Digital Computer Arithmetic

Part 4
Binary Floating-point Numbers

Soo-Ik Chae

Spring 2010

Preliminaries - Representation

- Floating-point numbers - provide a dynamic range of representable real numbers without having to scale the operands
- Representation - similar to scientific notation
- Two parts - significand (or mantissa) M and exponent (or characteristic) E
- The floating-point number F represented by the pair (M,E) has the value -

$$F = M\beta^{E}$$ (β - base of exponent)

- Base - common to all numbers in a given system - implied - not included in the representation of a floating point number

Preliminaries - Precision

- n bits partitioned into two parts - significand M and exponent E
- n bits - 2^n different values
- Range between smallest and largest representable values increases ⇒ distance between any two consecutive values increases
  - Floating-point numbers sparser than fixed-point numbers - lower precision
- Real number between two consecutive floating-point numbers is mapped onto one of the two
  - A larger distance between two consecutive numbers results in a lower precision of representation

Formats

- Significand M and exponent E - signed quantities
- Exponent - usually a signed integer
- Significand - usually one of two:
  - pure fraction, or
  - a number in the range [1, 2) (for $\beta=2$)
- Representing negative values - can be different
- Until 1980 - no standard - every computer system had its own representation method
  - transporting programs/data between two different computers was very difficult
- IEEE standard 754 is now used in most floating-point arithmetic units - details later
- Few computer systems use formats differing in partitioning of the n bits, representation of each part, or value of the base β

Significand Field

- Common case - signed-magnitude fraction
- Floating-point format - sign bit **S**, **e** bits of exponent **E**, **m** bits of unsigned fraction **M** (**m+e+1=n**)

S	Exponent E	Unsigned Significand M

- Value of **(S,E,M)** : $F = (-1)^S \cdot M \cdot \beta^E$
  ($(-1)^0=1$; $(-1)^1=-1$)

- Maximal value - **Mmax = 1-ulp**
- **ulp** -Unit in the last position - weight of the least-significant bit of the fractional significand
- Usually - not always - **ulp=2^{-m}**

The Base β

- β is restricted to 2^k (k=1,2,…) - simplifies decreasing significand and increasing exponent (and vice versa) at the same time
- Whenever an arithmetic operation results in a significand larger than Mmax = 1-ulp, it is necessary that significand is reduced and exponent increased: value remains unchanged
- Smallest increase in E is 1

$$M \cdot \beta^E = (M/\beta) \cdot \beta^{E+1}$$

- M/β - a simple arithmetic shift right operation if β is an integral power of radix
- If β=r=2 - shifting significand to the right by a single position must be compensated by adding 1 to exponent

Example

- Result of an arithmetic operation - $01.10100 \cdot 2^{100}$ - significand larger than Mmax
- Significand reduced by shifting it one position to the right, exponent increased by 1
- New result - $0.11010 \cdot 2^{101}$
- If $\beta=2^k$ - changing exponent by 1 is equivalent to shifting significand by k positions
- Consequently - only k-position shifts are allowed
- If $\beta=4=2^2$
  $01.10100 \cdot 4^{010} = 0.01101 \cdot 4^{011}$

Normalized Form

- Floating point representation not unique - $0.11010 \cdot 2^{101} = 0.01101 \cdot 2^{110}$
- With E=111 - significand=0.00110 - loss of a significant digit
- Preferred representation - one with no leading zeros - maximum number of significant digits - normalized form
- Simplifies comparing floating-point numbers - a larger exponent indicates a larger number; significands compared only for equal exponents
- For $\beta=2^k$ - significand normalized if there is a nonzero bit in the first k positions
- Example: Normalized form of $0.00000110 \cdot 16^{101}$ is $0.01100000 \cdot 16^{100}$

Range of Normalized Fractions

- Range of significand is smaller than [0,1-ulp]
- Smallest and largest allowable values are
- $M_{min} = 1/\beta$; $M_{max} = 1-ulp$
- Range of normalized fractions does not include the value zero - a special representation is needed
- A possible representation for zero - M=0 and any exponent E
- E=0 is preferred - representation of zero in floating-point is identical to representation in fixed-point
  - Execution of a test for zero instruction simplified

Representation of Exponents

- Most common representation - biased exponent
- E= E^{true} + bias (bias - constant; E^{true} - the true value of the exponent represented in two's complement)
- Exponent field - e bits ; range: $-2^{e-1} \leq E^{true} \leq 2^{e-1}-1$
- Bias usually selected as magnitude of most negative exponent 2^{e-1} $0 \leq E \leq 2^{e}-1$
- Exponent represented in the excess 2^{e-1} method
- Advantages:
  - When comparing two exponents (for add/subtract operations) - sign bits ignored; comparison like unsigned numbers
  - Floating-points with S,E,M format are compared like binary integers in signed-magnitude representation
  - Smallest representable number has the exponent 0

Example: Excess 64

- $e=7$
- Range of exponents in two's complement representation is $-64 \leq E^{true} \leq 63$
- 1000000 and 0111111 represent -64 and 63
- When adding bias 64, the true values -64 and 63 are represented by 0000000 and 1111111
- This is called: excess 64 representation
- Excess 2^{e-1} representation can be obtained by
  - Inverting sign bit of two's complement representation, or
  - Letting the values 0 and 1 of the sign bit indicate negative and positive numbers, respectively

Range of Normalized Floating-Point Numbers

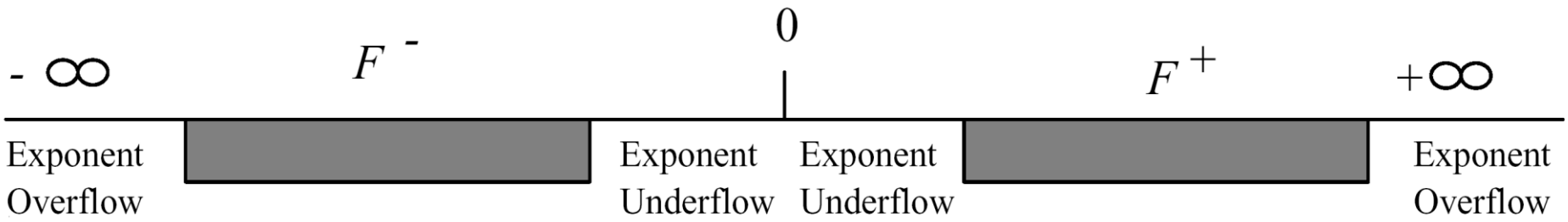
- Identical subranges for positive (F^+) and negative (F^-) numbers:

$$M_{min} \cdot \beta^{E_{min}} \leq F^+ \leq M_{max} \cdot \beta^{E_{max}}$$

  - (E_{min}, E_{max} - smallest, largest exponent)
- An exponent larger than E_{max} / smaller than E_{min} must result in an exponent overflow/underflow indication
- Significand normalized - overflow reflected in exponent
- Ways of indicating overflow:
  - Using a special representation of infinity as result
  - stopping computation and interrupting processor
  - setting result to largest representable number
- Indicating underflow:
  - Representation of zero is used for result and an underflow flag is raised - computation can proceed, if appropriate, without interruption

Range of Floating Point Numbers

- **Zero** is not included in the range of either F^+ or F^-



Example - IBM 370

- **Short floating-point format - 32 bits ; β=16**

S – sign bit	E – 7 bits, excess 64 exponent	M – 24 bits, unsigned fractional significand

$$F = (-1)^S \cdot M \cdot 16^{E-64}$$

- **Emin, Emax represented by 0000000, 1111111 - value of -64, +63**
- **Significand - six hexadecimal digits**
- **Normalized significand satisfies -**

$$M_{min} = 16^{-1} \leq M \leq M_{max} = 1 - 16^{-6} = 1 - 2^{-24}.$$

- **Consequently,**

$$F^{+}_{max} = (1 - 16^{-6}) \cdot 16^{63} \approx 7.23 \cdot 10^{75}$$

$$F^{+}_{min} = (16^{-1}) \cdot 16^{-64} \approx 5.4 \cdot 10^{-79}$$

Numerical Example - IBM 370

- $(S,E,M)=(C1200000)_{16}$ in the short IBM format - first byte is $(11000001)_2$
  - Sign bit is S=1 - number is negative
  - Exponent is 41_{16} and, bias is $64_{10}=40_{16}$, $E^{true}=(41-40)_{16}=1$
  - $M=0.2_{16}$, hence $F=(-0.0010)_2\cdot 16^1=(-2)_{10}$.
- Resolution of representation - distance between two consecutive significands -

$$ulp = 16^{-6} = 2^{-24} \approx 0.6 \cdot 10^{-7}$$

  - Short format has approximately 7 significant decimal digits
- For higher precision use the long floating-point format

sign bit	7 bits - excess 64 exponent	56 bits - unsigned fractional significand

- Range roughly the same, but resolution:

$$ulp = 16^{-14} = 2^{-56} \approx 10^{-17}$$

  - 17 instead of 7 significant decimal digits

Floating-Point Formats of Three Machines

	IBM/370	DEC/VAX	Cyber 70
Word length (double)	32 (64) bits	32 (64) bits	60 bits
Significand+{hidden bit}	24 (56) bits	$23+1$ $(55+1)$ bits	48 bits
Exponent	7 bits	8 bits	11 bits
Bias	64	128	1024
Base	16	2	2
Range of M	$\frac{1}{16} \leq M < 1$	$\frac{1}{2} \leq M < 1$	$1 \leq M < 2$
Representation of M	Signed-magnitude	Signed-magnitude	One's complement
Approximate range	$16^{63} \approx 7 \cdot 10^{75}$	$2^{127} \approx 1.9 \cdot 10^{38}$	$2^{1023} \approx 10^{307}$
Approximate resolution	$2^{-24} \approx 10^{-7}$ (10^{-17})	$2^{-24} \approx 10^{-7}$ (10^{-17})	$2^{-48} \approx 10^{-14}$

