Computer Organization and Architecture

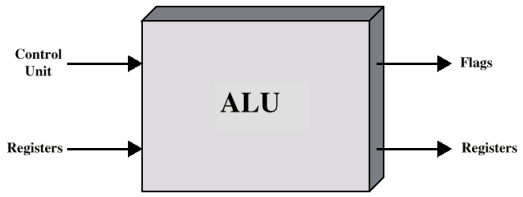
Chapter 9

Computer Arithmetic

Arithmetic & Logic Unit

- Performs arithmetic and logic operations on data – everything that we think of as "computing."
- Everything else in the computer is there to service this unit
- All ALUs handle integers
- Some may handle floating point (real) numbers
- May be separate FPU (math co-processor)
- FPU may be on separate chip (486DX +)

ALU Inputs and Outputs



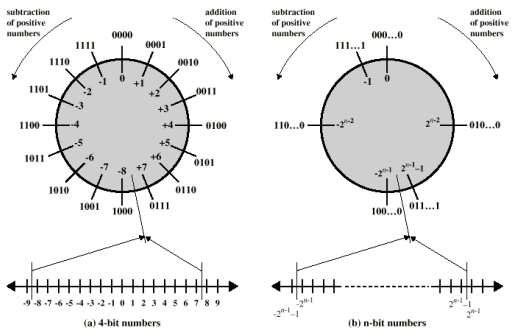
Integer Representation

- We have the smallest possible alphabet: the symbols 0 & 1 represent everything
- No minus sign
- No period
- Signed-Magnitude
- Two's complement

Benefits of 2's complement

- One representation of zero
- Arithmetic works easily (see later)
- Negating is fairly easy
  - 3 = 00000011
  - Boolean complement gives 11111100
  - Add 1 to LSB 11111101

Geometric Depiction of Twos Complement Integers



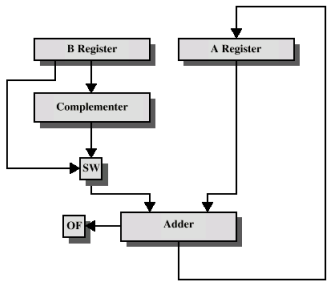
2's complement negation

- "Taking the 2's complement" (complement and add 1) is computing the arithmetic negation of a number
- Compute y = 0 – x
  - Or
- Compute y such that x + y = 0

Addition and Subtraction

- For addition use normal binary addition
  - 0+0=sum 0 carry 0
  - 0+1=sum 1 carry 0
  - 1+1=sum 0 carry 1
- Monitor MSB for overflow
  - Overflow cannot occur when adding 2 operands with the different signs
  - If 2 operand have same sign and result has a different sign, overflow has occurred
- Subtraction: Take 2's complement of subtrahend and add to minuend
  - i.e. a - b = a + (-b)
- So we only need addition and complement circuits

Hardware for Addition and Subtraction



OF = overflow bit
SW = Switch (select addition or subtraction)

Side note: Carry look-ahead

- Binary addition would seem to be dramatically slower for large registers
  - consider 0111 + 0011
  - carries propagate left-to-right
  - So 64-bit addition would be 8 times slower than 8-bit addition
- It is possible to build a circuit called a "carry look-ahead adder" that speeds up addition by eliminating the need to "ripple" carries through the word

Carry look-ahead

- Carry look-ahead is expensive
- If n is the number of bits in a ripple adder, the circuit complexity (number of gates) is O(n)
- For full carry look-ahead, the complexity is $O(n^3)$
- Complexity can be reduced by rippling smaller look-aheads: e.g., each 16 bit group is handled by four 4-bit adders and the 16-bit adders are rippled into a 64-bit adder

Multiplication

- A complex operation compared with addition and subtraction
- Many algorithms are used, esp. for large numbers
- Simple algorithm is the same long multiplication taught in grade school
  - Compute partial product for each digit
  - Add partial products

