Floating Point Arithmetic

- Topics
  - Numbers in general
  - IEEE Floating Point Standard
  - Rounding
  - Floating Point Operations
  - Mathematical properties
  - Puzzles test basic understanding
  - Bizarre FP factoids

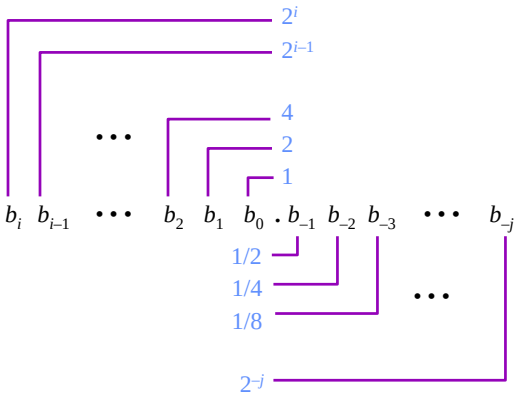
Numbers

- Many types
  - Integers
    - » decimal in a binary world is an interesting subclass
      - consider rounding issues for the financial community
  - Fixed point
    - » still integers but with an assumed decimal point
    - » advantage – still get to use the integer circuits you know
  - Weird numbers
    - » interval – represented by a pair of values
    - » log – multiply and divide become easy
      - computing the log or retrieving it from a table is messy though
    - » mixed radix – yy:dd:hh:mm:ss
    - » many forms of redundant systems – we've seen some examples
  - Floating point
    - » inherently has 3 components: sign, exponent, significand
    - » lots of representation choices can and have been made
- Only two in common use to date: int's and float's
  - you know int's - hence time to look at float's

Floating Point

- **Macho scientific applications require it**
  - require lots of range
  - compromise: exact is less important than close enough
    - » enter numeric stability problems
- **In the beginning**
  - each manufacturer had a different representation
    - » code had no portability
  - chaos in macho numeric stability land
- **Now we have standards**
  - life is better but there are still some portability issues
    - » primarily caused by details "left to the discretion of the implementer"
    - » fortunately these differences are not commonly encountered

Fractional Binary Numbers



- **Representation**
  - Bits to right of "binary point" represent fractional powers of 2
  - Represents rational number: $\sum_{k=-j}^{i} b_k \cdot 2^k$

Fractional Binary Numbers

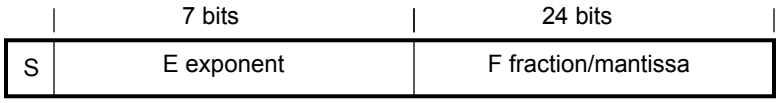
- **Value** **Representation**
  - 5-3/4 101.11_2
  - 2-7/8 10.111_2
  - 63/64 0.111111_2
- **Observations**
  - Divide by 2 by shifting right
  - Multiply by 2 by shifting left
  - Numbers of form $0.111111\ldots_2$ just below 1.0
    - » 1/2 + 1/4 + 1/8 + … + $1/2^i$ + … → 1.0
    - » We'll use notation 1.0 – ε

Representable Numbers

- **Limitation**
  - Can only exactly represent numbers of the form $x/2^k$
  - Other numbers have repeating bit representations
- **Value** **Representation**
  - 1/3 $0.0101010101[01]\ldots_2$
  - 1/5 $0.001100110011[0011]\ldots_2$
  - 1/10 $0.0001100110011[0011]\ldots_2$

Early Differences: Format

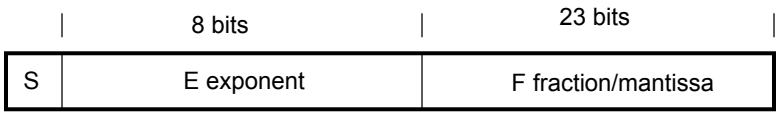
- **IBM 360 and 370**



Value = $(-1)^S \times 0.F \times 16^{E-64}$

excess notation – why?

- **DEC PDP and VAX**



Value = $(-1)^S \times 0.1F \times 2^{E-128}$

hidden bit

Early Floating Point Problems

- **Non-unique representation**
  - .345 x 10^{19} = 34.5 x 10^{17} ➔ need a normal form
  - often implies first bit of mantissa is non-zero
    - » except for hex oriented IBM ➔ first hex digit had to be non-zero
- **Value significance**
  - close enough mentality leads to a *too small to matter* category
- **Portability**
  - clearly important – more so today
  - vendor Tower of Floating Babel wasn't going to cut it
- **Enter IEEE 754**
  - huge step forward
  - portability problems do remain unfortunately
    - » however they have become subtle
    - » this is both a good and bad news story

Floating Point Representation

- **Numerical Form**
  - $-1^s M\ 2^E$
    - » Sign bit *s* determines whether number is negative or positive
    - » Significand *S* normally a fractional value in some range e.g. [1.0,2.0] or [0.0:1.0].
      - built from a mantissa *M* and a default normalized representation
    - » Exponent *E* weights value by power of two
- **Encoding**

  - MSB is sign bit
  - `exp` field encodes *E*
  - `frac` field encodes *M*

Problems with Early FP

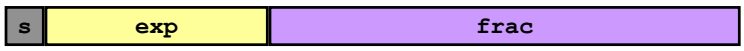
- **Representation differences**
  - exponent base
  - exponent and mantissa field sizes
  - normalized form differences
  - some examples shortly
- **Rounding Modes**
- **Exceptions**
  - this is probably the key motivator of the standard
  - underflow and overflow are common in FP calculations
    - » taking exceptions has lots of issues
      - OS's deal with them differently
      - huge delays incurred due to context switches or value checks
      - e.g. check for -1 before doing a sqrt operation