Hidden Bit

- A scheme to increase the number of bits in significand to increase precision
- For a base of 2 the normalized significand will always have a leading 1 - can be eliminated, allowing inclusion of an extra bit
- The resolution becomes ulp=2^{-24} instead of 2^{-23}
- The value of a floating-point number (S,f,E) in short DEC format is

$$(-1)^S 0.1f \cdot 2^{E-128}$$

- f - the pattern of 23 bits in significand field

Hidden Bit - representation of zero

- A zero significand field (f=0) represents the fraction $0.10_2=1/2$
- If f=0 and E=0 - with a hidden bit, this may represent the value $0.1\ 2^{0-128} = 2^{-129}$
- The floating-point number f=E=0 also represents 0 - a representation without a hidden bit
- To avoid double meaning - E=0 reserved for representing zero – so, smallest exponent for nonzero numbers is E=1
- Smallest positive number in the DEC/VAX system -

$$F^+_{min} = \tfrac{1}{2}2^{1-128} = 2^{-128}$$

- Largest positive number -

$$F^+_{max} = (1-2^{-24})\cdot 2^{255-128} = (1-2^{-24})\cdot 2^{127}$$

Floating-Point Operations

- Execution depends on format used for operands
- **Assumption:** Significands are normalized fractions in signed-magnitude representation ; exponents are biased
- Given two numbers

$$F_1 = (-1)^{S_1} \cdot M_1 \cdot \beta^{E_1 - bias} \quad ; \; F_2 = (-1)^{S_2} \cdot M_2 \cdot \beta^{E_2 - bias}$$

- Calculate result of a basic arithmetic operation yielding

$$F_3 = (-1)^{S_3} \cdot M_3 \cdot \beta^{E_3 - bias}.$$

- Multiplication and division are simpler to follow than addition and subtraction

Floating-Point Multiplication

- Significands of two operands multiplied like fixed-point numbers - exponents are added - can be done in parallel
- Sign S_3 positive if signs S_1 and S_2 are equal - negative if not
- When adding two exponents $E_1 = E_1^{True} + bias$ and $E_2 = E_2^{true} + bias$: bias must be subtracted once
- For $bias = 2^{e-1}$ (100...0 in binary) - subtracting bias is equivalent to adding bias - accomplished by complementing sign bit
- If resulting exponent E_3 is larger than E_{max} / smaller than E_{min} - overflow/underflow indication must be generated

Multiplication - postnormalization

- Multiplying significands M_1 and M_2 - M_3 must be normalized
- $1/\beta \le M_1, M_2 < 1$ - product satisfies $1/\beta^2 \le M_1 \cdot M_2 < 1$
- Significand M_3 may need to be shifted one position to the left
- Achieved by performing one base- β left shift operation - k base-2 shifts for $\beta = 2^k$ - and reducing the exponent by 1
- This is called the postnormalization step
- After this step - exponent may be smaller than E_{min} - exponent underflow indication must be generated

Floating-Point Division

- Significands divided - exponents subtracted - bias added to difference E_1-E_2
- If resulting exponent out of range - overflow or underflow indication must be generated
- Resultant significand satisfies $1/\beta \le M_1/M_2 < \beta$
- A single base-β shift right of significand + increase of 1 in exponent may be needed in postnormalization step - may lead to an overflow
- If divisor=0 - indication of division by zero generated - quotient set to $\pm\infty$
- If both divisor and dividend=0 - result undefined - in the IEEE 754 standard represented by NaN - not a number - also representing uninitialized variables and the result of $0 \cdot \infty$

Remainder in Floating-Point Division

- Fixed-point remainder - $R=X-QD$ (X, Q, D - dividend, quotient, divisor) - $|R| \leq |D|$ - generated by division algorithm (restoring or nonrestoring)
- Flp division - algorithm generates quotient but not remainder - $F1\ REM\ F2 = F1-F2 \cdot Int(F1/F2)$ ($Int(F1/F2)$ - quotient $F1/F2$ converted to integer)
- Conversion to integer - either truncation (removing fractional part) or rounding-to-nearest
- The IEEE standard uses the round-to-nearest-even mode - $|F1\ REM\ F2| \leq |F2|\ /2$
- $Int(F_1/F_2)$ as large as $\beta^{Emax-Emin}$ - high complexity
- Floating-point remainder calculated separately - only when required - for example, in argument reduction for periodic functions like sine and cosine

Floating-Point Remainder - Cont.

- **Brute-force** - continue direct division algorithm for E_1-E_2 steps
- **Problem** - E_1-E_2 can be much greater than number of steps needed to generate m bits of quotient's significand - may take an arbitrary number of clock cycles
- **Solution** - calculate remainder in software
- **Alternative** - Define a **REM-step** operation - X REM F2 - performs a limited number of divide steps (e.g., limited to number of divide steps required in a regular divide operation)
- Initial X=F1, then X=remainder of previous **REM-step** operation
- **REM-step** repeated until **remainder** $\leq$ F2/2

Addition and Subtraction

- Exponents of both operands must be equal before adding or subtracting significands
- When E1=E2 - β^{E1} can be factored out and significands M1 and M2 can be added
- Significands aligned by shifting the significand of the smaller operand |E1-E2| base-β positions to the right, increasing its exponent, until exponents are equal
- E1≥E2 - $F_1 \pm F_2 = \left((-1)^{S_1} \cdot M_1 \pm (-1)^{S_2} \cdot M_2 \cdot \beta^{-(E_1-E_2)}\right) \cdot \beta^{E_1-bias}$
- Exponent of larger number not decreased - this will result in a significand larger than 1 - a larger significand adder required

Addition/Subtraction - postnormalization

- Addition - resultant significand M (sum of two aligned significands) is in range $1/\beta \le M < 2$
- If $M>1$ - a postnormalization step - shifting significand to the right to yield M_3 and increasing exponent by one - is required (an exponent overflow may occur)
- Subtraction - Resultant significand M is in range $0 \le |M|<1$ - postnormalization step - shifting significand to left and decreasing exponent - is required if $M<1/\beta$ (an exponent underflow may occur)
- In extreme cases, the postnormalization step may require a shift left operation over all bits in significand, yielding a zero result

Example

- $F1=(0.100000)_{16} \cdot 16^{3}$; $F2=(0.FFFFFF)_{16} \cdot 16^{2}$
- Short IBM format ; calculate F1-F2

F_1	0.	1	0	0	0	0	0	$\cdot 16^3$
F_2 aligned	0.	0	F	F	F	F	F	$\cdot 16^3$
$F_1 - F_2$	0.	0	0	0	0	0	1	$\cdot 16^3$
Postnormalization	0.	1	0	0	0	0	0	$\cdot 16^{-2}$

- Significand of smaller number (F2) is shifted to the right - least-significant digit lost
- Shift is time consuming - result is wrong

Example - Cont.

- Correct result (with "unlimited" number of significand digits)

F_1	0.	1	0	0	0	0	0	0	$\cdot 16^3$
F_2 aligned	0.	0	F	F	F	F	F	F	$\cdot 16^3$
$F_1 - F_2$	0.	0	0	0	0	0	0	1	$\cdot 16^3$
Postnormalization	0.	1	0	0	0	0	0	0	$\cdot 16^{-3}$

- Error (also called loss of significance) is

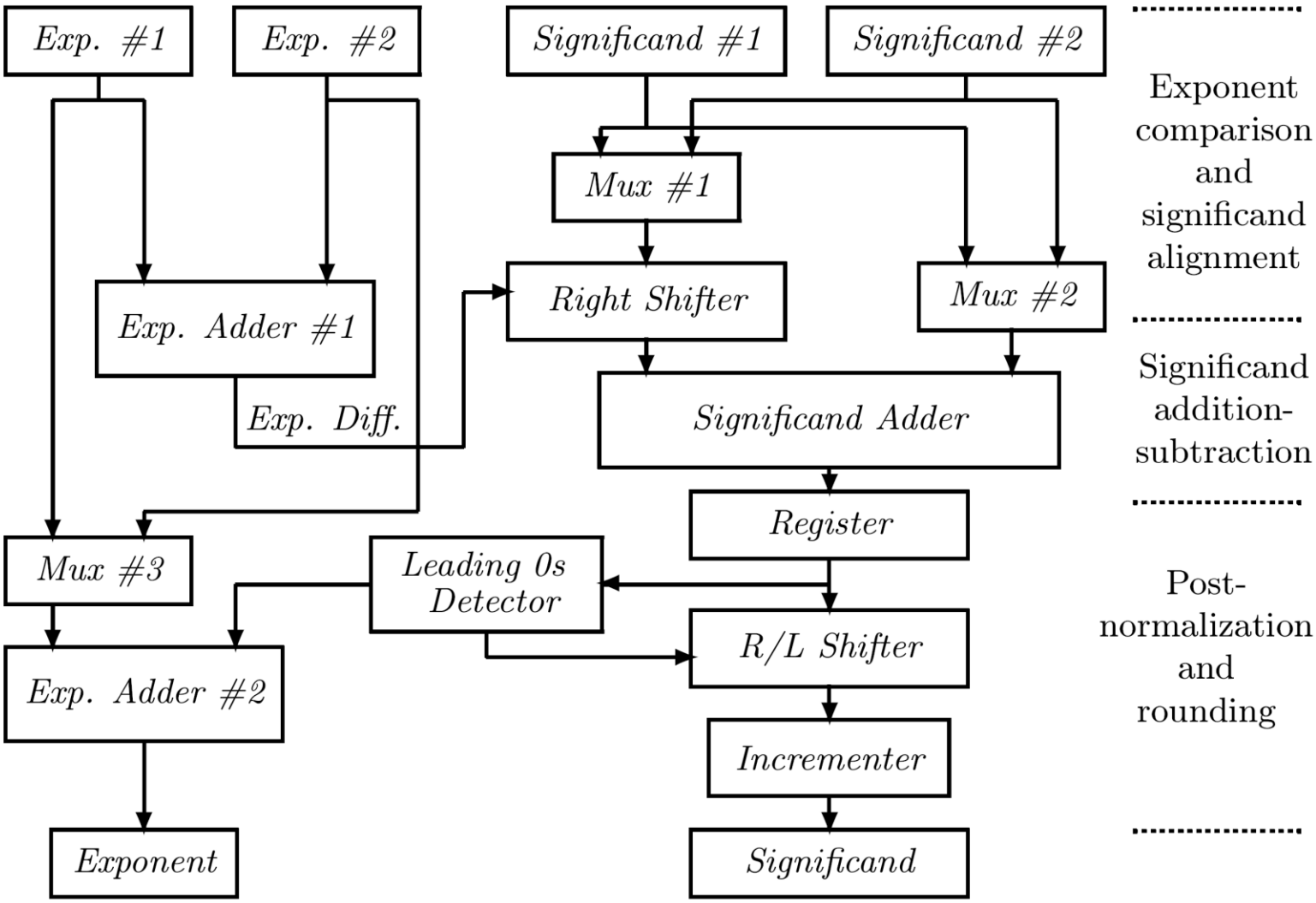
$$0.1 \cdot 16^{-2} - 0.1 \cdot 16^{-3} = 0.F \cdot 16^{-3}$$

