number of distinct values $\leq 2^N$ (and may be **much** $< 2^N$)

A Sampling of Floating Point Representations

Machines and Representations	Base (B)	Base-B Digits	Equiv. Decml.	Exp. Width	Format (Note)	Fraction Representation
Prime 550 series single	2	23	6	8	s/f/e	Two's-complement
IEEE Standard single	2	24	6	8	s/e/f	Sign-magnitude
DEC VAX F-format	2	24	6	8	s/e/f	Sign-magnitude
IBM System/370 short	16	6	6	7	s/e/f	Sign-magnitude
Burroughs 6700/7700	8	13	10	7	s/e/i	Sign-magnitude
Harris Series 500 double	2	38	11	8	s/f/e	Two's-complement
Prime 550 series double	2	47	13	16	s/f/e	Two's-complement
CDC 6600/CYBER 70	2	48	14	11	i/e/s	One's-complement
Cray-2 single	2	48	14	15	s/se/f	Sign-magnitude
DEC VAX G-format	2	52	15	11	s/e/f	Sign-magnitude
IEEE Standard double	2	53	15	11	s/e/f	Sign-magnitude
IBM System/370 long	16	14	15	7	s/e/f	Sign-magnitude
DEC VAX D-format	2	55	16	8	s/e/f	Sign-magnitude
Harris Series 500 quadruple	2	69	22	23	s/f/e	Two's-complement
Cray-2 double	2	96	28	15	s/se/f	Sign-magnitude
IBM System/370 extended	16	28	32	7	s/e/f	Sign-magnitude
DEC VAX H-format	2	112	33	15	s/e/f	Sign-magnitude

Note: s = sign, se = signed exponent, e = exponent, f = fraction, i = integer.

Real Numbers vs. Floating-Point Numbers

- “Real” numbers are mathematical abstractions: useful, but **unreal**
 - Abstract numbers have no inherent precision or size limitations
- **Real** computations are done with “realistic,” “actual” numbers
 - Always have finite precision and accuracy
 - Must discard “excess” digits somewhere
- You must think differently about floating point numbers and arithmetic:
 - Finite range and precision imply violation of mathematical “laws”
 - Floating point numbers don't satisfy all the mathematical properties of rationals
 - Nor those of mathematical “reals” (We'll see later how close they come...)

Consequences of Finite Precision

Precision Properties of Floating-Point Numbers

- Useful measure: the **ulp** (**u**nit in the **l**ast **p**lace)
 - $\text{ulp}(x) = \text{successor}(x) - x$ (“spacing” between neighbors)
 - $\text{relative ulp-error}(x) = \text{ulp}(x) / x$ (weight assigned to lowest-order bit of x)
- Adding 1 in the low-order digit position has an effect that varies between r^{-p} and $r^{-(p-1)}$
 - Compare $(0.999-0.998)$ to $(0.101-0.100)$: **relative ulp** varies from 10^{-3} to 10^{-2}
- Thus, $\text{relative-ulp}(x)$ varies by a factor r , as x varies by powers of r
 - This effect is known as “wobbling precision”
- Precision (relative) is **not** properly measured as r^{-p}
 - Magnitude of a normalized fraction varies from $1 - r^{-p}$ (≈ 1) to r^{-1} (i.e., by a factor of r)
 - Hence, relative precision is better approximated as $r^{-(p-1)}$
 - This affects how closely floating point arithmetic approximates “real” (i.e., unrealistic) arithmetic

Rounding

- Finite precision requires some treatment of “excess” digits
- Consider the value $2/3 = 0.666666\dots$ in $FP(10,4)$:
 1. Throw away the excess: result = 0.6666
 - Called “chopping” or “truncating” or “rounding toward zero”
 2. Round towards $+\infty$: $+2/3$ becomes 0.6667, $-2/3$ becomes -0.6666
 3. Round towards $-\infty$: $+2/3$ becomes 0.6666, $-2/3$ becomes -0.6667
 4. Round to nearest can be done two ways:
 - a. Add correctly signed $1/2$ ulp (“biased round to nearest”)
 - 0.12345 rounds to 0.1235
 - The mode used by DEC VAX arithmetic, IBM hex rounding instructions
 - b. For values exactly half-way between to representables, choose the result that ends in an even digit (“unbiased round to nearest”)
 - 0.12345 rounds to 0.1234
- Why is traditional rounding “biased”? In $FP(10,4)$, perform $X=(X-Y)+Y$ repeatedly, starting with $X=1.000$, $Y=-0.5555$:
 - Biased round yields: 1.001, 1.002, 1.003, ...
 - Unbiased round yields: 1.000, 1.000, 1.000, ...

Conversions To and From Decimal

- “**IN**” conversion: from decimal to machine radix
- “**OUT**” conversion: from machine radix to decimal
- Equivalent decimal digits: (an “**IN-OUT**” conversion definition):
 - In a floating point system $FP(r,p)$ with radix r and p base- r digits:
If (1) $10^J \neq r^K$ for any (J,K) , and
(2) All d -digit decimal floating-point numbers can be converted to a member of $FP(r,p)$, and when converted back to decimal will correctly recover the original decimal value,
Then $FP(r,p)$ can faithfully represent d -digit decimal numbers.
 - (This is the definition used for the table on slide 12)
 - Recoverability requires all conversions to be correctly rounded

Conversions To and From Decimal ...

Examples using S/390 short (6-digit) hex floating point:

- All 6-digit decimal floating point values can be converted to hex and back with complete recovery
- Even though $16^{-6} \approx 0.59 \times 10^{-7}$, not all 7-digit decimal floating point values can be recovered!
 - $16^{-5} \approx 0.95 \times 10^{-6}$ is a better “precision” estimate
- The six 7-digit decimal floating point values $0.625000\underline{x}E-1$ (where $\underline{x}=3 \dots 8$) are all represented by X'40100001'.
 - X'40100001' converted back to decimal gives $0.6250006E-1$

How Many “Significant Digits”?

- “IN-OUT” conversions: start with external decimal data, convert to internal format, and back to decimal

IN-OUT Conversions

Precision	Decimal Digits	Hex Digits	Binary Digits
short	6	6	21
long	15	14	51
extended	32 33	28	111

- Example: to recover external decimal values converted to **extended** internal format and back to decimal,
 - Extended hex float faithfully represents 32 decimal digits
 - Extended binary float faithfully represents 33 decimal digits
- FP(16,6) and FP(2,24) faithfully represent 6 decimal digits

How Many “Significant Digits”? ...

- “OUT-IN” conversions: start with internal machine data, convert to external decimal, and back to internal

OUT-IN Conversions

Precision	Binary Digits	Hex Digits	Decimal Digits
short	24	6	9
long	53	14	17 18
extended	113	28	35 36

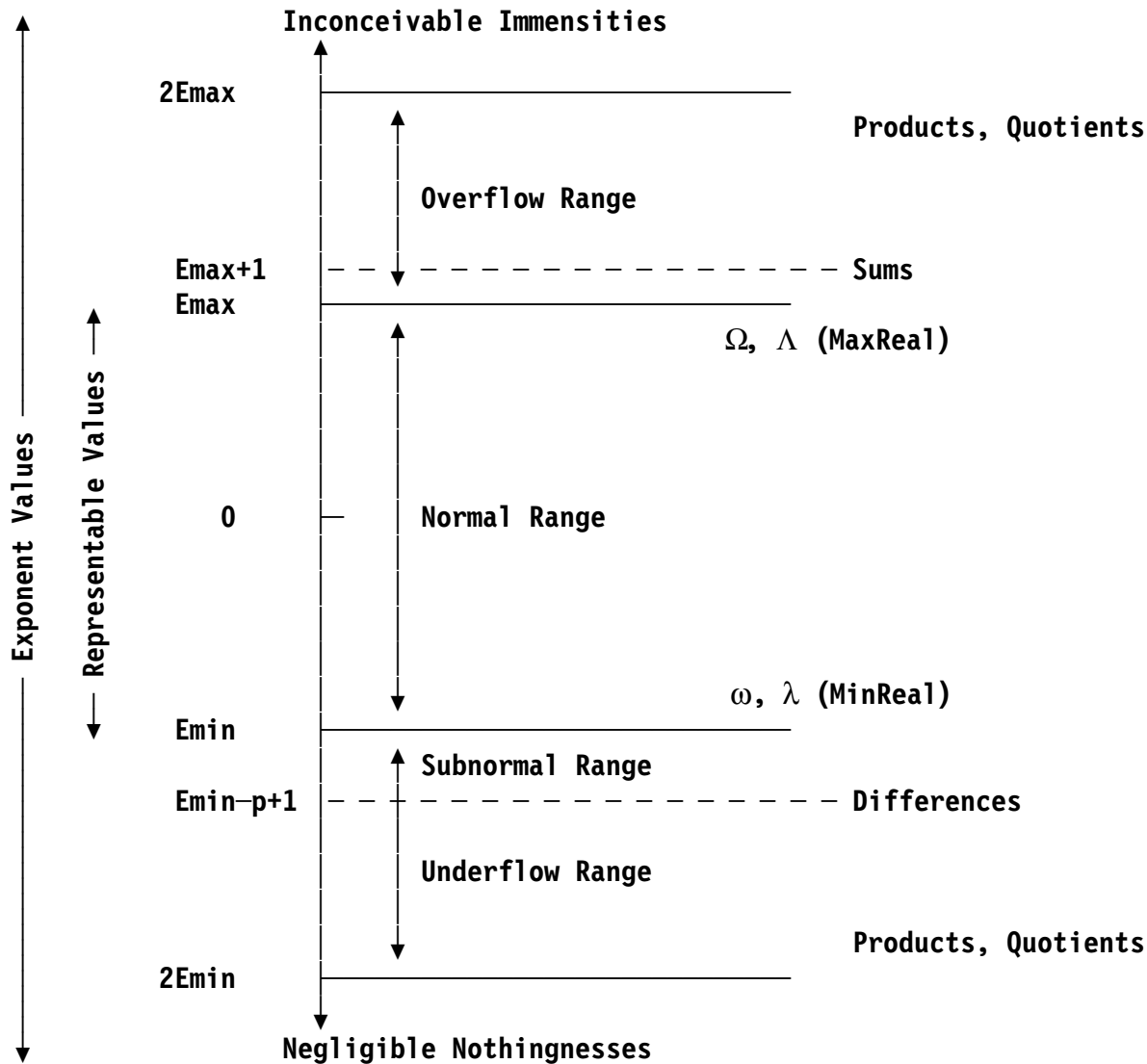
- Example: to recover **long** internal float values converted to decimal and back,
 - Long binary float requires 17 decimal digits for faithful representation
 - Long hex float requires 18 decimal digits for faithful representation
- “Rule of Thumb”: OUT-IN needs 3 more decimal digits than IN-OUT

Consequences of Finite Range

Value Ranges

- Let Ω (or Λ) designate the largest representable normalized floating point number (“MaxReal” or “Huge”)
 - Sometimes (carelessly) nicknamed “infinity”; but $\Omega \neq \infty$:
 - $\infty / 2 = \infty$, while $\Omega / 2$ is finite!
 - $\infty \times 0$ is meaningless, while $\Omega \times 0 = 0$!
 - S/390 hex: $\Omega = X'7FFF\dots FF' \approx 7.2 \times 10^{+75}$
- Let ω (or λ) designate the smallest representable normalized floating point number (“MinReal” or “Tiny”)
 - S/390 hex: $\omega = X'0010\dots 00' \approx 5.4 \times 10^{-79}$
- Many systems have a range asymmetry:
 - Some representable small numbers have reciprocals exceeding Ω
 - S/390 hex: no representable inverse for any number $\leq X'02100000'$
 - Some representable large numbers have reciprocals less than ω
 - IEEE binary: inverses of largest numbers underflow or are denormalized