IEEE Floating Point

- **IEEE Standard 754-1985 (also IEC 559)**
  - Established in 1985 as uniform standard for floating point arithmetic
    - » Before that, many idiosyncratic formats
  - Supported by all major CPUs
- **Driven by numerical concerns**
  - Nice standards for rounding, overflow, underflow
  - Hard to make go fast
    - » Numerical analysts predominated over hardware types in defining standard
      - you want the wrong answer – we can do that real fast!!
    - » Standards are inherently a "design by committee"
      - including a substantial "our company wants to do it our way" factor
    - » If you ever get a chance to be on a standards committee
      - call in sick in advance

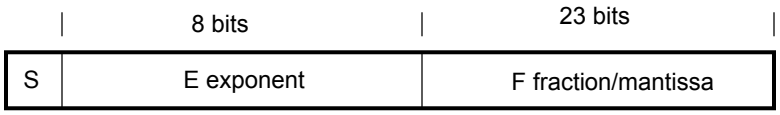
Floating Point Precisions

- **Encoding**



  - MSB is sign bit
  - `exp` field encodes *E*
  - `frac` field encodes *M*
- **IEEE 754 Sizes**
  - Single precision: 8 `exp` bits, 23 `frac` bits
    - » 32 bits total
  - Double precision: 11 `exp` bits, 52 `frac` bits
    - » 64 bits total
  - Extended precision: 15 `exp` bits, 63 `frac` bits
    - » Only found in Intel-compatible machines
      - legacy from the 8087 FP coprocessor
    - » Stored in 80 bits
      - 1 bit wasted in a world where bytes are the currency of the realm

IEEE 754 (similar to DEC, very much Intel)

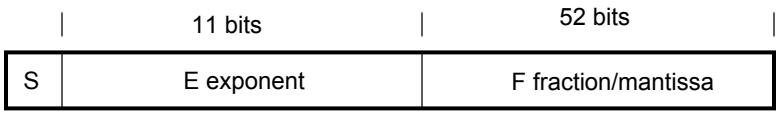


For "Normal #'s": Value = $(-1)^S \times 1.F \times 2^{E-127}$

- **Plus some conventions**
  - **definitions for + and – infinity**
    - » **symmetric number system so two flavors of 0 as well**
  - **denormals**
    - » **lots of values very close to zero**
    - » **note hidden bit is not used: V = $(-1)^S \times 0.F \times 2^{1-127}$**
    - » **this 1-bias model provides uniform number spacing near 0**
  - **NaN's (2 semi-non-standard flavors SNaN & QNaN)**
    - » **specific patterns but they aren't numbers**
    - » **a way of keeping things running under errors (e.g. sqrt (-1))**

More Precision

- **64 bit form (double)**



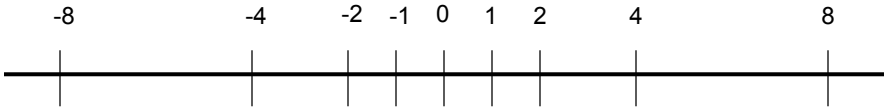
"Normalized"
Value = $(-1)^S \times 1.F \times 2^{E-1023}$

» **NOTE: 3 bits more of E but 29 more bits of F**

- **also definitions for 128 bit and extended temporaries**
  - **temps live inside the FPU so not programmer visible**
    - » **note: this is particularly important for x86 architectures**
    - » **allows internal representation to work for both single and doubles**
      - **any guess why Intel wanted this in the standard?**

From a Number Line Perspective

represented as: +/- 1.0 x {2^0, 2^1, 2^2, 2^3} – e.g. the E field



How many numbers are there in each gap??

2^n for an n-bit F field

Distance between representable numbers??

same intra-gap
different inter-gap

Floating Point Operations

- **Multiply/Divide**
  - similar to what you learned in school
    - » add/sub the exponents
    - » multiply/divide the mantissas
    - » normalize and round the result
  - tricks exist of course
- **Add/Sub**
  - find largest
  - make smallest exponent = largest and shift mantissa
  - add the mantissa's (smaller one may have gone to 0)
- **Problems?**
  - consider all the types: numbers, NaN's, 0's, infinities, and denormals
  - what sort of exceptions exist

IEEE Standard Differences

- 4 rounding modes
  - round to nearest is the default
    - » others selectable
      - via library, system, or embedded assembly instruction however
    - » rounding a halfway result always picks the even number side
    - » why?
- Defines special values
  - NaN – not a number
    - » 2 subtypes
      - quiet ➔ qNaN
      - signalling ➔ sNaN
    - » +∞ and –∞
  - Denormals
    - » for values < $|1.0 \times 2^{Emin}|$ ➔ gradual underflow
- Sophisticated facilities for handling exceptions
  - sophisticated ➔ designer headache

Benefits of Special Values

- Denormals and gradual underflow
  - once a value is < $|1.0 \times 2^{Emin}|$ it could snap to 0
  - however certain usual properties would be lost from the system
    - » e.g. x==y ⬌ x-y == 0
  - with denormals the mantissa gradually reduces in magnitude until all significance is lost and the next smaller representable number is really 0
- NaN's
  - allows a non-value to be generated rather than an exception
    - » allows computation to proceed
    - » usual scheme is to post-check results for NaN's
      - keeps control in the programmer's rather than OS's domain
    - » rule
      - any operation with a NaN operand produces a NaN
      - hence code doesn't need to check for NaN input values and special case then
      - sqrt(-1) ::= NaN

Benefits (cont'd)