- Solution to problem - guard digits - additional digits to the right of the significand to hold shifted-out digits
- In example - a single (hexadecimal) guard digit is sufficient

Steps in Addition/Subtraction of Floating-Point Numbers

- **Step 1:** Calculate difference d of the two exponents - $d=|E1 - E2|$
- **Step 2:** Shift significand of smaller number by d base-β positions to the right
- **Step 3:** Add aligned significands and set exponent of result to exponent of larger operand
- **Step 4:** Normalize resultant significand and adjust exponent if necessary
- **Step 5:** Round resultant significand and adjust exponent if necessary

Circuitry for Addition/Subtraction



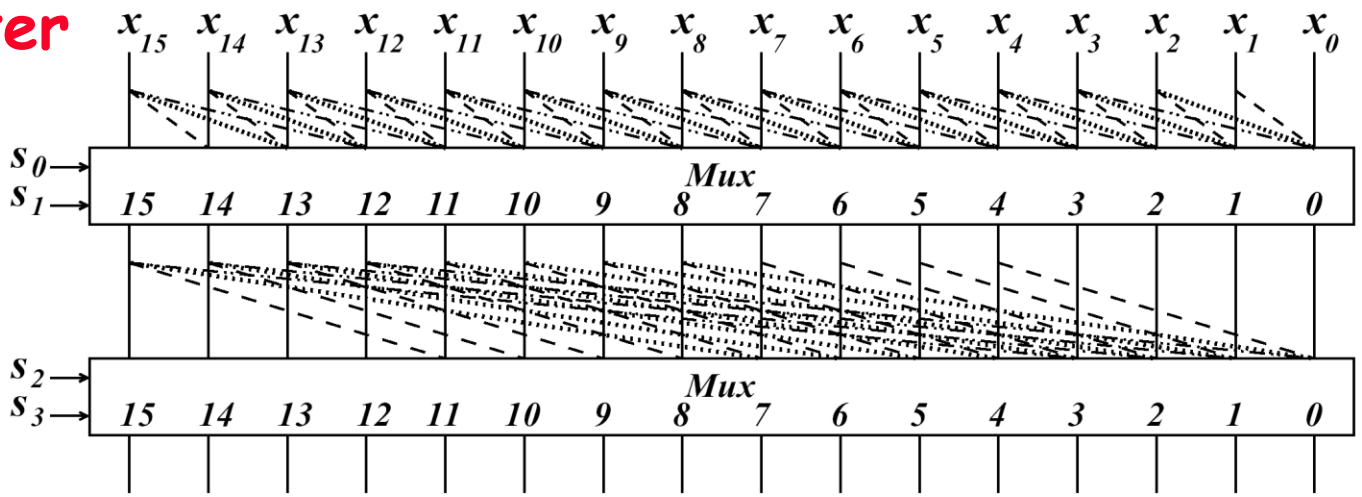
Shifters

- 1st shifter - right (alignment) shifts only ; 2nd shifter - right or left (postnormalization) shifts ; both perform large shifts (# of significand digits)
- Combinatorial shifter - generate all possible shifted patterns - only one at output according to control bits
  - Such shifters capable of circular shifts (rotates) - known as barrel shifters
  - Shift registers require a large and variable number of clock cycles, thus combinatorial shifters commonly used
- If implemented as a single level array - each input bit is directly connected to m (or more) output lines - conceptually simple design
- For m=53 (number of significand bits in IEEE double-precision format) - large number of connections (and large electrical load) - bad solution

Two levels Barrel Shifters

- first level shifts bits by 0, 1, 2 or 3 bit positions
- second level shift bits by multiples of 4 (0,4,8,...,52)
- shifts between 0 and 53 can be performed

- **Radix-4 shifter**
- **16 bits**



- 1st level - each bit has 4 destinations ; 2nd level - each bit has 14 destinations - unbalanced

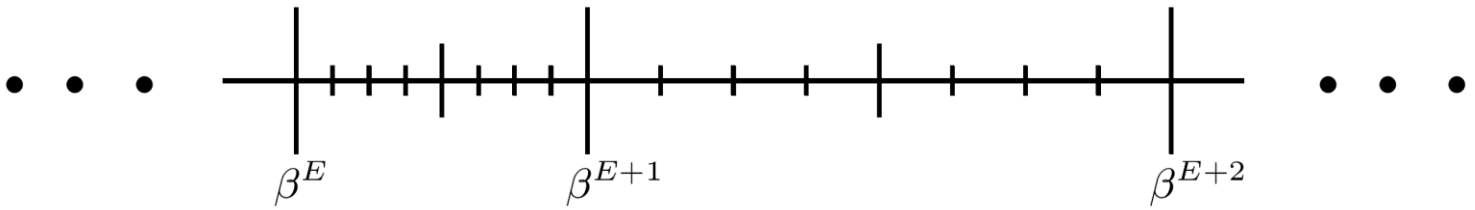
- **Radix-8 shifter** - 1st level shifts 0 to 7 bit positions ; 2nd level shifts by multiples of 8 (0,8,16,24,...,48)
  - 1st level - each bit has 8 destinations ; 2nd level - each bit has 7 destinations

Choice of Floating-Point Representation

- IEEE standard 754 commonly used - important to understand implications of a particular format
- Given n - total number of bits - determine
  - m - length of significand field
  - e - length of exponent field (m+e+1=n)
  - β - value of exponent base
- Representation error - error made when using a finite-length floating-point format to represent a high-precision real number
- x - a real number ; Fl(x) - its machine representation
- Goal when selecting format - small representation error
- Error can be measured in several ways

Measuring Representation Error

- Every real number x has two consecutive representations $F1$ and $F2$ satisfying $F1 \le x \le F2$
- $Fl(x)$ can be set to either $F1$ or $F2$
- $Fl(x)-x$ - absolute representation error
- $\delta(x)=(Fl(x)-x)/x$ - relative representation error
- If $F1=M\beta^E$ then $F2=(M+ulp)\beta^E$
- Maximum absolute error = half distance between $F1$ and $F2$ = $ulp\cdot\beta^E$ - increases as exponent increases



Measure of Representation Accuracy

- MRRE - maximum relative representation error - upper bound of $\delta(x)$

$$\delta(x) \leq \frac{\frac{1}{2}\, ulp\, \beta^E}{M\beta^E} = \frac{1}{2}\frac{ulp}{M} \leq \frac{1}{2}\frac{ulp}{\frac{1}{\beta}} = \frac{1}{2}\, ulp \cdot \beta$$

- MRRE increases with exponent base β - decreases with ulp (or number of significand bits m)
- Good measure if operands uniformly distributed
- In practice - larger significands less likely to occur
- First digit of a decimal floating-point operand will most likely be a 1; 2 is the second most likely
- Operands follow the density function

$$\frac{1}{M \ln \beta}\ ; \qquad 1/\beta \leq M \leq 1$$

Different Accuracy Measure

- ARRE - average relative representation error
- Absolute error varies between 0 and 1/2 ulp·β^E
- Average absolute error is 1/4 ulp·β^E
- Relative error is 1/4 ulp/M

$$\mathrm{ARRE} = \int_{\frac{1}{\beta}}^{1} \frac{1}{M \ln \beta} \frac{ulp}{4M}\, dM = \frac{\beta - 1}{\ln \beta} \frac{ulp}{4}$$

Range of Representation

- The range of the positive floating-point numbers - β^{Emax} - must be considered when selecting a floating-point format
- For a large range - increase β and/or number of exponent bits e
- Increasing β increases representation error
- Increasing e decreases m and increases ulp - higher representation error
- Trade-off between range and representation error

Range - Accuracy Trade-off

β	e	m	Range	MRRE	ARRE
2	9	22	$2^{2^8-1} = 2^{255}$	$0.5 \cdot 2^{-22} \cdot 2 = 2^{-22}$	$0.180 \cdot 2^{-21}$
4	8	23	$4^{2^7-1} = 2^{2^8-2} = 2^{254}$	$0.5 \cdot 2^{-23} \cdot 4 = 2^{-22}$	$0.135 \cdot 2^{-21}$
16	7	24	$16^{2^6-1} = 2^{2^8-4} = 2^{252}$	$0.5 \cdot 2^{-24} \cdot 16 = 2^{-21}$	$0.169 \cdot 2^{-21}$

- If several floating-point representations have same range - select smallest MRRE or ARRE
- If several representations have same MRRE (or ARRE) - select the largest range
- Example: 32-bit word - m+e=31 - all three representations have about the same range
- Using MRRE as measure - β=16 inferior to other two
- Using ARRE as measure - β=4 is best
- β=2 + hidden bit reduces MRRE and ARRE by a factor of 2 - the smallest representation error

Execution Time of Floating-Point Operations

- One more consideration when selecting a format
- Two time-consuming steps - aligning of significands before add/subtract operations ; postnormalization in any floating-point operation
- Observation - larger β - higher probability of equal exponents in add/subtract operations - no alignment ; lower probability that a postnormalization step needed
- No postnormalization in 59.4% of cases for β=2; 82.4% for β=16

Alignment shift	$\beta = 16$	$\beta = 2$
0	47.3%	32.6%
1	26.0%	12.1%
≥ 2	26.7%	55.3%