Multiplication Example

- 1011 Multiplicand (11 dec)
- x 1101 Multiplier (13 dec)
- 1011 Partial products
- 0000 Note: if multiplier bit is 1 copy
- 1011 multiplicand (place value)
- 1011 otherwise zero
- 10001111 Product (143 dec)
- Note: need double length result

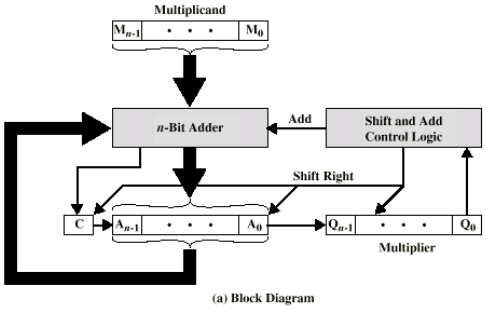
Simplifications for Binary Arithmetic

- Partial products are easy to compute:
  - If bit is 0, partial product is 0
  - If bit is 1, partial product is multiplicand
- Can add each partial product as it is generated, so no storage is needed
- Binary multiplication of unsigned integers reduces to "shift and add"

Control logic and registers

- 3 n bit registers, 1 bit carry register CF
- Register set up
  - Q register <- multiplier
  - M register <- multiplicand
  - A register <- 0
  - CF <- 0
- CF for carries after addition
- Product will be 2n bits in A Q registers

Unsigned Binary Multiplication

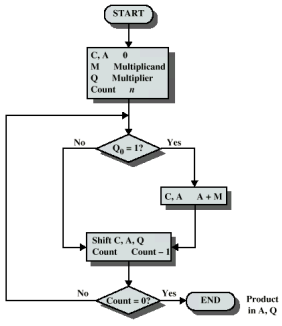


(a) Block Diagram

Multiplication Algorithm

- Repeat n times:
  - If Q_0 = 1 Add M into A, store carry in CF
  - Shift CF, A, Q right one bit so that:
    - A_{n-1} <- CF
    - Q_{n-1} <- A_0
    - Q_0 is lost
- Note that during execution Q contains bits from both product and multiplier

Flowchart for Unsigned Binary Multiplication



Execution of Example

C	A	Q	M		
0	0000	1101	1011	Initial Values	
0	1011	1101	1011	Add	First Cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second Cycle
0	1101	1111	1011	Add	Third Cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth Cycle
0	1000	1111	1011	Shift	

Two's complement multiplication

- Shift and add does not work for two's complement numbers
- Previous example as 4-bit 2's complement:
  -5 (1011) * -3 (1101) = -113 (10001111)
- What is the problem?
  - Partial products are 2n-bit products

```
    1011
  x1101
--------
00001011   1011 × 1 × 2^0
00000000   1011 × 0 × 2^1
00101100   1011 × 1 × 2^2
01011000   1011 × 1 × 2^3
--------
10001111
```

When the multiplicand is negative

- Each addition of the negative multiplicand must be negative number with 2n bits
- Sign extend multiplicand into partial product

```
    1001  (9)                 1001  (−7)
  x0011  (3)                x0011  (3)
--------                  --------
00001001  1001 × 2^0      11111001  (−7) × 2^0 = (−7)
00010010  1001 × 2^1      11110010  (−7) × 2^1 = (−14)
--------                  --------
00011011  (27)            11101011  (−21)
```

(a) Unsigned integers (b) Twos complement integers

- Or sign extend both operands to double precision
- Not efficient

When the multiplier is negative

- When the multiplier (Q register) is negative, the bits of the operand do not correspond to shifts and adds needed
- 1101 <->1*2^3 + 1*2^2 + 1*2^0
  = -(2^3 + 2^2 + 2^0)
- But we need
  -(2^1 + 2^0)

The obvious solution

- Convert multiplier and multiplicand to unsigned integers
- Multiply
- If original signs differed, negate result
- But there are more efficient ways