Exponent Range: Representable and Computable Values



Overflow and Underflow

- Exponent range violations are called “overflow” and “underflow”
 - Exponent $> E_{\max}$ = overflow
 - Exponent $< E_{\min}$ = underflow
- Exponents of products and quotients of “normal” numbers can be nearly double the allowed exponent range
 - Possible problems with directed rounding (e.g. IEEE):
 - Numbers near Ω^2 (exponent near $2E_{\max}$) can be “rounded down” to Ω
 - Numbers near ω^2 (exponent near $2E_{\min}$) can be “rounded up” to ω
- Sums of “normal” numbers can overflow E_{\max} by 1
- Differences of “normal” numbers can underflow E_{\min} by (p-1)
 - May imply lost accuracy
 - Various ways to handle these “subnormal” numbers

Handling Overflow and Underflow

- Treatment (or mis-treatment) of range violations has pervasive impacts on programming
 - Operations on overflowed values are difficult to manage
 - The problems caused by range violations make ulps appear trifling...
- Range violations customarily handled by “fixups” of many types
 - Overflows may be set to Ω , ω , ∞ , or zero by various systems
 - Underflows may be set to zero
 - Overflows and underflows may return a finite value with modified characteristic (e.g. S/390 hex floating point interruptions “wrap” modulo 128)
 - etc.
- Fixups may be done by hardware, software, or both

Meaningless Computations

- Programs may encounter pathologies: overflows, underflows, division by zero, square root of a negative argument, etc.
- Some machines respond with zero, ∞ , Ω , others may provide explicit representations
- CDC: representations for ∞ , “Indefinite,” “Indeterminate” (“Not a Number”)
 - Arithmetic: NaN op any \rightarrow NaN; overflows $\rightarrow \infty$, underflows $\rightarrow 0$; $0 \times \infty \rightarrow$ NaN
- IBM Stretch: representations for ∞ , infinitesimal (ϵ), OMZ (“Order of Magnitude Zero” with significand = 0)
 - Arithmetic: $\infty + x \rightarrow \infty$; $\epsilon + x \rightarrow x$; etc.
 - System/360 significance exception has roots in OMZ
- DEC VAX: exponent = 0 and minus sign is a “reserved operand”
 - Causes an interruption when accessed

**Real-Mathematics Fictions,
Floating-Point Facts**

“Everything You Know Is Wrong”

The Laws of “Real” Arithmetic

- Closure** The product (ab) and sum $(a+b)$ of the real numbers a and b are reals
- Commutative** (1) $a+b = b+a$, and (2) $ab = ba$
- Associative** (1) $(a+b)+c = a+(b+c)$
(2) $(ab)c = a(bc)$
- Distributive** $a(b+c) = ab + ac$
- Units, Identities** There are reals 0 and 1 such that
(1) $a+0 = 0+a = a$
(2) $a \times 1 = 1 \times a = a$
- Inverses** For any real a
(1) There is a real $-a$ with $a+(-a) = (-a)+a = 0$
(2) If $a \neq 0$, there is a real a^{-1} with $a \times a^{-1} = a^{-1} \times a = 1$

The Laws of “Real” Arithmetic ...

- Consequences:

No Divisors of Zero

If $ab = 0$, then at least one of a or b must vanish

Cancellation

If $ab = ac$ and $a \neq 0$, then $b=c$.

Subtraction

If $(a-b)$ means $a+(-b)$, then $(a+b)-b = a$

Division

If $b \neq 0$, and a/b means ab^{-1} , then $b(a/b) = a$

- Inequalities

1. If $a < b$, then for all c , $a+c < b+c$

2. If $a < b$ and $c < d$, then $a + c < b + d$

3. If $b < c$, and a is positive, then $ab < ac$

Real vs. Realistic (Floating Point) Arithmetic

- The “laws of real arithmetic” hold only approximately (or not at all)
 - For most cases, it's pretty close
 - But: you should always know where the problems will occur
 - Small and large problems lurk everywhere
- Let $FP(r,p)$ be a floating point system with radix r and precision p :
 - **FP(r,p,Round)** means arithmetic in $FP(r,p)$ is calculated to full precision and then is rounded
 - **FP(r,p,Chop)** means arithmetic in $FP(r,p)$ is calculated to full precision and then is chopped
 - **FP(r,p,ChopG)** means arithmetic in $FP(r,p)$ is calculated to at least $(p+1)$ digits (at least one guard digit is used), and *then* is chopped (e.g., IBM Hex)
 - **FP(r,p,Chop0)** means arithmetic in $FP(r,p)$ is calculated with p digits (no guard digits are used), and then is chopped

Floating Point Arithmetic Considerations

- Full precision means “as though it's infinite precision” before rounding or chopping
- Full precision (rounded or chopped) arithmetic can be safely approximated with finite, fixed-length registers, plus
 - One “Guard Digit”
 - One “Rounding Digit” (if rounding is desired)
 - One “Sticky Digit” (for both rounding and chopping)
- Costs of “doing it right” are reasonable today
- Costs of “doing it fast” still involve trade-offs

The Facts of Floating Point Arithmetic

Closure

“The product (ab) and sum ($a+b$) of the floating-point numbers a and b ” :

Fact: they may not exist in $FP(r,p)$

Example: Overflow and underflow

Example: IBM Hex ($FP(16,6)$) contains 84357.0 and 0.84375, but not their sum, difference, quotient, or product.

Commutative

“(1) $a+b = b+a$, and (2) $ab = ba$ ”:

Fact: true for almost all floating point systems

Example: $ab \neq ba$ on the Cray-1

- The Cray-1 traded commutativity for speed

The Facts of Floating Point Arithmetic ...

Associative

“(1) $(a+b)+c = a+(b+c)$, and (2) $(ab)c = a(bc)$ ”:

Fact: both fail to hold

(1) If a , b , and c have the same sign, then the results differ by at most

- 1 ulp, in $FP(r,p,Chop)$ and $FP(r,p,ChopG)$
- r ulps, in $FP(r,p,Round)$

(2) The results differ by at most

- r ulps, in $FP(r,p,Chop)$ and $FP(r,p,ChopG)$

Distributive

“ $a(b+c) = ab + ac$ ”:

Fact: fails to hold

If b and c have the same sign, then the results differ by at most

- r ulps, in $FP(r,p,Chop)$ and $FP(r,p,ChopG)$

The Facts of Floating Point Arithmetic ...

Units

“There are floating-point numbers 0 and 1 such that

$$(1) a+0 = 0+a = a,$$

Fact: true for many systems (but underflow may intrude)

$$(2) a \times 1 = 1 \times a = a”:$$

Fact: true for almost all systems; fails in FP(r,p,Chop0)

Examples:

- The multiplicative identity ($1 \times a = a$) failed on the original IBM System/360 in long arithmetic (until corrected by IFPEC, the “Improved Floating Point Engineering Change”) due to the lack of a guard digit.
- No guard digit on Cray, CYBER, Univac

Fact: May not be true for special values (e.g. “Not a Number”)

The Facts of Floating Point Arithmetic ...

Inverses

“For any floating-point number a

(1) There is a real $-a$ with $a+(-a) = (-a)+a = 0$ ”:

Fact: true in all known systems

“(2) If $a \neq 0$, there is a floating-point number a^{-1} with $a \times a^{-1} = a^{-1} \times a = 1$ ”:

Fact: fails to hold in many systems.

- There may be no representable inverses

S/390 hexadecimal floating point has no representable inverse for any magnitude $\leq X'02100000'$ (16^{-63})

The Facts of Floating Point Arithmetic ...

Inverses (continued)

- Several a 's may have the same a^{-1} ($=1 \div a$)

Example: if $r > 2$, a number with fraction $(1 - r^{-p})$ has $r-1$ inverses in $FP(r,p)$ with chopped arithmetic, and approximately $(r-1)/2$ inverses with rounded arithmetic

Example: In S/390 hexadecimal floating point, $a^{-1} = b^{-1}$ means only that a and b may differ by 16 ulps.

- Some a^{-1} values give $a \times a^{-1} \neq 1$

Example: In S/390 hexadecimal floating point, numbers with fraction $(1 - n \times r^{-p})$ have inverse r^{-1} if $1 \leq n \leq 15$

- Multiplying any fraction by a power of the radix leaves the fraction unchanged, except in $FP(r,p,Chop0)$
 - This was a serious problem with the original System/360 long hex float

Example: In S/390 hexadecimal floating point, $a \times (1/a)$ may differ from 1 by 16 ulps.

The Facts of Floating Point Arithmetic: Consequences

No Divisors of Zero

“If $ab = 0$, then at least one of a , b must vanish”:

Fact: frequently fails if overflow or underflow occurs

Cancellation

“If $ab = ac$ and $a \neq 0$, then $b=c$ ”:

Fact: b and c can differ by at most

- $r-1$ ulps, in $FP(r,p,ChopG)$
- r ulps, in $FP(r,p,Round)$

The Facts of Floating Point Arithmetic: Consequences ...

Subtraction

“If $(a-b) = a + (-b)$, then $(a+b)-b = a$ ”:

Fact: frequently fails near underflow thresholds, or if $|b|$ greatly exceeds $|a|$

- If $(a+b)$ underflows to zero, then $(a+b)-b = -b$!
- If $|b|$ is much larger than $|a|$, then $(a+b)-b = 0$!

Subtractions can easily lose significant digits

Division

“If $b \neq 0$, and a/b means ab^{-1} , then $b(a/b) = a$ ”:

Fact: fails to hold

If $b(a/b) = c$, then $|a|$ and $|c|$ can differ by at most

- r ulps, in $FP(r,p,ChopG)$

The Facts of Floating Point Arithmetic: Inequalities

1. “If $a < b$, then for all c , $a+c < b+c$ ”:

Fact: $a+c \leq b+c$

2. “If $a < b$ and $c < d$, then $a + c < b + d$ ”:

Fact: $a + c \leq b + d$

3. “If $b < c$, and a is positive, then $ab < ac$ ”:

Fact: $ab \leq ac$

- Strict mathematical inequalities must be weakened to tolerate equality in floating point systems
- Inequalities do not persist across floating point operations!
 - The fact that *equalities* don't persist is better known
 - Some floating tests for equality are simply bitwise comparisons

Floating Point Peculiarities and Pathologies

The following oddities have happened on widely-used machines:

- Some numbers have no inverse
- a has an inverse, but $a \times a^{-1} \neq 1$
- $z = y$ but $z - t \neq y - t$
- $1/3 \neq 9/27$
- $y \times z \neq z \times y$;
- $z \neq 1 \times z \neq 0$;
- $x + y \neq y + x$
- $2 \times y \neq y + y$
- $1 \times y \neq y$
- $0.5 \times y \neq y / 2$

Floating Point Peculiarities and Pathologies ...

- $((y \times z)/y)/z < .00001$ (caused by overflow to Ω)
- $((y \times z)/y)/z > 100000.$ (caused by overflow to ω)
- $y > 1 > z > 0$ but $y/z = 0$ (caused by overflow to 0)
- $y/z < 0.99$ but $y-z = 0$ (caused by underflow to 0)
- $|z| < 1$, but the machine claims $|z| \geq 1$
- The expressions $x < y$ and $x - y < 0$ are exactly equivalent on some computers but not on others.
- Some machines have different over/underflow thresholds for multiplication and/or division than for addition and/or subtraction.
- Some machines treat all sufficiently tiny nonzero numbers z as if they were zero during multiplication and division, but not during addition or subtraction.

Why Does Floating Point Work At All?