- **Infinities**
  - converts overflow into a representable value
    - » once again the idea is to avoid exceptions
    - » also takes advantage of the close but not exact nature of floating point calculation styles
  - e.g. 1/0 ➔ +∞
- **A more interesting example**
  - identity: arccos(x) = 2*arctan(sqrt(1-x)/(1+x))
    - » arctan x assymptotically approaches π/2 as x approaches ∞
    - » natural to define arctan(∞) = π/2
      - in which case arccos(-1) = 2arctan(∞) = π

Special Value Rules

Operations	Result
n / ±∞	?
±∞ x ±∞	
nonzero / 0	
+∞ + +∞	
±0 / ±0	
+∞ - +∞	
±∞ / ±∞	
±∞ X ±0	
NaN any-op anything	NaN (similar for [any op NaN])

Special Value Rules

Operations	Result
n / ±∞	±0
±∞ x ±∞	?
nonzero / 0	
+∞ + +∞	
±0 / ±0	
+∞ - +∞	
±∞ / ±∞	
±∞ X ±0	
NaN any-op anything	NaN (similar for [any op NaN])

Special Value Rules

Operations	Result
n / ±∞	±0
±∞ x ±∞	±∞
nonzero / 0	?
+∞ + +∞	
±0 / ±0	
+∞ - +∞	
±∞ / ±∞	
±∞ X ±0	
NaN any-op anything	NaN (similar for [any op NaN])

Special Value Rules

Operations	Result
n / ±∞	±0
±∞ x ±∞	±∞
nonzero / 0	±∞
+∞ + +∞	?
±0 / ±0	
+∞ - +∞	
±∞ / ±∞	
±∞ X ±0	
NaN any-op anything	NaN (similar for [any op NaN])

Special Value Rules

Operations	Result
n / ±∞	±0
±∞ x ±∞	±∞
nonzero / 0	±∞
+∞ + +∞	+∞ (similar for -∞)
±0 / ±0	?
+∞ - +∞	
±∞ / ±∞	
±∞ X ±0	
NaN any-op anything	NaN (similar for [any op NaN])

Special Value Rules

Operations	Result
n / ±∞	±0
±∞ x ±∞	±∞
nonzero / 0	±∞
+∞ + +∞	+∞ (similar for -∞)
±0 / ±0	NaN
+∞ - +∞	?
±∞ / ±∞	
±∞ X ±0	
NaN any-op anything	NaN (similar for [any op NaN])

Special Value Rules

Operations	Result
n / ±∞	±0
±∞ x ±∞	±∞
nonzero / 0	±∞
+∞ + +∞	+∞ (similar for -∞)
±0 / ±0	NaN
+∞ - +∞	NaN (similar for -∞)
±∞ / ±∞	?
±∞ X ±0	
NaN any-op anything	NaN (similar for [any op NaN])

Special Value Rules

Operations	Result
n / ±∞	±0
±∞ x ±∞	±∞
nonzero / 0	±∞
+∞ + +∞	+∞ (similar for -∞)
±0 / ±0	NaN
+∞ - +∞	NaN (similar for -∞)
±∞ / ±∞	NaN
±∞ X ±0	?
NaN any-op anything	NaN (similar for [any op NaN])

Special Value Rules

Operations	Result
n / ±∞	±0
±∞ x ±∞	±∞
nonzero / 0	±∞
+∞ + +∞	+∞ (similar for -∞)
±0 / ±0	NaN
+∞ - +∞	NaN (similar for -∞)
±∞ / ±∞	NaN
±∞ X ±0	NaN
NaN any-op anything	NaN (similar for [any op NaN])

Floating Point Precisions

- Encoding

  - MSB is sign bit
  - `exp` field encodes *E*
  - `frac` field encodes *M*
- IEEE 754 Sizes
  - Single precision: 8 `exp` bits, 23 `frac` bits
    - » 32 bits total
  - Double precision: 11 `exp` bits, 52 `frac` bits
    - » 64 bits total
  - Extended precision: 15 `exp` bits, 63 `frac` bits
    - » Only found in Intel-compatible machines
      - legacy from the 8087 FP coprocessor
    - » Stored in 80 bits
      - 1 bit wasted in a world where bytes are the currency of the realm

IEEE 754 Format Definitions

S	Emin	Emax	Fmin	Fmax	IEEE Meaning
1	all 1's	all 1's	000...001	111...111	Not a Number
1	all 1's	all 1's	all 0's	all 0's	Negative Infinity
1	000...001	111...110	anything	anything	Negative Numbers
1	all 0's	all 0's	000...001	111...111	Negative Denormals
1	all 0's	all 0's	all 0's	all 0's	Negative Zero
0	all 0's	all 0's	all 0's	all 0's	Positive Zero
0	all 0's	all 0's	000...001	111...111	Positive Denormals
0	000...001	111...110	anything	anything	Positive Numbers
0	all 1's	all 1's	all 0's	all 0's	Positive Infinity
0	all 1's	all 1's	000...001	111...111	Not a Number

"Normalized" Numeric Values

- **Condition**
  - `exp` ≠ `000…0` and `exp` ≠ `111…1`
- **Exponent coded as *biased* value**
  - *E* = *Exp* – *Bias*
    - » ***Exp*** **: unsigned value denoted by** `exp`
    - » ***Bias*** **: Bias value**
      - **Single precision: 127 (*Exp*: 1…254, *E*: -126…127)**
      - **Double precision: 1023 (*Exp*: 1…2046, *E*: -1022…1023)**
      - **in general: *Bias* = 2^{e-1} - 1, where e is number of exponent bits**
- **Significand coded with implied leading 1**
  - *S* = `1.xxx…x`$_2$
    - » `xxx…x`**: bits of** `frac`
    - » **Minimum when** `000…0` **(*M* = 1.0)**
    - » **Maximum when** `111…1` **(*M* = 2.0 – ε)**
    - » **Get extra leading bit for "free"**