Fast multiplication

- Consider the product 6234 * 99990
  - We could do 4 single-digit multiplies and add partial sums
- Or we can express the product as
  $6234 * (10^6 - 10^1)$
- In binary x * 00111100 can be expressed as
  $x * (2^5 + 2^4 + 2^3 + 2^2) = x * 60$
- We can reduce the number of operations to 2 by observing that 00111100 = 01000000 – 00000010 (64-4 = 60)
  - $x * 00111100 = x * 2^6 - x * 2^2$
  - Each block of 1's can be reduced to two operations
  - In the worst case 01010101 we still have only 8 operations

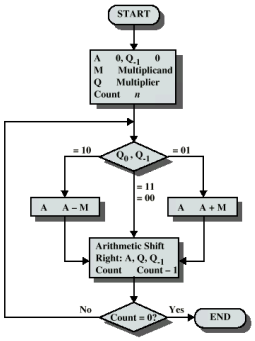
Booth's Algorithm Registers and Setup

- 3 n bit registers, 1 bit register logically to the right of Q (denoted as Q_{-1})
- Register set up
  - Q register <- multiplier
  - Q_{-1} <- 0
  - M register <- multiplicand
  - A register <- 0
  - Count <- n
- Product will be 2n bits in A Q registers

Booth's Algorithm Control Logic

- Bits of the multiplier are scanned one at a a time (the current bit Q_0)
- As bit is examined the bit to the right is considered also (the previous bit Q_{-1})
- Then:
  - 00: Middle of a string of 0s, so no arithmetic operation.
  - 01: End of a string of 1s, so add the multiplicand to the left half of the product (A).
  - 10: Beginning of a string of 1s, so subtract the multiplicand from the left half of the product (A).
  - 11: Middle of a string of 1s, so no arithmetic operation.
- Then shift A, Q, bit Q_{-1} right one bit using an *arithmetic* shift
- In an arithmetic shift, the msb remains unchanged

Booth's Algorithm



Example of Booth's Algorithm (7*3 = 21)

A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial Values	
1001	0011	0	0111	A A - M	First Cycle
1100	1001	1	0111	Shift	First Cycle
1110	0100	1	0111	Shift	Second Cycle
0101	0100	1	0111	A A + M	Third Cycle
0010	1010	0	0111	Shift	Third Cycle
0001	0101	0	0111	Shift	Fourth Cycle

Example: -3 * 2 = -6 (-3 = 1101)

A	Q	Q_{-1}	M	C/P	Comment
0000	1101	0	0010		Initial Values
1110	1101	0	0010	10	A <- A - 2 = -2
1111	0110	1	0010		>>1
0001	0110	1	0010	01	A <- A + 2
0000	1011	0	0010		>>1
1110	1011	0	0010	01	A <- A - 2 = -2
1111	0101	1	0010		>>1
1111	1010	1	0010	11	>>1 A:Q = -6

Example: 6 * -1 = -6 (1111 = -1)

A	Q	Q_{-1}	M	C/P	Comment
0000	1111	0	0110		Initial Values
1010	1111	1	0110	10	A <- A - 6 = -6
1101	0111	1	0110		>>1
1110	1011	1	0110	11	>>1
1111	0101	1	0110	11	>>1
1111	1010	1	0110	11	>>1 A:Q = -6

Example: 3 * -2 = -6 (1110 = -2)

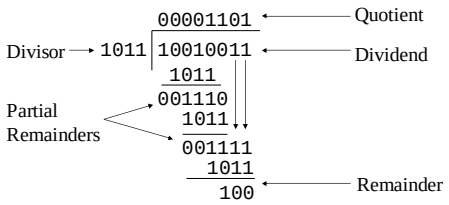
A	Q	Q_{-1}	M	C/P	Comment
0000	0011	0	1110		Initial Values
0010	0011	0	1110	10	A <- A -(-2) = 2
0001	0001	1	1110		>> 1
0000	1000	1	1110	11	>> 1
1110	1000	1	1110	01	A <- A +(-2) = -2
1111	0100	0	1110		>>1
1111	1010	0	1110	00	>> 1 A:Q = -6