1. It closely approximates the way most of us do calculations (most of the time)
2. It handles most of the “dirty work” automatically
3. “The secret of success of floating-point computation lies in the fact that we continue to do arithmetic to p digits of precision even though the accuracy of our intermediate results has degraded so that we can only guarantee that a few digits are significant.” (Sterbenz, p. 78)

Some General Conclusions

1. You ***must*** “think floating point”

- And not “think real”: real numbers are merely a pleasant abstraction
- Algebraic relations are not computational relations!

2. Small radix r means ...

- Relative error grows more slowly
- Post-normalization is needed more frequently
- More exponent bits needed for a given exponent range
 - Example: binary requires 2 more exponent bits than hex for same range

3. Rounding in preference to chopping means ...

- Magnitude of average error tends to be smaller
- Arithmetic operations may require extra cycles

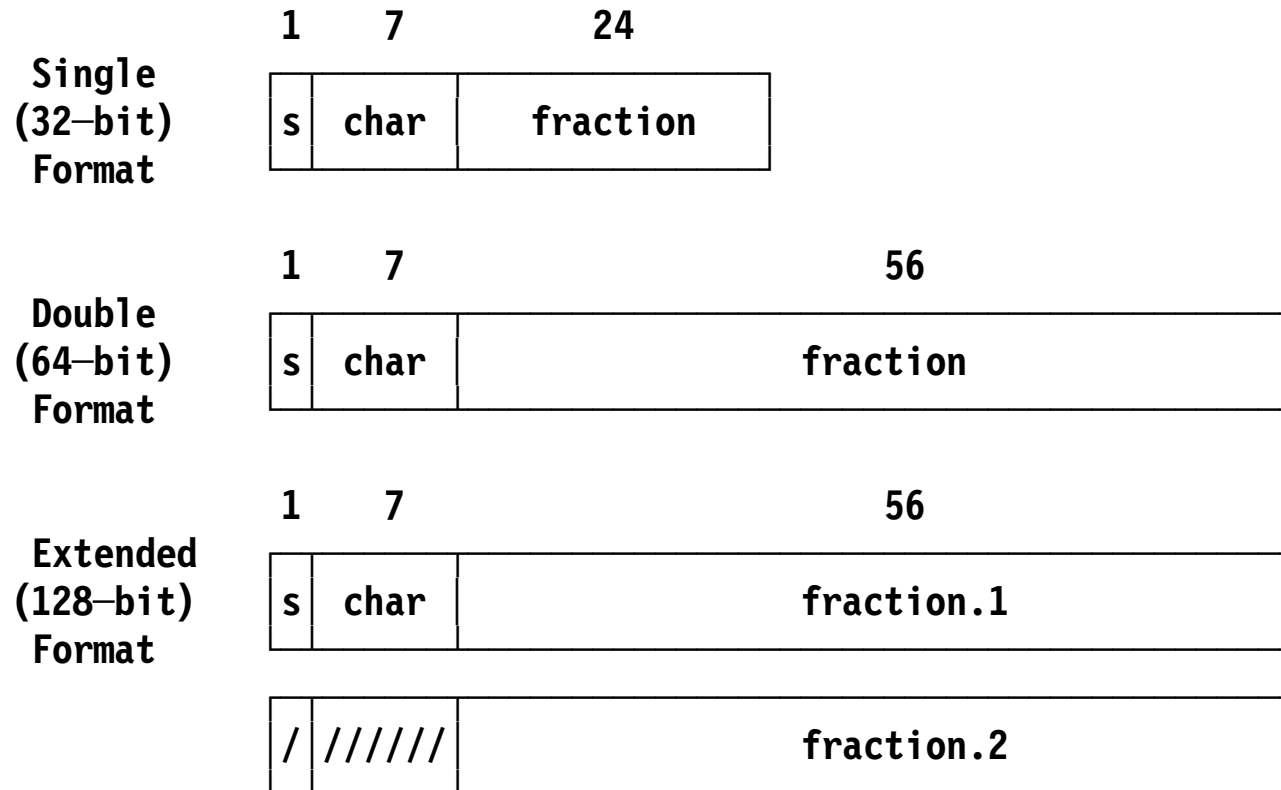
4. Always bear in mind ***Murphy's Law of Floating Point.***

Anything that can go wrong, does on some computer.

A Review of IBM Hex Floating Point

IBM Hex Floating Point

- Three radix-16 representations:
 1. Short precision: 6 hex digits (*not* 24 bits!)
 2. Long precision: 14 hex digits
 3. Extended precision: 28 hex digits (fraction is in two parts)



IBM Hex Floating Point ...

- All representations have same exponent range
 - 7-bit, base-16 exponent
 - Exponent values: -64 to $+63$
 - Magnitudes range from 5.4×10^{-79} to $7.2 \times 10^{+75}$
- Unnormalized values permitted throughout exponent range
 - Redundant representations for many values

Example: in short hex float, $1 = X'41100000'$, $X'42010000'$, $X'43001000'$, etc.

Example: $X'nn000000'$ (“pseudo-zeros”) treated as zero for all nn

 - True zero has characteristic 00 (exponent = -64)- Have effect only on addition and subtraction
- Multiplication and division are not affected, because the operands are pre-normalized internally (without causing exponent spills)

IBM Hex Floating Point ...

- Arithmetic truncates (usually, but not always, toward zero!)
 - Example: $1.0 - 16^{-7} = 1.0$, not $1.0 - 16^{-6}$ (result not rounded toward zero)
 - Example: $X'42100000' - X'40FFFFFF' = X'41F00001'$ (instead of $X'41F00000'$); result truncated away from zero, with error = 15/16 ulp.
- Square root is rounded
 - Rounding isn't biased (source and target operands have same lengths)
- No reserved or special values
 - Zero arithmetic results delivered as +0 (“true zero”)
- Rounding modes previously only via “ACRITH” feature on 4361
 - Now available on ESA/390

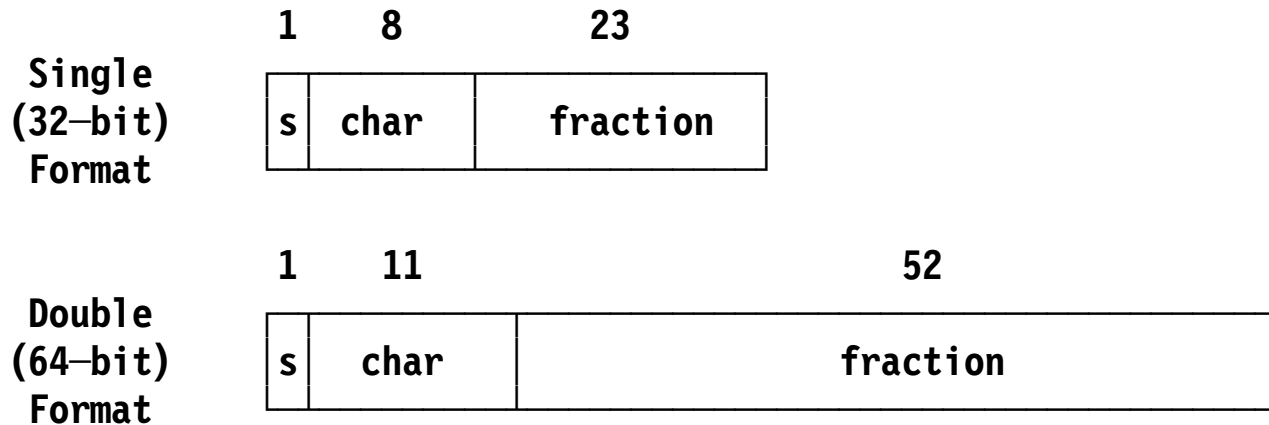
IEEE Floating Point Standard

ANSI / IEEE Standard 754-1985

“...meticulous attention to details that hardly matter to most people but matter, when they do matter, very much.” (Kahan)

IEEE Floating Point Standard: Representations

- No redundant representations
- Two fully-defined basic formats: single and double precision



s: sign bit (0 = +, 1 = -)

char: characteristic (“biased exponent”)
 $C_{\min} = 0$, $C_{\max} = \text{all 1-bits}$

fraction: has an extra *implied high-order 1-bit* for normal values;
significands therefore have 24-bit and 53-bit precisions

IEEE Floating Point Standard: Normal Values

length	precision	E_{\min}	E_{\max}	bias
single	24	-126	+127	+127
double	53	-1022	+1023	+1023

- Exponent range: E_{\min} to E_{\max}
- Characteristic **C** = exponent **e** + **bias**
 - Exponent's characteristic range: $C_{\min}+1$ to $C_{\max}-1$
 - C_{\min} and C_{\max} are reserved for special values (see slides 52-54)
- Significand = 1.fraction (leading 1-bit is implied)
 - Fraction is therefore always normalized
 - $1 \leq \text{significand} < 2$

IEEE Floating Point Standard: Normal Values ...

- Normal numbers
 - Single precision (short) value: $\pm (1.\text{fraction}) \times 2^{\text{char}-127}$
 - Magnitudes: 1.175×10^{-38} to $3.403 \times 10^{+38}$
 - Double precision (long) value: $\pm (1.\text{fraction}) \times 2^{\text{char}-1023}$
 - Magnitudes: 2.2231×10^{-308} to $1.7977 \times 10^{+308}$
- Zero
 - Characteristic = C_{\min} (zero); fraction = 0
 - Signed
- Examples of single precision values:

1.0	=	B' 0	01111111	000.....000	'	=	X'3F800000'
15.0	=	B' 0	10000010	1110000...000	'	=	X'41700000'
0.1	=	B' 0	01111011	10011001...011	'	=	X'3DCCCCCD'
Max	=	B' 0	11111110	111.....111	'	=	X'7F7FFFFFFF'
Min	=	B' 0	00000001	000.....000	'	=	X'00800000'
-0.0	=	B' 1	00000000	000.....000	'	=	X'80000000'
		s	←char→	←fraction→			

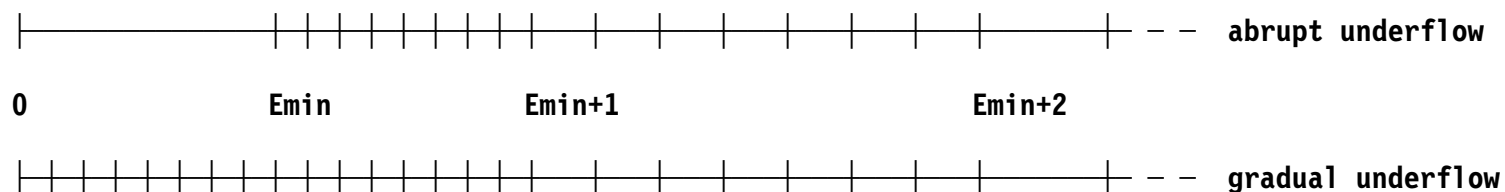
IEEE Floating Point Standard: Special Values

- Designated by reserved characteristics C_{\min} (zero), C_{\max} (all 1-bits)
- Denormalized numbers
 - Characteristic = C_{\min} (zero)
 - Fraction $\neq 0$ (no implied “1.” before the fraction)
 - Value: $\pm (0.\text{fraction}) \times 2^{E_{\min}}$
 - Examples of single precision values:

1.0E-40	=	B' 0 00000000	00...011000010	'	=	X'000116C2'
Largest Denorm	=	B' 0 00000000	111.....111	'	=	X'007FFFFFF'
Smallest Denorm	=	B' 0 00000000	000.....001	'	=	X'00000001'
		s	←char→			←fraction→

IEEE Floating Point Standard: Special Values ...