Normalized Encoding Example

- **Value**
  - `Float F = 15213.0;`
  - 15213_{10} = 11101101101101_2 = 1.1101101101101_2 X 2^{13}
- **Significand**
  - *S* = 1.1101101101101_2
  - *M* = `frac` = 1101101101101000000000_2
- **Exponent**
  - *E* = **13**
  - *Bias* = **127**
  - *Exp* = **140** = 10001100_2

Floating Point Representation:								
Hex:	4	6	6	D	B	4	0	0
Binary:	0100	0110	0110	1101	1011	0100	0000	0000
140:	100	0110	0					
15213:			*1*110	1101	1011	01		

Denormal Values

- **Condition**
  - `exp` = `000…0`
- **Value**
  - Exponent value $E = 1-Bias$
    - » E= 0 – Bias provides a poor denormal to normal transition
    - » since denormals don't have an implied leading 1.xxxx….
  - Significand value $M =$ `0.xxx…x`$_2$
    - » `xxx…x`: bits of `frac`
- **Cases**
  - `exp` = `000…0`, `frac` = `000…0`
    - » Represents value 0 (denormal role #1)
    - » Note that have distinct values +0 and –0
  - `exp` = `000…0`, `frac` ≠ `000…0`
    - » 2^n numbers very close to 0.0
    - » Lose precision as get smaller
    - » "Gradual underflow" (denormal role #2)

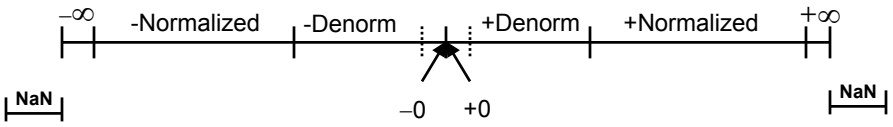
Special Values

- **Condition**
  - `exp` = `111…1`
- **Cases**
  - `exp` = `111…1`, `frac` = `000…0`
    - » Represents value ∞ (infinity)
    - » Operation that overflows
    - » Both positive and negative
    - » E.g., 1.0/0.0 = –1.0/–0.0 = +∞, 1.0/–0.0 = –∞
  - `exp` = `111…1`, `frac` ≠ `000…0`
    - » Not-a-Number (NaN)
    - » Represents case when no numeric value can be determined
      - E.g., sqrt(–1), ∞ – ∞

NaN Issues

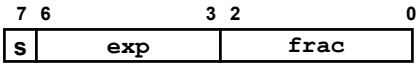
- Esoterica which we'll subsequently ignore
- qNaN's
  - F = .1u…u (where u can be 1 or 0)
  - propagate freely through calculations
    - » all operations which generate NaN's are supposed to generate qNaN's
    - » EXCEPT sNaN in can generate sNaN out
      - 754 spec leaves this "can" issue vague
- sNaN's
  - F = .0u…u (where at least one u must be a 1)
    - » hence representation options
    - » can encode different exceptions based on encoding
  - typical use is to mark uninitialized variables
    - » trap prior to use is common model

Summary of Floating Point Encodings



Tiny Floating Point Example

- **8-bit Floating Point Representation**
  - the sign bit is in the most significant bit.
  - the next four bits are the exponent, with a bias of 7.
  - the last three bits are the `frac`
- **Same General Form as IEEE Format**
  - normalized, denormalized
  - representation of 0, NaN, infinity



  - Remember in this representation
    - » $V_N = -1^S \times 1.F \times 2^{E-7}$
    - » $V_{DN} = -1^S \times 0.F \times 2^{-6}$

Values Related to the Exponent

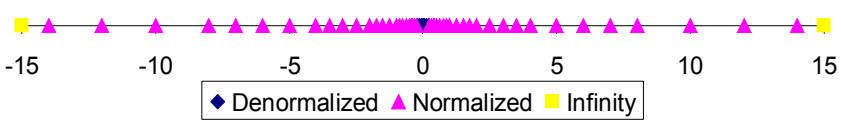
Exp	exp	E	2^E	
0	0000	-6	1/64	(denorms)
1	0001	-6	1/64	(same as denorms = smooth)
2	0010	-5	1/32	
3	0011	-4	1/16	
4	0100	-3	1/8	
5	0101	-2	1/4	
6	0110	-1	1/2	
7	0111	0	1	
8	1000	+1	2	
9	1001	+2	4	
10	1010	+3	8	
11	1011	+4	16	
12	1100	+5	32	
13	1101	+6	64	
14	1110	+7	128	
15	1111	n/a		(inf or NaN)

Dynamic Range

	s	exp	frac	E	Value	
Denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	← closest to zero
	0	0000	010	-6	2/8*1/64 = 2/512	
	…					
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	← largest denorm
Normalized numbers	0	0001	000	-6	8/8*1/64 = 8/512	← smallest norm
	0	0001	001	-6	9/8*1/64 = 9/512	
	…					
	0	0110	110	-1	14/8*1/2 = 14/16	
	0	0110	111	-1	15/8*1/2 = 15/16	← closest to 1 below
	0	0111	000	0	8/8*1 = 1	
	0	0111	001	0	9/8*1 = 9/8	← closest to 1 above
	0	0111	010	0	10/8*1 = 10/8	
	…					
	0	1110	110	7	14/8*128 = 224	
	0	1110	111	7	15/8*128 = 240	← largest norm
	0	1111	000	n/a	inf	

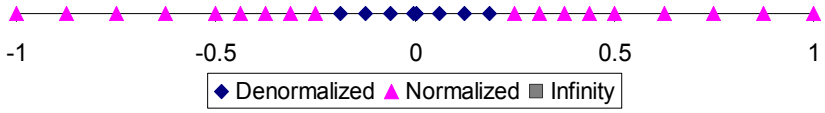
Distribution of Values

- 6-bit IEEE-like format
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is 3
- Notice how the distribution gets denser toward zero.