Division

- More complex than multiplication to implement (for computers as well as humans!)
  - Some processors designed for embedded applications or digital signal processing lack a divide instruction
- Basically inverse of add and shift: shift and subtract
- Similar to long division taught in grade school

Unsigned Division In Principle

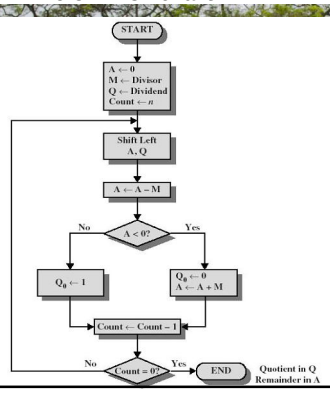
147 / 11 = 13 with remainder 4



Unsigned Division algorithm

- Using same registers (A,M,Q, count) as multiplication
- Results of division are quotient and remainder
  - Q will hold the quotient
  - A will hold the remainder
- Initial values
  - Q <- 0
  - A <- Dividend
  - M <- Divisor
  - Count <- n

Unsigned Division Flowchart



Example

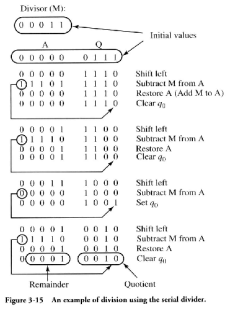


Figure 3-15 An example of division using the serial divider.

Two's complement division

- More difficult than unsigned division
- Algorithm:
  1. M <- Divisor, A:Q <- dividend sign extended to 2n bits; for example 0111 -> 00000111 ; 1001-> 11111001
     (note that 0111 = 7 and 1001 = -3)
  2. Shift A:Q left 1 bit
  3. If M and A have same signs, perform A <- A-M otherwise perform A <- A + M
  4. The preceding operation succeeds if the sign of A is unchanged
     - If successful, or (A==0 and Q==0) set Q_0 <- 1
     - If not successful, and (A!=0 or Q!=0) set Q_0 <- 0 and restore the previous value of A
  5. Repeat steps 2,3,4 for n bit positions in Q
  6. Remainder is in A. If the signs of the divisor and dividend were the same then the quotient is in Q, otherwise the correct quotient is 0-Q

2's complement division examples

A	Q	M = 0011
0000	0111	Initial value
0000	1110	shift
1101		subtract
0000	1110	restore
0001	1100	shift
1110		subtract
0001	1100	restore
0011	1000	shift
0000		subtract
0000	1001	set $Q_0 = 1$
0001	0010	shift
1110		subtract
0001	0010	restore

(a) (7)/(3)

A	Q	M = 1101
0000	0111	Initial value
0000	1110	shift
1101		add
0000	1110	restore
0001	1100	shift
1110		add
0001	1100	restore
0011	1000	shift
0000		add
0000	1001	set $Q_0 = 1$
0001	0010	shift
1110		add
0001	0010	restore

(b) (7)/(–3)

2's complement division examples

A	Q	M = 0011
1111	1001	Initial value
1111	0010	shift
0010		add
1111	0010	restore
1110	0100	shift
0001		add
1110	0100	restore
1100	1000	shift
1111		add
1111	1001	set $Q_0 = 1$
1111	0010	shift
0010		add
1111	0010	restore

(c) (–7)/(3)

A	Q	M = 1101
1111	1001	Initial value
1111	0010	shift
0010		subtract
1111	0010	restore
1110	0100	shift
0001		subtract
1110	0100	restore
1100	1000	shift
1111		subtract
1111	1001	set $Q_0 = 1$
1111	0010	shift
0010		subtract
1111	0010	restore

(d) (–7)/(–3)