- Denormalized numbers allow “gradual underflow”
 - Most helpful for addition and subtraction
 - Can avoid many problems caused by “abrupt” underflows to zero
 - $(Y-X)+X$ gives X if underflow of $(Y-X)$ causes abrupt “flush to zero”
 - $(Y-X)+X$ gives Y with denormalization (gradual underflow) of $(Y-X)$
- Abrupt underflow has no normal numbers smaller than E_{\min}

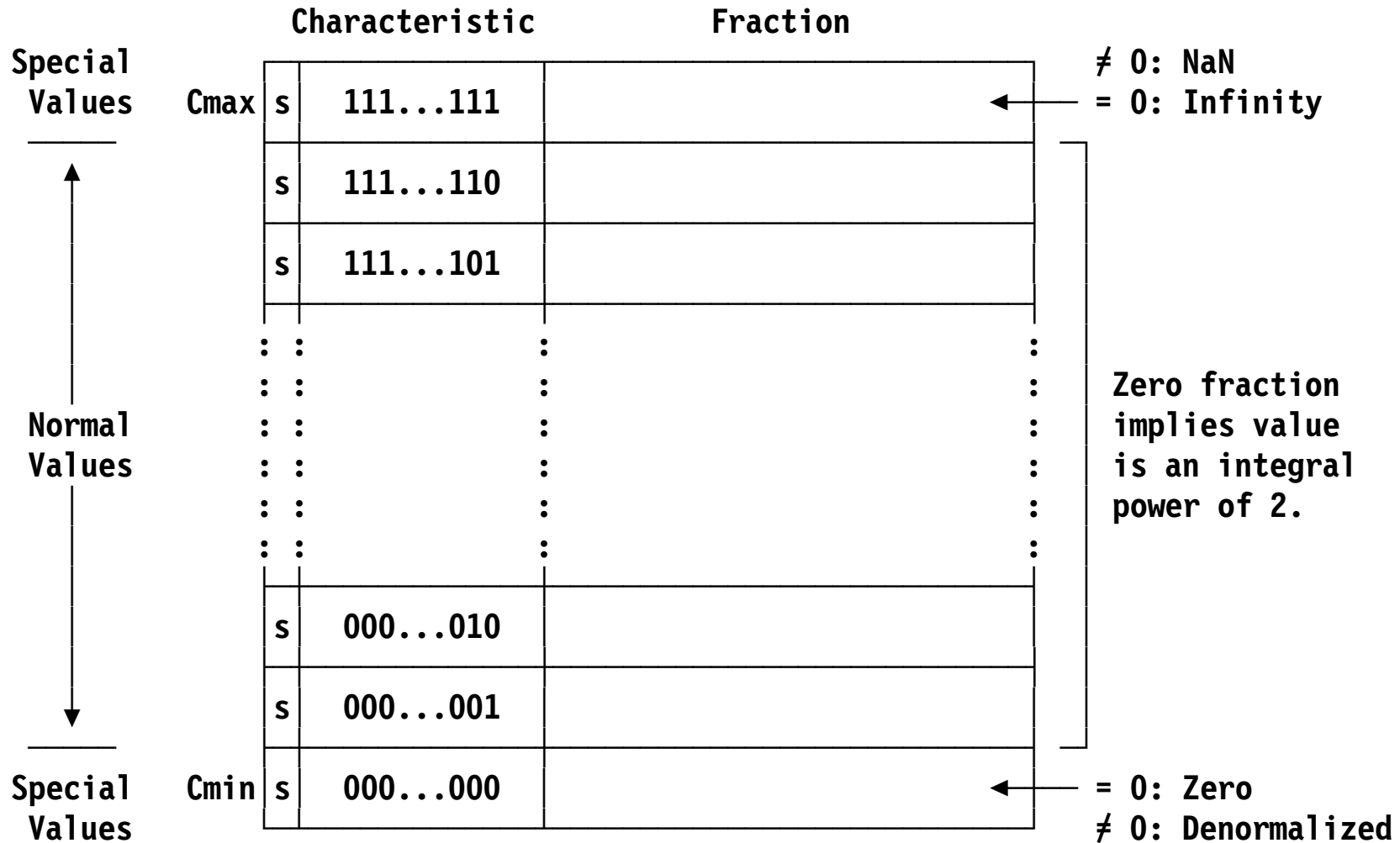


IEEE Floating Point Standard: Special Values ...

- Infinity
 - Characteristic = C_{\max} (all 1-bits); fraction = 0
 - Signed
- Not-a-Number (NaN)
 - Characteristic = C_{\max} (all 1-bits); fraction \neq 0
 - Fraction value may be used by implementation to indicate specific conditions
 - Two types are required: “quiet” (QNaN) and “signaling” (SNaN)
 - Standard doesn't define distinct representations
 - Sign is ignored (“not interpreted”) in determining it's a NaN
- Examples of single precision values:

```
–Infinity = B' 1 11111111 0000....0000 ' = X'FF800000'  
a Quiet NaN = B' 0 11111111 1100....0000 ' = X'7FE00000'  
a Signaling NaN = B' 0 11111111 0100....0000 ' = X'7FA00000'  
s <-char-> <-fraction->
```

IEEE Floating Point: Full Range of the Representation



IEEE Floating Point Standard: Required Capabilities

- Arithmetic operations
 - add, subtract
 - divide, multiply, remainder (which is *very* complicated!)
 - compare (always exact; sign of zero is ignored)
 - ± 0 are only two distinct IEEE bit patterns that compare equal
- Square Root
- Float/Integer conversions
 - All supported float and integer formats inter-convertible
 - Rounded float-to-integer conversion
- Conversions between float and decimal
 - Decimal conversions assumed to be done in software
- Exceptions and exception handling rules

IEEE Floating Point Standard: Required Capabilities ...

Conforming implementations must recognize five exception conditions:

1. Invalid operation

- Mathematically meaningless operations
 - a. $|\infty| - |\infty|$
 - b. $\infty \times 0$
 - c. $\infty \div \infty$
 - d. $0 \div 0$
 - e. $x \text{ REM } y$ with $x = \infty$ or $y = 0$
 - f. $\text{SQRT}(\text{negative value})$
- Computationally meaningless operations
 - a. Operation on a Signaling NaN (SNaN)
 - b. Relational operation involving a NaN (“un-ordered comparison”)
 - c. Integer-conversion fault (NaN, ∞ , out of range)

IEEE Floating Point Standard: Required Capabilities ...

2. Division by zero

- N.B.: $0 \div 0$ is an invalid operation

3. Exponent overflow

4. Exponent underflow

- Result may be denormalized

5. Inexact result

- May also occur coincident with overflow or underflow

Exceptions can trap, set a flag, or both

- Flags are required, traps are recommended
 - Not all implementations support traps
- Non-trapped (masked, disabled) exceptions must set a flag
 - Default: proceed without trap
- Trap and non-trap results may differ! (see slides 71, 113)

IEEE Floating Point Standard: Required Capabilities ...

- Four rounding modes

- To nearest (the default; “ties” round to even); up; down; to zero (“chop”)
- Integer conversions are subject to rounding mode

UP: **+3.5** → **+4**, **-3.5** → **-3**
DOWN: **+3.5** → **+3**, **-3.5** → **-4**
Toward Zero: **+3.5** → **+3**, **-3.5** → **-3**
To Nearest: **±3.5** → **±4**, **±2.5** → **±2**

- Remainder defined by $r = x - y \times n$, where n is the integer nearest the exact value of x/y ; if $|n - x/y| = 1/2$, n is even
- Language rules may limit the choices
 - E.g. C/C++ standard requires rounding “toward zero”

- Arithmetic rules **require** creation of -0 in some cases!

- $\text{SQRT}(-0)$, $X - X$ (rounded down), $(-0) + (-0)$
- 0 remainder, 0 product, 0 quotient, 0 result of FIX
 - E.g., $(+2) \text{ REM } (-1)$, $(+0) \times (-1)$, $(+0) \div (-1)$, $\text{FIX}(-0.2)$
 - E.g., $|A| \div (-\infty)$, where A is finite

IEEE Floating Point Standard: Required Capabilities ...

- Arithmetic on infinities is always “exact” and correctly signed
 - If A is finite, then
$$A \pm \infty = \pm \infty, |\infty| + |\infty| = \infty, A \times \infty = \infty, A \div \infty = 0, A \div 0 = \infty,$$
etc.
 - Only exception condition recognized is “invalid operation” (e.g., $0 \times \infty, \infty \div \infty, |\infty| - |\infty|$, etc.)
- Overflow: default = $\pm \infty$ or “ $\pm \text{MaxReal}$ ” depending on rounding mode
 - Note potential problems in using MaxReal in subsequent computations!
- Single precision is required in all conforming implementations
- IEEE standard forbids double rounding in any operation
 - “The result cannot suffer more than one rounding error”

IEEE Floating Point Standard: Further Details

- “Extended” Precisions are suggested
 - Single Extended: at least 11 exponent bits, at least 32 fraction bits
 - Most implementations use double for “single extended”
 - Double Extended: at least 15 exponent bits, at least 64 fraction bits
 - Implementations *should* support
 - the extended format corresponding to the widest basic format supported
 - mixed-length operations
- Relational Predicates for languages (20 suggested)
 - Standard recommends implementation of many new relations to accommodate testing and comparing of NaNs
 - Example: $(A \text{ ?> } B)$ is true if $(A > B)$ or the relation is unordered
 - NaNs have no ordering relationship to anything (including themselves)
 - Example: (NaN eq NaN) is false, even if the NaNs are identical!

IEEE Floating Point Standard: Recommended Functions

1. `COPYSIGN(x,y)`: returns x with sign of y , for any y
2. `-x`: is x copied with sign reversed (not $0-x$)
3. `SCALB(y,N)`: returns $y \times 2^N$ for integral N
4. `LOGB(x)`: returns unbiased exponent of x
5. `NEXTAFTER(x,y)`: returns next representable neighbor of x in the direction of y
6. `FINITE(x)`: returns TRUE if x is not an infinity or a NaN, FALSE otherwise
7. `ISNAN(x)`: returns TRUE if x is a NaN
8. `x<>y`: is TRUE only when $x<y$ or $x>y$
9. `UNORDERED(x,y)`: returns TRUE if x is unordered with y
10. `CLASS(x)`: tells which of ten classes x falls into:
 - Signaling NaN; quiet NaN; $-\infty$; negative normalized nonzero; negative denormalized; -0 ; $+0$; positive denormalized; positive normalized nonzero; $+\infty$.

IEEE Floating Point Standard: Justifications

1. Facilitate program portability
 - Easier exchange of programs and floating point data among systems
 - Reproducible results (depending on the compiler, math library,...)
2. Enhance safety
 - Fewer computational eccentricities to trap the unwary
 - Reduce anomalous behavior to the inescapable minimum
 - Minimize idiosyncrasies and programming contortions
 - Simpler code is usually faster, cheaper, more robust
3. Reduce behavior unpredictability at precision and range thresholds
4. Minimize impact of wobbling precision
5. Enhance capabilities
 - Better exception handling facilities

IEEE Standard 854 (1987) generalizes 754 to binary and decimal formats

- Conformance to 754 gives conformance to 854

ESA/390 Floating Point Enhancements

Web page:

<http://www.ibm.com/s390/ieee/>

ESA/390 Floating Point Enhancements: Overview

- Design goals include:
 - Affordability
 - Minimal impact to existing applications
 - Serviceability
 - Compatibility
 - Definitive support
 - Standard conformance
- ESA/390 supports **both** hexadecimal floating point (HFP) and IEEE binary floating point (BFP)
 - Fully-conforming implementation of IEEE floating point standard for single and double precision
 - Extended precision lacks *only* the remainder function!
 - Precisely rounded decimal/binary conversions handled in software
 - Easy migration from HFP, and to and from other platforms
- Software support in OS/390, HLASM, C/C++, LE, Java, UNIX System Services, etc.