- This is of limited practical significance when a barrel shifter is used

The IEEE Floating-Point Standard

- Four formats for floating-point numbers
- First two:
  - basic single-precision 32-bit format and
  - double-precision 64-bit format
- Other two - extended formats for intermediate results
- Single extended format - at least 44 bits
- Double extended format - at least 80 bits
- Higher precision and range than corresponding 32- and 64-bit formats

Single-Precision Format

- Most important objective - precision of representation
- Base 2 allows a hidden bit - similar to DEC format
- Exponent field of length 8 bits for a reasonable range

S	8 bits - biased exponent E	23 bits - unsigned fraction f

- 256 combinations of 8 bits in exponent field
  - E=0 reserved for zero (with fraction f=0) and denormalized numbers (with fraction $f \neq 0$)
  - E=255 reserved for $\pm\infty$ (with fraction f=0) and NaN (with fraction $f \neq 0$)
- For $1 < E < 254$ -

$$F = (-1)^S \; 1.f \; 2^{E-127}.$$

IEEE vs. DEC

- Exponent bias - 127 instead of $2^{e-1} = 2^7 = 128$
- Larger maximum value of true exponent - 254-127=127 instead of 254-128=126 - larger range
- Similar effect - significand of 1.f instead of 0.1f -
- Largest and smallest positive numbers -

$$F^+_{max} = (2 - 2^{-23}) \cdot 2^{254-127} = (1 - 2^{-24}) \cdot 2^{128}$$

$$F^+_{min} = 1.0 \cdot 2^{1-127} = 2^{-126}$$

- instead of

$$F^+_{max} = (1-2^{-24}) \cdot 2^{127} \text{ and } F^+_{min} = 2^{-128}$$

- Exponent bias and significand range selected to allow reciprocal of all normalized numbers (in particular, F^+_{min}) to be represented without overflow - not true in DEC format

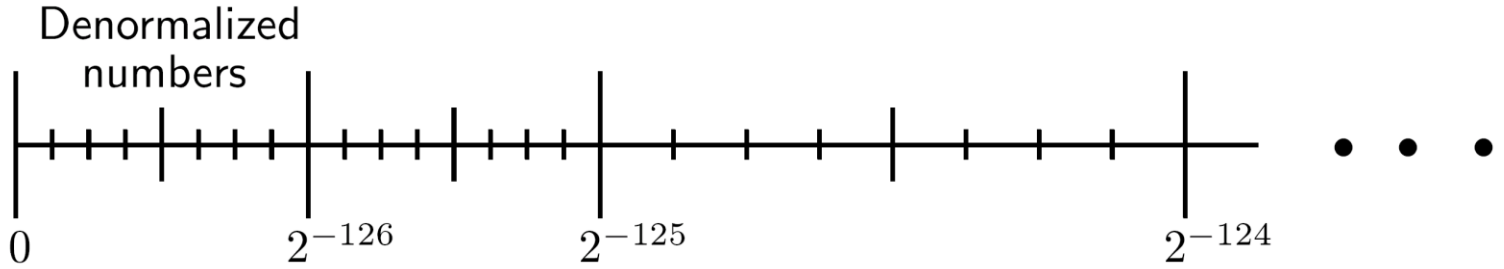
Special Values in IEEE Format

	$f = 0$	$f \neq 0$
$E = 0$	0	Denormalized
$E = 255$	$\pm\infty$	NaN

- $\pm\infty$ - represented by f=0, E=255, S=0,1 - must obey all mathematical conventions: $F+\infty=\infty$, $F/\infty=0$
- Denormalized numbers - represented by E=0 - values smaller than smallest normalized number - lowering probability of exponent underflow
- $F=(-1)^S \cdot 0.f \cdot 2^{-126}$
- Or - $F=(-1)^S \cdot 0.f \cdot 2^{1-127}$ - same bias as normalized numbers

Denormalized Numbers

- No hidden bit - significands not normalized
- Exponent - -126 selected instead of 0-127=-127 - smallest normalized number is $F^{+}min = 1 \cdot 2^{-126}$
- Smallest representable number is $2^{-23} \cdot 2^{-126} = 2^{-149}$ instead of 2^{-126} - gradual (or graceful) underflow
- Does not eliminate underflow - but reduces gap between smallest representable number and zero; 2^{-149} = distance between any two consecutive denormalized numbers = distance between two consecutive normalized numbers with smallest exponent 1-127=-126



Denormals & Extended formats

- Denormalized numbers not included in all designs of arithmetic units that follow the IEEE standard
  - Their handling is different requiring a more complex design and longer execution time
  - Even designs that implement them allow programmers to avoid their use if faster execution is desired
- The single-extended format for intermediate results within evaluation of complex functions like transcendental and powers
- Extends exponent from 8 to 11 bits and significand from 23+1 to 32 or more bits (no hidden bit)
  - Total length is at least 1+11+32=44 bits

NaN (E=255)

- f≠0 - large number of values
  - Two kinds - signaling (or trapping), and quiet (nontrapping) - differentiated by most significant bits of fraction - remaining bits contain system-dependent information
  - Example of a signaling NaN - uninitialized variable
  - It sets Invalid operation exception flag when arithmetic operation on this NaN is attempted ; Quiet NaN - does not
  - Turns into quiet NaN when used as operand if Invalid operation trap is disabled (avoid setting Invalid Op flag later)
  - Quiet NaN produced when invalid operation ($0 \cdot \infty$) attempted - this operation had already set the Invalid Op flag once. Fraction field may contain a pointer to offending code line
  - Quiet NaN, as operand will produce quiet NaN result and not set exception. For example, NaN+5=NaN. If both operands quiet NaNs, result is the NaN with smallest significand

Double-Precision Format

- Main consideration - range; exponent field - 11 bits

S	11 bits - biased exponent E	52 bits - unsigned fraction f

- E=0,2047 reserved for same purposes as in single-precision format
- For $1 \le E \le 2046$ - $F = (-1)^S \; 1.f \; 2^{E-1023}$
- Double extended format - exponent field - 15 bits, significand field - 64 or more bits (no hidden bit), total number of bits - at least 1+15+64=80

	Single	Double
Word length	32 bits	64 bits
Fraction + hidden bit	$23+1$ bits	$52+1$ bits
Exponent	8 bits	11 bits
Bias	127	1023
Approximate range	$2^{128} \approx 3.8 \cdot 10^{38}$	$2^{1024} \approx 9 \cdot 10^{307}$
Smallest normalized number	$2^{-126} \approx 10^{-38}$	$2^{-1022} \approx 10^{-308}$
Approximate resolution	$2^{-23} \approx 10^{-7}$	$2^{-52} \approx 10^{-15}$

Round-off Schemes

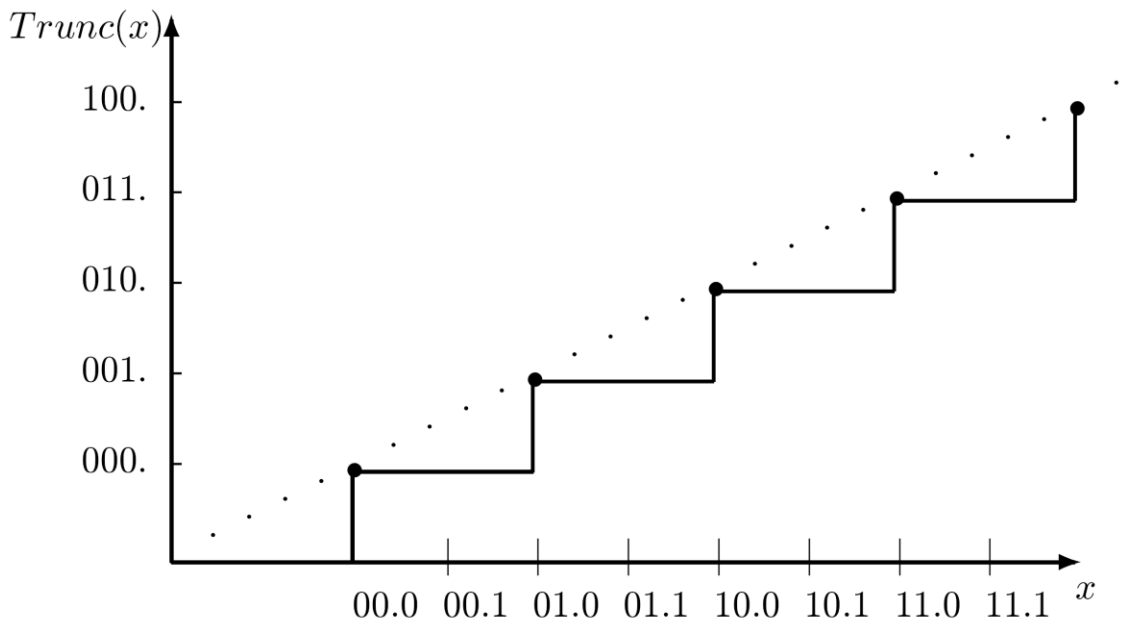
- Accuracy of results in floating-point arithmetic is limited even if intermediate results are accurate
- Number of computed digits may exceed total number of digits allowed by format - extra digits must be disposed of before storing
- Example - multiplying two significands of length m - product of length 2m - must be rounded off to m digits
- Considerations when selecting a round-off scheme -
  - Accuracy of results (numerical considerations)
  - Cost of implementation and speed (machine considerations)

Requirements for Rounding

- x,y - real numbers; Fl - set of machine representations in a given floating-point format; Fl(x) - machine representation of x
- Conditions for rounding:
  - $Fl(x) \leq Fl(y)$ for $x \leq y$
  - If $x \in Fl$ - $Fl(x)=x$
  - If F1, F2 consecutive in Fl and $F1 \leq x \leq F2$, then either $Fl(x)=F1$ or $Fl(x)=F2$
- d - number of extra digits kept in arithmetic unit (in addition to m significand digits) before rounding
- Assumption - radix point between m most significant digits (of significand) and d extra digits
- Example - Rounding 2.99_{10} into an integer

Truncation (Chopping)