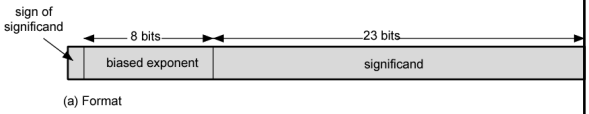
2's complement remainders

- 7 / 3 = 2 R 1
- 7 / -3 = -2 R 1
- -7 / 3 = -2 R -1
- -7 / -3 = 2 R -1
- Here the remainder is defined as:
  Dividend = Quotient * Divisor + Remainder

IEEE-754 Floating Point Numbers

- Format was discussed earlier in class
- Before IEEE-754 each family of computers had proprietary format: Cray,Vax, IBM
- Some Cray and IBM machines still use these formats
- Most are similar to IEEE formats but vary in details (bits in exponent or mantissa):
  - IBM Base 16 exponent
  - Vax, Cray: bias differs from IEEE
- Cannot make precise translations from one format to another
- Older binary scientific data not easily accessible

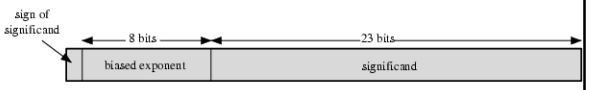
IEEE 754



(a) Format

- +/- 1.significand x $2^{exponent}$
- Standard for floating point storage
- 32 and 64 bit standards
- 8 and 11 bit exponent respectively
- Extended formats (both mantissa and exponent) for intermediate results

Floating Point Examples



(a) Format

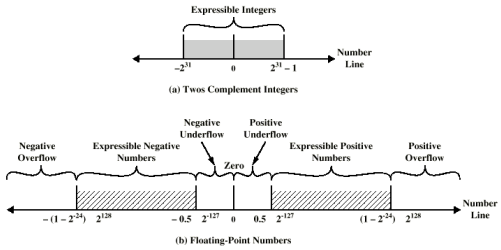
$1.1010001 \times 2^{10100} = 0\ 10010011\ 10100010000000000000000 = 1.638125 \times 2^{20}$
$-1.1010001 \times 2^{10100} = 1\ 10010011\ 10100010000000000000000 = -1.638125 \times 2^{20}$
$1.1010001 \times 2^{-10100} = 0\ 01101011\ 10100010000000000000000 = 1.638125 \times 2^{-20}$
$-1.1010001 \times 2^{-10100} = 1\ 01101011\ 10100010000000000000000 = -1.638125 \times 2^{-20}$

(b) Examples

FP Ranges

- For a 32 bit number
  - 8 bit exponent
  - +/- $2^{256} \approx 1.5 \times 10^{77}$
- Accuracy
  - The effect of changing lsb of mantissa
  - 23 bit mantissa $2^{-23} \approx 1.2 \times 10^{-7}$
  - About 6 decimal places

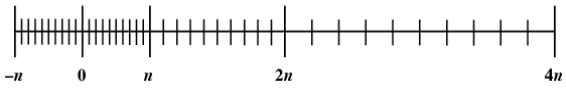
Expressible Numbers



(a) Twos Complement Integers

(b) Floating-Point Numbers

Density of Floating Point Numbers



- Note that there is a tradeoff between density and precision

  For a floating point representation of n bits, if we increase the precision by using more bits in the mantissa then then we decrease the range

  If we increase the range by using more bits for the exponent then we decrease the density and precision

Floating Point Arithmetic Operations

Floating Point Numbers	Arithmetic Operations
$X = X_S \times B^{X_E}$ $Y = Y_S \times B^{Y_E}$	$X + Y = \left(X_S \times B^{X_E - Y_E} + Y_S\right) \times B^{Y_E}$ $X - Y = \left(X_S \times B^{X_E - Y_E} - Y_S\right) \times B^{Y_E}$ $X_E \le Y_E$
	$X \times Y = \left(X_S \times Y_S\right) \times B^{X_E + Y_E}$
	$\frac{X}{Y} = \left(\frac{X_S}{Y_S}\right) \times B^{X_E - Y_E}$

Examples:

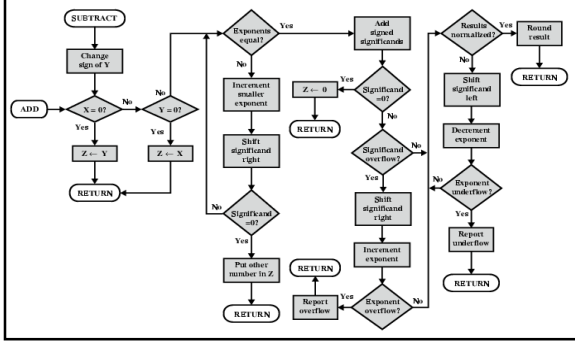
$X = 0.3 \times 10^2 = 30$
$Y = 0.2 \times 10^3 = 200$

$X + Y = (0.3 \times 10^{2-3} + 0.2) \times 10^3 = 0.23 \times 10^3 = 230$
$X - Y = (0.3 \times 10^{2-3} - 0.2) \times 10^3 = (-0.17) \times 10^3 = -170$
$X \times Y = (0.3 \times 0.2) \times 10^{2+3} = 0.06 \times 10^5 = 6000$
$X + Y = (0.3 + 0.2) \times 10^{2-3} = 1.5 \times 10^{-1} = 0.15$

FP Arithmetic + /-

- Addition and subtraction are more complex than multiplication and division
- Need to align mantissas
- Algorithm:
  - Check for zeros
  - Align significands (adjusting exponents)
  - Add or subtract significands
  - Normalize result

FP Addition & Subtraction Flowchart for Z <- X + Y and Z <- X - Y



Zero check

- Addition and subtraction identical except for sign change
- For subtraction, just negate subtrahend (Y in Z = X-Y) then compute Z = X+Y
- If either operand is 0 report the other as the result

Significand Alignment

- Manipulate numbers so that both exponents are equal
- Shift number with smaller exponent to the right – if bits are lost they will be less significant
  - Repeat
    - Shift mantissa right 1 bit
    - Add 1 to exponent
  - Until exponents are equal
- If mantissa becomes 0 report other number as result

Addition

- Add mantissas together, taking sign into account
- May result in 0 if signs differ
- Can result in mantissa overflow by 1 bit (carry)
  - Shift mantissa right and increment exponent
  - Report error if exponent overflow

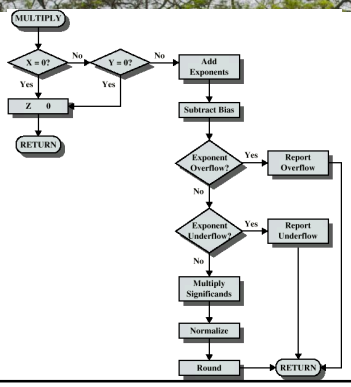
Normalization

- While (MSB of mantissa == 0)
  - Shift mantissa left one bit
  - Decrement exponent
  - Check for exponent underflow
- Round mantissa

FP Arithmetic Multiplication and Division

- Simpler processes than addition and subtraction
  - Check for zero
  - Add/subtract exponents
  - Multiply/divide significands (watch sign)
  - Normalize
  - Round

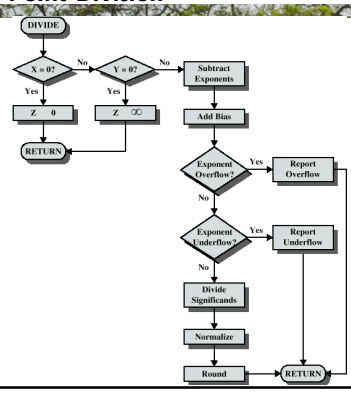
Floating Point Multiplication



Multiplication

- If either operand is 0 report 0
- Add exponents
  - Because addition doubles bias, first subtract the bias from one exponent
- If exponent underflow or overflow, report error
  - Underflow may be reported as 0 and overflow as infinity
- Multiply mantissas as if they were integers (similar to 2's comp mult.)
  - Note product is twice as long as factors
- Normalize and round
  - Same process as addition
  - Could result in exponent underflow