Distribution of Values (close-up view)

- **6-bit IEEE-like format**
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is 3



Interesting Numbers

Description	exp	frac	Numeric Value
Zero	00…00	00…00	0.0
Smallest Pos. Denorm.	00…00	00…01	$2^{-\{23,52\}} \times 2^{-\{126,1022\}}$
▪ Single ≈ 1.4×10^{-45}			
▪ Double ≈ 4.9×10^{-324}			
Largest Denormalized	00…00	11…11	$(1.0 - \varepsilon) \times 2^{-\{126,1022\}}$
▪ Single ≈ 1.18×10^{-38}			
▪ Double ≈ 2.2×10^{-308}			
Smallest Pos. Normalized	00…01	00…00	$1.0 \times 2^{-\{126,1022\}}$
▪ Just larger than largest denormalized			
One	01…11	00…00	1.0
Largest Normalized	11…10	11…11	$(2.0 - \varepsilon) \times 2^{\{127,1023\}}$
▪ Single ≈ 3.4×10^{38}			
▪ Double ≈ 1.8×10^{308}			

Special Encoding Properties

- FP Zero Same as Integer Zero
  - All bits = 0
    - » note anomaly with negative zero
    - » fix is to ignore the sign bit on a zero compare
- Can (Almost) Use Unsigned Integer Comparison
  - Must first compare sign bits
  - Must consider -0 = 0
  - NaNs problematic
    - » Will be greater than any other values
    - » What should comparison yield?
  - Otherwise OK
    - » Denorm vs. normalized
    - » Normalized vs. infinity
    - » due to monotonicity property of floats

Rounding

- IEEE 754 provides 4 modes
  - application/user choice
    - » mode bits indicate choice in some register
      - from C - library support is needed
    - » control of numeric stability is the goal here
  - modes
    - » unbiased: round towards nearest
      - if in middle then round towards the even representation
    - » truncation: round towards 0 (+0 for pos, -0 for neg)
    - » round up: towards +inifinity
    - » round down: towards – infinity
  - underflow
    - » rounding from a non-zero value to 0 ➔ underflow
    - » denormals minimize this case
  - overflow
    - » rounding from non-infinite to infinite value

Floating Point Operations

- **Conceptual View**
  - First compute exact result
  - Make it fit into desired precision
    - » Possibly overflow if exponent too large
    - » Possibly round to fit into `frac`
- **Rounding Modes (illustrate with $ rounding)**

	$1.40	$1.60	$1.50	$2.50	–$1.50
Zero	$1	$1	$1	$2	–$1
Round down (-∞)	$1	$1	$1	$2	–$2
Round up (+∞)	$2	$2	$2	$3	–$1
Nearest Even (default)	$1	$2	$2	$2	–$2

Note:
1. Round down: rounded result is close to but no greater than true result.
2. Round up: rounded result is close to but no less than true result.

Closer Look at Round-To-Even

- **Default Rounding Mode**
  - Hard to get any other kind without dropping into assembly
  - All others are statistically biased
    - » Sum of set of positive numbers will consistently be over- or under-estimated
- **Applying to Other Decimal Places / Bit Positions**
  - When exactly halfway between two possible values
    - » Round so that least significant digit is even
  - E.g., round to nearest hundredth

1.2349999	1.23	(Less than half way)
1.2350001	1.24	(Greater than half way)
1.2350000	1.24	(Half way—round up)
1.2450000	1.24	(Half way—round down)

Rounding Binary Numbers

- **Binary Fractional Numbers**
  - "Even" when least significant bit is 0
  - Half way when bits to right of rounding position = $100\ldots_2$
- **Examples**
  - Round to nearest 1/4 (2 bits right of binary point)

Value	Binary	Rounded	Action	Rounded Value
2 3/32	10.00011_2	10.00_2	(<1/2—down)	2
2 3/16	10.00110_2	10.01_2	(>1/2—up)	2 1/4
2 7/8	10.11100_2	11.00_2	(1/2—up)	3
2 5/8	10.10100_2	10.10_2	(1/2—down)	2 1/2

Guard Bits and Rounding

- **Clearly extra fraction bits are required to support rounding modes**
  - e.g. given a 23 bit representation of the mantissa in single precision
  - you round to 23 bits for the answer but you'll need extra bits in order to know what to do
- **Question: How many do you really need?**
  - why?

Rounding in the Worst Case

- Basic algorithm for add
  - subtract exponents to see which one is bigger d=Ex - Ey
  - usually swapped if necessary so biggest one is in a fixed register
  - alignment step
    - » shift smallest significand d positions to the right
    - » copy largest exponent into exponent field of the smallest
  - add or subtract signifcands
    - » add if signs equal – subtract if they aren't
  - normalize result
    - » details next slide
  - round according to the specified mode
    - » more details soon
    - » note this might generate an overflow ➔ shift right and increment the exponent
  - exceptions
    - » exponent overflow ➔ ∞
    - » exponent underflow ➔ denormal
    - » inexact ➔ rounding was done
    - » special value 0 may also result ➔ need to avoid wrap around

Normalization Cases

- Result already normalized
  - no action needed
- On an add
  - you may end up with 2 leading bits before the "."
  - hence significand shift right one & increment exponent
- On a subtract
  - the significand may have n leading zero's
  - hence shift significand left by n and decrement exponent by n
  - note: common circuit is a L0D ::= leading 0 detector