- d extra digits removed - no change in m remaining digits - rounding towards zero
- For F1 ≤ x ≤ F2 - Trunc(x) results in F1 (Trunc(2.99)=2)
- Fast method - no extra hardware
- Poor numerical performance - Error up to ulp
- Trunc(x) lies entirely below ideal dotted line (infinite precision)



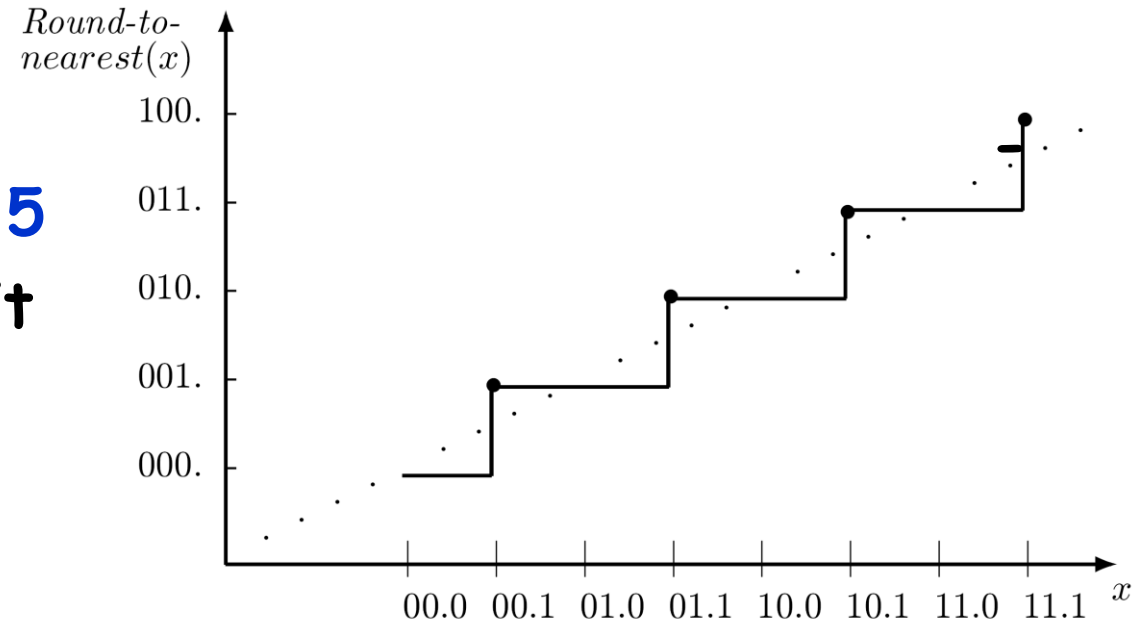
Rounding Bias

- Rounding bias - measures tendency of a round-off scheme towards errors of a particular sign
- Ideally - scheme is unbiased or has a small bias
- Truncation has a negative bias
- Definition - Error=Trunc(x)-x ; for a given d - bias is average error for a set of 2^d consecutive numbers with a uniform distribution
- Example - Truncation, d=2
- X is any significand of length m
- Sum of errors for all 2^d =4 consecutive numbers=-3/2
- Bias=average error=-3/8

Number	$Trunc(x)$	Error
$X.00$	X	0
$X.01$	X	$-1/4$
$X.10$	X	$-1/2$
$X.11$	X	$-3/4$

Round to Nearest Scheme

- $F1 \le x \le F2$ - Round(x)=nearest to x out of F1,F2 - used in many arithmetic units
- Obtained by adding 0.1_2 (half a ulp) to x and retaining the integer (chopping fraction)
- Example - x=2.99 - adding 0.5 and chopping off fractional part of 3.49 results in 3
- Maximum error - x=2.50 - 2.50+0.50=3.00 result=3, error=0.5
- A single extra digit (d=1) is sufficient



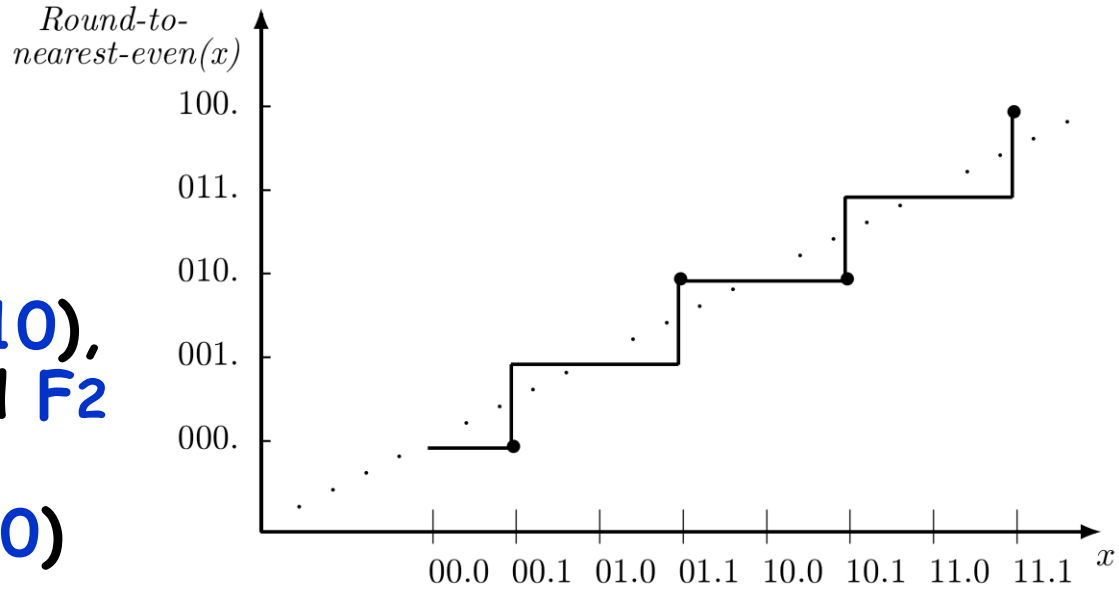
Bias of Round to Nearest

- Round(x) - nearly symmetric around ideal line - better than truncation
- Slight positive bias - due to round up of X.10
- d=2 :

Number	*Round-to-nearest(x)*	Error
$X.00$	X	0
$X.01$	X	$-1/4$
$X.10$	$X + 1$	$+1/2$
$X.11$	$X + 1$	$+1/4$

- Sum of errors=1/2, bias=1/8, smaller than truncation
- Same sum of errors obtained for d>2 - bias=$1/2 \cdot 2^{-d}$

Round to Nearest Even

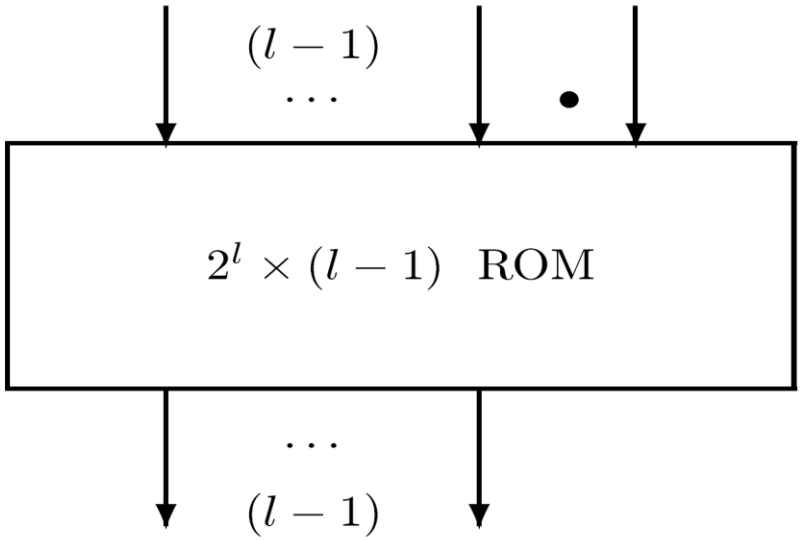


- In case of a tie (X.10), choose out of F1 and F2 the even one (with least-significant bit 0)
- Alternately rounding up and down - unbiased
- Round-to-Nearest-Odd - select the one with least-significant bit 1
- d=2 :
- Sum of errors=0
- Bias=0
- Mandatory in IEEE floating-point standard

Number	$Round(x)$	Error	Number	$Round(x)$	Error
$X0.00$	$X0.$	0	$X1.00$	$X1.$	0
$X0.01$	$X0.$	$-1/4$	$X1.01$	$X1.$	$-1/4$
$X0.10$	$X0.$	$-1/2$	$X1.10$	$X1.+1$	$+1/2$
$X0.11$	$X1.$	$+1/4$	$X1.11$	$X1.+1$	$+1/4$

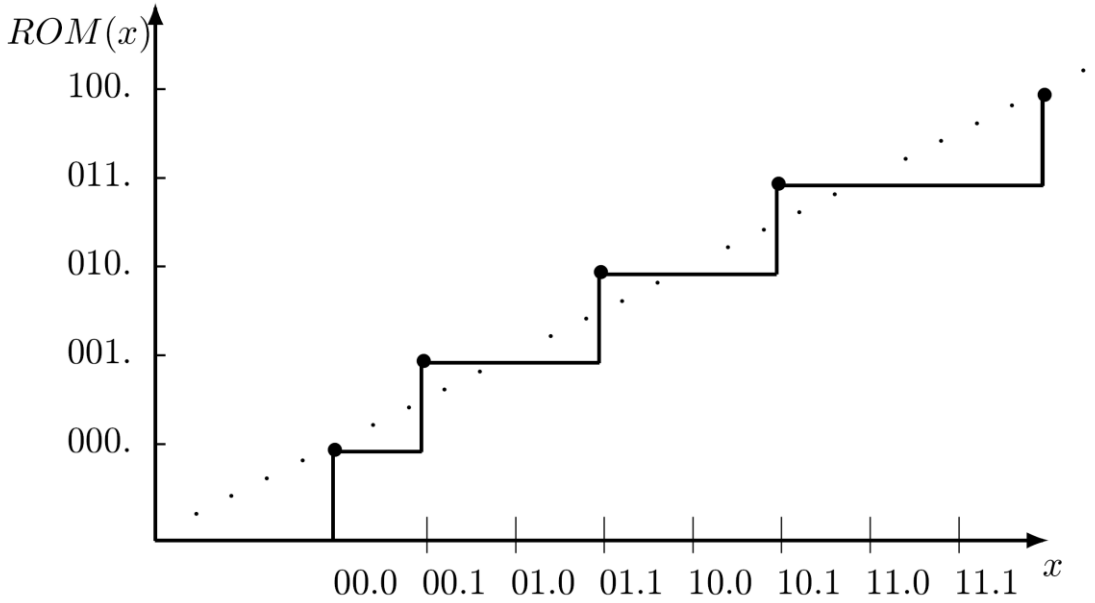
ROM Rounding

- Disadvantage of round-to-nearest schemes - require a complete add operation - carry propagation across entire significand
- Suggestion - use a ROM (read-only memory) with look-up table for rounded results
- Example - a ROM with
  l address lines - inputs are l-1 (out of m) least significant bits of significand and most significant bit out of d extra bits