Floating Point Division



Division

- If divisor is 0 report error or infinity; dividend 0 then result is 0
- Subtract divisor exponent from dividend exp.
  - Removes bias so add bias back
- If exponent underflow or overflow, report error
  - Underflow may be reported as 0 and overflow as infinity
- Divide mantissas as if they were integers (similar to 2's comp mult.)
  - Note product is twice as long as factors
- Normalize and round
  - Same process as addition
  - Could result in exponent underflow

IEEE Standard for Binary Arithmetic

- Specifies practices and procedures beyond format specification
  - Guard bits (intermediate formats)
  - Rounding
  - Treatment of infinities
  - Quiet and signaling NaNs
  - Denormalized numbers

Precision considerations

- Floating point arithmetic is inherently inexact except where only numbers composed of sums of powers of 2 are used
- To preserve maximum precision there are two main techniques:
  - Guard bits
  - Rounding rules

Guard bits

- Length of FPU registers > bits in mantissa
- Allows some preservation of precision when
  - aligning exponents for addition
  - Multiplying or dividing significands
- We have seen that results of arithmetic can vary when intermediate stores to memory are made in the course of a computation

Rounding

- Conventional banker's rounding (round up if 0.5) has a slight bias toward the larger number
- To remove this bias use round-to-even:
  - 1.5 -> 2
  - 2.5 -> 2
  - 3.5 -> 4
  - 4.5 -> 4
  - Etc

IEEE Rounding

- Four types are defined:
  - Round to nearest (round to even)
  - Round to + infinity
  - Round to – infinity
  - Round to 0

Round to nearest

- If extra bits beyond mantissa are 100..1.. then round up
- If extra bits are 01… then truncate
- Special case: 10000…0
  - Round up if last representable bit is 1
  - Truncate if last representable bit is 0

Round to + /- infinity

- Useful for interval arithmetic
  - Result of fp computation is expressed as an interval with upper and lower endpoints
  - Width of interval gives measure of precision
  - In numerical analysis algorithms are designed to minimize width of interval

Round to 0

- Simple truncation, obvious bias
- May be needed when explicitly rounding following operations with transcendental functions

Infinities

- Infinity treated as limiting case for real arithmetic
- Most arithmetic operations involving infinities produce infinity

Quiet and Signaling NaNs

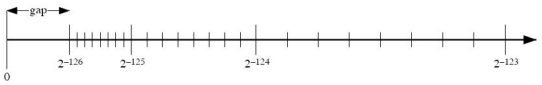
- NaN = Not a Number
- Signaling NaN causes invalid operation exception if used as operand
- Quiet NaN can propagate through arithmetic operations without raising an exception
- Signaling NaNs are useful for initial values of uninitialized variables
- Actual representation is implementation (processor) specific

Quiet NaNs

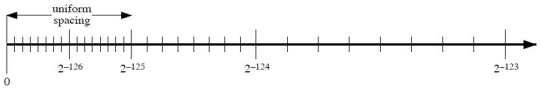
Operation	Quiet NaN Produced by
Any	Any operation on a signaling NaN
Add or subtract	Magnitude subtraction of infinities: $(+\infty) + (-\infty)$ $(-\infty) + (+\infty)$ $(+\infty) - (+\infty)$ $(-\infty) - (-\infty)$
Multiply	$0 \times \infty$
Division	$\frac{0}{0}$ or $\frac{\infty}{\infty}$
Remainder	x REM 0 or ∞ REM y
Square root	$\sqrt{x}$ where $x < 0$

Denormalized Numbers

- Handle exponent underflow
- Provide values in the "hole around 0"



(a) 32-bit format without denormalized numbers



(b) 32-bit format with denormalized numbers

Unnormalized Numbers

- Denormalized numbers have fewer bits of precision than normal numbers
- When an operation is performed with a denormalized number and a normal number, the result is called an "unnormal" number
- Precision is unknown
- FPU can be programmed to raise an exception for unnormal computations