ESA/390 Hardware Enhancements

- New processor features
- Hexadecimal floating point extensions
- Floating point support instructions for HFP and BFP
- Binary (IEEE) floating point support
 - Data types, exceptions, rounding modes, instructions

ESA/390 Floating Point Enhancements: Processor Features

- Extensions controlled by “AFPR” bit (CR0.13)
- 12 additional floating point registers (AFPR)
 - 1, 3, 5, 7, 8 through 15
 - pairing always between registers FPR_n and FPR_{n+2} (as before):
0-2, 1-3, 4-6, 5-7, 8-10, 9-11, 12-14, 13-15
 - Usable by HFP and BFP operations
- Three new BFP data types
- Floating Point Control (FPC) register
- Data exception extended to store Data Exception Code (DXC)
 - Locations 144-147
 - Old “data exception” (Program Interruption Code 7) now called “decimal data exception” with zero DXC
- 121 new instructions for HFP, BFP and FP Support
 - Five new instruction formats

ESA/390 Floating Point Enhancements: BFP Data Types

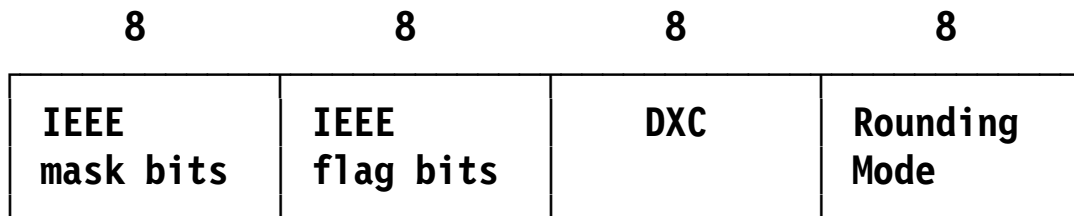
- Three data lengths
 - Short (4 bytes) and long (8 bytes) same as IEEE-standard single and double
 - Extended (16 bytes): 15-bit exponent, 112-bit fraction (113-bit significand)
 - Magnitudes: 3.4×10^{-4392} to $1.2 \times 10^{+4392}$
 - Extra precision important for intermediate computations with long variables!
 - Equivalent to “quad” format in HP PA-RISC, DEC Alpha, SUN SPARC
 - BFP long and extended formats satisfy IEEE requirement for “Single Extended” and “Double Extended”
 - Storage-operand lengths match register-operand lengths
 - Helps avoid some optimization problems (see slide 124)
- Maximum and minimum normalized and denormalized values:

Extreme Values	Short	Long	Extended
Max Norm	3.4×10^{38}	1.8×10^{308}	1.2×10^{4932}
Min Norm	1.2×10^{-38}	2.2×10^{-308}	3.4×10^{-4932}
Min Denorm	1.4×10^{-45}	4.9×10^{-324}	6.5×10^{-4966}

ESA/390 Floating Point Enhancements: BFP Data Types ...

- Infinities (characteristic all 1-bits, zero fraction)
- NaNs (characteristic all 1-bits, non-zero fraction) (see slide 104 also)
 - SNaN (Signaling NaN): leftmost fraction bit F0 = 0
 - The rest of the fraction must have non-zero bit(s) (otherwise it's an infinity)
 - Causes invalid-operation exception for almost all instructions
 - QNaN (Quiet NaN): leftmost fraction bit F0 = 1
 - Special “hardware” QNaN delivered as default for masked IEEE Invalid-operation exception: F0 = 1, rest of fraction = 0
 - Delivered operands:
 - SNaNs chosen over QNaNs
 - NaNs of same type are chosen in order of operand number
- Six classes of floating point data (see “Test Data Class” instruction)
 - Zero; normalized; denormalized; infinity; QNaN; SNaN

ESA/390 Floating Point Control Register



- Mask bits disable and enable IEEE exception “trap” conditions:

<u>bit</u>	<u>Condition</u>
0	IEEE invalid operation
1	IEEE divide by zero
2	IEEE overflow
3	IEEE underflow
4	IEEE inexact result
5–7	Reserved

- Flag Bits
 - Same definitions as for mask bits
 - Set on by masked exceptions; stay on until explicitly reset
- Masks and flags are ignored for HFP and FP Support instructions

ESA/390 Floating Point Control Register ...

- Terminology: IEEE “trap” = S/390 “interruption”
- Actions on exception conditions:
 - Mask bit = 1: set exception code in DXC field, take interruption
 - Mask bit = 0: set flag bit in FPCR, take default action
 - **Note:** results may differ between masked/unmasked exceptions:
 1. **Invalid operation:** taking interruption suppresses the instruction
 - add unlike infinities; subtract like infinities; $0 \times \infty$; $0/0$ or ∞/∞ ; $(x \text{ REM } y)$ with $x = \infty$ or $y = 0$; $\text{SQRT}(<0)$; Compare and Signal with QNaN; conversion fault: masked default = special “hardware” QNaN
 - Signaling NaN: masked default = corresponding QNaN
 2. **Divide by 0:** interrupt suppresses; masked default = $\pm \infty$
 3. **Underflow:** interrupt wraps exponent; masked default = denormalized result or zero
 4. **Overflow:** interrupt wraps exponent; masked default = infinity or MaxReal
 5. **Inexact result:** deliver rounded result (can also occur with BFP overflow or underflow)

ESA/390 Floating Point Control Register ...

- DXC (Data Exception Code) indicates which exception/trap occurred
 - 00 Decimal data exception
 - 01 Invalid AFPR used by HFP or BFP; FP-extensions hardware is installed, but CR0.13 bit = 0
 - 02 BFP instruction; FP hardware installed, but CR0.13 bit = 0
 - 08 IEEE inexact, result truncated
 - 0C IEEE inexact, result incremented
 - 10 IEEE underflow, exact result
 - 18 IEEE underflow, result inexact and truncated
 - 1C IEEE underflow, result inexact and incremented
 - 20 IEEE overflow, exact result
 - 28 IEEE overflow, result inexact and truncated
 - 2C IEEE overflow, result inexact and incremented
 - 40 IEEE division by zero
 - 80 IEEE invalid operation
- Bits 0-4 match masks and flags;
bit 5 indicates inexact result was incremented in magnitude

ESA/390 Floating Point Enhancements: Rounding Modes

- Rounding mode indicated in FPCR (bits 30-31):
 - 0** Unbiased round to nearest
 - 1** Round toward zero (chop/truncate)
 - 2** Round toward $+\infty$
 - 3** Round toward $-\infty$
- Some instructions can specify an explicit rounding-method mask field:
 - 0** Round according to the current rounding mode in the FPCR
 - 1** Biased round to nearest
 - 4** Unbiased round to nearest
 - 5** Round toward zero (chop)
 - 6** Round toward $+\infty$
 - 7** Round toward $-\infty$

Used by Convert to Fixed, Load FP Integer, Divide to Integer instructions

ESA/390 HFP Enhancements

- Previous hex floating point exceptions and Program Mask bits renamed with “HFP” prefix
 - E.g. “Exponent overflow” becomes “HFP exponent overflow”
- 26 new instructions (54 previously available)
 - Convert from fixed (all 3 precisions; round toward zero)
 - Convert to fixed (all 3 precisions, with selectable rounding)
 - Load FP integer (rounded toward zero, HFP result; all 3 precisions)
 - Extended-precision ops (similar to existing short and long ops)
 - LTXR, LCXR, LNXR, LPXR (rectify low-order sign and characteristic)
 - CXR, SQXR
 - LEXR (does biased round, like LRER, LRDR)
 - Multiply short \times short to yield short (MEE, MEER)
 - Square root for short and long storage operands (SQE, SQD)
 - Load lengthened (eliminates need to clear registers first)
 - LDE(R), LXE(R), LXD(R)

ESA/390 Floating Point Support Instructions

- 8 new instructions
- 3 Load Zero, 1 Load Extended, 4 BFP-to-HFP, HFP-to-BFP conversion
 - None cause any HFP or IEEE-defined exceptions
 - CC=3 for “special case” results (may get MIN, MAX, or ∞)
- Load zero (HFP and BFP, for all three operand lengths

LZXR F0 Zero F0/F2 (old HFP code required two instructions)

- Subtracting a register from itself can cause IEEE exceptions!
...or HFP Significance exception if not masked off!
- Load extended (LXR, register-register): replaces 2 LDRs

ESA/390 Floating Point Support Instructions ...

- Conversions between BFP and HFP replace 30+ instructions with one!
- Convert HFP to BFP (long HFP to short or long BFP)

TBDR F2, RM, F3 Convert long HFP in F3 to long BFP in F2
TBEDR F9, RM, F1 Convert long HFP in F1 to short BFP in F9

- **Note:** rounding modifier RM=0 means “round toward zero,” **not** “according to current rounding mode”

- Convert BFP to HFP (short or long BFP to long HFP)

THDER F0, F1 Convert short BFP in F1 to long BFP in F0
THDR F12, F5 Convert long BFP in F5 to long BFP in F12

- **Note:** no rounding modifier!
- Exceeding range gives 0 or \pm HexMax

ESA/390 Floating Point Enhancements: BFP Instructions

- 87 new instructions (almost all mnemonics contain a 'B')
- Actions are more complex than HFP equivalents, due to
 - rounding modes
 - NaNs and infinities
 - maskable exceptions
 - different exponent ranges
- Full range of usual operations
 - Add, subtract, multiply, divide, square root
 - Multiply includes short \times short = long, long \times long = extended
 - NaN op NaN gives 1st operand
 - Load, Load Complement/Negative/Positive
 - These don't signal invalid operation with SNaN operands!
 - Load Complement implements IEEE recommended function $-x$ (see slide 62)
 - CC = 3 generally indicates NaN operand or result (no CC = 3 for HFP ops)
- No unnormalized data (or unnormalizing arithmetic operations)
 - Denormalized numbers are default result of underflow

ESA/390 Floating Point Enhancements: BFP Instructions ...

- BFP Comparisons (HFP instructions unchanged)
- Normal compare:

CEBR F1,F9 Compare long float values in F1, F9

- Invalid operation signaled if at least one operand is a SNaN
 - CC=3 for comparisons involving QNaNs (“unordered”)
 - **Note:** CC=3 setting is new with BFP compare instructions
 - Compare and Signal: like Compare, but all NaNs signal Invalid Op
- KEBR F1,F9 Compare long float values in F1, F9, signal all NaNs**
- Sets CC=3 for comparisons involving any NaN
 - Comparing mixed-length operands requires lengthening the shorter
 - Implements recommended IEEE functions $\lt;$, $\gt;$, UNORDERED (see slide 62)

ESA/390 Floating Point Enhancements: BFP Instructions ...

- Convert to 32-bit fixed-point binary integer from float
 - Previous HFP code to convert long float in F0 to integer in R0:

```
AD  0,DCON      Add and normalize
STD 0,FTEMP     Store the intermediate sum
L   0,FTEMP+4   Get the integer result in R0
-- -- --
FTEMP DS  D      Doubleword temporary
DCON  DC  X'4F08 0000 0000 0000'  Conversion constant
```

- 3 instructions and 3 storage references
- New HFP and BFP code to do the same (note rounding-mode mask!):

```
CFDR  R0,RM,F0   HFP to Integer
```

```
CFDBR R0,RM,F0   BFP to Integer
```

- 1 instruction and no storage references!
- Convert to fixed supports rounding mask field
- If invalid operation and mask bit = 0, sets CC = 3

ESA/390 Floating Point Enhancements: BFP Instructions ...