ROM Rounding - Examples

- ROM has 2^l rows of $l-1$ bit each - correct rounding in most cases
- When all $l-1$ low-order bits of significand are 1's - ROM returns all 1's (truncating instead of rounding) avoiding full addition
- Example - $l=8$ - fast lookup - 255 out of 256 cases are properly rounded
- Example: $l=3$



Bias of ROM Rounding

- Example - $l=3$; $d=1$
- Sum of errors=1
- Bias=1/8

Number	$ROM(x)$	Error	Number	$ROM(x)$	Error
$X00.0$	$X00.$	0	$X10.0$	$X10.$	0
$X00.1$	$X01.$	$+1/2$	$X10.1$	$X11.$	$+1/2$
$X01.0$	$X01.$	0	$X11.0$	$X11.$	0
$X01.1$	$X10.$	$+1/2$	$X11.1$	$X11.$	$-1/2$

- In general - $bias=1/2[(1/2)^{d}-(1/2)^{l-1}]$
- When l is large enough - ROM rounding converges to round-to-nearest - bias converges to $1/2(1/2)^{d}$
- If the round-to-nearest-even modification is adopted - bias of modified ROM rounding converges to zero

Rounding and Interval Arithmetic

- Four rounding modes in IEEE standard
  - Round-to-nearest-even (default)
  - Round toward zero (truncate)
  - Round toward ∞
  - Round toward $-\infty$
- Last 2 - useful for Interval Arithmetic
  - Real number a represented by lower and upper bounds a_1 and a_2
  - Arithmetic operations operate on intervals
  - Calculated interval provides estimate on accuracy of computation

$$[a_1, a_2] + [b_1, b_2] = [a_1 + b_1, a_2 + b_2]$$

$$[a_1, a_2] - [b_1, b_2] = [a_1 - b_2, a_2 - b_1]$$

$$[a_1, a_2] \times [b_1, b_2] = [\min\{a_1b_1, a_1b_2, a_2b_1, a_2b_2\}, \max\{a_1b_1, a_1b_2, a_2b_1, a_2b_2\}]$$

  - Lower bound rounded toward $-\infty$, upper - toward ∞

Guard Digits for Multiply/Divide

- Multiplication has a double-length result - not all extra digits needed for proper rounding
- Similar situation - adding or subtracting two numbers with different exponents
- How many extra digits are needed for rounding and for postnormalization with leading zeros ?
- Division of signed-magnitude fractions - no extra digits - shift right operation may be required
- Multiplying two normalized fractions - at most one shift left needed if $\beta=2$ (k positions if $\beta=2^k$) $\Rightarrow$ one guard digit (radix β) is sufficient for postnormalization
- A second guard digit is needed for round-to-nearest - total of two - G (guard) and R (round)
- Exercise - Same for range [1,2) (IEEE standard)

Guard, Round and Sticky digits

- Round-to-nearest-even - indicator whether all additional digits generated in multiply are zero - detect a tie
- Indicator is a single bit - logical OR of all additional bits - sticky bit
- Three bits - G, R, S (sticky) - sufficient even for round-to-nearest-even
- Computing S when multiplying does not require generating all least significant bits of product
- Number of trailing zeros in product equals sum of numbers of zeros in multiplier and multiplicand
- Other techniques for computing sticky bit exist

Guard digits for Add/Subtract

- **Add/subtract** more complicated - especially when final operation (after examining sign bits) is subtract
- **Assumption** - normalized signed-magnitude fractions

F_1	0.	1	0	0	0	0	0	$\cdot 16^3$
F_2 aligned	0.	0	F	F	F	F	F	$\cdot 16^3$
$F_1 - F_2$	0.	0	0	0	0	0	1	$\cdot 16^3$
Postnormalization	0.	1	0	0	0	0	0	$\cdot 16^{-2}$

- **Subtract** - for postnormalization all shifted-out digits of subtrahend may need to participate in subtraction
  - Number of required guard digits = number in significand field - double size of significand adder/subtractor
- If subtrahend shifted more than 1 position to right (pre- alignment) - difference has at most 1 leading zero
- At most one shifted-out digit required for postnormalization

Subtract - Example 1

- Calculating A-B
- Significands of A and B are 12 bits long, base=2, $E_A - E_B = 2$ - requiring a 2-bit shift of subtrahend B in pre-alignment step

A	0.100000101100	00
B aligned	0.001100000001	10
$A - B$	0.010100101010	10
Postnormalization	0.101001010101	

- Same result obtained even if only one guard bit participates in subtraction generating necessary borrow

Subtract - Example 2

- Different if most significant shifted-out bit is 0
- Same two significands - $E_A - E_B = 6 \rightarrow$ B's significand shifted 6 positions

A	0.100000101100	000000
B aligned	0.000000110000	000110
$A - B$	0.011111111011	111010
Postnormalization	0.111111110111	

  - If only one guard bit - 4 least significant bits of result after postnormalization would be 1000 instead of 0111
  - Long sequence of borrows - seems that all additional digits in B needed to generate a borrow
- Possible conclusion : in the worst case - number of digits doubled
- Statement : Enough to distinguish between two cases:
  - (1) All additional bits (not including the guard bit) are 0
  - (2) at least one of the additional bits is 1

Proof of Statement

* All extra digits in A are zeros (not preshifted)
* Resulting three least significant bits in A-B (011 in example 2) are independent of exact position of 1's in extra digits of B
* We only need to know whether a 1 was shifted out or not - sticky bit can be used - if 1 is shifted into it during alignment it will be 1 - otherwise 0 - logical OR of all extra bits of B
* Sticky bit participates in subtraction and generates necessary borrow
* Using G and S -

		G	S
A	0.100000101100	0	0
B aligned	0.000000110000	0	1
$A-B$	0.011111111011	1	1
Postnormalization	0.111111110111		

* G and S sufficient for postnormalization
* In round-to-nearest - an additional accurate bit needed - sticky bit not enough - G,R,S required

Example 3 (E_A-E_B=6)

- Correct result

A	0.100000101100	000000
B aligned	0.000000110000	010110
$A-B$	0.011111111011	101010
Postnormalization	0.111111110111	0

- Using only G and S

		G	S
A	0.100000101100	0	0
B aligned	0.000000110000	0	1
$A-B$	0.011111111011	1	1
Postnormalization	0.111111110111	1	

- Round bit after postnormalization - 0, sticky bit cannot be used for rounding
- Using G, R, S

		G	R	S
A	0.100000101100	0	0	0
B aligned	0.000000110000	0	1	1
$A-B$	0.011111111011	1	0	1
Postnormalization	0.111111110111	0		

- Correct R=0 available for use in round-to-nearest
- For round-to-nearest-even: sticky bit needed to detect a tie available - serves two purposes

Example 4 - No Postnormalization

- Rounding requires a round bit and a sticky bit
- For round-to-nearest-even
  - original G can be an R bit
  - original R and S ORed to generate a new sticky bit S
- $E_A - E_B = 6$

A	0.100001010100			
B	0.110000010001			
		G	R	S
A	0.100001010100	0	0	0
B aligned	0.000000110000	0	1	1
$A - B$	0.100000100011	1	0	1
		R	S	
Before rounding	0.100000100011	1	1	
After round-to-nearest	0.100000100100			

Adding ulp in rounding

- If R=0 no rounding required - sticky bit indicates whether final result is exact/inexact (S=0/1)
- If R=1 operation in round-to-nearest-even depends on S and least-significant bit (L) of result
- If S=1 rounding must be performed by adding ulp
- If S=0 - tie case, only if L=1 rounding necessary
- Summary - round-to-nearest-even requires adding ulp to significand if $RS + R\bar{S}L = R(S + L) = 1$
- Adding ulp may be needed for directed roundings
- Example: in round toward $+\infty$, ulp must be added if result is positive and either R or S equals 1
- Similarly - in round toward $-\infty$ when result negative and R+S=1

IEEE Format Rounding Rules

LSB	R	S	Operation	$\overline{Error}$
0	0	0	$+0$	0
0	0	1	$+0$	$-0.25\ ulp$
0	1	0	$+0$	$-0.50\ ulp$
0	1	1	$+0.5\ ulp$	$+0.25\ ulp$
1	0	0	$+0$	0
1	0	1	$+0$	$-0.25\ ulp$
1	1	0	$+0.5\ ulp$	$+0.50\ ulp$
1	1	1	$+0.5\ ulp$	$+0.25\ ulp$
			Total	0

(a) Round-to-nearest-even scheme

R	S	Operation	$\overline{Error}$
0	0	$+0$	0
0	1	$+0$	$-0.25\ ulp$
1	0	$+0$	$-0.50\ ulp$
1	1	$+0$	$-0.75\ ulp$
		Total	$-0.375\ ulp$

(b) Round-to-zero scheme

Sign	R	S	Operation
$+$	0	0	$+0$
$+$	0	1	$+1\ ulp$
$+$	1	0	$+1\ ulp$
$+$	1	1	$+1\ ulp$
$-$	0	0	$+0$
$-$	0	1	$+0$
$-$	1	0	$+0$
$-$	1	1	$+0$