Alignment and Normalization Issues

- During alignment
  - smaller exponent arg gets significand right shifted
    - » ➔ need for extra precision in the FPU
      - the question is again how much extra do you need?
      - Intel maintains 80 bits inside their FPU's – an 8087 legacy
- During normalization
  - a left shift of the significand may occur
- During the rounding step
  - extra internal precision bits (guard bits) get dropped
- Time to consider how many guard bits we need
  - to do rounding properly
  - to compensate for what happens during alignment and normalization

For Effective Addition

- Result
  - either normalized
  - or generates 1 additional integer bit
    - » hence right shift of 1
    - » ➔ need for f+1 bits
    - » extra bit is called the *rounding* bit
- Alignment throws a bunch of bits to the right
  - need to know whether they were all 0 or not
  - ➔ hence 1 more bit called the *sticky* bit
    - » sticky bit value is the OR of the discarded bits

For Effective Subtraction

- There are 2 subcases
  - if the difference in the two exponents is larger than 1
    - » alignment produces a mantissa with more than 1 leading 0
    - » hence result is either normalized or has one leading 0
      - in this case a left shift will be required in normalization
      - ➔ an extra bit is needed for the fraction plus you still need the rounding bit
      - this extra bit is called the *guard* bit
    - » also during subtraction a borrow may happen at position f+2
      - this borrow is determined by the sticky bit
  - the difference of the two exponents is 0 or 1
    - » in this case the result may have many more than 1 leading 0
    - » but at most one nonzero bit was shifted during normalization
      - hence only one additional bit is needed for the subtraction result
      - but the borrow to the extra bit may still happen

Answer to the Guard Bit Puzzle

- You need at least 3 extra bits
  - Guard, Round, Sticky
- Hence minimal internal representation is

Round to Nearest

- Add 1 to low order fraction bit L
  - if G=1 and R and S are *NOT* both zero
- However a halfway result gets rounded to even
  - e.g. if G=1 and R and S are both zero
- Hence
  - let rnd be the value added to L
  - then
    - » rnd = G(R+S) + G(R+S)' = G(L+R+S)
- OR
  - always add 1 to position G which effectively rounds to nearest
    - » but doesn't account for the halfway result case
  - and then
    - » zero the L bit if G(R+S)'=1

The Other Rounding Modes

- Round towards Zero
  - this is simple truncation
    - » so ignore G, R, and S bits
- Rounding towards ± ∞
  - in both cases add 1 to L if G, R, and S are not all 0
  - if sgn is the sign of the result then
    - » rnd = sgn'(G+R+T) for the + ∞ case
    - » rnd = sgn(G+R+T) for the - ∞ case
- Finished
  - you know how to round

Floating Point Operations

- **Multiply/Divide**
  - similar to what you learned in school
    - » add/sub the exponents
    - » multiply/divide the mantissas
    - » normalize and round the result
  - tricks exist of course
- **Add/Sub**
  - find largest
  - make smallest exponent = largest and shift mantissa
  - add the mantissa's (smaller one may have gone to 0)
- **Problems?**
  - consider all the types: numbers, NaN's, 0's, infinities, and denormals
  - what sort of exceptions exist

FP Multiplication

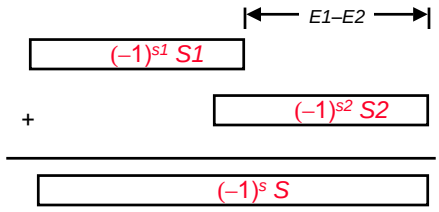
- **Operands**

  $(-1)^{s1}\ S1\ \ 2^{E1}$ * $(-1)^{s2}\ S2\ \ 2^{E2}$
- **Exact Result**

  $(-1)^{s}\ S\ \ 2^{E}$
  - Sign *s*: *s1* XOR *s2*
  - Significand *S*: *S1* * *S2*
  - Exponent *E*: *E1* + *E2*
- **Fixing: Post operation normalization**
  - If $M \geq 2$, shift *S* right, increment *E*
  - If *E* out of range, overflow
  - Round *S* to fit `frac` precision
- **Implementation**
  - Biggest chore is multiplying significands
    - » same as unsigned integer multiply which you know all about

FP Addition

- **Operands**
  - $(-1)^{s1}\ M1\ \ 2^{E1}$
  - $(-1)^{s2}\ M2\ \ 2^{E2}$
  - **Assume** $E1 > E2$ ➔ *sort*
- **Exact Result**
  - $(-1)^{s}\ M\ \ 2^{E}$
  - **Sign** s**, significand** S**:**
    - » **Result of signed align & add**
  - **Exponent** E**:** $E1$
- **Fixing**
  - **If** $S \geq 2$**, shift** S **right, increment** E
  - **if** $S < 1$**, shift** S **left** k **positions, decrement** E **by** k
  - **Overflow if** E **out of range**
  - **Round** S **to fit** `frac` **precision**



Mathematical Properties of FP Add

- **Compare to those of Abelian Group (i.e. commutative group)**
  - **Closed under addition?** **YES**
    - » **But may generate infinity or NaN**
  - **Commutative?** **YES**
  - **Associative?** **NO**
    - » **Overflow and inexactness of rounding**
  - **0 is additive identity?** **YES**
  - **Every element has additive inverse** **ALMOST**
    - » **Except for infinities & NaNs**
- **Monotonicity**
  - $a \geq b \Rightarrow a+c \geq b+c$**?** **ALMOST**
    - » **Except for infinities & NaNs**

Math Properties of FP Mult

- **Compare to Commutative Ring**
  - Closed under multiplication? YES
    - » But may generate infinity or NaN
  - Multiplication Commutative? YES
  - Multiplication is Associative? NO
    - » Possibility of overflow, inexactness of rounding
  - 1 is multiplicative identity? YES
  - Multiplication distributes over addition? NO
    - » Possibility of overflow, inexactness of rounding
- **Monotonicity**
  - $a \geq b \ \& c \geq 0 \Rightarrow a * c \geq b * c$? ALMOST
    - » Except for infinities & NaNs