- Convert from 32-bit fixed-point binary integer to float
 - Previous HFP code to convert integer in R0 to long float in F0:

```
X    0,DCON+4  Invert sign bit of integer
ST   0,FLOAT+4 Store result in long constant
LD   0,FLOAT   Load the unnormalized value
AD   0,DCON    Add and normalize
```

— — —

```
DCON DC  X'CE00 0000 8000 0000'  -2**31
FLOAT DC X'4E00 0000 0000 0000'  Pseudo-zero, Exponent = +14
```

- 4 instructions and 4 storage references
- New HFP and BFP code to do the same:

```
CDFR  F0,R0    Integer to HFP
CDFBR F0,R0    Integer to BFP
```
- 1 instruction and no storage references!
- CC unchanged; rounding according to current rounding mode

ESA/390 Floating Point Enhancements: BFP Instructions ...

- Load and Test, Load Lengthened
- BFP Load and Test sets CC=3 is result is a NaN
 - HFP Load and Test instructions never set CC=3
- New “lengthening” instructions for HFP and BFP:

LDER	r1,r2	HFP short to long
LXDR	r1,r2	HFP long to extended
LXER	r1,r2	HFP short to extended

LDEBR	r1,r2	BFP short to long
LXDBR	r1,r2	BFP long to extended
LXEBR	r1,r2	BFP short to extended

- Condition code is unchanged
- Invalid operation signaled if operand is SNaN
- NaNs extended with trailing zeros, shortened by truncating on the right
- Stable encoding of special values in NaNs must use leftmost bits
- Previous HFP code to lengthen a short float:

SDR	F0,F0	Clear FPreG 0
LE	F0,X	Load the short float; F0 now has long value

ESA/390 Floating Point Enhancements: BFP Instructions ...

- Load FP Integer (rounds FP operand to FP integer value)

- Previous HFP code for integer part of long float in F0:

```
AD    F0,=X'4F00 0000 0000 0000'  Force fraction part off
```

- New HFP and BFP code to do the same:

```
FIDR  F0,F0    HFP integer part of F0
FIDBR F0,R0    BFP integer part of F0
```

- CC is unchanged

- Load rounded (long or extended to short, long to short)

- Three new HFP instructions added (LEDL, LDXL, LEXL)

- Alternate mnemonics for existing ops: LEDL = old LRER, LDXL = old LRDR

- Exponent overflow possible only if operand = HFP MaxReal

- BFP operations (LEDBR, LDXBR, LEXBR) must account for exponent width reduction

- Possible exceptions: Overflow, Underflow, Inexact, Invalid Op

- Rounding controlled by current rounding mode

ESA/390 Floating Point Enhancements: BFP Instructions ...

- Divide to integer
- Produces 2 BFP results: integer quotient, exact (rounded) remainder
 - Similar to MOD/modulo, but not the same!
- May need re-execution to complete
 - CC setting indicates status of calculation
- Example: calculate remainder of 1000/0.73 using short BFP:

```
LE    F8,=EB'1000'    Set up dividend
LE    F3,=EB'0.73'    Set up divisor
DIV   FIEBR F8,F3,F0,RM  Partial quotient in F0, partial remainder in F8
BC    2,DIV           Iterate if not complete
-- --                Final remainder in F8
```

– Rounding modifier RM selects rounding mode

- HFP (long precision) code for remainder of 1000/0.73:

```
LD    F0,=D'1000'    Get numerator (A)
DD    F0,=D'0.73'    Compute quotient (A/B)
AD    F0,=X'4F00 0000 0000 0000' Force fraction part off
MD    F0,=D'0.73'    Compute (B * IntegerPart(A/B))
LCDR  F0,F0          Complement
AD    F0,=D'1000'    A - (B * Int(A/B)) is the remainder in F0
```


ESA/390 Floating Point Enhancements: BFP Instructions ...

- Multiply-and-add, Multiply-and-subtract (short and long)

- $op1 = op3 \times op2 \pm op1$, with precise intermediate result

MAEBR F5,F2,F6 Evaluate $F5 = F5 + (F2 * F6)$ in short BFP

- Only the final result is rounded (possibly non-standard result!)
- Result NaN taken from operand in order op3, op2, op1
 - SNaN defaulted to QNaN takes precedence
- Consistent with the same operations on RS/6000

ESA/390 Floating Point Enhancements: BFP Instructions ...

- Test Data Class: bit pattern of second operand address selects first-operand class(es) to be tested

Class	+ sign	- sign
Zero	20	21
Norm	22	23
Denorm	24	25
Infinity	26	27
QNaN	28	29
SNaN	30	31

- Match of mask bit to operand class sets CC=1
- No invalid-operation signal for SNaNs (so you can test for them!)
- 12-bit pattern fits in instruction's displacement field
- Implements recommended IEEE functions FINITE, ISNAN, CLASS (see slide 62)

ESA/390 Floating Point Enhancements: Control Instructions

- FPCR Load/Store (register-storage), Set/Extract (register-register)

SFPC	R1	Set FPCR from GPR 1
EFPC	R1	Extract FPCR into GPR 1
LFPC	D2(B2)	Load FPCR from memory
STFPC	D2(B2)	Store FPCR into memory

- Set Rounding Mode: rightmost two bits of 2nd-operand address

SRNM	0(R5)	Set rounding mode from bits 30–31 of GPR 5
SRNM	1	Set rounding mode to 'toward zero'

ESA/390 Software Enhancements

- OS/390 support
- High Level Assembler support
- C/C++ support, including LE and runtime libraries
- Considerations for Java, DB2, CICS, IMS, other products
- Programming conventions

OS/390 Operating System Support

- Delivered with OS/390 V2R6
 - Can emulate FP Extensions on older hardware; useful for testing, development, validation
- Task initiation
 - TCB or SRB not initially enabled for BFP ops or AFPR, FPCR
 - First exception is trapped:
 - If FP-extensions hardware is installed, turn on CR0.13, enable register saving
 - May want to use AFPRs for HFP only
 - If DAT on and no hardware, simulate instructions and enable register saving
 - Registers initialized to zero for new tasks
 - FPCR = 0 disables traps, resets flags, sets rounding mode to “nearest”
- XCTL, LINK, LOAD-and-CALL: callee gets caller's registers
 - LE enclaves: LINK requires special initialization handling
- IEAFP service call to stop saving/restoring registers
 - Can reduce task-switching overheads

OS/390 Operating System Support ...

- Exceptions:
 - DXC = 0 for old decimal data exception (Program Interruption Code 7)
 - DXC passed in parm area to ESPIE, ESTAE, FRRs
 - SPIE must use DXC in FPCR
- Two CVT bits:
 - CVTBFP (in CVTOSLV2): you can use FP Extensions (may be emulated)
 - CVTBFPH (in CVTFLAG2): the FP Extensions hardware is present
- Sysplex: WLM directs jobs to BFP-capable systems
- UNIX System Services: enhancements to `fork()`, `exec()`, `spawn()`, `ptrace()`, `pthread_create()`, and **od** command

ESA/390 Floating Point: HLASM Support

- Delivered with High Level Assembler R3
 - Fully compatible with existing HFP instructions and data
- All new HFP, BFP and FP Support instructions
- Extend DC syntax of E/D/L-type constants for BFP data:

DC type — B — ' nominal value  '

- $m = 1$ for biased round to nearest; $m = 4, 5, 6, 7$ have same meanings as rounding-modifier masks (see slide 73)
- Default is R4 (“round to nearest”)

ESA/390 Floating Point: HLASM Support ...

- Signed zeros supported

NegZero	DC	EB'-0'	generates X'80000000'
	DC	EH'-0'	generates X'80000000'
OldZero	DC	E'-0'	generates X'00000000'

- Examples of BFP conversions:

Short1	DC	EB'1.0'	generates X'3F800000'
Long10th	DC	DB'0.1'	generates X'3FB99999 9999999A'
Tenth	DC	EB'0.1'	generates X'3DCCCCCD' (rounded to nearest)
	DC	EB'0.1R5'	generates X'3DCCCCC' (rounded toward zero)
ExtBig	DC	LB'1.5E4000'	generates X'73E73A97 C4B5FD80 5291940A C3F3C992'

- Minimum bit-length modifiers are 9, 12, 16 for E, D, and L
 - No fraction bits remain at minimum bit length

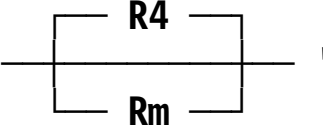
ESA/390 Floating Point: HLASM Support ...

- Extreme values:
 - Nominal values may be (MAX) and (MIN)
 - (MIN) is the minimum normalized value
 - Minimum denormalized value = X'000...001'
- Special values:
 - Nominal values may be (SNaN), (QNaN), (NaN), (INF)
 - Leading fraction bits for NaNs: 01 for (SNaN), 11 for (QNaN), 10 for (NaN) (the special “hardware” QNaN) (see slide 69)
 - Minimum bit lengths for NaNs: 11, 14, 18 for E, D, and L
- Examples:

ShortMin	DC	EB'(Min)'	generates	X'00800000'
ShortInf	DC	EB'(Inf)'	generates	X'7F800000'
LongSNaN	DC	DB'(SNaN)'	generates	X'7FF40000 00000000'
ExtMax	DC	LB'(Max)'	generates	X'7FFEFFFF FFFFFFFF FFFFFFFF FFFFFFFF'
LongDMin	DC	XL8'1'	generates	X'00000000 00000001'
- “Parameterized” NaNs discussed at slide 104

ESA/390 Floating Point: HLASM Support ...

- Improved and directed rounding supported for HFP data also:

DC type — H — ' nominal value 

ShortH1 DC EH'1.0'
LongH100 DC DH'100'
ExtBigH DC LH'1.5E75'

- Existing HLASM float hex conversions are very good;
but “difficult numbers” require more highly precise conversions
- Examples:

DC D'.303325544866797714604E-10' generates X'382159DA E5B7B6BD' (old)

DC DH'.303325544866797714604E-10' generates X'382159DA E5B7B6BE' (new)

DC D'.185240322463448422373E-23' generates X'2D23D4A8 0F402692' (old)

DC DH'.185240322463448422373E-23' generates X'2D23D4A8 0F402693' (new)

ESA/390 Floating Point: HLASM Toolkit Feature Support

- All new HFP, BFP and FP Support instructions supported by
 - Interactive Debug Facility
 - New AFPR command to enable display of FPCR, AFP registers
 - Disassembler
 - Program Understanding tool

C/C++ Compiler Support

- OS/390 V2R6 required (compiler uses some runtime library routines)
 - BFP instructions not used at compile time
- No support for mixing binary/hex floating point in one compilation unit
 - Emphasis is on BFP
 - Assumption: mixed-mode applications rare
 - Supported... but the user is responsible for managing the mixtures!
- Many new or changed options:

`arch(3)`

OK to use AFPR, all new instructions

`float([no]afp)`

Selects whether or not to use AFPR

`float(ieee|hex)`

Selects float mode

C/C++ Compiler Support ...

`float([no]fold)`

OK to evaluate constant expressions at compile time

`round(N|M|P|Z)`

Determines compile-time rounding of constant expressions

`float([no]rrm)`

rrm means runtime changes to rounding mode; compiler restricts some optimizations

`[no]strict`

Restrict compiler optimizations so results are determinable and repeatable; `nostrict` relaxes conformance to IEEE rules and allows some reorderings for performance gains

`float([no]maf)`

maf means OK to use multiply-and-add, multiply-and-subtract ops

- Strictly-compliant code generated when `strict,nomaf` specified
 - If `nostrict` and `float(ieee)`, get maf

C/C++ and LE Runtime Library Support

- Library functions include routines
 - To perform correctly rounded decimal/float conversions
 - Uses same algorithms as HLASM
 - For users who must manage mixed floating point modes during execution:
 - Return FP mode of compilation unit
 - Override FP mode of compilation unit
 - Query and set fields of FPCR
 - Determine available hardware and operating system support
 - Convert between HFP and BFP
 - ... and others, documented in the LE Vendor Interface book
 - Other functions are documented on the LE web site
- Math functions: Sun's “Freely Distributed Math Library” (fdlibm)
 - Long (64-bit) precision only
 - Default rounding mode only

C/C++ and LE Runtime Library Support ...