(c) Round-to-plus-infinity scheme

Sign	R	S	Operation
$-$	0	0	$+0$
$-$	0	1	$+1\ ulp$
$-$	1	0	$+1\ ulp$
$-$	1	1	$+1\ ulp$
$+$	0	0	$+0$
$+$	0	1	$+0$
$+$	1	0	$+0$
$+$	1	1	$+0$

(d) Round-to-minus-infinity scheme

Adding ulp in rounding

- Adding ulp after significands were added increases execution time of add/subtract
- Can be avoided - all three guard bits are known before significands added
- Adding 1 to L can be done at the same time that significands are added
- Exact position of L is not known yet, since a postnormalization may be required
- However, it has only two possible positions and two adders can be used in parallel
- Can also be achieved using one adder

Digital Computer Arithmetic

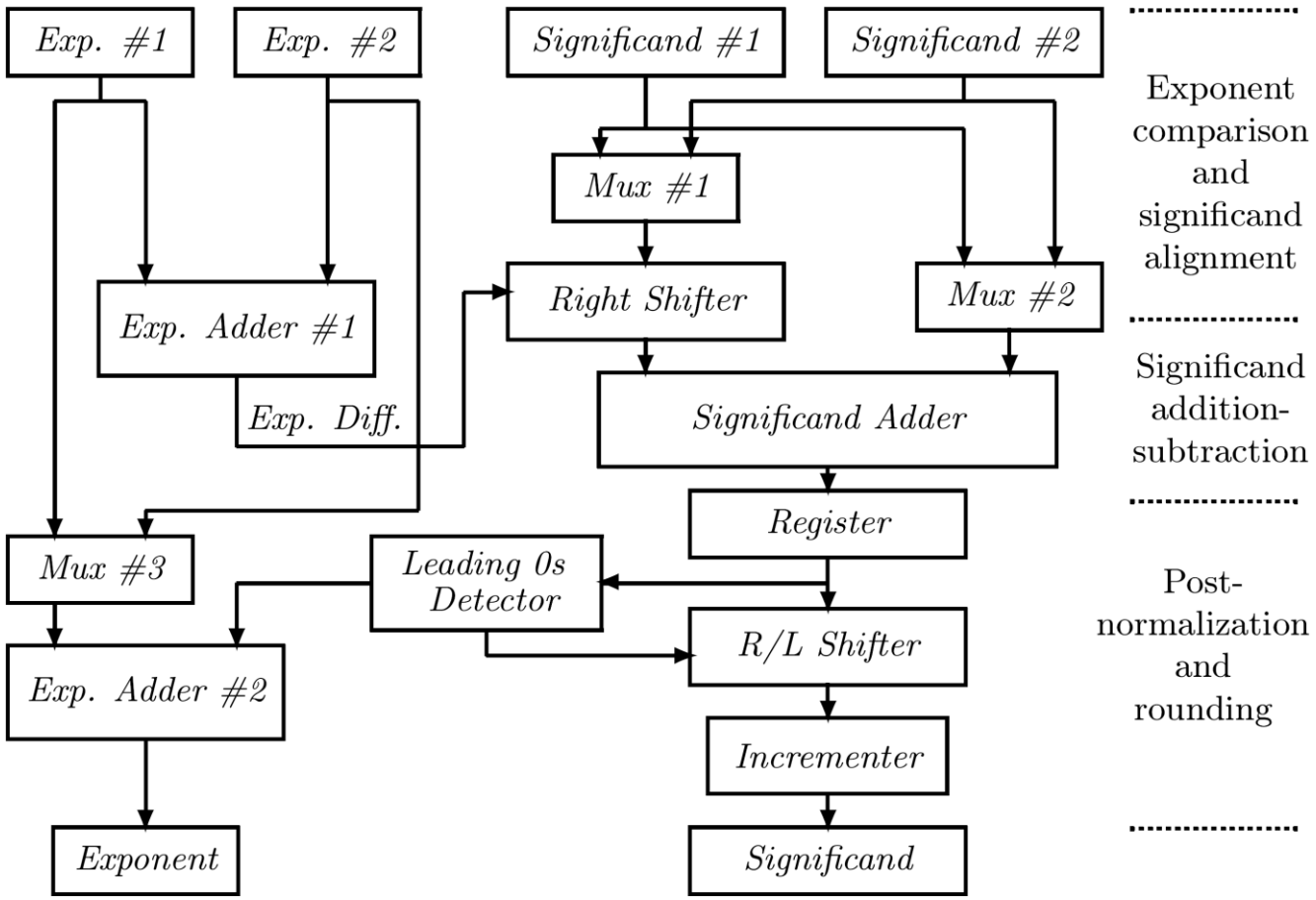
Part 4-C
Floating-Point Arithmetic - III

Soo-Ik Chae

Spring 2009

Floating-Point Adders

- Addition - large number of steps executed sequentially - some can be executed in parallel



Effective Addition/Subtraction

- Distinguish between effective addition and effective subtraction
  - Depends on sign bits of operands and instruction executed
- Effective addition:
  - (1) Calculate exponent difference to determine alignment shift
  - (2) Shift significand of smaller operand, add aligned significands
- The result can overflow by at most one bit position
  - Long postnormalization shift not needed
  - Single bit overflow can be detected and, if found, a 1-bit normalization is performed using a multiplexor

Eliminate Increment in Rounding

- Significand adder designed to produce two simultaneous results - sum and sum+1
  - Called compound adder; can be implemented in various ways (e.g., carry-look-ahead or conditional sum)
- Round-to-nearest-even - use rounding bits to determine which of the two should be selected
- These two are sufficient even if a single bit overflow occurs
  - In case of overflow, 1 is added in R position (instead of LSB position), and since R=1 if rounding needed, a carry will propagate to LSB to generate correct sum+1
- Directed roundings - R not necessarily 1 - sum+2 may be needed

Effective Subtraction

- Massive cancellation of most significant bits may occur - resulting in lengthy postnormalization
- Happens only when exponents of operands are close (difference ≤ 1) - pre-alignment can be eliminated
- Two separate procedures -
  - (1) exponents are close (difference ≤ 1) - only a postnormalization shift may be needed
  - (2) exponents are far (difference>1) - only a pre-alignment shift may be needed

Step	*CLOSE*	*FAR*
1	Predict exponent	Subtract exponents
2	Subtract significands Predict number of leading zeroes	Align significands
3	Postnormalization	Subtract significands
4	Select properly rounded result or negate result	Select properly rounded result

CLOSE Case

- Exponent difference predicted based on two least significant bits of operands - allows subtraction of significands to start as soon as possible
  - If 0 - subtract executed with no alignment
  - If ±1 - significand of smaller operand is shifted once to the right (using a multiplexor) and then subtracted from other significand
- In parallel - true exponent difference calculated
  - If > 1 - procedure aborted and FAR procedure followed
  - If ≤ 1 - CLOSE procedure continued
- In parallel with subtraction - number of leading zeros predicted to determine number of shift positions in postnormalization

CLOSE Case - Normalization and Rounding

- Next - normalization of significand and corresponding exponent adjustment
- Last - rounding - precomputing sum, sum+1 - selecting the one which is properly rounded - negation of result may be necessary
- Result of subtraction usually positive - negation not required
- Only when exponents equal - result of significand subtraction may be negative (in two's complement) - requiring a negation step
- No pre-alignment - no guard bits - no rounding (exact result)
- Negation and rounding steps - mutually exclusive

FAR Case

- First - exponent difference calculated
- Next - significand of smaller operand shifted to right for alignment
- Shifted-out bits used to set sticky bit
- Smaller significand subtracted from larger - result either normalized or requiring a single-bit-position left-shift (using a multiplexor)
- Last step - rounding

Leading Zeros Prediction Circuit

- Predict position of leading non-zero bit in result of subtract before subtraction is completed
- Allowing to execute postnormalization shift immediately following subtraction
- Examine bits of operands (of subtract) in a serial fashion, starting with most significant bits to determine position of first 1
- This serial operation can be accelerated using a parallel scheme similar to carry-look-ahead

Alternative Prediction of Leading 1

- Generate in parallel intermediate bits e_i - $e_i=1$ if
  - (1) $a_i = b_i$ and
  - (2) a_{i-1} and b_{i-1} allow propagation of expected carry (at least one is 1)
  - Subtract executed by forming one's complement of subtrahend and forcing carry into least significant position - carry expected
- $e_i = \overline{(a_i \oplus b_i)}\,(a_{i-1} + b_{i-1})$ -

  $e_i=1$ if carry allowed to propagate to position i
  - If forced carry propagates to position i - i-th bit of correct result will also be 1
  - If not - correct result will have a 1 in position $i-1$ instead
  - Position of leading 1 - either same as e_i or one to the right
- Count number of leading zeros in e_i - provide count to barrel shifter for postnormalization - at most one bit correction shift (left) needed

Exceptions in IEEE Standard

- Five types : overflow, underflow, division-by-zero, invalid operation, inexact result
- First three - found in almost all floating-point systems ; last two - peculiar to IEEE standard
- When an exception occurs - status flag set (remains set until cleared) - specified result generated
  - Example - a correctly signed ∞ for division-by-zero
- Separate trap-enable bit for each exception
- If bit is on when corresponding exception occurs - user trap handler is called
- Sufficient information must be provided by floating-point unit to trap handler to allow taking action
  - Example - exact identification of exception causing operation

Overflow - Trap Disabled

- **Overflow** exception flag set whenever exponent of result exceeds largest value allowed
- **Example** - single-precision - overflow occurs if E>254
- **Final result** determined by sign of intermediate (overflowed) result and rounding mode:
  - Round-to-nearest-even - ∞ with sign of intermediate result
  - Round toward 0 - largest representable number with sign of intermediate result
  - Round toward $-\infty$ - largest representable number with a plus sign if intermediate result positive; otherwise $-\infty$
  - Round toward ∞ - largest representable number with a minus sign if intermediate result negative; otherwise $+\infty$

Overflow - Trap Enabled

- Trap handler receives intermediate result divided by 2^{α} and rounded
- α = 192 / 1536 for single / double-precision format
- Chosen in order to translate the overflowed result as nearly as possible to middle of exponent range so that it can be used in subsequent operations with less risk of causing further exceptions