The 5 Exceptions

- **Overflow**
  - generated when an infinite result is generated
- **Underflow**
  - generated when a 0 is generated after rounding a non-zero result
- **Divide by 0**
- **Inexact**
  - set when overflow or rounded value
  - DOH! the common case so why is an exception??
    - » some nitwit wanted it so it's there
    - » there is a way to mask it as an exception of course
- **Invalid**
  - result of bizarre stuff which generates a NaN result
    - » sqrt(-1)
    - » 0/0
    - » inf - inf

Floating Point in C

- **C Guarantees Two Levels**
  - `float` single precision
  - `double` double precision
- **Conversions**
  - Casting between `int`, `float`, and `double` changes numeric values
  - `Double` or `float` to `int`
    - » Truncates fractional part
    - » Like rounding toward zero
    - » Not defined when out of range
      - Generally saturates to TMin or TMax
  - `int` to `double`
    - » Exact conversion, as long as int has ≤ 53 bit word size
  - `int` to `float`
    - » Will round according to rounding mode

Floating Point Puzzles

```
int x = …;
float f = …;
double d = …;
```

Assume neither `d` nor `f` is NAN

- `x == (int)(float) x` ?
- `x == (int)(double) x`
- `f == (float)(double) f`
- `d == (float) d`
- `f == -(-f);`
- `2/3 == 2/3.0`
- `d < 0.0 ⇒ ((d*2) < 0.0)`
- `d > f ⇒ -f < -d`
- `d * d >= 0.0`
- `(d+f)-d == f`

Floating Point Puzzles

```
int x = …;
float f = …;
double d = …;
```

Assume neither `d` nor `f` is NAN

- `x == (int)(float) x` NO – 24 bit significand
- `x == (int)(double) x` ?
- `f == (float)(double) f`
- `d == (float) d`
- `f == -(-f);`
- `2/3 == 2/3.0`
- `d < 0.0 ⇒ ((d*2) < 0.0)`
- `d > f ⇒ -f < -d`
- `d * d >= 0.0`
- `(d+f)-d == f`

Floating Point Puzzles

```
int x = …;
float f = …;
double d = …;
```

Assume neither `d` nor `f` is NAN

- `x == (int)(float) x` NO – 24 bit significand
- `x == (int)(double) x` YES – 53 bit significand
- `f == (float)(double) f` ?
- `d == (float) d`
- `f == -(-f);`
- `2/3 == 2/3.0`
- `d < 0.0 ⇒ ((d*2) < 0.0)`
- `d > f ⇒ -f < -d`
- `d * d >= 0.0`
- `(d+f)-d == f`

Floating Point Puzzles

```
int x = …;
float f = …;
double d = …;
```

Assume neither `d` nor `f` is NAN

- `x == (int)(float) x` NO – 24 bit significand
- `x == (int)(double) x` YES – 53 bit significand
- `f == (float)(double) f` YES – increases precision
- `d == (float) d` ?
- `f == -(-f);`
- `2/3 == 2/3.0`
- `d < 0.0 ⇒ ((d*2) < 0.0)`
- `d > f ⇒ -f < -d`
- `d * d >= 0.0`
- `(d+f)-d == f`

Floating Point Puzzles

```
int x = …;
float f = …;
double d = …;
```

Assume neither `d` nor `f` is NAN

- `x == (int)(float) x` NO – 24 bit significand
- `x == (int)(double) x` YES – 53 bit significand
- `f == (float)(double) f` YES – increases precision
- `d == (float) d` NO – loses precision
- `f == -(-f);` ?
- `2/3 == 2/3.0`
- `d < 0.0 ⇒ ((d*2) < 0.0)`
- `d > f ⇒ -f < -d`
- `d * d >= 0.0`
- `(d+f)-d == f`

Floating Point Puzzles

```
int x = …;
float f = …;
double d = …;
```

Assume neither `d` nor `f` is NAN

- `x == (int)(float) x` NO – 24 bit significand
- `x == (int)(double) x` YES – 53 bit significand
- `f == (float)(double) f` YES – increases precision
- `d == (float) d` NO – loses precision
- `f == -(-f);` YES – sign bit inverts twice
- `2/3 == 2/3.0` ?
- `d < 0.0 ⇒ ((d*2) < 0.0)`
- `d > f ⇒ -f < -d`
- `d * d >= 0.0`
- `(d+f)-d == f`

Floating Point Puzzles

```
int x = …;
float f = …;
double d = …;
```

Assume neither `d` nor `f` is NAN

- `x == (int)(float) x` NO – 24 bit significand
- `x == (int)(double) x` YES – 53 bit significand
- `f == (float)(double) f` YES – increases precision
- `d == (float) d` NO – loses precision
- `f == -(-f);` YES – sign bit inverts twice
- `2/3 == 2/3.0` NO – 2/3 = 0
- `d < 0.0 ⇒ ((d*2) < 0.0)` ?
- `d > f ⇒ -f < -d`
- `d * d >= 0.0`
- `(d+f)-d == f`

Floating Point Puzzles

```
int x = …;
float f = …;
double d = …;
```

Assume neither `d` nor `f` is NAN

- `x == (int)(float) x` NO – 24 bit significand
- `x == (int)(double) x` YES – 53 bit significand
- `f == (float)(double) f` YES – increases precision
- `d == (float) d` NO – loses precision
- `f == -(-f);` YES – sign bit inverts twice
- `2/3 == 2/3.0` NO – 2/3 = 0
- `d < 0.0 ⇒ ((d*2) < 0.0)` YES - monotonicity
- `d > f ⇒ -f < -d` ?
- `d * d >= 0.0`
- `(d+f)-d == f`