- Class library support for BFP packaged in DLLs
 - Assumption: one copy of the DLL will be shared by all applications
 - Feature-test macro and name-mangling rules distinguish BFP/HFP routines
- Language Environment
 - Runtime routines can handle modules with different FP modes
 - PPA2 has FP mode flag bit
 - Routines sensitive to mode (like `printf`, `scanf`) test it to process data in mode of compilation unit
 - Assembler macros can set PPA2 flag to indicate BFP is used
 - LE enclave initiation saves creator's non-volatile FPRs, sets FPCR = 0 (restored on enclave termination)
 - Supports BFP exceptions, AFPR for BFP and HFP exceptions
 - Signal catchers get control with BFP exceptions disabled (FPCR saved/restored by runtime)
 - Dump formats FPCR and AFPR

Java Language Definition

- Symbolics for INF, NAN, MAX, MIN (=DMIN; no NMIN)
- Casting functions for bits-to-float
- Relationals with NaNs return false
(not “unordered”: $x \neq x$ is true for NaNs)
- Only rounding mode is “to nearest”
- Overflows give ∞ , underflows give zero
- Integer-to-float conversions give no indication of lost bits
- Float-to-integer conversions:
 - NaN gives 0
 - Representable values round to zero (truncate)
 - Non-representable values (including ∞) give MAXINT or MININT (no exceptions indicated)
- Double-to-float rounded to nearest, or to 0 or ∞ (no exceptions indicated)
- Coalesces all NaN values to “a single conceptual NaN value”
- Java's support of the IEEE standard is (currently) somewhat limited...

Java Support: Implementations

- Improved performance for (previously emulating) applications
- Java Virtual Machine (JVM) on OS/390 V2R6 and new hardware will generate BFP instructions
 - Uses C/C++ runtime library
 - Old hardware: use Java Runtime Library IEEE simulation
- Just-In-Time (JIT) compiler on OS/390 V2R6 and new hardware will generate BFP instructions
 - Old hardware: use Java Runtime Library IEEE simulation
- High-Performance Java (HPJ) compiler R1 uses IEEE simulation
 - March 1999 PTF provides AFPR and BFP-instruction support

Other Products' Support

- VS Fortran:
 - AFPR support for HFP (APAR PQ26305)
- DB2 UDB for OS/390
 - Phased approach to support, over several releases
 - Initially: BFP converted to HFP internally
 - Don't use BFP data until “Toleration PTFs” (APARs PQ30062, PQ30063) are applied
 - Utilization of new instructions will require compatibility testing
 - A new data type and related capabilities will be considered
- CICS
 - Transaction Server currently does not support BFP applications; Open Transaction Environment tasks can use BFP
- IMS is “transparent” to floating point data types
- DFSORT intends to provide BFP support equivalent to HFP
- Component Broker intends BFP support in the future
- TCP/IP intends to add support to RPC for BFP

ESA/390 Floating Point Software Conventions

- Old floating point register convention: all FPRs volatile
 - Too few FPRs to justify optimizing contents across calls
- Now: FPRs 0-7 volatile; 8-15 saved and restored across calls
 - Called routine must save modified non-volatile FP registers
 - Allows optimization of variables into registers across calls
 - No impact on programs using old conventions for FPRs 0-6

ESA/390 Floating Point Software Conventions ...

- Floating Point Control register
 - Volatile: need not save/restore exception flags, DXC
 - Non-volatile: must save/restore masks, rounding mode
- Application is responsible for knowing what data types are used
 - Assumption: rarely intermix HFP and BFP data types
- Asynchronous exits must preserve FPRs, FPCR
- BAKR, PC calls get caller's FPRs and FPCR
 - Not saved on linkage stack

ESA/390 Floating Point Software Conventions ...

- Standard: “NaNs should ... afford retrospective diagnostic information inherited from invalid or unavailable data and results.”
- Potential value in “tagging” or “parameterizing” NaNs
 - Can distinguish/describe many types of error or warning conditions
 - Propagation of NaNs helps identify initial error condition
- NaN representations and parameterizations
 - Recommend reserving the two high-order fraction bits
 - 01** SNaN (rest of fraction = 0)
 - 10** Hardware-generated default QNaN (rest of fraction = 0)
 - 11** Software-created default QNaN (rest of fraction = 0)
 - Recommend using at most 21 high-order bits for NaN parameter values
 - Longer fields may suffer truncation
- Discussion question: Should we reserve the high-order bit of this 21-bit field to distinguish between IBM/user “tagging” info?

**Contrasting Binary and
Hexadecimal Floating Point
on ESA/390**

Binary and Hex Floating Point: General Differences

- Summary of HFP and BFP data formats

Format	sign width	characteristic width	characteristic bias	fraction width
Hex Short	1	7	64	24
Hex Long	1	7	64	56
Hex Extended	1	7	64	112
Binary Short	1	8	127	24
Binary Long	1	11	1023	53
Binary Extended	1	15	16383	113

Binary and Hex Floating Point: General Differences ...

1. Range: Significant differences

- Short IEEE binary has half the range of short S/390 hex
- Double IEEE binary has four times the range of long S/390 hex
- Extended IEEE binary has 64 times the range of extended S/390 hex

2. Precision: Minor differences

- Short IEEE binary has 0 to 3 more fraction bits than short S/390 hex
- Long IEEE binary has 0 to 3 fewer fraction bits than long S/390 hex
- Extended IEEE binary has 1 to 4 more fraction bits than extended S/390 hex

3. Special values: Significant differences

- S/390 hex has no representation for NaNs or infinities

4. Unnormalized/denormalized values: Important differences

- S/390 hex supports (but rarely produces) unnormalized data
- IEEE denorms can indicate “lost” precision

Binary and Hex Floating Point: Representation Differences

- A 24-bit binary fraction is **not** the same as a 6-hex-digit fraction!
 - Computationally: 6 hex digits “statistically equivalent” to approximately 22 bits
- Example: consider the decimal value 0.1
 - In binary: 0.0001 1001 1001 1001 1001 1001 1001 1001 ... etc.
- Short hex-normalized float fraction has 21 significant bits:
 - Normalized, rounded fraction is X'19999A' (relative error $\approx 2^{-22}$)
 - Stored value is X'4019999A'
- Single precision IEEE float fraction has 24 significant bits:
 - Normalized, rounded fraction is X'CCCCCD' (relative error $\approx 2^{-25}$)
 - With the implied 1-bit, stored value is
B'0 01111011 10011001100110011001101' or X'3DCCCCCD'

Binary and Hex Floating Point: Computational Differences

- Length coercions handled very differently
 - BFP must round when shortening, under rounding-mode control
- IEEE longer precisions are not simple extensions of shorter precisions
 - Different exponent widths (unlike S/390 hex!)
 - Can't simply truncate low-order fraction bits, or append zeros
 - Overflow, underflow possible when shortening precision
- Exceptions have similarities and differences (see slide 113)
- Inverses of tiny hex numbers overflow: $\text{HexMax} \times \text{HexMin} \approx 1/16$
- Inverses of large binary numbers are denormals (if underflow is masked off); $\text{BinMax} \times \text{BinMin} \approx 4$
 - IEEE advantage: inverses exist for all normal values
 - Lose a bit or two or precision for inverses of numbers near BinMax
- Initial hardware implementation optimized for HFP (naturally!)
 - Single internal format for HFP and BFP
 - Almost all BFP operations pipelined in hardware

Binary vs. Hex Floating Point: 4-Byte (Short) Precision

	IEEE	IBM System/370 Hex												
Short (32-bit) Format	<table border="1"> <tr> <td>1</td> <td>8</td> <td>23</td> </tr> <tr> <td>s</td> <td>char</td> <td>fraction</td> </tr> </table>	1	8	23	s	char	fraction	<table border="1"> <tr> <td>1</td> <td>7</td> <td>24</td> </tr> <tr> <td>s</td> <td>char</td> <td>fraction</td> </tr> </table>	1	7	24	s	char	fraction
1	8	23												
s	char	fraction												
1	7	24												
s	char	fraction												
Characteristic	8 bits (exp: -126 to +127) range $10^{**}(-38)$ to $10^{**}(+38)$	7 bits (exp: -64 to +63) range $10^{**}(-78)$ to $10^{**}(+75)$												
Fraction	base 2; implied leading 1-bit preceding the radix point; 24 bits = 6 decimal digits	base 16; 6 hex digits (21-24 bits) = 6 decimal digits												
Conversion Problems	unrepresentable magnitudes	→ loss of precision ← unrepresentable spec. vals.												
Special Values	Infinities; De-normalized numbers; Not-a-Number (NaN)	(none)												
Rounding	up, down, to zero, to nearest	(none) (except ACRITH assist)												

Binary vs. Hex Floating Point: 8-Byte (Long) Precision

	IEEE	IBM System/370 Hex												
Long (64-bit) Format	<table border="1"> <tr> <td>1</td> <td>11</td> <td>52</td> </tr> <tr> <td>s</td> <td>char</td> <td>fraction</td> </tr> </table>	1	11	52	s	char	fraction	<table border="1"> <tr> <td>1</td> <td>7</td> <td>56</td> </tr> <tr> <td>s</td> <td>char</td> <td>fraction</td> </tr> </table>	1	7	56	s	char	fraction
1	11	52												
s	char	fraction												
1	7	56												
s	char	fraction												
Characteristic	11 bits (exp: -1022 to +1023) range $10^{**}(-308)$ to $10^{**}(+308)$	7 bits (exp: -64 to +63) range $10^{**}(-78)$ to $10^{**}(+75)$												
Fraction	base 2; implied leading 1-bit preceding the radix point; 53 bits = 15 decimal digits	base 16; 14 hex digits (53-56 bits) = 15 decimal digits												
Conversion Problems	<p>loss of precision ←</p> <p>→ unrepresentable magnitudes</p> <p>→ unrepresentable spec. vals.</p>													
Special Values	Infinities; De-normalized numbers; Not-a-Number (NaN)	(none)												
Rounding	up, down, to zero, to nearest	(none) (except ACRITH assist)												

Binary vs. Hex Floating Point: 16-Byte (Extended) Precision

	IEEE	IBM System/390 Hex
Extended (128bit) Format	<div style="display: flex; justify-content: space-around; align-items: center;"> 1 15 112 </div> <div style="display: flex; justify-content: space-around; align-items: center; border: 1px solid black; padding: 5px;"> s char fraction </div> <div style="display: flex; justify-content: space-around; align-items: center;"> /-/ </div>	<div style="display: flex; justify-content: space-around; align-items: center;"> 1 7 56 8 56 </div> <div style="display: flex; justify-content: space-around; align-items: center; border: 1px solid black; padding: 5px;"> s char fraction /// fraction </div>
Characteristic	15 bits (exp: -16382 to +16383) range $10^{**}(-4392)$ to $10^{**}(+4392)$	7 bits (exp: -64 to +63) range $10^{**}(-78)$ to $10^{**}(+75)$
Fraction	base 2; implied leading 1-bit preceding the radix point; 113 bits = 33 decimal digits	base 16; 28 hex digits (109-112 bits) = 32 decimal digits
Conversion Problems		<ul style="list-style-type: none"> —→ loss of precision —→ unrepresentable magnitudes —→ unrepresentable special values
Special Values	Infinities; De-normalized numbers; Not-a-Number (NaN)	(none)
Rounding	up, down, → zero, to nearest	(none)

IEEE vs. IBM Hex Floating Point: Exception Differences

370 Exception	Maskable	Masked Action	Unmasked (Interrupt) Action
Exponent Overflow	No		Wrap (scale) characteristic
Exponent Underflow	Yes	Deliver True Zero	Wrap (scale) characteristic
Zero Divide	No		Dividend operand unchanged
Lost Significance	Yes	Deliver True Zero	Deliver pseudo-zero

IEEE Exception	Maskable	Masked Action	Unmasked (Interrupt) Action
Invalid Operation	Yes	Deliver QNaN	Implementation-defined
Zero Divide	Yes	Signed Infinity	Implementation-defined
Exponent Overflow	Yes	Infinity or <u>MaxReal</u>	Re-biased (scaled) result
Exponent Underflow	Yes	De-norm or zero	Re-biased (scaled) result
Inexact Result	Yes	Deliver result	Deliver calculated result

Note: Both 370 and IEEE results can differ between masked and unmasked actions

Other IEEE Implementations

IEEE-754 Standard

“...hardware components that require software support to conform to the standard shall not be said to conform apart from such software.”