Example

- Multiplying 2^{127} (with E=254 in single-precision) by 2^{127} - overflowed product has E=254+254-127 =381 after being adjusted by 127
- Result overflows - E>254
- If product scaled (multiplied) by 2^{-192} - E=381-192=189 - "true" value of 189-127=62
- Smaller risk of causing further exceptions
- Relatively small operands can result in overflow
- Multiply 2^{64} (E=191 in single-precision) by 2^{65} (E=192)
- Overflowed product - E=191+192-127=256
- Exponent adjusted by 192 - E=256-192=64 - "true" value of 64-127=-63

Underflow - Trap Enabled

- Underflow exception flag is set whenever the result is a nonzero number between -2^{Emin} and 2^{Emin}
- $Emin=-126$ in single-precision format; 1022 in double-precision format
- Intermediate result delivered to underflow trap handler is the infinitely precise result multiplied by 2^{a} and rounded
- $a=192$ in single precision format; 1536 in double-precision format

Underflow - Trap Disabled

- Denormalized numbers allowed
- Underflow exception flag set only when an extraordinary loss of accuracy occurs while representing intermediate result (with a nonzero value between $\pm 2^{Emin}$) as a denormalized number
- Such a loss of accuracy occurs when either guard bit or sticky bit is nonzero- indicating an inexact result
- In an arithmetic unit where denormalized numbers are not implemented - delivered result is either zero or $\pm 2^{Emin}$

Underflow - Trap Disabled - Example

- Denormalized numbers implemented
- Multiply 2^{-65} by 2^{-65}
  - Result - $E=(127-65)+(127-65)-127=-3 < 1$
  - Cannot be represent as a normalized number
  - Result 2^{-130} represented as the denormalized number $0.0001\ 2^{-126}$ - $f=.0001$; $E=0$
- No underflow exception flag is set
- If second operand is $(1+ulp)\ 2^{-65}$
  - Correct product is $(1+ulp)\ 2^{-130}$
  - Converted to a denormalized number - $f=.0001$; $E=0$
  - Now sticky bit = 1
- Inexact result - underflow exception flag is set

Invalid Operation

- Flag is set if an operand is invalid for operation to be performed
- Result - when invalid operation trap is disabled - quiet NaN
- Examples of invalid operations :
  - Multiplying 0 by ∞
  - Dividing 0 by 0 or ∞ by ∞
  - Adding +∞ and -∞
  - Finding the square root of a negative operand
  - Calculating the remainder x REM y where y=0 or x=∞
  - Any operation on a signaling NaN

Division by Zero & Inexact Result

- Divide-by-zero exception flag is set whenever divisor is zero and dividend is a finite nonzero number
- When corresponding trap is disabled - result is a correctly signed ∞

- Inexact Result flag is set if rounded result is not exact or if it overflows without an overflow trap
- A rounded result is exact only when both guard bit and sticky bit are zero - no precision was lost when rounding
- Allows performing integer calculations in a floating-point unit

Accumulation of Round-off Errors

- Rounding in floating-point operations - even with best rounding schemes - results in errors that tend to accumulate as number of operations increases
  - ϵ - relative round-off error in a floating-point operation
  - $*$ - any floating-point arithmetic operation $+, -, \times, \div$
  - $Fl(x * y)$ - rounded or truncated result of $x * y$

$$\epsilon = \frac{Fl(x \star y) - (x \star y)}{(x \star y)}$$

$$Fl(x \star y) = (x \star y) \cdot (1 + \epsilon)$$

Upper Bounds of Relative Errors

- Truncation -
- Absolute error - maximum is least-significant digit of significand.
- Relative error - worst case when normalized result is smallest

$$|\epsilon_{trunc}| \leq \frac{2^{-m}}{1/2} = 2^{-m+1}.$$

- Round-to-nearest -
- Absolute error - maximum is half of ulp
- Relative error -

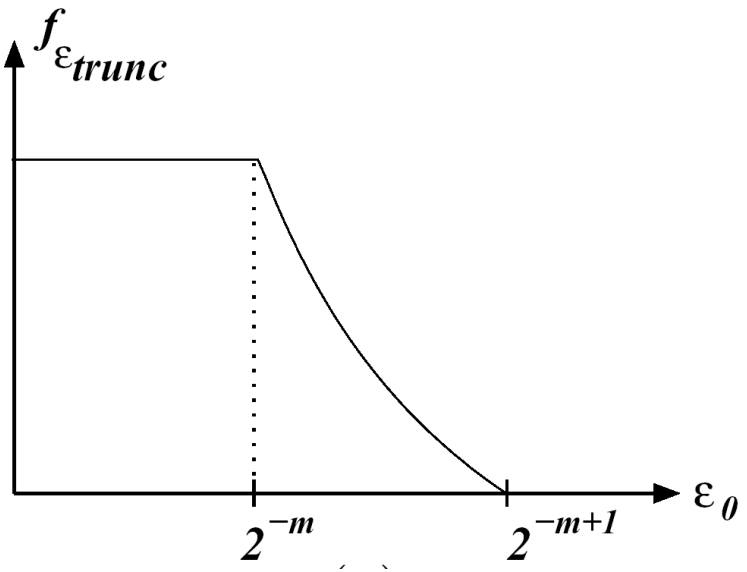
$$|\epsilon_{round}| \leq \frac{1}{2}2^{-m+1} = 2^{-m}$$

Distribution of Relative Truncation Error

- **Density function** of relative truncation error -

$$f_{\epsilon_{trunc}}(\epsilon_0) = \begin{cases} 2^{m-1}/\ln 2 & \text{if} \quad 0 \le \epsilon_0 < 2^{-m} \\ (\frac{1}{\epsilon_0} - 2^{m-1})/\ln 2 & \text{if} \quad 2^{-m} \le \epsilon_0 < 2^{-m+1} \end{cases}$$

- **Relative truncation errors** -
  - uniformly distributed in $[0,\ 2^{-m}]$
  - reciprocally in $[2^{-m},\ 2^{-m+1}]$



- **Average** relative error -

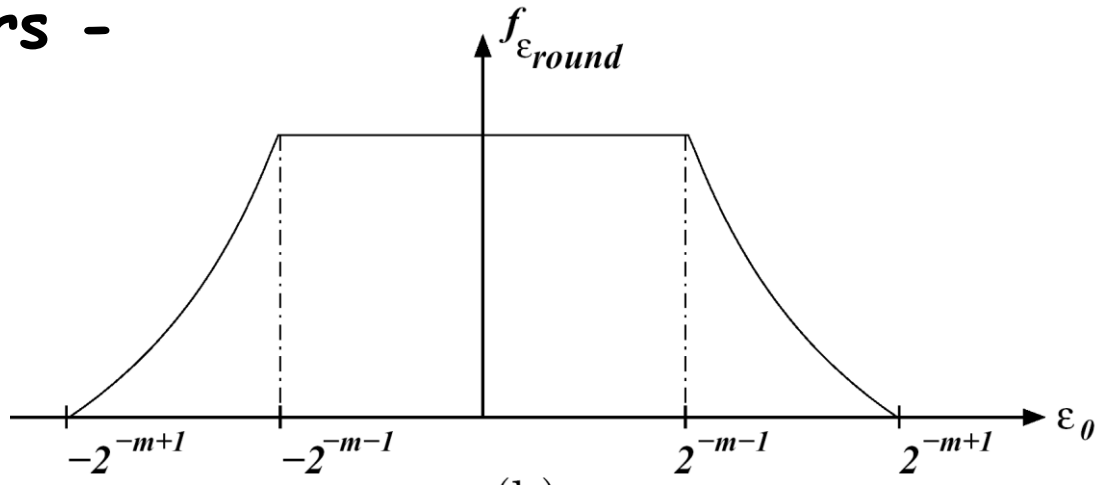
$$\overline{\epsilon_{trunc}} = 2^{-m-1}/\ln 2 \approx 0.72 \cdot 2^{-m}$$

Distribution of Relative Rounding Error

- **Density function** of relative rounding error -

$$f_{\epsilon_{round}}(\epsilon_0) = \begin{cases} 2^{m-1}/\ln 2 & \text{if} \quad -2^{-m-1} \le \epsilon_0 \le 2^{-m-1} \\ (\frac{1}{2|\epsilon_0|} - 2^{m-1})/\ln 2 & \text{if} \quad 2^{-m-1} < |\epsilon_0| < 2^{-m} \end{cases}$$

- Relative rounding errors -
  - uniformly distributed in $[-2^{-m-1}, 2^{-m-1}]$
  - reciprocally elsewhere
  - symmetric



- **Average relative error = 0**
- Analytical expressions are in very good agreement with empirical results

Accumulation of Errors - Addition

- Adding two intermediate results A_1, A_2
  - correct values - A_1^c, A_2^c
  - relative errors ϵ_1, ϵ_2

$$A_1 = A_1^c(1+\epsilon_1), \qquad A_2 = A_2^c(1+\epsilon_2)$$

- Assumption - no new error introduced in addition - relative error of sum

$$\frac{A_1^c \cdot \epsilon_1 + A_2^c \cdot \epsilon_2}{A_1^c + A_2^c} = \frac{A_1^c}{A_1^c + A_2^c} \cdot \epsilon_1 + \frac{A_2^c}{A_1^c + A_2^c} \cdot \epsilon_2$$

- Relative error of sum - weighted average of relative errors of operands
- If both operands positive - error in sum dominated by error in larger operand

Accumulation of Errors - Subtraction

- Subtracting two intermediate results A_1, A_2 - more severe error accumulation
- Relative error - $$\frac{A_1^c}{A_1^c - A_2^c} \cdot \epsilon_1 - \frac{A_2^c}{A_1^c - A_2^c} \cdot \epsilon_2$$
  - If A_1, A_2 are close positive numbers - accumulated relative error can increase significantly
  - If ϵ_1, ϵ_2 have opposite signs - inaccuracy can be large
- Accumulation of errors for a long sequence of floating-point operations depends on the specific application - difficult to analyze - can be simulated
- In most cases - accumulated relative error in truncation is higher than in round-to-nearest