Floating Point Puzzles

```
int x = …;
float f = …;
double d = …;
```

Assume neither `d` nor `f` is NAN

- `x == (int)(float) x` NO – 24 bit significand
- `x == (int)(double) x` YES – 53 bit significand
- `f == (float)(double) f` YES – increases precision
- `d == (float) d` NO – loses precision
- `f == -(-f);` YES – sign bit inverts twice
- `2/3 == 2/3.0` NO – 2/3 = 0
- `d < 0.0 ⇒ ((d*2) < 0.0)` YES - monotonicity
- `d > f ⇒ -f < -d` YES - monotonicity
- `d * d >= 0.0` ?
- `(d+f)-d == f`

Floating Point Puzzles

```
int x = …;
float f = …;
double d = …;
```

Assume neither `d` nor `f` is NAN

- `x == (int)(float) x` NO – 24 bit significand
- `x == (int)(double) x` YES – 53 bit significand
- `f == (float)(double) f` YES – increases precision
- `d == (float) d` NO – loses precision
- `f == -(-f);` YES – sign bit inverts twice
- `2/3 == 2/3.0` NO – 2/3 = 0
- `d < 0.0 ⇒ ((d*2) < 0.0)` YES - monotonicity
- `d > f ⇒ -f < -d` YES - monotonicity
- `d * d >= 0.0` YES - monotonicity
- `(d+f)-d == f` ?

Floating Point Puzzles

```
int x = …;
float f = …;
double d = …;
```

Assume neither `d` nor `f` is NAN

- `x == (int)(float) x` NO – 24 bit significand
- `x == (int)(double) x` YES – 53 bit significand
- `f == (float)(double) f` YES – increases precision
- `d == (float) d` NO – loses precision
- `f == -(-f);` YES – sign bit inverts twice
- `2/3 == 2/3.0` NO – 2/3 = 0
- `d < 0.0 ⇒ ((d*2) < 0.0)` YES - monotonicity
- `d > f ⇒ -f < -d` YES - monotonicity
- `d * d >= 0.0` YES - monotonicity
- `(d+f)-d == f` NO – not associative

Gulf War: Patriot misses Scud

- By 687 meters
  - Why?
    - » clock ticks at 1/10 second – can't be represented exactly in binary
    - » clock was running for 100 hours
      - hence clock was off by .3433 sec.
    - » Scud missile travels at 2000 m/sec
      - 687 meter error = .3433 second
  - Result
    - » SCUD hits Army barracks
    - » 28 soldiers die
- Accuracy counts
  - floating point has many sources of inaccuracy - BEWARE
- Real problem
  - software updated but not fully deployed

Ariane 5

- Exploded 37 seconds after liftoff
- Cargo worth $500 million

- Why
  - Computed horizontal velocity as floating point number
  - Converted to 16-bit integer
  - Worked OK for Ariane 4
  - Overflowed for Ariane 5
    - » Used same software

Ariane 5

- From Wikipedia

Ariane 5's first test flight on 4 June 1996 failed, with the rocket self-destructing 37 seconds after launch because of a malfunction in the control software, which was arguably one of the most expensive computer bugs in history. A data conversion from 64-bit floating point to 16-bit signed integer value had caused a processor trap (operand error). The floating point number had a value too large to be represented by a 16-bit signed integer. Efficiency considerations had led to the disabling of the software handler (in Ada code) for this trap, although other conversions of comparable variables in the code remained protected.

Remaining Problems?

- Of course
- NaN's
  - 2 types defined Q and S
  - standard doesn't say exactly how they are represented
  - standard is also vague about SNaN's results
  - SNaN's cause exceptions QNaN's don't
    - » hence program behavior is different and there's a porting problem
- Exceptions
  - standard says what they are
  - but forgets to say WHEN you will see them
    - » Weitek chips ➔ only see the exception when you start the next op ➔ GRR!!

More Problems

- IA32 specific
  - FP registers are 80 bits
    - » similar to IEEE but with e=15 bits, f=63-bits, + sign
    - » 79 bits but stored in 80 bit field
      - 10 bytes
      - BUT modern x86 use 12 bytes to improve memory performance
      - e.g. 4 byte alignment model
  - the problem
    - » FP reg to memory ➔ conversion to IEEE 64 or 32 bit formats
      - loss of precision and a potential rounding step
  - C problem
    - » no control over where values are – register or memory
    - » hence hidden conversion
      - -ffloat-store gcc flag is supposed to not register floats
      - after all the x86 isn't a RISC machine
      - unfortunately there are exceptions so this doesn't work
    - » stuck with specifying long doubles if you need to avoid the hidden conversions

From the Programmer's Perspective

- Computer arithmetic is a huge field
  - lots of places to learn more
    - » your text is specialized for implementers
    - » but it's still the gold standard even for programmers
      - also contains lots of references for further study
  - from a machine designer perspective
    - » it takes a lifetime to get really good at it
    - » lots of specialized tricks have been employed
  - from the programmer perspective
    - » you need to know the representations
    - » and the basics of the operations
    - » they are visible to you
    - » understanding ➔ efficiency
- Floating point is always
  - slow and big when compared to integer circuits

Summary

- **IEEE Floating Point Has Clear Mathematical Properties**
  - Represents numbers of form $M \times 2^E$
  - Can reason about operations independent of implementation
    - » As if computed with perfect precision and then rounded
  - Not the same as real arithmetic
    - » Violates associativity/distributivity
    - » Makes life difficult for compilers & serious numerical applications programmers
- **Next time we move to FP circuits**
  - arcane but interesting in nerd-land