General Observations

- The IEEE standard allows a lot of flexibility!
- Many older implementations of the IEEE standard “cut corners”
 - Implement only a subset of the standard (but claim “support” or even “adherence”)
 - E.g., single, extended, rounding, remainder, SQRT, int/float conversions
 - E.g., incomplete support of exception handling
- Some implementations provide greater precision or range internally (in registers) than is (or can be) stored in memory
 - Intel, RS/6000 require special actions for strict conformance
 - Hardware defaults often imply greater precision than operands provide
 - Exact reproducibility of results across platforms may be affected
- IBM hardware: Intel boxes, AS/400, RS/6000, System/390
 - Some systems may not support all standard-required items in hardware (e.g., remainder, SQRT, int/float conversions) or standard-recommended items (e.g., extended)

1. Fully standard-conforming implementation
2. Provides single, double, and double-extended (80 bits)
 - Register operands are extended to 80-bit length; high-order 1-bit is explicit
 - Storing an extended operand as single or double causes extra rounding
 - Single and double formats exist in memory only
3. Precision control (PC) bits determine result precision
 - Register operands can be coerced (rounded) to single and double precision by setting PC bits
 - Rounding clears unused bits on right to 0's when precision is reduced
 - Overflows and underflows may not be detected until result is stored
 - Setting PC bits required for IEEE standard conformance (in single or double) for arithmetic results
 - But not for exceptions; some are detected only on stores
4. Reduced precision: possible performance increase or decrease
 - Performance is precision-sensitive
5. Also provides denormal-operand exception

Intel ...

1. Transcendental functions (error < 1 ulp if rounded to nearest):
SIN, COS, SINCOS, TAN, ATAN, $2^X - 1$, $Y \times \log_2 X$, $Y \times \log_2(X+1)$
2. IEEE functions SCALB and LOGB
3. On-chip constants: 0, 1, π , $\log_2 10$, $\log_2 e$, $\log_{10} 2$, $\log_e 2$
 - Internal precision > 64; rounded to required precision on load
4. SNaN: high-order fraction bit = 0
 - Never generate SNaN as a computational result
5. SNaN converted to QNaN by setting high-order fraction bit to 1
 - ESA/390 uses same SNaN/QNaN convention
6. “Real Indefinite”: *negative* QNaN with fraction = B'100..00'
 - Generated as default result for many operations
7. QNaN: response for masked invalid operation, or when at least one operand is a QNaN
8. Storing a NaN truncates the significand on the right
9. Double-NaN result is the NaN with the larger fraction

RISC System/6000

- IEEE standard conformance complete for double precision
- All arithmetic done in (at least) double precision
 - Hardware is “tuned” for double precision
 - Single precision operands extended to double when loaded into registers
 - Multiply-add instructions ($\pm A \times B \pm C$) use 106-bit intermediate fraction
 - Fast! (Typical example of trading conformance for performance)
 - Answers differ from double precision operations and from standard
 - Directed rounding of negated results may give unexpected results
 - Extended precision implemented (in software) as double+double
- Multiple-NaN-operand results delivered in order of operand number
 - ESA/390 uses same convention

RISC System/6000 ...

- Near-complete standard conformance for single precision
 - Depends on supporting software for conformance
 - Single-precision store assumes previous “round-to-single” operation
 - Store truncates exponent and fraction bits without tests or exceptions
 - Conforming single precision results require rounding after each operation
 - Double rounding inevitable (but bias $\approx 2^{-29}$)
 - No hardware square root
- Exception detection and reporting is expensive
 - Invalid operation flags indicate which type of invalidity
 - Trapped results already set to defaults!

Binary and Hexadecimal: Differences to Watch Out For

- Things to think about when
 - porting to S/390 from other IEEE-supporting platforms
 - enhancing HFP applications to support BFP

Standard Conformance and Language Issues

- Not all hardware implementations consistently support all Standard items
 - But results are usually “closely” reproducible
- Satisfying IEEE standard says nothing about satisfying language standards
 - IEEE's suggested language enhancements (for “unordered” relations) require significant extensions
 - Possible conflict between IEEE standard's requirements and language standards (which may be “biased round to nearest” or “round toward zero”)
- How (and whether) to provide control of rounding mode?
 - Dynamic vs. static vs. none? A callable service? New language elements?
 - Should compile-time expression evaluation be influenced by rounding mode?
 - If rounding mode is changed at execution time, compiler code-motion optimizations can be fooled. (“User Beware”?)
- Syntax for initializing data to NaN or infinity at compile time?
- How to handle relational expressions that may have NaN as a value?

Coding Implications

- Don't transliterate HFP code to BFP without analysis!
- Single-mode computation recommended
- Be **very** careful when mixing data types

Bit Pattern	HFP Value	BFP Value
X'3F80 0000'	0.03125	1.0
X'4110 0000'	1.0	9.0
X'4264 0000'	100.0	57.0
X'7FFF FFFF'	MaxReal	QNaN

- New instructions provide fast type conversion
- Remember that most bit patterns are valid for HFP and BFP instructions

Compiling and Optimization Implications

- Have to know what representation to use for constants (literals) and data initialization
 - Single mode per compilation unit simplifies many choices
- Optimization oddities: algebraic “identities” and constant propagation
 - Signed zeros, NaNs, and infinities require care in optimization

$0 / X \rightarrow 0 ?$	Fails for $X = 0$
$1 * X \rightarrow X ?$	Fails for $X = \text{NaN}$
$0 * X \rightarrow 0 ?$	Fails for $X = \text{NaN}$ or infinity
$X - X \rightarrow 0 ?$	Fails for $X = \text{NaN}$ or infinity
$X / X \rightarrow 1 ?$	Fails for $X = \text{NaN}$ or infinity
$X = X \rightarrow \text{TRUE} ?$	Fails for $X = \text{NaN}$
$X > Y \rightarrow Y < X ?$	Fails for $X = \text{NaN}$
$-X = 0 - X ?$	Fails for $X = +0$

- **Hand-coded optimizations** should be reviewed carefully!
- Optimization oddities: code motion
 - Most compilers won't move constant expressions out of loops if they might cause exceptions
 - IEEE permits exceptions where none would occur with S/390 hex (e.g., when shortening; inexact; invalid operation)

Compiling and Optimization Implications ...

- Optimization oddities: register re-use
 - Unpredictability of results vs. optimization level
 - Different results from register-length vs. storage-length differences

```
REAL*4 A, B, C, D, E
  - - -
A = B + C
E = A + D
```

With no optimization, the first sum is stored in single precision.

With optimization, A is kept in a register; final result may easily be different.

- Intel example: optimized results will change unless precision control instructions are inserted
- Unlocalized optimization effects on answers
 - In the above example, changes elsewhere in the program may cause the register holding A to be spilled, de-optimizing $E=B+C+D$ (possibly changing results)
- “Long” internal registers (e.g., Intel) imply several problems:
 - Possible double rounding
 - Shortening from extended to double or single would cause double rounding
 - Violates standard (cannot claim full conformance)
 - Unlikely to cause numerically significant errors, however

Library/Run-Time Issues

- NaNs and infinities represented externally
 - Run-time libraries provide an explicit representation
- Mapping of IEEE Standard exceptions onto existing language and run-time exception-handling rules
 - Should not be a problem, except for “inexact”:
most language standards have no such concept
- Is handling (if any) of signed zeros a problem?
 - Structure/field compares in storage (of +0 to -0) will cause problems.

Summary

Summary

ESA/390+OS/390

1. Rich and robust implementation of IEEE binary floating point
 - Fully-conforming support of an open, industry standard
2. Additional floating point registers benefit all floating point applications
3. Easy migration from older machines via simulation
4. Compatible with all existing hexadecimal floating point applications
 - Numerous enhancements available for hexadecimal floating point
 - Inter-convertability for BFP and HFP data
5. Enhanced portability between ESA/390 and other platforms

References

- *Architecture and Software Support for IEEE Floating Point*, IBM Journal of Research and Development (to appear).
- ANSI/IEEE Std 754-1985: *IEEE Standard for Floating Point Arithmetic*, The Institute of Electrical and Electronics Engineers, Inc, 345 East 47th Street, New York, NY 10017, USA; 1985.
- ANSI/IEEE Std 854-1987: *IEEE Standard for Radix-Independent Floating Point Arithmetic*, The Institute of Electrical and Electronics Engineers, Inc, 345 East 47th Street, New York, NY 10017, USA; 1987.
- David W. Matula, *A Formalization of Floating-Point Numeric Base Conversion*, IEEE Trans. Electronic Computers, EC19, pp. 681-692, 1970; *In-and-Out Conversions*, Communications of the ACM, vol. 11, no. 1, January 1968, pp. 47-50.
- Donald E. Knuth, *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms*, Addison-Wesley, 1998.

References ...

- Amos Omondi, *Computer Arithmetic Systems*, Prentice-Hall, 1994.
- Articles in *IEEE Computer*, January 1980 and March 1981.
- J.H. Wilkinson, *Rounding Errors in Algebraic Processes*, Dover Publications, 1994.
- E Schwarz et al., *The S/390 G5 Floating Point Unit Supporting Hex and Binary Architectures*, Proceedings of the 14th IEEE Symposium on Computer Arithmetic, Adelaide, Australia, April 14-16, 1999.
- H. Kuki and W. J. Cody, *A Statistical Study of the Accuracy of Floating Point Number Systems*, Communications of the ACM, vol. 16, no. 4, April 1973, pp. 223-230.
- W. Kahan, *Why do we need a floating point arithmetic standard?*, UC Berkeley, March 5, 1981.

References ...

- Pat H. Sterbenz, *Floating Point Computation*, Prentice-Hall, 1974.

“Nobody should ever have to know that much about floating-point arithmetic. But I'm afraid sometimes you might.” (Richard W. Hamming)
- David Goldberg, *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, ACM Computing Surveys, Vol. 23, No. 1, March 1991, p.5.

“...the apparently widespread belief that floating point is not a quantifiable subject.... It is possible to reason rigorously about floating point.”
- *A Conversation with William Kahan*, Dr. Dobb's Journal, Dec. 1997, p. 18.

“Most numerical computation doesn't matter, therefore a great deal of it can be wrong without deflecting the course of history. Some numerical computation matters a lot. We don't usually know what it is until afterwards.”